UNIVERSITE JOSEPH FOURIER-GRENOBLE I

THESE

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER – GRENOBLE I
Discipline : Informatique

Présentée et soutenue publiquement le 4 Juillet 2006 par

Ouafa Hachani

TITRE

Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation

Directeurs de thèse : Jean-Pierre Giraudin et Daniel Bardou

Composition du jury

Président	:	Yves Chiaramella
Rapporteurs	:	Marianne Huchard
	:	Bernard Coulette
Examinateur	:	Michel Léonard

Thèse préparée au sein de l'équipe SIGMA du laboratoire LSR-IMAG
(Logiciels, Systèmes et Réseaux)
Université Joseph Fourier – Grenoble I

À mes parents qui m'ont toujours soutenue
À mes frères et ma sœur à qui je souhaite une excellente réussite

Remerciements

C'est un agréable devoir de s'acquitter des dettes de reconnaissance cumulées tout au long de ce travail de recherche. Je voudrais adresser mes sincères remerciements à tous ceux qui m'ont si efficacement aidée à préparer le mieux possible cette thèse.

Je remercie vivement mon directeur de thèse, M. *Jean-Pierre Giraudin*, et mon co-encadreur, M. *Daniel Bardou*, pour avoir dirigé et encadré mes travaux de recherches, ainsi que pour leurs conseils certainement très utiles pour mon avenir. Je suis aussi très reconnaissante envers eux pour leur soutien et leurs encouragements permanents.

Je tiens à remercier également M. *Yves Chiaramella* qui me fait l'honneur de présider le jury, ainsi que Mme *Marianne Huchard* et M. *Bernard Coulette*, qui ont accepté d'évaluer ce travail sur lequel ils ont apporté leurs critiques grandement appréciées. Je remercie aussi M. *Michel Léonard* qui a examiné mon travail.

Je remercie l'ensemble du personnel technique et administratif du laboratoire LSR-IMAG pour leur aide tout au long de cette thèse. Je tiens à remercier également tous les membres de l'équipe SIGMA pour leur bonne humeur, leur disponibilité et leur efficacité. Je remercie tout particulièrement mes collègues de bureau (Laurent, Ibtissem, Nicolas, etc.) pour leur patience et surtout pour leur amitié.

Je ne saurais oublier tous mes amis (Leila, Akram, etc.) qui m'ont aidée de près ou de loin à mener cette thèse à son terme, et à supporter la vie loin de ma famille. Enfin, je ne pourrais terminer sans remercier mes parents pour leur éternel soutien, ainsi que mes frères et ma soeur pour leurs encouragements.

Table des matières

Liste des figures

Liste des tableaux

Introduction Générale

Contexte et problématique

Une bonne modularisation, lisibilité et compréhension des logiciels, la possibilité de réutiliser ceux-ci, de les faire évoluer facilement et de manière fiable, sont des objectifs très recherchés en génie logiciel. Pour approcher de tels objectifs et favoriser une ingénierie des systèmes d'information de bonne qualité, de nombreux modèles (Objets, composants, agents, etc.) et méthodes de développement (Merise, OMT, UML, etc.) ont été adoptés ces dernières années. Les approches Objet et Composant, ainsi que UML sont certainement les plus largement utilisées. Cependant, si ces modèles et méthodes ont des avantages certains, ils ont aussi leur lot d'inconvénients et souffrent d'un certain nombre de lacunes qui les rendent, aujourd'hui, insuffisants pour prendre en compte la complexité des nouvelles applications. Ainsi, devant la complexité croissante des systèmes d'information et leur incessante évolution rendant leur développement plus difficile, plus coûteux et moins fiable, l'ingénierie des systèmes d'information se tourne actuellement, de plus en plus, vers de nouvelles problématiques d'étude : notamment, l'ingénierie de nouvelles techniques et méthodes de développement facilitant l'évolution et favorisant la réutilisation des systèmes produits. Cette pratique se traduit aujourd'hui par l'existence d'approches de développement basées sur la notion d'Aspect, l'ingénierie dirigée par les modèles (IDM ou MDA pour *Model Driven Architecture*), etc. Diverses approches favorisant la réutilisation s'imposent ainsi de plus en plus dans tout le processus de développement, et en particulier lors des phases d'analyse et de conception.

Le besoin d'améliorer et d'imposer la réutilisation au niveau de la phase d'implémentation est à présent relativement satisfait, par exemple, par les concepts fondamentaux Objet et Composant. Les langages de programmation par objets mettent en effet à la disposition des développeurs divers mécanismes et concepts permettant de se rapprocher de plus en plus des objectifs attendus, notamment au travers d'une meilleure organisation et structuration des programmes. Ces mécanismes et concepts de l'approche Objet de base ou étendus pour les composants (l'encapsulation des données et des opérations applicables sur ces données, les notions de classes et d'objets, les mécanismes d'instanciation et d'héritage, le polymorphisme et la liaison dynamique, etc.) présentent des avantages reconnus quant à la modularisation, la lisibilité, l'évolution et la réutilisation des différentes classes fonctionnelles représentant les objets du monde réel. Cependant, il faut pouvoir programmer des fonctionnalités inter-classes (i.e. des interactions complexes entre divers objets) ainsi que des besoins non fonctionnels (tels que la sécurité, la persistance des données, etc.), des propriétés dites transversales à plusieurs classes du système. L'approche Objet avec son faible niveau de granularité ne fournit pas (ou peu[1]) de support direct permettant une représentation explicite et localisée de telles préoccupations transversales, et dont le code se retrouve généralement décliné et éclaté dans la description des différentes classes de base constituant le système. Les composants offrent une solution partielle assemblant des objets fonctionnels et des objets techniques, mais ils ne facilitent pas l'expression des fonctionnalités transversales. L'évolution et la réutilisation de telles préoccupations, mais aussi des classes du système qui mêlent le code relatif à ces dernières avec le code de leurs préoccupations de base, se trouvent ainsi compromises.

[1] L'approche Objet fournit un nombre limité de formes d'interconnexion de primitives (invocation de méthodes) rendant difficile la prise en compte d'interactions complexes et augmentant le couplage des objets. Elle propose peu de solutions pour faciliter l'adaptation et l'assemblage d'objets.

Au niveau des phases en amont du cycle de développement, c'est le but des approches à base de patrons, par exemple, que d'assurer la réutilisation en proposant des connaissances caractérisées, approuvées et réutilisables dès la phase de recensement de besoins. Dans le cadre de cette thèse, nous nous intéressons plus particulièrement aux patrons de conception de type produit. Ces patrons permettent de réduire la complexité d'un grand nombre de problèmes spécifiques à la conception. Ils présentent des solutions génériques permettant de résoudre certains problèmes de structuration et d'organisation des programmes. Cependant, s'ils sont reconnus très utiles et si aujourd'hui de nombreux patrons de conception ont été identifiés et proposés, ils n'ont été conçus qu'en termes de classes. Ainsi, en définissant en général des préoccupations transversales à plusieurs classes ou objets coopératifs, la plupart des patrons de conception trouvent leurs implantations dispersées et enchevêtrées dans la description des différentes classes du système. Ceci pose plusieurs problèmes d'utilisation des patrons, rend difficile de retracer l'application d'un ou de plusieurs patrons dans un programme, et diminue par conséquent leur évolution et leur réutilisation. Les imitations des patrons peuvent aussi nuire à la lisibilité et à la compréhension du code du système utilisant ces patrons, et donc diminuer l'évolution et la réutilisation de celui-ci.

Les modèles et langages introduits dans le cadre de l'approche Aspect récemment proposée, offrent de nouveaux concepts et mécanismes permettant de structurer les programmes en appliquant au mieux le principe de séparation des préoccupations (*Separation of Concerns*). Par exemple, la programmation par aspects (*Aspect-Oriented Programming*) et ses langages (tel que AspectJ) prônent une décomposition des programmes non seulement en classes mais aussi en aspects. Un aspect est une unité de décomposition des applications encapsulant totalement et exclusivement une préoccupation transversale au découpage de l'application en classes (encapsulant les préoccupations dites de base). Les concepts et mécanismes de cette nouvelle technologie (aspect, introduction, points de jonction, tissage, etc.) assurent ainsi une meilleure structure et organisation des programmes, comparativement à ce qui est produit avec l'approche Objet. D'une manière générale, les modèles et langages Aspect consistent à réaliser une application en deux temps. Tout d'abord, les différentes préoccupations de base et transversales sont définies et isolées (les unes des autres) par diverses unités de décomposition. Le système attendu est produit par la suite, par la composition de ces préoccupations selon certaines règles de composition. En proposant de retarder l'intégration des différentes unités de décomposition, l'approche Aspect relâche donc le couplage entre les différentes préoccupations de base et transversales, offrant ainsi de nouvelles perspectives quant à la lisibilité, la compréhension, la traçabilité, l'évolution et la réutilisation de ces préoccupations. En ce sens, cette approche peut permettre de nouvelles réalisations de patrons de conception par objets en palliant leurs problèmes d'utilisation et en améliorant leur traçabilité, et par là même leur évolution et leur réutilisation. Toutefois, un certain manque de consensus sur les définitions des concepts et mécanismes fondamentaux de l'approche Aspect, qui sont très rarement considérés au niveau de la phase de conception, rend difficile la définition de nouvelles structures de patrons génériques et communes à diverses techniques spécifiques de programmation par aspects, pour ne proposer que de nouvelles solutions d'implémentation étroitement liées à un langage de programmation particulier.

Objectifs de la thèse

Sur la base des constats évoqués ci-dessus, nous nous intéressons plus particulièrement dans le cadre de cette thèse à l'utilisation conjointe des patrons de conception produit et de l'approche Aspect. Notre premier objectif porte, d'une part, sur l'identification des apports de l'utilisation des mécanismes et concepts Aspect (développés en programmation) dans l'implémentation des patrons de conception par objets déjà existants, et d'autre part, sur l'évolution de ces derniers vers des patrons à base d'aspects. Deux raisons pour une telle étude peuvent au moins être évoquées, nous les explicitons dans ce qui suit.

- Comme nous l'avons introduit précédemment, l'approche Aspect permet d'améliorer l'implémentation des patrons de conception par objets et de garantir leur traçabilité. Elle fournit un support direct pour une représentation explicite de ceux-ci, contribuant à garder visible et isolée dans une unité modulaire l'imitation de chaque patron dans le code des applications. L'évolution et la réutilisation du code relatif aux imitations des patrons, mais aussi de celui relatif aux applications, sont ainsi facilitées.
- Si la plupart des développeurs sont de plus en plus familiarisés avec les patrons de conception par objets, les nouvelles solutions par aspects de ces patrons leur permettent de réutiliser et d'appliquer au mieux les savoirs capitalisés par ces derniers dans le cadre d'un développement par aspects. De plus, ces nouvelles solutions peuvent être utilisées pour faire évoluer une conception par objets vers une conception par aspects plus profitable.

Les phases amont d'un cycle de développement d'un système doivent tenir compte des avantages complémentaires procurés par les deux approches Aspect et Patron.

La réutilisation des nouvelles solutions d'implémentation par aspects des patrons Objet est d'autant plus efficace si chacune des ces solutions est associée à une structure conceptuelle abstraite, décrivant cette solution de manière indépendante d'un langage de programmation par aspects particulier. Aucun langage de modélisation par aspects n'est cependant offert pour la représentation de telles structures. Pour s'affranchir de cette limite dans l'évolution des patrons Objet en patrons Aspect, nous proposons de définir un cadre conceptuel pour la modélisation de structures par aspects de patrons, ou dans un contexte plus large encore, l'expression de toute modélisation par aspects non liée à l'utilisation exclusive d'une technique de programmation par aspects particulière. Notre objectif est de promouvoir plus encore, à la manière de l'approche MDA, l'utilisation de divers modèles pertinents à différents niveaux d'abstraction tout au long du processus de conception et de développement, afin notamment de faciliter l'évolution et d'augmenter la réutilisation.

Par ailleurs, pour une évolution plus complète des patrons Objet, il convient de s'intéresser non seulement à la représentation de leurs nouvelles structures, mais également à la description au sein d'un formalisme approprié de toutes les informations utiles et nécessaires à leur bonne application. Notre objectif n'est pas d'introduire un nouveau formalisme de plus dans la littérature abondante de formalismes de description de patrons, mais plutôt d'adapter aux aspects un formalisme déjà existant.

Ce travail sur les aspects et les patrons de conception s'intéresse aussi, dans un cadre plus général, à l'évolution des approches d'ingénierie de systèmes d'information à base de patrons, afin de pouvoir les adopter dans un contexte de développement à base d'aspects. Une telle ingénierie passe nécessairement par la définition, la spécification et l'organisation de patrons de conception destinés à être réutilisés. Un autre objectif principal de notre travail est donc de définir un nouveau système de patrons originaux capitalisant des expertises en matière de conception par aspects. Ces patrons doivent être clairement décrits, coordonnés et hiérarchisés au sein d'un formalisme, permettant ainsi d'offrir un cadre pour une démarche pour réaliser une conception et une programmation par aspects de qualité.

Plan de la thèse

Ce présent rapport de thèse est organisé en cinq chapitres :

- Le premier chapitre constitue un état de l'art sur la réutilisation et les composants, et plus particulièrement les patrons de conception, dans l'ingénierie des systèmes d'information. Il rappelle notamment la notion de patrons de conception par objets, d'ingénierie et de

réutilisation de ces patrons. Il identifie et présente également, à travers un cas d'étude, quelques problèmes et limites pouvant se poser lors de l'utilisation de ces derniers.

- Le deuxième chapitre constitue un état de l'art sur les modèles et langages actuellement proposés dans le cadre de l'approche Aspect, pourvoyant une meilleure séparation des préoccupations (*SoC*). Il met l'accent notamment sur la programmation par aspects (*AOP*) ainsi que sur l'approche *Hyperspace* et son langage Hyper/J. Après avoir introduit le principe de *SoC* et présenté les différents types de préoccupations possibles, ce deuxième chapitre rappelle et analyse les principales approches de développements classiques en identifiant leurs problèmes et limites dans le but de motiver l'approche Aspect.
- Le troisième chapitre traite de la réalisation des patrons de conception par objets à l'aide d'AspectJ et d'Hyper/J. Il vise à trouver des améliorations à la traçabilité de ces patrons, et par là même à leur réutilisation, mais aussi à pallier aux problèmes liés à leur utilisation dans une approche strictement Objet. Il s'agit en effet d'étudier les apports et conséquences, au niveau de l'implémentation de ces patrons de conception, des mécanismes et concepts nouvellement introduits dans le cadre de l'approche Aspect.
- Le quatrième chapitre considère l'intégration des aspects au niveau de la phase de conception, pour l'expression de structures de (patrons de) conceptions par aspects indépendamment d'un langage de programmation par aspects en particulier. Il présente notre approche par métamodélisation et transformation de modèles. Cette approche est basée sur un langage général de modélisation par aspects (Aspect/UML) et deux langages de modélisation spécifiques respectivement à AspectJ (AspectJ/UML) et Hyper/J (HyperJ/UML), tous deux définis comme des extensions d'UML. Un processus et des règles de transformation d'un modèle instance du métamodèle général vers des modèles instances spécifiques respectivement à AspectJ et à Hyper/J sont également explicités dans ce chapitre.
- Le chapitre cinq traite de l'ingénierie des patrons de conception par aspects par abstraction de structures par aspects de patrons de conception déjà existants. Il présente le formalisme de description de patrons par aspects (AP-Sigma) que nous proposons, identifie et spécifie chacun des nouveaux patrons proposés en les mettant en relation pour constituer un véritable système de patrons guidant la conception.

La conclusion de ce rapport reprend les contributions de cette thèse. Elle explicite également les principales perspectives de ce travail de recherche. La figure 1 constitue un guide de lecture de cette thèse.

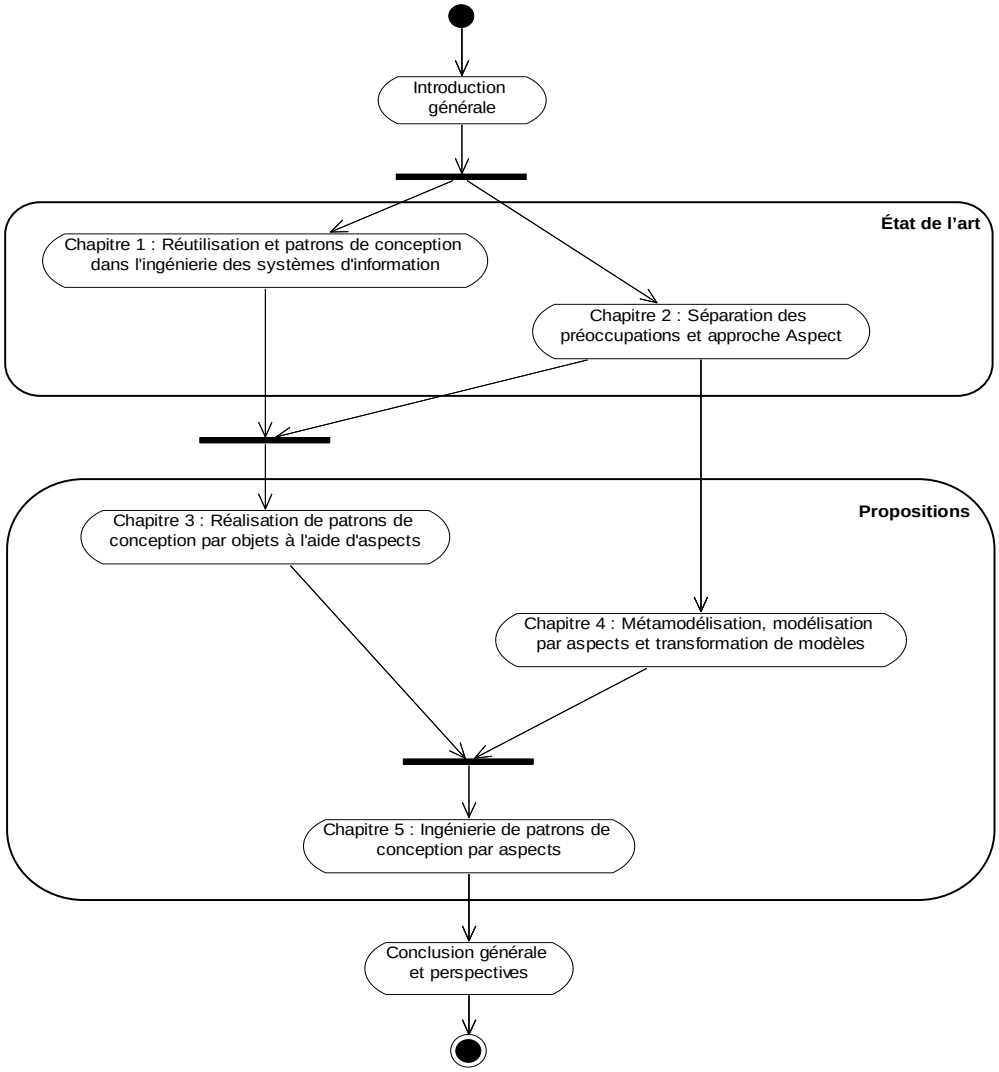


Figure 1. *Plan de la thèse*

Chapitre 1

Réutilisation et patrons de conception dans l'ingénierie des systèmes d'information

Le domaine de l'ingénierie des systèmes d'information (SI) est très demandeur en techniques et méthodes de développement, devant améliorer aussi bien la qualité des produits que la performance des processus utilisés pour les élaborer. Ces besoins ont fait émerger un ensemble de solutions de développement utiles, adaptant un savoir et un savoir-faire important en matière d'ingénierie des SI. Désormais, la complexité croissante de ces systèmes et leur évolution de plus en plus rapide, ont motivé un intérêt accru pour une approche de développement plus efficace : la réutilisation. Cette approche permet de construire un système nouveau à partir d'éléments (objets, composants, etc.) existants, éprouvés et réutilisables. Introduite d'abord pour améliorer la productivité des équipes de programmation, les tendances actuelles exploitent la réutilisation dans les phases en amont du cycle de développement dès l'expression des besoins et la compréhension du domaine du système.

Étant une forme particulière de composants réutilisables, les patrons d'ingénierie constituent en ce sens une des voies les plus pertinentes en matière de réutilisation. Ils répondent, en effet, à ce besoin tout en capitalisant d'importantes connaissances acquises et approuvées par plusieurs développeurs de SI. De telles connaissances sont réutilisables dès la phase de recensement des besoins jusqu'à la phase d'implémentation. Ainsi, une grande variété de patrons a d'ores et déjà été proposée. D'ailleurs, plusieurs techniques et méthodes ont été définies pour l'ingénierie de ces patrons et leur réutilisation dans le développement des SI. Nous nous intéressons plus particulièrement, dans le cadre de cette thèse, aux patrons de conception par objets. Ces patrons sont reconnus comme étant très utiles, ils offrent plusieurs avantages et intérêts. Toutefois, en dépit de leurs nombreuses qualités, l'utilisation de ce type de patrons présente quelques problèmes et limites. Ces limites contredisent un but majeur de la réutilisation, celui d'assurer une meilleure traçabilité des transformations des préoccupations au sein d'un processus de développement « sans couture » (seamless) [Reenskaugh *et al.*, 92], [Barbier 94], [Corriveau 96].

Ce chapitre présente un état de l'art sur la réutilisation et les patrons d'ingénierie en général, et sur les patrons de conception par objets en particulier. La première section définit la réutilisation et présente brièvement ses deux dimensions : dimension produit et dimension processus. La deuxième section traite des patrons d'ingénierie, elle considère un ensemble de méthodes et techniques proposées pour la réutilisation et l'ingénierie de ces derniers. La troisième section introduit les patrons de conception plus concrètement à travers les patrons du GoF (*Gang of Four*) [Gamma *et al.*, 95]. Nous mettons en avant leur utilité et montrons les problèmes et limites de leur utilisation. Pour illustrer nos propos, nous considérons un cas d'étude d'un système simple appelé SEE. La quatrième section termine et conclut ce chapitre.

1. Réutilisation dans l'ingénierie des SI

Cette section introduit le concept de réutilisation dans l'ingénierie des SI, elle présente de plus ses deux dimensions : produit et processus.

1.1 Réutilisation

La réutilisation fut évoquée en 1968 par M. McIlroy dans une conférence de l'OTAN sur l'ingénierie du logiciel (article réédité dans [McIlory 76]), suite à un échec patent de la part de développeurs pour livrer des logiciels de qualité à temps et à prix compétitifs. Elle vise donc trois objectifs : diminuer les coûts de développement et de maintenance, réduire le temps de mise sur le marché (*time-to-market*), et améliorer la qualité du logiciel. La réutilisation se définit comme une nouvelle approche de développement de systèmes, qui propose notamment de construire un système nouveau à partir de composants logiciels sur étagère (*COTS, Commercial Off The Shelf*). Elle s'oppose ainsi aux approches traditionnelles de développement, dans lesquelles la construction d'un nouveau système part de rien (*from scratch*) et nécessite de réinventer de

grandes parties d'applications à chaque fois. Depuis, cette idée s'est imposée et la réutilisation constitue un enjeu majeur de recherche en ingénierie des SI. Introduite tout d'abord pour essentiellement améliorer la productivité des équipes de programmation, en particulier pour le développement des interfaces, les tendances actuelles exploitent la réutilisation dans les phases en amont du processus d'ingénierie des SI : phases de recensement des besoins, d'analyse et de conception. Les recherches sur la réutilisation ont progressivement évolué ces dernières années, passant de la réutilisation de code à la réutilisation de connaissances et de démarches de raisonnement de différents niveaux. Cette pratique présente un enjeu supplémentaire pour garantir une meilleure qualité des développements. Il s'agit d'améliorer la traçabilité des transformations des composants de différents niveaux à travers les diverses phases de développement. Cette nouvelle tendance est aujourd'hui à l'origine de plusieurs types de composants et diverses méthodes de développement basées sur ces derniers, que nous explicitons dans ce chapitre.

La réutilisation dans l'ingénierie des SI concerne en général deux dimensions : la dimension produit et la dimension processus. La dimension produit concerne les différents types de composants réutilisables. Dans la dimension processus sont considérées, d'une part, les différentes activités spécifiques à la production de ces composants, et d'autre part, les activités spécifiques à la mise en œuvre d'une approche de développement par réutilisation de ces composants. Si les propositions concernant les types de composants sont aujourd'hui nombreuses, les méthodes de développement à base de composants réutilisables restent encore assez empiriques. Les deux sous-sections suivantes traitent respectivement de ces deux dimensions.

1.2 Composants réutilisables

Pour intégrer la réutilisation dans tout le processus de développement des SI, une grande variété de composants réutilisables a déjà été proposée. D'une manière générale, un composant est vu comme une solution testée et acceptée pour résoudre un problème fréquemment rencontré lors du développement de systèmes. Toutefois, plusieurs critères [Conte *et al.*, 01a] permettent de distinguer les très nombreux types de composants proposés aujourd'hui dans la littérature.

- *Composants de différents types de connaissances*. Les composants peuvent capitaliser des connaissances de type *produit* ou *processus*. Les composants de type *produit* correspondent au but à atteindre, ils sont destinés à être réutilisés et intégrés directement dans le système en cours de développement. Les patrons de [Gamma *et al.*, 95], les architectures logicielles et les bibliothèques de fonctions sont des exemples de composants produit. Les composants *processus* correspondent plutôt au chemin à parcourir pour atteindre le résultat. Ils servent à définir des fragments de démarches facilitant le développement. Les patrons de [Ambler 98] sont des exemples de composants processus.
- *Composants de différentes couvertures*. La couverture caractérise le degré de généralité des composants. Certains peuvent être *généraux*, d'autres plus spécifique à un *domaine* ou à une *entreprise*. C'est le problème traité par le composant qui détermine sa couverture et donc son type. Si le problème est très fréquent dans de nombreux domaines d'application (resp. spécifique à un domaine particulier, ou à une entreprise particulière) celui-ci est dit général (resp. de domaine, ou d'entreprise). Les frameworks spécifiques à la comptabilité sont des exemples de composants de domaine.
- *Composants de différentes portées*. La portée d'un composant détermine la phase du cycle de développement concernée par celui-ci. Nous distinguons par exemple les composants d'*analyse*, de *conception* ou encore d'*implémentation*. Les frameworks d'architecture sont par exemple des composants de conception. De manière générale, la plupart des composants actuels sont dit « orientés activité » et sont dédiés exclusivement à une phase particulière du cycle de développement. Il existe cependant des composants qui concernent plus d'une phase, et sont

définis avec une certaine forme de traçabilité à travers les différentes phases concernées par ces composants.

- *Composant de différentes natures de solutions*. La solution d'un composant peut être de nature *conceptuelle* ou *logicielle*. Les composants logiciels correspondent à la forme la plus ancienne et la plus répandue de composants réutilisables. Ils permettent de réutiliser des fragments de programmes destinés à être intégrés directement dans le code (au niveau de la phase d'implémentation) [Meijler *et al.*, 97], [D'Souza *et al.*, 99], [Heineman *et al.*, 01]. Les composants CCM [CCM 02] et EJB [EJB 03] sont des exemples de composants logiciels. Les composants conceptuels ([Ambler 98], [Gamma *et al.*, 95], etc.) proposent des petites structures ou des architectures permettant de spécifier les SI.
- *Composants de différents niveaux de granularité*. La granularité mesure le plus souvent le nombre d'unités modulaires constituant le composant. Elle spécifie le degré de complexité de sa structure interne ou de l'architecture qu'il propose. Elle peut être *faible* (moins de 10 entités), *moyenne* (moins de 100 entités) ou *forte* (plus de 100 entités). La plupart des composants proposés sont de faible granularité, ils sont généralement définis par quelques entités modulaires atomiques telles que les classes. Les patrons de conception du GoF [Gamma *et al.*, 95], ou ceux d'analyse de Coad [Coad 92] offrent, par exemple, des petites structures de deux à six classes. Il existe cependant des composants de structures plus complexes. Le framework MVC [Schmuker 87] de smalltalk 80 en est un exemple.
- *Composants de différentes ouvertures*. L'ouverture d'un composant caractérise son niveau de transparence. Selon que son utilisation nécessite ou non la modification de sa structure interne, il convient de distinguer plusieurs types de composants qui vont de la *boîte noire* (où la structure interne est invisible et non modifiable), à la *boîte blanche* (où la structure interne est visible et modifiable), en passant par la *boîte en verre* (où la structure interne est visible mais non modifiable) ou la *boîte grise* (où la structure interne est visible et modifiable par paramètres).

1.3 Processus de réutilisation

Deux processus complémentaires sont considérés dans cette dimension : l'ingénierie de composants réutilisables (*design for reuse*) et l'ingénierie de SI par réutilisation de composants (*design by reuse*).

1.3.1 Ingénierie de SI par réutilisation

L'ingénierie de SI par réutilisation couvre les tâches de *recherche*, de *sélection*, d'*adaptation* et d'*intégration* de composants. La recherche de composants consiste à rechercher un composant parmi une collection, permettant de répondre à un problème particulier. Elle tient donc compte du besoin du développeur. Dans la plupart des cas, il n'y a pas un seul composant candidat mais plusieurs, il est donc nécessaire de comparer l'ensemble des candidats en vue de sélectionner le plus approprié. Une fois sélectionné, le composant doit être adapté au contexte du système en cours de développement. Les techniques d'adaptation sont nombreuses : depuis la modification du composant, jusqu'au mécanisme d'instanciation, en passant par la paramétrisation et la spécialisation [Cauvet *et al.*, 99], [Cauvet *et al.*, 01]. Enfin, la réutilisation d'un composant est effectivement close par l'intégration de celui-ci au système considéré. Il est à noter ici que le processus d'ingénierie de SI peut être très différent, selon qu'il s'agit ou non de considérer l'ensemble des tâches ainsi identifiées. Par exemple, la tâche de recherche est nécessaire dans le contexte de certains types de composants. En revanche, elle a peu d'intérêt, par exemple, dans le contexte d'un framework.

Une approche idéale de réutilisation devrait permettre de générer de manière quasi-automatique un système d'information, de sorte que le rôle du développeur se limite à fournir les propriétés et besoins spécifiques à son système. Une telle approche doit être systématique et complète, pour couvrir l'ensemble

des tâches tout en maîtrisant leur complexité. En ce sens, plusieurs travaux [Soukup 95], [Albin-Amiot *et al.*, 01], [Conte *et al.*, 01c], [Guéhéneuc 03], [Arnaud *et al.*, 04], [Khayati 05], etc. centrent leurs efforts sur la recherche de méthodes systématiques qui sont mieux adaptées au développement de systèmes par réutilisation. Ils privilégient la mise en oeuvre de nouveaux environnements de développement, qui fournissent les fonctionnalités suivantes :

- la gestion des collections de composants,
- la spécification des SI à base de composants,
- l'implémentation par génération de code à partir des spécifications,
- la validation des spécifications des SI,
- la maintenance des SI.

1.3.2 Ingénierie de composants réutilisables

L'ingénierie de composants réutilisables recouvre les tâches d'*identification*, de *spécification*, de *qualification*, d'*organisation* et d'*implémentation* de composants. L'ensemble de ces tâches est essentiel, puisqu'il a pour objectif de produire les ressources nécessaires à la mise en œuvre d'une approche d'ingénierie par réutilisation. Toutefois, il n'existe pas d'approches spécifiquement dédiées à l'ensemble de ces tâches. Nous distinguons en revanche plusieurs techniques allant dans ce sens.

– *Identification*. Les connaissances susceptibles de contenir des éléments réutilisables et approuvés peuvent être des produits de développement existants (programmes, modèles conceptuels, etc.), ou des expressions de connaissances (documents techniques, etc.). F. Semmak [Semmak 98] définit diverses techniques d'identification de composants, appartenant à deux grandes approches. Une première approche dite *approche par rétro-ingénierie*, elle vise à abstraire des éléments réutilisables à partir de l'analyse de produits existants. Cette approche ascendante permet d'évaluer les similarités et les différences entre produits, afin d'identifier de manière systématique des éléments candidats à la réutilisation. Dans ce cas, deux techniques de base sont principalement utilisées : *l'analyse de similarités* [Castano *et al.*, 94], [Spanoudakis *et al.*, 96] et *la recherche par analogie* [Falkerhainer *et al.*, 89], [Gentner 83], [Holyoak *et al.*, 89]. Bien que ces deux techniques aient sensiblement le même objectif, le degré de généralité des composants résultats de leur application est plus grand dans les techniques de recherche par analogie. En effet, la recherche par analogie, contrairement aux techniques d'analyse de similarités, évalue les ressemblances sémantiques entre éléments candidats (tels que la nature des liens entre objets, leur organisation au sein de systèmes, etc.) ; plutôt que leurs similitudes syntaxiques qui rendent nécessaire une terminologie commune et donc le plus souvent mono-domaine. La deuxième approche d'identification de composants est dite *approche par analyse de domaine*, elle consiste aussi à identifier les éléments communs à des systèmes similaires, afin d'élaborer un modèle de domaine. Elle peut couvrir plusieurs sous-tâches, allant d'une analyse de contexte pour définir les limites du domaine et les sources de connaissances en rapport avec celui-ci, jusqu'à une conception détaillée de ces connaissances. Dans le type d'approche, deux techniques qualifiées de *statique* et de *dynamique* [Cauvet *et al.*, 01] sont proposées. Dans la première technique [Wartik *et al.*, 92], il s'agit de considérer une famille de systèmes ayant des fonctionnalités similaires (telles que la gestion des comptes bancaires, la gestion de stock, les systèmes de réservation, etc.) dans l'objectif de produire un modèle de domaine réutilisable : un modèle qui exprime les ressemblances (partie commune) et les différences (parties spécifiques) entre les systèmes considérés. Dans la deuxième technique [Semmak 98], un domaine est défini par un ensemble de problèmes, pour lesquels différentes solutions sont envisageables. Les modèles de domaine obtenus sont donc orientés problèmes et non plus solutions [Grosz *et al.*, 96], contrairement à la première démarche. Comme problèmes

spécifiques à un domaine particulier, nous pouvons considérer, par exemple, l'optimisation des réservations ou la satisfaction des attentes des clients spécifiques au domaine de réservation de places d'avions.

- *Spécification et principes d'abstraction et de variabilité*. La réutilisation des composants est d'autant plus efficace s'ils sont associés à une spécification pertinente. Une telle spécification doit favoriser leur réutilisation tout en fournissant l'ensemble des informations utiles et nécessaires à leur bonne utilisation (telles que les indications et les contraintes d'utilisation, les forces de la solution proposée,…). Elle doit permettre de décider, par exemple, de la pertinence d'utiliser un composant ou un autre au moment de la recherche de ceux-ci. L'*abstraction* et la *généricité* sont deux techniques phares en matières de spécification de composants. Tout modèle de spécification doit idéalement appliquer ces deux principes.

  L'abstraction consiste à distinguer deux types de connaissances dans la spécification d'un composant : les connaissances effectivement réutilisables (un modèle s'il s'agit d'un composant conceptuel par exemple, ou un fragment de programme s'il s'agit d'un composant logiciel) et celles requises pour une utilisation correcte de celui-ci [Cauvet *et al.*, 01]. Ces dernières mettent en avant, par exemple, le problème du composant ou encore son contexte d'utilisation. Nous distinguons ainsi plusieurs approches de spécification : orientée problème, par descripteur de composant (ou par classification) et par méta-connaissance. La première est exclusivement centrée sur la définition explicite du problème du composant. La deuxième propose d'associer à chaque élément d'une collection de composants une liste de mots clés, tout en le classant. La dernière approche s'applique exclusivement sur les composants définis sous la forme de classes. Elle propose de spécifier les composants sur deux niveaux : le niveau application (présentant les parties réutilisables des classes) et le niveau méta (associant des métaclasses aux classes du composant, afin de guider le développeur dans le processus de réutilisation).

  La généricité introduit une forme de variabilité dans la spécification des composants [Cauvet *et al.*, 01]. En effet, le plus souvent, la solution proposée par un composant n'offre qu'une réalisation possible de celui-ci. Pour augmenter sa réutilisation, un composant doit offrir plusieurs réalisations possibles : sa solution doit ainsi être flexible afin de pouvoir l'adapter aux spécificités du système en cours de développement. Ce principe consiste donc à distinguer, dans l'ensemble des connaissances spécifiant le composant (outre la partie réalisation), entre une partie fixe et une partie variable. La partie variable permet d'exprimer différents choix de développement, par exemple, ou encore de définir des extensions possibles de la réalisation proposée.

- *Qualification*. La qualification des composants s'apparente à une forme de validation de ceux-ci. Elle consiste à vérifier la qualité de composants identifiés du point de vue de leur réutilisation. Peu de techniques de qualification sont actuellement proposées, de plus le résultat de cette tâche est en général très subjectif. Une telle qualification relève plus de la « qualité » du processus d'ingénierie des composants réutilisables que des composants eux-mêmes.

- *Organisation*. L'organisation des composants traite des différents types de liens entre ceux-ci, afin de pouvoir utiliser conjointement plusieurs composants pour obtenir des architectures plus importantes. Elle a donc une influence directe sur la manière d'exploiter les composants dans le processus d'ingénierie de SI. Elle facilite la sélection de composants et améliore leur réutilisation. Deux approches d'organisation de composants sont actuellement proposées [Cauvet *et al.*, 01] : *statique* ou *dynamique*. L'approche statique prescrit entièrement le processus de recherche et/ou de composition des composants, sans tenir compte des spécificités du système en cours de développement. Elle est largement utilisée, par exemple, dans les frameworks qui prédéfinissent la structure du produit final. L'approche dynamique propose, en revanche, de coupler un ensemble

de composants avec un processus qui automatise et/ou guide la construction d'un produit par la recherche, la composition, et l'adaptation des composants. Elle ne prédéfinit ni la recherche, ni la composition de ceux-ci, et vise une construction par réutilisation qui tient compte des besoins du système en cours de développement.

– *Implémentation.* L'implémentation des composants consiste à outiller la réutilisation (gestion automatique de leurs collections, recherche, sélection, composition, adaptation, …) de ceux-ci.

Ces différentes techniques contribuent à la mise en place de bases de composants pour capitaliser des savoirs et des savoir-faire réutilisables dans le cadre d'une ingénierie par réutilisation (cf. figure 1.1).

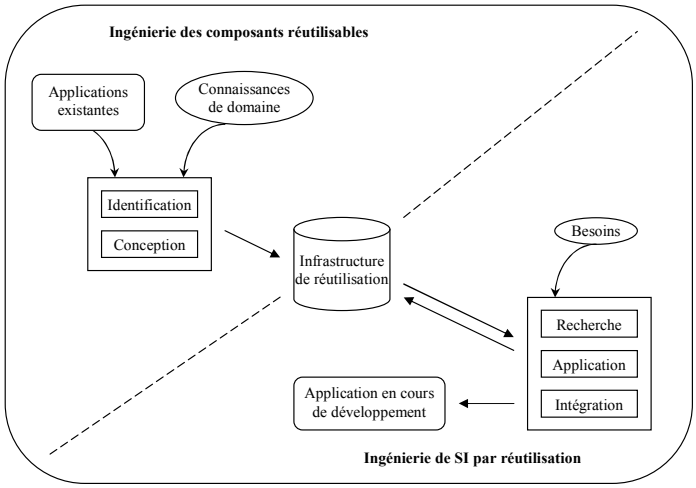


Figure 1.1. *Ingénierie pour la réutilisation et ingénierie par réutilisation de composants [Cauvet* et al.*, 01]*

2. Réutilisation et patrons d'ingénierie

La section 1 nous a permis d'introduire brièvement les différents types de composants réutilisables, et de faire une synthèse de quelques techniques proposées pour l'ingénierie et la réutilisation de ceux-ci. Dans cette section l'accent est mis sur ce qui relève plus particulièrement de la réutilisation dans l'ingénierie des SI à base de patrons.

2.1 Patrons d'ingénierie

Cette sous-section introduit le concept de patron d'ingénierie et le définit. Elle propose de plus de discuter l'utilité des patrons dans l'ingénierie des SI.

Historique. La notion de patron a été généralisée par C. Alexander, dans le domaine de l'architecture des bâtiments [Alexander *et al.*, 77], [Alexander 79]. C. Alexander constatait que les grands bâtisseurs, au fil des siècles, n'avaient pas suivi un modèle préétabli avec des règles rigoureuses, mais que les architectures avaient été adaptées les unes après les autres et à leurs environnements. Il a remarqué avec ses collègues que certaines solutions de conception, considérées comme efficaces et bien acceptées, étaient récurrentes et s'appliquaient dans différentes situations de construction. Cette idée est concrétisée au travers de 253 patrons propres aux problèmes récurrents en architecture décrits d'une manière uniforme pour des raisons de commodité, de clarté et de diffusion [Alexander *et al.*, 77]. Des approches similaires à celle d'Alexander sont apparues dans d'autres disciplines. Nous pouvons citer, par exemple, les patrons dédiés à l'hydraulique [Asplund *et al.*, 73]. L'un des patrons de C. Alexander le plus fréquemment cité est « *a place to wait* » (un endroit pour attendre), qui propose des solutions conceptuelles au problème de l'attente.

UN ENDROIT POUR ATTENDRE [Alexander 79]

Quelle que soit la raison pour laquelle les personnes attendent (salle d'attente d'un médecin, embarquement sur un avion, rendez-vous d'affaire,…), il est inévitable que ces gens tournent en rond et perdent leur temps à ne rien faire. Ils ne peuvent pas mettre à profit ce temps, parce que l'attente n'est pas prévisible et qu'ils doivent attendre sur le lieu même de leur rendez-vous. Tant qu'ils ne sauront pas exactement quand leur tour viendra, ils ne pourront aller se promener ou même s'asseoir à l'extérieur.

C'est pourquoi, dans les endroits où les gens sont susceptibles d'attendre, il faut créer une situation pour rendre cette attente positive. Associer à cette attente une autre activité, journaux, café, télévision, afin que cette pièce ne devienne plus seulement un lieu d'attente mais un lieu d'activité. Ou à l'inverse, transformer celle-ci en un lieu de relaxation et de rêverie.

Dans le domaine de l'informatique, les premiers patrons ont été présentés par K. Beck et W. Cunninghan [Beck *et al.*, 87] lors de la conférence OOPSLA de 1987. Il s'agissait d'une première adaptation du langage de patrons d'Alexander à la conception et à la programmation Objet. Un ensemble de patrons plus spécifiquement dédiés à la conception par objets a été élaboré, un peu plus tard, par E. Gamma [Gamma 91] en 1991 lors des travaux de sa thèse. Ces travaux ont été suivis en 1995 d'un premier ouvrage collectif [Gamma *et al.*, 95]. À la même période, dans le cadre de l'ingénierie de systèmes d'information P. Coad [Coad 92] proposait de faciliter l'analyse d'un système en identifiant les besoins selon 7 patrons pré-établis. Il définit un patron Objet comme une abstraction de structures de classes similaires, une abstraction qui peut être réutilisée, encore et encore, pour le développement d'applications. Depuis, de nombreux ouvrages et articles ont été publiés sur les patrons [Gamma *et al.*, 95], [Coad 95], [Coplien 96], [Buschmann 96], [Coad *et al.*, 97], [Booch 97], [Fowler 97], [Ambler 98], [Larman 02], etc. La première conférence internationale dédiée à la notion de patrons de conception date de 1994 (PLoP), la première conférence européenne s'est déroulée en 1996 (EuroPLoP).

Définition. Nous partageons complètement la définition d'Alexander : « un patron d'architecture, tout comme un patron de couture, capitalise un savoir-faire permettant de résoudre un problème récurrent du domaine (l'architecture ou la couture) … chaque patron décrit à la fois un problème qui se produit très fréquemment dans votre environnement et l'architecture de la solution à ce problème ; de telle façon que vous puissiez utiliser cette solution des millions de fois sans jamais l'adapter deux fois de la même manière ». D'ailleurs, sa définition reste très proche de celle proposée par le dictionnaire : « un patron est une forme finie, originale, un modèle proposé ou accepté pour imitation ; quelque chose considéré comme un exemple destiné à être copié ». De manière générale, un patron constitue une base de savoir et de savoir-faire pour résoudre un problème récurrent dans un domaine particulier. La spécification de ces connaissances réutilisables :

- permet d'identifier le problème à résoudre par capitalisation et organisation de connaissances d'expériences,
- propose une solution possible, correcte, générale et consensuelle pour y répondre,
- et, offre les moyens d'adapter cette solution au contexte spécifique.

Dans le domaine de l'ingénierie des SI les patrons sont des composants réutilisables alliant problèmes, contextes et solutions (cf. § 2.2).

Utilité. Les patrons ont d'indéniables qualités en termes d'usages, mais aussi, en termes d'intérêts dans l'ingénierie des SI. Ils apportent des solutions efficaces à divers problèmes d'analyse et de conception, permettant ainsi de créer des conceptions approuvées et réutilisables. L'écriture des programmes est également facilitée par les patrons qui proposent des consignes types pour une bonne implémentation. D'ailleurs, la validation de spécifications de systèmes est souvent guidée par des patrons, qui conduisent à de meilleures conceptions. Les patrons offrent également un vocabulaire commun à l'ensemble de l'équipe

de développement [Fowler 97], [Gamma *et al.*, 95], [Larman 02]. Ils améliorent la documentation, et donc la compréhension des applications en cours de développement. Celles-ci deviennent, par conséquent, plus facile à faire évoluer et à maintenir.
En termes d'intérêts, les patrons constituent un véritable moyen de transfert de connaissances. Ils permettent l'échange du savoir et savoir-faire des développeurs expérimentés, ayant rencontré et résolu plusieurs fois les mêmes problèmes [Schmidt 95]. Ils offrent un gain de temps et une efficacité, tout en réduisant largement les tâtonnements fastidieux nécessaires à l'élaboration d'un système de qualité. De plus, tout en augmentant les facultés d'abstraction des développeurs, les patrons apportent également une meilleure compréhension des systèmes. Ils permettent, en effet, d'aborder des systèmes complexes de manière plus simple. Ils aident à mieux structurer les applications, guidant la perception du monde réel et permettant d'en obtenir une description à un niveau d'abstraction élevé. Il devient ainsi facile, par exemple, de comprendre le fonctionnement d'un framework, sans qu'il soit nécessaire de se pencher sur les détails de son implémentation. Les patrons capturent donc les propriétés essentielles d'une architecture, tout en masquant les détails non pertinents. Ils facilitent, de la sorte, la communication autour d'abstractions, de concepts et de techniques que décrivent ou portent ces patrons, dans le but d'assurer de meilleures compréhension, évolution et maintenance des applications. D'autant plus que la documentation de celles-ci se trouve enrichie et simplifiée, facilitant le transfert des spécifications, des modèles et des concepts, notamment avec le personnel non technique qui n'est pas apte à comprendre tous les détails d'implantation, mais est, en revanche, plus disposé à saisir les abstractions véhiculées [Schmidt 95].

2.2 Différents types de patrons d'ingénierie de SI

Dans ce qui suit, nous introduisons les différents types de patrons que nous illustrons avec quelques exemples concrets. Parmi les différents critères de classification des composants (cf. § 1.2), trois critères sont particulièrement intéressants pour classifier les patrons d'ingénierie : type de connaissance, couverture et portée. Les autres critères prennent des valeurs uniques caractérisant ainsi ce type de composants. En effet, tous les patrons offrent des solutions conceptuelles, correspondent à des composants de type « boîte blanche » et ont une granularité faible. En fait, d'une manière générale la nature de la solution d'un patron peut être logicielle ou conceptuelle. Cependant, comme un patron capitalise des connaissances qui traitent de problèmes logiciels et de leurs solutions, il est de nature conceptuelle plutôt que logicielle. Les solutions conceptuelles proposées par les patrons sont exprimées, en général, sous la forme de spécifications semi-formelles qu'il s'agit d'adapter pour réutiliser. Ces spécifications décrivent un savoir ou un savoir-faire capitalisant des connaissances réutilisables, ayant des forces et des faiblesses, et précisent les moyens pour adapter ces connaissances. Un patron est ainsi un composant de type boîte blanche, dont il faut connaître le contenu pour le réutiliser. Comme l'indique P. Coad, dans sa définition des patrons, la solution d'un patron est en général constituée d'un petit nombre de classes qui ne dépasse pas la dizaine, ce qui constitue une granularité faible pour les patrons [Coad 95].

En faisant varier les trois critères retenus pour comparer les patrons, nous en distinguons plusieurs types (cf. figure 1.2). Les patrons les plus connus, comme ceux du GoF [Gamma *et al.*, 95] et les GRASP [Larman 02] par exemple, sont des patrons généraux de type produit, dédiés à la phase de conception. Les patrons de P. Coad [Coad 95] sont cependant dédiés à la phase d'analyse. Les patrons de [Adams *et al.*, 95] concernent plutôt la phase de recensement des besoins. Les patrons de S.W. Ambler [Ambler 98] sont, par contre, des patrons processus généraux qui couvrent l'ensemble du cycle de développement des systèmes. Ils fournissent des collections de démarches, d'actions et/ou de tâches, à suivre pour le développement des systèmes. Les patrons proposés par L. Gzara [Gzara 00] sont spécifiques à un domaine d'application (les systèmes d'information Produit – SIP- ou Product Data Management). Ils couvrent les étapes d'analyse et de conception des SIP, tout en combinant patrons produit et patrons processus.

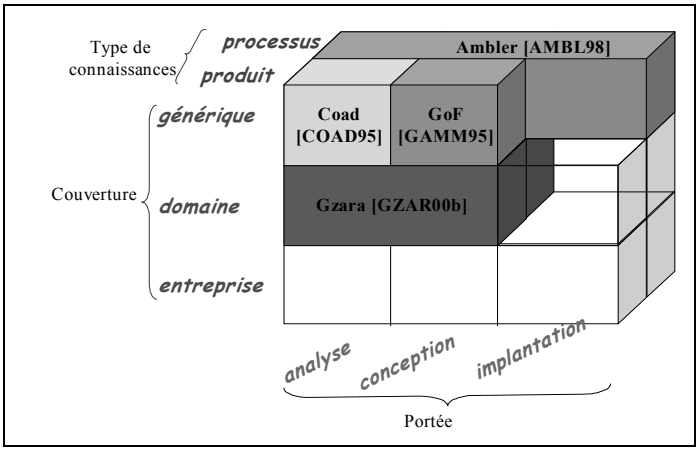
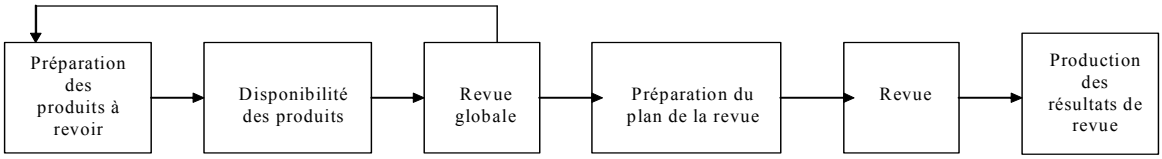


Figure 1.2. *Système de patrons* [Conte *et al.*, 01c]

2.2.1 Patrons produit et patrons processus

Selon le type de connaissances capitalisées par les patrons, nous distinguons les patrons produit et les patrons processus. Les *patrons produit* décrivent un savoir. Ils offrent une prescription, que l'on adopte ou pas, pour construire tout ou partie des spécifications ou de l'implantation du système à développer. Quelques exemples concrets de patrons produit sont présentés dans le paragraphe suivant (cf. § 2.2.2). Les *patrons processus* décrivent un savoir-faire sous la forme d'actions et/ou de tâches destinées à être suivies pour le développement des systèmes. Les actions composantes peuvent elles-mêmes être des patrons. Un patron processus décrit en général un processus de développement dans le but de répondre à des besoins spécifiques ; il est ainsi exprimé sous la forme d'adaptations à une situation concrète. Le patron Revue Technique (*Technical Review*) (cf. tableau 1.1), extrait du système de patrons proposé par Scott W.Ambler [Ambler 98], est un exemple d'un patron processus.

Tableau 1.1. *Exemple d'un patron processus (Technical Review [Ambler 98])*

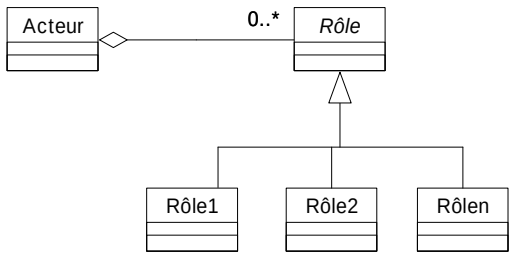
Patron Revue Technique (*Technical Review*) [Ambler 98]
Type. Patron processus de type tâche
But. Décrit comment organiser, conduire et suivre la revue des délivrables d'un logiciel.
Forces. Il y a 4 raisons pour lesquelles le patron "Revue Technique" doit être appliqué. − Les délivrables (des modèles, etc.) ont besoin d'être validés pour que le développement progresse. − Les détections plus tôt des défauts d'un délivrable font baisser le coût de développement. − Un travail est mieux validé par autrui. − L'avancement du travail doit être notifié aux autres.
Contexte initial. Il y a des délivrables à inspecter et ils sont prêts à être inspecté.
Solution. 
Contexte résultant. Les qualités des délivrables (par rapport aux exigences des utilisateurs) sont assurées par l'équipe d'inspection. L'équipe de développement comprend mieux les délivrables produits et comment ils sont intégrés au projet en cours.

2.2.2 Patrons de différents niveaux d'ingénierie

Les patrons sont le plus souvent classifiés en fonction de la phase d'ingénierie à laquelle ils s'adressent. Nous distinguons, par exemple, les patrons d'analyse, de conception, etc. Les métapatterns et les patrons d'architecture sont d'autres types de patrons qui s'adressent généralement à la phase de conception.

- Les *patrons dédiés à la phase de recensement de besoins* permettent une description efficace des besoins des utilisateurs, sous la forme de comportements génériques attendus du système à développer. Il s'agit en général de comportements fonctionnels. Dans certains cas, les comportements non fonctionnels, tels que la performance ou la fiabilité, peuvent cependant être considérés. Les patrons dédiés aux systèmes de télécommunication tolérants aux pannes [Adams *et al.*, 95] traitent, par exemple, de quelques besoins non fonctionnels tels que la fiabilité [Rising 00]. Usuellement, dans la littérature, les patrons de recensement de besoins font partie des patrons d'analyse.
- Les *patrons d'analyse* traitent des problèmes qui apparaissent lors de l'analyse des besoins des systèmes. Ils aident le développeur dans la construction de modèles devant représenter au mieux les besoins de son SI. Un patron de gestion de ressources permet, par exemple, de répondre à un problème de ressources humaines qui comprend l'affectation de personnes à des activités (ou encore, à un problème de gestion de ressources matérielles, pour réaliser l'affectation de machines dans un contexte de production). Le patron Rôle (cf. tableau 1.2) proposé par P.Coad [Coad 92] est un exemple de patron d'analyse.

Tableau 1.2. *Exemple d'un patron d'analyse générale (Rôle [Coad 92])*

Patron Rôle [Coad 92]
Intention. Le patron Rôle donne la possibilité à un objet d'adhérer et/ou de perdre plusieurs rôles.
Motivation. Tout objet peut avoir des propriétés variables selon le rôle joué. Un livre de bibliothèque a, par exemple, des propriétés intrinsèques telles que le numéro d'exemplaire, le N° ISBN, le nombre de pages, etc. Il pourra jouer plusieurs rôles fortement dépendants des processus métiers de la bibliothèque : l'emprunt, la réservation et la maintenance.
Solution. La classe Acteur est associée à une classe Rôle abstraite. Chaque acteur est lié à des instances des classes Rôles spécifiques concrètes (Rôle1, Rôle2, etc.), chacune d'elles représentant un de ses rôles. Cette approche a l'avantage d'être plus concise et flexible que l'utilisation de l'héritage multiple. La figure ci-dessous montre le diagramme de classes associé à ce patron. 

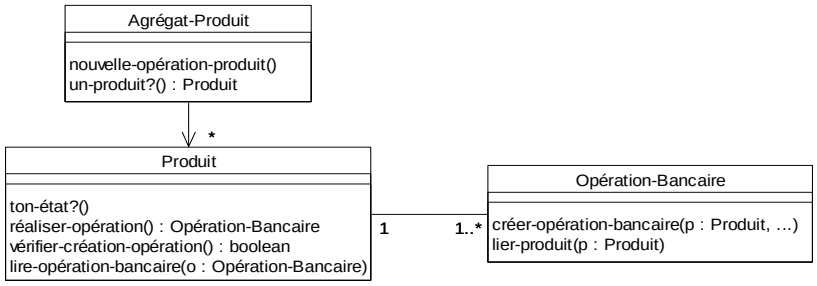
Rôle est un patron d'intérêt général. Il existe aussi des patrons d'analyse qui sont plus dédiés à des métiers particuliers, tel que le patron Nouvelle-Opération-Bancaire [Front *et al.*, 99] (cf. tableau 1.3).

Tableau 1.3. *Exemple d'un patron d'analyse de domaine (Nouvelle-Opération-Bancaire [Front* et al.*, 99])*

Patron Nouvelle-Opération-Bancaire [Front *et al.*, 99]
Intention. Traiter uniformément toute demande de prise en compte d'une nouvelle opération bancaire.

Motivation. La prise en compte d'une nouvelle opération bancaire doit être traitée de manière uniforme pour faciliter la maintenance et l'évolution du système. Une opération bancaire peut être un versement ou un retrait sur un compte, mais aussi la création d'un nouveau compte par exemple.

Solution. La classe Agrégat-Produit détient une méthode permettant la réalisation de la demande d'opération ; elle connaît et contrôle ses produits. La classe Produit connaît et contrôle la création de ses opérations bancaires. La classe Opération-Bancaire maintient un lien vers son produit.



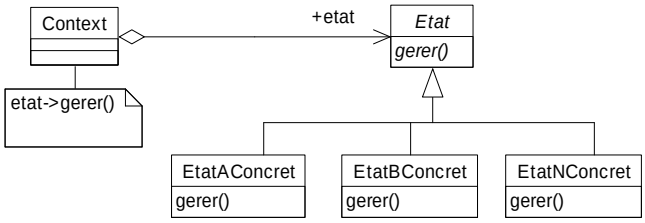
– Les *patrons de conception* (*design patterns*) sont certainement les patrons les plus connus et sans doute les plus utilisés parmi les patrons d'ingénierie. Ils identifient, nomment et abstraient les connaissances convenues de plusieurs expériences, qui répondent à des problèmes de conception communs (qu'il s'agisse de conception détaillée, globale ou liée à des architectures particulières). Le patron État (cf. tableau 1.4) du GoF [Gamma *et al.*, 95] est un exemple de patron de conception.

Tableau 1.4. *Exemple d'un patron de conception générale (État [Gamma* et al*., 95])*

Patron État [Gamma *et al.*, 95]

Intention. Permet à un objet de modifier son comportement quand son état interne change. Tout se passe comme si l'objet changeait de classe.

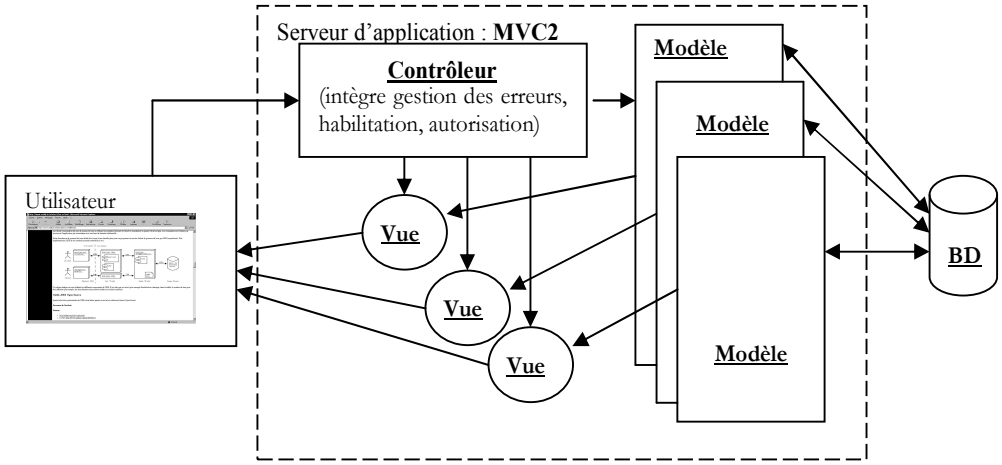
Structure.



Constituants. La classe Contexte définit l'interface intéressant les clients (les états sont transparents pour l'utilisateur). Chaque contexte est lié à une instance qui définit son état courant. La classe Etat spécifie l'interface commune de toutes ses sous-classes ; il s'agit d'une classe abstraite. Les sous-classes concrètes de la classe État implantent le comportement spécifique associé à un état du contexte.

– Les *patrons d'architecture* sont des patrons de conception qui déterminent la structure de base d'une application. Ils offrent un ensemble de sous-systèmes prédéfinis, spécifient leurs responsabilités, et les collaborations et communication entre ceux-ci. Ils incluent, de plus, les règles et les conseils pour établir leurs associations [Buschmann *et al.*, 96]. Tout patron d'architecture décrit en effet les règles d'organisation et de fonctionnement, les stratégies de haut niveau, les propriétés et les mécanismes globaux d'une application, permettant ainsi de créer des architectures extensibles et inter-opérables. Le patron MVC2 (cf. tableau 1.5) permet par exemple de dissocier le modèle de l'interface [Turner *et al.*, 03].

Tableau 1.5. *Exemple d'un patron d'architecture (MVC2 [Turner* et al.*, 03])*

Modèle-Vue-Contrôleur 2 [Turner *et al.*, 03]
Intention. Assure l'indépendance du modèle par rapport à l'interface. Les objets sont séparés en trois composants : le modèle, la vue et le contrôleur. Le modèle contient la logique et l'état de l'application. La vue représente l'interface utilisateur. le contrôleur réagit aux requêtes de l'utilisateur en effectuant les actions nécessaires sur le modèle.
Solution. 

– Les *patrons d'implémentation* (ou *idiomes*) sont des patrons de bas niveau qui traitent des problèmes d'implémentation. Ils peuvent décrire, par exemple, comment implémenter certains mécanismes absents dans des langages (simulation de l'héritage multiple en Java, par exemple). Ils sont généralement spécifiques à une technologie, voire, à un langage de programmation donné.

Nous avons illustré quatre types de patrons de différents niveaux du processus de développement d'un SI. À travers l'ensemble des exemples présentés, nous avons remarqué qu'il sont principalement considérés comme étant des composants génériques. Ils sont également décrits indépendamment d'une technologie donnée, même si la solution générique proposée par un patron peut comporter des conseils d'implémentation et des exemples de code [Gamma *et al.*, 95]. Il est à noter enfin qu'il existe un autre type de patrons, il s'agit des *métapatterns*. Ces patrons constituent des bases structurelles pour la définition de nouveaux patrons de conception similaires [Pree 95]. Ils sont souvent définis en factorisant, à un niveau d'abstraction plus élevé que celui considéré par les patrons de conception, les similarités des patrons déjà existants. Par analogie avec les contre-exemples en pédagogie, il existe également les contre-patrons (*antipatterns*) qui représentent des solutions erronées couramment rencontrées dans des résolutions de problèmes [Brown *et al.*, 98].

2.3 Collections et formalismes de description de patrons

Les patrons interdépendants, proposés dans la littérature, sont décrits d'une manière uniforme par des formalismes. Ils sont d'ailleurs regroupés dans des collections de patrons.

2.3.1 Formalismes de description de patrons

La spécification (cf. § 1.3.2) d'un patron peut, en général, être décrite sous une forme narrative (c'est le cas pour le patron « *a place to wait* » (cf. § 2.1)), ou sous une forme structurée qui améliore la lisibilité des formalismes (cf. patrons de [Gamma *et al.*, 95] par exemple). Un formalisme est donc la structure (i.e. ensemble de rubriques) que l'on peut utiliser, pour décrire uniformément les spécifications de plusieurs

patrons interdépendants. De nombreux formalismes de description sont proposés dans la littérature [Coad 92], [Gamma *et al.*, 95], [Ambler 97], [Fowler 97]. La plupart de ceux-ci sont quasiment équivalents. Ils diffèrent, le plus souvent, par le nombre et le degré de détail (plus ou moins élevé) de leurs rubriques ainsi que par le type des patrons qu'ils décrivent. Toutefois, chaque formalisme doit décrire obligatoirement pour chaque patron, le contexte, le problème, et la solution proposée pour celui-ci. Le formalisme de S.W. Ambler [Ambler 98] est dédié, par exemple, à la description des patrons processus. Le formalisme de E. Gamma [Gamma *et al.*, 95], présenté ci-dessous, est en revanche dédié à la représentation de patrons de type produit. Il comporte 14 rubriques :

- *Nom*. Identifie le patron.
- *Classification*. Permet d'organiser les patrons selon deux critères. Le *rôle* qui traduit ce que fait le patron (création, structuration, comportement), et le *domaine* qui précise si le patron s'applique en général à une ou plusieurs classes ou objets.
- *Intention*. Précise le problème de conception soulevé par le patron.
- *Alias*. Donne d'autres noms connus pour le patron.
- *Motivation*. Présente un scénario d'application du patron posant son problème dans un contexte particulier. Elle permet de mieux comprendre l'intérêt et la solution du patron.
- *Indications d'utilisation*. Reconnaît des situations dans lesquelles le patron peut être utilisé.
- *Structure*. Représente les classes participant dans la solution du patron et montre leurs interactions dans des diagrammes OMT.
- *Constituants*. Définit les classes et/ou objets participants ainsi que leurs responsabilités.
- *Collaboration*. Discute la manière dont les constituants collaborent pour assumer leurs responsabilités.
- *Conséquences*. Décrit comment le patron réalise ses objectifs et présente les résultats, compromis et impacts de l'application de la solution du patron.
- *Implantation*. Explique la technique d'implémentation du patron. Elle présente les pièges existants et propose des solutions types en fonction du langage utilisé, si elles existent.
- *Exemple de code*. Donne des parties de code (dans le langage C++) illustrant la solution du patron.
- *Utilisations remarquables*. Présente des imitations du patron considéré dans des applications connues (dans des frameworks, par exemple).
- *Patrons apparentés*. Référence d'autres patrons utilisant ou utilisés par celui-ci.

Par respect à ce qui a déjà été énoncé dans le section 1.3 (Ingénierie des composants réutilisables), un formalisme idéal doit appliquer les principes d'abstraction et de généricité (ou variabilité). Il est à remarquer, cependant, que la plupart des formalismes existants présentent quelques limites qui défavorisent la réutilisation. En effet, les informations utiles pour sélectionner, adapter et utiliser un ou plusieurs patrons en coordination, sont disséminées dans de nombreuses rubriques. Elles sont de plus décrites, le plus souvent, en langage naturel sous la forme de texte non structuré, et donc mal adaptées à des techniques automatisées d'utilisation de patrons. Seules les rubriques décrivant les réalisations des patrons sont, en général, formelles (Diagrammes UML, etc.). Elles sont toutefois trop souvent incomplètes car limitées à des diagrammes de classes. Enfin, la plupart des relations régissant les ensembles de patrons interdépendants ne sont pas assez significatives pour guider la réutilisation et la combinaison de plusieurs patrons. Les recherches actuelles proposent en ce sens d'uniformiser les formalismes de description des patrons produit et processus, de manière à permettre leur coexistence et améliorer leurs descriptions. C'est le cas par exemple du formalisme P-Sigma [Conte *et al.*, 01a], ayant pour but d'homogénéiser et d'extraire une sémantique commune à divers formalismes de représentation de patrons. Dans le chapitre 5 nous étendons le formalisme P-Sigma pour décrire des patrons par aspects.

2.3.2 Collections de patrons

Les patrons interdépendants sont organisés et regroupés dans des collections de patrons. Nous distinguons principalement les catalogues, les systèmes et les langages de patrons. Ces trois types de collections se différencient par leur organisation et leurs objectifs.

- Un *catalogue de patrons* est une collection de patrons regroupés dans un petit nombre de catégories variées, incluant généralement des références croisées entre les patrons [Buschmann *et al.*, 96]. Tout catalogue de patrons propose d'aider les développeurs à sélectionner et à choisir les patrons appropriés à leurs besoins spécifiques, à l'aide de la classification qu'il propose et des relations entre les patrons qu'il regroupe. Les patrons proposés par E. Gamma [Gamma *et al.*, 95] forment, par exemple, un catalogue de patrons dont la classification est essentiellement basée sur les deux critères de sélection rôle et domaine, et les relations entre patrons se réduisent à la relation patrons apparentés. [Rising 00] propose également un catalogue de patrons, qui regroupe les patrons par domaine d'application.
- Un *système de patrons* ajoute une structure profonde, des interactions plus précises, et une uniformité par rapport aux catalogues de patrons [Appleton 97]. Il définit un ensemble de patrons combinés dans un tout cohésif, qui relève les catégories et les relations inhérentes à chacun de ceux-ci, dans le but d'atteindre un objectif commun [Front 97]. Les relations entre patrons d'une telle collection définissent, de plus, les règles permettant de combiner ces derniers. L'objectif est de faire apparaître facilement la nécessité d'autres patrons, vis-à-vis d'un patron donné, et d'exprimer leurs combinaisons possibles. Les systèmes de patrons permettent ainsi de guider les développeurs en leur offrant des architectures réutilisables : ils présentent des collections de solutions qui coopèrent pour résoudre un problème complexe, de façon ordonnée, pour obtenir un but prédéfini. Dans un système de patrons, la collection doit être suffisamment complète et organisée pour répondre à une problématique commune [Gzara 00]. Un système de patrons bien élaboré doit permettre aux développeurs d'avoir la liberté d'exprimer leurs problèmes, et de concevoir eux-mêmes les solutions qui répondent à leurs besoins spécifiques (dans le contexte où les patrons sélectionnés peuvent être appliqués). Cette notion de système a été introduite par l'architecte C. Alexander.
- Un *langage de patrons* est une collection structurée de patrons, qui permettant de transformer des besoins et des contraintes en une architecture réutilisable [Coplien 98]. Il intègre des règles et des guides précisant comment et quand appliquer un ensemble de patrons en vue de résoudre un problème spécifique, qu'un patron utilisé seul ne pourrait résoudre. Ces règles et guides suggèrent l'ordre d'application des patrons du langage. La principale différence entre un langage et un système de patrons réside dans la robustesse, la plus grande facilité de compréhension et la complétude d'un langage. Les langages de patrons montrent, dans l'idéal, toutes les combinaisons possibles de patrons et leurs variations pour produire des architectures complètes et approuvées. Tout langage de patron inclut une véritable démarche d'utilisation de patrons partant d'un problème complexe et proposant, au travers des patrons, un enchaînement de solutions de plus en plus fines. Dans un langage de patrons, le guidage dans l'enchaînement d'utilisation des patrons est plus directif que dans un système de patrons.

2.4 Ingénierie et réutilisation des patrons

Les patrons constituent une voie très pertinente dans l'ingénierie des SI. Il est désormais essentiel de s'intéresser à leur utilisation, afin d'être en mesure de développer efficacement des systèmes de qualité. Il est indispensable d'ailleurs de s'intéresser à leur ingénierie. Nous présentons brièvement, dans ce qui suit,

un ensemble de techniques existantes offrant de bonnes potentialités pour la réutilisation (imitation, intégration,…) et l'ingénierie des patrons (relations alternative, extension,…).

2.4.1 Réutilisation des patrons pour l'ingénierie des SI

Un patron est une forme plus ou moins réalisée, qui offre un modèle accepté et proposé pour une *imitation*. Il s'agit ensuite d'*intégrer* le résultat d'une ou de plusieurs imitations, afin de disposer d'un modèle cohérent d'un système. Imitation puis intégration d'imitations sont les deux opérations fondamentales pour développer un système d'information à base de patrons réutilisables, en plus des opérations de recherche et sélection (cf. § 1.3.1, l'imitation est l'équivalent de l'adaptation). Nous illustrons ces deux opérations à travers l'exemple du patron « Nouvelle opération bancaire » introduit dans la section 2.2.2.

Imitation. L'imitation d'un patron consiste à *dupliquer* puis *adapter* sa solution à un contexte spécifique. Adapter un duplicata d'un patron revient, à renommer, redéfinir, ajouter ou encore supprimer une ou plusieurs propriétés des classes de la solution de ce patron. Pour illustrer cette opération d'imitation, considérons un concepteur qui désire spécifier, dans le contexte d'un système bancaire, une opération de création d'un nouveau compte. Nous supposons que ce concepteur dispose d'un catalogue de patrons, qui contient le patron « Nouvelle opération bancaire ». Pour obtenir le modèle résultat, il s'agit tout simplement de réaliser une duplication de la solution de ce patron, suivie d'une adaptation concrète des 3 classes génériques (cf. figure 1.3). Cette adaptation consiste, par exemple, à renommer les classes Agrégat-Produit, Produit et Opération-Bancaire qui deviennent respectivement Agence, Client et Compte. Cette adaptation peut se poursuivre par la redéfinition, l'ajout ou la suppression d'autres propriétés. C'est le cas, par exemple, de la méthode verser() ajoutée à la classe Compte, ou encore, le cas de l'argument Compte ajouté à l'opération créer-opération-bancaire().

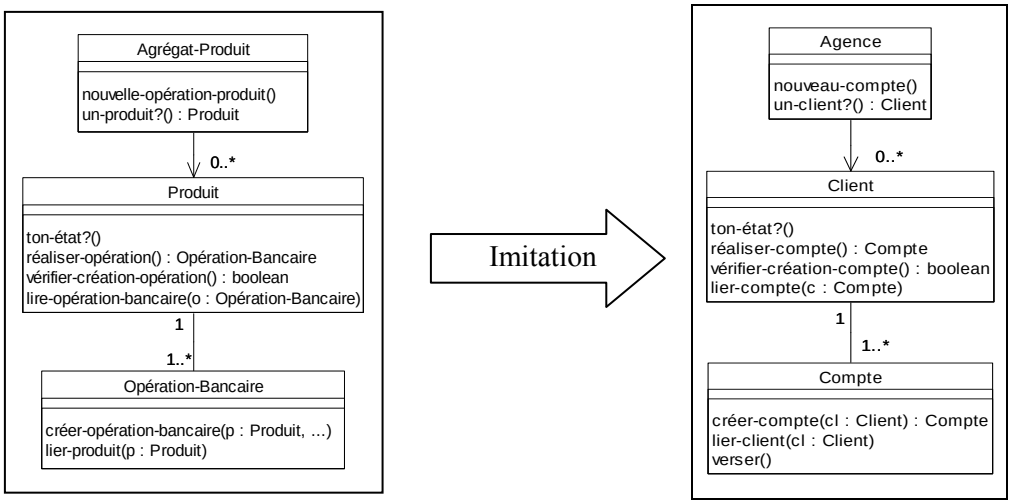


Figure 1.3. *Exemple d'imitation d'un patron*

Il est intéressant de noter ici que l'imitation de la solution d'un patron peut conduire à réaliser de mauvaises imitations. En effet, l'expression de la solution d'un patron de manière semi-formelle est généralement insuffisante pour obtenir une bonne imitation. Il faut considérer, en plus, toutes les informations utiles qui expliquent, complètent et précisent cette solution semi-formelle, de manière à respecter l' « essence » [Arnaud *et al.*, 04] d'un patron au moment de son imitation. Les travaux de N. Arnaud [Arnaud *et al.*, 04] traitent, par exemple, de ce problème. Dans le cadre de nos travaux, nous ne considérons pas ce problème. Nous nous intéressons, en revanche, au problème de traçabilité des imitations de patrons après leur intégration dans une spécification plus large.

Intégration d'imitations de patrons. Afin de disposer d'un modèle cohérent du système à concevoir, il est indispensable de savoir intégrer le résultat d'imitations de plusieurs patrons (ou de plusieurs imitations d'un même patron) qui correspondent à plusieurs sous-modèles. Les imitations doivent absolument rester correctes vis-à-vis des essences de leurs patrons imités pour une intégration efficace. Cette opération d'intégration est aussi problématique. Nous distinguons, en particulier, les problèmes de synonymies et d'homonymies de classes, mais aussi de propriétés. Un autre problème est celui de la cohésion du modèle résultat. L'*union* et l'*association* [Rieu *et al.*, 99] sont deux techniques de base à utiliser afin de pallier ces problèmes et ainsi aboutir à la spécification d'un système cohérent. L'union consiste à remplacer deux classes par une seule, qui regroupe alors l'ensemble des propriétés des deux classes initiales. Cette opération doit tenir compte des conflits de noms et des polysémies sur les propriétés. Il est intéressant de remarquer qu'il ne s'agit pas d'une simple union de classes, mais plutôt, d'une union de deux rôles qui sont joués par un compte dans deux cas d'utilisation par exemple : dans cette situation, nous pouvons utiliser le patron « Rôle » de P. Coad (cf. § 2.2.2) pour réaliser cette intégration. Les associations servent ensuite à lier les différents éléments du modèle résultat. [Rieu *et al.*, 99] montre des exemples d'intégrations de plusieurs imitations.

2.4.2 Ingénierie de patrons par réutilisation de patrons existants

La mise en œuvre d'une approche par réutilisation de patrons passe obligatoirement par l'ingénierie de ceux-ci. Cette ingénierie fait appel, le plus souvent, à des connaissances empiriques qui peuvent être formalisées ou non sous la forme de patrons pour apporter une solution consensuelle à un problème donné. Il est évident ainsi que bon nombre de patrons sont actuellement issus d'analyses de domaines et de systèmes existants. Toutefois, il est actuellement rare qu'un problème ne puisse être résolu, du moins en partie, par des patrons existants. Il est possible ainsi d'identifier de nouveaux patrons à partir d'autres patrons déjà existants ; la production finale de ces patrons doit alors suivre le reste des tâches (spécification, qualification, organisation et implémentation) identifiées dans la section 1.3.2.

Plusieurs techniques existantes, basées sur diverses relations organisant les patrons entre eux, peuvent être utilisées afin de produire de nouveaux patrons à partir de patrons existants. Nous supposons disposer, par exemple, d'une ou de plusieurs collections de patrons, qui peuvent être de même type ou de types différents (métier spécifique/commun, de conception, d'analyse, etc.). Ces patrons sont, le plus souvent, liés entre eux par des relations primaires ou des relations secondaires (exprimées en fonction des relations primaires) [Rieu *et al.*, 99], [Khayati 05]. Les relations primaires sont au nombre de trois [Zimmer 95], [Noble 98a], [Noble 98b] : les relations *alternative*, *extension/raffinement* et *utilisation*. Ces relations permettent, à la fois, de positionner le problème d'un patron donné vis-à-vis des problèmes existants, et de situer sa solution par rapport aux solutions proposées par d'autres patrons. Cette organisation doit permettre un positionnement précis de nouveaux patrons, par rapport à des patrons existants. Il s'agit d'extraction de nouveaux problèmes et de découverte de nouvelles solutions, pour la construction de nouveaux patrons. Une construction est guidée principalement par les relations primaires. Nous définissons dans ce qui suit chacune de ces relations, et nous expliquons comment on peut construire, sur la base de ces relations, de nouveaux patrons. Nous utilisons ces relations, dans le cadre de nos travaux, afin de pouvoir définir nos patrons spécifiques à la conception par aspects.

Utilisation. Chaque patron peut établir une référence vers d'autres patrons qu'il peut utiliser, ou par qui il peut être utilisé. Par définition, *un patron P1 utilise un patron P2, si une partie des problèmes posés par P1 peuvent être résolus en partie ou complètement par P2* [Rieu *et al.*, 99]. Cette relation est présente, par exemple, dans le formalisme de P. Coad dans la rubrique Combinaison. Elle est considérée, également, dans le formalisme de E. Gamma au niveau de la rubrique Patrons apparentés. Elle permet de construire de nouveaux patrons qui utilisent des patrons existants. Comme un patron transforme généralement un problème soit en

solution soit en nouveaux problèmes, il utilise donc d'autres patrons qui peuvent résoudre en partie ses problèmes. Ceci est d'ailleurs une garantie de robustesse de la solution proposée par le nouveau patron.

Alternative. Dans la plupart des collections, les patrons se réfèrent à des alternatives appartenant à d'autres collections ou à la même collection. Par définition, *un patron P1 est une alternative d'un patron P2, si P1 a le même problème que P2 mais propose une solution différente* [Rieu *et al.*, 99]. En effet, les patrons n'ont pas de caractère définitif ; ils peuvent évoluer avec le temps ou disparaître selon les besoins. D'autant plus qu'il existe rarement une solution unique à un problème donné. Tout patron peut admettre donc des alternatives en proposant de nouvelles solutions conceptuelles ; il permet ainsi la définition de nouveaux patrons. M. Fowler [Fowler 97a] propose, par exemple, des alternatives conceptuelles au patron Rôle de P. Coad [Coad 92].

Extension et raffinement. Le raffinement est une restriction du problème d'un patron. Inversement, l'extension est une généralisation du problème du patron. Par définition, *un patron P1 est une extension d'un patron P2, si P1 peut résoudre les problèmes posés par P2. P2 est ainsi un raffinement de P1* [Rieu *et al.*, 99]. Nous expliquons ces deux relations à travers l'exemple d'extension des patrons Rôle de P. Coad et État de E. Gamma (déjà présentés dans la section 2.2.2). Le patron Rôle permet d'associer différents rôles à une même instance au cours de son cycle de vie. Le patron État permet la gestion de l'évolution d'un objet à travers son état, dans le contexte d'un rôle donné. Une extension des problèmes soulevés par ces deux patrons permet d'identifier un nouveau patron nommé Rôle-Dynamique-d'Objet présenté en détail dans [Rieu *et al.*, 99]. L'intention de ce patron est d'associer différents rôles à un objet et de pouvoir gérer l'évolution de celui-ci dans le cadre d'un rôle donné. Il permet de représenter des objets multirôles, et d'établir la différence entre les rôles statiques (cas du patron Rôle) et les rôles dynamiques (application du patron Etat sur les différents rôles). La solution de ce nouveau patron consiste donc en une extension par l'union des solutions proposées respectivement par ces deux patrons. Il existe d'autres techniques d'extension, comme par exemple, l'extension par abstraction d'une ou de plusieurs autres solutions de patrons déjà existants. Elle consiste donc à produire des patrons qui répondent à des problèmes dont les contextes sont plus généraux. Dans le cadre de nos travaux, nous adoptons plutôt cette deuxième alternative, pour définir nos patrons Aspect.

3. Application et limites des patrons de conception par objets

La section 2 nous a permis d'introduire le concept de patron en général, de présenter les principales opérations d'ingénierie des SI à base de patrons, et d'expliquer quelques unes des techniques utilisées dans leur production. Dans nos travaux de thèse, nous nous intéressons plus particulièrement aux patrons de conception introduits dans le cadre de l'approche Objet. Très utiles, ces patrons présentent en revanche des limites et des problèmes liés à leur utilisation, plus particulièrement lors de leur imitation et implémentation. Les patrons ont, en effet, le défaut de « disparaître » dans les conceptions et les implémentations des applications qui les utilisent. Il devient ainsi difficile de retracer et d'isoler l'imitation d'un patron, afin de la maintenir et de la faire évoluer sans perdre l'« essence » du patron. Nous considérons dans ce qui suit un cas d'étude pour illustrer ces problèmes et limites, après avoir décrit ce type de patrons.

3.1 Patrons de conception par objets

Historique. Le terme « patron de conception » (*design pattern* en anglais) a été introduit par C. Alexander *et al.* [Alexander *et al.*, 77], [Alexander 79] dans le domaine de l'architecture des bâtiments, en réaction aux idées en vogue des années 70. L'idée d'appliquer ce concept dans le cadre de développement des

applications à base d'objets, est proposée par K. Beck et W. Cunninghan [Beck *et al.*, 87] lors de la conférence OOPSLA de 1987. Cette idée prend forme, un peu plus tard, lorsque K. Beck rencontre E. Gamma à ECOOP'90. Ce dernier avait remarqué l'existence de certaines structures récurrentes, lors du développement du cadre d'application ET++ [Weinand *et al.*, 88], [Weinand *et al.*, 89] durant son doctorat. Un ensemble de patrons plus spécifiquement dédiés à la conception à base d'objets a ainsi été élaboré, et identifié ensuite par E. Gamma en 1991, lors de ses travaux de thèse [Gamma 91]. Deux ans après, E. Gamma, R. Helm, R. Johnson et J. Vlissides présentaient la première esquisse d'un catalogue de patrons de conception par objets (*object-oriented design patterns)* [Gamma *et al.*, 93], dans ECOOP'93 à Kaiserslautern (Allemagne). Le catalogue complet [Gamma *et al.*, 95] a été publié deux ans plus tard. Il comporte 23 patrons de conception, extraits de plusieurs cadres d'applications, comme ET++, Graphics [Vlissides *et al.*, 88], Interviews [Linton *et al.*, 89], MacApp [Apple, 89], Unidraw [Vlissides et *al.*, 90], ObjectWorks [ParcPlace, 90], HotDraw [Johnson 92], RTL [Johnson *et al.*, 92], NextStep [Next 94] et ObjectWindows [Borland 94]. Dans un but d'intégrer et d'améliorer la réutilisation au niveau de la phase de conception, les auteurs de ce catalogue ne proposaient pas une nouvelle approche de développement comme le faisaient C. Alexander *et al.* en architecture. Il s'agit plutôt d'une nouvelle forme complétant les approches classiques de développement à base d'objets. Ce catalogue est désormais convaincant ; la communauté de recherche en génie logiciel a très bien accepté l'ensemble des 23 patrons ainsi proposés. Ils ont été, depuis, utilisés et intégrés à plusieurs méthodes de développement à base d'objets. Plusieurs publications, comme [Coplien *et al.*, 95], [Pree 95], [Vlissides *et al.*, 96], [Buschmann 96], [Buschmann *et al.*, 96], [Martin *et al.*, 98], [Larman 02] présentent de nouveaux patrons de conception par objets. Des patrons d'implémentation [Beck 96], [Beck 97] et des patrons d'analyse [Coad *et al.*, 95], [Fowler 97] spécifiques à un niveau d'ingénierie ou à un domaine d'application à base d'objets ont vu également le jour. Nous parlons ainsi de « *software patterns* » en opposition à « *design patterns* », pour désigner les patrons de développement à base d'objets utilisés durant le processus de développement d'applications. Désormais, les efforts actuels en terme de production de patrons concernent essentiellement le développement à base d'objets.

Définition. Un patron de conception par objets identifie un problème devant être résolu dans un contexte spécifique, et propose une solution possible et approuvée à ce problème. Une définition donnée par [Gamma *et al.*, 95] pour les patrons de conception est la suivante :

> *« Un patron de conception donne un nom, isole et identifie les principes fondamentaux d'une structure générale, pour en faire un moyen utile à l'élaboration d'une conception par objets réutilisables ».* [Gamma *et al.*, 95]

Identifier un patron de conception par objets par une étiquette unique offre un vocabulaire commun, dénotant l'expertise des concepteurs. Ces derniers peuvent alors décrire et discuter plus facilement leurs applications à un niveau abstrait, en dissimulant les détails de conception. Chaque patron considère un problème unique. Il décrit également le contexte de la solution qu'il propose à ce problème, mais aussi les forces et les conséquences qui résultent de son imitation et les compromis sous-entendus [Gamma *et al.*, 95]. La solution proposée par un patron est le plus souvent présentée avec son adéquation éventuelle à plusieurs autres contraintes d'implémentation ou de conception. En effet, les patrons de conception par objets sont généralement représentés par respect aux principes d'abstraction et de variabilité (cf. § 1.3.2). Ils permettent ainsi plusieurs réalisations possibles d'une même solution générique, et proposent les informations utiles et nécessaires à la définition de ces réalisations alternatives.

Utilité. Les patrons de conception par objets présentent plusieurs intérêts et sont largement utilisés dans l'ingénierie des SI, mais également par plusieurs autres types de composants tels que les frameworks et les composants logiciels, par exemple. En effet, les frameworks sont le plus souvent conçus à l'aide de patrons de conception par objets, qui facilitent la compréhension de leur conception et améliorent par conséquent leur maintenance et leur évolution. En ce sens Johnson [Johnson 92] propose, par exemple,

d'utiliser un ensemble de patrons de conception de Gamma [Gamma *et al.*, 95] tels que Composite, État, Observateur et Décorateur, pour documenter le framework d'éditeurs graphiques HotDraw. Les composants tels que CORBA (CCM) [CCM 02], EJB [EJB 03] et .Net [Net 05] utilisent également les patrons de conception dans leurs conceptions internes. D'ailleurs, les nouveaux systèmes se servent désormais, de plus en plus, de ces patrons pour gérer certains services non fonctionnels (tels que la persistance, la distribution ou encore la sécurité), et offrir la possibilité d'adapter ces services à plusieurs environnements d'utilisation. L'objectif ici est d'améliorer la réutilisation des systèmes, tout en séparant la partie fonctionnelle de la partie non fonctionnelle. Les patrons de conception sont aussi utilisés dans la composition de tels composants logiciels, permettant de créer ainsi des applications plus efficaces et réutilisables. En d'autres termes, il est nécessaire de coordonner la séparation des propriétés fonctionnelles et non fonctionnelles dès la phase de conception en utilisant les patrons de conception avant l'implémentation avec un langage adéquat Il convient ainsi de noter l'utilité de ces patrons de conception qui présentent, toutefois, quelques limites et problèmes que nous développons à travers le cas d'étude suivant.

3.2 Cas d'étude

Pour illustrer nos propos, considérons l'exemple d'un système simple que nous adaptons à partir de celui introduit dans [Tarr *et al.*, 00]. Par la suite nous appelons ce système SEE (*Software Engineering Environment*). L'objectif est de concevoir une application permettant de gérer des expressions dont la grammaire est donnée ci-dessous.

```
Expression := VariableExpression | NumberExpression | PlusOperator | MinusOperator | UnaryPlusOp
                | UnaryMinusOp
PlusOperator := Expression '+' Expression
MinusOperator := Expression '-' Expression
UnaryPlusOp := '+' Expression
UnaryMinusOp := '-' Expression
VariableExpression := ('A'| 'B' | 'C' | ... | 'Z') +
NumberExpression := ('0' | '1' | '2' | ... | '9') +
```

Les besoins fonctionnels attendus de cette application sont les suivants :

- Le SEE doit supporter la spécification et la représentation d'expressions de la forme « b+5 » ; des expressions basées principalement sur les opérateurs binaires et/ou unaires d'addition « + » et de soustraction « - », et sur des littéraux (variables ou nombres) possédant des valeurs numériques.
- Le SEE doit permettre en plus l'évaluation, l'affichage et la vérification syntaxique et sémantique d'expressions.
- Le SEE doit pouvoir offrir optionnellement une fonction de « logging » d'opérations (i.e. un journal d'exécution).

Sur la base des spécifications des besoins recensés ci-dessus, un diagramme de classes UML (cf. figure 1.6) est construit progressivement pour cette application. Il s'agit d'une solution possible, dont les différentes expressions sont représentées sous la forme d'arbres syntaxiques abstraits (*Abstract Syntax Trees* ou *AST*).

3.2.1 Application du patron *Composite*

Pour satisfaire le premier besoin, une structure des expressions du système SEE est définie par la figure 1.4. Une super-classe abstraite Expression est définie pour représenter d'une manière générale tous les types d'expressions possibles. Par ailleurs, une classe concrète est définie pour chaque type d'expression : VariableExpression, NumberExpression, UnaryPlusOp, UnaryMinusOp, PlusOperator et

MinusOperator. Les classes concrètes VariableExpression et NumberExpression héritent de la même classe abstraite Literal en regard de leurs propriétés communes. Pour la même raison, les classes concrètes UnaryPlusOp et UnaryMinusOp héritent de la classe abstraite UnaryOperator. Enfin, les classes PlusOperator et MinusOperator héritent de la classe abstraite BinaryOperator à cause de leurs propriétés communes.

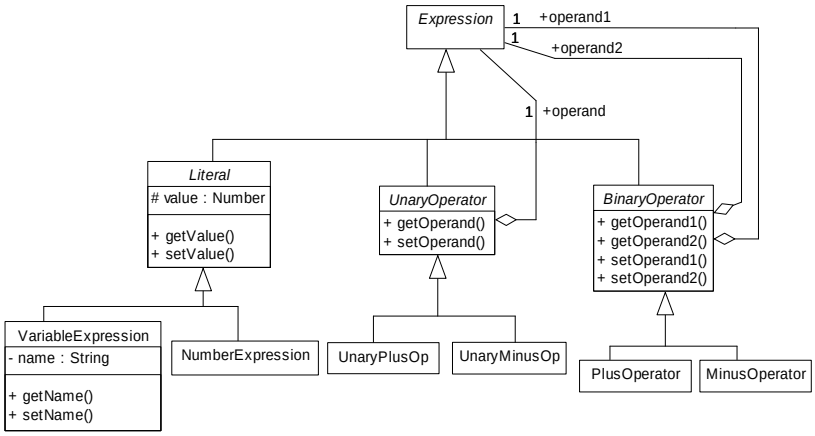
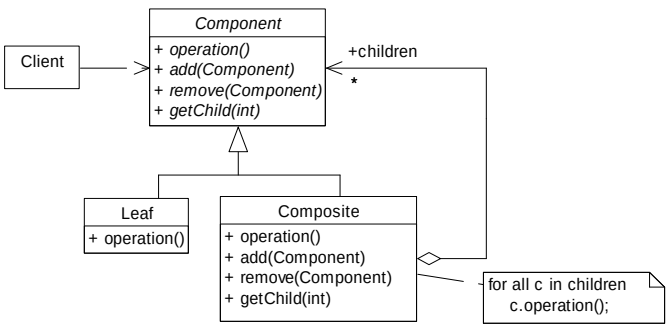


Figure 1.4. *Application du patron* Composite *sur le système SEE*

Comme le montre cette figure, la structure d'arbres AST peut être supportée dans le diagramme par le patron *Composite* dont une partie de la spécification est donnée ci-dessous (cf. tableau 1.6). En effet, on souhaite que le client n'ait pas à se préoccuper de la différence entre combinaisons d'objets AST individuels, et qu'il puisse traiter de façon uniforme tous les objets AST quels que soit leurs types.

Tableau 1.6. *Patron* Composite *du GoF [Gamma* et al.*, 95]*

Patron Composite [Gamma *et al.*, 95]
Intention. Le patron *Composite* propose de composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci.
Indications d'utilisation. On utilisera le *Composite* dans les situations suivantes. - On souhaite représenter des hiérarchies de l'individu à l'ensemble. - On souhaite que le client n'ait pas à se préoccuper de la différence entre objets composés et objets individuels. Les clients pourront traiter de façon uniforme tous les objets d'une structure.
Structure. 
Participants. - Component. Déclare l'interface des objets entrant dans la composition, implémente le comportement par défaut qui convient pour l'interface commune à toutes les classes, etc.

- Leaf. Représente des objets feuille dans la composition. Une feuille n'a pas d'enfants. Elle définit le comportement d'objets primitifs dans la composition.
- Composite. Définit le comportement des composants dotés d'enfants. Il stocke les composants enfants et implémente les opérations liées aux enfants dans l'interface Component.
- Client. Manipule les objets de la composition à l'aide de l'interface composant.

Dans cette utilisation spécifique du patron *Composite* deux types de feuilles (Leaf) sont spécifiés (VariableExpression et NumberExpression) au travers d'une super-classe Literal. Quatre types de composites (Composite) sont aussi distingués (UnaryPlusOp/UnaryMinusOp et PlusOperator/MinusOperator) regroupés respectivement selon deux super-classes (UnaryOperator et BinaryOperator). Il est à noter enfin que le patron *Composite* est appliqué trois fois sur le système SEE.

3.2.2 Une première solution problématique

En ce qui concerne le deuxième besoin fonctionnel (i.e. évaluation, affichage et vérification), plusieurs solutions de conception sont envisageables. Dans la solution que nous proposons, la classe abstraite Expression définit une opération pour chacune des trois fonctionnalités requises. Les sous-classes AST redéfinissent ces opérations en offrant les méthodes appropriées, respectivement, à l'évaluation (evaluate()), à l'affichage (display()) et à la vérification syntaxique et sémantique (check()) de l'expression en question, comme le montre la figure 1.5 ci-dessous.

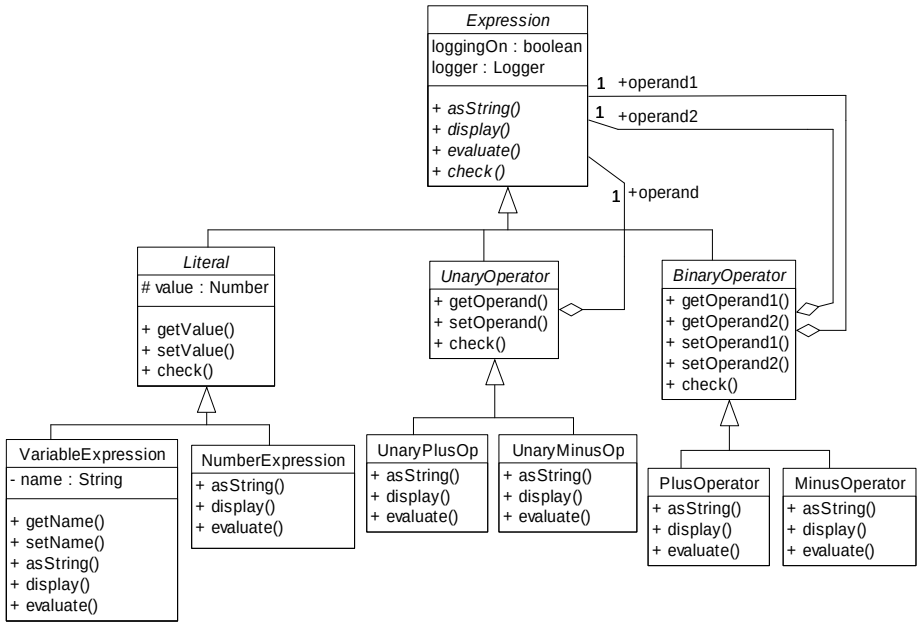


Figure 1.5. *Diagramme de classes détaillé du système SEE*

Enfin, quant à la fonction de « logging », elle est modélisée séparément par les deux classes Logger et LogFile (cf. figure 1.6). L'objectif est que pour chaque méthode invoquée sur un AST donné, la méthode Logger.beforeInvoke() est exécutée avant d'exécuter cette dernière. De même, la méthode Logger.afterInvoke() est exécutée juste avant que la méthode invoquée termine et retourne son résultat. La classe Logger permet ainsi à l'application d'enregistrer dans une instance de la classe LogFile tout ce qui se passe au moment de son exécution. La classe Logger permet, en plus, à l'application d'activer optionnellement cette fonctionnalité de « loggin », à n'importe quel moment au cours de son exécution, grâce à la méthode Logger.turnLoggingOn(). Elle lui permet de désactiver cette fonctionnalité à n'importe quel moment grâce à la méthode Logger.turnLoggingOff().

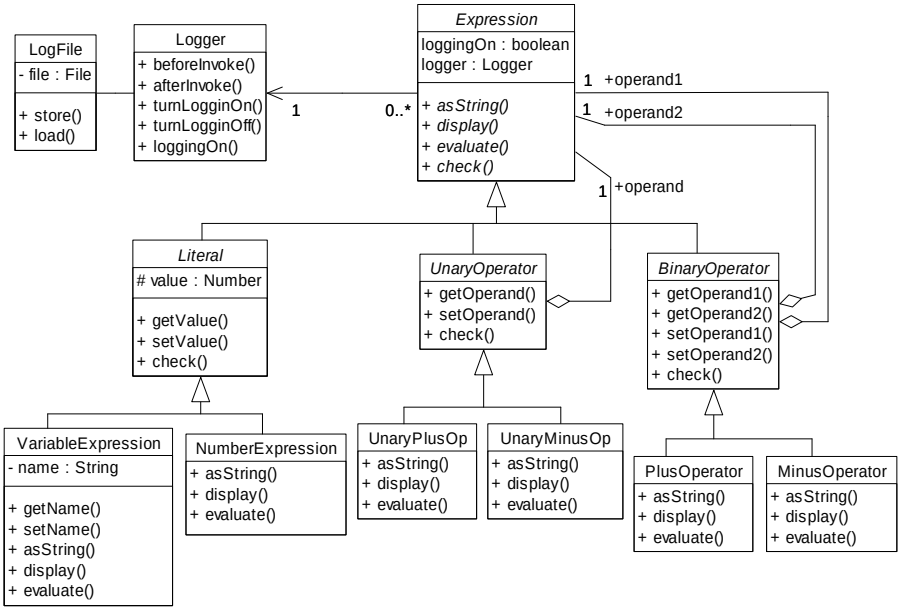


Figure 1.6. *Diagramme de classes complété du système SEE*

Ce troisième diagramme de classes (cf. figure 1.6) correspond à une conception possible de l'application SEE dans une approche strictement Objet. Cette conception présente dès lors quelques problèmes et limites préjudiciables à la maintenance, à l'évolution et à la réutilisation de cette application :

- la transformation de quelques besoins fonctionnels (i.e. les fonctionnalités d'évaluation, d'affichage et de vérification) jusqu'à la phase de conception ne garantit pas un développement « sans couture ». En effet, bien identifiées et isolées au niveau de la phase d'analyse, ces fonctionnalités se retrouvent dispersées sur l'ensemble des classes de l'application, où chaque classe contient une méthode dédiée à chacune de ces fonctionnalités. Maintenir et faire évoluer l'une de ces fonctionnalités devient ainsi une tâche complexe, puisque plusieurs classes doivent être modifiées. De même, ajouter une nouvelle opération applicable à cette structure d'expressions est une tâche assez difficile.
- encapsulée en partie dans la classe Logger, la fonctionnalité de « logging » est pour ainsi dire isolée aussi bien au niveau de la phase d'analyse qu'au niveau de la phase de conception. Toutefois, c'est le rôle des différentes méthodes des classes d'expressions que de coopérer avec la classe Logger pour réaliser complètement cette fonctionnalité ; c'est le cas en particulier des méthodes evaluate() et check(). Le code de cette fonctionnalité se retrouve ainsi dispersé. Il est non explicitement localisable et donc difficile à maintenir et faire évoluer. Les appels aux méthodes Logger.beforeInvoke() et Logger.afterInvoke() sont enchevêtrés avec le reste du code des classes d'expressions. Celui-ci se trouve donc moins lisible, difficilement compréhensible et moins réutilisable dans d'autres environnements ou contextes d'application. Les classes d'expressions et la classe Logger sont étroitement liées, elles sont non réutilisables indépendamment.

A présent, si l'on considère qu'il ne doit y avoir qu'une seule instance de Logger et une seule instance de LogFile dans l'application, ce qui est plus logique, cette conception ne le garantit pas. Nous montrons dans ce qui suit comment nous pouvons faire évoluer cette solution pour prendre en compte ce nouveau besoin. Nous montrons également comment nous pouvons l'améliorer pour une meilleure modularisation et séparation des besoins d'évaluation, d'affichage, de vérification, et de « logging », depuis la phase de conception jusqu'à la phase d'implémentation. Nous nous basons pour ce faire sur quatre patrons de [Gamma *et al.*, 95], tout en montrant leur utilité : *Visiteur*, *Observateur*, *Décorateur* et *Singleton*.

3.2.3 Application du patron *Visiteur*

Le patron *Visiteur* (cf. tableau 1.7) permet d'ajouter une ou plusieurs opérations applicables aux éléments d'une hiérarchie de classes existantes, de façon non « destructive », en les plaçant dans une autre hiérarchie. Les classes ConcreteVisitor1 et ConcreteVisitor2 correspondent chacune à un comportement ajouté à chacune des sous-classes de la classe abstraite Element. Chaque comportement est activé par appel de l'opération accept(). Dans chaque classe jouant le rôle de visiteur concret, le comportement correspondant peut admettre une définition différente pour chaque sous-classe de Element.

Tableau 1.7. *Patron* Visiteur *du GoF [Gamma* et al.*, 95]*

Patron *Visiteur* [Gamma *et al.*, 95].

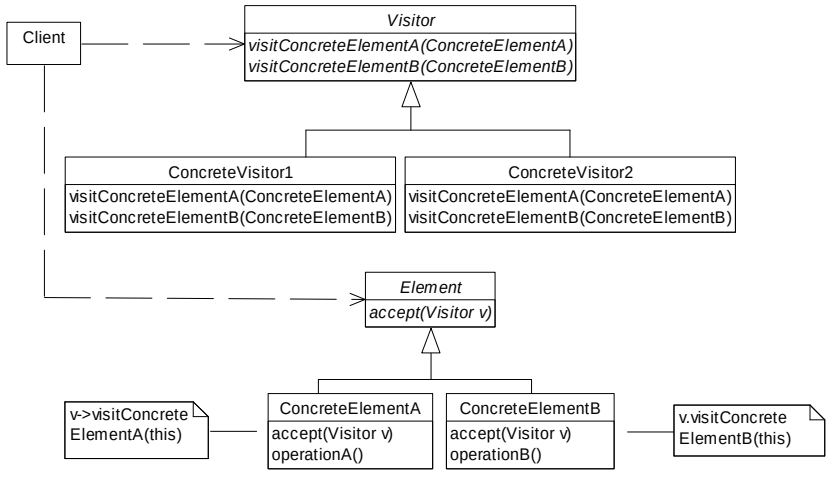
Intention. Visiteur réalise la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

Indications d'utilisation.

On utilisera le modèle *Visiteur* dans les circonstances suivantes.

- Une structure d'objets contient beaucoup de classes différentes d'interfaces distinctes, et l'on veut réaliser des opérations sur ces objets qui dépendent de leurs classes.
- Il s'agit d'effectuer plusieurs opérations distinctes et sans relation entre elles, sur les objets d'une structure, et ceci en évitant de polluer leurs classes avec des opérations. Le visiteur permet de regrouper toutes les opérations du même type dans une seule classe, quand la structure est partagée par plusieurs applications qui seules en ont besoin.
- Les classes qui définissent la structure d'objets changent rarement, mais on doit souvent définir de nouvelles opérations sur cette structure. Modifier les classes de la structure impose de redéfinir l'interface avec tous les visiteurs, ce qui peut être coûteux. Si les classes de la structure d'objets changent souvent, il est sans aucun doute préférable de définir les opérations dans des classes.

Structure.



Participants.

- Visitor. Déclare une opération visitConcreteElementX() pour chaque classe élément concret ConcreteElementX. Le nom de l'opération et sa signature identifient la classe émettrice de la requête vers le visiteur, qui peut ensuite accéder à l'élément directement, au travers de son interface particulière.
- ConcreteVisitor. Concrétise par codage chaque opération déclarée par la classe Visitor.
- Element. Définit une opération accept() qui prend pour argument un visiteur.
- ConcreteElement. Réalise le codage d'une opération accept().

Pour pallier au problème de dispersion du code des fonctionnalités d'évaluation, d'affichage et de validation, il est nécessaire d'isoler et d'encapsuler chacune de ces fonctionnalités. Pour répondre à ce besoin, nous nous servons du patron *Visiteur* souvent utilisé pour appliquer une opération sur une structure d'objets définie par le modèle *Composite*. Il s'agit de définir trois opérations applicables à la hiérarchie de classes modélisant les différents AST (cf. figure 1.4). Pour ce faire, il suffit tout simplement de créer trois visiteurs concrets correspondant chacun à une fonctionnalité donnée et de définir un visiteur abstrait. Chaque visiteur concret contient autant de méthodes correspondant à l'opération qu'il encapsule, que de classes concrètes représentant les différents AST. Ces méthodes seront ensuite invoquées et appliquées sur les classes des AST par l'unique intermédiaire de l'opération accept() devant être ajoutée à chacune de ces dernières. Les méthodes accept() prennent en paramètres les visiteurs concrets, auxquels elles délèguent les requêtes en invoquant les méthodes appropriées aux classes AST. La figure 1.7 montre le résultat d'application du patron *Visiteur* sur le système SEE. Il est à remarquer que toutes les définitions (i.e méthodes) propres à une fonctionnalité donnée sont isolées et regroupées dans une seule classe. Le code correspondant est moins dispersé (à l'exception de l'opération accept() augmentant la définition des classes d'expressions) et plus lisible. Il est plus facile ainsi de maintenir, de faire évoluer et de réutiliser ces fonctionnalités. D'ailleurs, le code des classes d'expressions devient plus propre, et l'enchevêtrement de leur code avec celui correspondant aux trois fonctionnalités est réduit à l'opération accept(). Le patron *Visiteur* permet ainsi de rajouter de nouvelles opérations sur la hiérarchie des AST, en ajoutant tout simplement un nouveau visiteur concret, et sans modifier les classes d'expressions. De plus, il augmente la traçabilité de chacune de ces opérations et facilite leur évolution.

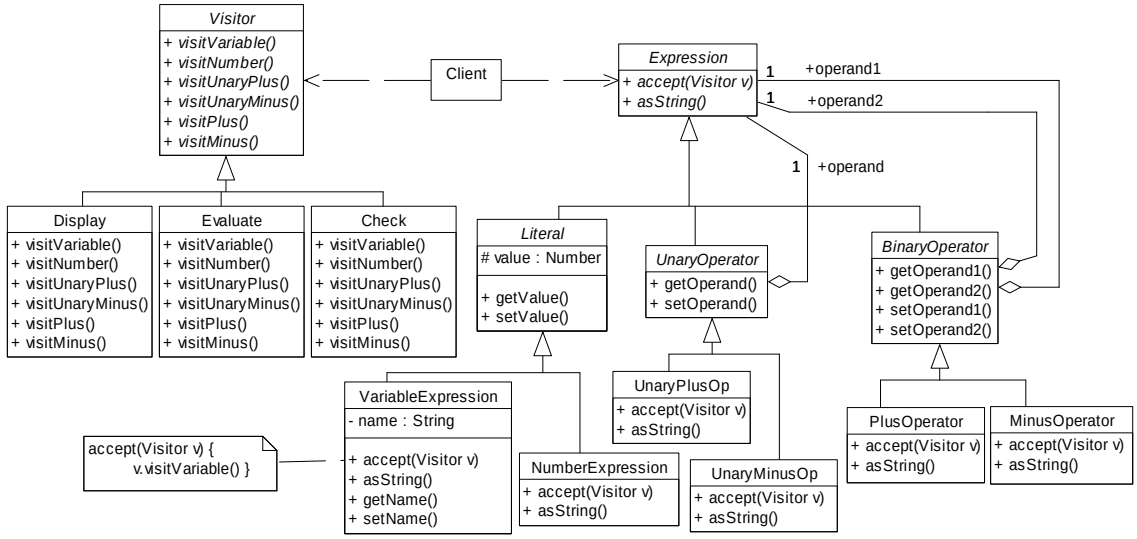


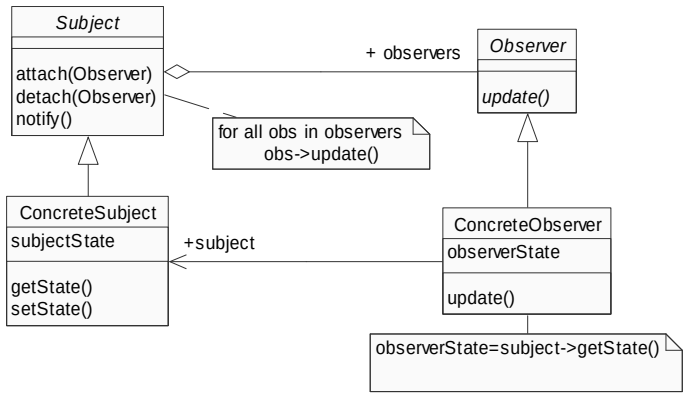
Figure 1.7. *Application du patron* Visiteur *sur le système SEE*

Toutefois, si nous désirons ajouter une nouvelle classe d'arbres à la structure d'expression (par exemple, l'expression de multiplication), le modèle *Visiteur* est mal adapté, il rend difficile l'évolution de cette application. En effet, tout ajout d'une nouvelle classe concrète d'expressions, impose l'introduction d'une nouvelle opération abstraite dans la classe visiteur abstrait, et l'introduction des différentes méthodes qui l'implémentent dans l'ensemble des classes visiteurs concrets. La hiérarchie des classes visiteurs peut être ainsi difficile à maintenir si l'on ajoute fréquemment de nouvelles classes d'arbres. Dans de telles situations, il est probablement plus facile de laisser éclater, dans les classes qui composent la structure d'expressions, la définition des différentes fonctionnalités. Voila pourquoi il est primordial de déterminer, avant d'appliquer ce modèle, si les fonctionnalités à appliquer à la structure d'objets ont une plus grande probabilité d'évoluer et de changer que les classes des objets qui composent cette structure.

3.2.4 Application du patron *Observateur*

Le patron *Observateur* (cf. tableau 1.8) permet d'établir une interdépendance de type un à plusieurs, afin que le changement d'état d'un objet soit notifié aux objets qui en dépendent et qu'ils soient automatiquement mis à jour. Ce patron est l'un des plus connus et des plus utilisés, il est intégré dans le noyau de langages tels que Smalltalk [Goldberg *et al.*, 83] et Java par exemple. Un observateur maintient une référence vers le sujet observé, les attributs nécessaires à la représentation de l'état de ce sujet et l'opération de mise à jour de cette représentation. Un sujet maintient des références vers plusieurs observateurs, et assume la responsabilité de notifier à ceux-ci tout changement de son état.

Tableau 1.8. *Patron* Observateur *du GoF [Gamma* et al.*, 95]*

Patron *Observateur* [Gamma *et al.*, 95].
Intention. *Observateur* définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.
Indications d'utilisation. On utilisera le patron *Observateur* dans les situations suivantes : - Quand un concept a deux représentations, l'une dépendante de l'autre. - Quand la modification d'un objet nécessite de modifier les autres, et que l'on ne sait pas combien ils sont. - Quand un objet doit être capable de faire une notification à d'autres objets sans faire d'hypothèse sur la nature de ces objets.
Structure.    **Participant**. - Subject. Connaît ses observateurs. Un nombre quelconque d'observateurs peut observer un sujet. Il fournit une interface pour attacher et détacher les objets observateurs. - Observer. Définit une interface de mise à jour pour les objets qui doivent être notifiés de changements dans un sujet. - ConcreteSubject. Mémorise les états qui intéressent les objets ConcreteObserver. Il envoie une notification à ses observateurs lorsqu'il change d'état. - ConcreteObserver. Gère une référence sur un objet ConcreteSubject et mémorise l'état qui doit rester pertinent pour le sujet. Il fait l'implantation de l'interface de mise à jour de l'observateur pour conserver la cohérence de son état avec le sujet.

Pour réduire le couplage et supprimer le lien direct existant entre les classes d'expressions et la classe Logger, et augmenter leurs réutilisations indépendantes, nous pouvons utiliser le patron *Observateur*. Les classes d'arbres joueront le rôle de ConcreteSubject et la classe Logger le rôle de ConcreteObserver. Le lien entre les classes représentant les différentes AST et le « logger » est assuré à un niveau d'abstraction plus élevé, entre les classes abstraites Expression (jouant le rôle de Subject) et la classe Observer (cf. figure 1.8).

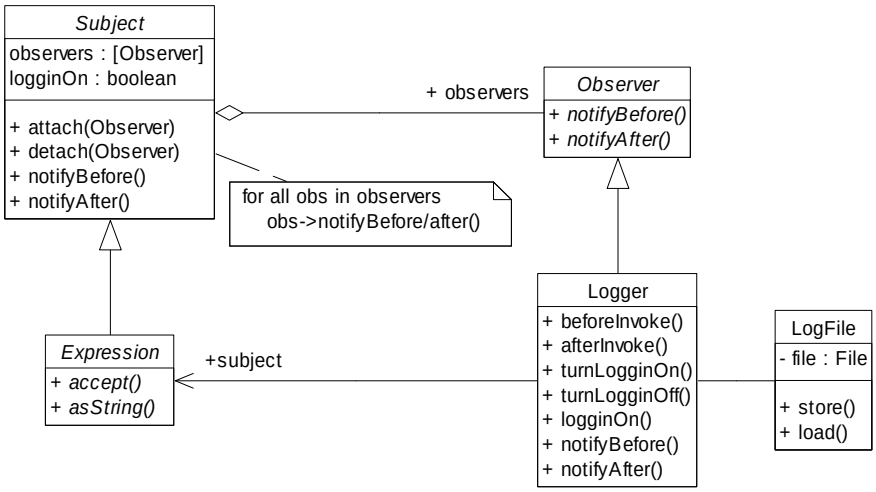


Figure 1.8. *Application du patron* Observateur *sur le système SEE*

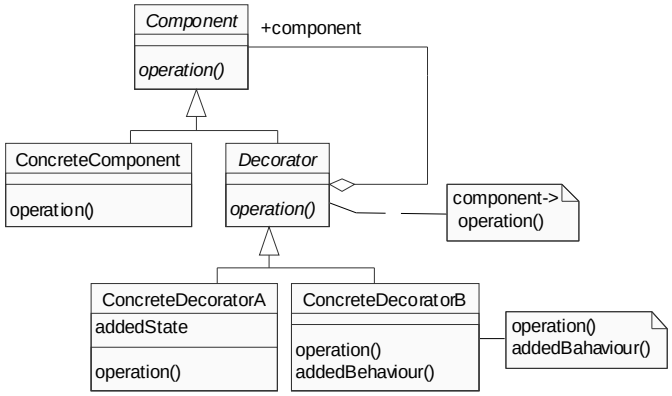
Le patron *Observateur* permet entre autre de diminuer relativement l'enchevêtrement du code spécifique à la fonctionnalité de « logging » avec celui des classes d'expressions. En effet, une grande partie du protocole de collaboration entre ces classes et la classe Logger est spécifiée au niveau des classes abstraites jouant le rôle de Subject et d'Observer, à travers les couples de méthodes notifyBefore() et notifyAfter(). L'enchevêtrement se réduit ainsi aux appels de ces deux méthodes. En effet, pour mettre en œuvre la fonctionnalité de « logging » avec le modèle observateur, il s'agit d'invoquer respectivement ces deux dernières au début et à la fin de l'exécution de chacune des méthodes des classes AST, dont l'exécution est destinée à être enregistrée dans le journal d'exécution. Chacune de ces deux méthodes fait appel, à son tour, à la méthode notifyBefore() ou notifyAfter() de l'instance « logger », invoquant respectivement les méthodes beforeInvoke() ou afterInvoke() de la classe Logger.

En conclusion, bien qu'il diminue le couplage entre les classes d'expressions et la classe Logger, *Observateur* n'élimine pas complètement l'enchevêtrement du code relatif à la fonctionnalité du « logging » avec celui correspondant aux classes d'expressions. Leurs méthodes font toujours appel explicitement aux méthodes notifyBefore() et notifyAfter() spécifique à cette fonctionnalité. D'ailleurs, utilisée conjointement avec le modèle *Visiteur*, l'application du patron *Observateur* devient très complexe, elle diminue la lisibilité et la compréhension du code et nuit par conséquent à l'évolution et à la réutilisation de celui-ci. En effet, chacune des méthodes définies dans les classes visiteurs concrets doit faire appel aux deux méthodes de notification, ce qui augmente d'avantage l'enchevêtrement du code de cette fonctionnalité. Pour remédier à ce problème d'enchevêtrement, une alternative possible à ce patron est le patron Décorateur. Par ailleurs, *Observateur* ne garantit pas la séparation et l'encapsulation de cette fonctionnalité au niveau de la conception, ni au niveau de l'implémentation. Sa définition est encore dispersée dans l'ensemble des classes de l'application.

3.2.5 Application du patron *Décorateur*

Le patron *Décorateur* permet d'étendre les fonctionnalités d'un objet, tout en lui attachant dynamiquement des responsabilités supplémentaires. Il fournit une alternative souple à la dérivation en sous-classes, pour étendre le comportement d'un type donné. Une partie de la spécification de ce patron est donnée dans ce qui suit (cf. tableau 1.9). Le décorateur transmet les requêtes à son objet composant. Il peut éventuellement effectuer des opérations supplémentaires avant et après la transmission de la requête.

Tableau 1.9. *Patron* Décorateur *du GoF [Gamma* et al.*, 95]*

Patron *Décorateur* [Gamma *et al.*, 95].
Intention. *Décorateur* attache dynamiquement des responsabilités supplémentaires à un objet. Les décorateurs fournissent une alternative souple à la dérivation pour étendre les fonctionnalités.
Indications d'utilisation. On utilisera le modèle *Décorateur* : - pour ajouter dynamiquement des responsabilités à des objets individuels, ceci d'une façon transparente, c'est-à-dire, sans affecter les autres objets, - pour rajouter des responsabilités qui peuvent être retirées, - quand l'extension par dérivation est impraticable. Il peut arriver parfois que l'on ait un grand nombre d'extensions indépendantes possibles. Il en résulte une prolifération explosive de sous-classes pour permettre chaque combinaison. D'autres fois, la définition de classe pourra être cachée, ou encore inaccessible pour la dérivation.
Structure.    **Participants**. - Component. Définit l'interface des objets qui peuvent recevoir dynamiquement des responsabilités supplémentaires. - ConcreteComponent. Définit un objet auquel des responsabilités peuvent être adjointes. - Decorator. Gère une référence à un objet Component et définit une interface conforme à celle du Component. - ConcreteDecorator. Ajoute des responsabilités à un composant.

Le patron *Décorateur* constitue une alternative au patron *Observateur* permettant d'éliminer le couplage entre les classes d'expressions et la classe Logger. Il permet, par rapport à l'Observateur, de pallier aux problèmes d'enchevêtrement et de dispersion du code de la fonction de « logging ». La figure 1.9 montre le résultat d'application de ce patron. Il est à remarquer que la fonction de « logging » est bien séparée et encapsulée dans une classe à part (i.e LoggingDecorator). Son code n'est plus dispersé dans l'ensemble des classes d'expressions. D'ailleurs, il n'est plus enchevêtré avec le reste du code de ces dernières. De plus, comme le montre cette figure, les classes d'expressions ne sont plus directement associées à la classe Logger : l'association est établie entre la classe LoggingDecorator et la classe Logger. Le code de l'application devient ainsi plus facile à comprendre et à faire évoluer et par conséquent plus réutilisable.

Cette nouvelle alternative doit garantir une notification de « logging » efficace et consistante. Toutefois, pour assurer que des appels successifs entre objets d'une expression composite soient augmentés à chaque notification de « logging », il faut savoir bien gérer les protocoles entre les décorateurs liés à ces objets. Ceci est une tâche assez complexe qui introduit une nouvelle forme de couplage : elle rend le code difficile à comprendre, à maintenir et à faire évoluer.

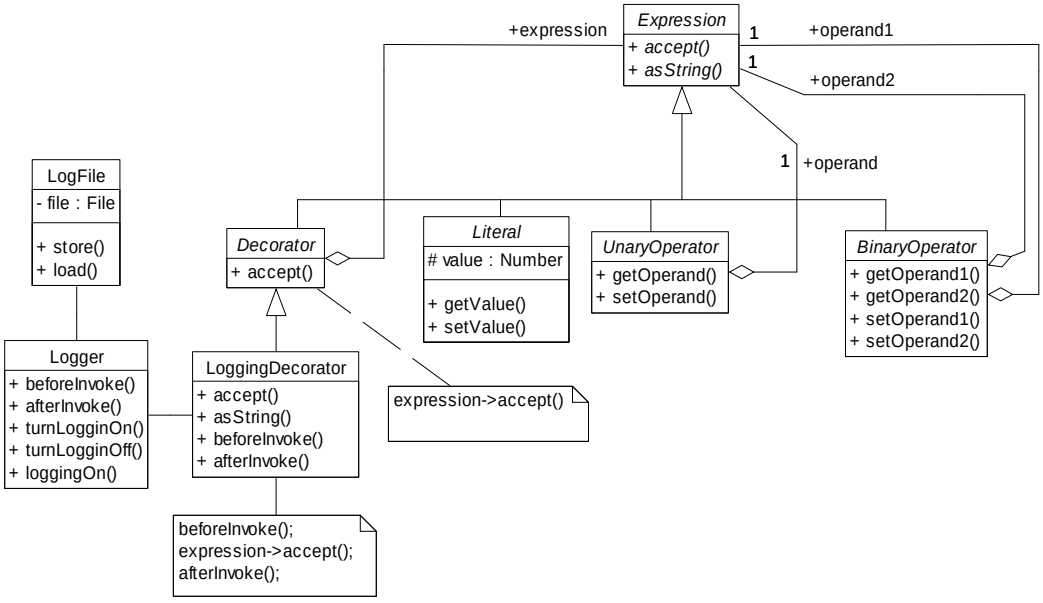


Figure 1.9. *Application du patron* Décorateur *sur le système SEE*

3.2.6 Application du patron *Singleton*

Le patron *Singleton* garantit qu'une classe n'a qu'une seule instance. Une partie de la spécification de ce patron est donnée ci-dessous (cf. tableau 1.10).

Tableau 1.10. *Patron* Singleton *du GoF [Gamma* et al.*, 95]*

Patron *Singleton* [Gamma *et al.*, 95].
Intention. *Singleton* garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe.
Indications d'utilisation. On utilisera le modèle *Singleton* : - s'il doit n'y avoir exactement qu'une instance d'une classe, qui, de plus, doit être accessible aux clients en un point bien déterminé, - si l'instance unique doit être accessible par dérivation en sous-classes, et si l'utilisation d'une instance étendue doit être permise aux clients, sans qu'ils aient à modifier leur code.
Structure. Singleton - instance : Singleton - Singleton() + instance() : Singleton **Participants**. Singleton. Définit une opération instance() qui donne au client l'accès à son unique instance. Il s'agit d'une opération de classe. Singleton peut avoir la charge de créer sa propre instance unique.

Pour répondre aux nouveaux besoins et garantir une instance unique de Logger, respectivement de LogFile dans l'application SEE, nous utilisons le patron *Singleton*. Il s'agit de rendre privés les constructeurs de ces deux classes et d'y ajouter une nouvelle opération publique : l'opération instance() qui donne au client l'accès, respectivement, à l'unique instance de Logger et à l'unique instance de LogFile. La figure 1.10 montre le résultat d'application de ce patron sur ces deux classes.

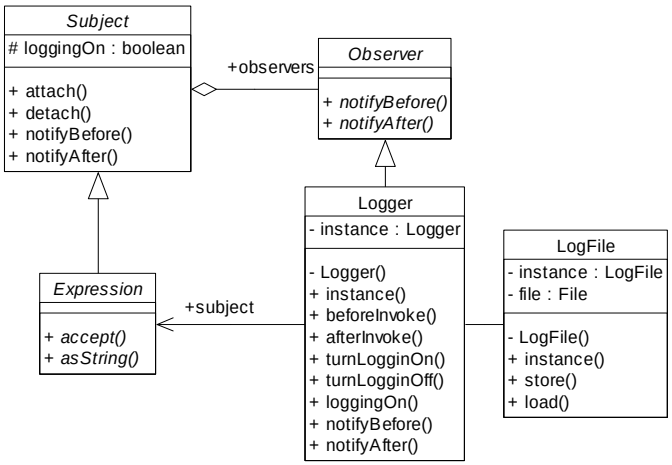


Figure 1.10. *Application du patron* Singleton *sur le système SEE*

3.2.7 Une meilleure solution utilisant les patrons de conception par objets

La Figure 1.11 montre le résultat final d'application des patrons *Composite*, *Visiteur*, *Observateur* et *Singleton* sur le système SEE.

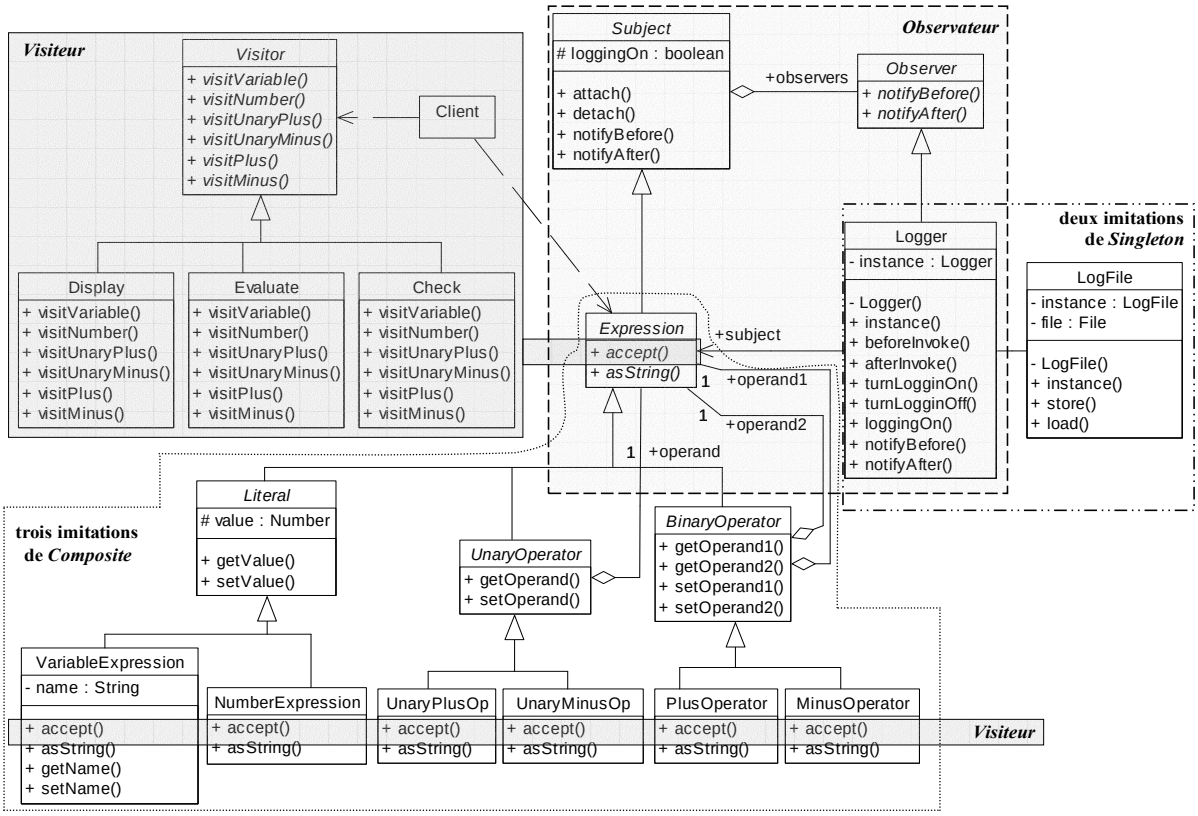


Figure 1.11. *Résultat d'application des patrons* Composite*,* Visiteur*,* Observateur *et* Singleton *sur le système SEE*

Comme le montre cet exemple d'étude, les patrons de conception par objets sont d'une grande utilité pour structurer la solution. Ils permettent d'améliorer et de pallier certains (mais pas tous) problèmes de conception pouvant nuire à l'évolution, à la maintenance, ou encore à la réutilisation des applications. Toutefois, ils introduisent à leur tour plusieurs autres problèmes que nous identifions dans ce qui suit.

3.3 Problèmes et limites d'utilisation des patrons de conception par objets

En améliorant les solutions à quelques problèmes, l'utilisation des patrons peut affecter la conception et donc l'implémentation des applications, tout en introduisant d'autres problèmes dont quelques-uns sont déjà reconnus et relevés dans [Gamma *et al.*, 95]. En effet, basés exclusivement sur l'*héritage*, la *composition*, ou encore la *délégation explicite*, les patrons imposent une structure aux conceptions et aux programmes qui soulève plusieurs problèmes et limites liés directement à la *dispersion* et à *l'enchevêtrement* des éléments conceptuels de ces patrons et de leurs codes. Nous développons dans ce qui suit ces problèmes, et nous discutons les limites liées à l'utilisation des patrons. Nous proposons également de faire une synthèse des problèmes soulevés, tout en considérant l'ensemble des patrons du GoF [Gamma *et al.*, 95]. Ce catalogue a été choisi en raison de sa notoriété, mais aussi en raison de la large utilisation des patrons qu'il propose et leur assez grande aptitude pour résoudre des problèmes récurrents et complexes de conception détaillée.

3.3.1 Problèmes liés à l'héritage

Pour répartir les comportements entre les différentes classes participantes d'un patron (i.e. attribuer différents rôles aux classes membres), de nombreux patrons de conception utilisent l'héritage. Cela renforce l'interdépendance du code des classes de l'application et de celui des patrons, et nuit à leur réutilisation séparée. De plus, si le langage de programmation n'intègre pas l'héritage multiple, ou un héritage multiple limité à l'héritage d'interfaces, tel que Java, l'application d'un patron sur une classe possédant déjà une superclasse est difficile.

Nous illustrons ce problème sur le patron *Observateur* (cf. § 3.2.4) bien que sa mise en œuvre soit grandement facilitée dans de nombreux langages de programmation par objets, tels que Smalltalk ou Java, dans lesquels il se trouve directement intégré dans le noyau. Ce problème se pose néanmoins pour plusieurs autres patrons du GoF (cf. § 3.3.6) non présentés explicitement dans ce chapitre. L'utilisation du patron Observateur impose en fait, pour toute classe appelée à jouer le rôle d'observateur concret (respectivement de sujet concret), d'avoir pour super-classe Observer (respectivement Subject), et interdit de la sorte toute autre super-classe en l'absence d'héritage multiple. Toutefois, utiliser un langage avec héritage multiple conduit également à un autre problème : puisque des liens d'héritage peuvent être introduits lors de l'imitation d'un patron, il faut systématiquement reconsidérer chacun de ceux-ci dans le cas d'une réutilisation séparée d'une classe. Dans notre cas d'étude, les classes Expression et Logger ne possèdent pas de super-classes, la contrainte d'héritage multiple ne se pose pas. Cependant, l'application du patron Observateur à ces deux classes, crée une forme d'interdépendance entre les classes d'expressions (respectivement la classe Logger) et la classe abstraite Subject (respectivement la classe abstraite Observer), et nuit à leur réutilisation séparée dans d'autres contextes d'utilisation.

3.3.2 Problème de rupture d'encapsulation

L'utilisation conjointe de la délégation explicite et de la composition, afin de placer des comportements relatifs à certaines classes dans d'autres classes, pose un problème de rupture d'encapsulation. Par exemple, dans le cas de *Visiteur* (cf. § 3.2.3), les comportements relatifs aux éléments concrets sont déportés dans les classes visiteurs. Néanmoins, il est fort probable que les opérations visitConcreteElementX(), exécutées dans les contextes des classes visiteurs doivent accéder à des attributs (ou des opérations) définis dans les sous-classes de Element qui délèguent ces comportements. L'utilisation du patron *Visiteur* impose donc, dans certains[2] langages tels que Java, de rendre ces attributs publics (ou de leur définir des accesseurs

[2] Ce problème peut être atténué lors de l'impléméntation effective du patron de diverses manières suivant le langage de programmation utilisé. On peut, par exemple, en C++, utiliser le mécanisme de « relâchement de l'encapsulation par amitié » (friend).

publics en adoptant la *Loi de Demeter* [Lieberherr *et al.*, 89]) alors que l'on aurait pu souhaiter les déclarer privés ou protégés. Cela contrarie l'encapsulation, l'un des principes fondamentaux de l'approche Objet [Snyder 86], et augmente le couplage entre les différents éléments de l'application et les visiteurs concrets. Ce problème, également reconnu dans [Gamma *et al.*, 95], est similaire aux problèmes liés au mécanisme de la délégation implicite [Bardou *et al.*, 96]. Dans l'exemple d'étude, chacune des opérations spécifiques aux fonctionnalités d'évaluation, d'affichage et de vérification, placées dans les classes visiteurs concrets, doit accéder aux propriétés privées des sous-classes de Expression posant ainsi un problème de rupture d'encapsulation. Rappelons par ailleurs que pour ce faire ces opérations prennent en paramètre des instances de classes d'expressions, ce qui augmente le couplage entre ces dernières et les visiteurs concrets (Display, Evaluate et Check).

3.3.3 Problème d'indirection et de surcharge de l'implémentation

La délégation explicite (ou indirection de messages) est très largement utilisée dans les patrons de conception par objets. Elle permet de répartir les comportements entre les objets en interaction tout en réduisant le couplage entre les définitions de ceux-ci. Un inconvénient majeur de cette indirection est l'augmentation du nombre de messages entre objets qui se traduit par la diminution de la lisibilité du code, des problèmes d'efficacité et l'introduction de dépendances préjudiciables à l'évolution de l'application. Ce problème a été également relevé dans [Gamma *et al.*, 95] et dans [Bosch 96a]. Cette indirection est présente dans le patron *Décorateur* (cf. § 3.2.5) par exemple, où chaque instance d'un décorateur concret est associée à une instance d'une classe concrète de la structure d'objets composants, et l'appel de n'importe quelle opération d'une instance décorateur résulte en l'exécution de l'opération de l'objet composant (telle qu'elle est définie dans cette dernière instance). Dans notre cas d'étude, tout appel à une opération d'un décorateur d'expression résulte en l'appel de l'opération appropriée de l'instance d'expression liée à ce dernier. Cette indirection n'est ni complexe, ni difficile à mettre en œuvre. Toutefois, en grande quantité, les opérations des décorateurs surchargent l'implémentation et augmentent le nombre des messages échangés. Ces opérations diminuent, de plus, la lisibilité du code et introduisent des dépendances, qui nuisent à l'évolution et à la réutilisation de l'application.

3.3.4 Problème de confusion et de traçabilité

La plupart des patrons de conception par objets décrivent des collaborations entre plusieurs objets qui sont représentées d'une façon indirecte dans le code d'une application. Il en découle que leurs implémentations sont très généralement déclinées et éclatées dans plusieurs classes ; il est donc difficile de distinguer les parties de programme propres à l'imitation (cf. § 2.4.1) d'un patron du reste du code de l'application. Ce problème est reconnu aussi sous le nom de « *Problème de Traçabilité* » dans [Bosch 98]. J. Soukup [Soukup, 95] et J. Bosch [Bosch, 96] soulèvent d'ailleurs ce problème. Ils confirment que les programmeurs auraient tendance à perdre de vue les patrons après leur implémentation. Dans notre cas d'étude, en l'absence de documentation rigoureuse, il est très difficile, par exemple, de percevoir et de localiser toutes les imitations des patrons appliqués.

L'implémentation d'un patron n'étant pas isolée, elle est confondue avec le reste du code de l'application. D'ailleurs, l'imitation d'un patron nécessite une adaptation de classes déjà existantes et éventuellement l'ajout ou la suppression de nouvelles classes et de leurs propriétés. En procédant par modification des noms des participants formels du patron et de leurs propriétés, le code spécifique à une imitation d'un patron est facilement confondu avec le reste du code de l'application. De plus, un patron n'étant pas toujours implémenté de la même manière (tout patron admet plusieurs réalisations possibles, sa structure peut varier selon plusieurs compromis d'implémentation), il est difficile de retracer son imitation dans le code d'une application. Cette représentation indirecte d'un patron pose ainsi plusieurs problèmes de maintenance, réutilisation et évolution. L'implémentation de *Visiteur* (cf. § 3.2.3), par

exemple, est éclatée dans les classes de la hiérarchie des visiteurs et celles de la structure d'éléments. De plus, à l'exception de l'utilisation d'une convention de nommage, ou d'un autre artifice de documentation, rien ne distingue dans le code le statut particulier des opérations accept () parmi toutes les opérations définies dans les classes d'éléments. De même, rien ne dénote le fait que les classes visiteurs concrets détiennent des opérations pour le compte d'autres classes.

Dans notre cas d'étude, il est difficile par exemple de comprendre, dans l'imitation du patron *Visiteur*, la sémantique exacte des opérations d'évaluation, d'affichage ou encore de vérification (visitVariable(VariableExpression), visitNumber(NumberExpression), etc.) détenues par les classes visiteurs concrets. Par ailleurs, bien qu'il soit possible de déterminer que les méthodes accept() des classes d'expressions ne fassent qu'appeler ces opérations, leur identification comme composantes du patron *Visiteur* n'est pas explicite. Au-delà de ce problème de confusion, le code de l'imitation de ce patron est ainsi dispersé dans l'ensemble des classes, il est donc difficile de tracer son imitation dans l'application SEE.

3.3.5 Limites d'utilisation des patrons de conception par objets

Outre ces quatre problèmes d'utilisation des patrons de conception dans une approche strictement Objet, nous distinguons notamment deux limites. Ces limites sont principalement et directement liées à la dispersion et à l'enchevêtrement du code des imitations des patrons.

- **Code non réutilisable**. Le code d'une imitation d'un patron est dispersé et éclaté dans l'ensemble des classes qu'il impacte. Il est non isolé, et donc difficilement réutilisable indépendamment de l'application. En effet, de manière générale, les patrons de conception permettent la réutilisation de solutions conceptuelles et non pas d'implémentation. D'ailleurs, l'implémentation d'un patron est le plus souvent guidée par différents compromis, elle peut adopter différentes formes ou variations. Les variations les plus évidentes sont celles qu'induit le contexte d'utilisation sur les noms fictifs des participants du patron et les noms de leurs propriétés, sur les choix des structures de données (listes ou tableaux, etc. pour représenter les collections par exemple), et sur l'organisation même des classes participantes dans le patron. Toutes ces variations sont trop spécifiques à un problème donné, pour être réutilisables ailleurs [Soukup 95].
  Par ailleurs, l'enchevêtrement du code relatif aux imitations des patrons, avec celui des classes sur lesquelles il s'applique, nuit abondamment à la réutilisation de ces classes dans d'autres contextes applicatifs (indépendants des patrons imités). Dans le cas d'étude que nous considérons, si nous utilisons le patron *Observateur* pour une meilleure réalisation de la fonctionnalité de « logging », il devient difficile, en revanche, de directement réutiliser la hiérarchie d'expressions dans un autre contexte applicatif, qui ne nécessite pas la mise en œuvre de cette fonctionnalité. En effet, pour réutiliser ces classes, il faut tout d'abord comprendre tout le protocole de collaboration du patron *Observateur*, afin de pouvoir localiser ensuite le code relatif à celui-ci qui est enchevêtré avec le reste du code des classes d'expressions. Ce qui revient à enlever le lien d'héritage (l'héritage de la classe Subject), à supprimer les opérations redéfinies aux niveaux de la classe Expression (attach() et detach()) et aussi, à supprimer tous les appels à notifyBefore() et notifyAfter() enchevêtrés avec l'ensemble des opérations des classes d'expressions.
- **Maintenance et évolution contraignantes**. La dispersion et l'enchevêtrement du code des imitations des patrons pénalisent la maintenance et l'évolution de celui-ci, mais aussi du code relatif à l'application. En effet, étant dispersé et éclaté dans plusieurs classes de l'application, le code relatif à une imitation d'un patron est difficilement localisé. Il est par conséquent difficile à maintenir et à faire évoluer. D'ailleurs, l'enchevêtrement du code des imitations des patrons, avec le reste du code de l'application, résulte en un code non lisible et nuit par conséquent à la

maintenance et à l'évolution de toute l'application devant respecter les contraintes d'implémentation de ces imitations. En effet, comme le confirme G. Florijn [Florijn *et al.*, 97], l'implémentation d'un patron doit respecter certaines contraintes d'implémentations dictées le plus souvent par leurs spécifications. Toute modification d'une application, utilisant des patrons, doit ainsi prendre en compte et respecter toutes les contraintes de leur implémentation.

3.3.6 Synthèse

Nous proposons dans ce qui suit de faire la synthèse des quatre problèmes identifiés précédemment. En effet, bien qu'explicités sur quatre patrons de conception du GoF (*Visiteur*, *Observateur*, *Décorateur* et *Singleton*) que nous avons pris pour exemple dans ce chapitre, ces problèmes identifiés concernent la majorité des patrons du catalogue de Gamma [Gamma *et al.*, 95]. Nous avons pu observer que le problème de confusion et de traçabilité est présent dans la totalité des 23 patrons ; le problème d'indirection et de surcharge d'implémentation concerne 20 patrons dont 10 sont également touchés par le problème de rupture d'encapsulation. Les problèmes liés à l'héritage, quant à eux, n'affectent essentiellement que 7 patrons (*Adaptateur*, *Composite*, *Chaîne de responsabilité*, *Commande*, *Itérateur*, *Observateur* et *Patron de méthode*). Le tableau 1.11 fait la synthèse des problèmes présents dans chaque patron du catalogue.

Tableau 1.11. *Problèmes liés aux 23 patrons du GoF*

		Problème de confusion et de traçabilité	Problème d'indirection et de surcharge d'implémentation	Problèmes de rupture d'encapsulation	Problèmes liés à l'héritage
Créateur	*Fabrique Abstraite*	x	x		
	Monteur	x	x		
	Fabrication	x	x		
	Prototype	x			
	Singleton	x			
Structurel	*Adaptateur*	x	x	x	x
	Pont	x	x	x	
	Composite	x	x		x
	Décorateur	x	x	x	
	Façade[3]				
	Poids Mouche	x	x		
	Procuration	x	x		
Comportemental	*Chaîne de responsabilité*	x	x		x
	Commande	x	x		x
	Interpréteur	x	x		
	Itérateur	x	x	x	x
	Médiateur	x	x	x	
	Mémento	x	x	x	
	Observateur	x	x	x	x
	État	x	x	x	
	Stratégie	x	x	x	
	Patron de méthode	x			x
	Visiteur	x	x	x	

On remarque que ces quatre problèmes sont des déclinaisons des deux problèmes récurrents en programmation par objets : l'*enchevêtrement* et la *dispersion* du code [Hursh *et al.*, 95] :

[3] Le patron Façade propose de fournir une interface de plus haut niveau unifiée, à l'ensemble des interfaces d'un sous-système, afin de faciliter l'utilisation de ce dernier. La structure par objets de ce patron ne pose pas de problèmes ; ses imitations sont facilement localisables dans le code d'une application finale et les définitions des classes sur lesquelles s'applique ce patron sont inchangées.

- L'enchevêtrement du code relatif aux imitations des patrons dans les différentes classes conduit à la confusion, et nuit donc à la traçabilité des choix de conception.
- La dispersion du code du patron dans les différentes classes intervenantes est liée à l'utilisation de la délégation explicite et du mécanisme d'héritage dans la réalisation de ces patrons. Cette situation impose souvent de rompre l'encapsulation et de surcharger l'implémentation.

La dispersion du code de l'imitation d'un patron dans différentes classes compromet sa localisation (ou traçabilité) et diminue par conséquent sa maintenance, son évolution et sa réutilisation. D'autre part, l'enchevêtrement du code de l'imitation d'un patron dans les classes sur lesquelles il s'applique nuit à la lisibilité du code des classes, à la maintenance, à l'évolution et à la réutilisation de l'application indépendamment du patron imité.

3.4 Limites des solutions actuelles

Plusieurs travaux [Soukup 95], [Meijers 96], [Meijler *et al.*, 96], [Kim *et al.*, 95], [Budinsky *et al.*, 96], [Eden *et al.*, 97], [Bosch 96a], [Bosch 96b], [Conte *et al.*, 01b], [Conte *et al.*, 01c] ont d'ores et déjà proposé d'améliorer la traçabilité des patrons de conception par objets, depuis la phase de conception jusqu'à la phase d'implémentation. L'objectif est de garantir une meilleure évolution et réutilisation de ceux-ci, mais aussi des applications sur lesquelles ils s'appliquent. Ces travaux proposent d'automatiser l'application de ces patrons, et offrent divers outils adoptant différentes approches de développement. D'une manière générale, indépendamment de l'approche adoptée, la plupart de ces outils proposent, entre autres, d'assurer le lien entre l'imitation d'un patron au niveau conceptuel et son code dans l'implémentation de l'application. Tout en assurant un développement sans couture, ce lien permet ainsi de tracer le code d'un patron et d'améliorer l'évolution et la réutilisation. D'autres travaux, détaillés ci-dessous, s'intéressent à la réification des patrons sous la forme de classes.

- Jiri Soukup [Soukup 95] propose, par exemple, d'isoler et d'encapsuler l'essence et le protocole d'un patron sous la forme d'une classe, appelée *Pattern Class*. Toute classe de patron contient seulement le code d'implémentation de la solution générique de celui-ci, sans considérer les classes de l'application sur lesquelles il s'applique. Cette classe est ensuite considérée comme une classe amie des classes de l'application participant dans l'imitation du patron.
- M. Meijers [Meijers 96] propose un outil permettant d'intégrer les imitations de patrons de conception à du code Smalltalk. Il se base pour ce faire, sur un modèle en fragments qu'il définit pour représenter les instances de patrons. Dans ce modèle, chaque instance de patron est représentée par plusieurs fragments liés aux classes concrètes participants dans cette instance : chaque fragment est doté d'un rôle particulier. Cette approche étend la notion de *Pattern Class* de J. Soukup [Soukup 95], dans le sens où le modèle de fragments est plus détaillé et permet une meilleure manipulation des participants d'une instance d'un patron. Cette solution s'apparente à la notion d'objets morcelés [Bardou *et al.*, 96] , [Bardou 98].
- LayOM (*Layered Object Model*) [Bosch 96a], [Bosch 96b] est un modèle permettant une représentation explicite des instances de patrons de conception au niveau du code. Il s'agit d'une extension du modèle Objet conventionnel, basé sur plusieurs nouveaux concepts tels que les états (*states*), catégories (*categories*) et couches (*layers*). Les couches sont utilisées pour représenter les instances de patrons au niveau implémentation. Elles encapsulent les objets participant dans l'instance, et interceptent les messages envoyés et reçus par ceux-ci. Les couches sont organisées en classes, et chaque classe d'une couche représente un concept particulier (tel qu'une relation avec un autre objet ou un autre patron de conception).

– [Albin-Amiot *et al.*, 01] proposent un métamodèle dédié à la représentation structurelle et comportementale des patrons. Il sert à l'instanciation de ceux-ci et à la génération du code associé. Il est utilisé également pour la détection d'imitations de patrons. La structure de ce métamodèle est similaire à celle proposée par [Pagel *et al.*, 96]. Albin-Amiot *et al.* proposent, dans ce travail, de représenter l'instanciation (i.e. le modèle abstrait) d'un patron et ses imitations au niveau du code, par des classes.

Plusieurs autres travaux, tels que [Sunyé 99], [Borne *et al.*, 99], [Sunyé *et al.*, 00], [Le Guennec *et al.*, 00], [Guéhéneuc 03], [France *et al.*, 04], ont été présentés et comparés dans [El-Boussaidi *et al.*, 04]. Ils proposent tous de pallier les problèmes d'application et de traçabilité des patrons depuis la phase de conception jusqu'à la phase d'implémentation. Toutefois, ces approches dissimulent quelques uns des problèmes, mais pas tous, évoqués dans la section 3.3. Les problèmes liés, par exemple, à l'héritage, à la rupture d'encapsulation ou encore à l'indirection ne sont pas résolus, d'où le besoin de nouvelles approches.

4. Conclusion

Ce chapitre nous a permis, dans un premier temps, de montrer l'importance de la réutilisation de composants dans l'ingénierie des SI. Nous nous sommes intéressés, ensuite, aux approches de développement de SI basées sur la réutilisation de patrons d'ingénierie, une forme particulière de composants réutilisables. Nous avons montré et expliqué l'importance de l'utilisation de telles approches. Enfin, nous avons considéré plus particulièrement les patrons de conception par objets, propos de nos travaux de thèse. Les patrons de conception par objets définissent des solutions génériques à des problèmes récurrents en conception par objets. Établis à partir de solutions éprouvées, ils 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, tout en facilitant l'évolution et la réutilisation. Toutefois, en dépit de leurs nombreuses qualités, les patrons de conception par objets sont, néanmoins, le plus souvent, définis comme des collaborations entre plusieurs classes et ont le défaut ainsi de généralement « disparaître » dans l'implémentation des applications. Leur identification dans les implémentations d'applications est difficile, et ne repose généralement que sur une documentation rigoureuse, ou au mieux sur une organisation minutieuse des programmes en fichiers, librairies, ou paquetages (suivant les possibilités du langage de programmation utilisé). Le code de ces patrons étant donc dispersé et enchevêtré avec le reste du code de l'application, il pose en plus plusieurs problèmes d'utilisation et nuit à la réutilisation et à l'évolution du code des imitations des patrons et de celui relatif au reste de l'application. Nous avons détaillé notamment dans ce chapitre quatre problèmes liés à l'utilisation des patrons de conception par objets notamment ceux issus de [Gamma *et al.*, 95]. Il s'agit des problèmes de confusion, d'indirection, de rupture d'encapsulation et des problèmes liés à l'héritage.

Nous sommes convaincus que pour pallier les limites et remédier aux problèmes d'utilisation des patrons, il convient de découpler et d'isoler l'implémentation de ceux-ci de celle du reste de l'application. D'ailleurs, plusieurs travaux de recherches (cf. § 3.4), proposent en ce sens de nouvelles techniques de mise en oeuvre de patrons de conception, afin d'améliorer principalement leur traçabilité et de faciliter par conséquent l'évolution et la réutilisation de ceux-ci, mais aussi de l'application. Toutefois, ces approches ne permettent pas de pallier l'ensemble des problèmes liés à l'utilisation des patrons. De nouveaux modèles et langages de programmation associés, s'appuyant sur les acquis de l'approche Objet, voient le jour depuis quelques années sous la désignation plus générale d'approche Aspect. Par essence, ces nouvelles techniques prônent une décomposition des applications et des programmes, non seulement en classes représentant les entités du monde réel, mais également, en unités modulaires représentant les

préoccupations transversales au découpage de l'application en termes de classes. Nous sommes persuadés, en ce sens, que les mécanismes et concepts introduits dans le cadre de ces modèles et langages Aspect permettent de nouvelles solutions pour les patrons de conception par objets, contribuant ainsi à garder visible et isolée l'imitation de chaque patron dans le code d'une application. Il devient alors possible de modifier une implémentation d'un patron, de la remplacer par une implémentation équivalente ou encore par l'implémentation d'un autre patron apparenté, sans toucher au reste du code de l'application. Nous proposons ainsi d'utiliser les mécanismes et concepts Aspect, pour pallier les problèmes et limites d'utilisation des patrons, et résoudre les problèmes d'enchevêtrement et de dispersion de code. Le chapitre suivant fait un état de l'art des nouvelles techniques introduites dans le cadre de cette approche Aspect pour mieux les appréhender, et pouvoir en profiter ensuite dans le but d'améliorer la traçabilité et la réutilisation des patrons de conception depuis la phase de conception jusqu'à la phase d'implémentation.

Chapitre 2

Séparation des préoccupations et approche Aspect

L'évolution et la réutilisation des SI sont d'autant plus difficiles que les implémentations des systèmes techniques sur lesquels ils s'appuient sont contraignantes. L'étude des approches classiques de développement (telles que l'approche Objet, la métaprogrammation, l'approche à base de composants, etc.) montre, d'ailleurs, que le problème principal dans l'évolution et la réutilisation de tels systèmes est un problème d'organisation des programmes. Ces organisations sont le plus souvent non, ou peu, adaptées pour la représentation explicite de certaines préoccupations des systèmes, devant idéalement être modularisées indépendamment selon le principe de séparation des préoccupations. Il s'agit des préoccupations transversales comme par exemple la persistance des données ou encore la sécurité (cf. 1.2). La mise en œuvre de telles préoccupations dans les approches classiques résulte, en effet, en deux problèmes récurrents. Ce sont les problèmes de dispersion et d'enchevêtrement du code, admettant plusieurs conséquences qui affectent les applications de plusieurs façons tout en rendant plus difficiles leurs évolutions et réutilisations.

Afin de pallier ces deux problèmes, de nouveaux modèles et langages de programmation associés, s'appuyant le plus souvent sur les acquis de l'approche Objet, voient le jour depuis quelques années sous la désignation plus générale d'approche Aspect ou d'approche par aspects[4]. Par essence, ces nouvelles techniques permettent la programmation séparée et modulaire des préoccupations transversales. Elles proposent, en effet, de décomposer les programmes non seulement en unités modulaires propres aux préoccupations de base, mais aussi en unités modulaires spécifiques aux préoccupations transversales. Elles offrent de plus des moyens permettant la composition et l'intégration des diverses unités pour produire finalement des systèmes plus réutilisables. Afin de comprendre les principes de l'approche Aspect, nous proposons dans ce chapitre d'introduire les principaux modèles actuellement proposés dans le cadre de celle-ci. Nous proposons également de décrire les principaux travaux de recherche construits autour de ces modèles.

La section 1 de ce chapitre introduit le principe de séparation des préoccupations et les types de préoccupations transversales. La section 2 identifie les limites et problèmes des approches de développement classiques, qui constituent la motivation principale de l'approche Aspect. La section 3, propose un panorama des différents modèles introduits dans le cadre de cette nouvelle approche. Un accent est mis plus particulièrement sur la programmation par aspects et son langage AspectJ, d'une part, et sur la séparation multidimensionnelle des préoccupations ou l'approche Hyperspace et son langage Hyper/J, d'autre part. Il s'agit de deux modèles et langages de programmation par aspects les plus aboutis de nos jours. La section 4 fait la synthèse des modèles considérés et présente un état de l'art des travaux menés dans le cadre du développement de systèmes à base d'aspects. La section 5 conclut ce chapitre.

1. Séparation des préoccupations

1.1 Principe de base

Face à la complexité croissante des systèmes de nos jours, plutôt que d'attaquer leur développement de front, il est préférable de les décomposer en unités modulaires. Chaque unité traite d'un sous-problème particulier, devant être adressé indépendamment des autres problèmes. Le système final est ensuite construit par composition de l'ensemble des unités modulaires considérées. C'est l'approche désormais classique de modularisation évoquée depuis les années 1970 [Gauthier *et al.*, 70], [Parnas 72], [Dijkstra 76]

[4] Nous emploierons indistinctement les deux expressions « approche Aspect » ou « approche par aspects » que ce soit pour mettre en évidence l'objectif d'exprimer différents aspects ou le type de l'approche.

et qui a été adoptée par l'ensemble des approches de développement traditionnelles (procédurale, fonctionnelle, objet, etc.) et plusieurs autres approches plus récentes (réflexive, à base de composants, etc.). Toutes ces approches visent peu ou prou à appliquer un principe aujourd'hui bien approuvé dans l'ingénierie des SI, et connu sous la désignation plus générale de séparation des préoccupations (*Separation of Concerns* ou *SoC*) [Hursch *et al.*, 95]. Une préoccupation est l'abstraction d'un but particulier ou une unité modulaire d'intérêt. Un système typique admet principalement deux types de préoccupations : des préoccupations fonctionnelles et des préoccupations techniques (non fonctionnelles). Les préoccupations apparaissent dans les différentes phases du cycle de développement. Nous pouvons distinguer par exemple : des préoccupations liées aux données, à leur persistance et à leur contrôle d'accès, ou encore, des préoccupations architecturales telles que l'interopérabilité entre applications, ou des préoccupations de gestion organisationnelle, etc. L'abstraction d'une carte de crédit admet, par exemple, une préoccupation fonctionnelle représentant le processus de paiement. Elle admet aussi d'autres préoccupations techniques telles que : la gestion de l'authentification lors de la connexion, l'intégrité de la transaction de paiement, etc.

Le principe de séparation des préoccupations a pour objectif principal de représenter de manière abstraite et explicite les différentes préoccupations d'un système, à tous les niveaux du cycle de développement, et de les adresser indépendamment tout en identifiant leurs interrelations afin de les composer ensuite. Ce principe est au cœur de tout développement ou ingénierie moderne. Il s'agit d'un principe clé pour la réduction de la complexité de développement de grands systèmes. Il améliore en effet la compréhension de tels systèmes, facilite leur maintenance et leur évolution, et augmente leur réutilisation. Par exemple, garder séparé et bien localisé le code relatif à toute préoccupation particulière d'un système présente plusieurs avantages. Il devient ainsi plus facile de comprendre comment la préoccupation est adressée dans le code, dès lors qu'elle est bien identifiée et localisée. De plus son code est plus facile à modifier, à étendre et à réutiliser. Ceci reste valable pour toute l'application.

1.2 Préoccupations transversales

Le principe de séparation des préoccupations exprime une bonne modularisation et une bonne qualité du code, il n'indique pas cependant comment y arriver par un processus précis de développement. Plusieurs modèles et langages de programmation associés offrent, en ce sens, différents mécanismes et concepts qui permettent une meilleure organisation des programmes selon ce principe. Il s'agit en général d'organiser les programmes en unités modulaires séparées (par exemple, procédures, fonctions, objets, etc.) concernant chacune une préoccupation particulière. Cependant, comme le note Kiczales *et al.* [Kiczales *et al.*, 97], les modèles classiques se concentrent et traitent essentiellement de la séparation et la composition de préoccupations fonctionnelles. Ils ne garantissent pas, en revanche, une représentation modulaire et séparée de plusieurs autres préoccupations possibles (fonctionnelles ou non), qui affectent à la fois plus d'une unité modulaire fonctionnelle. Le code de telles préoccupations, dites transversales (*Crosscutting Concerns*), se trouve ainsi dispersé et enchevêtré dans l'ensemble du code des préoccupations fonctionnelles de base. Ceci soulève plusieurs problèmes préjudiciables à la compréhension, la maintenance, l'évolution et la réutilisation du code relatif à ces préoccupations transversales, mais aussi de celui relatif à l'application.

À travers l'expérience des différentes approches de développement classiques nous pouvons remarquer qu'il existe différents types de préoccupations transversales, dont les principales familles sont les suivantes :

- Préoccupations liées aux données (persistance de données, contrôle d'accès, etc.).
- Préoccupations de logging/tracing, de gestion d'exceptions, de debugging et de tests, etc.

- Préoccupations de gestion organisationnelle, workflow, etc.
- Préoccupations techniques et architecturales, telles que la configuration, la distribution, la synchronisation, l'interopérabilité entre applications hétérogènes, la sécurité, la gestion de performance, etc.
- Préoccupations fonctionnelles mettant en jeu plusieurs classes/objets en collaboration (souvent capturées dans des patrons de conception ou d'implémentation).

2. Limites et problèmes des approches classiques pour la séparation des préoccupations transversales

Nous discutons, dans ce qui suit, des principales approches classiques bâties autour du paradigme Objet, afin de montrer leurs limites et problèmes vis-à-vis de la séparation, la modularisation et l'intégration des préoccupations transversales.

2.1 Procédures, modules et objets

Dans le cadre de la programmation procédurale ou fonctionnelle, l'accent est mis sur les fonctionnalités attendues du système concret. Les constructions de base offertes par les langages qui supportent ce paradigme de programmation sont les procédures et fonctions, qui peuvent admettre des arguments et renvoyer des valeurs. Les données du système sont globales à l'ensemble des constructions du programme, et donc partagées par celles-ci. Avec l'augmentation de la complexité des SI de nos jours, la programmation procédurale atteint ses limites : il devient très difficile de maintenir, de faire évoluer et de réutiliser les applications à base de procédures et fonctions.

La programmation modulaire utilise les principes d'encapsulation et d'abstraction des données, deux principes cristallisés à l'origine avec le langage Modula-2 [Wirth 84] et appuyés aujourd'hui par le langage ADA [Booch 97]. Selon ces deux principes, l'ensemble des procédures et/ou fonctions dépendantes et les données qu'elles manipulent sont regroupées et séparées dans un module à part définissant un type abstrait. Un programme est alors constitué de plusieurs modules et la collaboration inter modules se fait à travers des interfaces (les détails d'implémentation de chaque module sont cachés aux autres modules). L'abstraction et l'encapsulation augmentent la lisibilité et donc la compréhension du code des différents types définis par les modules, facilitant ainsi la maintenance et l'évolution de ceux-ci. Toutefois, poussés à l'extrême, ces deux principes entraînent une multiplication des dépendances et donc des interactions entre modules qui correspondent souvent à des préoccupations transversales. Ainsi, toute modification ou extension d'une ou de plusieurs interactions est contraignante, et peut entraîner une refonte complète ou partielle des modules affectés, dans le but d'adapter les données, les procédures et les fonctions de ces derniers. D'autant plus que ces dépendances et interactions diminuent la réutilisation directe de chacun de ces modules dans un autre contexte d'application. La programmation modulaire atteint ainsi ses limites face à la complexité croissante et l'évolution continue des SI. La décomposition des programmes en modules fonctionnels est mal adaptée pour l'ensemble des interactions, qui se trouvent non encapsulées et dispersées (car transverses au découpage du programme en termes de modules fonctionnels). Toutes ces interactions de natures différentes conduisent à un enchevêtrement et une dispersion du code, qui n'est pas souhaitable et nuit à la lisibilité, à la compréhension et par conséquent à l'évolution et à la réutilisation de toute l'application.

La programmation par objets (*Object-Oriented Programming* ou *OOP*) apporte une solution élégante aux problèmes posés par la programmation modulaire, où elle prend ses racines. Elle propose de composer les programmes en classes d'objets. Les objets réifient les entités du monde réel. Les classes abstraient les

différents objets d'un même type et encapsulent les données et les opérations qui leurs sont associées. L'encapsulation augmente la modularité et la compréhension des classes correspondant aux préoccupations fonctionnelles de base, elle facilite donc leur évolution. L'abstraction, utilisée avec le mécanisme d'héritage, le polymorphisme et la liaison dynamique, améliore la réutilisation. Cette approche offre grâce à tous ces mécanismes un moyen significatif pour le développement de systèmes réutilisables ; elle facilite également l'évolution et l'extension des classes de manière incrémentale. Elle est ainsi largement adoptée et est aujourd'hui appuyée par de nombreux langages de programmation (Smalltalk [Goldberg *et al.*, 83], Eiffel [Meyer 92], [Jézéquel 96], Java, etc.) et outils de développement. Toutefois, et tout comme l'approche modulaire, le niveau de granularité très bas de cette approche (i.e. objet/classe) pose quelques problèmes [Meyer 97], [Finch 98], [Villalobos 03]. La programmation par objets se trouve mal adaptée pour une représentation explicite, modulaire et séparée d'interactions complexes entre groupes d'objets [Shtern 04], [Oussalah *et al.*, 05], qui sont en collaboration pour exprimer certaines préoccupations transverses au découpage des applications en termes de classes. Le code relatif à de telles préoccupations se trouve ainsi dispersés et enchevêtrés avec le reste du code des classes de l'application, affectées par ces dernières. Pour illustrer ceci, considérons l'exemple d'une application graphique basée sur un éditeur de figures simple, issue de [AspectJ 02]. Le diagramme de classes de cette application est donné par la figure 2.1. Les figures (Figure) consistent en un certain nombre d'éléments (FigureElement), qui peuvent être des points (Point) ou des lignes (Line). L'éditeur de figures utilise aussi un afficheur (Display) pour l'affichage des figures. La méthode incrXY() définie par la classe FigureElement permet d'incrémenter les coordonnées d'un élément d'une figure. Elle fait appel aux méthodes setX() et setY() pour la mise à jour des coordonnées d'un point. Elle peut faire appel aussi aux méthodes setP1() et setP2() (qui font appel à leur tour aux méthodes setX() et setY()) pour la mise à jour des coordonnées d'une ligne. Toute figure dont les coordonnées de ses éléments sont modifiées doit être réaffichée. La mise à jour de l'affichage d'une figure est assurée, dans cette conception, par l'ensemble des méthodes d'accès en écriture et la méthode incrXY() qui collaborent pour mettre indirectement en œuvre la préoccupation d'affichage. Il s'agit en effet d'une préoccupation transversale dont la définition (et par conséquent le code) se trouve dispersée dans au moins les deux classes Point et Line, et enchevêtrée avec le reste du code des méthodes directement concernées par cette préoccupation. Le code de ces méthodes mêle plusieurs préoccupations en même temps (par exemple, l'accès en écriture à l'abscisse d'un point et la mise à jour de son affichage par la méthode setY()) ; il est non propre et moins lisible et par conséquent peu compréhensible. Ceci nuit fortement à l'évolution des classes correspondantes et à leur réutilisation directe dans un autre contexte d'application, qui ne requiert pas la fonction d'affichage par exemple. D'ailleurs, le code relatif à la fonction d'affichage et sa mise à jour étant dispersé dans l'ensemble des méthodes en collaboration, il est difficile à localiser, à comprendre et donc à faire évoluer. De plus, sa réutilisation est impossible du moment où il n'est pas explicitement isolé.

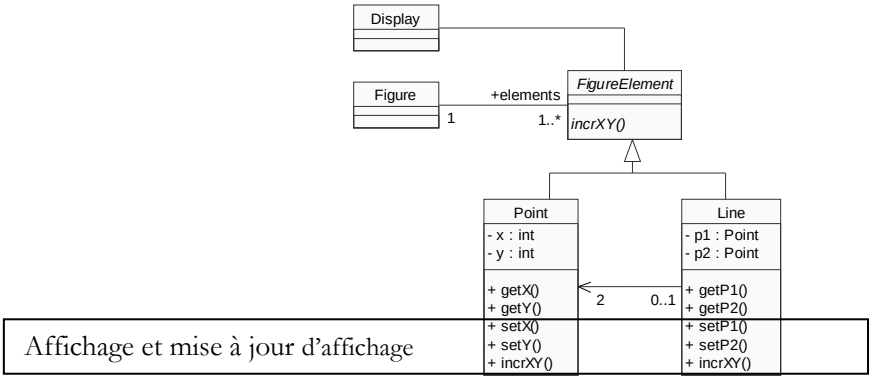


Figure 2.1. *Diagramme de classes UML d'un éditeur de figures [AspectJ 02]*

Comme le montre cet exemple, le code des applications à base d'objets peut être difficile à comprendre, à maintenir et à faire évoluer : il est difficile de s'affranchir des dépendances et interactions entre classes afin de pouvoir les réutiliser séparément. Ces mêmes problèmes existent également pour les représentations et les codes des préoccupations transversales. En ce sens, pour une meilleure lisibilité, compréhension, évolution, et surtout une bonne réutilisation du code de l'ensemble d'une application à base d'objets, divers patrons de conception permettant par exemple une meilleure représentation explicite de certaines préoccupations transversales ont été proposés dans la littérature. Mais, comme nous l'avons montré dans le premier chapitre (cf. § 1.3), ces patrons présentent, eux-mêmes, plusieurs limites et introduisent à leur tour différents problèmes liés à la dispersion et à l'enchevêtrement du code relatif à leurs préoccupations. En conclusion, malgré ses limites et problèmes conséquents, la programmation par objets fournit des éléments de réponse à la réutilisation tout en donnant une cohérence conceptuelle à la notion de modularisation, d'abstraction ou de séparation de quelques préoccupations. La programmation par objets facilite l'évolution et augmente la réutilisation des systèmes complexes. En effet, la vocation de l'objet réside dans sa capacité à s'ajuster aux nouveaux besoins de développement efficace. En ce sens, plusieurs extensions ont été proposées pour améliorer ce paradigme et de nouvelles approches ont vu le jour, dont le but est d'accentuer le pouvoir d'abstraction, de modularisation, et de faciliter l'évolution et la réutilisation des applications à base d'objets tout en offrant des concepts toujours plus évolués.

2.2 Programmation réflexive

La programmation réflexive est basée sur des Protocoles à MétaObjets (*Méta-Object Protocols* ou *MOP*) [Maes 87], [Ferber 89], [Kiczales *et al.*, 91]. Un langage réflexif est un langage dont le modèle est décrit dans ce même langage, et qui peut être modifié, adapté, par programme. Plusieurs langages réflexifs ont été proposés dans la littérature, dont nous pouvons citer principalement Smalltalk [Goldberg et al., 83], ObjVLisp [Cointe 87], Common Lisp Object System (*CLOS*) [Kiczales et al., 91], CodA [Affer 95], OpenC++ [Chiba 95] et OpenJava [Tatsubori *et al.*, 00]. Un protocole à métaobjets (associé à un langage réflexif) décrit d'une part, les concepts du langage et leurs relations, et d'autre part, la sémantique ou le comportement par défaut des méthodes de ces concepts. Un Métaobjet est une entité qui définit en partie la structure et le comportement d'un ou de plusieurs objets de même nature. Dans le cadre d'un langage de programmation par objets, toute classe est le métaobjet de ses instances. De la même manière, toute métaclasse est le métaobjet d'une ou de plusieurs classes du langage. La métaclasse racine est généralement sa propre instance et hérite de la classe Objet pouvant ainsi être manipulée comme une entité du programme. Les *MOP* définissent donc, à la fois, la façon dont les entités du langage sont réifiées et le protocole de création et de manipulation de ces objets. Ils permettent ainsi de modifier l'implémentation et d'étendre les fonctionnalités et l'expressivité du langage. Ceci est possible grâce à la séparation du quoi (fonctionnalités) et du comment (implantation), tout en repoussant le comment dans un niveau méta contrôlant le niveau fonctionnel de base grâce à la réflexivité. La réflexivité est le pouvoir de manipuler sa propre structure et son propre comportement comme n'importe quelle donnée, permettant ainsi d'étendre et d'adapter le langage aux besoins de l'utilisateur d'une manière incrémentale et modulaire. Il suffit tout simplement de modifier les métaobjets pour modifier le langage. Les protocoles à métaobjets ont ainsi une vocation plus générale que d'autres modèles en ce qui concerne l'adaptabilité, car ils permettent de modifier directement le langage tout en proposant d'ouvrir celui-ci aux modifications du développeur. Pour ce faire, tout *MOP* définit des points d'accès permettant la modification du comportement du langage considéré. Par exemple, les métaclasses d'un langage sont en général responsables de la sémantique de l'envoi et de réception des messages entre entités de l'application : la méthode définissant cette sémantique représente un point d'accès. Cette méthode intercepte tous les messages échangés entre objets de l'application, et se charge de leur exécution. Elle peut ainsi être modifiée pour changer le

comportement ou le fonctionnement de base des méthodes de classes, qui sont associées à la métaclasse détenant cette méthode. En ce sens, certaines préoccupations transversales à l'ensemble des classes d'une application donnée, peuvent être exprimées dans un niveau méta, à travers les méthodes des métaclasses du langage même. Nous pouvons, par exemple, modifier la méthode d'interception de messages de la métaclasse réifiant les classes de l'application, de manière à pouvoir garder la trace dans un journal l'exécution de toutes ou certaines méthodes de cette dernière. Cette approche permet donc de garder séparées et encapsulées des préoccupations transversales dans des unités modulaires (i.e. les métaclasses du langage même), et d'éviter que leur code ne se disperse dans l'ensemble des classes de l'application affectées par celles-ci.

Bien qu'elle soit extrêmement flexible, cette approche pose certains problèmes. La pratique montre que la métaprogrammation est plus difficile à maîtriser que les autres approches traditionnelles. En effet, celui qui met en œuvre les préoccupations doit connaître le protocole à métaobjets, ou faire appel à un métaprogrammeur. D'autant plus qu'il est difficile de raisonner au niveau méta, cela demande de la formation et de l'entraînement. De plus, un programmeur n'est jamais sûr de la sémantique du langage qu'il utilise : il ne lui est pas facile de connaître les modifications effectuées par les autres programmeurs. Ceci peut conduire à des erreurs de programmation qui peuvent avoir d'importantes conséquences sur le système pouvant mener au blocage. D'autre part, si plusieurs préoccupations transversales affectent une classe donnée, elles doivent être toutes définies dans la même métaclasse associée à la classe affectée. Ce qui augmente l'enchevêtrement du code et ne garantit pas la séparation entre préoccupations transversales. Cette approche ne permet pas d'exprimer clairement le protocole de composition des différentes préoccupations qui se retrouvent enchevêtrées dans les métaclasses.

La faiblesse principale de la réflexivité vient de son cadre extrêmement général. En effet, la réflexivité pouvant potentiellement être appliquée à tout type d'environnement ou de langage, il devient extrêmement difficile d'en définir les limites et les apports de manière précise. De plus, elle n'offre qu'un niveau restreint de séparation des préoccupations et de nombreux efforts restent nécessaires lorsque plusieurs préoccupations distinctes entrent en jeu.

2.3 Programmation générique et mixins

La programmation générique repose sur le principe de paramétrage de classes. Une classe définie dans un langage générique peut admettre un ou plusieurs paramètres génériques formels, caractérisant son degré d'ouverture. De nombreux langages de programmation à base d'objets supportent la programmation générique : CLOS [Kiczales *et al.*, 91], C++ [Stroustrup 97], Eiffel [Meyer 92], et Java (à partir de la version 1.5) [Austin 04]. Les classes génériques permettent de découpler les préoccupations liées à la définition de structures de données, par exemple, de la spécification du ou des types de données concrets que détiennent ces structures. Il s'agit de la séparation de préoccupations de données.

Les *Mixins* [Bracha *et al.*, 90], [Czarneci *et al.*, 99] et les *Mixins Layers* [Smaragdakis *et al.*, 98] sont deux techniques dérivées de la programmation générique, qui permettent, en plus, la séparation d'autres types de préoccupations transversales. Un *mixin* est un fragment de classe destiné à être composé avec d'autres classes ou *mixins*. Le terme mixin est introduit à l'origine dans Flavors [Moon 86], le prédécesseur de CLOS [Demighiel *et al.*, 87], [Keene 89]. La différence entre une classe régulière (modélisant une entité du monde réel) et un mixin, est qu'un mixin modélise quelques fonctionnalités. En ce sens, un mixin n'est jamais destiné à être utilisé seul, mais, à être composé avec d'autres classes ou *mixins* utilisant les fonctionnalités qu'il modélise. Une manière de mettre en œuvre les *mixins* dans le cadre de l'approche Objet est d'utiliser les classes et l'héritage multiple. Dans ce cas, un mixin est représenté comme étant une classe pouvant être dérivée en une ou plusieurs classes composées, à partir d'un nombre de *mixins* et de

classes régulières, par héritage multiple. Les *mixins* modélisent et séparent donc les différentes préoccupations fonctionnelles, et celles transversales au découpage de l'application en termes de classes régulières. Une autre manière de faire, dans le cadre de la programmation générique, est d'utiliser les classes et un héritage paramétré. Dans ce cas, un mixin est modélisé comme une classe dont le paramètre générique précise sa superclasse, permettant ainsi d'implémenter une extension des fonctionnalités existantes ou de nouvelles fonctionnalités qui sont orthogonales à un ensemble de classes régulières. Toutefois, comme les approches introduites précédemment, la programmation générique et ses dérivés présentent certaines limites, d'autant plus qu'elles correspondent à des approches trop restrictives de séparation de préoccupations. L'utilisation de l'héritage multiple comme une technique de mise en œuvre des *mixins* est aussi problématique, surtout dans le cas où le langage considéré pour le codage final ne permet pas l'héritage multiple (tel que Java, par exemple).

2.4 Approche à base de composants

L'approche à base de composants permet de concevoir et de développer des systèmes par assemblage de composants réutilisables, à l'image par exemple des composants électroniques ou des composants mécaniques. La programmation à base de composants [Szyperski 02] propose une nouvelle approche de séparation de préoccupations techniques (telles que la persistance, la synchronisation, la distribution, etc.), augmentant notamment la réutilisation de composants fonctionnels. Il s'agit de regrouper dans des unités modulaires logicielles distinctes (appelées composants) du code correspondant à des préoccupations de type métier, tout en gardant séparées les préoccupations techniques du système. L'idée de base est de décomposer les applications en des préoccupations relativement indépendantes, et d'adresser chacune (décrire, analyser, concevoir et implémenter, etc.) sans devoir se référer aux autres préoccupations ou tout au moins de manière marginale et bien définie (au moyen d'interfaces ou de contrats). Les préoccupations techniques transversales à ces unités métiers sont adressées séparément. De nombreux concepts tels que les notions d'interfaces requises et fournies ou les contrats, les intercepteurs, les connecteurs, les conteneurs ou encore les événements, permettent aux composants de mieux séparer ces préoccupations techniques. Ceci structure le programme, et permet donc de mieux maîtriser sa complexité [Brooks 95]. En particulier, dans un contexte de programmation à grande échelle [DeRemer *et al.*, 76], ces unités métiers deviennent des unités de configuration et de déploiement. Aujourd'hui, des propositions ont émergé sur les composants. Elles sont pour la plupart orientées très fortement sur l'aspect implémentation (bas niveau) (*EJB* [Sun-EJB 04], *.NET* [Lowy 03], *CCM* [OMG-CCM 02], etc.). D'un autre côté les composants de haut niveau préconisés par les ADL (*Architectures Description Languages*) comme Fractal [Bruneton 04], [Bruneton *et al.*, 04a], [Bruneton *et al.*, 04b], perdent leurs atouts dès qu'ils sont projetés au niveau implémentation. Dans tous les cas, à terme, les enjeux sont l'amélioration de la qualité de la productivité, et l'industrialisation de la production de logiciels à travers la réutilisation.

Les composants avec des niveaux de granularité et d'abstraction plus élevés que les objets, sont mieux adaptés à une industrie de réutilisation. Toutefois, ils sont sujets aux problèmes de dispersion et d'enchevêtrement de code de certaines préoccupations transversales. En effet, les systèmes à base de composants utilisent les modèles objets comme support pour la spécification, la conception et l'implémentation des composants et leurs interactions [Garlan 00], [Khammaci *et al.*, 05], [Medvidovic *et al.*, 02]. Chaque composant est représenté, par exemple, par une ou plusieurs classes. Les interfaces des composants peuvent être représentées par des interfaces de classes, et les interactions entre objets d'un même composant peuvent être définies en termes d'associations et d'appels de méthodes. Plusieurs dépendances entre objets d'un même composant trouvent ainsi leurs définitions dispersées et enchevêtrées dans le reste du code des classes du composant. Les dépendances et interactions entre composants ne sont pas modularisées, et leurs définitions se trouvent dispersées et enchevêtrées. D'autre

part, les préoccupations ou services techniques supportés par la plupart des modèles de composants sont réalisés, aujourd'hui, sous la forme de canevas d'objets. Ils sont intégrés aux modèles grâce au mécanisme d'héritage, qui se révèle insuffisant pour une implémentation modulaire de ces préoccupations. Leurs implémentations se trouvent, en effet, entrelacées avec le reste du code des composants. De plus, ce mécanisme d'extension n'est pas suffisant pour enrichir les modèles de base par d'autres préoccupations, les préoccupations mises en œuvre par cette approche sont très restrictives. Il est complexe, voire difficile souvent, de pouvoir composer plus d'un service technique à la fois.

2.5 Synthèse des approches classiques

Nous pouvons remarquer à travers la discussion des principales approches classiques de développement étudiées que, quel que soit le modèle de programmation considéré, la séparation et la modularisation de certains types de préoccupations ne sont pas complètement respectées et appliquées.

- L'unique dimension de décomposition en classes proposée par l'approche Objet présente, par exemple, plusieurs limites. En effet, les classes ne fournissent pas, ou peu, de supports directs pour représenter de façon modulaire les préoccupations transversales au découpage d'une application en termes de classes. D'ailleurs, le seul moyen d'interconnecter différentes classes d'objets se limite à l'invocation de méthodes, rendant ainsi difficile la séparation d'interactions complexes entre groupes d'objets en collaboration.
- L'utilisation seule de la métaprogrammation pour séparer les préoccupations transversales est restrictive. Elle est en plus complexe à mettre en œuvre.
- L'apport de la programmation générique est limité. Ce paradigme de programmation permet uniquement la séparation de quelques types, et non pas tous, de préoccupations transversales (notamment les préoccupations de données).
- Les modèles à base de composants proposent de structurer les applications en modules indépendants et réutilisables, tout en permettant une mise en œuvre de manière transparente et flexible de certaines préoccupations techniques (telles que la gestion de la distribution, de la synchronisation, des transactions ou encore de la persistance). Toutefois, le nombre de préoccupations prises en compte par ces modèles de composants est très limité.

Chacune de ces approches de développement classiques propose, en effet, sa propre solution pour la séparation d'un ou de plusieurs types de préoccupations transversales. Ces approches diffèrent, cependant, par la manière dont sont adressées la composition et l'adaptation de l'ensemble des préoccupations (transversales ou non) d'une application. Nous proposons, dans le tableau suivant (cf. tableau 2.1), de caractériser ces approches selon les critères suivants :

- *Couverture*. La solution proposée supporte-t-elle la séparation de tous les types de préoccupations distingués ci-dessus ?
- *Expressivité*. La représentation des préoccupations est-elle explicite, modulaire et séparée ? Le programme résultant est-il lisible et compréhensible ?
- *Intégrabilité*. La solution proposée permet-elle d'intégrer proprement et facilement les différents types de préoccupations (transversales ou non) qu'elle supporte ? Le programme résultant est-il efficace et correct ? La sémantique du programme résultant est-elle facilement maîtrisable (lisible, facile à comprendre, à faire évoluer et à réutiliser) ?
- *Adaptabilité*. La solution proposée permet-elle d'ajouter, de retirer et de modifier facilement certaines préoccupations, sans modifier l'application de base ?

Tableau 2.1. *Caractérisation des approches classiques pour la séparation des préoccupations*

	Approche Objet	*Programmation générique et Mixin*	*Programmation réflexive*	*Approche Composant*
Couverture	moyenne	restrictive	plutôt bonne	quelques préoccupations techniques
Expressivité	moyenne	plutôt bonne pour un nombre faible de types de préoccupations.	plutôt bonne mais difficile à mettre en oeuvre.	plutôt bonne.
Intégrabilité	mauvaise, sauf dans le cadre restreint de patrons.	complexe à mettre en oeuvre.	très complexe à mettre en oeuvre et souvent impossible.	plutôt bonne mais très bas niveau
Adaptabilité	bonne dans le cadre restreint de patrons.	plutôt bonne.	plutôt bonne.	excellente

2.6 Problèmes récurrents et conséquences

La mise en œuvre des préoccupations transversales dans le cadre des approches classiques conduit, le plus souvent, à plusieurs problèmes récurrents que nous avons déjà relevés dans le contexte particulier de l'implémentation des patrons de conception par objets (cf. § 3.3, chapitre 1) et que nous reprenons ici dans le contexte plus général du développement de systèmes informatiques, que celle-ci s'appuie sur des patrons de conception ou non. Ces problèmes découlent d'une manière générale de deux problèmes de fond récurrents en développement traditionnel : les problèmes de dispersion et d'enchevêtrement du code.

- **L'enchevêtrement du code**. Les unités modulaires de décomposition d'un système logiciel peuvent interagir de manière simultanée avec plusieurs préoccupations transverses. Il est fréquent, par exemple, que les développeurs aient à penser simultanément à la persistance, à la synchronisation ou encore à la sécurité d'accès aux données d'une même préoccupation fonctionnelle de l'application. Ceci conduit à la présence d'éléments de définition de plusieurs préoccupations transverses dans la définition d'une unique préoccupation fonctionnelle de base, donnant lieu ainsi à un code enchevêtré.
- **La dispersion du code**. Les préoccupations transversales concernent, en général, une ou plusieurs préoccupations de base. Leurs définitions se trouvent ainsi disséminées au travers de l'ensemble des unités modulaires représentant les préoccupations de bases qu'elles affectent. La définition d'une préoccupation transversale se trouve ainsi dispersée dans plusieurs unités modulaires.

Combinés ensemble, ces deux problèmes récurrents contrarient le principe de séparation de préoccupations, affectant ainsi la conception et l'implantation d'applications de plusieurs façons.

- **Code non lisible et difficile à comprendre**. L'enchevêtrement (respectivement, la dispersion) du code d'une ou de plusieurs préoccupations transversales avec celui (respectivement, dans celui) relatif à l'ensemble des préoccupations de base, produit souvent un code final mêlant plusieurs préoccupations dissimulées. Ceci diminue la lisibilité du code qui se trouve moins compréhensible.
- **Mauvaise traçabilité**. Les préoccupations transversales trouvant leur code dispersé dans celui relatif au reste de l'application, sont difficiles à localiser dans le code final et donc non traçables.
- **Maintenabilité et évolution difficiles**. Le code étant non lisible et moins compréhensible, il est très difficile de pouvoir localiser pour maintenir ensuite et faire évoluer les différentes préoccupations de l'application. D'ailleurs, toute modification ou évolution d'une ou plusieurs

préoccupations transversales, est très complexe dès lors qu'elle affecte toutes les préoccupations de base concernées par celle-ci.

- **Faible efficacité et productivité**. La définition simultanée de plusieurs préoccupations transversales en même temps et dans une même unité modulaire, éloigne l'attention du développeur du but final de l'application, vers des besoins annexes, limitant ainsi son efficacité et sa productivité.
- **Faible réutilisation du code**. Le code des différentes préoccupations transversales étant dispersé, il est non réutilisable. D'ailleurs, le code relatif aux préoccupations de base étant « non propre » (mélangé avec celui des préoccupations transversales affectant ces dernières), il est difficilement réutilisable dans plusieurs autres contextes d'utilisation.

3. Approche Aspect

La section précédente nous a permis de montrer que les modèles classiques de programmation présentent, le plus souvent, certaines limites et problèmes conséquents vis-à-vis de la séparation et de la représentation explicite et modulaire des préoccupations transversales. De la même façon ils n'offrent pas, ou peu, de solutions efficaces pour la composition et l'adaptation de telles préoccupations. Ces observations sont aujourd'hui au cœur de plusieurs travaux de recherche, dont le but principal est d'offrir des modèles permettant une meilleure séparation et composition de tout type de préoccupations. De nouveaux modèles et langages de programmation associés sont ainsi actuellement proposés. Ces modèles sont connus sous la désignation plus générale d'approche Aspect. Ils prônent une décomposition des programmes non seulement en unités modulaires représentant les préoccupations fonctionnelles de base, mais aussi en unités modulaires dédiées à la représentation des préoccupations transversales. Ils offrent de plus des solutions adéquates de composition de préoccupations (on parle aussi de tissage), afin de construire des systèmes efficaces.

Les modèles et langages de programmation, nouvellement introduits dans le cadre de l'approche Aspect, sont définis le plus souvent comme étant des déclinaisons (ou des extensions) des modèles et langages de programmation déjà existants (procédurale et fonctionnelle, Objet, Composant, etc.). Nous nous intéressons plus particulièrement aux extensions par aspects du paradigme Objet, car notre objectif est d'apporter des solutions aux problèmes liés à l'utilisation des patrons de conception par objets. Nous proposons dans ce qui suit de présenter certaines de ces extensions. Il n'est pas nécessaire et utile de s'attarder sur l'ensemble des déclinaisons existantes, nous essayons de considérer en revanche les principales familles de solutions actuelles. Un accent est mis toutefois, notamment, sur la programmation par aspects et l'approche Hyperspace, les deux modèles Aspect que nous utilisons pour fonder notre proposition.

3.1 Programmation adaptative

La programmation adaptative (*Adaptative Programming* ou *AP*) [Lieberherr 94], [Lopes *et al.*, 94a], [Lopes *et al.*, 94b], [Lieberherr *et al.*, 94], [Lieberherr 95], [Palsberg *et al.*, 95], [Orleans *et al.*, 01], [Lieberherr *et al.*, 01], connue aussi sous le nom de programmation orientée patron, et son implémentation DemeterJ [Lieberherr *et al.*, 97] sont destinées à l'origine à résoudre des problèmes de dépendances entre comportements et structures d'objets d'une application, de manière à assurer une meilleure séparation de ces préoccupations. En effet, toute collaboration dans un programme à base d'objets fait intervenir le plus souvent plusieurs (petites) méthodes, qui interagissent tout en mettant en jeu entre elles de nombreuses propriétés structurelles, augmentant ainsi les dépendances entre objets intervenant dans cette collaboration.

De telles collaborations sont très complexes et difficiles à comprendre, toute modification de ces collaborations nécessite la modification d'un grand nombre de méthodes, et affecte donc plusieurs classes de l'application. Toute modification dans la structure des objets de l'application nécessite une refonte de la collaboration. Tout ceci nuit énormément à la maintenance, à l'évolution et la réutilisation des programmes. Partant de ce constat, la programmation adaptative propose de garder séparés et encapsulés, les comportements relatifs à de telles collaborations dans des unités modulaires distinctes. Ces préoccupations sont abstraites sous forme de patrons. Tous les patrons d'une application sont classés en différentes catégories, chacune des catégories représente une abstraction particulière et donc une préoccupation donnée. Par exemple, les patrons de propagation d'opérations sur les données, les patrons de transport qui assurent la paramétrisation, et les patrons de synchronisation modélisant les accès concurrents entre applications, modélisent différentes préoccupations transverses.

3.2 Programmation par rôles ou points de vue

La programmation par rôles ou par points de vues [Kristensen *et al.*, 96], [Gottlob *et al.*, 96], [Bardou *et al.*, 96], [Abiteboul *et al.*, 91], [McDirmid *et al.*, 03], par exemple, propose de structurer les programmes en groupes d'objets. Chaque groupe représente un point de vue du système à développer. Dans le cadre de cette approche, et contrairement à ce qui est admis dans l'approche Objet, une entité du monde réel est représentée par un ou plusieurs objets. Chaque objet modélise un rôle particulier joué par cette entité (selon le point de vue considéré) et faisant partie d'un unique groupe d'objets. Les différentes vues subjectives du système représentent l'ensemble des préoccupations de l'application, et chaque préoccupation est représentée par les différents rôles et leurs interactions. Il est à noter que ce paradigme ne fait pas la distinction explicite entre préoccupations de base et préoccupations transversales. En effet, chaque groupe d'objets représente une préoccupation au sens large, et toutes les préoccupations sont au même niveau. Bien que cette approche permette une meilleure séparation des préoccupations, elle présente certaines limites. Elle induit le plus souvent des problèmes de partage de données et de recouvrement de diverses définitions entre préoccupations distinctes. L'intégration de propriétés d'objets représentant une même entité du monde réel, mais appartenant à des groupes d'objets différents, est en effet très complexe à mettre en œuvre.

3.3 Programmation par sujets

La programmation par sujets (*Subject-oriented Programming* ou *SOP*) propose une autre technique de séparation de préoccupations introduite par [Ossher *et al.*, 93]. A la manière de la programmation par rôles ou points de vue, elle propose de structurer les programmes en sujets qui peuvent être dus aux différents développeurs, utilisateurs du système, et/ou différents contextes d'utilisation. Chaque sujet représente une préoccupation particulière. Il est défini comme étant une collection d'états et de comportements (un groupe d'objets/classes ou fragments de classes liés entre eux grâce à l'héritage ou autres relations) qui reflètent une perception du monde réel du système au sens large. Les sujets d'une application sont tous vus au même niveau, aucun sujet n'est a priori dominant : on parle de préoccupations au sens large, sans différencier entre préoccupation de base ou préoccupations transversales. Le système final est produit ensuite par composition de l'ensemble des sujets considérés. Ce processus de composition est connu sous le nom d'intégration. Les sujets intégrés peuvent être à leur tour composés avec d'autres sujets simples ou intégrés, pour produire des sujets plus importants qui traitent de plusieurs préoccupations à la fois. L'intégration de l'ensemble des sujets est déterminée par un ensemble de règles de composition [Ossher *et al.*, 95], [Ossher *et al.*, 96]. Toute règle de composition met en jeu deux ou plusieurs sujets (les classes qui les composent et leurs propriétés structurelles et comportementales) destinés à être intégrés. Les règles de composition sont de trois types : règles de mise en correspondance, de combinaison, ou de combinaison

et de mise en correspondance en même temps. Les règles de mise en correspondance indiquent les correspondances, si elles existent, entre classes (respectivement propriétés des classes) ayant des noms identiques ou différents, et appartenant à différents sujets destinés à être intégrés. Il est possible d'utiliser un type particulier de règle permettant la mise en correspondance en même temps de classes et de leurs propriétés d'une manière implicite. Toutefois, il est possible aussi d'utiliser des règles de mise en correspondance et de combinaison en même temps, pour indiquer que certaines des propriétés de ces classes ne doivent pas être mises en correspondance. Les règles de combinaison précisent la façon dont les classes (respectivement propriétés des classes) de sujets doivent être composées (fusion, redéfinition, ou encore fusion avec un ordre d'exécution particulier). La définition des règles de composition est encapsulée dans un ou plusieurs modules de composition, qui mettent en jeu plusieurs sujets à intégrer ensemble.

Ce nouveau paradigme possède un certain nombre d'avantages quant à la conception et à l'implémentation de systèmes complexes, tout en respectant le principe de séparation des préoccupations. Ces avantages sont les suivants :

- Il offre la possibilité de développement d'objets décentralisés [Ossher *et al.*, 94]. Les développeurs de différentes applications, qui partagent les mêmes objets, peuvent avoir leurs propres définitions des objets qu'ils partagent selon leurs propres vues subjectives. En gardant séparées ces diverses définitions, ce paradigme assure une meilleure séparation des préoccupations des systèmes.
- Chaque préoccupation du système étant définie et encapsulée dans un unique sujet cohérent, le code de l'application (et donc des préoccupations) est plus propre, lisible, facile à comprendre et par conséquent plus facile à faire évoluer et à réutiliser.
- Il permet de faire évoluer plus facilement les fonctionnalités d'un système, non prévues a priori, et ce sans toucher à l'application : le code existant reste inchangé. Pour ce faire, il suffit tout simplement de définir et d'encapsuler dans un nouveau sujet l'évolution attendue (modifications ou extensions), et de préciser les règles de composition (de ce sujet avec les sujets à faire évoluer) nécessaires pour mettre en œuvre cette évolution.

3.4 Filtres de composition

L'approche par filtres de composition (*Composition Filters* ou *CF*) [Aksit *et al.*, 88], [Aksit 89], [Bergmans 94], [Aksit *et al.*, 96] est motivée par la difficulté d'adresser, de manière indépendante et séparée, la coordination de messages complexes échangés entre objets en collaboration pour réaliser une ou plusieurs préoccupations particulières. Partant de ce constat, Aksit *et al.* proposent d'étendre le paradigme Objet en ajoutant le concept de filtre de messages, qui intercepte l'envoi et la réception de tout message envoyé ou reçu par un objet. Deux types de filtres sont considérés : les filtres d'entrée et les filtres de sortie interceptant, respectivement, les messages reçus ou envoyés par un objet. Dans ce nouveau modèle, comme le montre la figure 2.2, un objet consiste donc en une interface et un objet interne (*inner object*). L'interface est composée, entre autres, d'un nombre arbitraire de filtres de composition, qui sont regroupés dans deux modules spécifiques respectivement à l'interception des messages reçus et envoyés. Un filtre est en fait conçu comme étant un objet qui évalue et manipule les messages réifiés, tout au long de leur cheminement. Il peut accepter ou rejeter un message, tout en déclenchant une série d'actions supplémentaires et nécessaires à la réalisation de la fonctionnalité attendue à travers le message intercepté. Ceci dépend du message même et de quelques conditions qui portent sur l'état interne de l'objet augmenté par le filtre. Les filtres peuvent, par exemple, modifier les messages en changeant leurs destinataires. Ils peuvent être utilisés pour rediriger les messages vers des objets internes, qui sont encapsulés dans l'interface, ou vers des objets externes référencés par la même interface. Un message ne peut être reçu par

un objet destinataire qu'après avoir passé tous les filtres d'entrée augmentant cet objet. De la même façon, un message ne peut être envoyé qu'après avoir passé tous les filtres de sortie appliqués à l'objet émettant ce message. Chaque filtre représente une préoccupation particulière. Les filtres sont regroupés (liés grâce à l'opérateur logique « Et ») et ordonnés dans des modules, permettant ainsi la composition de préoccupations transverses qui sont orthogonales. Dans le cas de préoccupations non orthogonales, un nouveau module de spécification de superposition est nécessaire pour spécifier la composition des différents filtres. Les détails de ce modèle sont explicités dans [Bergmans 94]. Ce modèle a été implémenté comme extension de quelques langages de programmation par objets existants, tels que C++ [Glandrup 95] ou Smalltalk [Mordhorst *et al.*, 95]. Les filtres de composition peuvent aussi être implémentés par des métaobjets, présents au moment de l'exécution et modifiables dynamiquement. Il existe plusieurs types de filtres de messages prédéfinis : les filtres de délégation (*delegation filters*) spécifiques à la délégation de messages, les filtres d'attente (*wait filters*) empilant les messages pour leur exécution, les filtres d'erreurs (*error filters*) pour la gestion d'exceptions, et plusieurs autres types qui peuvent être ajoutés.

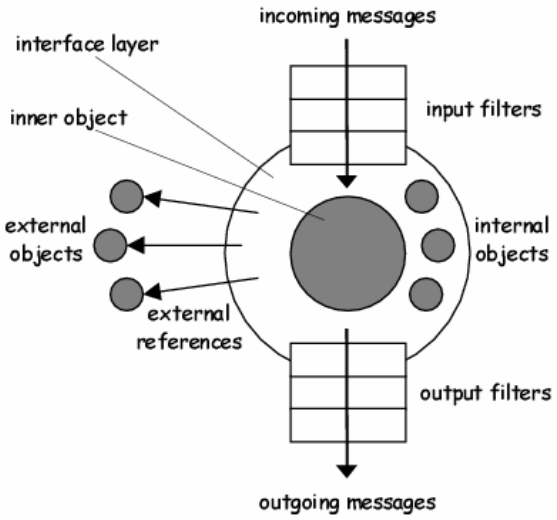


Figure 2.2. *Éléments d'un objet dans le modèle Composition Filters [Aksit* et al.*, 96]*

Il est à noter qu'il s'agit d'une technique relativement puissante, permettant une bonne représentation modulaire et séparée de différentes préoccupations, telles que la synchronisation [Bergmans 94], les contraintes du temps réel [Aksit *et al.*, 94], la transaction [Aksit *et al.*, 92], la gestion d'erreurs [Bergmans 94], etc. De manière générale, chaque préoccupation dont la définition est basée principalement sur l'interception de messages, et l'exécution de quelques actions avant ou après l'exécution des méthodes invoquées par ces messages, peut être représentée dans ce modèle.

3.5 Séparation multidimensionnelle des préoccupations (MDSoC) : Hyperspace et Hyper/J

Une autre approche parmi celles les plus connues est la séparation multidimensionnelle des préoccupations (*Multidimentional Separation of Concerns* ou *MDSoC*) [Tarr *et al.*, 99]. Cette approche est définie comme étant une extension de la programmation par sujets (cf. § 3.3). Le terme « multidimensionnelle » dénote une séparation des préoccupations qui implique une décomposition des programmes en plusieurs dimensions arbitraires, où chaque dimension regroupe un ensemble de préoccupations particulières. L'ensemble des classes d'une application constitue, par exemple, une dimension. Il s'agit de la dimension des préoccupations des classes. La séparation de l'ensemble des préoccupations d'une application tout au long de ces dimensions est simultanée : aucune des dimensions

identifiées n'est dominante. Les recouvrements et interactions entre préoccupations orthogonales des différentes dimensions sont déterminés et spécifiés par des règles d'intégration. Cette approche présente l'avantage de pouvoir supporter de nouvelles préoccupations et dimensions de préoccupations (non prévues a priori) d'une manière incrémentale, à chaque fois qu'elles se présentent tout au long du cycle de vie d'un système. Elle permet ainsi une re-modélisation à la demande, sans modifier l'application déjà développée. L'approche *Hyperspace* est une forme particulière de la séparation multidimensionnelle de préoccupations. Nous proposons dans ce qui suit de présenter les principes de base de cette approche (cf. § 3.5.1) ainsi que ceux de son implémentation Hyper/J (cf. § 3.5.2). Pour une meilleure compréhension et illustration de ces principes de base, nous reconsidérons l'exemple de l'éditeur graphique déjà introduit dans la section 2.1 de ce chapitre.

3.5.1 L'approche Hyperspace

L'approche *Hyperspace* [Ossher *et al.*, 99] permet la définition de diverses dimensions (*dimensions*) de préoccupations (*hyperslices*) utiles et nécessaires aux différentes phases du processus de développement d'un système. Chaque préoccupation est composée d'une ou de plusieurs unités intégrables (*integrable units*) qui sont organisées dans un hyperespace (*hyperspace*) (cf. figure 2.3).

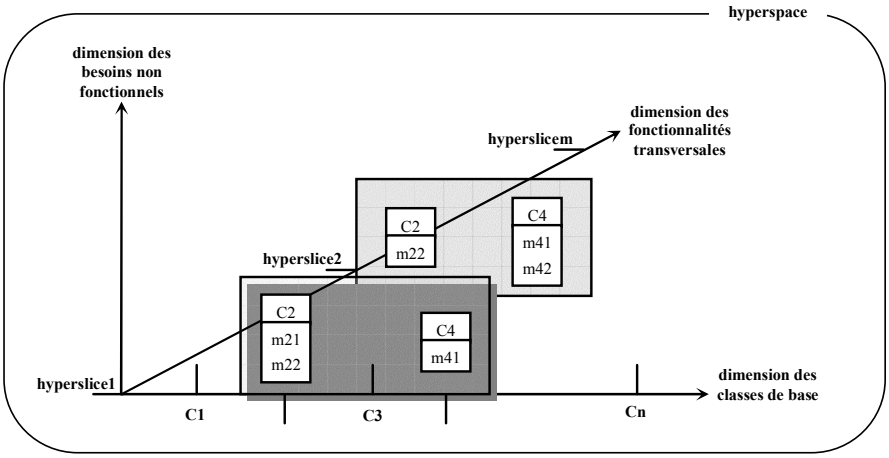


Figure 2.3. *Principes de l'approche Hyperspace*

Une unité intégrable est une construction du langage considéré, qui peut être, par exemple, une déclaration de variable, une méthode (m21, m42, etc.), une classe (C1 ... Cn), une interface, etc. On distingue les unités primitives, qui sont définies comme étant atomiques, des unités composées qui regroupent plusieurs autres unités (primitives ou composées). Les méthodes et attributs sont ainsi des unités primitives, alors que les classes ou paquetages sont des unités composées. Les unités d'une même préoccupation sont identifiées et regroupées dans un hyperslice (*hyperslice1* … *hyperslicem*), encapsulant la préoccupation considérée. Chaque unité appartient à un hyperslice, et chaque hyperslice appartient à une dimension donnée de l'hyperspace (dimensions des classes de base, des fonctionnalités transversales, etc.). Les unités des différents hyperslices sont destinées à être intégrées grâce à diverses relations de composition et règles d'intégration spécifiques, comme c'est le cas pour la programmation par sujets. Les relations de composition sont au nombre de trois ; chacune précise une manière particulière dont les unités de deux ou plusieurs hyperslices doivent être composées (fusion par noms, fusion mais pas par noms, redéfinition par noms). Plusieurs autres règles d'intégration spécifiques sont définies ; elles permettent de compléter et de préciser en détail la manière dont les unités intégrables des hyperslices à composer doivent être intégrées. Toutefois, la principale particularité de cette approche par rapport à l'approche *SOP*, est d'imposer la propriété de complétude aux hyperslices (correspondant aux sujets dans l'approche *SOP*). Dans ce modèle, tout hyperslice doit être complètement déclaré : il ne peut pas

référencer des unités non déclarées dans ce même hyperslice. Toutefois, un hyperslice ne requiert pas une définition complète de ses déclarations : une méthode peut être spécifiée sans pour autant être implémentée, par exemple (l'implémentation de cette méthode est ajoutée lors de la phase de composition ou d'intégration). Cette complétude élimine le couplage entre hyperslices. Pour comprendre et illustrer tous les principes de cette approche, nous considérons dans ce qui suit son implémentation Hyper/J.

3.5.2 Hyper/J

Hyper/J [Ossher *et al.*, 00], [Tarr *et al.*, 00] est une extension du langage Java, supportant la séparation multidimensionnelle des préoccupations. Il offre en plus des concepts fondamentaux du langage Java, de nouveaux mécanismes et concepts permettant de décomposer les programmes en hyperslices, et d'intégrer ceux-ci. L'ensemble des relations et règles utiles à l'intégration de deux ou plusieurs hyperslices est précisé dans un fichier de spécification contenant deux parties importantes, il s'agit du fichier correspondant à l'hypermodule (*hypermodule file*). La première partie spécifie explicitement les noms des hyperslices destinés à être composés. La deuxième partie identifie les relations composition de leurs unités intégrables. Elle spécifie les unités correspondantes des différents hyperslices considérés, et définit leurs règles d'intégration. Hyper/J considère comme unités intégrables les paquetages, les classes, les interfaces, les opérations, les méthodes et les attributs. Les hyperslices sont également considérés comme étant des unités intégrables. Les paquetages, classes, interfaces et hyperslices constituent des unités composites, alors que, les autres unités sont considérées comme étant primitives. Certaines règles de composition offertes par Hyper/J peuvent spécifier en même temps les correspondances entre unités, et la manière dont celles-ci doivent être intégrées. La relation *mergeByName* indique, par exemple, que les unités ayant les mêmes noms correspondent et doivent être fusionnées. D'autres règles sont utilisées uniquement pour indiquer les correspondances entre unités, alors que d'autres servent seulement à spécifier la manière dont ces unités seront intégrées. Le tableau 2.2 donne les trois règles de composition générales offertes par Hyper/J, ainsi que l'ensemble des règles d'intégration pouvant être utilisées conjointement avec celles-ci (si nécessaire). En effet, dans le cas où la règle de composition générale n'est pas applicable à une ou plusieurs unités intégrables, il est nécessaire d'indiquer ces exceptions grâce aux règles d'intégration.

Tableau 2.2. *Stratégies générales de composition et règles spécifiques d'intégration dans Hyper/J*

Stratégies générales de composition	
mergeByName	Indique que les unités intégrables de noms identiques, des différents hyperslices spécifiés dans le fichier de l'hypermodule, correspondent et doivent être fusionnées.
nonCorrespondingMerge	Indique que les unités intégrables de noms identiques des différents hyperslices à composer, ne correspondent pas et ne doivent pas être fusionnées.
overrideByName	Indique que les unités intégrables de noms identiques correspondent et doivent être redéfinies, par respect à l'ordre d'apparition des hyperslices spécifiés dans le fichier de l'hypermodule. Dans le cas de cette relation, la définition d'une méthode du premier hyperslice spécifié, remplace, par exemple, la définition de toutes les méthodes de nom identique des autres hyperslices.
Règles spécifiques d'intégration	
equate	Fait correspondre deux ou plusieurs unités de noms identiques ou différents.
order	Spécifie un ordre particulier entre unités devant être fusionnées.
rename	Permet de renommer une unité intégrable.
merge	Indique que les unités en paramètre de cette règle doivent être fusionnées.
override	Indique que les unités en paramètre de cette règle doivent être redéfinies, par respect à leur ordre d'apparition dans la règle. La définition de la première unité remplace celles du reste des unités considérées.
match	Comme *equate*, *match* fait correspondre deux ou plusieurs unités intégrables de nom identique ou non. Toutefois, il est possible d'utiliser des expressions régulières avec cette règle pour spécifier l'ensemble des unités correspondantes.

bracket	Indique qu'une ou plusieurs méthodes doivent être précédées et/ou suivies d'une autre méthode, augmentant leurs définitions.
summary	Spécifie la valeur finale de retour du résultat de fusion de plusieurs méthodes, pouvant chacune admettre une valeur de retour.
noMerge	Indique que deux ou plusieurs unités correspondantes ne doivent pas être fusionnées.

Pour toute application Hyper/J, l'ensemble des unités intégrables est spécifié tout d'abord, dans un fichier définissant l'hyperespace (*hyperspace specification file*). L'organisation de ces unités en dimensions et hyperslices, dans cet hyperespace, est spécifiée ensuite dans un troisième fichier (*concern mapping file*). Il s'agit d'identifier et de faire correspondre les différentes unités avec leur hyperslices correspondants, associés chacun à une dimension particulière de préoccupations. Pour illustrer l'utilisation d'Hyper/J, nous montrons dans ce qui suit comment nous pouvons réaliser l'éditeur de figures, présenté dans la section 2.1, dans l'approche Hyperspace.

Pour ce faire, il s'agit de regrouper dans un premier paquetage nommé baseApplication l'ensemble des classes de l'application, sans prendre en compte la préoccupation d'affichage et de mise à jour de l'affichage d'une figure par les classes Line et Point. Dans ce paquetage, les méthodes setX() et setY(), setP1() et setP2() et la méthode incrXY(), traitent exclusivement de la mise à jour des coordonnées des points et des lignes ; elles n'effectuent pas cependant de mise à jour de l'affichage de leurs figures correspondantes. Pour prendre en compte cette dernière préoccupation, il faut redéfinir dans un deuxième paquetage, nommé displayConcern, les nouvelles versions des classes Line et Point capables de mettre à jour l'affichage des figures à chaque fois que les coordonnées de leurs points et lignes changent. Il s'agit en fait de redéfinir l'ensemble des méthodes setX() et setY(), setP1() et setP2() et incrXY(), tout en les augmentant par le code nécessaire à la mise à jour de l'affichage. Le code suivant correspond au fichier de spécification de l'hyperespace de cette application.

```
hyperspace FigureEditor
        class baseApplication.* ,
        displayConcern.* ;
```

Pour mettre en correspondance l'ensemble des unités intégrables de cet espace (les deux paquetages, les classes et leurs propriétés) avec leurs préoccupations et dimensions de décomposition, il faut définir le fichier suivant.

```
concerns
        package baseApplication : Feature.Kernel
        package displayConcern : Feature.Display
```

La production du code final de l'application consiste à composer les deux hyperslices ainsi identifiés. Dans ce cas d'application simple, seule la relation de composition *overrideByName* est suffisante si les classes et leurs méthodes portent les mêmes noms dans les deux paquetages. Il faut faire correspondre par nom ces unités intégrables, et remplacer au moment de la composition les méthodes concernées par l'affichage et sa mise à jour dans le paquetage baseApplication, par leurs nouvelles définitions spécifiées dans le paquetage displayConcern : voir ci-dessous le code correspondant à cette composition. Il est à noter que l'ordre de spécification des hyperslices est important.

```
hypermodule DisplayFigure
        hyperslices :
                Feature.Display,
                Feature.Kernel,
        Relationships:
                overrideByName;
 end hypermodule;
```

Il s'agit ici d'une solution possible en Hyper/J, pour réaliser la préoccupation de mise à jour d'affichage. Une alternative à cette solution consiste à ne mettre dans la définition des méthodes setX() et setY(), setP1() et setP2() et incrXY() du paquetage displayConcern, que le code nécessaire à la mise à jour de l'affichage. Il faut ensuite utiliser la relation de composition générale *mergeByName* à la place de *overrideByName*, et d'inverser l'ordre de spécification des hyperslices dans le fichier de définition de l'hypermodule. Il est possible aussi dans ce deuxième cas de laisser *overrideByName* comme relation de composition générale. Toutefois, pour lever l'exception sur les méthodes concernées, il faut utiliser la règle d'intégration *bracket* pour chacune de ces méthodes.

3.6 Programmation par aspects (AOP) et AspectJ

La programmation par aspects (*Aspect-Oriented Programming* ou *AOP*) [Kiczales *et al.*, 97] propose un nouveau paradigme de programmation, dont les fondements ont été définis au centre de recherche de Xerox parc, à Palo Alto au milieu des années 90. Ce paradigme de programmation a émergé suite à différents travaux de recherche, dont l'objectif principal était d'améliorer la modularité des applications afin de faciliter leur évolution et leur réutilisation. Nous proposons dans ce qui suit d'introduire les principaux concepts de ce paradigme. Pour une meilleure compréhension de ceux-ci, nous considérons ensuite une implémentation de l'AOP : le langage AspectJ, dont nous illustrons l'usage à travers l'exemple d'éditeur graphique introduit dans la section 2.1. L'implémentation de l'AOP ne se résume pas en effet à AspectJ, ni au monde de l'Objet. Il existe de nombreuses propositions d'implémentation dont nous distinguons, par exemple, l'extension AspectC [Coady *et al.*, 01] du langage C permettant la programmation par aspects dans le cadre de l'approche procédurale et fonctionnelle. Dans le cadre de l'approche Objet, plusieurs autres propositions intéressantes ont vu le jour : AspectS [Hirschfeld 02] pour Smalltalk, AspectC++ [Spinczyk *et al.*, 02] pour C++, etc. Nous considérons à la fin de cette section une implémentation de type framework de l'AOP.

3.6.1 Principaux concepts de l'AOP

La programmation par aspects propose de structurer les programmes en les décomposant en aspects et classes. En effet, G. Kiczales *et al.* donnent une définition de l'AOP qui peut être appliquée à tout paradigme de programmation classique (i.e. approche procédurale ou fonctionnelle, approche Objet, etc.), et qui prône en général une décomposition des programmes en termes d'aspects et de composants. Un composant est défini comme étant une unité modulaire de décomposition primaire (ou une procédure généralisée [Kiczales *et al.*, 97]) qui peut correspondre à une procédure ou une fonction, ou encore à un objet ou un composant, etc. Toutefois, comme nous nous intéressons à l'utilisation conjointe des patrons de conception par objets et de l'AOP, nous considérons ici que composant est synonyme de classe. Un aspect est alors une unité modulaire de décomposition transversale au découpage de l'application en termes de classes. C'est l'élément de séparation et de modularisation des préoccupations transversales. Nous pouvons ainsi définir sur une classe, mêlant différentes préoccupations transversales, autant d'aspects que nécessaire pour y placer les bouts de codes qui se trouvaient enchevêtrés (cf. figure 2.4), et qui correspondent à ces diverses préoccupations. Chaque aspect permet, par ailleurs, de regrouper le code relatif à une préoccupation particulière qui se trouvait dispersée dans plusieurs classes. Un aspect peut donc affecter une ou de plusieurs classes en même temps. Les bouts de codes regroupés et encapsulés dans les aspects sont, en effet, destinés à augmenter ou à modifier les définitions des classes de base. Ils sont définis sous la forme de propriétés structurelles et comportementales tels que les attributs et les méthodes, mais aussi, sous la forme d'autres concepts nouvellement introduits par l'AOP tels que les perfectionnements (*Advice*), les introductions et les déclarations de liens de parentés (généralisation ou abstraction) entre les classes.

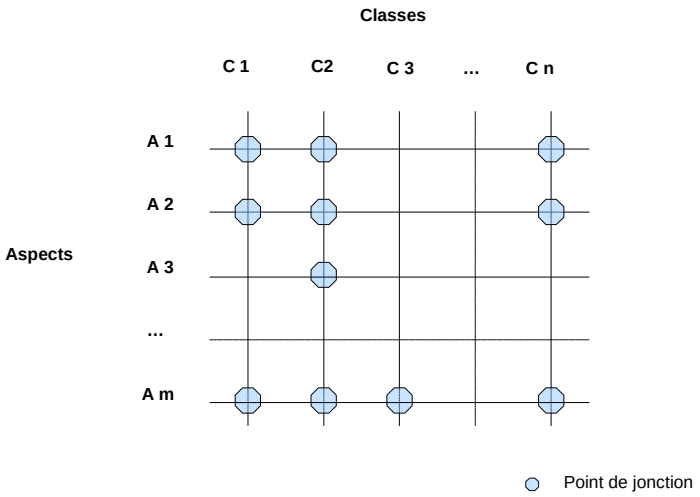


Figure 2.4. *Entrecroisement entre aspects et classes en AOP [Bardou 98]*

L'interaction entre classes et aspects affectant les comportements de base de celles-ci, est définie à l'aide de points de recouvrement (*Pointcuts*) qui sont construits sur la base de points de jonctions (*Join Points*) (cf. figure 2.4). Enfin, la composition des aspects et des classes, appelée tissage (*Weaving*) a généralement lieu lors de la compilation, par respect à ces points de jonction. Cette « composition retardée » offre, en diminuant le couplage entre aspects et classes, de nouvelles perspectives d'évolution et de réutilisation. Les problèmes d'enchevêtrement et de dispersion de code peuvent être résolus grâce à cette double décomposition. Pour appréhender le modèle de cette approche nous détaillons, ci-dessous, une de ses implémentations la plus connue et utilisée de nos jours, le langage AspectJ (dans sa version 1.1).

3.6.2 AspectJ

AspectJ [Kiczales *et al.*, 01] est aujourd'hui une référence pour l'AOP et un bon moyen pour illustrer son modèle de programmation. Il s'agit d'une extension du langage Java, développée à l'origine au Xerox Parc par G. Kiczales et son équipe, permettant de définir des aspects. AspectJ offre également un tisseur d'aspects permettant de composer le code des classes de base (écrit en java) avec celui relatif aux aspects, afin de générer le code de l'application finale. Étant une extension du langage Java, en plus des concepts fondamentaux de l'Objet (i.e. classe, interface, attribut, opération, méthode, association, généralisation…), AspectJ introduit de nouveaux concepts et mécanismes Aspect (*aspect*, *introduction*, *advice*, *pointcut*… [AspectJ 02]). Il s'agit d'un ensemble d'éléments de construction permettant, d'une part, de définir dans des aspects le code relatif aux préoccupations transversales (destinés à augmenter ou à modifier le code des classes de base), et de préciser, d'autre part, la manière (i.e. où, quand et comment) dont ce code sera composé avec celui des classes de l'application. AspectJ permet de définir deux types d'entrecroisement de préoccupations transversales avec les classes de base : entrecroisement statique (*static crosscutting*) et entrecroisement dynamique (*dynamic crosscutting*).

- Les entrecroisements statiques sont destinés à augmenter la définition des classes de base de l'application, tout en leur ajoutant de nouvelles propriétés structurelles et/ou comportementales. Ils permettent également de rajouter de nouvelles relations (des liens de parenté, des associations, etc.) entre classes. Dans ce cas d'entrecroisements, il est nécessaire de spécifier à chaque fois la ou les cibles d'impact (le « où » d'entrecroisement), en plus des propriétés ou des relations mises en jeu. Pour ce faire, AspectJ offre les concepts d'introduction (*introduction*) et de déclaration de relations inter-types (*declare parents*).

– Les entrecroisements dynamiques permettent la définition d'éléments destinés à modifier le comportement (les opérations et méthodes) des classes de base d'une application. Dans ce cas d'entrecroisement, il est nécessaire de pouvoir préciser la ou les cibles d'impact, mais aussi, la manière d'entrecroisement (le « quand » et le « comment »). Pour ce faire AspectJ offre les concepts de points de jonction (*join points*), de points de recouvrement (*pointcuts*) et de perfectionnements (*advices*).

Nous détaillons ci-dessous les concepts et mécanismes fondamentaux de ce langage, plus concrètement, à travers l'exemple d'éditeur de figures introduit dans la section 2.1.

Point de jonction (*Join Point*). Un point de jonction est un point précis du flot d'exécution des programmes, à partir duquel les aspects peuvent dynamiquement interagir avec les classes. L'appel à une méthode, la réception d'appel, le retour d'une méthode, l'accès à un attribut, etc. sont des exemples de points de jonction. Nous illustrons quelques-uns de ces points de jonction à travers notre exemple d'application. Supposons que nous avons à exécuter la partie de programme suivant [Kiczales *et al.*, 01].

```
Point P1 = new Point ( 0,  0 ) ;
Point P2 = new Point ( 4,  4 ) ;
Ligne Ln = new Ligne ( P1,   P2 ) ;
Ln.IncrXY ( 3,  6 ) ;
```

L'exécution des trois premières lignes de ce code crée les trois objets P1, P2 et Ln, représentés dans la figure 2.5 par les grands cercles. Dans cette figure, les petits rectangles (en cascade) des différents cercles représentent les méthodes offertes respectivement par P1, P2 et Ln.

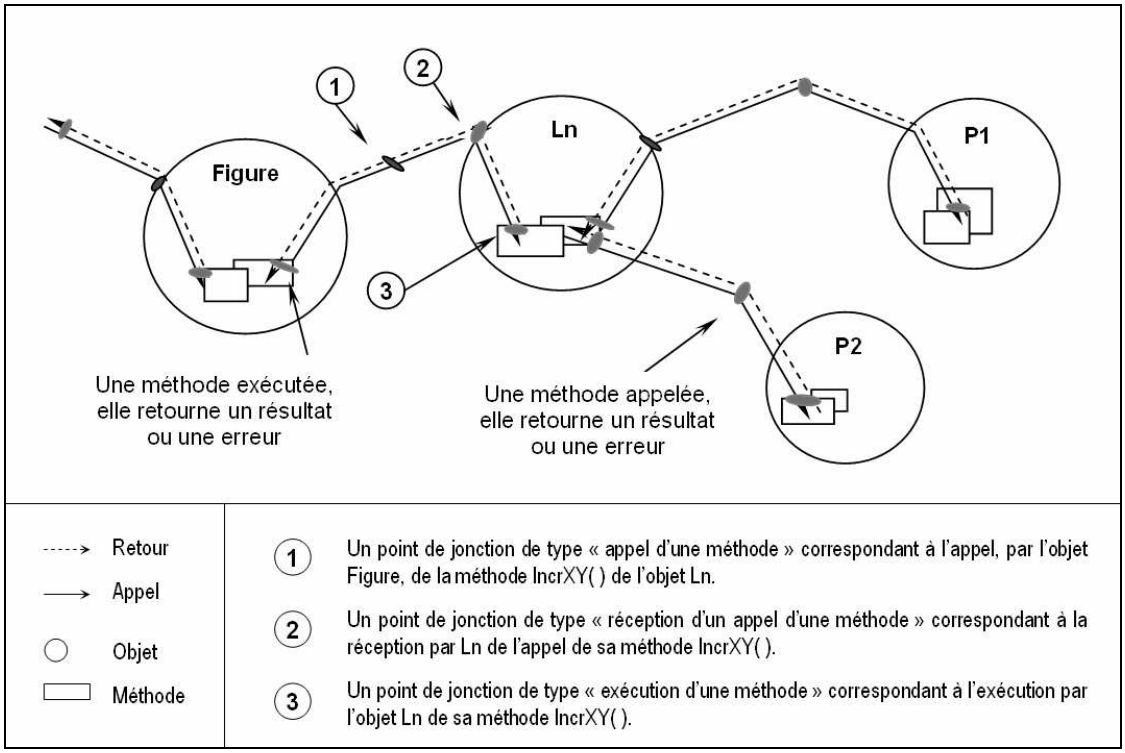


Figure 2.5. *Exemples de points de jonction (issues de [Kiczales* et al.*, 01])*

L'exécution de la dernière ligne de ce code enchaîne des appels et des exécutions de plusieurs méthodes, propres à la ligne Ln et ses deux points P1 et P2, et qui correspondent à une séquence de points de jonction, dont quelques-uns sont représentés dans la figure 2.5. Un point de jonction peut être un simple

appel à une méthode (1), une réception d'un appel à une méthode (2) ou encore une exécution d'une méthode (3) (appelé respectivement en anglais, *method call*, *method call reception* et *method execution join point*). Nous distinguons aussi d'autres types de points de jonction. La Figure 2.6 montre des points de jonction de type lecture ou écriture d'un attribut (appelés respectivement en anglais *field get* et *field set joint point*). Le tableau 2.3 résume les grandes classes des différents types de points de jonction qu'offre AspectJ dans sa version 1.1.

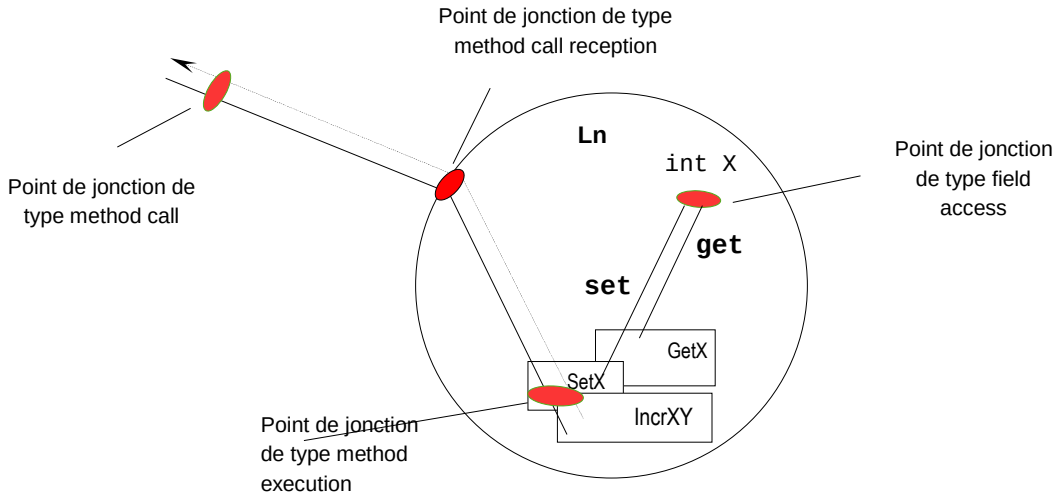


Figure 2.6. *Types de points de jonction*

Tableau 2.3. *Différents types de points de jonction définis dans AspectJ [Kiczales* et al.*, 01]*

Type de point de jonction	Description
Method call reception/execution	Réception d'un appel d'une méthode par un objet. Exécution d'une méthode d'un objet.
Constructor call reception/execution	Réception d'un appel de constructeur. Exécution d'un constructeur.
Object initialization	Initialisation d'un objet, à travers le constructeur de sa classe.
Class initialization	Initialisation des variables statiques d'une classe.
Field get/set	Accès en lecture/écriture à un attribut d'un objet ou d'une classe.
Exception handler execution	Interception d'exécution de gestion d'exception

Point de recouvrement (*Pointcut*). Un point de recouvrement (appelé également, coupe transversale) conditionne un entrecroisement dynamique entre un aspect et une ou plusieurs classes. Il s'agit d'un prédicat construit sur la base d'un ou plusieurs points de jonction, et éventuellement, d'autres paramètres précisant le contexte d'exécution. Il peut être nommé, et peut avoir des paramètres dans ce cas, ou anonyme. Tout point de jonction d'une coupe transversale est indiqué par un désignateur (*pointcut designator*), il s'agit des points de recouvrement primitifs. Dans le cas d'un point de recouvrement composé de plusieurs autres points de recouvrement primitifs, ou encore composés, les points de jonction considérés (respectivement, les points de recouvrement composés) sont combinés grâce aux opérateurs logique *and*, *or* et *not*. AspectJ offre plusieurs désignateurs permettant de définir les différents types de points de jonction distingués précédemment. Le tableau suivant (cf. tableau 2.4) résume l'ensemble de ces désignateurs.

Tableau 2.4. *Différents types de désignateurs définis dans AspectJ [Kiczales* et al.*, 01]*

Désignateurs de points de recouvrement	**Description**
call ()	Désigne l'appel à une ou plusieurs méthodes ou constructeurs.
execution ()	Désigne l'exécution d'une ou plusieurs méthodes ou constructeurs.
(Static)initialization ()	Désigne l'initialisation d'un objet ou d'une classe.
get ()	Désigne l'appel à une méthode accesseur, en mode lecture, d'un attribut.
set ()	Désigne l'appel à une méthode accesseur, en mode écriture, d'un attribut.
handler ()	Désigne une exception donnée

AspectJ offre plusieurs autres désignateurs (*within*, *withincode*, *this*, *target*, *args*, *cflow*, *cflowbelow*, *if*, *adviceexecution*, *preinitialization*) qui permettent de préciser et de conditionner encore plus l'entrecroisement entre les aspects et les classes. Le désignateur *target* permet de préciser, par exemple, l'objet ou la classe dont la méthode est invoquée. Il est possible d'utiliser des expressions régulières (avec le wildcards *) pour définir les coupes transverses. Voici quelques exemples de points de recouvrement.

```
call ( * Point.set* ( ) )
target (Point) && call ( int * ( ) )
call ( *  * ( .. )) && ( within (Line) || within (Point))
```

La figure 2.7 montre un autre exemple de point de recouvrement, nommé setter, sans paramètre. Il s'agit d'un point de recouvrement composé bâti sur un point de recouvrement primitif (target (Point)) et un autre composé (call (void setX(int)) | | call (void setY(int))).

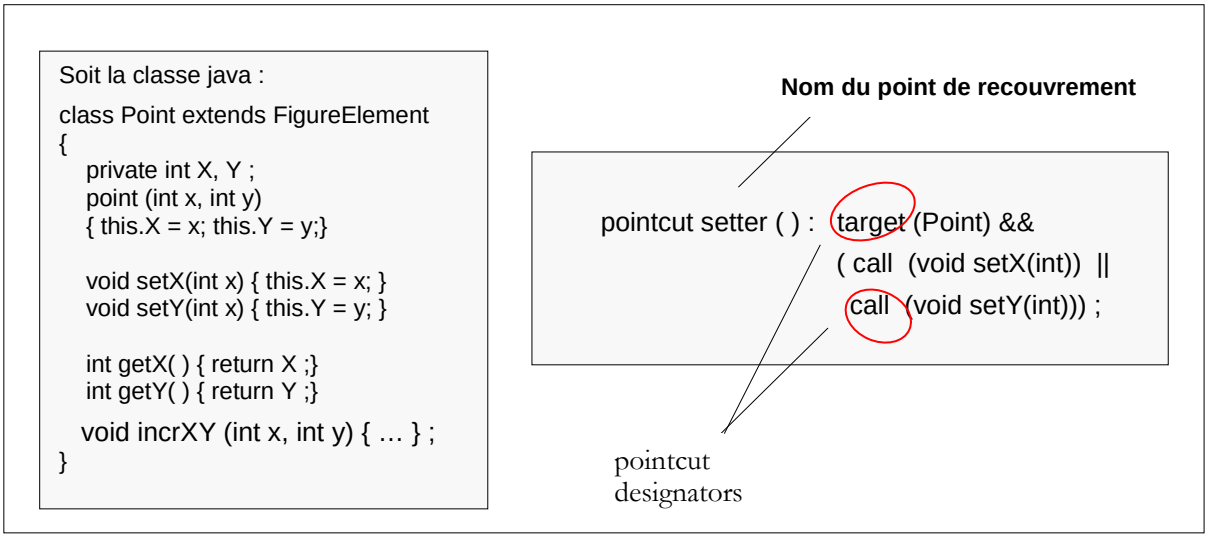


Figure 2.7. *Exemple de point de recouvrement composé*

Le code suivant reprend le même exemple de point de recouvrement, mais défini avec des paramètres précisant le contexte d'exécution. L'objet cible correspond à la valeur de la variable p, et la valeur du paramètre de la méthode affectée (setX ou setY) correspond à la valeur de la variable newVal.

```
pointcut setter (Point p, int newVal) :   target (p) && args (newval) &&
                                          ( call (void setX(int))  || call  (void setY(int))) ;
```

Perfectionnement (advice). Un perfectionnement est un fragment de code dont l'exécution est conditionnée par un point de recouvrement. Il peut être exécuté avant, après ou à la place du point de recouvrement qui lui est attaché (perfectionnements *before*, *after*, *after returning*, *after throwing* et *around*). Les perfectionnements de type *before* et *after* sont destinés à augmenter le comportement des classes de base. Deux variantes de perfectionnement de type *after* sont distinguées: *after returning* et *after throwing* qui s'exécutent, respectivement, après un retour sans échec d'une méthode ou avec échec (interruption due à une levée d'exception). Le code suivant montre un exemple de perfectionnement de type *after returning*, réalisant la mise à jour de l'affichage après l'exécution de l'une des méthodes setX() et setY(), setP1() et setP2() et incrXY().

```
pointcut move (FigureElement elem ) :
    target (elem) &&
    ( call ( void Line.setP1 (Point) )  ||
      call ( void Line.setP2 (Point) )    ||
      call ( void Point.setX (int) )        ||
      call ( void Point.setY (int) )        ||
      call ( void FigureElement.incrXY(int, int) ) ) ;

after (FigureElement elem ) returning : Deplacements ( elem ) {
    Display.update ( elem ) ; }
```

Les perfectionnements de type *around* sont plutôt destinés à remplacer le comportement des classes de base : ils substituent leurs codes aux codes originaux des propriétés comportementales qu'ils affectent. Toutefois, il peuvent éventuellement contrôler l'exécution du code original et décider de son achèvement, grâce à l'utilisation de *proceed()*, une forme spéciale d'invocation de méthode. En effet, l'utilisation de *proceed()* dans le corps d'un perfectionnement de type *around* permet d'exécuter le code original de la méthode affectée. Grâce à cette forme particulière d'invocation de méthode, il est possible aussi d'augmenter le comportement de n'importe quelle méthode d'une classe, avec des fragments de code destinés à être exécutés respectivement avant et après le code original de la méthode mise en jeu (invoqué par *proceed()*). En ce sens, il est possible de définir un perfectionnement de type *before* avec un perfectionnement de type *around*, spécifiant le code à exécuter avant la méthode affectée et faisant appel ensuite à *proceed()*. De la même façon, il est possible également de définir un perfectionnement de type *after* grâce à un perfectionnement de type *around*, avec un appel à *proceed()* avant la spécification du code destiné à être exécuté après cette méthode. Le code suivant montre un exemple de perfectionnement de type *around*.

```
around (  ) : call (Display.update () ) {
      if ( ! Display.disabled (  ) )
          proceed (  ) ; }
```

Un aspect peut admettre plusieurs perfectionnements définis sur un même point de recouvrement. Il est nécessaire dans ce cas de pouvoir spécifier l'ordre d'application de ces perfectionnements. Pour ce faire, AspectJ fixe les règles suivantes.

- Les perfectionnements de type *around* sont les premiers à être appliqués et exécutés dans leur ordre de déclaration dans l'aspect. Dans le cas où *proceed()* est invoquée, le prochain perfectionnement est exécuté. Le code qui suit *proceed()* est empilé pour être exécuté à la fin.
- Tous les perfectionnements de type *before* sont exécutés dans leur ordre de déclaration dans l'aspect. Ce qui est également valable pour les perfectionnements de type *after*.

Introduction. Une introduction est une définition dans un aspect d'une nouvelle propriété (attribut, opération ou méthode) destinée à étendre la définition d'une classe ou une interface. Le code suivant montre quelques exemples de déclarations d'introductions. Il s'agit d'ajouter un attribut name de type String, et deux méthodes setName() et getName() à la classe Point.

```
public String Point.name ;
public void Point.setName ( String name ) { this.name = name ; }
public String Point.getName ( ) { return name ; }
```

Déclaration d'héritage (*declare parents*). Une déclaration d'héritage permet de modifier la hiérarchie d'héritage existante en intervenant sur les ancêtres directs d'une classe, qu'ils soient des classes ou des interfaces. La ligne de code suivante indique, par exemple, que les classes Point et Line héritent de la classe GeometricObject.

```
declare parents : (Point || Line) extends GeometricObject ;
```

Aspect. Un aspect se définit en AspectJ d'une façon similaire aux classes Java : la définition d'un aspect peut contenir, comme la définition d'une classe, des déclarations d'attributs et des définitions de méthodes propres à l'aspect, mais aussi des définitions de points de recouvrement et de perfectionnements, des introductions et des déclarations d'héritage. Le code suivant montre un exemple d'aspect réalisant la préoccupation de mise à jour d'affichage. Un aspect peut être déclaré privilégié (*privileged*), ce qui lui donne accès aux propriétés privées ou protégées des classes avec lesquelles il interagit. Il peut de plus hériter d'un autre aspect ou d'une interface, comme il peut être abstrait. Il est intéressant en effet de faire des hiérarchies d'aspects, avec des aspects abstraits, pour la capitalisation et la réutilisation. Un aspect est abstrait s'il contient des déclarations de points de recouvrement abstraits (i.e. qui ne spécifient pas le ou les points de jonction correspondants), ou des déclarations d'opérations abstraites.

```
aspect UpdateDisplay
{
    pointcut move (FigureElement elem ) :
        target (elem) &&
       ( call ( void Line.setP1 (Point) )  ||
      call ( void Line.setP2 (Point) )    ||
      call ( void Point.setX (int) )      ||
      call ( void Point.setY (int) )      ||
      call ( void FigureElement.incrXY(int, int) ) ) ;

after (FigureElement elem ) returning : move ( elem ) {
    Display.update ( elem ) ; }
}
```

Outre la relation d'héritage entre aspects, nous distinguons de plus les relations de précédence et de dominance. Il est possible, en effet, que deux ou plusieurs aspects impactent une même classe au même point de jonction. Dans ce cas, il est nécessaire de préciser l'ordre d'exécution des perfectionnements des différents aspects, basés sur ce point de jonction. Pour ce faire, AspectJ définit la relation de dominance entre aspects et la relation de précédence, permettant d'ordonnancer leurs perfectionnements.

3.6.3 Une implémentation de type framework de l'AOP

Une autre proposition des plus abouties, implémentant *l'AOP*, est *JAC* (*Java Aspect Components*) [Pawlak *et al.*, 01], [Pawlak 02]. Contrairement à AspectJ et aux autres langages précédemment cités, *JAC* n'est pas un langage de programmation par aspects, mais un framework permettant la gestion dynamique des aspects d'une application à base de composants, écrite en Java. Dans ce framework, un aspect correspond tout simplement à un ensemble d'objets et aucune extension du langage Java n'est requise pour la définition de ceux-ci. Ils sont cependant définis par sous-typage ou extension des classes, et par implémentation des interfaces, fournies par le paquetage « core » de ce framework. *JAC* offre également une bibliothèque d'aspects prédéveloppés, pouvant être configurés pour répondre aux exigences d'une application particulière. Un programme écrit en *JAC* comprend en fait trois parties : les programmes d'aspects (*Aspect programs*), le tisseur et l'aspect de composition. Les programmes d'aspects implémentent l'ensemble des préoccupations transverses de l'application. Ils sont construits à partir d'un ensemble d'objets d'aspects (*aspect objets*) de différents types : des encapsulateurs (*wrapper*), des rôles ou des gestionnaires d'exceptions. Un encapsulateur permet d'intercepter n'importe quelle méthode d'un objet et d'exécuter des perfectionnements de type *before*, *after* ou *around*. Une même méthode d'un objet peut avoir plusieurs encapsulateurs, pouvant être ajoutés et supprimés dynamiquement au moment de l'exécution de l'application. Ceci fait l'avantage de *JAC* par rapport au reste des langages de programmation par aspects, qui ne permettent qu'un tissage statique de préoccupations. Un objet rôle correspond à une méthode destinée à être ajoutée dynamiquement à un ou plusieurs objets. Le tisseur (*weaver*) se charge de déployer les objets d'aspects sur les objets de base de l'application, par respect aux points de recouvrement définis dans un fichier de configuration. Enfin, l'aspect de composition (*composition aspect*) précise les règles de composition des programmes d'aspects. Le fonctionnement de JAC est basé sur Javassist [Chiba 95], qui permet d'instrumenter le code binaire des classes de base de l'application lors de leur chargement.

4. Bilan sur l'approche Aspect

Nous proposons dans ce qui suit de faire la synthèse des modèles de programmation par aspects détaillés ci-dessus, et d'expliciter les différents travaux actuellement menés dans le cadre de l'approche Aspect.

4.1 Synthèse des modèles de programmation par aspects

Dans la section 3 nous avons appréhendé les principaux modèles de programmation introduits dans le cadre de l'approche Aspect. Chacun de ces modèles propose sa propre solution et offre de nouveaux concepts et mécanismes, qui permettent une meilleure modularisation des préoccupations transversales vis-à-vis des approches classiques de développement. Ces concepts et mécanismes permettent, en fait, une décomposition et une recomposition efficaces des programmes, tout en gardant bien séparées et encapsulées les différentes préoccupations transversales ou de base d'une application ; ce qui garantit une meilleure lisibilité des programmes qui sont plus compréhensibles, et par conséquent, plus faciles à maintenir, à faire évoluer et à réutiliser. Ces concepts sont plus ou moins équivalents ; ils permettent tous, d'une manière ou d'une autre, de réaliser les mêmes types de préoccupations. D'ailleurs, [Elrad *et al.*, 01] confirme ce constat, à travers une discussion entre les fondateurs de la programmation par aspects, ceux de l'approche par filtres de composition et ceux de la séparation multidimensionnelle des préoccupations. Il convient de noter, en effet, que les modèles que nous considérons sont similaires et qu'ils se différencient et se caractérisent, cependant, par plusieurs autres critères qui sont les suivants :

- *Nature de la décomposition*. Les mécanismes et concepts proposés par chacun de ces paradigmes permettent-ils de spécifier les aspects d'une manière expressive ?
- *Composition des aspects*. Les modèles Aspect se différencient le plus souvent par leurs solutions d'intégration et d'identification des interactions entre aspects, tout en considérant les entrecroisements statiques et dynamiques. La solution proposée permet-elle de composer facilement les différents aspects, tout en maîtrisant la sémantique désirée ? En d'autres termes, la solution proposée permet-elle de gérer les conflits et les zones de recouvrement entre aspects ?
- *Généricité*. La solution proposée permet-elle de prendre en compte les différents types de préoccupations ?
- *Nature du tissage*. Le tissage des aspects est-il uniquement statique ? Est-il possible de retirer, ajouter ou remplacer les aspects d'une manière dynamique ?

Le tableau 2.5 caractérise chacune des approches considérées selon ces critères.

4.2 Travaux menés dans le cadre de l'approche Aspect

Depuis quelques années, l'intérêt de la communauté scientifique pour l'approche Aspect s'est accru de manière considérable. De nombreux travaux menés sur les aspects, couvrant tout le cycle de développement de systèmes d'information, ont ainsi vu le jour. Ces travaux traitent notamment de l'analyse des besoins par aspects (*Aspect-Oriented Requirement Analysis* ou *early aspects*), ou de la conception par aspects (*Aspect-Oriented Design* ou *AOD*), ou plus généralement du développement de logiciels par aspects (*Aspect-Oriented Software Development* ou *AOSD*). Ils considèrent, par exemple, l'extension ou encore la définition de nouveaux langages de modélisation par aspects (le chapitre 4 fait un état de l'art de quelques uns de ces travaux), pour mieux intégrer les aspects au niveau des phases de conception. D'autres travaux s'intéressent plutôt à la définition des fondements de cette approche Aspect. Nous pouvons citer, par exemple, les travaux de [Filman *et al.*, 00]. De tels travaux traitent aussi, le plus souvent, de la définition de ce qu'est un langage de programmation par aspects en général. Il convient de noter ici, que plusieurs ateliers (*workshops*) sont régulièrement organisés sur les thèmes cités précédemment, dans le cadre de la principale conférence sur le développement à base d'aspects (la conférence *AOSD*). Nous pouvons citer, par exemple, l'atelier *AOD*, l'atelier *Early Aspects* ou l'atelier *AOM* qui s'organise également dans le cadre de la conférence internationale sur *UML*. Parmi les autres principales pistes explorées autour de l'approche Aspect, nous pouvons citer également les travaux de ré-ingénierie qui proposent d'analyser les applications à base d'objets déjà existantes, pour extraire des aspects en vue de les ré-écrire dans une approche Aspect. Les travaux de [Ettinger *et al.*, 04] en sont un exemple. Cette technique vise à remanier certaines applications pour en améliorer la modularité, et donc en faciliter la maintenance et l'évolution et en augmenter les possibilités de réutilisation. D'autres travaux s'intéressent par ailleurs à la réutilisation des aspects mêmes. Ils essayent d'adapter différentes techniques pour rendre les aspects réutilisables. Nous pouvons notamment citer les travaux de [Walker *et al.*, 03] et [Douence *et al.*, 04]. Des propositions ont été faites aussi pour rendre le tissage des aspects dynamique et réversible, afin de pouvoir changer à l'exécution l'ensemble des aspects utilisés par une application [Ségura-Devillechaise *et al.*, 03], [Vasseur 04].

Par ailleurs, vu les limites des modèles de composants industriels ou académiques pour adresser la séparation et la modularisation de certaines préoccupations transversales, plusieurs travaux ont été menés sur l'utilisation conjointe de l'approche Aspect et différents modèles de composants déjà existants. Dans ce qui suit, nous passons en revue les principales propositions, qui considèrent l'utilisation conjointe des aspects et des modèles industriels et académiques de composants.

Tableau 2.5. *Caractérisation des modèles de programmation proposés dans le cadre de l'approche Aspect*

	Décomposition	**Composition**	**Généricité**	**Tissage**
Programmation adaptative (*AP*)	Sépare les algorithmes de leurs structures de données. Utilise l'indirection pour travailler sur les structures des classes, tout en gardant séparées les collaborations entre celles-ci. Elle offre une décomposition moyenne.	Utilise des directives de propagation, qui portent des informations sur les structures de données de l'application et la manière dont-elles doivent être parcourues par les algorithmes. Il s'agit d'une composition moyenne.	Bonne, mais restrictive	Statique
Programmation par points de vue (*ViewPoints*)	Découpe les programmes en vues d'utilisateurs (plusieurs vues pour une même classe). Les collaborations sont assurées à travers les différentes interfaces. Il s'agit d'une décomposition plutôt bonne.	Les classes des différentes vues définissent les objets et les interfaces des classes requises pour la collaboration. Elle offre une composition plutôt bonne.	Plutôt Bonne	Statique
Filtres de composition (*CF*)	Les filtres de composition supportent différentes vues sur les classes, tout en définissant ce qui doit être exécuté autour des messages échangés et qui peuvent ou non être acceptés. Il s'agit d'une bonne décomposition.	Les filtres sont attachés aux classes. Leur composition est définie par ordonnancement des filtres orthogonaux, et par spécification de superposition des filtres non orthogonaux. Il s'agit d'une composition bonne, mais un peu complexe à mettre en œuvre.	Bonne	Statique
SOP*, *MDSoC*, *Hyperspace	Décompose les programmes en hyperslices de même niveau, qui encapsulent différentes vues sur les classes de l'application. Elle offre ainsi une décomposition plutôt bonne.	Utilise des règles d'intégration et des stratégies générales de composition. Elle définit de plus, quelques règles d'intégration qui permettent de garantir une sémantique correcte pour le résultat. Elle offre ainsi une composition plutôt bonne, mais délicate à mettre en œuvre.	Très bonne	Statique
Programmation par aspects (*AOP*)	Décompose les programmes en classes et aspects. Il s'agit d'un découpage asymétrique, très bon en termes d'expressivité.	Utilise les règles de précédence entre perfectionnements d'un même aspect, et les relations de précédence et dominance entre aspects. Elle offre une composition très bonne, souvent restrictive.	Excellente	Statique, et dynamique dans le cas de *JAC*

- *Aspects et composants industriels*. Les modèles *EJB*, *CCM* et *.NET* supportent un certain nombre de préoccupations transversales pour le développement d'applications distribuées, notamment la persistance, les transactions et la sécurité, qui ne peuvent pas être complètement et proprement encapsulées dans des composants. En effet, intégrée le plus souvent dans les modèles sous forme de canevas, l'implémentation de telles préoccupations souffre de plusieurs autres déficiences. Cette technique de réalisation démontre bien le caractère « entrelacé » de tels services dans les composants. La programmation explicite de ces services au sein d'un composant (par héritage des classes du canevas) ne permet pas en effet de modulariser ces préoccupations transversales. D'autant plus qu'elle n'est pas suffisante pour enrichir les modèles de base. Ce mécanisme d'extension est en particulier limité à l'interface utilisée lors de la construction des composants. À part les restrictions inhérentes à ces modèles, les canevas ne fournissent aucun support pour la combinaison des différents services transverses entre eux. Afin d'associer à ces préoccupations une implémentation modulaire et plus flexible, et donc continuer à raisonner en terme de *SoC*, plusieurs extensions à base d'aspects telle que [Burke 04] ont été proposées pour le modèle EJB par exemple. [Cohen *et al.*, 04] présente une extension introduisant fortement les aspects au sein de *J2EE* ou *.NET* [Lowy 03], afin de pallier les lourdeurs et limitations de ce modèle.
- *Aspects et composants académiques*. Les modèles de composants académiques, ou architectures logicielles à base de composants, décrivent les systèmes comme un ensemble de composants qui communiquent entre eux par l'intermédiaire de connecteurs. Comme les modèles de composants industriels, leurs objectifs consistent à réduire les coûts de développement, à améliorer la qualité du logiciel, et enfin à augmenter la réutilisation des modèles. Ces architectures ont pour objectifs supplémentaires de faire partager des concepts communs aux utilisateurs de systèmes et à construire des systèmes hétérogènes à base de composants réutilisables sur étagères (*components-off-the-shelf*). Pour asseoir le développement de telles architectures, il est nécessaire de disposer de notions formelles et d'outils d'analyse de spécifications architecturales. Les langages de description d'architectures (*ADL* ou *Architecture Description Langage*) constituent une bonne réponse. En guise d'exemple, Fractal est une infrastructure générale de composition de logiciels qui supporte la programmation à base de composants [Bruneton 04], [Coupaye *et al.*, 03]. Elle prend en compte la définition des services et des besoins d'un composant, les composants composites, la description formelle d'architectures, les liens entre composants, les composants partagés, etc. Il s'agit d'un langage de description d'architectures (*ADL* ou *Architecture Description Langage*) basé sur XML qui peut être utilisé pour décrire les configurations de composants. Fractal favorise la réutilisation des éléments en utilisant les mécanismes d'héritage et de composition. Un composant dans Fractal peut hériter ou être composé de composants existants ce qui augmente le pouvoir de réutilisation. Toutefois, Fractal considère seulement les composants et leurs structures, et par conséquent en Fractal un système n'est rien d'autre qu'un groupe de composants reliés entre eux. Les interactions entre composants n'ont pas suscité beaucoup d'attention. En effet, dans Fractal les connecteurs sont définis implicitement et leur sémantique est enfouie au sein des composants. De plus, Fractal ne fournit aucun moyen de spécifier les propriétés non fonctionnelles (i.e. les préoccupations transversales) telles que la qualité du service ou la performance d'une application. Afin de s'affranchir de ces restrictions, des approches explorant l'extension de l'architecture de ce modèle de composant ont été proposées. Nous pouvons citer par exemple Fractal-AOP [Fakih *et al.*, 04], [Pessemier *et al.*, 04a], [Pessemier *et al.*, 04b].

Différentes approches ont ainsi été proposées pour étendre les modèles de composants industriels et académiques, tout en utilisant les aspects. Elles s'intéressent, le plus souvent, à l'extension des conteneurs ou d'autres éléments architecturaux de ces modèles (les connecteurs dans Fractal par exemple), pour la définition de nouveaux modèles plus expressifs. Nous pouvons citer par exemple AspectJEE [Cohen *et al.*, 04] et Caesar [Mezini *et al.*, 03], définis comme des extensions de modèles académiques. Ou encore, *JBossAOP* défini comme une extension du serveur d'applications libre *JBoss* [Burke 04] qui implémente le modèle *EJB*. Ces modèles et plusieurs autres sont présentés dans [Noyé *et al.*, 05], leurs principales caractéristiques sont également comparées. Il s'agit d'une comparaison représentative de l'état de l'art qui ne vise pas l'exhaustivité [Noyé *et al.*, 05]. En conclusion, l'intégration des aspects et des composants est en cours. L'attente est forte dans le domaine des infrastructures industrielles, pour lesquelles les aspects pourraient fournir des moyens souples et bien structurés pour l'intégration des services techniques et l'adaptation à des contextes d'exploitation spécifiques. Dans les propositions actuelles, aspects et composants restent toutefois des entités distinctes et les modèles d'aspects utilisés sont en général simples. L'étape suivante consisterait à développer des modèles de composants et d'aspects plus intégrés avec des capacités d'abstraction et de modularisation ainsi que des aspects plus expressifs, mettant de l'intelligence dans les interactions, etc.

Il reste enfin à noter que plusieurs travaux proposent aujourd'hui de combiner aspects, patrons et composants pour le développement de systèmes d'information, vue l'importance et l'efficacité de chacune de ces trois approches. Un atelier sur cette thématique s'organise également, tous les ans depuis 2002, dans le cadre de la conférence *AOSD*.

5. Conclusion

Ce chapitre présente le principe de séparation de préoccupations (*SoC*). Il passe en revue ensuite les principales approches de développement de systèmes appliquant ce principe. Nous avons discuté notamment, dans une première partie, certaines approches classiques : l'approche objet, la métaprogrammation, l'approche à base de composants, etc. Nous avons montré les limites présentées par de telles approches, vis-à-vis de l'implémentation séparée et modulaire de préoccupations dites transversales. Nous avons distingué aussi les différents types de préoccupations transversales et identifié les problèmes et conséquences liés à leur implémentation. Il s'agit des problèmes d'enchevêtrement et de dispersion du code. Sans remettre en cause l'utilité des fondements de ces approches traditionnelles, la section 3 de ce chapitre nous a permis de faire un état de l'art de plusieurs autres modèles de programmation (*SOP*, *Composition Filters*, *MDSoC* et *Hyperspace*, *AOP*, etc.) nouvellement introduits dans le cadre de l'approche Aspect. Ces nouveaux modèles permettent une meilleure décomposition et composition des programmes, pour une meilleure séparation des préoccupations transversales, et donc de meilleures évolutions et réutilisations des systèmes.

Comme le montre la section 4, l'engouement pour cette approche récemment apparue est grandissant. Les travaux de recherche menés sur les aspects sont de plus en plus nombreux. Dans le cadre de notre travail de recherche, nous nous intéressons plus particulièrement à l'utilisation conjointe de l'approche Aspect et des patrons de conception par objets dans l'ingénierie des SI. Un de nos principaux objectifs, dans ce travail, est de garantir une meilleure implémentation modulaire pour ces patrons, de manière à pouvoir tracer leurs imitations dans une application et améliorer par conséquent leur évolution et leur réutilisation. D'autres objectifs sont bien évidemment primordiaux : il est naturel de vouloir éliminer également les problèmes liés à leur utilisation (il s'agit des problèmes d'indirection, de confusion, de rupture d'encapsulation, et des problèmes liés à l'héritage, tous identifiés dans le premier chapitre (cf. § 3.3)). Nous proposons dans le chapitre suivant de réaliser les patrons de conception du GoF

[Gamma *et al.*, 95] en AspectJ et en Hyper/J. Il s'agit d'explorer les principaux apports et conséquences de l'utilisation de ces deux langages dans l'implémentation de patrons de conception par objets.

Chapitre 3

Réalisation de patrons de conception par objets à l'aide d'aspects

Les patrons de conception par objets constituent un moyen efficace pour la conception et la composition de plusieurs types de composants réutilisables, et la conception de grands systèmes complexes. Ils améliorent la qualité et la compréhension des programmes, facilitent leur évolution et augmentent notamment leur réutilisation. Toutefois, s'ils sont reconnus utiles, et si aujourd'hui de nombreux patrons de conception par objets ont été identifiés et proposés, les structures des solutions qu'ils proposent actuellement posent plusieurs problèmes et limites directement liés à la dispersion et à l'enchevêtrement du code de leurs imitations (cf. § 3.3 du chapitre 1). De telles structures sont définies en effet sous la forme de collaborations entre plusieurs classes et objets, des collaborations réalisées le plus souvent grâce aux techniques d'héritage, de composition et de délégation explicite. Leurs imitations sont en conséquence difficilement localisées dans le code final d'une application ; elles conduisent aux problèmes et limites d'utilisation des patrons.

Les langages et modèles de programmation, proposés dans le cadre de l'approche Aspect, offrent de nouveaux concepts et mécanismes permettant de réaliser les mêmes types d'adaptations qu'apportent les patrons de conception sur les classes et objets d'une application, tout en gardant bien encapsulées et séparées les définitions de ces adaptations. De tels concepts et mécanismes Aspect offrent, d'ailleurs, des alternatives quant aux techniques d'héritage, de composition et de délégation explicite, qui sont à l'origine des problèmes posés par les patrons. Ces mécanismes et concepts permettent, en ce sens, de nouvelles solutions plus efficaces pour les patrons de conception par objets, contribuant à améliorer leur traçabilité depuis la phase de conception jusqu'à la phase d'implémentation. Ceci devrait faciliter l'évolution et augmenter la réutilisation des patrons, mais aussi l'évolution et la réutilisation des applications sur lesquelles s'appliquent ces patrons. L'objectif de ce chapitre est d'identifier et d'étudier l'apport des concepts et mécanismes Aspect dans l'implémentation des patrons de conception par objets. Nous considérons, pour ce faire, le catalogue des 23 patrons de conception du *GoF* [Gamma *et al.,* 95] comme support de validation pour notre expérimentation. Nous proposons par ailleurs d'utiliser l'AOP (et le langage AspectJ) et l'approche Hyperspace (et le langage Hyper/J) pour l'implémentation de ces patrons dans une approche Aspect.

Ce chapitre est organisé en quatre sections. La section 1 motive l'utilisation de l'approche Aspect pour l'implémentation des patrons de conception par objets. Nous identifions et discutons ensuite, dans la section 2, les apports de cette nouvelle approche d'implémentation de ces patrons. Pour illustrer nos propos, nous considérons le cas d'étude introduit dans le chapitre 1 (cf. § 3.2). Nous développons les solutions en AspectJ et Hyper/J de quelques patrons appliqués sur ce cas d'étude. Nous présentons dans cette même section l'analyse générale de l'ensemble des nouvelles solutions par aspects, que nous proposons pour les 23 patrons du GoF dans les deux langages considérés. La section 3 traite des travaux connexes à notre proposition et présente une discussion de notre travail. La section 4 conclut enfin ce chapitre.

1. Motivations, objectifs et approche

Nous proposons dans cette partie de motiver, tout d'abord, l'utilisation de la technologie Aspect pour l'implémentation des patrons de conception par objets. Nous définissons ensuite nos objectifs et précisons l'approche que nous adoptons pour les réaliser.

1.1 Motivations

L'idée d'utiliser les nouveaux concepts et mécanismes, introduits dans le cadre de l'approche Aspect, pour l'implémentation des patrons de conception par objets, relève principalement de deux constatations que nous développons dans ce qui suit.

1.1.1 Patrons de conception *vs* préoccupations transversales

Les patrons de conception par objets consistent, le plus souvent, à établir une collaboration sur un ensemble dans un but donné. Il s'agit de buts qui correspondent à des préoccupations transversales particulières, adaptant de différentes façons les classes des applications sur lesquelles s'appliquent ces patrons. Les solutions structurelles des patrons sont définies pour favoriser de telles adaptations. En examinant l'ensemble des 23 patrons du *GoF*, nous avons pu identifier plusieurs types d'adaptations (ou de préoccupations) possibles, dont voici les principales :

- ajout d'une nouvelle interface ou une superclasse à une classe existante ;
- ajout de fonctionnalités (i.e. propriétés comportementales) à une classe existante ;
- redéfinition de la structure d'une classe existante, par ajout de nouvelles propriétés structurelles ;
- redéfinition des propriétés comportementales d'une classe existante ;
- redéfinition des propriétés des classes existantes, dans le but de mettre en œuvre tout un protocole de collaboration entre celles-ci.

Les solutions Objet actuellement proposées par les patrons de conception pour réaliser de telles adaptations sont insuffisantes, car elles ne garantissent pas la traçabilité des imitations des patrons au niveau de l'implémentation. Elles imposent de plus une structure aux programmes nuisant à la lisibilité et la compréhension du code final d'une application. En revanche, les mécanismes et concepts introduits dans le cadre de l'approche Aspect permettent une réalisation directe et facile de ces différents types d'adaptations, tout en gardant « propre » le code final d'une application. La qualité du code est induite par le fait que ces mécanismes et concepts permettent de garder le code relatif à de telles adaptations isolé et encapsulé dans des unités modulaires à part. L'approche Aspect offre ainsi de nouvelles alternatives quant à la réalisation des patrons de conception par objets, pour améliorer leur traçabilité et donc leur réutilisation.

1.1.2 Alternatives à l'héritage, la composition et la délégation explicite dans l'approche Aspect

La mise en œuvre des patrons de conception nécessite des techniques d'implémentation appropriées, pour une meilleure utilisation de ceux-ci dans le développement de systèmes de qualité. Les techniques fondées sur le paradigme Objet ne sont pas satisfaisantes. Elles se basent principalement sur le mécanisme d'héritage, sur la composition et sur la délégation explicite, qui posent plusieurs problèmes et limites (déjà détaillés dans la section 3.3 du chapitre 1). En revanche, les mécanismes et concepts introduits dans le cadre de l'approche Aspect offrent de nouvelles alternatives. Ils permettent ainsi de réaliser les mêmes adaptations attendues par les patrons, tout en palliant les problèmes et limites liés à l'utilisation de ces mécanismes et concepts Objet.

Il est possible, par exemple, d'établir une relation d'héritage entre deux classes sans avoir à les lier effectivement ; aucune dépendance directe n'est ainsi établie entre la super-classe et la classe fille. Celles-ci restent indépendantes et donc facilement réutilisables dans plusieurs autres contextes d'utilisation. De plus, les nouvelles définitions des propriétés de la super-classe enrichissant la sous-classe restent isolées de cette dernière classe qui reste inchangée. D'ailleurs, grâce aux mécanismes et concepts Aspect, il est possible de simuler l'héritage multiple même si le langage de base ne le permet pas (tel que Java, par exemple). À ce sujet, Hanenberg présente dans [Hanenberg *et al.*, 01] une discussion intéressante sur l'utilisation en général du mécanisme d'héritage dans le cadre de la programmation par aspects. Il montre notamment comment on peut réaliser une relation d'héritage simple, multiple ou mixte grâce aux concepts et mécanismes de l'*AOP*. Les mécanismes et concepts Aspect permettent par ailleurs de composer deux ou plusieurs classes, et de les faire collaborer de façon non destructive. Les définitions des classes mises en jeu restent en effet inchangées : aucun lien d'association ou de composition n'est réellement instauré entre

celles-ci. De plus, les adaptations requises pour la définition du protocole de collaboration des classes impactées sont découplées de ces classes, et restent en conséquence isolées. Les classes impactées se trouvent complètement indépendantes des contextes des patrons, et peuvent être réutilisées séparément dans plusieurs autres contextes d'application. D'ailleurs, cette façon de faire devrait permettre de pallier en conséquence les problèmes liés à l'utilisation de la délégation explicite (cf. § 3.3 du chapitre 1).

1.2 Objectifs et approche

Sur la base des deux constatations précédentes, nous sommes convaincus que l'approche Aspect permet de nouvelles solutions pour les patrons de conception par objets, tout en palliant leurs limites et problèmes. Nous avons choisi d'implémenter, à titre de validation, les patrons de conception du GoF [Gamma *et al.*, 95] à l'aide des mécanismes issus des langages de programmation relevant de la technologie Aspect. Nous avons choisi ce catalogue de patrons parce qu'il est l'un des plus connus et utilisés de nos jours. Dans notre approche du problème, nous considérons ce catalogue de patrons essentiellement comme un catalogue de problèmes récurrents en conception par objets plutôt que comme un catalogue de solutions. En effet, les solutions des patrons ne sont décrites dans [Gamma *et al.*, 95] qu'en termes de concepts Objet, et notre propos est de tirer parti des nouvelles possibilités offertes par les mécanismes et concepts Aspect. Nous retenons donc essentiellement l'intention et les indications d'utilisation des patrons auxquels nous nous intéressons. Nous proposons d'utiliser l'AOP et le langage AspectJ d'une part, et l'approche Hyperspace et le langage Hyper/J d'autre part. Notre choix a porté sur ces deux modèles et leurs langages respectifs AspectJ et Hyper/J, car il s'agit des deux modèles et langages les plus aboutis, connus, et utilisés de nos jours par les praticiens de l'approche Aspect. Il est d'autant plus intéressant de s'intéresser à ces deux modèles de programmation qu'ils ont été reconnus différents : Hyper/J est symétrique, tandis que AspectJ est asymétrique [Harrison *et al.*, 02]. Nous considérons ici que l'AOP s'applique sur le paradigme Objet et que « composant » est alors synonyme de « classe ».

2. Solutions par aspects des patrons de conception du GoF

Cette section présente le résultat de réalisation des 23 patrons de conception du GoF à l'aide d'AspectJ et de Hyper/J[5]. Nous proposons plus particulièrement de détailler dans ce qui suit les solutions de réalisation des patrons *Visiteur*, *Observateur* et *Singleton*, pour une meilleure compréhension de l'ensemble des nouvelles solutions par aspects que nous proposons ensuite pour l'ensemble des 23 patrons. De plus, afin de pouvoir montrer et expliquer les apports de ces nouvelles solutions, nous re-considérons le système SEE (déjà présenté dans le chapitre 1) sur lequel nous appliquons les solutions en AspectJ des trois patrons considérés. Nous revenons ensuite sur les solutions par aspects du reste de l'ensemble des 23 patrons du GoF pour les présenter brièvement. Nous proposons, enfin, de faire la synthèse de l'ensemble des solutions proposées, tout en précisant leurs apports et leurs propriétés caractéristiques, et en présentant une nouvelle classification des patrons du GoF. Une comparaison des solutions en AspectJ avec celles réalisées à l'aide de Hyper/J est également présentée dans cette synthèse.

Préambule : *notations utilisées pour la représentation graphique des nouvelles solutions par aspects des patrons du GoF*

Nous présentons ci-dessous les notations graphiques que nous avons initialement utilisées pour la représentation des nouvelles solutions par aspects des 23 patrons du GoF. Il s'agit de notations provisoires que nous avons définies uniquement pour ce travail d'implémentation. Par la suite, nous avons

[5] Nous ne détaillons pas l'ensemble des implémentations en AspectJ et Hyper/J proposées pour les 23 patrons du GoF, elles sont cependant toutes disponibles sur notre site web [Web Hachani].

proposé d'autres notations plus significatives, dans le cadre de la métamodélisation des concepts fondamentaux d'AspectJ et d'Hyper/J (cf. chapitre 4). Bien que nous disposions de ces nouvelles notations au moment de la rédaction de ce chapitre, nous avons tout de même choisi d'utiliser les notations provisoires. L'objectif étant, notamment, de montrer le besoin d'éléments de représentation plus appropriés aux deux langages considérés, et donc de motiver par là même le reste de notre travail de recherche.

— Pour la représentation graphique des solutions en AspectJ nous proposons d'adapter la notation de [Suzuki *et al.*, 99]. Il s'agit d'une extension d'UML [OMG-UML 04] pour les aspects. Dans cette notation (cf. figure 3.1), une relation de réalisation connectant une classe (ou une interface) à un aspect signifie que cet aspect entrecroise ou impacte cette classe (ou cette interface). Un aspect est représenté par un stéréotype particulier de classe : « aspect ». Nous pouvons aussi utiliser le stéréotype « privileged aspect » pour désigner un aspect privilégié.

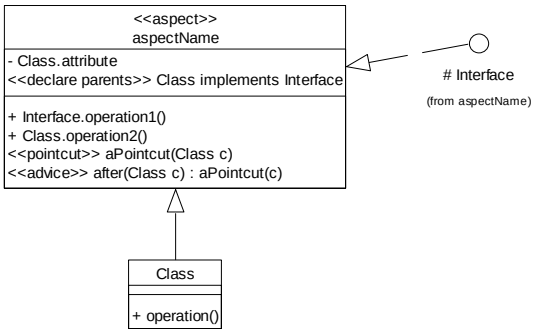


Figure 3.1. *Notation pour AspectJ*

Le premier compartiment d'un aspect peut contenir en plus des déclarations d'attributs propres à celui-ci, des déclarations d'introductions d'attributs dans une classe (ou une interface) qui sont représentées sous la forme class/interface.attribute. Il peut contenir également des déclarations de liens de parenté qui sont représentés sous la forme « declare parents » class/interface extends/implements class/interface. Le deuxième compartiment d'un aspect peut contenir des opérations propres à celui-ci, mais aussi des déclarations d'introductions d'opérations qui sont représentées sous la forme class/interface.operation(). Ce dernier compartiment peut contenir également des déclarations de points de recouvrement et des déclarations de perfectionnement qui sont représentés respectivement, sous la forme « pointcut » specification ou « advice » specification. Un aspect peut, par ailleurs, déclarer une interface ou une classe interne à celui-ci, qui peut être publique, privée ou protégée (cf. figure 3.1, # Interface (from aspectName)). Il reste enfin à noter que les aspects, les opérations introduites, ou les points de recouvrement abstraits sont représentés par une fonte italique en respect de la norme UML.

— Pour la représentation graphique des solutions en Hyper/J, nous nous basons exclusivement sur la notation d'UML. Nous proposons de représenter chaque hyperslice par exactement un paquetage (cf. figure 3.2-(a)), qui regroupe toutes les unités composables propres à celui-ci. Pour la spécification de l'hyperspace, des concerns et de l'hypermodule de composition spécifiques à une application donnée, nous nous servons par ailleurs d'exemples de codes en Hyper/J (cf. figure 3.2-(b)).

L'exemple de code suivant (b) consiste, par exemple, à augmenter la définition de l'opération operation() de la classe ClasseDeBase.Class de l'hyperslice Features.kernel par du code supplémentaire, spécifié par l'opération operation() de la classe Package.Class de l'hyperslice Features.unConcern. Ceci est possible grâce à la règle de composition générale mergeByName définie par l'hypermodule unHypermodule.

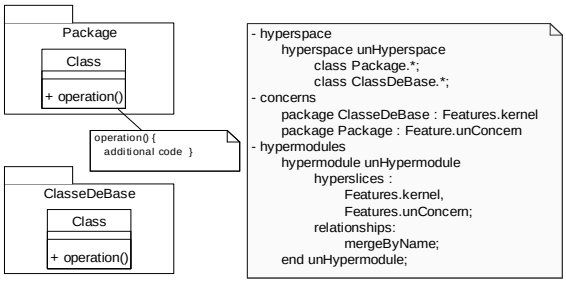


Figure 3.2. *Notation pour Hyper/J*

2.1 Exemples de réalisation et d'application des patrons Visiteur, Observateur *et* Singleton *en AspectJ et Hyper/J*

Il s'agit de montrer, dans ce qui suit, comment on peut améliorer les patrons Objet par l'utilisation des mécanismes et concepts Aspect, à travers les exemples des solutions en AspectJ et Hyper/J que nous proposons notamment pour les patrons *Visiteur*, *Observateur* et *Singleton*. Pour illustrer les solutions en AspectJ des trois patrons considérés nous reprenons, ensuite, le système SEE détaillé à la section 3.2 du chapitre 1.

2.1.1 Le patron *Visiteur* par aspects

Nous présentons et expliquons ci-dessous les solutions en AspectJ et en Hyper/J du patron *Visiteur*.

Réalisation du patron *Visiteur* à l'aide d'AspectJ. L'intention du patron *Visiteur* étant d'ajouter des comportements à une hiérarchie de classes d'éléments existantes, l'utilisation d'introductions d'AspectJ (cf. 3.6.2, chapitre 2) permet une réalisation simple et directe d'une telle préoccupation. Il s'agit d'ajouter statiquement des opérations à une classe, sans modifier la définition de celle-ci. Il est inutile donc de définir une hiérarchie de classes visiteurs, ni d'ajouter des opérations accept(), pour le moins artificielles, aux classes d'éléments (cf. tableau 1.7 du premier chapitre). La figure 3.3 ci-dessous montre la solution en AspectJ que nous proposons pour le patron *Visiteur*.

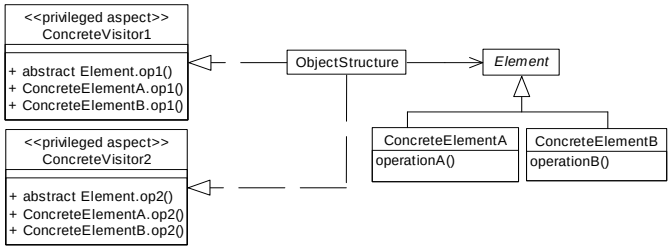


Figure 3.3. *Solution en AspectJ du patron* Visiteur

Les aspects ConcreteVisitor1 et ConcreteVisitor2 constituent le patron *Visiteur*, chacun d'eux contient les introductions nécessaires à l'ajout d'un comportement donné aux classes d'éléments. Il s'agit de définir une introduction pour chaque version d'opération devant être ajoutée. Ajouter un comportement supplémentaire 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 garantit sa traçabilité et facilite son évolution et sa réutilisation ; il en est de même pour le code propre à l'application.

La réalisation de *Visiteur* en AspectJ 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 solution et la structure initiale de *Visiteur*, le fait que chaque classe visiteur concrète 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, ce qui laisse de nombreuses possibilités quant à l'organisation des aspects jouant le rôle de visiteurs.

Réalisation du patron *Visiteur* à l'aide d'Hyper/J. La figure 3.4 montre la solution en Hyper/J que nous proposons pour le patron *Visiteur*, ainsi qu'un exemple de code correspondant au module de composition de deux imitations spécifiques de *Visiteur* avec les classes de la structure d'objets considérée.

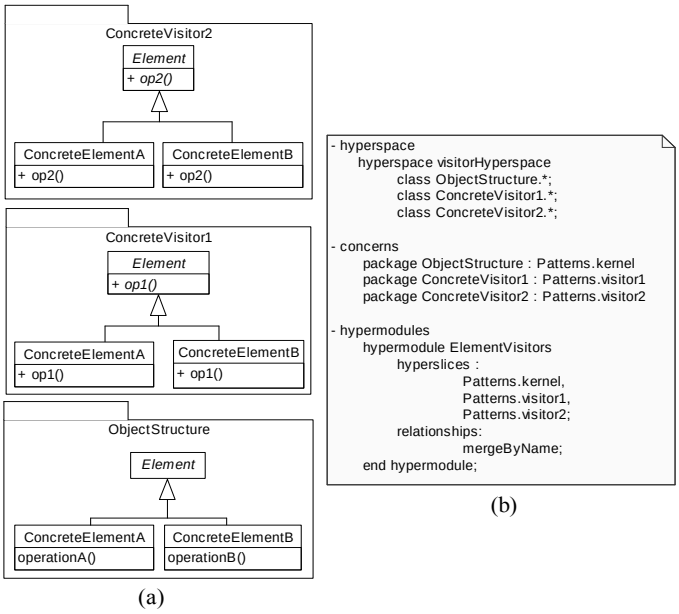


Figure 3.4. *Solution en Hyper/J du patron* Visiteur

Nous proposons de regrouper dans un premier hyperslice Patterns.kernel l'ensemble des classes de la structure d'objets considérée (Element et ConcreteElementk). Nous proposons par ailleurs de définir un hyperslice ConcreteVisitork par imitation spécifique de *Visiteur*, assurant ainsi la traçabilité de chacune de ces imitations (cf. figure 3.4-(a)). Un tel hyperslice déclare pour chaque classe d'éléments détenue par l'hyperslice Patterns.kernel une nouvelle classe correspondante ayant le même nom, pour reproduire enfin la même structure d'objets. Ces nouvelles classes déclarent et définissent une nouvelle opération (opk()) applicable aux éléments de la structure d'objets de base. Chaque hyperslice ConcreteVisitork est destiné à être composé avec l'hyperslice Patterns.kernel par respect de la règle de composition générale mergeByName, spécifiée par l'hypermodule ElementVisitors (cf. figure 3.4-(b)). Cette règle indique que toutes les classes de noms identiques appartenant aux différents hyperslices sont destinées à être fusionnées. Ainsi la classe résultat de la composition des différentes classes ConcreteElementA, par exemple, regroupe les opérations : operationA(), op1() et op2().

Cette solution en Hyper/J du patron *Visiteur* nous permet de garantir la traçabilité de chaque imitation spécifique à ce patron : une imitation par hyperslice. De plus, nous remarquons que la composition de chaque imitation de *Visiteur* avec les classes de l'application se fait de manière transparente à ces dernières. Leurs définitions se trouvent par conséquent inchangées. Ce qui facilite leur évolution et augmente leur réutilisation directe dans d'autres contextes d'application.

2.1.2 Le patron *Observateur* par aspects

Nous montrons et expliquons dans ce qui suit les solutions d'implémentation du patron *Observateur* à l'aide d'AspectJ et d'Hyper/J.

Réalisation du patron *Observateur* à l'aide d'AspectJ. L'intention du patron *Observateur* est d'assurer la cohérence entre un sujet, détenant une représentation interne de son état, et ses observateurs, qui en détiennent une représentation externe. Il permet ainsi de définir une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état interne, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour. Ce patron repose sur un protocole de communication qui reste le même pour toutes ses imitations. Ce protocole est défini de façon abstraite dans les classes abstraites de la structure originale proposée dans [Gamma *et al.*, 95] (cf. tableau 1.8). Le reste du patron *Observateur* doit par contre être défini de façon propre à chaque imitation. La figure 3.5 ci-dessous montre la solution en AspectJ que nous proposons pour ce patron, et présente un exemple de code correspondant.

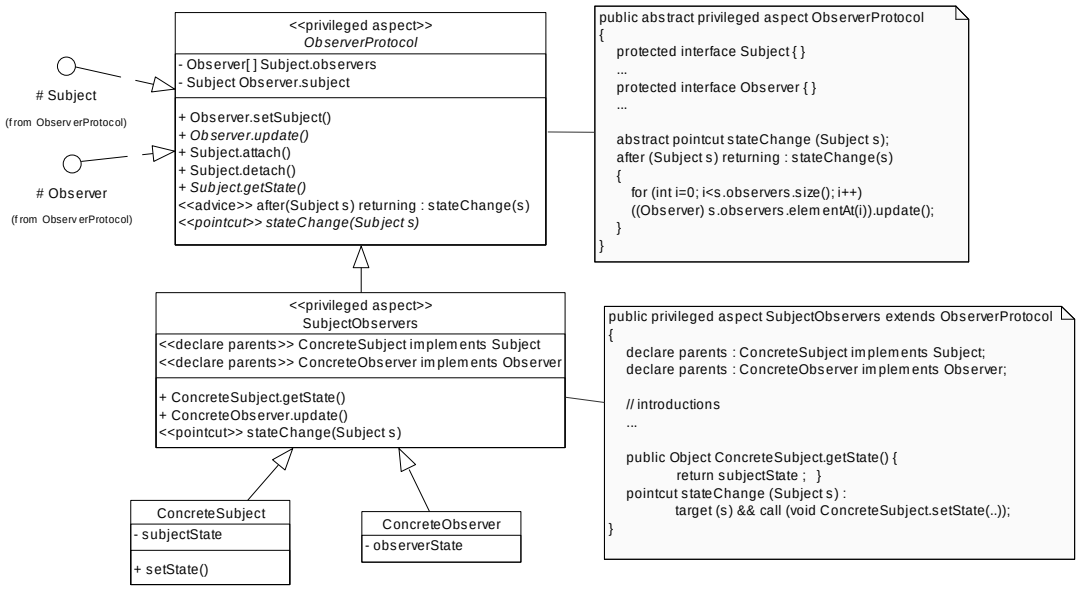


Figure 3.5. *Solution en AspectJ du patron* Observateur

Nous proposons d'isoler une partie abstraite générale dans un aspect abstrait noté ObserverProtocol, et une partie concrète particulière à chaque imitation dans un aspect concret noté SubjectObservers qui concrétise ObserverProtocol. Ces deux aspects sont destinés à être superposés aux classes de l'application, devant jouer un rôle de sujet ou d'observateur, telles qu'elles ont été définies avant l'application du patron. Le patron *Observateur* se trouve en conséquence parfaitement séparé du reste de l'application, ce qui garantit sa traçabilité et facilite son évolution et sa réutilisation ; il en est de même pour le code propre à l'application. L'aspect abstrait ObserverProtocol déclare deux interfaces internes protégées (Subject et Observer) et détient l'ensemble des introductions nécessaires, pour ces deux interfaces, spécifiant respectivement le comportement minimal des classes jouant le rôle de sujet ou d'observateur. Ces deux interfaces sont définies pour des raisons de typage, et sont déclarées comme étant protégées car non utilisées par les clients des classes jouant le rôle de sujet ou d'observateur. En effet, ces deux interfaces sont utiles uniquement dans le cadre des imitations d'*Observateur*, mais pas en dehors de ces cadres. Remarquons par ailleurs qu'il est possible d'associer à ces interfaces des définitions concrètes de méthodes et des déclarations d'attributs par le biais des introductions. Ces méthodes et attributs se trouveront, lors du tissage, héritées par les classes ConcreteSubject ou ConcreteObserver car celles-ci acquièrent, respectivement, le type Subject ou Observer par les déclarations d'héritage (*declare parents*) détenues par

l'aspect SubjectObservers (voir code de SubjectObservers). ObserverProtocol déclare de plus tous les éléments nécessaires au fonctionnement du protocole de communication de façon générale : il s'agit du point de recouvrement (*pointcut*) abstrait stateChange et du perfectionnement (*advice*) attaché à ce dernier. En effet, comme le montre l'exemple du code de la figure 3.5 le changement d'état est défini généralement dans l'aspect abstrait par stateChange qui conditionne l'exécution d'un perfectionnement de type *after* réalisant la notification aux observateurs. Ce point de recouvrement est destiné à être concrétisé de façon spécifique par chaque aspect concret dénotant une imitation du patron (voir code de SubjectObservers).

Réalisation du patron *Observateur* à l'aide d'Hyper/J. La figure 3.6 ci-dessous montre la solution en Hyper/J que nous proposons pour ce patron, et présente un exemple de code correspondant au module de composition d'une imitation de celui-ci avec les classes sur lesquelles il s'applique.

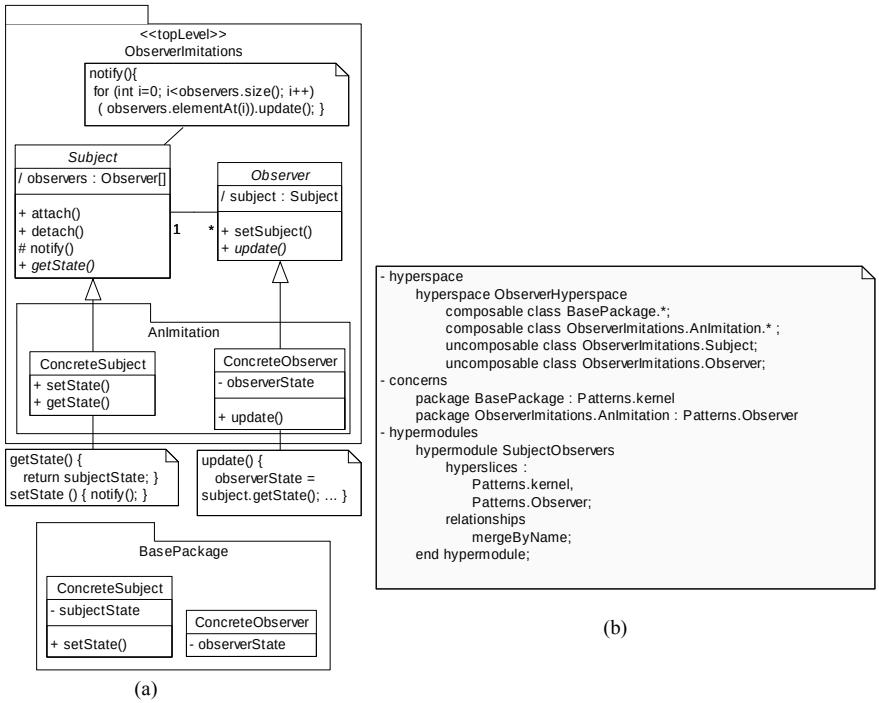


Figure 3.6. *Solution en Hyper/J du patron* Observateur

La solution que nous proposons préconise de définir deux hyperslices (Patterns.Observer et Patterns.Kernel, référençant l'ensemble des classes composables spécifiées par l'hyperspace ObserverHyperspace (cf. figure 3.6-(b)). Remarquons que les classes abstraites Subject et Observer du paquetage ObserverImitations ne sont pas destinées à être composées. Elles font toutefois partie de l'Hyperspace ObserverHyperspace car elles sont utilisées par les classes du sous-paquetage AnImitation du paquetage ObserverImitations (cf. figure 3.6-(a)). Patterns.Observer référence les classes ConcreteSubject et ConcreteObserver propres au paquetage AnImitation. Ces deux classes représentent une imitation particulière du patron *Observateur*, s'appliquant réellement sur les classes de base ConcreteSubject et ConcreteObserver du paquetage BasePackage. L'hyperslice Patterns.Kernel référence par ailleurs ces deux dernières classes de base, devant jouer respectivement le rôle de sujet et d'observateur, telles qu'elles ont été définies avant l'application de ce patron.

Notons que chacun de ces deux hyperslices correspond à une préoccupation (*concern*) particulière, appartenant à la dimension Patterns de l'hyperspace ObserverHyperspace. Les classes composables (et donc leurs opérations et attributs) appartenant à ces deux hyperslices sont destinées à être intégrées en respect des règles définies par le module de composition SubjectObservers (cf. figure 3.6-(b)). Il s'agit ici

tout simplement de la règle mergeByName spécifiant la stratégie de composition générale. En effet, cette règle est largement suffisante pour indiquer que les classes (respectivement, leurs opérations et attributs) de noms identiques correspondent et sont destinées à être composées par fusion. Ainsi, la classe ConcreteSubject du paquetage AnImitation est fusionnée, par exemple, avec la classe ConcreteSubject du paquetage BasePackage. Il en est de même pour leurs opérations spécifiques setState(). La classe résultat de cette composition regroupe donc l'ensemble des opérations de ces deux classes, et le code de son opération setState() est composé de ceux relatifs aux deux opérations spécifiques setState(). Il convient de noter ici que, conformément à l'ordre de déclaration des deux hyperslices considérés par l'hypermodule SubjectObservers (Pattern.Kernel puis Pattern.Observer), c'est le code de l'opération setState() du paquetage AnImitation qui vient augmenter (à la fin) celui relatif à l'opération setState() du paquetage BasePackage. Ainsi, à chaque fois que cette dernière opération est invoquée sur un sujet donné, elle fera la notification aux observateurs de cet objet après avoir modifié son état.

Les concepts et mécanismes offerts par Hyper/J nous ont permis de réaliser des imitations du patron *Observateur* de manière modulaire, assurant ainsi leur traçabilité. Toute imitation de ce patron se trouve, en effet, encapsulée dans exactement un paquetage (AnImitation et son hyperslice correspondant Pattern.Observer) et découplée du reste de l'application (BasePackage). Elle est composée d'ailleurs de manière transparente et non destructive avec les classes de l'application devant jouer le rôle de sujet ou d'observateur. Ce qui facilite l'évolution et augmente la réutilisation de ces classes dans d'autres contextes d'application. Qu'en est-il par ailleurs de la réutilisation d'*Observateur* ? Remarquons que la réalisation d'*Observateur* en Hyper/J que nous proposons préconise de prévoir un paquetage emboîté pour chaque imitation particulière de celui-ci, et un paquetage englobant pour l'ensemble des imitations de ce dernier. Ce paquetage englobant définit de manière abstraite le protocole de communication du patron *Observateur* qui reste le même pour toutes ses imitations, à travers les classes abstraites Subject et Observer qui peuvent être directement réutilisées dans plusieurs autres contextes d'application. Contrairement, les classes des paquetages représentant les imitations spécifiques sont moins réutilisables.

2.1.3 Le patron *Singleton* par aspects

Réalisation du patron *Singleton* à l'aide d'AspectJ. Le patron *Singleton* garantit qu'une classe n'a qu'une seule instance et fournit donc un point d'accès de type global à cette classe. La structure initiale (par objets) de ce patron contraint la classe impactée par le codage explicite, dans celle-ci, du comportement nécessaire à la réalisation de cette préoccupation (cf. tableau 1.10, chapitre 1). L'imitation du patron se trouve ainsi enchevêtrée avec le code de la classe contexte considérée, ce qui nuit fortement à la réutilisation directe de cette dernière dans un autre contexte d'utilisation qui ne nécessite pas l'application de *Singleton*. AspectJ permet, grâce à ses concepts d'introduction, de point de recouvrement et de perfectionnement de réaliser l'adaptation attendue du patron *Singleton*, de manière transparente à la classe contexte. La figure 3.7 montre la solution en AspectJ que nous proposons pour ce patron, et un exemple de code correspondant.

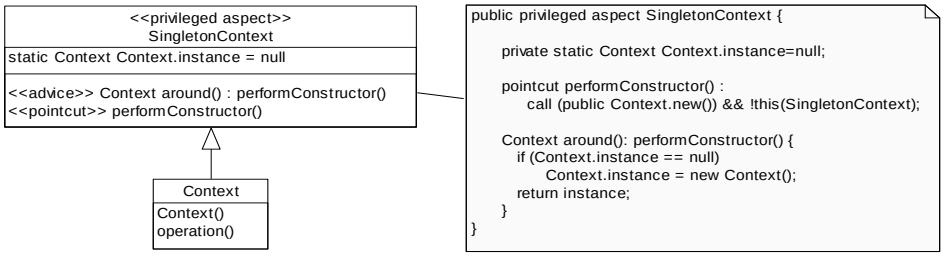


Figure 3.7. *Solution en AspectJ du patron* Singleton

La solution que nous proposons consiste à définir un aspect SingletonContext pour chaque imitation de *Singleton*. Cet aspect introduit dans la classe Context un attribut instance représentant l'instance unique de Context. Il détient de plus une déclaration d'un point de recouvrement performConstructor interceptant les requêtes demandant à créer de nouveaux objets de Context. Ce point de recouvrement conditionne l'exécution d'un perfectionnement de type *around* dont le code correspondant s'exécute à la place de celui du constructeur d'origine de Context, afin de créer son instance unique si elle n'existe pas et la retourner. La définition de la classe Context reste ainsi inchangée, et l'imitation du patron *Singleton* est isolée et encapsulée dans SingletonContext.

Cette solution pour le patron *Singleton* préconise de définir un aspect concret par imitation. Cette solution n'est pas réutilisable car étroitement liée au contexte d'application du patron *Singleton*. Remarquons qu'il est possible, toutefois, de proposer à ce patron une solution plus réutilisable qui propose de définir un aspect abstrait commun à toutes les imitations du patron *Singleton*. La figure 3.8 ci-dessous montre cette solution alternative. L'aspect AbstractSingleton déclare une interface interne (et protégée) Singleton indispensable pour le typage de toute classe contexte devant avoir une unique instance. Cet aspect abstrait détient également la déclaration d'une introduction dans Singleton de l'attribut instance spécifiant l'instance statique et unique de toute classe Context. Il déclare, par ailleurs, un point de recouvrement abstrait performConstructor qui conditionne l'exécution d'un perfectionnement de type around. performConstructor est destiné à être concrètement défini par l'aspect ContextSingleton qui concrétise AbstractSingleton, afin d'intercepter tout appel au constructeur de Context. Le code du perfectionnement associé à ce point de recouvrement s'exécutera ainsi à la place de celui du constructeur d'origine de contexte, afin de créer son instance unique si elle n'existe pas et de la retourner.

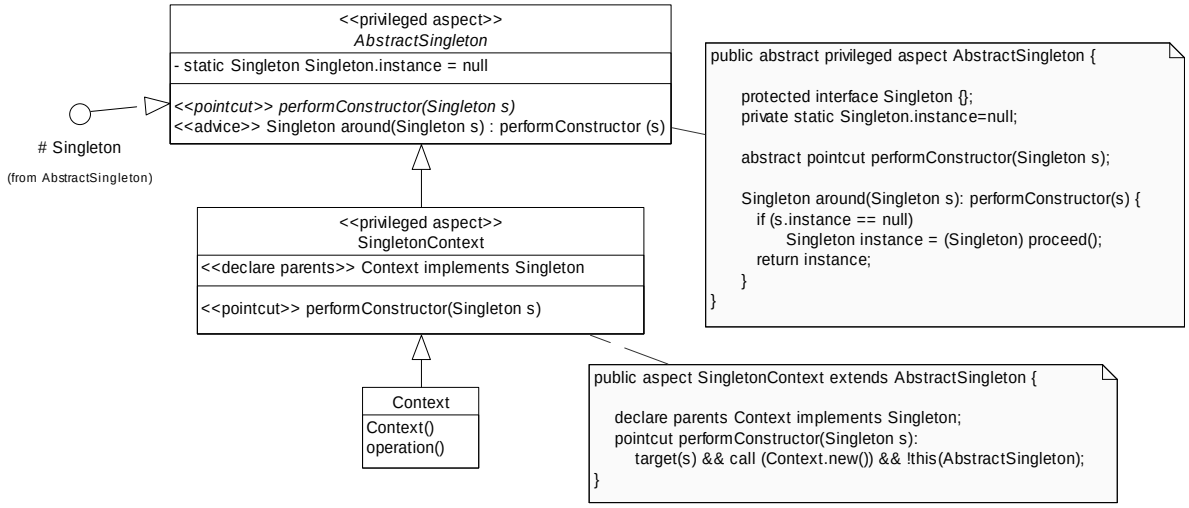


Figure 3.8. *Une solution alternative à la solution en AspectJ du patron* Singleton

Réalisation du patron *Singleton* à l'aide d'Hyper/J. Grâce à sa règle d'intégration *override*, Hyper/J nous permet de remplacer tout code d'une opération, y compris les constructeurs, par un autre code différent. Nous avons ainsi essayé de proposer pour le patron *Singleton*, à la manière de sa solution en AspectJ, une solution en Hyper/J qui consiste à remplacer le code du constructeur de Context. Une telle solution n'a pu être mise en œuvre car nous avons constaté qu'il est impossible de remplacer complètement le code d'un constructeur par un autre code différent, celui-ci peut uniquement être étendu. Ceci étant, une solution en Hyper/J possible pour le patron *Singleton* consiste à utiliser conjointement la stratégie de composition générale *mergeByName* et la règle d'intégration spécifique *summary functions*, permettant de déterminer la valeur unique à retourner parmi les deux résultats attendus du constructeur de

base et de celui qui le remplace. Nous n'avons pas toutefois pu mettre en œuvre cette solution, car à ce jour la fonctionnalité *summary function*s n'est pas complètement implémentée dans la dernière version disponible du langage Hyper/J [Tarr *et al.*, 00] que nous utilisons.

2.1.4 Exemple d'application et évaluation

Afin de montrer et expliquer les apports des nouvelles solutions proposées, nous reprenons ici la solution initiale du système SEE (cf. § 3.2.2 du chapitre 1) sur laquelle nous proposons d'appliquer les solutions en AspectJ[6] des trois patrons détaillés précédemment. La figure 3.9 rappelle le diagramme de classes de cette solution initiale. Nous rappelons aussi, succinctement, l'ensemble des besoins pour appliquer les trois patrons considérés :

- séparation et encapsulation de chacune des fonctionnalités d'évaluation, d'affichage et de vérification par application du patron *Visiteur* ;
- modularisation de la fonction de « logging » par l'utilisation du patron *Observateur* ;
- utilisation du patron *Singleton* pour garantir des instances uniques pour Logger et LogFile.

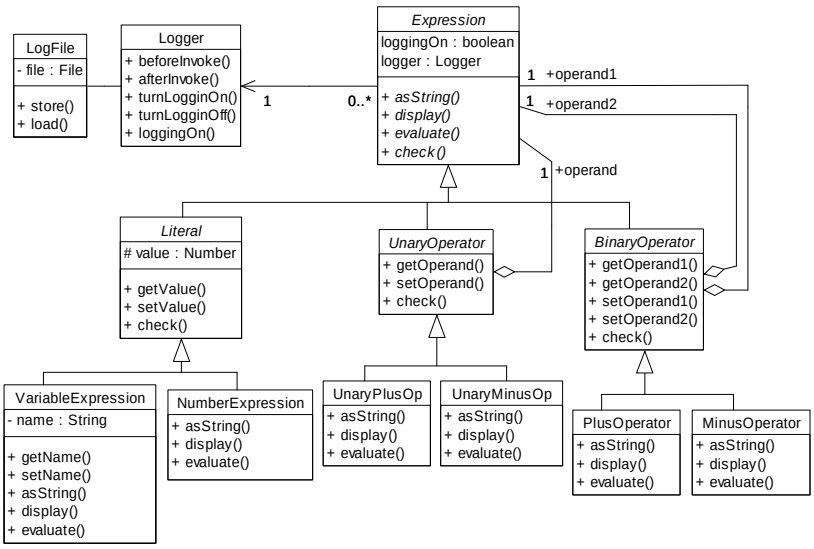


Figure 3.9. *Diagramme de classes du système SEE*

La figure 3.10 montre le résultat d'application des solutions en AspectJ des trois patrons considérés sur le système SEE. Nous expliquons ci-dessous comment les problèmes et les limites de ces patrons sont évités.

— La solution de réalisation à l'aide d'AspectJ du patron *Visiteur* résout les problèmes qu'il pose dans une approche strictement Objet (cf. tableau 1.11, chapitre 1). Le problème de confusion ne se pose plus, le code relatif à l'imitation du patron étant clairement séparé du code relatif aux classes d'expressions. En particulier, il n'est pas nécessaire d'ajouter une méthode accept() dans la hiérarchie des classes d'expressions sur laquelle s'applique *Visiteur*, ce qui présente de plus l'avantage de pouvoir appeler les comportements introduits par des noms plus explicites. 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 d'expressions sur laquelle s'applique le patron. Le problème de rupture d'encapsulation est

[6] Nous nous limitons ici à l'application des solutions en AspectJ uniquement parce qu'elles sont plus faciles à appréhender, et que la version actuelle d'Hyper/J ne permet pas une solution finie de *Singleton*.

aussi évité : la délégation n'est plus utilisée et, dans notre exemple, les différentes définitions des opérations d'évaluation, d'affichage ou de vérification sont directement exécutées dans le contexte de leurs classes d'expressions appropriées. Il reste à noter enfin que les définitions des classes d'expressions sont inchangées, et que la définition de chacune des trois fonctionnalités est séparée et encapsulée dans un seul aspect concret (EvaluateExpression, CheckExpression et DisplayExpression, selon la fonctionnalité considérée). Le code de l'application devient donc plus lisible et compréhensible. L'évolution et la réutilisation de l'imitation du patron, mais aussi de l'ensemble des classes d'expressions du système SEE sont par conséquent plus faciles.

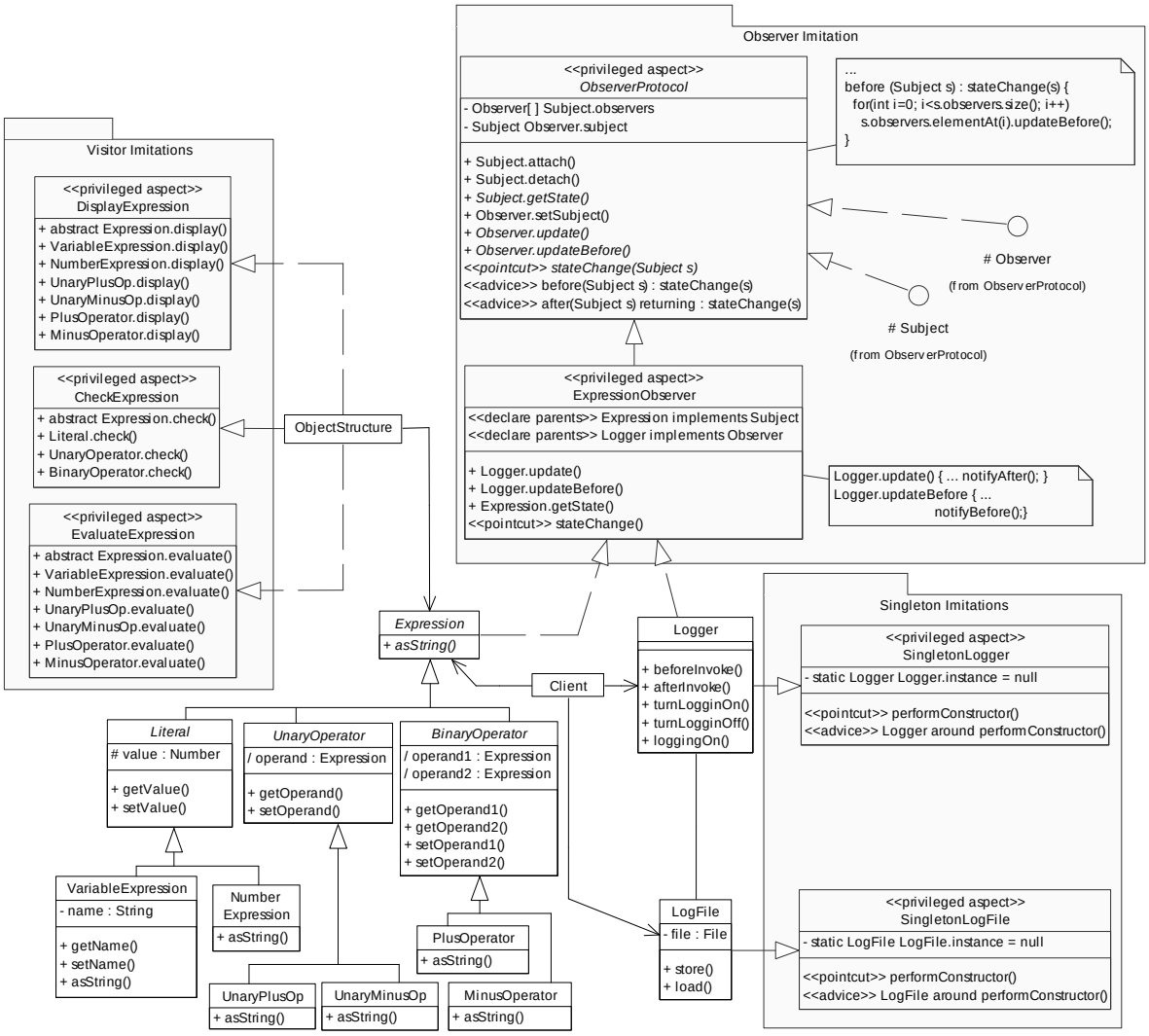


Figure 3.10. *Application des solutions en AspectJ des patrons* Visiteur, Observateur *et* Singleton *sur le système SEE*

— De la même façon, les problèmes posés par l'utilisation de la structure par objets d'*Observateur* (cf. tableau 1.11, chapitre 1) ne se posent plus dans le cadre de sa nouvelle solution par aspects. Il n'y a pas de confusion : le code propre à l'imitation de ce patron est localisé dans un aspect concret (ExpressionObserver) et un aspect abstrait (ObserverProtocol) détenant toutes les définitions générales à toutes les imitations de ce patron. Il est à noter également que les définitions des classes Expression et Logger sont inchangées. Le code de l'application devient donc plus lisible et plus facile à faire évoluer. L'utilisation d'un point de recouvrement et d'un perfectionnement limite d'ailleurs l'indirection lors de la notification, et toute expression n'a pas besoin de rendre publique l'accès à son état interne du moment où l'aspect ExpressionObserver est privilégié. On pourrait penser ici qu'il y a une rupture de l'encapsulation

des classes d'expressions. Cette levée d'encapsulation n'est, toutefois, effective que dans le strict cadre du patron *Observateur* et non pour l'ensemble du système SEE. Enfin, les liens d'héritage sont ajoutés de manière transparente par l'aspect concret dénotant l'imitation du patron, ce qui ne contraint en rien la hiérarchie d'héritage des classes d'expressions dont les définitions sont inchangées. Il est à remarquer par conséquent que la réutilisation de l'imitation du patron, mais aussi celle des classes sur lesquelles s'applique cette imitation, est augmentée.

— Le patron *Singleton* est plus particulièrement concerné par le problème de confusion. Ce problème ne se pose plus du moment où les définitions des classes Logger et LogFile sont inchangées, et que le code propre à chacune des deux imitations de ce patron est séparé et encapsulé dans les aspects SingletonLogger et SingletonLogFile. En effet, l'introduction des références vers les instances uniques de ces deux classes, ainsi que la redéfinition de leurs constructeurs initiaux, sont transparentes à ces dernières. Ces classes peuvent d'ailleurs être directement réutilisées dans d'autres contextes d'application ne réclamant pas l'utilisation du patron *Singleton*.

En conclusion, comme nous l'avons montré à travers l'exemple d'application précédent, nous remarquons que les nouvelles solutions proposées permettent de résoudre tous les problèmes d'utilisation des patrons (problèmes de rupture d'encapsulation, problème d'indirection et de surcharge de l'implémentation, problème de confusion et de traçabilité, et les problèmes liés à l'héritage). De plus, par comparaison du diagramme présenté par la figure 3.10 et de celui montré par la figure 1.11 du chapitre 1, on peut noter que ces nouvelles solutions garantissent une meilleure traçabilité des imitations des patrons. Elles permettent ainsi de repousser l'ensemble des limites d'application des patrons dans une approche strictement Objet. En effet, les nouvelles solutions par aspects des patrons augmentent la lisibilité des codes des applications, et facilitent leur évolution et leur réutilisation.

2.2 Application aux vingt autres patrons du GoF

Nous avons considéré l'ensemble des 23 patrons de conception du GoF, pour lesquels nous avons proposé des solutions de réalisation en AspectJ et Hyper/J. Nous décrivons dans ce qui suit, à titre de validation, les solutions de réalisation en AspectJ des vingt patrons restant, afin de montrer leur faisabilité et les apports. Nous avons remarqué que ces solutions présentent des propriétés caractéristiques communes, nous permettant de la sorte de les rassembler dans des groupes de patrons. Nous présentons ces groupes, tout en expliquant pour chacun d'eux, un ou deux exemples d'implémentation de patrons types. Nous nous limitons à la présentation des solutions AspectJ dans cette section parce qu'elles nous semblent plus faciles à appréhender que celles en Hyper/J[7].

2.2.1 Patrons *Adaptateur*, *Interpréteur*, *Décorateur* et *Procuration*

Les patrons *Adaptateur*, *Interpréteur*, *Décorateur* et *Procuration* sont directement réalisables dans une approche Aspect. Nous remarquons que leurs nouvelles solutions se basent essentiellement sur les mécanismes et concepts introduits dans le cadre des deux langages considérés.

Réalisation des patrons* Adaptateur *et* Interpréteur *à l'aide d'AspectJ. Comme dans le cas du patron *Visiteur*, les patrons *Adaptateur* et *Interpréteur* sont directement implémentés grâce aux mécanismes Aspect d'extension d'interfaces (i.e. *Parents declaration* et *Introduction* en AspectJ, et les relations *mergeByName* et *merge* dans Hyper/J). Nous expliquons ci-dessous les réalisations en AspectJ de ces deux patrons.

[7] La totalité des implémentations, en AspectJ et en Hyper/J, que nous proposons pour l'ensemble des patrons du GoF, est disponible sur notre site web [Web Hachani].

- Le patron *Interpréteur* propose pour un langage particulier, représenté en général par des classes d'expressions données sous la forme d'un *Composite*, de définir un interpréteur utilisant la représentation de ce langage afin d'interpréter ses phrases [Gamma *et al.*, 95]. La solution en AspectJ que nous proposons pour ce patron consiste, tout simplement, à la manière de la solution proposée pour *Visiteur*, à ajouter de manière transparente à chacune des classes d'expressions une méthode d'interprétation adéquate tout en utilisant le concept d'*introduction*. Un seul aspect est donc nécessaire, il détient toutes les variantes requises de l'opération d'interprétation. Celui-ci étend, par ailleurs, la définition de la classe Context contenant les informations à caractère global pour l'interpréteur (cf. structure par objet d'*Interpréteur* dans [Gamma *et al.*, 95]) par l'ajout de propriétés (attributs et opérations) nécessaires à la production du résultat d'interprétation.

- le patron *Adaptateur* propose de convertir l'interface d'une classe déjà existante en une autre conforme à l'attente du client. Il permet ainsi de collaborer à des classes, qui n'auraient pu le faire du fait d'interfaces incompatibles [Gamma *et al.*, 95]. La figure 3.11 montre la solution en AspectJ que nous proposons pour ce patron. L'utilisation du concept d'*introduction* offert par ce langage permet de doter directement la classe impactée Adaptee par une nouvelle interface conforme à celle de Target (interface requise par le client), tout en lui ajoutant les opérations nécessaires (request()). Chacune des opérations nouvellement introduites fait, tout simplement, appel à son opération correspondante spécifique à la classe impactée (specificRequest()).

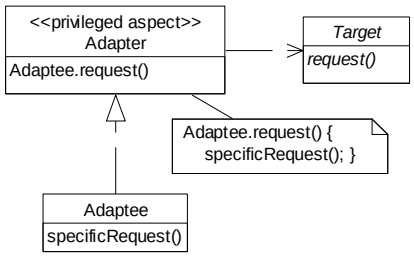


Figure 3.11. *Solution en AspectJ du patron* Adaptateur

Comme nous l'avons montré et expliqué pour la solution en AspectJ du patron *Visiteur* (cf. 2.1.4), les nouvelles solutions proposées pour *Adaptateur* et *Interpréteur* permettent des imitations de patrons modulaires et séparées, et plus traçables. De plus, les classes adaptées par le patron le sont d'une manière transparente, et s'en trouvent donc inchangées ; elles sont plus faciles à faire évoluer et à réutiliser.

Réalisation des patrons* Décorateur *et* Procuration *à l'aide d'AspectJ. Comme *Singleton*, les patrons *Décorateur* et *Procuration* sont, par ailleurs, facilement et directement réalisables grâce notamment aux concepts et mécanismes d'adaptation de comportement (i.e. *Advice* en AspectJ, et les relations *overrideByName*, *bracket*, etc. en Hyper/J). Nous expliquons ci-dessous les solutions en AspectJ que nous proposons pour ces deux patrons.

- Le patron *Décorateur* permet d'attacher dynamiquement des responsabilités supplémentaires à un objet, offrant ainsi une alternative souple à la dérivation de classes pour étendre les fonctionnalités de celles-ci [Gamma *et al.*, 95]. Pour ce faire, en AspectJ, nous proposons de définir un aspect pour chaque fonctionnalité ajoutée. Toute fonctionnalité supplémentaire est destinée à être superposée à une opération initiale (operation(), dans la structure par objets de *Décorateur*) de la classe impactée grâce à l'utilisation d'un point de recouvrement (PerformOperation), qui conditionne l'exécution d'un perfectionnement réalisant cette fonctionnalité supplémentaire. Un tel perfectionnement peut être de type *before*, *after* ou *around* avec un appel à *proceed*() afin de pouvoir effectuer des opérations supplémentaires avant et après l'exécution de l'opération

d'origine. Nous avons décidé de définir un aspect par fonctionnalité dans un souci de conserver l'un des avantages de la structure initiale de *Décorateur*, le fait que chaque classe « décorateur concret » (ConcreteDecoratorX) localise toutes les définitions relatives à une fonctionnalité ajoutée. En ce sens, chaque aspect peut, de plus, détenir un ensemble d'introductions d'attributs et d'opérations nécessaires au fonctionnement de la nouvelle fonctionnalité ajoutée. Tous les aspects ainsi définis sont enfin composés conformément à l'ordre d'exécution des fonctionnalités supplémentaires requises par le client de la classe cible d'impact. Ceci est une solution par aspects possible pour le patron *Décorateur* qui offre plusieurs avantages quant à la modularisation, l'évolution et la réutilisation de l'imitation du patron mais aussi du code de l'application de base. Pour retirer une fonctionnalité il suffit, avec cette nouvelle solution, de supprimer l'un des aspects ainsi définis. Toutefois, comme le tissage d'aspects en AspectJ est statique, cette solution ne permet pas de supprimer une fonctionnalité supplémentaire qui doit pouvoir être retirée dynamiquement. Il est possible, cependant, d'assurer ce besoin par le biais de paramètres supplémentaires conditionnant l'exécution des fonctionnalités ajoutées. De tels paramètres sont destinés à être spécifiés au moment de la création de l'objet même. Il convient donc, dans ce cas, d'étendre la classe impactée par l'ajout de nouveaux attributs spécifiant ces paramètres et un nouveau constructeur prenant en compte ces derniers. L'exécution ou non d'une fonctionnalité ajoutée peut alors être décidée dynamiquement grâce aux valeurs de ces paramètres.

- Le patron *Procuration* fournit à un objet tiers un mandataire ou un remplaçant, pour contrôler l'accès à cet objet [Gamma *et al.*, 95]. La solution en AspectJ que nous proposons pour la procuration de protection consiste, en effet, à intercepter toutes les requêtes demandant à exécuter les opérations devant proposer des accès protégés (telle que request(), dans la structure origine par objets de ce patron), et à décider de l'exécution ou non de ces opérations. Ceci est possible grâce à la définition d'un aspect concret qui détient la déclaration d'un point de recouvrement et d'un perfectionnement conditionnant l'exécution des opérations en question. La figure 3.12 ci-dessous présente la solution en AspectJ que nous proposons pour la procuration de protection.

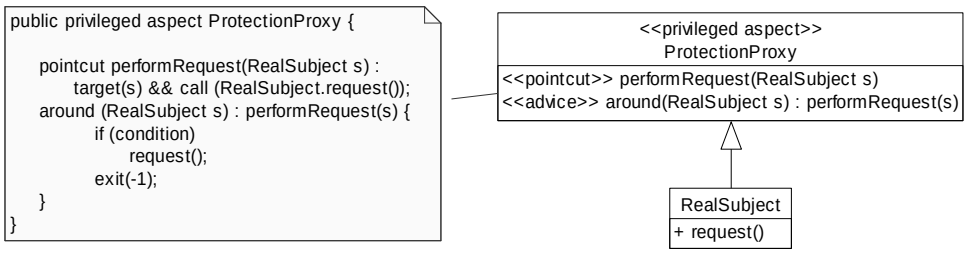


Figure 3.12. *Solution en AspectJ du patron* Procuration de protection

Grâce à l'utilisation des points de jonctions et des perfectionnements, les nouvelles solutions en AspectJ de ces deux patrons permettent d'adapter les classes impactées de manière transparente, tout en gardant isolées et encapsulées les imitations des patrons. Ceci résout les problèmes d'utilisation de ces deux patrons et facilite l'évolution et la réutilisation du code de l'application.

2.2.2 Patrons *Stratégie*, *Patron de méthode*, *Fabrication* et *Fabrique abstraite*

Les patrons *Stratégie*, *Patron de méthode*, *Fabrication* et *Fabrique Abstraite* proposent, dans le fond, de donner un comportement polymorphe aux instances d'une classe contexte donnée, sans modifier sa définition de base. Le comportement à adapter peut consister en une ou plusieurs propriétés comportementales, en fonction du patron considéré. Dans *Stratégie* et *Fabrication* il s'agit, par exemple,

d'une seule propriété (une opération dans *Stratégie* et l'opération de création dans *Fabrication*), alors qu'il s'agit de plusieurs propriétés dans *Patron de méthode* (opérations primitives) et *Fabrique abstraite* (opérations de création des objets produits concrets). Pour réaliser ces patrons en AspectJ, nous proposons pour chacun d'entre eux d'isoler et de regrouper, dans un aspect concret, la définition des différentes formes du comportement considéré. L'affectation des formes de comportement idoines aux instances de la classe impactée, est ensuite assurée grâce aux mécanismes Aspect d'extension d'interface et d'adaptation de comportement. Ces formes peuvent être partagées par plusieurs instances de la même classe. Mais, toute instance ne peut avoir qu'une seule forme tout au long de son exécution. Pour mieux comprendre le principe de fonctionnement des solutions que nous proposons pour ces quatre patrons, nous détaillons ci-dessous, à titre d'exemple, la solution en AspectJ du patron *Stratégie*.

Réalisation du patron* Stratégie *à l'aide d'AspectJ. *Stratégie* a pour intention de définir une famille d'algorithmes interchangeables pour une classe donnée, permettant à ces algorithmes d'évoluer indépendamment des clients qui les utilisent [Gamma *et al.*, 95]. La solution en AspectJ que nous proposons pour ce patron est montrée par la figure 3.13. Elle consiste, tout d'abord, à regrouper la définition des différentes formes d'algorithmes mises en jeu dans un aspect. Elle propose, ensuite, d'étendre la classe impactée (utilisant ces algorithmes) par l'ajout d'un attribut strategy, utilisé pour la représentation interne de l'algorithme idoine associée à un objet de type Contexte. Cet attribut est donc statiquement introduit par l'aspect ContextStrategies dans la classe Contexte. De la même façon un nouveau constructeur paramétré (Context(int stg)) est introduit dans cette même classe, permettant de spécifier la valeur de l'attribut strategy au moment de la création d'une instance de Contexte. L'exécution de l'algorithme approprié à chaque instance peut alors être assurée par l'utilisation d'un point de recouvrement (ou *pointcut*, dénotant l'appel à l'interface de la famille d'algorithmes) et d'un perfectionnement (*advice*) associé à ce dernier. Le perfectionnement détermine l'algorithme à exécuter grâce à la valeur attribuée à strategy.

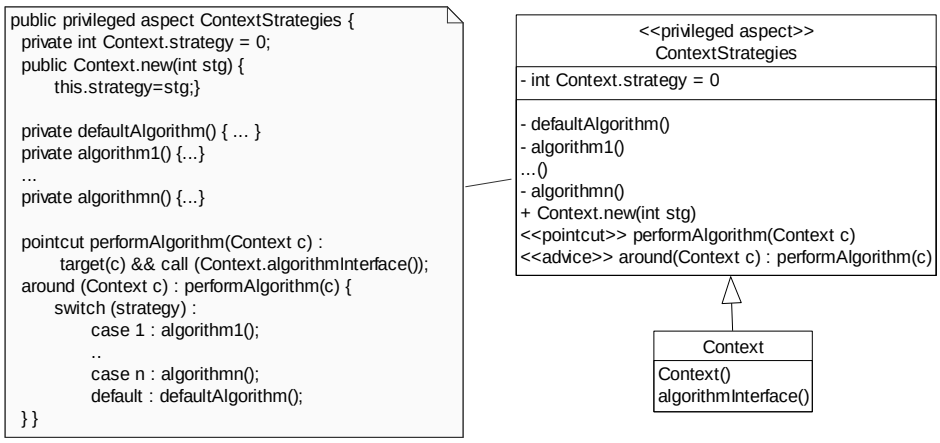


Figure 3.13. *Solution en AspectJ du patron* Stratégie

Réalisation de* Fabrication, Fabrique abstraite *et* Patron de méthode *à l'aide d'AspectJ. Nous expliquons brièvement le reste des solutions des trois autres patrons. Le patron *Fabrication* permet de doter une classe Creator par une opération permettant la création d'un objet de type abstrait Produit [Gamma *et al.*, 95]. Le choix de la classe de produit concrète (ConcreteProduct) à instancier est délégué aux instances de Creator. La solution en AspectJ de ce patron est similaire à celle proposée pour *Stratégie*, et la propriété comportementale mise en jeu correspond à la méthode de fabrication de produit. Dans le même esprit, le patron *Fabrique abstraite* permet également à une classe Factory de créer un objet complexe qui consiste en plusieurs produits, mais en laissant à ses instances le choix des classes concrètes des différents produits

composants à instancier. Ce patron met en jeu alors plusieurs opérations de création, il est réalisé de la même manière que les deux patrons précédents tout en utilisant plusieurs points de recouvrement et plusieurs perfectionnements. Ce qui est valable aussi pour le patron *Patron de méthode*, qui met en jeu plusieurs opérations primitives devant avoir des comportements variés d'une instance (de la classe impactée) à une autre.

Reste enfin à noter qu'avec ce principe de réalisation par aspects des quatre patrons considérés ci-dessus, nous pouvons garantir la traçabilité des imitations de ceux-ci (une imitation par aspect). De plus, les classes adaptées par ces patrons s'en trouvent ainsi inchangées, et donc plus faciles à faire évoluer et à réutiliser directement dans un autre contexte d'application.

2.2.3 Patrons *Monteur*, *Commande*, *État* et *Pont*

Réalisation des patrons* Monteur *et* Commande *à l'aide d'AspectJ. Les patrons *Monteur* et *Commande* proposent de fournir un comportement polymorphe (opérations de création du produit complexe pour *Monteur* et méthode d'exécution de la requête pour *Commande*) aux instances d'une même classe contexte (Director, respectivement, Invoker, voir les structures par objets de ces deux patrons dans [Gamma *et al.*, 95]). Un tel comportement est défini, en général, sur plusieurs opérations. A chaque instance de contexte doit être associée une définition appropriée du comportement mis en jeu, cette définition ne peut pas être échangée avec une autre tout au long de l'exécution de cette instance. Les solutions de réalisation par aspects que nous proposons pour ces deux patrons sont, en partie, similaires à celles proposées pour les quatre patrons précédents (*Stratégie*, *Patron de méthode*, *Fabrication* et *Fabrique Abstraite*). Toutefois, à la différence de ces quatre dernières solutions, les définitions des opérations définissant le comportement polymorphe sont regroupées et encapsulées dans différentes classes de comportement interchangeables, spécifiant chacune une forme de comportement particulier. Il s'agit ensuite de doter chacune des instances de la classe contexte d'un objet de comportement adéquat, au moment de sa création. Ce choix a été fait dans un souci d'isoler et de rendre interchangeables les différentes formes du comportement considéré, et de faciliter donc leur maintenance et leur évolution. Nous proposons de détailler ci-dessous, à titre d'exemple, la solution en AspectJ du patron *Monteur*.

Le patron *Monteur* dissocie la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction permette des représentations différentes [Gamma *et al.*, 95]. L'intention est donc de fournir un algorithme de création indépendant des parties qui composent l'objet complexe, et de la manière dont ces parties sont agencées. La solution en AspectJ que nous proposons pour ce patron est, en partie, similaire à sa structure par objets initiale. Elle préconise, en effet, de :

- définir dans la classe Director le squelette de l'algorithme de création (construct()), sous la forme d'un patron de méthodes qui appelle les différentes opérations de création primitives dont les comportements doivent être conçus comme modifiables ;
- d'encapsuler les différentes formes de comportements associés aux opérations primitives dans une hiérarchie de classes Builder. Chaque classe ConcreteBuilder implémente les opérations primitives spécifiant une représentation particulière.

Cependant, à la différence de sa structure initiale, la réalisation en AspectJ de *Monteur* préconise que les opérations primitives offrent des comportements par défaut, pour effectuer les étapes de l'algorithme de création avec une représentation quelconque. Elle propose, ensuite, d'établir le lien entre Director et Builder de manière transparente à Director, grâce à l'introduction dans ce dernier d'un nouvel attribut builder utilisé pour la représentation interne d'un objet de comportement délégué à une instance de Director. Cet attribut est statiquement introduit dans la classe Director par l'aspect DirectorBuilders, de même que la définition d'un nouveau constructeur prenant en paramètre un objet de type Builder. Ce

constructeur permet en effet d'associer, une fois pour toutes, un objet de type Builder à une instance de la classe Director. La figure 3.14 montre la structure en AspectJ que nous proposons pour ce patron. L'exécution des formes idoines des opérations primitives, effectuant les étapes de l'algorithme de création spécifique à une représentation donnée, peut alors être assurée par l'utilisation de différents perfectionnements associés à des points de recouvrement distincts. Chaque point de recouvrement devra intercepter tout appel à une seule opération primitive. Les perfectionnements associés à ces points de recouvrement exécutent les comportements appropriés à une instance particulière par délégation explicite des requêtes à son objet de comportement builder.

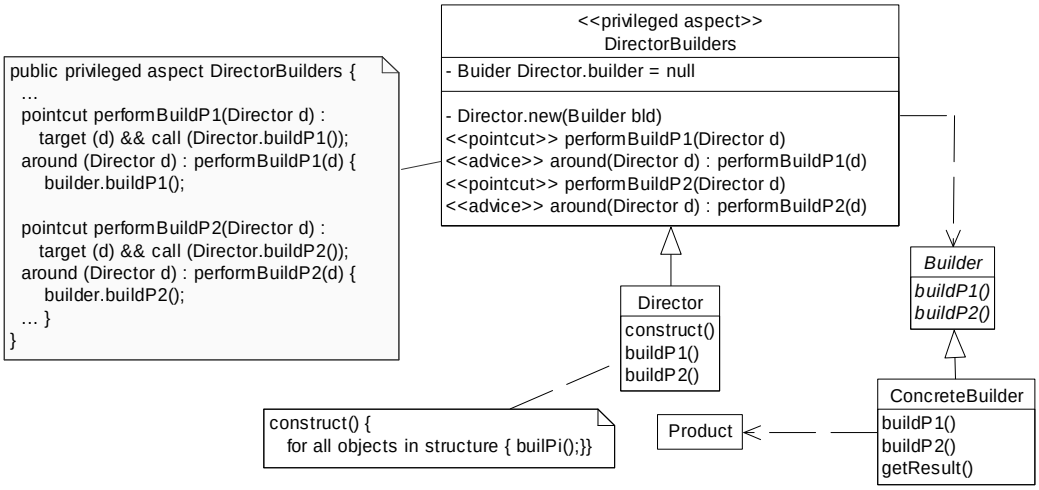


Figure 3.14. *Solution en AspectJ du patron* Monteur

Réalisation des patrons* État *et* Pont *à l'aide d'AspectJ. Les solutions de réalisation par aspects des patrons *État* et *Pont* proposent de fournir des fonctionnalités similaires, à quelques détails près. Nous expliquons ci-dessous les solutions en AspectJ que nous proposons pour ces deux patrons.

- Le patron *État* permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe [Gamma *et al.*, 95]. Il propose ainsi, dans le fond, d'offrir un comportement polymorphe aux instances d'une même classe Context (voir la structure par objets de ce patron dans [Gamma *et al.*, 95]), tout en leur associant différents objets d'état (de type State) interchangeables. En ce sens, à la différence des deux patrons précédents, *État* devra permettre en plus à toute instance de Context de changer de comportement de manière dynamique, tout en permutant son objet de comportement au moment de son exécution. Pour ce faire en AspectJ, nous proposons de conserver une solution similaire à celle proposée pour *Monteur*, offrant en plus une méthode accesseur setState() permettant de changer dynamiquement l'objet de comportement d'une instance donnée. La transition d'états d'une instance de Context est régie par son état interne. Les critères de transition d'états peuvent ainsi être déterminés et définis intégralement au niveau de l'aspect détenant l'imitation du patron. Il s'agit en fait d'intercepter tout appel à une méthode de Context pouvant modifier l'état interne des instances de celle-ci, grâce à des points de recouvrement et d'associer à chacun de ces points un perfectionnement distinct. Chaque perfectionnement vérifie après l'exécution de la méthode interceptée si l'état interne de l'instance en question est changé ; en un tel cas, l'objet d'état de cette instance est changé en conséquence.
- Le patron *Pont* propose de découpler une abstraction (Abstraction) de son implémentation (Implementor), afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre [Gamma *et al.*, 95]. L'intention est que l'implémentation de tout objet de type Abstraction, ou plus particulièrement, de type RefinedAbstraction peut être choisie parmi plusieurs et échangée avec une

autre lors de l'exécution de cet objet. Nous remarquons qu'il s'agit, en partie, de la même préoccupation que pour *État*, à l'exception des règles de changement d'objets de comportement qui sont indépendantes de l'état interne de l'instance en question. En effet, dans le cas de ce patron, c'est le client de l'objet Abstraction qui décide de la permutation, si besoin est, de l'implémentation destinée à être associée à cet objet. La solution en AspectJ que nous proposons pour ce patron est ainsi similaire à celle proposée pour *État*. La figure 3.15 ci-dessous présente la solution de *Pont*, avec un exemple de code.

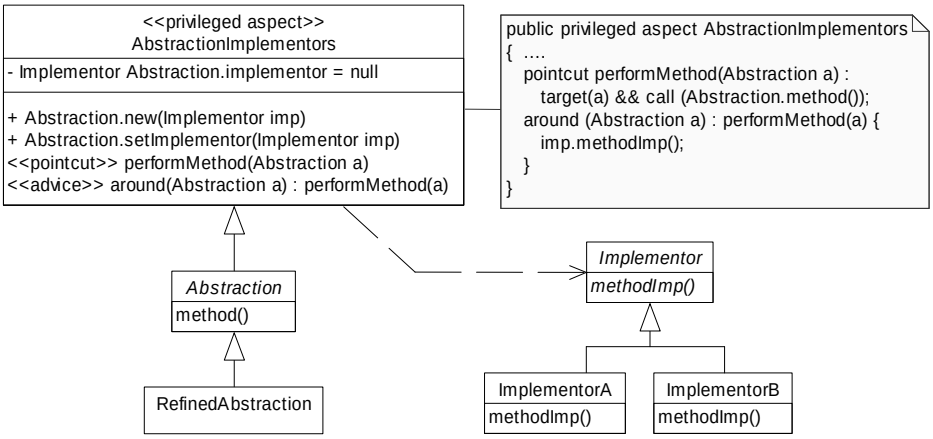


Figure 3.15. *Solution en AspectJ du patron* Pont

Comme le montre cette figure, la solution du patron *Pont* offre également une méthode d'accès setImplementor() introduite statiquement dans la classe Abstraction par l'aspect qui représente l'imitation de ce patron (AbstractionImplementors). Cette méthode permet de changer dynamiquement l'objet d'implémentation délégué à une instance d'Abstraction. Toutefois, la solution de ce patron ne considère aucune règle de changement d'objet d'implémentation, et laisse cette tâche à la charge du client des objets Abstraction.

Nous remarquons que les solutions de réalisation des quatre patrons considérés dans cette partie améliorent l'évolution et la réutilisation des classes adaptées par ces patrons, du moment où elles le sont de manière transparente et que leurs définitions de base sont inchangées. Par ailleurs, ces nouvelles solutions garantissent que chaque imitation d'un patron soit séparée et encapsulée dans un aspect à part.

2.2.4 Patrons *Prototype*, *Mémento*, *Itérateur*, *Composite*, *Médiateur*, *Chaîne de responsabilité* et *Poids mouche*

Comme pour le patron *Observateur*, les solutions de réalisation par aspects des patrons *Prototype*, *Mémento*, *Itérateur*, *Composite*, *Médiateur*, *Chaîne de responsabilité* et *Poids mouche* préconisent de définir pour chaque patron, un aspect abstrait commun à toutes ses imitations et un aspect concret par imitation spécifique. Elles diffèrent cependant par les éléments déclarés dans les deux aspects, selon la préoccupation du patron considéré.

Réalisation de* Prototype *et* Composite *à l'aide d'AspectJ. Les patrons *Composite* et *Prototype* proposent, implicitement et indépendamment de leurs intentions spécifiques, d'étendre l'interface de différentes classes existantes devant avoir un comportement commun, sans modifier leurs comportements originaux. Par exemple, le patron *Prototype* propose de doter des classes concrètes ConcretePrototypek (voir sa structure initiale dans [Gamma *et al.*, 95]) d'un nouveau comportement, permettant à n'importe quelle instance de l'une des classes considérées de se cloner elle-même. Il propose pour ce faire, dans sa structure initiale, de définir une super-classe abstraite Prototype utilisée pour le typage des différentes classes sur lesquelles il s'applique. Par ailleurs, afin de composer des objets en des structures arborescentes

pour représenter des hiérarchies composant/composé et traiter de la même et unique façon les différents types d'objets, le patron *Composite* propose aussi d'étendre les interfaces des classes représentant ces objets, par la définition d'une super-classe Component et ses deux sous-classes Leaf et Composite utilisées pour le typage des classes mises en jeu. Leaf et Composite spécifient, respectivement, le comportement commun à tous les objets individuels et le comportement commun à tous les objets combinaisons de ceux-ci [Gamma *et al.*, 95]. Nous proposons pour réaliser ces deux patrons en AspectJ de conserver des solutions similaires, isolant une partie abstraite générale et une partie concrète particulière à chaque imitation. La partie abstraite consiste en un aspect abstrait déclarant une ou plusieurs interfaces internes utilisées pour le typage, uniquement dans le contexte du patron. Ces interfaces spécifient, de manière abstraite, les comportements (opérations) destinés à étendre les interfaces initiales des classes sur lesquelles s'applique le patron. Remarquons que ces opérations sont associées à ces interfaces par le biais d'introductions. Elles sont destinées à être héritées et concrétisées par les classes concrètes, qui acquièrent les types d'interfaces grâce aux déclarations d'héritage détenues par l'aspect concret. Pour mieux comprendre ces réalisations, nous détaillons ci-dessous, à titre d'exemple, la solution en AspectJ du patron *Prototype* (cf. figure 3.16).

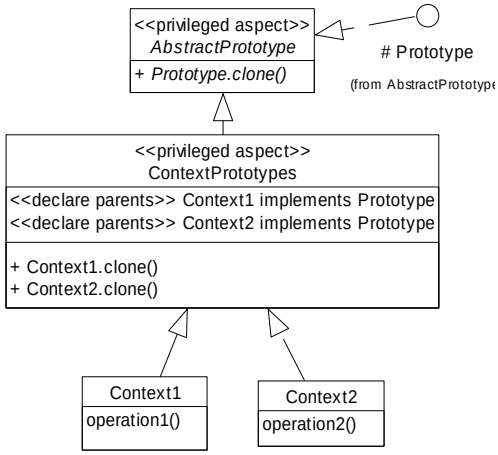


Figure 3.16. *Solution en AspectJ du patron* Prototype

Nous retrouvons dans cette figure un aspect abstrait AbstractPrototype, déclarant une interface interne Prototype et une introduction dans cette interface d'une opération abstraite clone(), qui spécifie le comportement à ajouter. La partie concrète est constituée d'un aspect ContextPrototypes qui concrétise AbstractPrototype et détient les introductions des différentes définitions de l'opération abstraite dans les classes Contextk. Cet aspect concret spécifie de plus les liens d'héritage entre ces dernières classes et l'interface Prototype.

Réalisation des patrons* Itérateur *et* Mémento *à l'aide d'AspectJ. Dans le principe, ces deux patrons proposent, dans un but particulier, d'étendre les fonctionnalités de base d'une classe contexte (Originator dans *Memento* et ConcreteAggregate dans *Itérateur*) sans mettre à découvert sa représentation interne. Les solutions en AspectJ de ces deux patrons proposent, par exemple, de faire intervenir une classe supplémentaire détenant toutes les propriétés (attributs et méthodes) nécessaires à la réalisation de la nouvelle fonctionnalité, pour le compte de la classe contexte. Ces solutions préconisent, par ailleurs, de définir pour chaque patron un aspect abstrait commun à toutes les imitations de celui-ci, et un aspect concret par imitation de patron. L'aspect abstrait déclare deux interfaces spécifiant respectivement, le rôle joué par la classe contexte et le rôle joué par la classe supplémentaire. Il associe également à ces deux interfaces des définitions d'opérations abstraites et des déclarations d'attributs, par le bais d'introductions. Il s'agit de l'ensemble des propriétés nécessaires à la réalisation de la nouvelle fonctionnalité. Les

opérations abstraites de ces deux interfaces sont destinées à être concrétisées par les classes participantes dans le patron, aussi par le biais d'introductions, au niveau de l'aspect concret. Nous détaillons ci-dessous, à titre d'exemple, la solution en AspectJ que nous proposons pour *Itérateur* afin d'illustrer ces réalisations.

Le patron *Itérateur* propose de fournir un moyen d'accès séquentiel aux éléments d'un agrégat d'objets [Gamma *et al.*, 95]. La figure 3.17 présente la solution en AspectJ de ce patron. Comme le montre cette figure, un aspect abstrait AbstractIterator déclare deux interfaces spécifiques respectivement, à l'agrégat d'objets (Aggregate) et à la classe représentant l'itérateur (Iterator). Remarquons que ce même aspect associe aussi, à l'interface Iterator, un ensemble d'opérations abstraites (first(), next(), etc.) et un attribut current, utiles pour assurer l'accès séquentiel aux éléments de l'agrégat d'objet considéré. Nous pouvons remarquer d'ailleurs qu'une opération abstraite createIterator() est également introduite dans l'interface Aggregate, elle permet de créer l'objet itérateur. Par ailleurs, un aspect concret ContextIterator dénotant une imitation particulière du patron est défini pour :

- attribuer le rôle Aggregate à la classe concrète Context participant dans le patron ;
- définir une classe interne ContextIterator représentant les objets itérateurs spécifiques à la classe Context, et lui attribuer le rôle Iterator ;
- concrétiser, par le biais d'introductions, toutes les opérations abstraites de l'interface Iterator et celles de l'interface Aggregate.

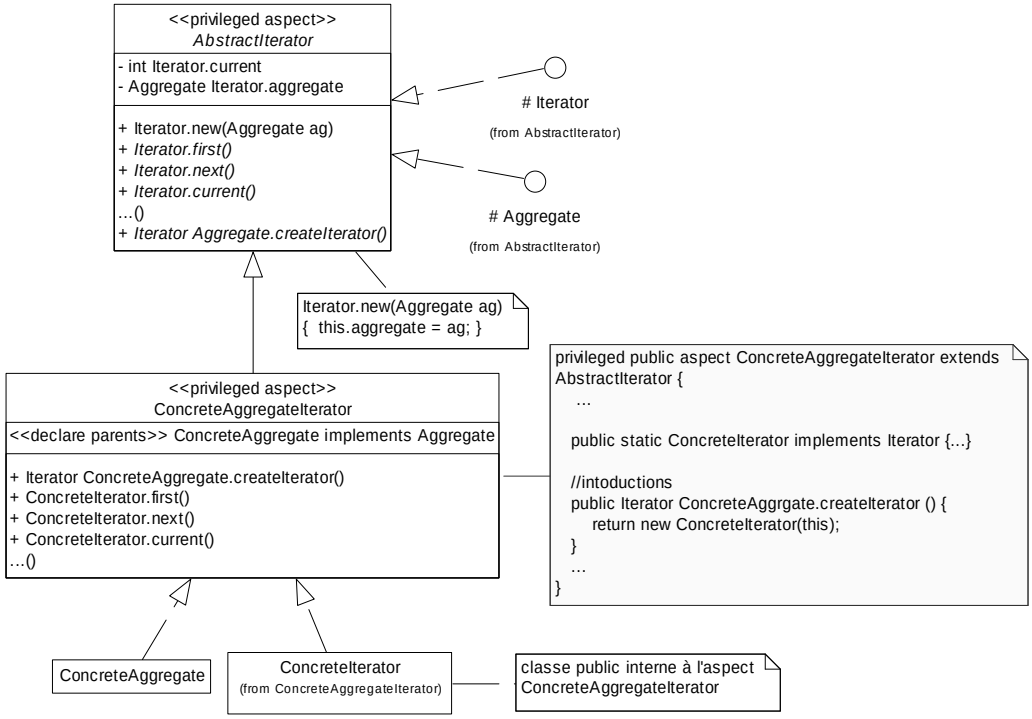


Figure 3.17. *Solution en AspectJ du patron* Itérateur

Réalisation de* Chaîne de responsabilité*,* Médiateur *et* Poids mouche *à l'aide d'AspectJ. Les solutions de réalisation en AspectJ de ces trois patrons sont très similaires à celle proposée pour le patron *Observateur* (cf. figure 3.5). Elles proposent de réaliser une fonctionnalité complexe par collaboration de plusieurs classes, et font intervenir en conséquence différentes opérations originaires de celles-ci. Pour ce faire, en AspectJ, ces solutions proposent pour chaque patron d'isoler une partie commune à toutes ses imitations dans un aspect abstrait, et de définir dans un aspect concret une partie concrète spécifique à chaque imitation de celui-ci. L'aspect abstrait déclare une ou plusieurs interfaces internes (selon le patron

considéré) définissant les différents rôles joués par les classes participantes, et attache à ces interfaces, par le biais d'introductions, des opérations avec des implémentations par défaut, si possible, spécifiant les comportements minimaux des classes devant jouer ces rôles. Remarquons que les opérations abstraites des interfaces considérées doivent être concrétisées par les classes participantes, toujours par le biais d'introductions, au niveau de chaque aspect concret dénotant une imitation particulière. L'aspect abstrait peut définir en plus, de manière abstraite si possible (comme dans le cas *d'Observateur* et *Chaîne de responsabilité*), le protocole de communication des classes participantes, grâce à un point de recouvrement abstrait auquel est associé un perfectionnement réalisant proprement la collaboration attendue. Le point de recouvrement abstrait est destiné à être concrétisé de façon spécifique par chaque aspect concret dénotant une imitation particulière d'un patron. Dans le cas où il n'est pas possible de définir le protocole de communication au niveau de l'aspect abstrait (c'est le cas des patrons *Médiateur* et *Poids mouche*), celui-ci est complètement défini par chaque aspect concret dénotant une imitation particulière. C'est au niveau de l'aspect concret que sont aussi définis les différents liens d'héritage entre les classes participantes et leurs interfaces appropriées. Il en est de même pour la concrétisation des opérations abstraites de ces interfaces.

Nous remarquons enfin que toutes les réalisations par aspects de l'ensemble des patrons considérés dans cette partie assurent une bonne traçabilité de chaque imitation spécifique. Il est à noter aussi que la réutilisation de ces patrons est augmentée grâce à leurs aspects abstraits qui peuvent être utilisés par toutes leurs imitations. Ceci est aussi valable pour les classes adaptées par ces patrons, dont les définitions de base restent inchangées. Ceci facilite par ailleurs leur évolution.

2.2.5 *Le patron* Façade

Pour le patron *Façade*, les solutions de réalisation en *AspectJ* et *Hyper/J* ne présentent aucun apport supplémentaire, par rapport à sa structure d'origine par objets. Ceci tient principalement au fait que le patron *Façade* ne propose pas d'adapter les définitions des classes d'un sous-système sur lequel il s'applique. Il propose, toutefois, de fournir tout simplement une interface de plus haut niveau, unifiée à l'ensemble des interfaces des classes qu'il considère, et qui rend le sous-système plus facile à utiliser. Sa structure initiale [Gamma *et al.*, 95] propose pour ce faire de définir une nouvelle classe fournissant l'ensemble des services attendus du sous-système. Nous proposons une solution similaire en AspectJ et Hyper/J.

3. Synthèse des nouvelles solutions par aspects des patrons du GoF

Nous proposons dans ce qui suit de faire la synthèse de l'ensemble des solutions en AspectJ et Hyper/J des patrons du *GoF*. Nous considérons tout d'abord les apports et les propriétés caractéristiques de ces nouvelles solutions. Nous proposons ensuite une nouvelle classification pour l'ensemble des 23 patrons, et nous présentons enfin une comparaison des solutions en AspectJ et celles en Hyper/J.

Le tableau 3.1 suivant présente une synthèse de l'ensemble des réalisations par aspects des 23 patrons considérés[8]. Il met en évidence deux des propriétés caractéristiques de ces solutions : la traçabilité et la réutilisation. Pour chaque patron, il identifie, en plus, le type d'adaptation prédominante, le contexte structurel d'une telle adaptation, ainsi que son contexte dynamique. Le contexte structurel indique si l'adaptation prédominante s'applique, dans le principe, à des classes ou à des objets. Le contexte dynamique indique par ailleurs si cette adaptation doit être appliquée de manière statique, au moment de l'instanciation des classes ou des objets impactés, ou au cours de l'exécution de ces objets. Nous proposons également de présenter dans ce même tableau le groupe de chaque patron. Nous distinguons exactement neuf groupes de patrons.

[8] Les patrons du GoF sont présentés dans ce tableau selon leur ordre d'apparition dans [Gamma *et al.*, 95].

Tableau 3.1. *Propriétés caractéristiques des nouvelles solutions par aspects des 23 patrons du GoF*

	Propriétés caractéristiques					
Patrons	***Traçabilité***	***Réutilisation***	***Type d'adaptation prédominante***	***Contexte structurel***	***Contexte dynamique***	***Groupes***
Fabrique abstraite	oui	non	redéfinition de plusieurs propriétés comportementales	classe	instanciation	3
Monteur	oui	non	redéfinition de plusieurs propriétés comportementales par composition	classe	instanciation	4
Fabrication	oui	non	redéfinition d'une propriété comportementale	classe	instanciation	3
Prototype	oui	oui	ajout de fonctionnalités	classe	statique	7
Singleton	oui	non	redéfinition du constructeur	classe	exécution	2
Adaptateur	oui	non	ajout de fonctionnalités	classe	statique	1
Pont	oui	non	redéfinition de plusieurs propriétés comportementales par composition	objet	exécution	5
Composite	oui	oui	ajout de fonctionnalités	class	statique	6
Décorateur	oui	non	redéfinition de propriétés comportementales	class	exécution	2
Façade	oui	non	aucune adaptation des classes existantes. Création d'une nouvelle fonctionnalité à part.	classe	statique	9
Poids Mouche	oui	oui	redéfinition de comportement dans le but d'une collaboration	objet	exécution	8
Procuration	oui	oui	redéfinition de comportement	classe	exécution	2
Chaîne de responsabilité	oui	oui	redéfinition de comportement dans le but d'une collaboration	objet	exécution	8
Commande	oui	oui	redéfinition de plusieurs propriétés comportementales par composition	classe	instanciation	4
Interpréteur	oui	non	ajout de fonctionnalités	classe	statique	1
Itérateur	oui	oui	ajout de fonctionnalités	classe	statique	6
Médiateur	oui	oui	redéfinition de comportement dans le but d'une collaboration	objet	exécution	8
Mémento	oui	oui	ajout de fonctionnalités	classe	statique	7
Observateur	oui	oui	redéfinition de comportement dans le but d'une collaboration	objet	exécution	8
État	oui	non	redéfinition de plusieurs propriétés comportementales par composition	objet	exécution	5
Stratégie	oui	non	redéfinition d'une propriété comportementale	classe	instanciation	3
Patron de méthode	oui	non	redéfinition de plusieurs propriétés comportementales	classe	instanciation	3
Visiteur	oui	non	ajout de fonctionnalités	classe	statique	1

3.1 Avantages et caractéristiques des solutions proposées

Nous remarquons, à travers ce tableau, que toutes les implémentations en *AspectJ* et *Hyper/J* garantissent une bonne traçabilité des imitations des patrons au niveau de l'implémentation. Elles améliorent donc, à la fois, la lisibilité et la compréhension du code relatif aux imitations des patrons, mais aussi de celui relatif aux classes sur lesquelles s'appliquent les patrons. Leur réutilisation est également

augmentée, et leur évolution est par conséquent facilitée. En effet, de manière générale, les nouvelles réalisations par aspects des 23 patrons de conception du GoF se caractérisent par les propriétés suivantes.

- *Traçabilité*. Toutes les implémentations des patrons en AspectJ et Hyper/J sont localisables. On arrive donc à tracer facilement tout le code relatif à une imitation particulière d'un patron. Dans le cas d'Hyper/J, par exemple, chaque imitation d'un patron est isolée et encapsulée dans exactement un *hyperslice*. De même que, dans AspectJ, toute imitation d'un patron est en général isolée et encapsulée dans un *aspect* concret. En effet, même si le noyau d'un patron (i.e. le protocole de collaboration qu'il définit) est souvent généralisé dans un aspect abstrait, celui-ci est réutilisé et partagé par l'ensemble des imitations de ce patron qui restent toujours localisables. Ce qui est aussi le cas pour Hyper/J, comme nous l'avons montré pour la solution du patron *Observateur*. Enfin, force est de constater que le code des imitations des patrons n'est plus enchevêtré dans le code relatif aux classes d'applications.

- *Évolution.* Le code d'une imitation d'un patron étant en général isolé et encapsulé dans exactement une seule unité modulaire (i.e. *aspect* ou *hyperslice*), il est donc plus lisible et plus compréhensible. La maintenance et l'évolution des imitations des patrons deviennent par conséquent plus faciles. Par ailleurs, le code des classes d'une application sur lesquelles s'appliquent un ou plusieurs patrons se trouve indépendant de celui relatif aux imitations de ces derniers. Ces classes sont donc plus lisibles et compréhensibles. Leurs maintenance et évolution deviennent également plus faciles.

- *Réutilisation*. Dans le cas d'AspectJ, pour exactement dix patrons de conception du *GoF* (*Commande*, *Observateur*, *Composite*, *Médiateur*, *Chaîne de responsabilité*, *Poids mouche*, *Prototype*, *Mémento*, *Itérateur* et *Procuration*) les définitions des noyaux invariants de ces derniers sont généralisées et isolées dans des aspects abstraits. De tels aspects peuvent être réutilisés et partagés par plusieurs imitations des patrons considérés, pour être adaptés aux contextes de leurs applications. Grâce à leurs solutions en AspectJ, les patrons du GoF deviennent ainsi plus réutilisables. Les solutions en Hyper/J de ces dix patrons présentent également, pour chacun de ces derniers, une partie invariante commune à ses imitations et qui peut être réutilisée dans plusieurs autres contextes d'application. Qu'en est-il par ailleurs de la réutilisation des classes sur lesquelles s'appliquent les nouvelles solutions par aspects des patrons ? Les classes d'une application adaptées par un ou plusieurs patrons le sont d'une manière transparente, et leurs définitions de base restent inchangées. Elles se trouvent donc indépendantes des imitations par aspects des patrons et sont, par conséquent, directement réutilisables dans plusieurs autres contextes d'application qui ne requièrent pas l'application de patrons sur de telles classes.

- *Composition transparente*. Nous remarquons que dans les deux cas d'AspectJ et Hyper/J le code de toute classe adaptée par un ou plusieurs patrons est toujours indépendant de celui relatif aux imitations de ceux-ci. La composition de l'ensemble des imitations des patrons appliquées sur une classe donnée, se fait donc de manière non destructive et transparente à cette dernière. Les imitations des différents patrons considérés ne sont pas confondues, chaque imitation est bien isolée à part.

3.2 Résolution des problèmes

Comme nous l'avons illustré dans les exemples de réalisation des patrons *Visiteur*, *Observateur* et *Singleton*, AspectJ et Hyper/J intègrent des mécanismes et concepts qui permettent la mise en œuvre des patrons de conception par objets, tout en évitant l'ensemble des problèmes liés à leur utilisation (cf. § 3.3 du premier chapitre pour l'ensemble des problèmes considérés). Nous revenons ici sur chacun de ces

problèmes d'utilisation des patrons, hors du cadre d'un patron considéré en particulier, ou d'un langage de programmation par aspects spécifique, pour expliquer les apports des nouvelles solutions proposées.

- Le *problème de confusion et de traçabilité* est simplement évité grâce à la définition même des unités modulaires (*aspect/hyperslice*), permettant d'isoler et d'encapsuler les préoccupations transversales. Ces unités nous fournissent effectivement une structure dans laquelle nous pouvons isoler le code propre à l'imitation des patrons. Il est à noter que même si certaines solutions proposées s'attachent à définir plusieurs unités, afin d'augmenter la réutilisation de définitions entre imitations du même patron, il est possible de localiser dans l'ensemble du code chaque imitation de patron par une seule unité modulaire.
- Le *problème d'indirection et de surcharge de l'implémentation* provient généralement de la nécessité de déporter une partie des adaptations d'une classe, afin d'en garder une définition séparée, dans d'autres classes. Les mécanismes et concepts Aspect (tel que l'introduction dans AspectJ par exemple) nous permettent de définir ces adaptations comme étant directement effectives pour une telle classe, tout en gardant leur définition séparée dans une unité modulaire.
- Le *problème de rupture d'encapsulation* ne se pose plus lorsque les adaptations directes sont utilisées à la place de délégations explicites. Dans le cas d'*AspectJ*, déclarer privilégié l'aspect dénotant l'imitation d'un patron est une solution simple et efficace, l'encapsulation n'étant levée que pour les seuls besoins du patron et non pour le reste de l'application.
- Enfin, *les problèmes liés à l'héritage* ne se posent plus. Les mécanismes et concepts Aspect (tels que la déclaration d'héritage dans *AspectJ*) nous permettent de modifier de façon transparente la hiérarchie d'héritage des classes existantes. Nous pouvons d'ailleurs réaliser facilement une forme d'héritage multiple, quand le langage Objet de base ne le permet pas.

3.3 Une nouvelle classification des patrons du GoF

Comme le montre le tableau 3.1, nous remarquons qu'il existe différents groupes de patrons par respect des propriétés de leurs adaptations. D'ailleurs, l'analyse de l'ensemble des implémentations de ces patrons en AspectJ et Hyper/J, confirme cette catégorisation. Les patrons d'un même groupe présentent en effet des analogies au niveau de leurs codes. Les groupes distingués sont les suivants.

- Groupe 1 : *Adaptateur, Visiteur* et *Interpréteur*. Ces patrons proposent d'adapter les interfaces des classes existantes, tout en leur ajoutant de manière statique une ou plusieurs propriétés structurelles et/ou comportementales. Ils se basent pour ce faire sur l'utilisation des mécanismes et concept Aspect d'extension d'interface.
- Groupe 2 : *Singleton*, *Proccuration* et *Décorateur*. Dans le cas de ces patrons, il s'agit plutôt d'adapter le comportement des classes existantes. Pour ce faire, ces patrons utilisent les mécanismes d'adaptation de comportement (tels que les *advice*s et *pointcut*s en AspectJ), permettant de modifier dynamiquement les définitions des propriétés comportementales concernées.
- Groupe 3 : *Stratégie*, *Fabrication, Patron de méthode* et *Fabrique abstraite*. Ces patrons proposent de modifier une ou plusieurs propriétés comportementales d'une classe existante, tout en donnant un comportement polymorphe aux instances de celle-ci. Les définitions idoines des propriétés comportementales mises en jeu sont spécifiées, une fois pour toutes, au moment de la création des instances de la classe cible.
- Groupe 4 : *Monteur* et *Commande*. Ces deux patrons proposent d'adapter plusieurs propriétés comportementales d'une classe contexte, tout en donnant un comportement polymorphe aux

instances de celle-ci. Pour ce faire, ils proposent de regrouper les différentes formes de comportement dans des objets, destinés à être associés à ces instances. En effet, à chaque instance de la classe contexte est associé un unique objet de comportement, défini une fois pour toutes au moment de sa création. Les requêtes envoyées à ces instances sont déléguées ensuite à leurs objets associés, grâce à l'utilisation des mécanismes Aspect d'adaptation de comportement.

- Groupe 5 : *État et Pont*. Comme les deux patrons précédents, ces deux patrons proposent également de donner un comportement polymorphe aux instances d'une même classe, tout en leurs associant des objets de comportement interchangeables. Ils permettent, cependant, de changer l'objet associé de n'importe quelle instance au cours de son exécution. Ceci permet de donner un comportement polymorphe à une même instance de la classe à adapter.
- Groupe 6 : *Itérateur* et *Composite*. Ces patrons proposent d'ajouter des responsabilités à une ou plusieurs classes contextes, dans des buts différents. Ils proposent en général d'augmenter les fonctionnalités de ces classes tout en étendant leurs interfaces, à travers des super-classes.
- Groupe 7 : *Mémento*, *Prototype*. Ces deux patrons proposent également d'augmenter les responsabilités d'une ou de plusieurs classes. Leurs solutions par aspects préconisent, cependant, de créer de nouvelles classes détenant les comportements supplémentaires pour le compte des classes adaptées.
- Groupe 8 : *Chaîne de responsabilité*, *Médiateur*, *Observateur* et *Poids mouche*. Ces patrons consistent à réaliser une nouvelle fonctionnalité complexe, mettant en jeu plusieurs classes en collaboration. Ils proposent pour ce faire :
  - d'étendre les interfaces de ces classes à travers des super-classes rôles, tout en leur ajoutant plusieurs propriétés nécessaires à l'accomplissement de la nouvelle fonctionnalité ;
  - et d'adapter les comportements de base de ces classes pour instaurer complètement le protocole de collaboration.
- Groupe 9 : *Façade*. Les solutions en AspectJ et en Hyper/J du patron *Façade* ne présentent aucune analogie avec les solutions des autres patrons ; ce patron constitue un groupe à part.

3.4 Comparaison des réalisations des patrons du GoF en AspectJ et Hyper/J.

Il existe certaines différences entre les implémentations en AspectJ et Hyper/J, qui conduisent à préférer le langage AspectJ pour une meilleure implémentation de ces patrons.

- Une différence majeure est celle liée à la spécification d'adaptations dynamiques. En effet, dans le cas d'AspectJ il est possible de spécifier des adaptations dynamiques d'une manière abstraite (comme le montre l'exemple du patron *Observateur* dont le point de recouvrement, spécifiant l'adaptation prédominante de ce patron, est défini sous la forme d'un *pointcut* abstrait au niveau de l'aspect abstrait *ObserverProtocol*). Ce qui augmente encore plus la réutilisation de ces patrons. Hyper/J ne permet aucune spécification d'adaptation générique, qui peut être spécialisée et adaptée ensuite selon le contexte d'application du patron.
- Le modèle de points de jonction offert par AspectJ est plus efficace que celui proposé par Hyper/J : il offre plusieurs possibilités quant à l'adaptation des comportements des classes.
- Les solutions que nous proposons en AspectJ réduisent de manière significative le nombre de participants impliqués dans une imitation d'un patron. En effet, l'imitation d'un patron en AspectJ se réduit le plus souvent à un seul Aspect concret, alors qu'elle est définie le plus souvent dans le cas d'Hyper/J par plusieurs classes.

4. Discussion et travaux connexes

Cette section présente une discussion relative à la transformation des patrons de conception par objets dans une approche Aspect, hors du cadre spécifique de notre travail. Nous commençons, pour ce faire, par présenter l'ensemble des travaux connexes, afin que cette discussion soit exhaustive. La suite de cette discussion est donnée sous la forme d'une liste de questions, auxquelles nous essayons d'apporter des éléments de réponse.

4.1 Travaux connexes

Il existe à notre connaissance, peu de travaux menés sur la réalisation des patrons de conception à l'aide des mécanismes et concepts introduits dans le cadre de l'approche Aspect. Nous présentons brièvement chacun de ces travaux, tout en positionnant le nôtre.

- [Lorenz 98] affirme que *Visiteur*, lorsqu'il est 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 notre pensée est inverse car nous nous basons directement sur les concepts Aspect pour définir les nouvelles solutions par aspects des patrons.
- [Noda *et al.*, 01] se sont intéressés à l'implémentation des patrons *Observateur*, *Composite* et *Adaptateur* de [Gamma *et al.*, 95] en *AspectJ* et *Hyper/J*. Ils s'attachent à définir très fidèlement un aspect correspondant à chaque classe introduite dans l'implémentation par objets d'un patron, puis à assurer la connexion entre les classes d'application et ces aspects en introduisant des aspects supplémentaires. Pour notre part, nous nous efforçons de proposer des solutions dans lesquelles un seul aspect dénote l'imitation d'un patron dans le code, en nous appuyant fortement sur l'intention des patrons plutôt que sur leur solution originale.
- [Hannemann *et al.,* 02] fournit aussi des implémentions de patrons de [Gamma *et al.*, 95] en *AspectJ*. Nous partageons avec ce travail un objectif et un domaine d'expérimentation communs. Nous avons comparé leurs résultats aux nôtres et nous avons distingué des différences[9]. Une différence majeure est qu'ils reproduisent le plus souvent, totalement ou partiellement, les structures par objets des solutions initialement proposées par Gamma. Par exemple, leur réalisation de *Visiteur* par aspects reproduit une partie de la structure proposée dans [Gamma *et al.*, 95], alors qu'en nous focalisant sur l'intention de *Visiteur*, nous aboutissons à une solution différente, ayant un nombre de constituants plus réduit.
- [Nordberg 01a], [Nordberg 01b] propose de nouvelles implémentations en AspectJ pour 12 patrons de conception par objets. Il montre comment ces implémentations peuvent mieux diriger, voire éliminer les dépendances (dues généralement à la délégation et l'indirection) dans la réalisation de ces patrons.
- [Hirschfeld *et al.*, 03] sont, à notre connaissance, les seuls à proposer de nouvelles implémentations en AspectS [Hirschfeld 02] de quelques patrons de conception du *GoF*. Ils laissent présager que l'on puisse bientôt fournir des descriptions de patrons incluant des exemples d'implémentation dans plusieurs langages de programmation par aspects.

[9] Ces différences sont explicitées dans la suite (cf. § 3.2 du chapitre 5) à travers l'expression des structures par aspects des solutions d'implémentation qu'ils proposent, ainsi que des solutions que nous proposons pour quelques uns des patrons du GoF.

Nous pouvons citer enfin les travaux de S. Clarke *et al.* qui ont mené un travail approfondi sur la conception par aspects, reposant sur des patrons de composition (*composition patterns*) [Clarke *et al.*, 01]. Ce travail consiste donc à découvrir des patrons de conception par aspects, plutôt qu'à étudier la transformation des patrons Objet.

4.2 Discussion des solutions proposées

4.2.1 Pourquoi cette transformation des patrons Objet est-elle utile ?

En comparant l'ensemble de ces travaux, nous remarquons que les apports d'utilisation des mécanismes et concepts Aspect, dans l'implémentation des patrons Objet, n'ont pas été présentés de la même façon par leurs auteurs. Par exemple, on peut considérer tout simplement l'utilité d'appliquer les patrons Objet dans le cadre d'une approche Aspect, ou discuter notamment, les apports en termes de réduction de dépendances, d'augmentation de réutilisation et/ou d'amélioration de maintenance et d'évolution à la fois des patrons et des applications.

Nous nous sommes concentrés, plus particulièrement, sur la possibilité de représenter de manière explicite, et de pouvoir donc localiser, les imitations des patrons au niveau du code. En effet, une idée importante de notre technique d'implémentation est que, pour tous les patrons sur lesquels nous nous penchons, nous nous efforçons d'aboutir à une solution dans laquelle une imitation d'un patron peut être identifiée par une seule unité modulaire (i.e. un *aspect* ou un *hyperslice*). Pour ce faire, nous considérons la description d'un patron plutôt comme la description d'un problème que comme celle de sa solution : nous retenons essentiellement l'intention des patrons pour exploiter au mieux les mécanismes d'AspectJ et Hyper/J. Notre motivation est d'améliorer la traçabilité des patrons dans le code, et améliorer par conséquent l'évolution et la réutilisation à la fois du code relatif aux imitations des patrons et de celui relatif à l'application. Ceci doit faciliter et améliorer, de plus, la rétro-conception des patrons qui est un problème difficile [Antoniol *et al.*, 98], [Paakki *et al.*, 00], [Arévalo 05].

4.2.2 Quels types de décomposition et de composition ?

L'implémentation des patrons en AspectJ et Hyper/J pose des problèmes de décomposition du code en classes et aspects ou hyperslices, et soulève par conséquent des problèmes de composition. Il existe en fait plusieurs possibilités pour la réalisation et l'application des patrons de conception dans une approche Aspect. La meilleure implémentation est certainement liée à la motivation principale de l'approche. Comme nous voulons principalement améliorer la traçabilité des patrons, nous choisissons d'avoir exactement un aspect ou un hyperslice pour dénoter une imitation d'un patron dans le code, tout en gardant quelques aspects abstraits (si le langage le permet) afin d'améliorer la réutilisation. On peut également choisir de regrouper toutes les imitations d'un même patron dans un seul aspect, pour une meilleure modularité et réutilisation (cette solution évite, dans le cas d'Hyper/J par exemple, de dupliquer la partie invariante d'un patron donné pour chaque imitation de celui-ci). On peut même penser à créer des aspects/hyperslices dénotant des préoccupations fonctionnelles de l'application, regroupant de la sorte plusieurs imitations de patrons distincts dans une même unité modulaire. D'ailleurs, toute réalisation d'un patron par aspects que nous proposons ne forme qu'une réalisation possible parmi plusieurs autres, comme nous l'avons déjà montré par exemple pour la solution en AspectJ du patron *Visiteur* (cf. § 2.1.1).

Au delà des choix et de la difficulté de décomposition, une autre difficulté réside dans la composition des différentes unités modulaires dénotant les patrons, avec celles qui représentent le reste de l'application. En effet, découpler les imitations des patrons dans plusieurs préoccupations séparées, augmente considérablement le nombre d'unités de l'application, qu'il faut savoir superposer et composer correctement.

4.2.3 Les patrons peuvent-ils disparaître ?

Les mécanismes et concepts Aspect permettent des implémentations directes pour certains patrons (tels que *Adaptateur*, *Visiteur*, *Décorateur*, etc. discutés dans la section 2.2). Les problèmes de ces patrons ne se posent alors plus dans le cadre d'une approche Aspect. Toutefois, nous pensons que ces patrons ne doivent pas disparaîtrent pour autant. En effet, un apport principal des patrons réside dans l'expertise qu'ils capitalisent : ils sont largement utilisés aujourd'hui par des développeurs qui s'y sont familiarisés et le besoin de réutiliser ces patrons se perpétuera. Le seul nom d'un patron, par exemple, peut suffire à résumer et expliquer toute une partie d'une application complexe. Pour ces raisons, nous sommes convaincus que les descriptions de ces patrons doivent subsister.

4.2.4 Les nouvelles solutions doivent-elles être dépendantes d'un langage de programmation ?

La plupart des travaux proposés, ainsi que les nôtres, ont mis en évidence les bénéfices que l'on peut tirer de l'implémentation des patrons Objet à l'aide des aspects, comparativement à une implémentation par objets. Toutefois, aucune structure de patrons de conception par aspects indépendante d'un langage de programmation n'a été proposée ; les implémentations considérées restant étroitement liées à un langage spécifique (*AspectJ* et *Hyper/J* pour la plupart, *AspectS* pour [Hirschfeld 02]). Il est important cependant de définir des solutions indépendantes des langages de programmation, afin de conserver l'esprit des patrons. Il convient alors de s'intéresser non seulement à leurs implémentations possibles, mais également à la description de leurs réalisations par aspects à un niveau d'abstraction plus élevé.

Une meilleure abstraction et représentation des nouvelles réalisations par aspects des patrons Objet est donc nécessaire, pour en améliorer leur réutilisation dès la phase de conception jusqu'à la phase d'implémentation. Toutefois, il n'existe en effet aujourd'hui que peu de consensus sur les concepts et mécanismes fondamentaux de la technologie Aspect et la diversité des langages proposés dans ce courant rend cette tâche difficile. Si la translation d'*AspectJ* vers *AspectS* est assez directe, la translation d'*AspectJ* vers *Hyper/J* est cependant beaucoup plus compliquée, car il s'agit de deux paradigmes de programmation différents. Cette situation peut constituer un obstacle quant à la définition de structures abstraites, comme elle peut supposer tout un travail de fond sur la définition de concepts et mécanismes communs à différents paradigmes Aspect. Il s'agit dans ce cas de déterminer l'ensemble de concepts communs et nécessaires à la définition de telles structures, travail que nous avons réalisé dans le cadre de cette thèse et que nous présentons dans les chapitres suivants.

4.2.5 Est-il possible de définir de nouveaux patrons Aspect ?

Les travaux considérés consistent principalement, comme c'est le cas pour notre proposition, à étudier la transformation des patrons Objet. Il est cependant intéressant de découvrir des patrons de conception spécifiques aux aspects, alors qu'*AspectJ* et d'autres modèles et langages de programmation par aspects commencent à être utilisés aujourd'hui dans le cadre de développements industriels. Il s'agit d'identifier des patrons qui répondant à des problèmes récurrents posés par la conception par aspects, plutôt que par les seules limites de la conception par objets.

Les analogies mises en évidence au niveau des structures par aspects que nous proposons pour les patrons de conception du *GoF*, nous laissent penser qu'il est possible de définir de nouveaux patrons de conception par aspects. Il est évident que si nous disposons d'un cadre conceptuel pour la représentation de ces structures par aspects, indépendamment d'un langage de programmation par aspects particulier, nous pouvons confirmer cette idée. De plus, il est intéressant, dans ce cas, de définir un formalisme adéquat pour la représentation et la description uniforme de tous ces patrons. Ceci nous permettra également d'organiser l'ensemble des patrons considérés et d'en préciser leurs relations.

5. Conclusion

Les patrons de conception sont reconnus comme étant très utiles pour améliorer la qualité et l'évolution des systèmes, nous avons ainsi besoin de techniques d'implémentation appropriées afin de les utiliser proprement dans le développement de logiciels de plus en plus complexes. Les techniques d'implémentation basées exclusivement sur les mécanismes et concepts Objet (tels que l'héritage, la composition, la délégation explicite, etc.) ne sont pas satisfaisantes, car elles n'assurent pas une bonne traçabilité et une réutilisation efficace des imitations des patrons au niveau de l'implémentation. D'autant plus que de telles imitations posent plusieurs problèmes (problème de confusion, d'indirection, de rupture d'encapsulation, et les problèmes liés à l'héritage (cf. § 3.3, chapitre 1)) qui sont préjudiciables à l'évolution et à la réutilisation des applications utilisant les patrons. Les modèles et langages de programmation introduits dans le cadre de l'approche Aspect offrent de nouveaux concepts et mécanismes permettant de nouvelles solutions de réalisation des patrons, qui peuvent résoudre les problèmes liés à l'utilisation de ces patrons et favoriser la traçabilité et la réutilisation de leurs imitations. En effet, nous avons proposé dans ce chapitre de nouvelles solutions de réalisation des 23 patrons de conception du GoF à l'aide d'AspectJ et Hyper/J, et nous avons discuté de la manière dont les mécanismes et concepts de ces deux langages par aspects peuvent résoudre les problèmes liés à l'utilisation des patrons, augmenter leur réutilisation, et assurer la traçabilité de leurs imitations au niveau du code. Nous avons particulièrement détaillé les solutions par aspects que nous proposons pour les patrons *Visiteur*, *Observateur* et *Singleton*, et nous avons illustré les apports de ces nouvelles solutions à travers un exemple d'application. Nous avons considéré ensuite les solutions des vingt autres patrons du GoF que nous avons présentées brièvement, pour confirmer les apports identifiés. Ce travail exhaustif sur les 23 patrons du GoF montre un autre résultat principal d'«aspectisation» de patrons de conception par objets : il s'agit d'une nouvelle classification des patrons du GoF, basée principalement sur les types d'adaptations qu'apportent ces patrons sur les classes d'applications.

De manière générale, nous avons recherché des solutions aboutissant à la localisation de chaque imitation d'un patron par exactement une unité modulaire (*aspect* dans le cas d'AspectJ et *hyperslice* dans le cas d'Hyper/J) dans l'ensemble du code d'une application. Une telle correspondance nous a permis en premier lieu de résoudre le problème de confusion et de traçabilité, et devrait dans la pratique permettre de mieux profiter des avantages reconnus des patrons de conception (facilité d'apprentissage et d'application, réutilisation [Gamma *et al.*, 93]) jusqu'à la phase d'implémentation. Aisément localisables dans le code, les imitations de patrons peuvent être modifiées, retirées ou ajoutées, ou encore remplacées par l'imitation équivalente d'un patron apparenté (voir [Gamma *et al.*, 95] pour la définition d'un patron apparenté). Ces choix d'implémentation offrent de nouvelles perspectives pour la rétro-conception des applications intégrant de telles imitations de patrons. Il devient possible de définir des métriques portant sur l'utilisation de patrons dans une application. De plus, les définitions des classes de ces applications sont inchangées car complètement indépendantes des imitations des patrons, et les problèmes d'indirection, de rupture d'encapsulation et ceux liés à l'héritage sont en conséquence évités.

Nous avons présenté une discussion de l'ensemble des nouvelles solutions par aspects que nous proposons, mais aussi des solutions proposées dans le cadre de plusieurs autres travaux connexes, visant à améliorer l'implémentation des patrons de conception par objets à l'aide des concepts et mécanismes aspects. Les nouvelles solutions de réalisation par aspects de patrons présentent cependant l'inconvénient d'être dépendantes d'un langage de programmation par aspects particulier. Il s'agit en effet de solutions d'implémentation spécifiques à un langage donné ; ce ne sont pas des solutions de conception génériques et réutilisables par plusieurs applications basées sur des techniques d'implémentation différentes. Il convient de proposer des solutions indépendantes des techniques d'implémentation, afin de mieux

préserver l'esprit des patrons de conception. Toutefois, un certain manque de consensus sur les concepts et relations fondamentales de la technologie Aspect et la diversité des modèles et langages de programmation proposés dans ce courant rendent difficile l'expression de nouvelles structures de patrons indépendantes d'un langage de programmation spécifiques. Nous proposons dans le chapitre suivant de pallier cette difficulté en adoptant une démarche par métamodélisation et transformation de modèles, et en nous appuyant sur un métamodèle général pour la modélisation par aspects. Nous avons élaboré ce métamodèle en isolant les concepts et relations communs à *AspectJ* et *Hyper/J*.

Chapitre 4

Métamodélisation, modélisation par aspects et transformation de modèles

Nous avons relevé dans le premier chapitre plusieurs problèmes et limites liés à l'utilisation des patrons de conception par objets. Nous avons proposé ensuite, dans le chapitre précédent, de nouvelles solutions notamment pour les patrons de conception du GoF [Gamma *et al.*, 95], en tirant profit des mécanismes et concepts des nouvelles techniques de programmation introduites dans le cadre de l'approche Aspect, et plus précisément dans les langages AspectJ [Kiczales *et al.*, 01] et Hyper/J [Ossher *et al.*, 00], pour éviter ces problèmes et limites. Ce travail, ainsi que les travaux de [Noda *et al.*, 01], [Nordberg 01a], [Nordberg 01b], [Hannemann *et al.*, 02] et [Hirschfeld *et al.*, 03] ont mis en évidence les bénéfices que l'on peut tirer d'une implémentation par aspects des patrons de conception, comparativement à une implémentation strictement Objet. Ces travaux se heurtent cependant à la difficulté d'exprimer des structures de patrons de conception qui soient indépendantes d'un langage spécifique de programmation par aspects, pour ne proposer que des solutions considérées très étroitement liées à l'un de ces langages (AspectJ, Hyper/J, ou encore AspectS [Hirschfeld 02]). Il n'existe en effet aujourd'hui que peu de consensus sur les concepts fondamentaux de l'approche Aspect, et la diversité des modèles et langages de programmation proposés dans ce courant rend difficile l'expression de nouvelles structures par aspects de patrons indépendamment d'un langage de programmation particulier.

Pour aborder cette difficulté, nous proposons dans le cadre de notre travail d'adopter une approche par métamodélisation et transformation de modèles. Ce chapitre présente et compare deux extensions du métamodèle d'UML [OMG-UML 04], prenant en compte respectivement les concepts spécifiques à AspectJ et Hyper/J, deux des principaux langages par aspects les plus connus et aboutis aujourd'hui. Un rapprochement des concepts des deux métamodèles obtenus nous a permis d'identifier un ensemble d'éléments de modélisation communs. Nous avons ainsi proposé et défini un métamodèle plus général à l'approche Aspect, intégrant des concepts Aspect plus abstraits que ceux qui sont spécifiques à ces deux langages. Ce chapitre présente ce métamodèle général et les deux métamodèles spécifiques, ainsi qu'un ensemble de règles de transformation de modèles, permettant de passer d'un modèle instance du métamodèle général à des modèles instances des métamodèles spécifiques respectivement à AspectJ et Hyper/J.

La figure 4.1 synthétise l'approche que nous proposons en présentant les trois niveaux de modélisation. Elle montre la séquence d'activités en termes de modélisation et métamodélisation, ainsi qu'en termes de transformation de modèles.

Le reste de ce chapitre est organisé comme suit. La section 1 discute différents travaux menés sur la modélisation des aspects, dans le but de motiver et d'argumenter notre proposition. La section 2 présente les trois métamodèles que nous proposons. La section 3 rappelle le principe de l'ingénierie dirigée par les modèles (IDM ou *MDA* pour *Model Driven Architecture* [OMG-MDA 03]) et introduit les principales approches de transformation de modèles existantes, dans le but de définir le processus de transformation que nous adoptons dans notre proposition. Elle présente ensuite l'ensemble des règles de transformation que nous proposons, respectivement, pour la projection vers AspectJ ou Hyper/J. Enfin, la section 4 conclut ce chapitre. Nous illustrons toute notre approche de modélisation et de transformation de modèles sur l'exemple du patron *Stratégie*.

1. Analyse des approches existantes et objectifs

Dans le but de motiver et d'argumenter notre choix pour l'approche par métamodélisation et transformation de modèles, nous présentons et analysons dans cette section les principaux travaux qui se sont intéressés à la modélisation des aspects. Nous définissons ensuite nos objectifs et précisons l'approche adoptée.

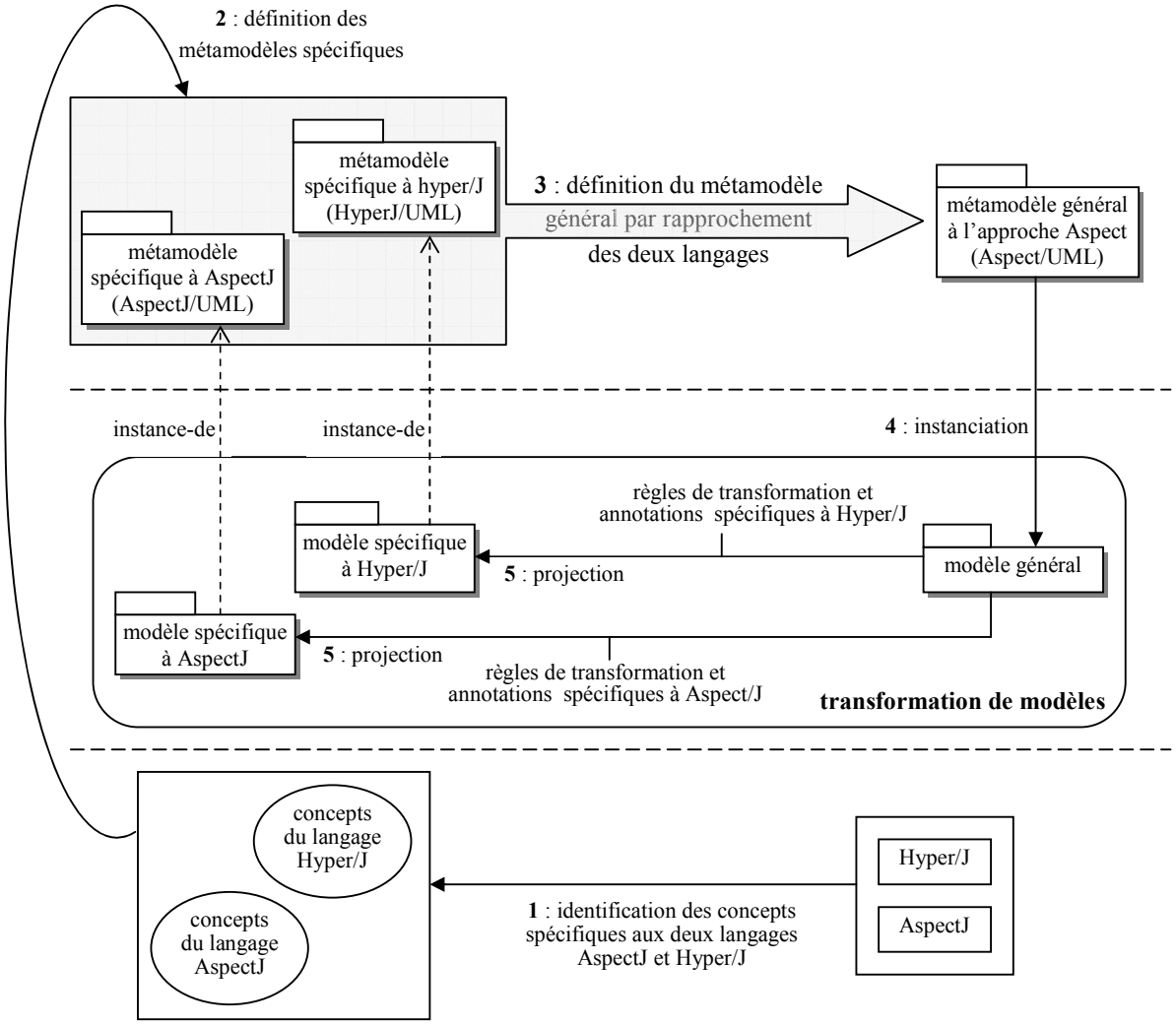


Figure 4.1. *Approche adoptée*

1.1 Modélisation des aspects : limites des approches existantes

Si les termes de programmation par aspects et de séparation avancée des préoccupations (*Advanced Separation of Concerns*) sont de plus en plus largement utilisés, la plupart des enjeux relatifs à l'analyse et à la conception par aspects relevés lors de leur introduction, notamment dans [Kickzales *et al.*, 97], sont encore d'actualité de nos jours : qu'est ce que l'analyse et la conception par aspects ? Comment identifier les aspects au niveau de ces phases préalables à la programmation ? Etc. Il règne en particulier toujours un « flou » sur les concepts fondamentaux de ce courant et leurs relations, que la diversité des propositions et des travaux menés dans ce cadre contribue à entretenir. Si cette imprécision présente au moins l'avantage de ne pas restreindre le champ des investigations et des possibilités dans l'élaboration de nouveaux modèles et langages de programmation par aspects, nous estimons qu'elle nuit cependant à l'utilisation effective de ces modèles et langages pour le développement de logiciels à moyenne et grande échelle.

Plusieurs travaux récemment proposés se sont ainsi intéressés à la séparation des préoccupations transversales (*aspects*, *hyperslices*, etc.) tout au long du cycle de développement, et en particulier au niveau de la phase de conception. Nous présentons et discutons ci-dessous la plupart de ces approches ou

propositions[10] de modélisation, dont les principales caractéristiques sont résumées dans le tableau 4.1. La synthèse de ces propositions est basée sur les propriétés suivantes :

- *Portée*. La portée d'une proposition détermine la phase du cycle de développement considérée par celle-ci : la phase d'analyse et de recensement des besoins (*analysis*), ou la phase de conception (*design*).
- *Couverture*. La couverture d'une approche traduit sa faculté d'abstraction, et définit donc son champ d'application. Elle peut être générale (*general*) ou spécifique (*specific*) à un domaine d'application particulier (tel que le développement d'outils de test et de pilotage par exemple).
- *Modèle support*. La plupart des travaux que nous considérons proposent, le plus souvent, de définir un ensemble de concepts et relations spécifiques à un modèle ou à un langage de programmation par aspects particulier, tels que *Composition patterns*, *AOP*, AspectJ, Hyper/J, etc. On parle dans ce cas de modèle support spécifique. Il existe cependant quelques travaux qui proposent des concepts et des relations indépendants d'un modèle ou d'un langage de programmation en particulier : le modèle support est alors dit général (*general*).
- *Extension*. La plupart des approches existantes proposent, pour la définition de leurs concepts et relations Aspect, d'étendre le métamodèle d'UML. Elles se distinguent cependant par leurs façons de faire : extension par stéréotypage (*stereotyping*), par métamodélisation (*metamodeling*), ou proposition d'un *profile* UML.
- *Exhaustivité*. Une proposition peut modéliser l'ensemble complet des concepts et relations de son modèle ou langage support, comme elle peut considérer exclusivement les principaux éléments de cet ensemble et ignorer le reste. Nous distinguons donc deux types de propositions : complète (*complete*) ou partielle (*partial*).
- *Vues*. Pour représenter un système à base d'aspects il convient le plus souvent de s'intéresser non seulement à la modélisation de sa structure statique, mais aussi à la modélisation de sa partie dynamique et donc à l'expression du résultat du tissage des aspects. Une proposition peut s'intéresser exclusivement à la vue structurelle (*structural*) ou comportementale (*behavioural*) d'un système, comme elle peut considérer les deux vues.
- *Notation graphique* Les travaux actuels se concentrent le plus souvent sur la définition de la sémantique précise des concepts et relations qu'ils proposent. Il existe cependant des propositions qui offrent en plus des syntaxes concrètes pour la représentation graphique de leurs éléments de modélisation par aspects. Dans le tableau 4.1 la colonne correspondante à cette propriété indique si l'approche considérée propose une syntaxe concrète ou non.

[10] La liste des propositions présentées dans le tableau 4.1 n'est pas exhaustive. Nous avons choisi de nous concentrer sur les principales propositions de modélisation des aspects les plus abouties et acceptées.

Tableau 4.1. *Propriétés caractéristiques des principales approches de modélisation des aspects*

Approche	Portée	Couverture	Modèle support	Exhaustivité	Extension	Vues	Notation graphique
[Suzuki *et al.*, 99]	design	general	AspectJ	partial	metamodeling and stereotyping	structural	yes
[Clarke 01]	analysis design	general	Composition Patterns [Clarke *et al.*, 01]	complete	metamodeling (Theme/UML)	structural behavioural	yes
[Aldawud *et al.*, 01] [Aldawud *et al.*, 03]	design	general	AOP (AspectJ)	partial	UML profile by stereotyping	structural	yes
[Stein 02]	design	general	AspectJ	complete	stereotyping	structural behavioural	yes
[Lions *et al.*, 02]	design	general	AspectJ	complete	metamodeling	structural	yes
[Zakaria *et al.*, 02]	design	general	AspectJ	partial	stereotyping (based on the UML profile of [Aldawud *et al.*, 01])	structural	yes
[Chavez *et al.*, 02]	design	general	general (but mainly inspired by *AOP* and AspectJ)	complete	metamodeling	structural	yes
[Herrmann 02]	design	general	Aspectual Collaborations [Herrmann *et al.*, 01], [Lieberherr *et al.*, 99]	complete	metamodeling	structural	yes
[Kandé *et al.*, 02]	design	general	AOP (AspectJ)	complete	stereotyping	structural behavioural	yes
[Low 02]	design	specific (for TAST[11])	AspectJ	partial	none	structural	yes
[Basch *et al.*, 03]	design	general	AspectJ	complete	UML profile by stereotyping	structural behavioural	yes
[Groher *et al.*, 03]	design	general	AspectJ and UFA [Herrmann 02]	partial	stereotyping	structural	yes
[Phillipow *et al.*, 03]	design	specific (for product lines)	Hyperspace (Hyper/J)	partial	metamodeling (Hyper/UML)	structural behavioural	yes
[Mellor 03]	design	general	none	indefinite	UML profile	structural	no
[Kulkarni *et al.*, 03]	design	general	none	indefinite	none (specification of aspects as models and weaving as model transformation)	structural	no
[Han *et al.*, 04a] [Han *et al.*, 04b]	design	general	AspectJ	complete	metamodeling, based on a Java metamodel	structural	yes

[11] TAST : Toolkit for ASpect-oriented Tracing

L'une des premières propositions de modélisation des aspects est celle de Suzuki et Yamamoto [Suzuki *et al.*, 99], elle intègre les aspects au niveau de la phase de conception pour la modélisation de systèmes réalisés plus précisément à l'aide d'AspectJ et qui sont indépendants d'un domaine d'application en particulier. Elle étend le métamodèle d'UML par métamodélisation et stéréotypage, et propose une notation pour la visualisation des aspects. Cette extension consiste notamment en l'ajout d'une nouvelle métaclasse nommée Aspect qui spécialise la métaclasse Classifier du métamodèle d'UML. Un aspect peut ainsi contenir un nombre arbitraire d'attributs et d'opérations et peut participer dans plusieurs associations, généralisations et avoir différentes relations de dépendance. Les introductions d'opérations et les perfectionnements (*advice*) sont définis comme étant des stéréotypes d'opérations de nom « weave » associés à une contrainte supplémentaire dépendante du type de l'élément modélisé (comme {introduction} pour les introductions d'opérations par exemple). Suzuki et Yamamoto proposent de modéliser la relation liant un aspect à une classe de base par une relation de dépendance de type abstraction (*abstraction dependency relationship*) stéréotypée par « realize ». C'est l'un des inconvénients de cette approche, car une réalisation est une relation entre une spécification et l'élément qui implémente cette spécification, alors que la relation d'entrecroisement entre aspects et classes a une sémantique complètement différente. Un autre inconvénient principal de cette approche est qu'elle est partielle, car elle ne spécifie pas, par exemple, comment modéliser les déclarations d'héritage (*declare parents*), les points de recouvrement (*pointcut*), etc. Cette approche offre, cependant, une notation graphique pour la représentation des aspects et les introductions d'attributs et d'opérations, mais pas pour les perfectionnements[12]. Elle propose aussi de montrer le résultat statique du tissage par la représentation de la structure des classes finales du système à modéliser, en utilisant des paquetages détenant, respectivement, les classes de base avec leurs structures originales et les aspects qui les entrecroisent, et les classes après tissage avec leurs nouvelles structures. Il s'agit, en effet, d'une première tentative de modélisation des aspects spécifique à un langage particulier de programmation (AspectJ). Elle est néanmoins insuffisante, car seulement quelques concepts d'AspectJ sont modélisés et de plus certains éléments de modélisation offerts sont inappropriés aux aspects (telle que la relation de réalisation pour modéliser une relation d'entrecroisement).

Plusieurs autres tentatives de modélisation des aspects, dont le modèle support est AspectJ, ont été proposées ensuite. La plupart de ces propositions préconisent d'étendre UML par l'usage exclusif des stéréotypes, des valeurs marquées et des contraintes. Nous pouvons citer par exemple les travaux de Aldawud *et al.* [Aldawud *et al.*, 01], [Aldawud *et al.*, 03] qui ont constitué la base de l'approche présentée dans [Zakaria *et al.*, 02]. Il s'agit de deux propositions dont les couvertures sont générales et qui offrent des éléments de modélisation pour quelques (mais pas tous les) concepts définis par AspectJ. Elles se concentrent par ailleurs, exclusivement, sur la représentation structurelle d'un système donné. Une autre proposition de couverture générale, plus complète que ces deux dernières, est celle de Stein [Stein 02]. Celle-ci propose des éléments de modélisation pour l'ensemble des concepts et mécanismes offerts par AspectJ, dans le moindre des détails. Elle propose, de plus, de nouveaux éléments de modélisation (ou stéréotypes UML) qui sont dédiés à la représentation dynamique d'un système à base d'AspectJ. Ces trois propositions s'accordent, toutefois, pour la définition d'un aspect par un stéréotype particulier de classe (la métaclasse Class du métamodèle d'UML), alors que d'autres propositions étendant UML par stéréotypage considèrent un aspect comme étant un stéréotype particulier de paquetage (la métaclasse Package) [Basch *et al.*, 03] ou de collaboration (la métaclasse Collaboration) [Kandé *et al.*, 02]. Ces deux dernières propositions ([Basch *et al.*, 03] et [Kandé *et al.*, 02]) se caractérisent aussi par une couverture générale ; elle

[12] Suzuki et Yamamoto considèrent les perfectionnements comme étant des stéréotypes particuliers d'opération. Toutefois, ils ne montrent pas, dans leur exemple support, comment représenter ces perfectionnements.

couvrent l'ensemble des concepts d'AspectJ et s'intéressent (comme dans le cas de la proposition de Stein) à la définition d'éléments de modélisation par aspects nécessaires à la représentation de la partie dynamique d'un système réalisé à l'aide d'AspectJ. Nous distinguons par ailleurs deux autres propositions de modélisation par aspects ayant pour modèle de base celui d'AspectJ, il s'agit des approches présentées respectivement par [Lions *et al.*, 02] et [Han *et al.*, 04a], [Han *et al.*, 04b]. À la différence de toutes les propositions discutées précédemment, celles-ci préconisent d'étendre le métamodèle d'UML par métamodélisation, comme le fait [Suzuki *et al.*, 99]. Elles proposent en effet d'étendre le métamodèle d'UML par l'ajout de nouvelles métaclasses définissant des concepts spécifiques à AspectJ, tels que les points de recouvrement (*pointcut*) et les perfectionnements (*advices*), etc. [Han *et al.*, 04a] proposent de fournir un langage de modélisation par aspect complet, notamment, pour la représentation structurelle de systèmes indépendants d'un domaine d'application particulier. Ils se basent pour ce faire sur un métamodèle dédié au langage Java, lui-même défini comme étant une extension du métamodèle d'UML par métamodélisation. [Lions *et al.*, 02] proposent par ailleurs pour valider leur approche d'extension d'OpenTool/UML par métamodélisation, de créer un métamodèle spécifique à la modélisation structurelle de systèmes à base d'AspectJ indépendants d'un domaine d'application particulier. Leur proposition est dite complète car elle couvre tout l'ensemble des concepts de leur modèle support, en l'occurrence AspectJ. Enfin, remarquons que tous les travaux spécifiques à AspectJ présentés jusqu'ici ont une couverture générale, ce qui n'est pas le cas pour la proposition de Low [Low 02] qui est spécifique au domaine de développement de systèmes ou d'outils de « *tracing* » (TAST pour Toolkit for ASpect-oriented Tracing).

La plupart des travaux considérés dans le tableau 4.1 offrent de nouveaux éléments de modélisation par aspects, qui sont étroitement liés aux concepts et mécanismes introduits par un langage de programmation par aspects en particulier, le plus souvent AspectJ. Il existe cependant d'autres propositions dont les modèles supports sont différents d'AspectJ : la proposition de [Clarke 01] est basée sur le modèle *Composition Patterns*, celle de [Herrmann 02] est basée sur le modèle *Aspectual Collaboration*s, et enfin, celle de [Phillipow *et al.*, 03] est basée sur le modèle de l'approche *Hyperspace* et donc sur celui de son langage Hyper/J. Nous distinguons par ailleurs le travail de [Grother *et al.*, 03] qui repose sur les concepts du modèle UFA [Herrmann 02] et quelques uns des concepts d'AspectJ (notamment *pointcut* et *advice*) qui ne sont pas définis par ce dernier modèle. Ces quatre propositions ont le plus souvent une couverture générale, seule la proposition de [Phillipow *et al.*, 03] est dédiée au développement des lignes de produit (*product lines*). Elles proposent par ailleurs d'étendre UML par métamodélisation, à l'exception de l'approche de [Grother *et al.*, 03] qui étend UML par stéréotypage. Cette dernière approche traite notamment comme [Herrmann 02] de la représentation structurelle des systèmes à base d'aspects, alors que celles de [Phillipow *et al.*, 03] et [Clarke 01] considèrent également la représentation des parties dynamiques de tels systèmes.

Par opposition à toutes les approches analysées ci-dessus, seuls les travaux de [Mellor 03] et [Kulkarni *et al.*, 03] sont indépendants d'un modèle ou d'un langage de programmation par aspects en particulier. [Chavez *et al.*, 02] prétendent aussi offrir un langage de modélisation indépendant, il est toutefois principalement basé sur les concepts de l'AOP et inspiré de ceux spécifiques à AspectJ. Par ailleurs, bien qu'elle soit exclusivement basée sur le modèle *Composition Patterns*, la proposition de [Clarke 01] peut être également considérée comme étant indépendante d'un langage ou modèle. En effet, Clarke propose un métamodèle permettant la conception de patrons de composition par aspects, et donne des directives pour la projection de tels patrons vers AspectJ et Hyper/J. L'analyse de ces travaux indépendants montre que [Mellor 03] et [Kulkarni *et al.*, 03] proposent des approches de modélisation très différentes de l'ensemble du reste des travaux considérés dans cette section. D'une part, [Mellor 03] intègre les aspects au niveau de la conception en utilisant le principe de MDA [OMG-MDA 03]. Il se concentre sur la modélisation du

tissage des aspects, tissage réalisé grâce à la transformation de modèles. D'autre part, dans [Kulkarni *et al.*, 03], l'approche MDA, la spécification de modèles d'aspects et de modèles de composants sont considérés. Le tissage d'un aspect avec un composant est également vu comme une transformation du modèle du composant considéré.

De manière générale l'analyse des travaux évalués dans le tableau 4.1 montre que :

- Toutes les propositions de modélisation par aspects sont dédiées à la phase de conception, seule celle de [Clarke 01] est en plus dédiée à la phase d'analyse. De manière générale, ces propositions se concentrent sur la représentation des structures statiques des systèmes, mais très rarement sur la représentation de leurs comportements. Elles offrent d'ailleurs, le plus souvent (à l'exception de [Low 02] et [Phillipow *et al.*, 03]), des langages de modélisation de couverture générale, de sorte qu'un même langage peut être utilisé pour l'expression de tout type d'aspects spécifique à un domaine d'application particulier.
- UML, le langage standard de modélisation par objets, est en général considéré comme un point de départ à la définition d'un langage de modélisation par aspects, car il s'agit d'un langage standard et générique qui peut être facilement étendu et adapté.
- Deux approches d'extension d'UML sont largement utilisées : stéréotypage ou métamodélisation. Toutefois, bien que ces deux techniques soient complémentaires, les propositions de modélisation actuelles choisissent, en général, l'une ou l'autre de ces deux techniques en raison de leurs avantages et inconvénients. Nous pensons cependant qu'une combinaison de ces deux techniques permet d'obtenir une extension d'UML pour les aspects qui soit plus avantageuse.
- Les propositions les plus abouties sont étroitement liées à un modèle ou un langage de programmation par aspects en particulier, le plus souvent AspectJ. Nous admettons que ces propositions sont intéressantes, car elles offrent des langages de modélisation ayant des niveaux d'abstraction plus élevés que ceux de leurs langages de programmation supports. De tels langages de modélisation présentent, cependant, l'inconvénient d'être spécifiques à une technique particulière. Ceci tient principalement au fait qu'ils sont en général conçus dans le but de pouvoir idéalement être utiles à la phase d'implémentation, et donc offrir la possibilité de générer du code Aspect à partir des modèles de conception. Nous pensons donc qu'il convient de définir un langage de modélisation plus général, permettant de spécifier des aspects à un niveau d'abstraction encore plus élevé, de manière indépendante des différentes techniques cibles.
- Souvent, les concepts fondamentaux des modèles supports ne sont pas complètement spécifiés. En effet, certains de ces concepts qui forment les briques de base de toute modélisation Aspect sont omis dans certaines propositions. Les éléments de modélisation nouvellement introduits pour spécifier de tels concepts sont parfois inappropriés : relation de réalisation pour la définition de la relation de *crosscutting*, classe ou collaboration pour la spécification d'un *aspect*, etc. Nous sommes convaincus qu'un effort initial doit porter alors sur l'identification et puis la définition de la sémantique exacte de ces concepts fondamentaux.

Si ces travaux sont intéressants, ils présentent toutefois quelques inconvénients (souvent très spécifiques à un domaine d'application particulier, ou à une technique d'implémentation particulière, etc.) et ne répondent pas parfaitement à nos objectifs que nous précisons dans ce qui suit. Nous proposons également de montrer nos choix de modélisation, tout en argumentant notre préférence pour l'approche par métamodélisation et transformation de modèles.

1.2 Objectifs et approche adoptée

Notre motivation principale, dans le cadre de cette thèse, est de pouvoir représenter des structures par aspects de patrons de conception qui soient indépendantes d'un modèle ou d'un langage de programmation par aspects en particulier. Nous voulons toutefois disposer d'un moyen permettant, dans un contexte plus large, l'expression de toute structure d'un système réalisé à base d'aspects. Nous proposons pour ce faire de définir un langage de modélisation basé sur un métamodèle que nous souhaitons général à l'approche Aspect, et applicable à différents domaines d'application. Un tel métamodèle devra décrire de manière formelle la syntaxe et la sémantique de tous les concepts (qui peuvent être comparables aux définitions des concepts et mécanismes de l'Objet proposées dans [Wegner 87] par exemple) ou éléments fondamentaux à la modélisation par aspects. Toutefois, dans le souci principal de pouvoir modéliser des structures statiques, nous nous concentrons uniquement dans le cadre de ce travail sur la définition d'éléments de modélisation nécessaires notamment à la représentation structurelle. Nous proposons par ailleurs de considérer UML [OMG-UML 04] comme base à la définition de notre langage de modélisation par aspects. En effet, tout comme les modèles et langages de programmation par aspects actuellement proposés sont très généralement construits comme des extensions, ou du moins des évolutions, de modèles et de langages de programmation par objets, la plupart des travaux portant sur les techniques de conception et de modélisation par aspects s'appuient très fortement sur les résultats acquis dans le cadre de l'approche Objet. En ce sens, UML le langage standard de modélisation par objets, est le plus souvent considéré comme point de départ à la définition d'un langage de modélisation par aspects, comme nous pouvons le constater d'ailleurs à travers l'analyse des principales propositions discutées ci-dessus. Ce choix est également justifié par plusieurs autres avantages d'UML :

- UML facilite l'expression et la communication d'une variété de modèles de conception qui permettent de modéliser de manière claire et précise la structure et le comportement d'un système d'information, indépendamment de toute méthode de développement ou de tout langage de programmation par objets.
- UML se base sur une syntaxe abstraite et une sémantique définissant un ensemble de concepts[13] et leurs relations, qui forment les briques de base de la modélisation Objet. Il offre en plus une notation graphique dont la syntaxe concrète est à la fois simple, intuitive et expressive, pour pouvoir communiquer facilement les différents modèles de conception.
- UML est fondé sur un métamodèle qui décrit de manière formelle tous les éléments de modélisation d'UML (i.e. les différents concepts et leurs relations), qui sont eux-mêmes modélisés avec UML. Cette définition récursive, appelée métamodélisation, présente un double avantage : elle permet de classer les concepts et relations par niveaux d'abstraction, et aussi de faire la preuve de la puissance d'expression de la notation UML, capable entre autres de ce représenter elle-même.
- UML facilite l'interopérabilité : les modèles UML peuvent être facilement spécifiés sous formes textuelles conformes à des standards OMG telles que XML et XMI, etc., afin de faciliter l'exploitation de leurs données.

[13] Dans l'annexe A nous rappelons notamment la syntaxe abstraite des principaux éléments de modélisation des paquetages « *Foundation* » et « *Model Management* » d'UML, qui sont complètement décrits de manière précise dans le document de spécification de ce langage [OMG-UML 04], et que nous considérons comme éléments de base pour la définition des nouveaux métamodèles que nous proposons dans ce chapitre.

- UML se veut un langage de modélisation non fermé, suffisamment général, extensible, et adaptable à plusieurs domaines d'applications spécifiques. Il se base en général pour ce faire sur un ensemble de mécanismes d'extension définis par son métamodèle : stéréotypes, valeurs marquées et contraintes. Il s'agit de l'extension d'UML par stéréotypage. Il est possible également d'étendre UML par métamodélisation.

Il convient de noter ici que chacune des deux techniques d'extension du métamodèle d'UML présente des avantages et des inconvénients que nous discutons dans ce qui suit, afin d'argumenter notre préférence à l'utilisation conjointe de ces deux techniques. Nous précisons ensuite l'approche que nous adoptons, d'une part, pour la définition de notre métamodèle général, et d'autre part, pour la modélisation par aspects.

1.2.1 Métamodélisation ou Stéréotypage

UML définit un ensemble d'éléments de modélisation (i.e. concepts et relations) dont chacun est représenté par une métaclasse au niveau de son métamodèle. Il offre également des mécanismes d'extension de ces éléments : les stéréotypes (*stereotypes*), les valeurs marquées (*tagged values*) et les contraintes (*constraints*), permettent l'extension et la spécialisation des classes de concepts et relations standards d'UML. Les stéréotypes permettent d'ajouter de nouvelles classes d'éléments de modélisation au métamodèle d'UML, par dérivation des métaclasses existantes, en plus du noyau prédéfini par UML. Les valeurs marquées étendent les attributs des classes d'éléments du métamodèle et les contraintes sont des relations sémantiques entre éléments de modélisation qui définissent des conditions que doit vérifier le système. Ces mécanismes ont été exploités dans le cadre de plusieurs travaux, visant à étendre UML pour intégrer les aspects, comme [Suzuki *et al.*, 99], [Stein 02], etc. (cf. tableau 4.1).

Une autre possibilité d'extension d'UML consiste à étendre son métamodèle en y ajoutant de nouveaux éléments originaux de modélisation et donc de nouveaux concepts et relations. Elle peut être complémentaire de l'utilisation des mécanismes d'extension. C'est cette approche par métamodélisation que nous adoptons dans notre proposition. En effet, l'usage exclusif des mécanismes d'extension nous semble insuffisant parce qu'il repose trop fortement sur la définition de stéréotypes : introduire un concept ou une relation propre à l'approche Aspect en se limitant à la définition d'un stéréotype consiste plus à se rapprocher superficiellement et artificiellement d'un élément de modélisation déjà existant dans UML que de définir un nouvel élément et de rechercher sa place dans le noyau d'UML déjà constitué. Nous pouvons, par exemple, critiquer le fait de considérer un aspect comme un stéréotype de la métaclasse Class du métamodèle d'UML comme le proposent [Suzuki *et al.*, 99], [Stein 02], etc., alors que la définition originale d'un aspect dans [Kiczales *et al.*, 97] oppose justement ces deux concepts. De plus, l'utilisation de stéréotypes peut très rapidement conduire à une divergence conséquente des extensions proposées comme le montre l'exemple de la spécification d'un aspect par un stéréotype de classe d'une part, et par un stéréotype de paquetage (Package) [Basch *et al.*, 03] ou de collaboration (Collaboration) [Kandé *et al.*, 02] d'autre part. La métamodélisation offre de plus amples possibilités de structuration dans la définition de nouveaux éléments de modélisation tout en permettant de les mettre en relation entre eux, mais également avec les concepts standards déjà présents dans le noyau d'UML. En particulier, l'usage de la généralisation permet de dégager des concepts et relations suffisamment abstraits pour pouvoir rapprocher de manière significative des modèles ou langages basés sur des concepts voisins mais néanmoins différents.

On peut noter, enfin, que l'approche par métamodélisation présente un inconvénient principal par rapport à l'approche par stéréotypage, ou *profile* UML. Il est difficile, en effet, d'utiliser un outil de modélisation standard UML déjà existant, pour exprimer des modèles instances du nouveau métamodèle résultat de l'extension. Nous tenons pourtant à adopter cette approche par métamodélisation dans la

définition des nouveaux concepts et relations de notre langage de modélisation, pour les raisons que nous avons avancées auparavant. Par ailleurs, pour la définition des types d'attributs des nouveaux concepts Aspect, et la définition de la notation graphique de ces concepts et leurs relations, nous proposons de plus d'adopter l'approche par stéréotypage. Nous réutilisons certains des éléments de visualisation déjà existants auxquels nous associons de nouveaux stéréotypes, et nous offrons de plus de nouveaux éléments de visualisation notamment pour les relations nouvellement introduites.

1.2.2 Approche adoptée : métamodélisation et transformation de modèles

Comme nous l'avons évoqué auparavant, l'effort initial de notre travail porte sur l'identification et la définition de la sémantique des concepts et relations de l'approche Aspect, pour la définition du métamodèle de notre langage de modélisation par aspects. Ces concepts forment les briques de base de la modélisation par aspects ; les développeurs doivent pouvoir les échanger facilement. Il faut donc s'accorder ensuite sur l'importance relative de chaque concept et choisir une représentation graphique dont la syntaxe soit à la fois simple, intuitive et expressive. Toutefois, un certain manque de consensus sur les principaux concepts et relations de l'approche Aspect, et la diversité des langages de programmation proposés dans ce courant, rendent difficile l'identification et la spécification d'éléments de modélisation généraux à cette approche. Nous proposons de définir et comparer deux extensions par métamodélisation d'UML, tenant compte des concepts introduits respectivement dans AspectJ et Hyper/J, deux des langages les plus connus et utilisés parmi ceux qui se réclament de l'approche Aspect [Aksit *et al.*, 92], [Pawlak 02], [Hirscheld 02], etc. Ces deux langages représentent, en effet, les deux modèles de programmation par aspects les plus aboutis : l'*AOP* (modèle asymétrique) et la séparation multidimensionnelle des préoccupations (modèle symétrique) qui sont complètement différentes. Les deux extensions du métamodèle d'UML que nous proposons nous permettent d'établir un rapprochement entre les deux langages, par l'identification de concepts et relations communs présents dans les deux nouveaux métamodèles résultats. Cet ensemble de concepts et relations constitue ainsi une base pour la définition de notre métamodèle général à l'approche Aspect. Parmi les travaux proposant une extension d'UML par métamodélisation pour les aspects, tels que [Clarke 01], [Chavez *et al.*, 02], etc. (cf. tableau 4.1), notre travail se distingue essentiellement par le fait que notre but est de rechercher des analogies sémantiques entre les concepts et relations propres à différents métamodèles de plusieurs langages de programmation par aspects, afin de pouvoir à terme identifier et isoler un ensemble d'éléments de modélisation communs et fondamentaux à l'approche Aspect. Nous sommes convaincus, de par la diversité des langages de programmation proposés que ces éléments communs sont nécessairement plus abstraits que ceux directement manipulés dans chacun de ces langages.

Nous adoptons une approche par métamodélisation dite *Bottom-Up* qui consiste, tout d'abord, à identifier et à définir pour chaque langage de programmation par aspects une sémantique de ses principaux concepts et relations (i.e. un métamodèle spécifique), à partir des termes (i.e. mots clés) spécifiques à ce langage et la syntaxe de celui-ci (cf. figure 4.2).

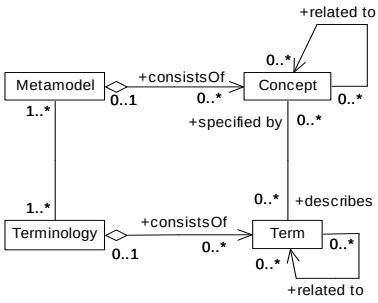


Figure 4.2. *Identification et définition de la sémantique des concepts spécifiques*

Une relation inter-concepts peut être une relation de généralisation/spécialisation, agrégation, composition, association, entrecroisement, etc. Ces types de relations sont eux-mêmes définis par des concepts au niveau du métamodèle spécifique à ce langage. Par ailleurs, parmi les relations inter-termes nous pouvons distinguer : restreint/étend, possède, synonyme, antonyme, associé, etc. Notre approche de métamodalisation consiste, ensuite, à déduire et à définir un ensemble de concepts communs plus abstraits que ceux spécifiques à chaque langage.

Les trois métamodèles ainsi spécifiés ; nous proposons ensuite pour la modélisation de structures par aspects des (patrons de) conceptions d'adopter l'idée de l'ingénierie dirigée par les modèles (IDM ou MDA [OMG-MDA 03]), qui préconise, pour augmenter la réutilisation, de définir des modèles indépendants destinés à être transformés pour obtenir des modèles spécifiques. La transformation de tels modèles se fait toujours dans le respect de règles de transformation, mais il existe cependant différentes approches de transformation de modèles. La deuxième étape de notre travail consiste donc à définir, de plus, un ensemble de règles de transformation de modèles permettant de passer d'un modèle instance du métamodèle général à un modèle instance du métamodèle spécifique à AspectJ ou Hyper/J (cf. figure 4.3). Elle consiste également à définir le processus de transformation adéquat à de telles projections[14].

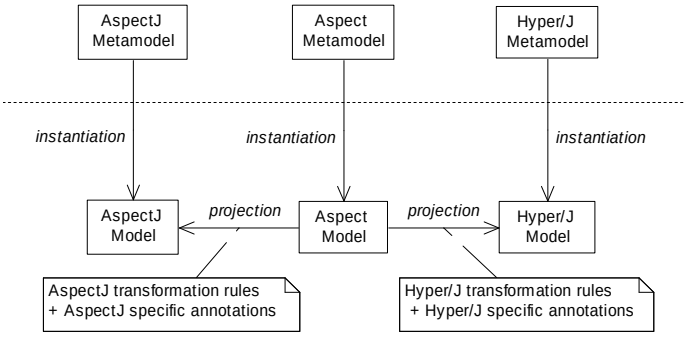


Figure 4.3. *Métamodélisation des aspects et transformation de modèles*

Nous proposons donc une approche par métamodélisation et transformation de modèles que nous basons sur un métamodèle général, défini comme une extension du métamodèle d'UML, intégrant des concepts Aspect plus généraux que ceux qui sont spécifiques à AspectJ et Hyper/J. De tels concepts doivent être assez abstraits pour permettre la modélisation de structures par aspects qui soient indépendantes d'un modèle ou d'un langage de programmation par aspects en particulier, mais aussi assez significatifs pour faciliter les projections de telles structures indépendantes vers des structures spécifiques.

2. Un métamodèle général pour la modélisation par aspects

Nous proposons dans cette section deux extensions par métamodélisation du métamodèle d'UML spécifiques respectivement à AspectJ et à Hyper/J. Nous proposons ensuite un métamodèle général que nous avons élaboré en rapprochant les concepts et relations communs aux deux métamodèles spécifiques. Nous débutons cette partie par un bref rappel des éléments du formalisme de spécification d'UML que nous utilisons également pour spécifier les nouveaux concepts des trois métamodèles que nous proposons.

[14] Nous rappelons brièvement dans la section 3.1 de ce chapitre les notions fondamentales de l'approche MDA, et précisons dans la section 3.2 le processus et l'approche de transformation de modèles que nous proposons.

2.1 Rappel des éléments du formalisme de spécification d'UML

Nous traitons principalement dans le cadre de notre proposition de la définition des éléments de modélisation structurelle qui sont nécessaires à la représentation de diagrammes de classes/aspects[15], plutôt qu'à ceux nécessaires à la modélisation dynamique des programmes. Pour la définition de l'ensemble des éléments que nous proposons, dans un souci de clarté et de rigueur, nous adoptons le formalisme utilisé par l'OMG dans la spécification d'UML. Nous en rappelons ci-dessous les trois principaux éléments [OMG-UML, 04] :

– *Syntaxe abstraite* (*Abstract syntax*). C'est un diagramme de classes UML décrivant l'ensemble des métaclasses qui définissent les éléments de modélisation du langage et leurs relations.

– *Règles de bonne formation* (*Well-formedness rules*). Elles décrivent la sémantique statique du métamodèle. Elles spécifient un ensemble de contraintes exprimées en OCL [OMG-OCL 03] : des informations supplémentaires importantes qui n'ont pas pu être spécifiées à l'aide d'éléments de modélisation définis par le métamodèle. Ces contraintes s'expriment dans le modèle mais s'appliquent à chacune des instances du modèle (*i.e.* à chacune des instances des métaclasses du métamodèle).

– *Sémantique* (*Semantics*). C'est une explicitation de la signification des éléments de modélisation du langage, en langage naturel.

Dans ce qui suit, nous présentons essentiellement les syntaxes abstraites des trois métamodèles que nous proposons. Nous donnons également quelques exemples[16] de règles de bonne formation complétant les spécifications de quelques concepts de ces métamodèles. Nous présentons par ailleurs pour chaque nouveau concept la syntaxe concrète permettant la représentation graphique de ce dernier.

2.2 AspectJ/UML : une extension d'UML pour AspectJ

Nous présentons dans cette sous-section la syntaxe abstraite et la notation graphique des concepts d'une extension du métamodèle d'UML spécifique à AspectJ (cf. § 3.6.2 du chapitre 2). En plus des concepts fondamentaux de l'Objet (i.e. Class, Interface, Attribut, Operation, Method, Association, Generalization, etc.), nous considérons donc ceux propres au langage AspectJ : Aspect, Introduction, Advice, Pointcut, etc. [Kiczales *et al.*, 01]. Chaque concept (ou élément de modélisation) du nouveau métamodèle possède une spécification qui contient la description unique de toutes ses caractéristiques. Nous utilisons les règles de bonne formation pour compléter une telle spécification, par des informations qui ne peuvent pas être exprimées de manière graphique sur le diagramme de classes de la syntaxe abstraite. Nous présentons de plus les nouveaux types primitifs, utilisés pour la modélisation de ces nouveaux concepts dans le métamodèle d'AspectJ/UML. Nous proposons enfin une notation graphique pour ces concepts.

2.2.1 Aspect

L'aspect est l'unité modulaire d'encapsulation d'une préoccupation transversale (*crosscutting concern*) en AspectJ. AspectJ définit un aspect d'une façon similaire aux classes en Java (cf. syntaxe textuelle d'un aspect dans l'annexe B) : un aspect peut contenir des définitions de propriétés structurelles et/ou comportementales qui lui sont propres, mais aussi des définitions de propriétés

[15] Les diagrammes de classes/aspects représentent la structure statique des programmes en termes de classes, aspects et relations entre ces éléments.

[16] Les spécifications complètes des trois métamodèles proposés, intégrant la syntaxe abstraite, les règles de bonne formation et la sémantique complète de l'ensemble des éléments de modélisation nouvellement introduits sont disponibles sur notre site web [Web Hachani] et dans l'annexe B.

d'entrecroisement (CrosscuttingFeature) comme les introductions (*introduction*s), les perfectionnements (*advice*s), etc. Un aspect peut contenir de plus, des spécifications de crosscutting (CrosscuttingSpecification) tels que les points de recouvrement (*pointcut*s, *named pointcut*s, etc.) et d'autres spécifications plus liées au fait qu'AspectJ soit une extension de Java (*custom compilation* et *softening exceptions*). Par ailleurs, un aspect peut hériter d'un autre aspect ou d'une classe, comme il peut implémenter une interface. Il peut être aussi abstrait et peut avoir différents types d'accès (*public*, etc.). De plus, un aspect peut participer à une ou plusieurs associations avec uniquement des classes, et peut donc être relié à différentes extrémités d'associations (cf. syntaxe abstraite de la métaclasse Association dans l'annexe A). Il s'agit, cependant, d'associations navigables uniquement dans un sens : d'un aspect vers une classe. Enfin, un aspect peut jouer le rôle d'une préoccupation de base qui peut être impactée par un ou plusieurs autres aspects. La figure 4.4 ci-dessous montre la syntaxe abstraite d'un Aspect.

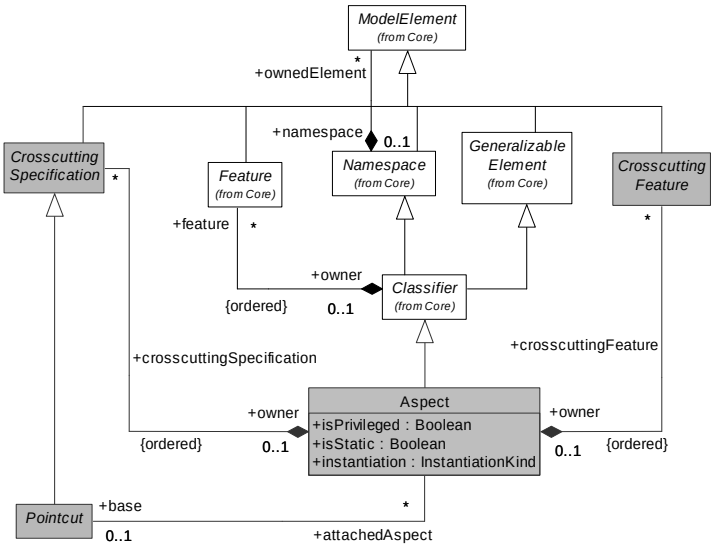


Figure 4.4. *AspectJ/UML — Syntaxe abstraite d'un aspect*

Remarquons qu'un aspect peut de plus être déclaré privilégié (*privileged*), ce qui lui donne accès aux propriétés privées ou protégées des classes avec lesquelles il interagit. Il peut être par ailleurs lié à un point de recouvrement restreignant son impact sur les classes de base de l'application. Un aspect possède également plusieurs autres propriétés caractéristiques et des associations que nous détaillons ci-dessous.

Attributs

isPrivileged	Indique si l'aspect est priviligié ou non. Il peut avoir les valeurs true ou false. La valeur par défaut est false.
isStatic	Spécifie si l'aspect est privé à une classe donnée. Dans un tel cas isStatic est mis à true, sinon il est mis par défaut à false.
instanciation	Spécifie le type d'instanciation de l'aspect. Les valeurs possibles sont : - singleton. C'est la valeur par défaut, elle indique que l'aspect considéré admet une instance unique ; - perThis/perTarget. Indique que l'aspect en question admet une instance pour chaque objet désigné par this ou par target ; - perCFlow/perCFlowBelow. Indique qu'une instance de l'aspect considéré est créée pour un flot d'exécution.

Associations

base	Spécifie, dans le cas où il ne s'agit pas d'un aspect singleton, l'ensemble des points de jonction pour lesquels l'aspect sera instancié.

crosscuttingFeature	Donne la liste ordonnée des propriétés d'entrecroisement de l'aspect.
crosscuttingSpecification	Donne l'ensemble des spécifications d'entrecroisement de l'aspect.
dominance	Spécifie l'ensemble des relations de précédence d'un aspect, dans lesquelles il participe en tant que dominant (cf. figure 4.8).
submission	Spécifie l'ensemble des relations de précédence d'un aspect, dans lesquelles il participe en tant que dominé (cf. figure 4.8).

Remarquons enfin que, définie comme étant un classificateur (*classifier*) et donc un élément généralisable, une classe peut hériter d'un autre classificateur, et donc d'un aspect par exemple (cf. syntaxe abstraite de GeneralizableElement dans l'annexe A). Toutefois, ceci doit être interdit car une classe ne peut en aucun cas hériter d'un aspect ou implémenter un aspect. De telles restrictions peuvent être spécifiées grâce à des règles de bonne formation en utilisant le langage OCL. Nous présentons ci-dessous quelques exemples de règles de bonne formation complétant la spécification de la métaclasse Aspect.

Règles de bonne formation

– Un aspect contenant au moins une déclaration d'un point de recouvrement abstrait est abstrait.

```
context Aspect inv :
    self.crosscuttingSpecification->exists (cS | cs.oclIsKindOf(NamedPointcut)
                                                and cS.isAbstract = #true)
    implies self.isAbstract = #true
```

– Une classe ne peut jamais hériter d'un aspect

```
context Generalization inv :
    self.child.oclIsKindOf(Class) implies not(self.parent.oclIsKindOf(Aspect))
```

2.2.2 Propriétés d'entrecroisement

Les propriétés d'entrecroisement (*crosscutting*) sont les propriétés structurelles et de comportements propres à une préoccupation transversale, qui s'entrecroisent avec une ou plusieurs classes d'une application. La figure 4.5 montre la syntaxe abstraite des différents types de ces propriétés.

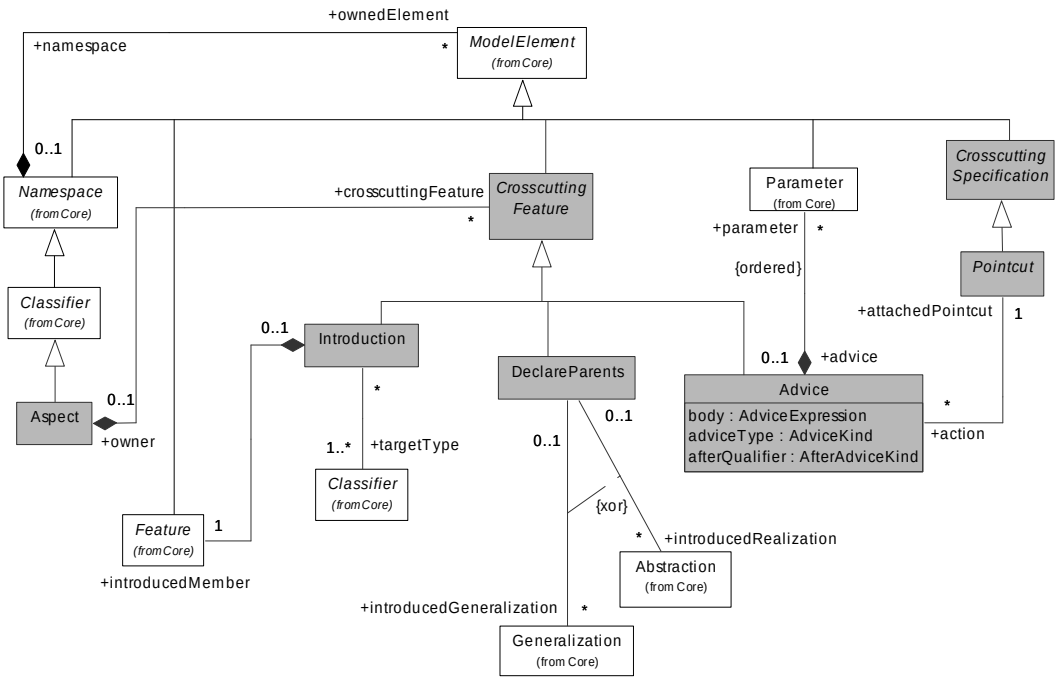


Figure 4.5. *AspectJ/UML — Syntaxe abstraite des propriétés d'entrecroisement*

Ces propriétés sont représentées notamment par les métaclasses Introduction, DeclareParents et Advice. Nous présentons chacune de ces classes dans ce qui suit.

Introduction (Introduction). Une introduction est une définition statique dans un aspect d'une nouvelle propriété (attribut, opération ou méthode, de classe ou d'instance) destinée à étendre la définition d'une classe, d'une interface ou d'un aspect. Elle met en relation d'une part la propriété à ajouter et d'autre part le ou les classificateurs impactés. Introduction est la métaclasse spécifiant les introductions. Nous décrivons ci-dessous les associations de cette métaclasse et quelques règles de bonne formation.

Associations

targetType	Spécifie les classes, interfaces, et/ou aspects dont les définitions sont destinées à être augmentées par la propriété à ajouter.
introducedMember	Spécifie la propriété à ajouter.

Règles de bonne formation

- Le type cible d'une introduction peut être uniquement, une classe, une interface ou un aspect.

```
context Introduction inv : self.targetType.oclIsKindOf(Class) or
                           self.targetType.oclIsKindOf(Interface) or
                           self.targetType.oclIsKindOf(Aspect)
```

- Une introduction peut ajouter des opérations abstraites uniquement dans des interfaces, des classes abstraites ou des aspects abstraits.

```
context Introduction inv : (self.introducedMember.oclIsTypeOf(Operation) and
                            self.introducedMember.isAbstract = #true)
    implies (self.targetType.oclIsKindOf(Interface) or
             self.targetType.oclAsType(Class) and (self.targetType.isAbstract = #true) or
             self.targetType.oclAsType(Aspect) and (self.targetType.isAbstract = #true))
```

Déclaration d'héritage ou de réalisation (DeclareParents). Une déclaration d'héritage ou de réalisation permet de modifier la hiérarchie d'une classe existante en intervenant sur les ancêtres directs de celle-ci, qu'ils soient des classes ou des interfaces. Elle met donc en jeu une relation de généralisation ou une relation de réalisation (un stéréotype particulier de la métaclasse Abstraction, cf. annexe A). Ces types de déclarations sont spécifiés au niveau du métamodèle par la métaclasse DeclareParents, dont les associations sont détaillées ci-dessous.

Associations

introducedGeneralization	Spécifie les relations de généralisation introduites par la déclaration d'héritage.
introducedRealization	Spécifie les relations de réalisation introduites par la déclaration de réalisation.

Perfectionnement (Advice). Un perfectionnement est un bout de code dont l'exécution est conditionnée par exactement une spécification d'entrecroisement de type point de recouvrement (nommé ou anonyme). Un perfectionnement peut être exécuté avant, après, ou à la place de la ou des propriétés comportementales désignées par le point de recouvrement qui lui est attaché. Par ailleurs, comme les opérations et méthodes en UML, un perfectionnement peut admettre une liste de paramètres en entrée et, le cas échéant, un paramètre résultat ; seuls les perfectionnements destinés à être exécutés à la place des propriétés comportementales peuvent avoir un paramètre en retour. Dans le métamodèle d'AspectJ/UML les perfectionnements sont spécifiés par la métaclasse concrète Advice, dont les attributs et les associations sont présentés ci-dessous. Nous donnons également un exemple de règle de bonne formation spécifique à cette métaclasse.

Attributs

body	Présente l'implémentation du perfectionnement.
adviceType	Spécifie le type du perfectionnement. Les valeurs possibles sont : - before indique que le code du perfectionnement doit être exécuté avant celui relatif à la propriété comportementale impactée ; - after indique que le code du perfectionnement doit s'exécuter après celui de la méthode impactée ; - around indique que le code du perfectionnement doit s'exécuter à la place de celui relatif à la méthode impactée.
afterQualifier	Spécifie dans le cas où il s'agit d'un perfectionnement de type after, si le code de celui-ci doit s'exécuter à chaque fois que la méthode impactée réussit son exécution, ou contrairement, quand l'exécution de celle-ci échoue. Les valeurs possibles pour cet attribut sont : - returning indique que le code du perfectionnement doit être exécuté obligatoirement après la fin de l'exécution de la méthode impactée ; - throwing indique que le code du perfectionnement ne doit s'exécuter que si la méthode impactée ne termine pas correctement son exécution ; - undefined est la valeur par défaut ; elle est obligatoire dans le cas où il s'agit d'un perfectionnement de type before ou around.

Associations

parameter	Donne la liste des paramètres du perfectionnement.
attachedPointcut	Spécifie le point de recouvrement qui conditionne l'exécution du perfectionnement.

Règle de bonne formation

– Les perfectionnements de type before ou after n'ont pas de paramètres en retour.

```
context Advice inv : (self.adviceType = #before or self.adviceType = #after)
                     implies (self.parameter->forAll(p | p.kind = #pdk_in))
```

2.2.3 Spécifications d'entrecroisement

Les spécifications d'entrecroisement (CrosscuttingSpecification) définissent, entre autres, les points de jonction auxquels les aspects peuvent dynamiquement interagir avec les classes pour modifier leurs comportements. Il s'agit des points de recouvrement atomiques et composites, qui peuvent être nommés ou anonymes. Nous distinguons par ailleurs, deux autres types de spécifications d'entrecroisement qui sont étroitement liées au fait qu'AspectJ est une extension de Java. Ces spécifications régissent d'une part la compilation et les exceptions d'autre part. La figure 4.6 montre la syntaxe abstraite des différentes métaclasses représentant l'ensemble des spécifications d'entrecroisement possibles en AspectJ.

Erreurs et avertissements de compilation (CustomCompilation). Un aspect peut détenir des déclarations d'erreurs et d'avertissements de compilation qui permettent de spécifier le message à renvoyer respectivement, en cas d'erreur ou d'avertissement au moment de la compilation d'un programme AspectJ. Chaque déclaration est définie par un type (type : error ou warning) et un message (message). Elle est par ailleurs attachée à exactement un point de recouvrement (/attachedPointcut) qui peut être atomique ou composite, et nommé ou anonyme. Elle ne peut cependant être attachée à un point de recouvrement qui expose le contexte d'exécution, et qui n'est pas statiquement déterminable. Dans le métamodèle AspectJ/UML, de telles déclarations sont spécifiées par la métaclasse CustomCompilation, dont une règle de bonne formation est présentée ci-dessous.

Règle de bonne formation. Tout point de recouvrement attaché à une CustomCompilation doit être statiquement déterminable.

```
context CustomCompilation inv : self.attachedPointcut.isStaticallyDeterminable() = #true
```

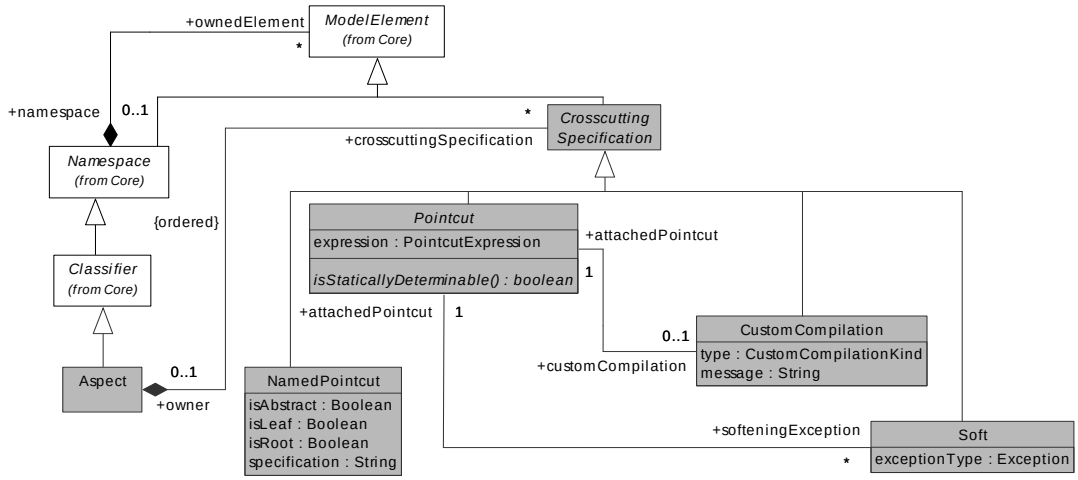


Figure 4.6. *AspectJ/UML — Syntaxe abstraite des spécifications d'entrecroisement*

Traitement d'exceptions (Soft). En AspectJ, un aspect peut détenir des déclarations de « softening » qui permettent de traiter une exception comme une RuntimeException Java. Une telle déclaration permet en effet, de s'affranchir de l'obligation que l'on a en Java d'encadrer tout code spécifique de levée d'exception dans un bloc try-catch ou de spécifier l'éventualité de sa levée dans une clause throws. Comme les déclarations d'erreur ou d'avertissement, toute déclaration de « softening » doit être attachée à exactement un point de recouvrement statiquement déterminable (/attachedPointcut). Ce type de déclaration est représenté au niveau du métamodèle par la métaclasse Soft, qui possède un attribut nommé exceptionType spécifiant le type de l'exception à traiter par la déclaration. Voici ci-dessous un exemple de règle de bonne formation complétant la spécification de cette nouvelle métaclasse.

Règle de bonne formation. Une déclaration de traitement d'exception n'a pas de nom.

```
context soft inv : self.name = ""
```

Points de recouvrement. AspectJ définit notamment deux types de points de recouvrement : les points de recouvrement anonymes et nommés qui peuvent être atomiques ou composites. De manière générale, un point de recouvrement est un prédicat construit sur la base d'un ou de plusieurs points de jonction, conditionnant une interaction dynamique entre un aspect et des classes. La figure 4.7 montre la syntaxe abstraite de l'ensemble des métaclasses du métamodèle représentant les différents types de points de recouvrement définis par AspectJ.

- ***Points de recouvrement anonymes*** (Pointcut). Les points de recouvrement anonymes possibles en AspectJ sont représentés de manière générale par la métaclesse abstraite Pointcut. Cette classe est définie comme étant un élément qui peut être atomique (PointcutDesignator) ou composite (ComposedPointcut) (dans les termes du patron *Composite* (cf. tableau 1.6 du chapitre 1)). Nous distinguons différents types de points de recouvrement atomiques : TypeBasedDesignator, MethodRelatedDesignator et ControlFlowDesignator.
- ***Points de recouvrement nommés*** (NamedPointcut). Par rapport à un point de recouvrement anonyme, un point de recouvrement nommé détient une spécification (i.e. une signature, par analogie aux signatures des opérations). Il est identifié par un nom et éventuellement une liste de paramètres en plus de la déclaration de ses points de jonction, définis par un point de recouvrement anonyme (Pointcut de manière générale) Il peut être abstrait et dans ce cas n'être lié à aucun point de recouvrement anonyme. Dans le métamodèle, les points de recouvrement nommés sont représentés par la métaclasse NamedPointcut.

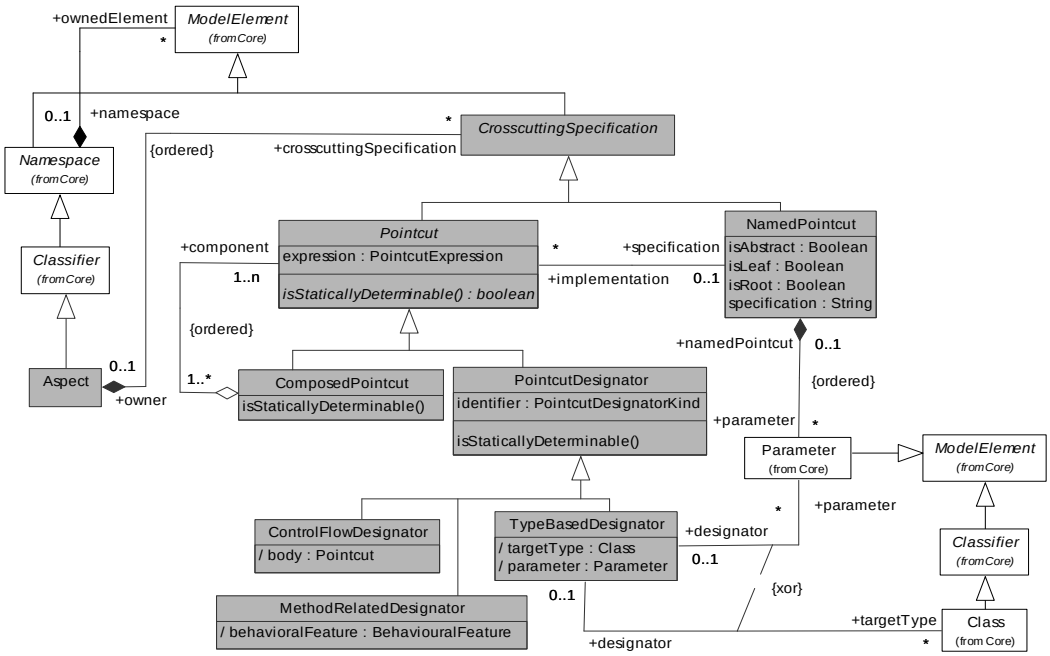


Figure 4.7. *AspectJ/UML — Syntaxe abstraite des points de recouvrement*

Nous détaillons ci-dessous chacune des métaclasses considérées tout en présentant ses attributs, ses associations et quelques-unes de ses règles de bonne formation.

La métaclasse NamedPointcut

Attributs (cf. annexe B pour l'ensemble des attributs)

specification	Représente la signature du point de recouvrement nommé.

Associations

implementation	Donne l'ensemble des points de jonction du point de recouvrement nommé.
parameter	Donne la liste des paramètres du point de recouvrement.

La métaclasse Pointcut

Attribut

expression	Représente l'expression du point du recouvrement.

Associations

specification	Donne la spécification du point de recouvrement quand il est nommé.
action	Donne les perfectionnements attachés au point de recouvrement.
attachedAspect	Désigne l'ensemble des aspects (s'ils existent) qui sont basés sur le point de recouvrement en question.
customCompilation	Spécifie la déclaration d'erreur ou d'avertissement attachée au point de recouvrement considéré, si elle existe.
softeningException	Spécifie l'ensemble des déclarations de « softening » attachées au point de recouvrement en question, si elles existent.

Règles de bonne formation

– Un point de recouvrement défini comme étant la base d'un aspect doit être anonyme.

```
context Aspect inv : self.attachedPointcut.specification->isEmpty
```

– *Opération additionnelle — isStaticallyDeterminable : Boolean* ;

La métaclasse ComposedPointcut

Un point de recouvrement composé est un prédicat construit sur la base de points de recouvrement atomiques et/ou composés qu'ils soient nommés ou anonymes.

Association

component Donne la liste ordonnée des composants du point de recouvrement.

Règles de bonne formation

- Tous les composants d'un point de recouvrement composé statiquement déterminable doivent être aussi statiquement déterminables.

```
context ComposedPointcut inv : self.isStaticallyDeterminable() = #true
    implies (self.component->forAll(c : Pointcut | c.isStaticallyDeterminable() = #true))
```

- *Opération additionnelle*

```
isStaticallyDeterminable : Boolean ;
isStaticallyDeterminable = self.component->forAll(c : Pointcut |
                                                    c.isStaticalyDeterminable() = #true)
```

La métaclasse PointcutDesignator

Un désignateur de point de recouvrement est un point de recouvrement atomique et anonyme. Il possède un identificateur indiquant son type et une expression déclarant l'ensemble des points de jonction auxquels les aspects peuvent dynamiquement interagir avec les classes de l'application. Un désignateur peut être lié à une ou plusieurs méthodes définies par leurs signatures, une ou plusieurs classes définies par leurs types, ou plus généralement à un autre point de recouvrement, d'où les spécialisations de cette classe en TypeBasedDesignator, MethodBasedDesignator et ControlFlowDesignator.

Attribut

identifier Spécifie l'identificateur du désignateur du point de recouvrement. Les valeurs possibles de cet attribut sont : call, execution, handler, adviceexecution, initialization, preinitialization, staticinitialization, get, set, withincode, this, target, args, cflow, cflowbelow, within, withincode et if.

Règle de bonne formation

Opération additionnelle — isStaticallyDeterminable : Boolean ;

```
isStaticallyDeterminable = (self.identifier <> #cflow and self.identifier <> #cflowbelow
                            and self.identifier <> #if and self.identifier <> #this
                            and self.identifier <> #target and self.identifier <> #args)
```

La métaclasse MethodRelatedDesignator

Association

behaviouralFeatures Donne la liste des opérations/méthodes mises en jeu par le désignateur.

Règle de bonne formation

Les valeurs possibles de l'identificateur d'un point de recouvrement de type MethodRelatedDesignator sont : call, get, set, execution, initialization, preinitialization, within et withincode.

```
context MethodRelatedDesignator inv :
    self.identifier = #call or self.identifier = #get or self.identifier = #set
                      or self.identifier = #execution or self.identifier = #initialization
                      or self.identifier = #preinitialization or self.identifier <> #within
                      or self.identifier = #withincode)
```

La métaclasse ControlFlowDesignator

Association

body	Spécifie le point de recouvrement sur lequel est défini le désignateur.

Règle de bonne formation

Les valeurs possibles de l'identificateur d'un point de recouvrement de type ControlFlowDesignator sont : cflow et cflowbelow.

```
context ControlFlowDesignator inv :
    self.identifier = #cflow or self.identifier = #cflowbelow)
```

La métaclasse TypeBasedDesignator

Les points de recouvrement de type TypeBasedDesignator sont, de manière générale, liés à des classes. De tels désignateurs peuvent toutefois faire référence à un ou plusieurs paramètres : uniquement, this, target et args peuvent le faire. Les valeurs possibles de l'attribut identifier spécifique à cette métaclasse sont : handler, staticinitialization, this, target et args.

Associations

targetType	Spécifie la liste des classes mises en jeu par le désignateur.
parameter	Donne la liste des paramètres du désignateur.

Règles de bonne formation

- Les désignateurs d'identificateurs this et target ne peuvent pas faire référence à plusieurs paramètres à la fois. Uniquement le désignateur args peut le faire.

```
context TypeBasedDesignator inv :
        (self.identifier = #this or self.identifier = #target)
        implies (self.parameter->size = 1)
```

- Seuls les désignateurs this, target et args peuvent avoir des paramètres, mais pas handler et staticinitialisation.

```
context TypeBasedDesignator inv :
        (self.identifier = #handler or self.identifier = #staticinitialization)
        implies (self.parameter->isEmpty)
```

2.2.4 Relations entre aspects et classes

En plus des relations d'héritage, d'association, etc. spécifiques à l'Objet, un aspect peut avoir des relations de précédence avec d'autres aspects et des relations d'entrecroisement avec des classes, des interfaces ou encore des aspects. La figure 4.8 montre la syntaxe abstraite de ces deux nouvelles relations.

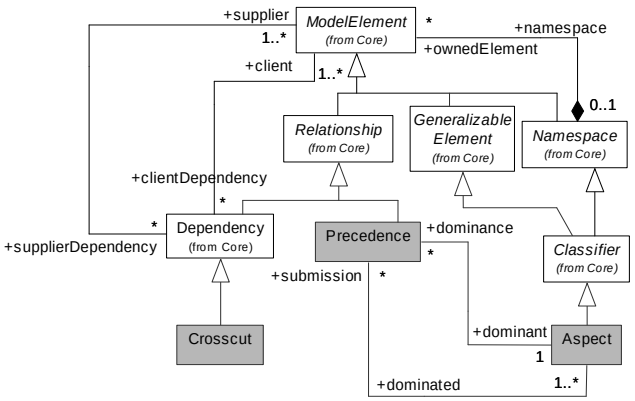


Figure 4.8. *AspectJ/UML — Syntaxe abstraite des relations d'entrecroisement et de précédence*

Relations d'entrecroisement (Crosscut). Une relation d'entrecroisement est une relation entre un élément d'entrecroisement et un (ou plusieurs) élément(s) cible(s) qui n'est pas de nature structurelle. Elle indique qu'un élément d'entrecroisement source requiert la présence d'un autre élément (ou d'autres éléments) cible(s) pour son bon fonctionnement ou son implémentation. Par exemple, une Introduction d'une propriété dans un classificateur donné nécessite la présence de ce dernier. De même, un Pointcut nécessite la présence de l'opération (ou des opérations) qu'il entrecroise. Ceci étant, nous avons choisi de faire également de Crosscut une spécialisation de Dependency (cf. syntaxe abstraite de la métaclasse Dependency dans l'annexe A), avec une restriction du nombre de clients mis en jeu (/client) qui doit être égal à un (cf. contraintes OCL Annexe B). Un aspect peut entrecroiser un ou plusieurs classificateurs par le biais d'une introduction, ou d'un point de recouvrement, ou encore d'une déclaration d'héritage. Des stéréotypes particuliers de Crosscut précisant la nature d'entrecroisement sont ainsi définis. Remarquons que les perfectionnements ainsi que les propriétés destinées à être ajoutées par le biais des introductions peuvent requérir, éventuellement, d'autres opérations de différentes classes. De telles relations de dépendance sont déjà définies par UML, plus précisément par la relation d'utilisation (Usage, cf. Annexe A). Nous n'ajoutons pas, de ce fait, de nouveaux stéréotypes pour ce type de relations. Les seuls stéréotypes que nous proposons sont donc les suivants :

Stéréotypes

addFeature	Représente une relation de dépendance entre une introduction et un ou plusieurs classificateurs. Il s'agit d'une relation de Crosscut stéréotypée.
addParent	Représente une relation de dépendance entre une déclaration d'héritage et deux ou plusieurs classificateurs. Il s'agit d'une relation de Crosscut stéréotypée.
intercept	Représente une relation de dépendance entre un point de recouvrement de type MethodBasedDesignator ou TypeBasedDesignator, et une ou plusieurs opérations/méthodes de classes. Il s'agit d'une relation de Crosscut stéréotypée.

Remarquons que, selon la nature d'entrecroisement (i.e. le stéréotype particulier), les nombres des éléments mis en relation ainsi que leurs types sont différents. Par exemple, une relation de Crosscut stéréotypée par « addFeature » est définie exclusivement entre une introduction et un classificateur. De telles restrictions sont spécifiées par des contraintes OCL, comme par exemple :

Règles de bonne formation

– addFeature met en relation exactement une introduction et un ou plusieurs classificateurs.

```
context addFeature inv : (self.client.oclIsTypeOf(Introduction)) &&
                         (self.supplier->size = 1) && (self.supplier.oclIsKindOf(Classifier))
```

Relations de précédence (Precedence). La relation de précédence est définie entre deux ou plusieurs aspects. Elle régit l'exécution des perfectionnements appartenant à ces différents aspects et attachés à un même point de recouvrement. Un aspect peut participer à plusieurs relations de précédence comme étant dominant, comme il peut participer à plusieurs relations de précédence et jouer le rôle d'un aspect dominé.

Associations

dominant	Désigne l'aspect dominant.
dominated	Spécifie l'ensemble des aspects dominés.

2.2.5 Types primitifs

Les types suivants sont des types primitifs du métamodèle AspectJ/UML, utilisés pour la modélisation des concepts nouvellement introduits. Il s'agit d'un certain nombre de nouvelles métaclasses étendant le paquetage *Data Types* d'UML. La figure 4.9 suivante présente ces types de données.

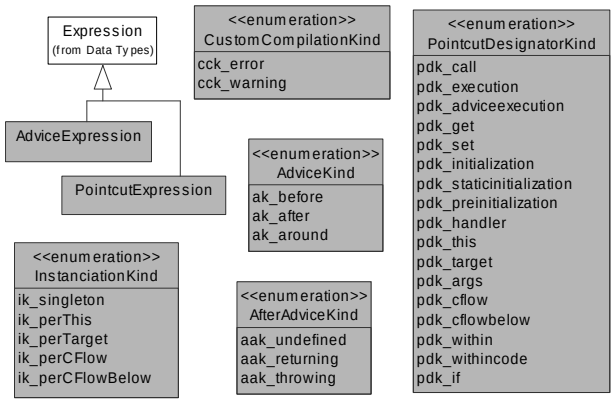


Figure 4.9. *AspectJ/UML — Types de données*

2.2.6 Notation graphique des concepts et relations d'AspectJ/UML

Nous proposons dans ce qui suit une syntaxe concrète permettant de visualiser les concepts et relations spécifiques à AspectJ/UML.

Les aspects et relations de précédence. Comme les classes, les aspects sont représentés par des rectangles compartimentés (cf. figure 4.10-(a)). Le premier compartiment symbolise l'aspect, il contient son nom et le stéréotype « aspect » ou « privileged aspect ». Conformément à la notation UML, un nom en italique précise que l'aspect est abstrait, et deux compartiments contiennent respectivement les attributs et opérations propres à l'aspect. Deux autres compartiments supplémentaires sont ajoutés. Le premier contient les introductions de l'aspect, alors que le deuxième est dédié à ses points de recouvrement et leurs perfectionnements associés. Les compartiments d'un aspect peuvent être éventuellement omis lorsque leur contenu n'est pas pertinent dans le contexte d'un diagramme particulier, on parle dans ce cas de représentation graphique simplifiée (cf. figure 4.10-(b)). Remarquons que les relations de précédence sont représentées par des flèches à pointe triangulaire pleine avec le mot-clé « dominates ».

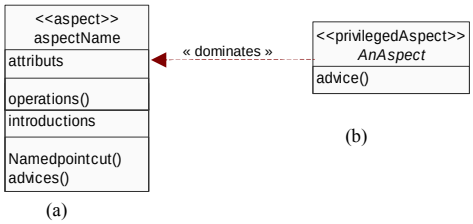


Figure 4.10. *AspectJ/UML — Différentes représentations d'un aspect*

Les introductions et relations d'entrecroisement. Les introductions sont regroupées par cible d'impact ; elles sont représentées au moyen de boîtes à coins carrés (à la manière de classes non nommées) portant le stéréotype « introduction ». Chaque introduction est liée à sa cible d'impact à l'aide de flèches en traits pointillés qui pointent vers les classifieurs destinations, et à l'aide du mot clé « addFeature » (cf. figure 4.11-(a)) représentant une relation de Crosscut stéréotypée. Les compartiments des introductions contiennent respectivement les attributs et les opérations destinés à être ajoutés à la cible d'impact. Un aspect peut aussi introduire de nouvelles propriétés dans une ou plusieurs classes cibles à travers une classe (ou une interface) conteneur. Un tel conteneur est déclaré, en général, comme étant interne et protégé ou privé à l'aspect (cf. figure 4.11-(b)). Les propriétés destinées à être ajoutées aux classes cibles sont introduites dans le conteneur qui doit être hérité ou réalisé par ces classes. Remarquons qu'une déclaration d'héritage ou de réalisation est représentée à l'aide d'une relation d'entrecroisement stéréotypée par « addParent », dont l'extrémité cible pointe la relation ajoutée (cf. figure 4.11-(b)).

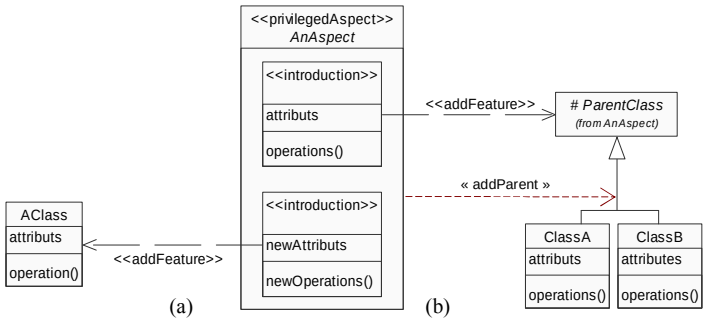


Figure 4.11. *AspectJ/UML — Représentation des introductions et des déclarations d'héritage ou de réalisation*

Les points de recouvrement et perfectionnements. Les perfectionnements sont regroupés par point de recouvrement dans un rectangle compartimenté (cf. figure 4.12), qui peut être lié à une ou plusieurs opérations ou classes impactées grâce à un ou plusieurs liens d'entrecroisement stéréotypés « intercept », à moins que le point de recouvrement considéré soit abstrait ou qu'il s'agisse d'un point de recouvrement primitif de désignateur adviceexecution. Le premier compartiment précise le point de recouvrement, alors que le deuxième est dédié aux perfectionnements. La syntaxe générale retenue pour la description des perfectionnements est la suivante.

advice_ type [after_qualifier] "(" arguments ")" [':' return_type]
avec advice_type ::= before | after | around
after_qualifier ::= returning | throwing
arguments ::= argument_name ':' argument_ type [',' arguments]

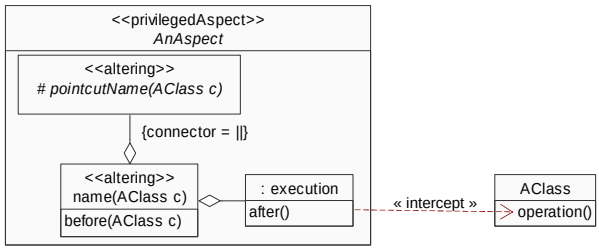


Figure 4.12. *AspectJ/UML — Représentation des points de recouvrement et des perfectionnements*

L'utilisateur organise à sa guise les différents perfectionnements. Il peut les regrouper par type par exemple, grâce à des stéréotypes : « before », « after » et « around ». Il faut faire attention, en revanche, à l'ordre de représentation des perfectionnements concurrents (i.e. de même type), qui indique l'ordre de précédence entre ceux-ci. Cet ordre peut également être rendu par l'utilisation d'un indice (i.e. un entier) ordonnant les différents perfectionnements de type identique.

Tout compartiment dédié à la représentation d'un point de recouvrement contient le stéréotype « altering », ainsi que la description de ce point. Un point de recouvrement nommé et abstrait est représenté par une spécification en italique, précisant obligatoirement son nom et éventuellement une liste de paramètres ainsi que sa visibilité. La syntaxe retenue pour la représentation des points de recouvrement primitifs, nommés ou anonymes, est donnée ci-dessous.

[visibility] [pointcut_name] ["(" arguments ")"] : pointcut_type
avec visibility::= '+' | '-'
arguments::= argument_name ':' argument_type [',' arguments]
pointcut_type::= call | execution | get | set | initialization | perinitialization
| handler | staticinitialization | adviceexecution

Rappelons que tout point de recouvrement primitif (de type call, execution, get, set, handler, (static)initialization, perinitialization ou encore adviceexecution) peut éventuellement préciser des conditions qui restreignent le contexte d'exécution de ses perfectionnements associés, en utilisant un ensemble de désignateurs : this, target, args, cflow(below), within(code). Ces désignateurs sont représentés dans le même compartiment grâce à des valeurs marquées (*tagged values*). Il est possible également de restreindre le contexte d'exécution d'un perfectionnement avec le désignateur if, qui est représenté en revanche par un commentaire attaché au point de recouvrement considéré.

Les points de recouvrement composés sont représentés à la manière des objets composites en UML, ils sont liés à leurs points de recouvrement composants grâce à des liens d'agrégation. De tels liens peuvent éventuellement être augmentés par des valeurs marquées spécifiant les connecteurs logiques (&& ou ||). De plus, ils sont indirectement liés à leurs opérations et classes impactées, à travers leurs points de recouvrement composants. La description d'un point de recouvrement composite est de la forme [visibility] [pointcut_name] ["(" arguments ")"]. Lorsqu'il s'agit d'un point de recouvrement composite anonyme, le compartiment correspondant ne contient aucune spécification ; il contient uniquement le mot clé « altering ». Un point de recouvrement composé peut éventuellement restreindre le contexte d'exécution de ses perfectionnements par des conditions supplémentaires ; celles-ci sont représentées comme dans le cas d'un point de recouvrement primitif.

2.3 HyperJ/UML : une extension d'UML pour Hyper/J

Hyper/J [Ossher *et al.*, 00] est une extension du langage Java pour l'approche Hyperspace (cf. § 3.5.1 du chapitre 2), ou plus généralement la séparation multidimensionnelle des préoccupations. Dans cette section nous présentons la syntaxe abstraite et la notation graphique des éléments de modélisation (ou concepts fondamentaux) d'HyperJ/UML, une extension du métamodèle d'UML prenant en compte les spécificités du langage Hyper/J : *hyperslice*, *integration rules*, etc. (cf. § 3.5.2 du chapitre 2). Pour chaque élément nous détaillons l'ensemble de ses caractéristiques et associations. Dans la sous-section 2.3.5 nous présentons l'ensemble des nouvelles métaclasses de données étendant le paquetage Data Types d'UML.

2.3.1 Unités intégrables

Hyper/J propose de décomposer les programmes en unités intégrables pouvant participer dans une ou plusieurs relations d'intégration. Il s'agit des unités structurantes propres aux préoccupations de base du programme et celles liées aux préoccupations transversales, devant être intégrées pour concevoir le programme final. Nous rappelons que l'approche Hyperspace est une approche symétrique, et que les préoccupations de base et transversales sont traitées de la même façon : on parle de préoccupations en général. Hyper/J distingue ainsi deux types d'unités intégrables : les unités primitives (i.e. *Attribut*, *Operation* et *Method*) et les unités composées qui sont les unités modulaires d'encapsulation des préoccupations (i.e. *Class*, *Interface* et *Hyperslice*). La figure 4.13 montre la syntaxe abstraite de ces éléments de modélisation. Dans le métamodèle d'UML, une métaclasse abstraite GeneralizableElement dénote l'ensemble des éléments pouvant participer à une relation de généralisation. De la même façon, une nouvelle métaclasse abstraite IntegrableUnit est ajoutée pour démarquer les éléments pouvant participer dans une relation d'intégration. Nous explicitons ci-dessous l'ensemble de ses associations.

Associations

adressedHyperslice	Donne la liste des préoccupations mettant en jeu une unité intégrable donnée.
integrationRule	Donne l'ensemble des relations d'intégration spécifiques appliquées à une unité intégrable donnée.

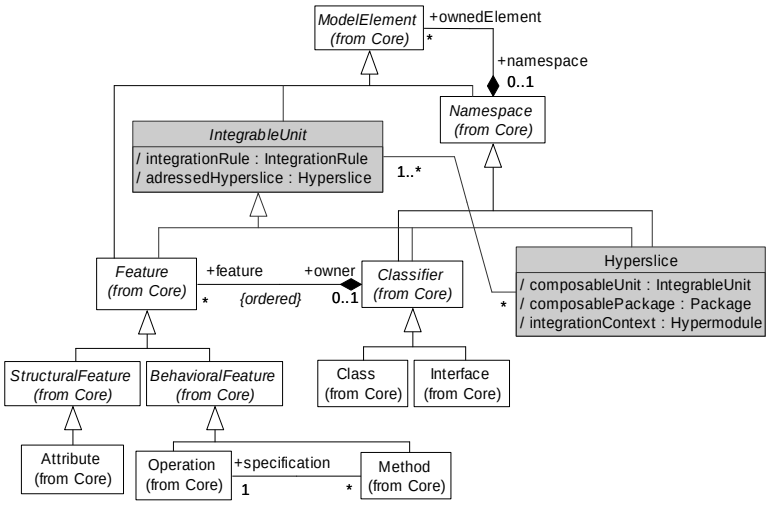


Figure 4.13. *HyperJ/UML — Syntaxe abstraite des unités intégrables*

2.3.2 Identification et décomposition des préoccupations

Hyper/J définit une matrice de préoccupations : Hyperspace. Il s'agit d'un espace structuré organisant l'ensemble des unités intégrables dans une matrice multidimensionnelle. Chaque axe représente une dimension de préoccupations (Dimension) et chaque point sur un axe, une préoccupation (Hyperslice) dans cette dimension. La figure 4.14 montre la syntaxe abstraite de ces éléments. Remarquons que chaque hyperspace est associé à deux ou plusieurs classificateurs (indiqués par son association /composableClassifier) représentant l'ensemble des unités intégrables propres aux différentes préoccupations définies dans le contexte de ce dernier. Chaque Hyperspace est aussi associé à plusieurs dimensions de préoccupations (/dimension). Chaque dimension peut regrouper plusieurs préoccupations (déterminées par son association /concern) qui sont encapsulées par différents hyperslices et associées à, au plus, un hyperspace (l'association /owner, côté Hyperspace).

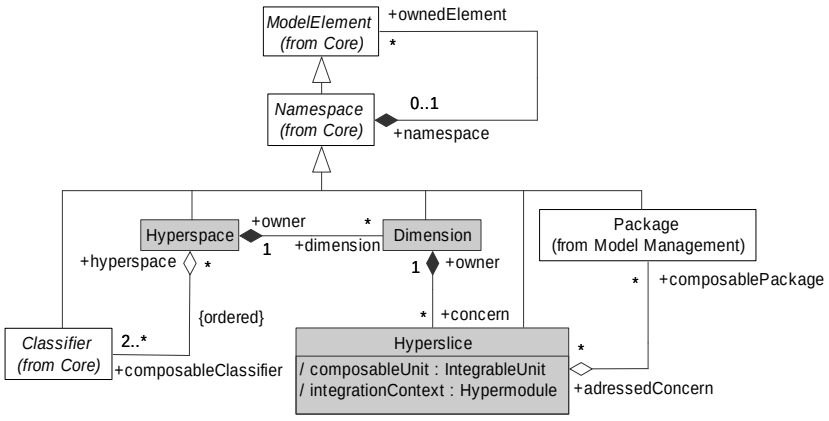


Figure 4.14. *HyperJ/UML — Syntaxe abstraite d'une matrice de préoccupations*

Remarquons par ailleurs que chaque hyperslice est associé à au plus une dimension (/owner), il s'agit d'un ensemble d'unités intégrables (identifiées par l'association /composableUnit) définies le plus souvent par des classificateurs et/ou des paquetages (indiqués par l'association /composablePackage) propres à une préoccupation donnée. En effet, un hyperslice peut être associé à différentes unités intégrables de type Feature, Class ou Interface, mais pas Hyperslice, d'où la règle de bonne formation suivante :

```
context Hyperslice inv : (self.composableUnit->forAll (cu: IntegrableUnit | cu.oclIsKindOf (Class)
                          or cu.oclIsKindOf (Interface) or cu.oclIsKindOf (Feature))
```

2.3.3 Composition des préoccupations

Comme il permet de séparer les différentes préoccupations de base et transversales, Hyper/J permet également la composition de ces préoccupations en termes d'hyperslices dans des contextes d'intégration particuliers : les hypermodules. La figure 4.15 montre la syntaxe abstraite d'un Hypermodule.

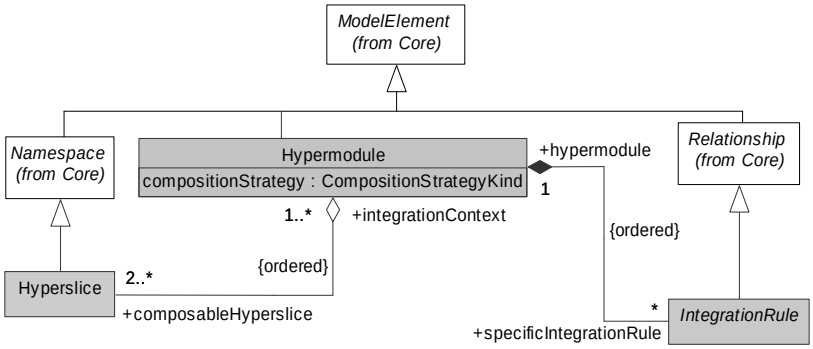


Figure 4.15. *HyperJ/UML — Syntaxe abstraite des hypermodules*

Chaque hypermodule déclare dans un ordre significatif l'ensemble des hyperslices à intégrer (indiqués par l'association /composableHyperslice). Il spécifie en plus la règle d'intégration générale (compositionStrategy), une relation de composition sémantique entre les unités intégrables propres aux différents hyperslices. Nous distinguons trois règles d'intégration générale : nonCorrespondingMerge, mergeByName et overrideByName, dont les sémantiques exactes sont complétées par l'ordre de déclaration des hyperslices mis en jeu. Un tel ordre spécifie en effet les priorités entre les unités des hyperslices à intégrer. Par ailleurs, un hypermodule peut éventuellement contenir des déclarations de règles d'intégrations spécifiques (indiquées par l'association /specificIntegrationRule) s'opposant le plus souvent à la règle d'intégration générale, et qui s'appliquent le plus souvent sur des unités intégrables primitives. Nous expliquons ci-dessous les différentes stratégies de composition générale.

compositionStrategy — Spécifie la règle de composition générale. Les valeurs possibles sont les suivantes :

- mergeByName indique que les unités intégrables de noms identiques sont destinées à être composées par fusion, par respect de l'ordre de déclaration de leurs hyperslices. Les unités du premier hyperslice précèdent celles du deuxième, et ainsi de suite.
- nonCorrespondingMerge indique que les unités de noms identiques ne sont pas destinées à être composées.
- overrideByName indique que les unités de noms identiques sont destinées à être composées par remplacement, selon l'ordre de déclaration de leurs hyperslices dans l'hypermodule. Les unités du premier hyperslice remplacent celles du deuxième, alors que les unités du deuxième remplacent celles du troisième...

Remarquons que chaque hyperslice peut être associé à plus d'un hypermodule (déterminés par son association /integrationContext) à la fois, et peut donc être composé avec plusieurs autres hyperslices dans d'autres contextes d'intégration.

2.3.4 Règles d'intégration spécifiques

En plus de la spécification de la règle de composition générale (compositionStrategy), Hyper/J permet la définition d'un ensemble de règles d'intégration spécifiques représentées par la métaclasse abstraite IntegrationRule (cf. figure 4.16), une spécialisation de la métaclasse Relationship. De telles règles consistent en un ensemble d'exceptions et/ou de spécialisations de la règle de composition générale dans les cas où celle-ci ne peut être appliquée.

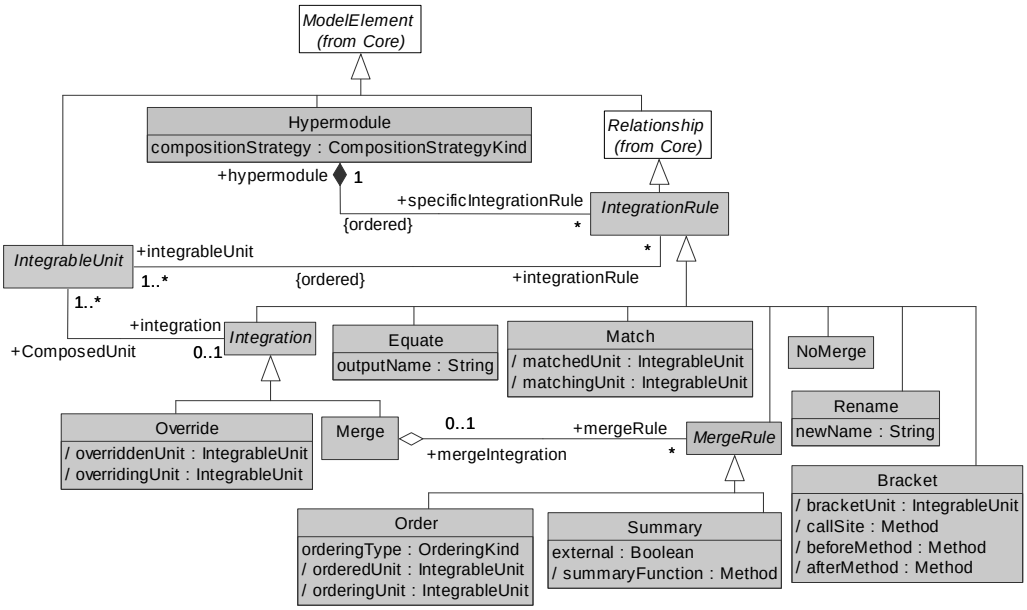


Figure 4.16. *HyperJ/UML — Syntaxe abstraite des règles d'intégration spécifiques*

D'une manière générale, les règles d'intégration spécifiques mettent en jeu une ou plusieurs unités intégrables (/integrableUnit) qui peuvent être de type Operation, Method, Attribute, Class, Interface et/ou Hyperslice, selon la règle considérée[17]. Par ailleurs, chaque règle d'intégration spécifique est associée à exactement un hypermodule (/hypermodule). Nous distinguons principalement les règles d'intégration spécifiques suivantes (cf. § 3.5.2, chapitre 2).

- Override et Merge. Deux spécialisations de la métaclasse abstraite Integration, mettant en jeu deux ou plusieurs unités intégrables en entrée et une ou plusieurs unités intégrables primitives ou composites en sortie (/composedUnit). Dans le cas de la règle Override nous distinguons deux types d'unités intégrables (en entrée) indiquées par les associations /overridenUnit et /overridingUnit, spécifiant respectivement les unités qui doivent être remplacées et leurs unités remplaçantes.
- Order et Summary. Deux spécialisations de la métaclasse abstraite MergeRule, qui peuvent éventuellement s'opposer à la règle d'intégration Merge spécifiant plus précisément les règles d'intégration. Ces deux règles peuvent être directement utilisées dans le contexte d'une règle de composition générale de type mergeByName, sans qu'elles soient obligatoirement associées à une règle d'intégration spécifique de type Merge. Chaque règle de type Order doit spécifier le type d'ordonnancement (before ou after) à travers son attribut orderingType. Elle spécifie de plus deux sous-ensembles ordonnés d'unités intégrables, qui sont déterminés par les associations /orderedUnit et /orderingUnit. Ces deux sous-ensembles indiquent respectivement, les unités devant être précédées et/ou suivies et les unités qui précèdent et/ou suivent. Par ailleurs, toute règle de type Summary est associée à exactement une propriété comportementale de type Method (à travers l'association /summaryFunction), déterminant le résultat à retourner. Une telle méthode peut être externe à l'ensemble des unités intégrables de l'hypermodule, en un tel cas l'attribut external a pour valeur true.

[17] Voir Annexe B pour les règles de bonne formation propres à chacune des métaclasses correpondant à l'ensemble des règles d'intégration spécifiques.

- Match et Equate. Elles sont utilisées pour indiquer que certaines unités intégrables sont équivalentes et correspondent les unes aux autres. Dans la règle Match nous distinguons deux sous-ensembles déterminés par les associations /matchedUnit et /matchingUnit, qui spécifient respectivement les unités qui doivent correspondre à d'autres et leurs unités correspondantes.
- NoMerge. Par opposition aux relations Override et Merge, cette relation indique que deux ou plusieurs unités intégrables qui correspondent par leurs noms n'ont pas à être intégrées (exception à l'intégration qu'elle soit de type Override ou Merge).
- Rename. Utilisée pour attribuer un nouveau nom à une unité intégrable résultat d'une intégration donnée. Le nouveau nom à attribuer est spécifié par l'attribut newName.
- Bracket. Indique si l'exécution d'un ensemble de méthodes doit être précédée et/ou suivie par l'exécution d'autres méthodes, spécifiées respectivement par les associations /beforeMethod et /afterMethod. Elle peut également indiquer le ou les sites d'appels (à travers l'association /callSite) de l'ensemble des méthodes mises en jeu, qui peuvent être de type Hyperslice, Class ou BehaviouralFeature.

2.3.5 Nouveaux types de données

La figure 4.17 donne l'ensemble des métaclasses ajoutées au paquetage *Data Types* d'UML, spécifiant les différents types de données nouvellement introduits.

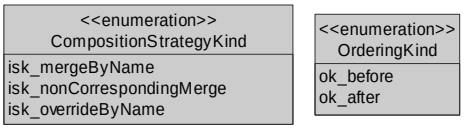


Figure 4.17. *HyperJ/UML — Types de données*

2.3.6 Notation graphique des concepts et relations d'HyperJ/UML

Nous proposons dans ce qui suit une syntaxe concrète pour les concepts et relations spécifiques à HyperJ/UML. Nous nous concentrons principalement sur la représentation des hyperslices et leurs dimensions englobantes, et sur les notations des hypermodules et des règles d'intégration.

Les hyperslices. Les hyperslices peuvent contenir des propriétés, des classificateurs, des associations, des généralisations, des paquetages, etc. Il est à noter qu'il n'y pas de relation de possession entre les éléments contenus dans un hyperslice et celui-ci, il s'agit uniquement d'un conteneur fictif qui définit une préoccupation donnée. Nous proposons de représenter chaque hyperslice par un dossier à la manière des paquetages, portant le stéréotype « hyperslice ». L'identification d'un hyperslice est obtenue par son nom (i.e. le nom de la préoccupation qu'il définit) qui est unique dans le contexte de sa dimension englobante, on parle alors de nom simple. Le nom complet d'un hyperslice est préfixé par le nom de sa dimension englobante de la forme « dimensionName::hypersliceName ». La figure 4.18 montre un exemple d'une représentation d'un hyperslice.

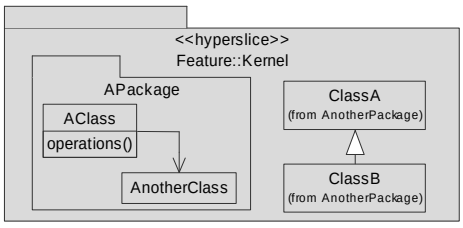


Figure 4.18. *HyperJ/UML — Représentation des hyperslices*

***Les hypermodules et règles d'intégration spécifiques* Summary *et* Rename**. Les hypermodules sont représentés ensemble avec certaines de leurs règles d'intégration spécifiques, à l'aide de rectangles compartimentés (cf. figure 4.19). Le premier compartiment symbolise l'hypermodule, il contient son nom et indique sa stratégie de composition générale de la forme « hypermoduleName: compositionStrategy ». Il contient également le stéréotype particulier « hypermodule ». Le deuxième compartiment est dédié aux règles d'intégration spécifiques de type Summary et Rename, mettant en jeu exactement une unité intégrable en entrée.

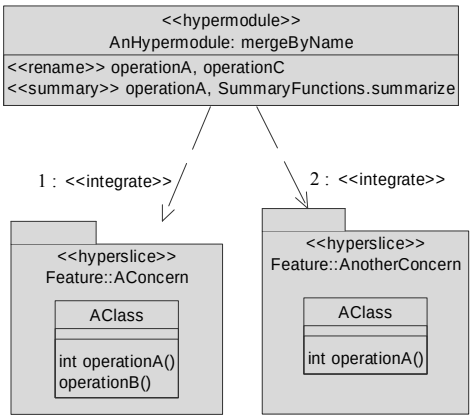


Figure 4.19. *HyperJ/UML — Représentation des hypermodules et des règles d'intégration spécifiques*

Chaque hypermodule est lié aux hyperslices qu'il intègre par des relations d'utilisation (Usage, cf. Annexe A) stéréotypées. En effet, toute relation entre un hypermodule et l'un de ses hyperslices est une relation unidirectionnelle, indiquant que l'hypermodule requiert la présence de ce dernier ainsi que de ses unités intégrables pour son implémentation. Nous avons ainsi décidé d'utiliser la relation d'utilisation avec le stéréotype particulier « integrate » pour lier les hypermodules aux hyperslices qu'ils intègrent. Remarquons par ailleurs que l'ordre de précédence entre les hyperslices d'un hypermodule donné est rendu par des indices (i.e. des entiers), attachés aux différentes relations de dépendance existant entre cet hypermodule et ces hyperslices. Par exemple, dans la figure 4.19 l'hyperslice Feature::AConcern domine l'hyperslice Feature::AnotherConcern.

Nous pouvons distinguer également dans cette même figure deux exemples de règles d'intégration spécifiques de types Rename et Summary. Remarquons que les règles instances de Rename sont représentées sous la forme « rename » operationName, newOperationName. La première règle de AnHypermodule en donne un exemple, indiquant que la méthode résultat de la fusion des deux opérations operationA() doit être renommée operationC(). Quant aux règles instances de Summary, elles sont représentées sous la forme « summary » operation, summaryFunction, comme le montre l'exemple de la figure 4.19. Cet exemple indique que le résultat de retour final des deux opérations operationA() à fusionner est déterminé par l'opération summarize() du paquetage SummaryFonctions.

Règles d'intégration spécifiques. À la différence des règles Rename et Summary, les autres règles d'intégration spécifiques (Match, Equate, Merge, NoMerge, Order, Bracket et Override) mettent en relation deux ou plusieurs unités intégrables en entrée. Nous proposons de présenter ces règles par des liens binaires en traits pointillés, connectant exactement deux unités intégrables. Plusieurs mots clés ou stéréotypes indiqués entre guillemets sont utilisés conjointement avec ces liens pour préciser le type de la règle représentée. Ainsi, une règle Equate mettant en relation, par exemple, trois unités intégrables est représentée par deux liens stéréotypés (« equate ») reliant deux à deux les trois unités considérées. Remarquons que dans le cas de cette dernière règle, ainsi que dans les cas des règles Match, Merge et

NoMerge il n'est pas important d'orienter les traits liant les unités intégrables. En ce qui concerne les traits représentant les règles d'intégration spécifiques Order, Bracket et Override ils doivent pointer vers les unités intégrables cibles d'impact. Ainsi, une instance de la règle Override est représentée, par exemple, par une flèche en traits pointillés pointant vers l'unité intégrable destinée à être remplacée. Le tableau 4.2 détaille les notations des différentes règles d'intégration spécifiques possibles en Hyper/J, et rappelle la sémantique de chacune de ces règles.

Tableau 4.2. *HyperJ/UML — Représentation des règles d'intégration spécifiques*

Règles	Notation	Sémantique
Match	« match »	Indique que les deux unités intégrables jointes se correspondent, qu'elles soient de noms identiques ou différents. Les deux unités connectées doivent être de même type : Class, Interface, Operation, Method ou Attribut.
Equate	« equate »	Même sémantique que Match. Cette relation peut connecter également deux unités intégrables de type Class, Interface, Operation, Method ou Attribut.
Merge	« merge »	Indique que les deux unités liées doivent être fusionnées. Elle peut connecter des unités intégrables de type Class, Interface, Operation, Method ou Attribut.
NoMerge	« noMerge »	Contrairement à Merge, cette relation indique que les deux unités liées ne doivent pas être fusionnées. Les unités connectées doivent avoir des types identiques : Class, Interface, Operation, Method ou Attribut.
Override	« override »	Indique que la définition de l'unité intégrable source doit remplacer celle de l'unité cible. Les deux unités connectées doivent être de même type : Class, Interface, Operation ou Method.
Order before	« order before »	Connecte deux unités intégrables de même type. Elle indique que l'unité source précède l'unité cible dans le cas d'une relation de type Merge. Les unités connectées peuvent être de type Hyperslice, Class, Interface, Operation ou Method.
Order after	« order after »	Indique que l'unité source suit l'unité cible dans le cas d'une relation de type Merge. Les unités connectées peuvent avoir comme type : Hyperslice, Class, Interface, Operation ou Method. Elles doivent cependant avoir un même type.
Bracket before	« bracket before » {from = callSite }	Indique que l'unité source doit être précédée par l'unité cible dans le cas d'une relation de type Merge. Les unités connectées doivent être d'un même type qui peut être Operation ou Method. La valeur marquée from peut être utilisée pour indiquer le site d'appel de la méthode/opération devant être précédée par l'unité cible.
Bracket after	« bracket after » {from = callSite }	Indique que l'unité source doit être suivie par l'unité cible dans le cas d'une relation de type Merge. Les unités connectées doivent avoir un type identique Operation ou Method. De la même façon que « bracket before » la valeur marquée from peut être utilisée pour indiquer le site d'appel de la méthode/opération devant être suivie par l'unité cible.

2.4 Aspect/UML : une extension d'UML pour la modélisation par aspects

Suite à la présentation dans les deux sous-sections précédentes de AspectJ/UML et HyperJ/UML, nous rapprochons dans cette partie (cf. § 2.4.1) les concepts fondamentaux des deux langages, afin d'identifier un ensemble d'éléments plus généraux qui leur soient communs. Ces éléments constituent une base pour la définition d'un métamodèle général à l'approche Aspect dont nous détaillons la syntaxe abstraite dans la sous-section 2.4.2.

2.4.1 Rapprochement d'AspectJ/UML et HyperJ/UML

Il ne s'agit pas d'une couverture exhaustive de tous les concepts spécifiques, mais d'un rapprochement des principaux concepts proposés par les deux métamodèles spécifiques. Le tableau 4.3 propose une juxtaposition des principaux concepts voisins.

Tableau 4.3. *Juxtaposition des concepts fondamentaux de AspectJ/UML et HyperJ/UML*

Concepts fondamentaux		AspectJ	Hyper/J
Unité modulaire d'encapsulation d'une préoccupation transversale		Aspect	Hyperslice
Entrecroisement statique	Propriétés d'entrecroisement	– Feature (/introducedMember) propres à une Introduction et donc à un Aspect – Relations d'Abstraction ou de Generalization (/introducedAbstraction, /introducedGeneralization) propres à une DeclareParents et donc à un Aspect	– Feature (/composableUnit) propres aux unités intégrables composites de type Classifier encapsulées par un Hyperslice – Relations d'Abstraction ou de Generalization définies entre les unités intégrables composites (Classifier) d'un Hyperslice
	Eléments impactés	Classifier (/targetType d'une Introduction, /child d'une Generalization ou /client d'une Abstraction)	Classifier (/composableUnit d'un Hyperslice jouant le rôle d'une préoccupation impactée)
	Spécifications d'entrecroisement	Spécification du type impacté à travers l'association /targetType d'une Introduction propre à un Aspect	– Stratégie de composition générale d'un Hypermodule : mergeByName ou NonCorrespondingMerge – Règles d'intégration spécifiques de type Merge propres à un Hypermodule
Entrecroisement dynamique	Propriétés d'entrecroisement	Advice (perfectionnements propres à un Aspect déterminés par son association /crosscuttingFeature)	BehavioralFeature (/composableUnit propres aux unités intégrables composites de type Class, qui sont encapsulées par un Hyperslice)
	Eléments impactés	BehavioralFeature (/behavioralFeature d'un MethodRelatedDesignator)	BehavioralFeature (/composableUnit propres aux unités intégrables composites de type Class encapsulées par un Hyperslice jouant le rôle d'une préoccupation impactée)
	Spécifications d'entrecroisement	– Les points de recouvrement de type MethodRelatedDesignator – Type des perfectionnements (attributs adviceType et afterQualifier d'un Advice)	– Stratégie de composition générale d'un Hypermodule : mergeByName, NonCorrespondingMerge ou overrideByName – Règles d'intégration spécifiques propres à un Hypermodule
Relations d'entrecroisement		Relations de Crosscut stéréotypées addFeature, addParent et intercept	– Stratégie de composition générale d'un Hypermodule – Règles d'intégration spécifiques de type Override et Merge
Relations de précédence		– Relation de Precedence entre aspects – Ordre de déclaration des perfectionnements (Advice) dans un aspect – Règles implicites : before, after, et around déterminées par l'attribut adviceType d'un Advice	– Ordre de déclaration des hyperslices dans un hypermodule – Règles d'intégration spécifiques de type Order – Règles d'intégration spécifiques de type Bracket

Comme le montre le tableau précédent, nous distinguons principalement six éléments fondamentaux propres à l'approche Aspect en général, qui sont pris en compte par AspectJ ainsi que par Hyper/J.

- *Éléments de base*. Ce sont les éléments impactés et concernés par une ou plusieurs préoccupations transversales : Classifier dans le cas d'un entrecroisement statique et BehavioralFeature dans le cas d'un entrecroisement dynamique.
- *Éléments d'encapsulation des préoccupations transversales*. Dans les deux cas d'AspectJ/UML et HyperJ/UML, un nouvel élément de modélisation encapsulant une préoccupation transversale est introduit : Aspect, respectivement Hyperslice.
- *Éléments d'entrecroisement*. Il s'agit des propriétés structurelles et de comportement propres à une préoccupation transversale donnée, qui concernent les éléments de base. Nous distinguons principalement des Features dans le cas d'un entrecroisement statique, des Advices et des BehavioralFeatures dans le cas d'un entrecroisement dynamique. Les relations de généralisation (Generalization) ou de réalisation (Abstraction) peuvent également constituer des éléments d'entrecroisement, dans le cas d'un entrecroisement statique.
- *Éléments de spécification d'entrecroisement*. Ils indiquent les endroits auxquels une unité modulaire d'encapsulation d'une préoccupation transversale peut interagir statiquement (*i.e.* en ajoutant de nouvelles propriétés structurelles et/ou de comportement) ou dynamiquement (*i.e.* en altérant le comportement prédéfini) avec les éléments de base. Hyper/J se sert des stratégies générales et des règles d'intégration (IntegrationRule) pour ce faire. Quant à AspectJ, il se base principalement sur la déclaration de points de recouvrement de type MethodRelatedDesignator, et sur la spécification des propriétés structurelles des perfectionnements (i.e. attributs adviceType et afterQualifier des Advices).
- *Relations de précédence*. Les deux langages traitent du problème de précédence soulevé lors de l'application de plusieurs préoccupations dans un même contexte. Chacun propose de définir une relation de précédence régissant le bon fonctionnement du programme : Precedence dans le cas d'AspectJ et ordre de déclaration des hyperslices dans le cas d'Hyper/J. Ils permettent également de définir des relations de précédence qui sont liées directement aux propriétés d'entrecroisement altérant les propriétés des éléments de base. Hyper/J se sert des règles d'intégration spécifiques Order et Bracket pour ce faire, alors qu'AspectJ se base sur la définition de l'ordre de déclaration des perfectionnements (propres à un même aspect) et la spécification de leurs propriétés.
- *Relations d'entrecroisement*. Les deux langages définissent différentes relations d'entrecroisement mettant en relation les propriétés d'un élément de base et celles d'un élément d'entrecroisement, indiquant que le premier est concerné par le second. Il s'agit des différentes relations Crosscut stéréotypées dans le cas d'AspectJ et de la stratégie de composition générale d'un Hypermodule dans le cas d'Hyper/J, qui se sert également des règles d'intégration Merge et Override pour préciser de plus les relations d'entrecroisement.

Le reste des concepts définis dans les deux métamodèles que nous proposons sont, à première vue, assez éloignés et très spécifiques et ne peuvent de ce fait être rapprochés de façon significative. En revanche, nous retenons l'abstraction de ces six éléments communs éminents comme base pour la construction de notre métamodèle général à l'approche Aspect. Nous en détaillons la syntaxe abstraite dans ce qui suit et présentons quelques règles de bonne formation complétant la spécification des métaclasses représentant l'ensemble de ces six éléments.

2.4.2 Concepts et relations fondamentaux du métamodèle général

Par rapprochement des concepts (et relations) spécifiques à AspectJ d'une part et à Hyper/J d'autre part, nous avons isolé dans la section précédente six concepts communs à ces deux langages. Nous les spécifions ci-dessous dans un métamodèle (Aspect/UML) qui regroupe de ce fait les éléments nécessaires à une modélisation par aspects indépendante, pouvant être par la suite transformée en une modélisation spécifique à l'un ou l'autre de ces deux langages. Nous présentons notamment la syntaxe abstraite de chaque élément et quelques règles de bonne formation (cf. Annexe B pour la spécification complète d'Aspect/UML).

Préoccupations. Dans l'approche Aspect, nous distinguons en général deux types de préoccupations : les préoccupations de base et les préoccupations transversales. La figure 4.20 montre les différents types de préoccupations possibles.

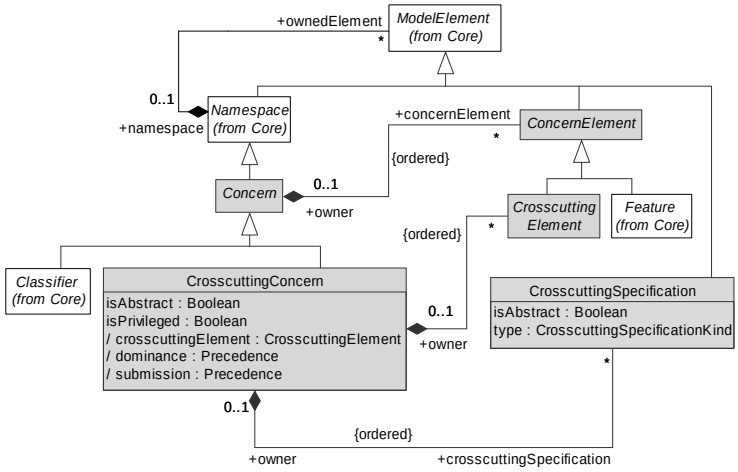


Figure 4.20. *Aspect/UML — Syntaxe abstraite des préoccupations*

Un Concern est une préoccupation transversale ou de base, qui peut avoir zéro ou plusieurs éléments de préoccupation (ConcernElement) de type Feature et/ou CrosscuttingElement (i.e. éléments d'entrecroisement), selon qu'il s'agit d'une préoccupation de base ou transversale. Les préoccupations de base sont les éléments de modélisation pouvant être impactés par zéro ou plusieurs préoccupations transversales. Comme les éléments de base dans les deux langages sont des classificateurs ou des propriétés comportementales, nous n'ajoutons pas une nouvelle classe au métamodèle d'UML pour les spécifier. Une préoccupation de base est donc plus spécifiquement instance de Classifier qui peut être impactée directement ou indirectement (c'est-à-dire à travers ses propriétés de comportement). Une préoccupation de base ne peut pas par ailleurs contenir des éléments de préoccupation de type CrosscuttingElement, d'où la règle de bonne formation suivante par exemple :

```
context Classifier inv :
        self.concernElement->forAll (ce: ConcernElement | ce.oclIsKindOf (Feature))
```

CrosscuttingConcern est l'unité d'encapsulation des préoccupations transversales. Il peut avoir en plus des propriétés standards (Feature), des éléments d'entrecroisement (CrosscuttingElement) et des éléments de spécification d'entrecroisement (CrosscuttingSpecification). Un CrosscuttingConcern peut être abstrait (attribut isAbstract) et peut ainsi contenir des spécifications d'entrecroisement abstraites. Il peut de plus être privilégié à la manière des aspects en AspectJ, et peut donc avoir visibilité sur toutes les propriétés des préoccupations de base qu'il impacte. Remarquons par ailleurs, qu'un CrosscuttingConcern peut participer dans plusieurs relations de précédence comme étant dominant (/dominance) ou dominé (/submission).

Éléments d'entrecroisement. La figure 4.21 montre l'ensemble des éléments d'entrecroisement possibles. Il s'agit des éléments d'une préoccupation transversale donnée affectant les éléments de base. Nous distinguons principalement les éléments d'altération (AlteringElement) et les éléments augmentant les définitions des éléments de base (AdditiveElement).

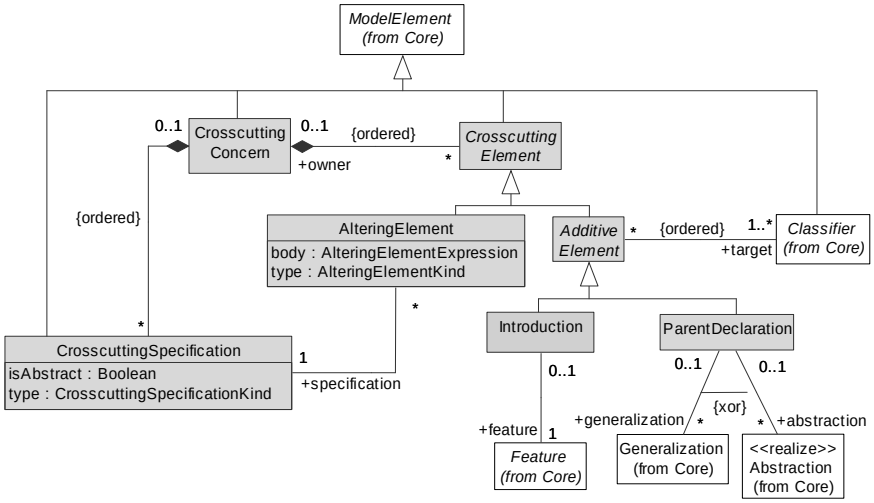


Figure 4.21. *Aspect/UML — Syntaxe abstraite des éléments d'entrecroisement*

Un AlteringElement est destiné à altérer des comportements (i.e. des propriétés comportementales) définis dans des préoccupations de base, il est associé à exactement une spécification d'entrecroisement qui spécifie l'ensemble des propriétés de comportement qu'il impacte. Il est caractérisé notamment par l'attribut type que nous détaillons dans ce qui suit.

Attributs

body	Spécifie le corps de l'élément d'altération.
type	Spécifie la manière dont l'élément d'altération doit modifier la ou les propriétés de comportement qu'il impacte. Les valeurs possibles sont : before, after, combination (before et after), replacement et narrowing (exécution sur condition) (cf. Annexe B pour les sémantiques exactes).

Une Introduction est une spécialisation de la métaclasse abstraite AdditiveElement. Elle consiste en l'ajout dans une ou plusieurs préoccupations de base (/target) d'une propriété structurelle ou comportementale (/feature). ParentDeclaration est une autre spécialisation de AdditiveElement, permettant d'étendre la définition d'une ou de plusieurs classes en intervenant sur les hiérarchies de leurs ancêtres directs, qu'ils soient des classes ou des interfaces. Elle peut donc être liée à une ou plusieurs relations de généralisation (Generalization) ou de réalisation (Abstraction), mais pas les deux types de relations à la fois.

Spécifications d'entrecroisement. Une spécification d'entrecroisement (CrosscuttingSpecification) indique le contexte structurel d'entrecroisement, à travers son association /target qui spécifie l'ensemble des propriétés de comportement impactées. Elle spécifie de plus s'il s'agit d'altérer le comportement d'une opération/méthode quelconque ou d'un constructeur, grâce à son attribut type détaillé ci-dessous. Chaque spécification d'entrecroisement peut être associée à zéro ou plusieurs éléments d'altération. Elle est abstraite si elle n'est associée à aucune propriété de comportement, dans un tel cas sa préoccupation transversale doit être également abstraite. D'où la règle de bonne formation suivante :

```
context CrosscuttingSpecification inv :
        (self.isAbstract = #true) implies (self.owner.isAbstract = #true)
```

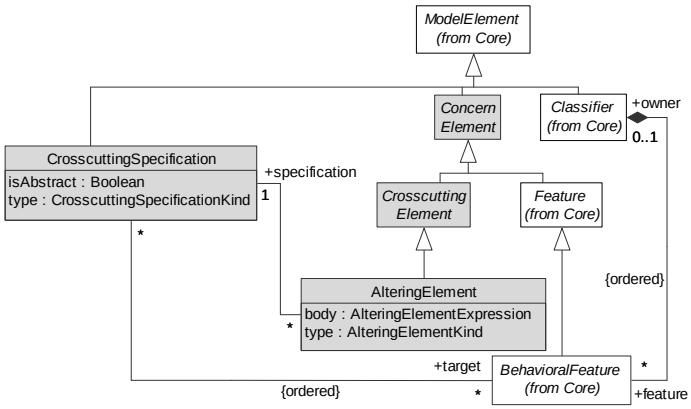


Figure 4.22. *Aspect/UML — Syntaxe abstraite des spécifications d'entrecroisement*

Attributs

isAbstract — Indique si la spécification est abstraite (true) ou pas (false, la valeur par défaut).

type — Indique le type de la ou des propriétés de comportement cibles d'impact. Les valeurs possibles sont :

- undefined est la valeur par défaut, elle est obligatoire lorsqu'il s'agit d'une spécification d'entrecroisement abstraite ;
- call indique qu'il s'agit de toute autre opération/méthode que le constructeur ;
- set indique qu'il s'agit d'une méthode accesseur en mode écriture d'un attribut donnée ;
- get indique qu'il s'agit d'une méthode accesseur en mode lecture d'un attribut donnée ;
- initialization indique qu'il s'agit d'un constructeur d'une classe ;
- staticInitialization indique qu'il s'agit d'altérer l'initialisation des comportements statiques d'une classe.

Relations d'entrecroisement et de précédence. En plus des concepts fondamentaux précédents, nous pouvons définir et rajouter deux nouvelles relations qui sont considérées par les deux langages : Crosscutting et Precedence. La figure 4.23 décrit la syntaxe abstraite de ces deux nouvelles relations.

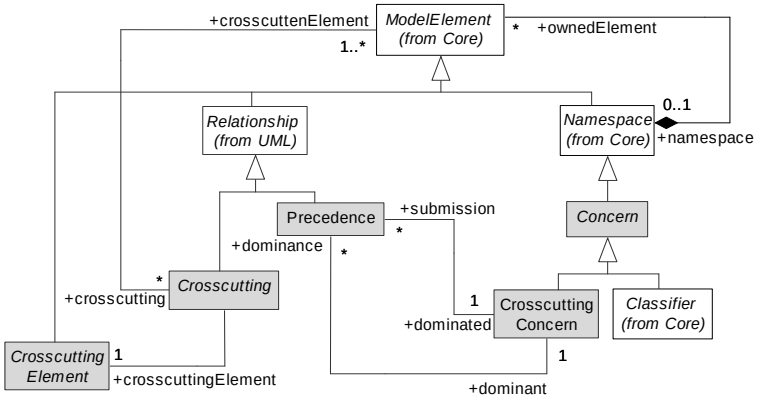


Figure 4.23. *Aspect/UML — Syntaxe abstraite des relations d'entrecroisement et de précédence*

Precedence est une relation directe définie entre exactement deux préoccupations transversales : une dominante déterminée par son association dominant et une dominée définie par son association dominated. Elle établit l'ordre dans lequel celles-ci s'entrecroisent avec les préoccupations de base dans le cas d'un entrecroisement dynamique, indiquant ainsi la manière dont les éléments d'altération concurrents

des préoccupations transversales considérées doivent être fusionnés avec la ou les propriétés comportementales qu'ils altèrent. Remarquons qu'une préoccupation transversale peut participer à zéro ou plusieurs relations de précédence comme étant dominante (/dominance), mais aussi à zéro ou plusieurs relations de précédence comme étant dominée (/submission).

Crosscutting est une relation de dépendance directe entre un élément d'entrecroisement (de type AdditiveElement) ou une spécification d'entrecroisement (CrosscuttingSpecification) source, définie par son association crosscuttingElement, et un élément (ou plusieurs autres éléments) cible(s) déterminé(s) par son association crosscuttenElement. Une relation d'entrecroisement peut mettre en relation, par exemple, une Introduction et un ou plusieurs classificateurs (*Class* ou *Interface*). Il s'agit dans ce cas d'une relation d'entrecroisement particulière. Nous proposons pour distinguer les différentes relations d'entrecroisement possibles de définir des stéréotypes pour Crosscutting que nous détaillons dans ce qui suit, avec deux exemples de règles de bonne formation complétant la spécification de la sémantique exacte de la métaclasse Add (un stéréotype particulier de la métaclasse Crosscutting).

Stéréotypes

add	Représente une relation d'entrecroisement entre un élément d'addition (AdditiveElement) et un ou plusieurs classificateurs, il s'agit d'une relation de Crosscutting stéréotypée.
alter	Représente une relation d'entrecroisement entre une spécification d'entrecroisement (CrosscuttingSpecification) et une ou plusieurs propriétés comportementales (BehavioralFeature) propres à des classificateurs de base, il s'agit d'une relation de Crosscutting stéréotypée.

Règles de bonne formation

- Les éléments d'entrecroisement associés à une relation d'entrecroisement stéréotypée Add doivent être obligatoirement de type Introduction ou ParentDeclaration. Par ailleurs, ses éléments entrecroisés ne peuvent être que de type Class ou Interface.

```
context Add inv :
        self.crosscuttingElement.oclIsKindOf(AdditiveElement)
        self.crosscuttenElement ->forAll(e : Element |
                e.oclIsTypeOf(Class) or e.oclIsTypeOf(Interface))
```

- Si l'élément d'entrecroisement d'une relation Add est de type ParentDeclaration, le nombre d'éléments entrecroisés est égal à 2.

```
context Add inv :
        (self.crosscuttingElement.oclIsTypeOf(ParentDeclaration))
        implies (self.crosscuttenElement->size = 2)
```

2.4.3 Notation graphique des concepts et relations d'Aspect/UML

Nous proposons dans ce qui suit une syntaxe concrète pour les éléments de modélisation du métamodèle Aspect/UML.

Préoccupations transversales et relations de précédence. Les préoccupations transversales (CrosscuttingConcern) sont représentées à la manière des classes, sous la forme de rectangles compartimentés (cf. figure 4.24-(a)). Le premier compartiment symbolise la préoccupation : il contient son nom et le stéréotype « crosscuttingConcern ». Il peut contenir de plus un symbole représentant un œil, dont la présence indique que la préoccupation transversale considérée est privilégiée : elle a visibilité sur les propriétés privées des préoccupations qu'elle impacte. Conformément à UML, un nom en italique précise qu'une préoccupation transversale est abstraite, et deux compartiments contiennent respectivement les attributs et opérations propres à la préoccupation représentée. Ces compartiments

peuvent être éventuellement omis lorsque leur contenu n'est pas pertinent dans un diagramme, pour obtenir une représentation graphique simplifiée (cf. figure 4.24-(b)).

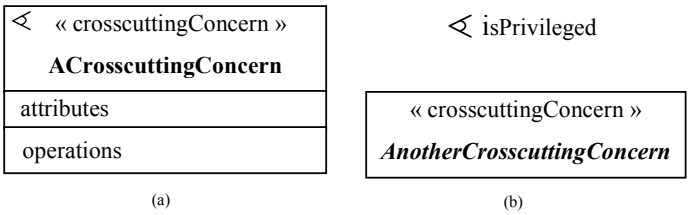


Figure 4.24. *Aspect/UML — Représentation des préoccupations transversales*

Éléments d'addition et relations d'entrecroisement. L'ajout d'une relation de généralisation ou de réalisation par une préoccupation transversale, grâce à une déclaration d'héritage (ParentDeclaration) est représenté par un lien d'entrecroisement dont l'extrémité cible, dénotée par un rectangle contenant une croix, désigne la relation ajoutée (cf. figure 4.25). Un tel lien représente une instance de la relation d'entrecroisement stéréotypée add.

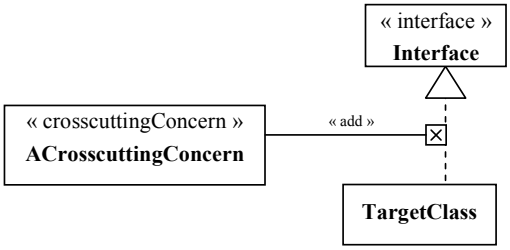


Figure 4.25. *Aspect/UML — Représentation des déclarations d'héritage et de réalisation*

Les introductions (Introduction) sont regroupées par cible d'impact. Elles sont représentées au moyen de rectangles compartimentés (à la façon de classes non nommées), portant le stéréotype « introduction » (cf. figure 4.26-(a)). Chaque introduction est reliée à sa préoccupation transversale par un trait simple, et à sa cible par un lien d'entrecroisement représentant une instance de la relation d'entrecroisement stéréotypée add. Les compartiments d'une introduction contiennent respectivement les attributs et les méthodes destinés à être ajoutés à la cible d'impact. La figure 4.26 montre un exemple d'introduction dans une classe abstraite interne à une préoccupation transversale (b).

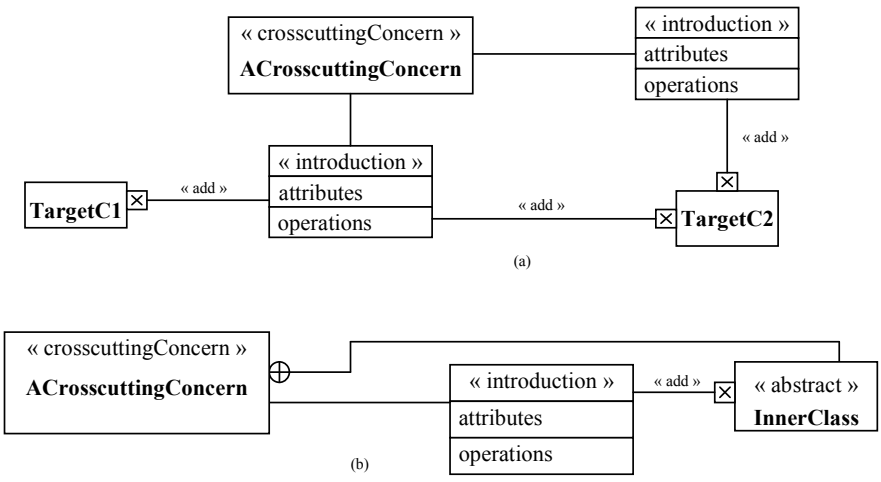


Figure 4.26. *Aspect/UML — Représentation des introductions*

Spécifications d'entrecroisement et éléments d'altération. La représentation des altérations permet de regrouper les éléments d'altération par spécification d'entrecroisement dans un rectangle compartimenté. Le premier compartiment précise la spécification d'entrecroisement ; il contient le stéréotype « altering » ainsi que sa description (cf. figure 4.27). La syntaxe retenue pour la représentation des spécifications d'entrecroisement est la suivante :

[crosscuttingSpecification_name] : crosscutting-Specification_type
avec crosscuttingSpecification_type ::= call | set | get | initialisation | staticInitialization.

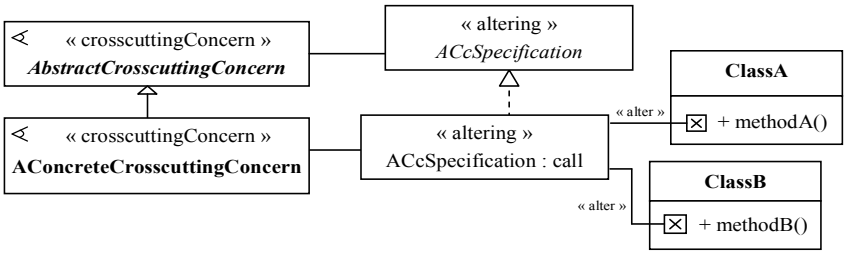


Figure 4.27. *Aspect/UML — Représentation des spécifications d'entrecroisement*

Remarquons que chaque spécification d'entrecroisement est liée à une ou plusieurs opérations impactées, grâce à un ou plusieurs liens d'entrecroisement avec le stéréotype « alter », à moins qu'elle soit abstraite. De tels liens représentent une instance de la relation stéréotypée Alter. Une spécification d'entrecroisement abstraite est dénotée par un nom en italique sans type.

Le deuxième compartiment est dédié aux éléments d'altération (cf. figure 4.28). La syntaxe retenue pour la description de ces éléments est la suivante.

alteringElement_ type ["(" arguments ")"] [':' return_type]
avec arguments ::= argument_name ':' argument_ type [',' arguments]
alteringElement_type ::= before | after | combination | replacement | narrowing

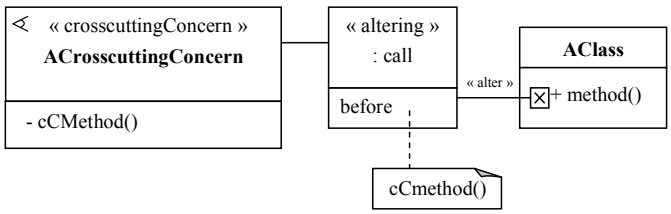


Figure 4.28. *Aspect/UML — Représentation des éléments d'altération*

Relations de précédence. Les relations de précédence entre préoccupations transversales sont représentées par des flèches à pointes triangulaires pleines, orientées de la préoccupation dominante vers la préoccupation dominée (cf. figure 4.24).

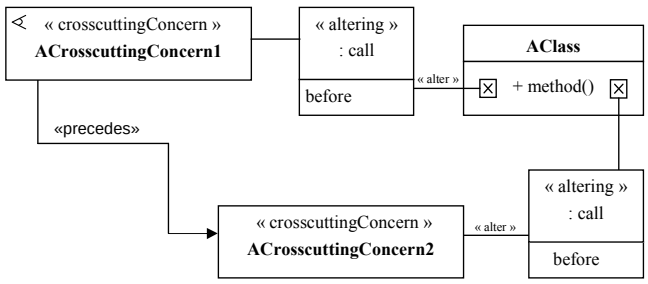


Figure 4.29. *Aspect/UML — Représentation des relations de précédence*

3. Transformation de modèles : d'un modèle général vers des modèles spécifiques à AspectJ et Hyper/J

Maintenant que nous disposons d'un moyen permettant l'expression de conceptions par aspects indépendantes d'une technique de programmation par aspects particulière, nous nous intéressons à la transformation de ces conceptions pour qu'elles deviennent plus spécifiques et puissent ainsi être facilement implémentées dans les langages de programmation par aspects actuels. Nous nous concentrons plus particulièrement dans le cadre de cette thèse aux projections vers AspectJ et Hyper/J, pour concrétiser notre approche de modélisation par aspects. Nous proposons pour ce faire des transformations de modèles permettant de projeter un modèle instance du métamodèle général vers un modèle instance de l'un des deux métamodèles spécifiques que nous proposons, respectivement, à AspectJ et Hyper/J. Notre démarche de modélisation par aspects s'inspire en effet du standard MDA (*Model Driven Architecture*) [OMG-MDA 03] ou de l'ingénierie dirigée par les modèles (IDM, en français) dont nous rappelons succinctement les principaux concepts dans la section 3.1, afin de situer notre proposition et donc définir le processus de transformation que nous adoptons et présentons dans la section 3.2.

3.1 Ingénierie dirigée par les modèles

L'objectif de cette section est de présenter brièvement l'approche IDM, afin de comprendre l'idée principale et les notions fondamentales de cette démarche. Nous ne cherchons pas cependant à couvrir l'intégralité de ses concepts ; nous nous concentrons plutôt sur les concepts que nous jugeons nécessaires pour comprendre notre proposition.

L'ingénierie dirigée par les modèles est une démarche de développement proposée par l'OMG (*Object Management Group*). Elle vise à fournir un cadre de développement basé sur différents modèles qui peuvent être exprimés dans plusieurs formalismes, permettant ainsi d'aborder différentes problématiques (notamment la séparation des spécifications fonctionnelles des spécifications non fonctionnelles) du génie logiciel dans un modèle unificateur. IDM conçoit l'intégralité du cycle de développement du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles, basé sur des règles définissant la cohérence entre les différents modèles utilisés et la manière de les transformer ou les composer au cours du développement. Nous pouvons entrevoir à travers cette définition générale que trois notions fondamentales sont au cœur de la démarche MDA : (1) le concept de modèle, (2) de formalisme et (3) de transformation de modèles.

3.1.1 Modèles et métamodèles

Un modèle est une description d'un système écrite dans un langage bien défini. L'utilisation des modèles permet de capturer les structures et les comportements des différentes vues d'un système à différents niveaux d'abstraction des détails de sa solution. Le standard MDA fait ainsi la différence, par exemple[18], entre modèles indépendants (PIM pour *Platform Independent Model*) et modèles dépendants des plate-formes d'exécution (PSM pour *Platform Specific Model*). Il définit en ce sens le processus de développement comme étant une transformation progressive d'un modèle PIM, qui spécifie la solution d'un système indépendamment des technologies de programmation, vers un modèle PSM qui décrit comment cette solution peut être implémentée dans une technologie particulière. Une telle transformation passe nécessairement par une formalisation des modèles manipulés, devant être écrits dans des langages de modélisation précis et clairement définis. Il s'agit du deuxième concept fondamental de l'approche MDA :

[18] Dans l'IDM, tout est considéré comme modèle, aussi bien les schémas que le code source ou le code binaire. Quatres types de modèles sont ainsi définis par l'IDM : les CIM, les PIM, les PDM et les PSM [OMG-MDA 03].

les formalismes ou langages de modélisation. Chaque langage est défini par la modélisation des concepts utilisés par les modèles à formaliser, il devient donc lui-même un sujet de modélisation et peut être représenté, comme tout système logiciel, par plusieurs modèles. Cette idée introduit la notion de métamodèle qui est défini tout simplement comme étant le modèle d'un langage de modélisation. Les usages et applications des langages de modélisation sont très divers. Chaque langage permet une spécification de la sémantique exacte des concepts à modéliser et de la façon de les combiner (i.e. leurs relations) pour créer un modèle bien formé, conforme à une syntaxe abstraite (i.e. le modèle du langage de modélisation ou métamodèle). Il offre aussi une notation pour la construction de tels modèles, qui est définie par une syntaxe concrète. Des correspondances entre concepts spécifiques à plusieurs langages de modélisation peuvent également être mises en place, pour développer des transformations entre modèles écrits dans des langages de modélisation différents [Bézivin *et al.*, 02]. Il s'agit du troisième concept clé de l'IDM : la transformation de modèles (notamment des PIM vers des PSM) que nous explicitons dans ce qui suit.

3.1.2 Transformation de modèles

La mise en œuvre de l'IDM est entièrement fondée sur les modèles et leurs transformations. Cependant, si les spécifications existantes telles qu'UML fournissent un moyen standard pour définir les modèles à transformer, la spécification de l'IDM concernant les techniques de transformation reste très informelle. Nous essayons dans ce qui suit d'expliciter le processus de transformation et quelques approches de transformation préconisées par l'IDM.

L'IDM définit des transformations entre divers types de modèles : de PIM vers PIM, de PIM vers PSM, de PSM vers PSM, etc. [OMG-MDA 03]. Toutefois, un processus de transformation de modèles est composé en général de trois tâches : (1) la définition des règles de transformation, (2) l'expression de telles règles et (3) l'exécution de celles-ci. La définition des règles de transformation est très ouverte, car l'IDM a une vocation unificatrice qui promeut la convergence des résultats d'un grand nombre de disciplines de développement logiciel. La détermination de ces règles doit être cependant précisée dans des contextes de transformation plus concrets. En effet, l'IDM préconise différentes approches de transformation de modèles dont les deux principales[19] sont : l'approche par marquage (i.e. annotations) (cf. figure 4.30-(b)) et l'approche guidée par les métamodèles (cf. figure 4.30-(a)).

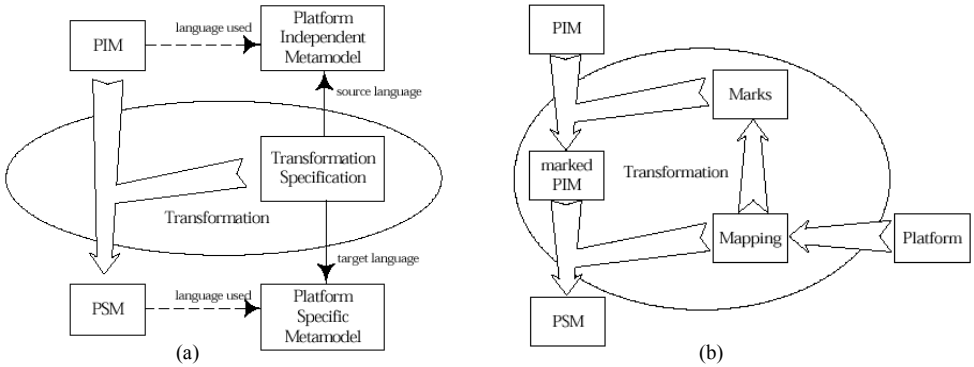


Figure 4.30. *Approches MDA de transformation de modèles* [OMG-MDA 03]

Si plusieurs travaux [Lemesle 98], [Bézivin 01], [Belaunde *et al.*, 02], [Peltier 02] ont étudié l'approche basée sur les métamodèles, il n'en est pas de même pour l'utilisation des annotations qui soulève plusieurs

[19] D'autres approches hybrides ou dérivées de ces deux approches existent ; elles ne sont pas cependant étudiées en détail par la spécification de MDA.

interrogations : quelles formes ont-elles ? Sont-elles spécifiques aux modèles PSM ou aux modèles PIM ? Existe-t-il une technique particulière permettant l'ajout de telles annotations aux modèles afin d'introduire les informations nécessaires à une projection particulière ? Etc. Nous considérons donc la technique de transformation par métamodèles pour expliciter le processus de transformation, notamment de PIM vers PSM. Dans ce cas particulier de transformation, la définition des règles de transformation consiste en la mise en correspondance des concepts communs entre le métamodèle source (PIMetamodel) et le métamodèle cible (PSMetamodel). Une fois identifiées, ces règles doivent être spécifiées dans un langage de spécification de règles donné, afin d'être exécutées. Cependant, il n'existe pas de langage standard. Des réponses à l'appel à proposition (RFP) de l'OMG pour la standardisation du processus de transformation ont proposé divers langages de différentes natures : déclaratif (comme XSLT pour *eXtensible Stylesheet Language Transformation*, permettant la spécification de règles XSL entre des représentations XML de métamodèles), impératif (tels que la réponse française TRL pour *Transformation Rules Language* [Alcatel *et al.*, 03], ou encore SimpleTRL qui est une simplification de TRL réalisée par [Sriplakich 03]), et hybride (c'est-à-dire déclaratif et impératif en même temps, tels que Mtrans [Peltier 03]). Toutes ces propositions ont conduit à la spécification du standard MOF/QVT (Queries/Views/Transformation) [OMG-QVT 05], qui est de nature hybride (déclaratif et impératif) et qui propose trois langages de spécification de transformations : le langage de relations (*Relations language*) et le langage *Core language* (deux langages déclaratifs permettant l'expression des mêmes règles de transformation à deux niveaux d'abstraction différents) et le langage standard impératif *Operational Mapping*. Ce dernier permet la spécification de suites d'opérations de transformation implémentant les relations (ou *core*) considérées, afin que les règles de transformation puissent être exécutées par un moteur de transformation.

Notons enfin que la transformation de modèles basée sur les métamodèles est une tâche très complexe. Les difficultés peuvent porter, par exemple, sur la définition des règles de transformation, leurs caractéristiques attendues (réversibles, etc.), le séquencement des opérations de transformation, la complétude du processus de transformation, etc. En effet, nous avons adopté la démarche de l'IDM pour compléter la définition de notre approche de modélisation par aspects, en étudiant l'approche de transformation par métamodèles, et nous avons été confrontés à diverses difficultés supplémentaires lors de la définition des règles de transformation. Nous expliquons dans ce qui suit ces difficultés et précisons l'approche et le processus de transformation que nous adoptons dans notre approche de modélisation par aspects.

3.2 Objectifs et processus de transformation adopté

Suite à la définition dans la section 2.4 du métamodèle général Aspect/UML, et des deux métamodèles AspectJ/UML (cf. 2.2) et HyperJ/UML (cf. 2.3) spécifiques, respectivement, à AspectJ et Hyper/J, nous complétons ici notre approche de modélisation par aspects par la proposition de transformations de modèles à l'instar de l'IDM. Notre objectif est de définir un processus unifié de transformation pouvant s'appliquer à toute transformation d'un modèle général vers un modèle spécifique, en dehors des contextes d'AspectJ et d'Hyper/J.

Pour ce faire, nous avons étudié, dans un premier temps, l'approche de transformation par métamodèles. Dans le cas de la figure 4.30-(a) présentée auparavant (cf. page 147), Aspect/UML représente le *Platform Independent Metamodel*, alors qu'AspectJ/UML et HyperJ/UML correspondent à deux exemples de *Platform Specific Metamodel*. Par ailleurs, les modèles instances d'Aspect/UML représentent des PIM, que nous proposons de transformer vers des modèles instances de l'un des deux métamodèles spécifiques à AspectJ ou Hyper/J et qui correspondent aux modèles PSM. Nous avons commencé par établir les correspondances entre les concepts communs des trois métamodèles considérés, pour définir

nos règles de transformation. Toutefois, nous avons été confrontés essentiellement à deux difficultés majeures.

La première difficulté porte sur la flexibilité de la projection. En effet, après avoir étudié les correspondances des concepts d'Aspect/UML vers les concepts spécifiques respectivement à AspectJ/UML et Hyper/J/UML, nous avons classé les concepts d'Aspect/UML dans deux catégories :

- un concept général admet un unique correspondant direct dans les concepts spécifiques ;
- un concept général admet plusieurs correspondants possibles dans les concepts spécifiques.

Le premier cas est trivial car la transformation est quasiment directe, ceci concerne notamment l'ensemble des concepts propres à l'Objet, mais aussi les concepts Aspect uniquement dans le cas de la projection vers AspectJ. Le deuxième cas est relativement complexe car il faut identifier quelles sont les différentes projections alternatives envisageables. Ce dernier cas concerne notamment la transformation de certains concepts d'Aspect/UML vers leurs correspondants spécifiques à HyperJ/UML, et plus précisément la transformation des éléments d'altération et leurs spécifications d'entrecroisement dans le cas d'une altération. Ceci est dû au fait qu'Hyper/J offre diverses règles d'intégration spécifiques qui, combinées avec les stratégies de composition générales possibles, permettent des solutions alternatives pour la réalisation d'une même adaptation attendue. La transformation dans ce dernier cas n'est pas directe et nécessite une certaine flexibilité pour ne pas limiter le concepteur à une seule règle de transformation qui n'est pas forcément la plus appropriée dans certains cas.

La deuxième difficulté réside dans le besoin d'annotations typiques à la projection dans un métamodèle spécifique cible, dans le but de permettre la génération de code à partir d'un modèle spécifique par exemple. En effet, après avoir déterminé les correspondances entre les concepts du métamodèle général et ceux des métamodèles spécifiques, nous avons examiné de près leurs propriétés caractéristiques (i.e. les attributs de chaque métaclasse modélisant un concept considéré), et nous avons remarqué que certains détails absents au niveau des éléments d'un modèle général sont nécessaires pour assurer une transformation complète de celui-ci vers un modèle spécifique affiné. Par exemple, en projetant un modèle général vers un modèle spécifique à AspectJ, on a besoin d'assigner une valeur à l'attribut instantiation (singleton, perThis, etc. (cf. figure 4.4 et figure 4.9)) pour chaque aspect, afin de pouvoir générer du code.

Pour pallier ces deux difficultés, une solution possible préconisée par l'IDM est l'ajout d'annotations (cf. figure 4.30-(b)) aux modèles PIM, permettant d'introduire les informations nécessaires à une projection donnée vers un PSM et de guider le processus de transformation par le choix de règles de transformation à appliquer. Nous proposons donc d'adopter une approche de transformation de modèles dite hybride, utilisant conjointement l'approche par marquage et l'approche par métamodèles. Ainsi, nous sommes amenés à doter les éléments des modèles généraux d'annotations qui peuvent être définies sous la forme de valeurs marquées (*tagged values*), et peuvent avoir éventuellement des valeurs par défaut. De telles annotations doivent permettre de sélectionner une règle de transformation parmi plusieurs alternatives possibles (dans le cas de la projection vers Hyper/J, par exemple), mais aussi de compléter les informations nécessaires à une projection particulière. Le processus de transformation que nous proposons est ainsi basé sur trois modèles (cf. figure 4.31). Le premier modèle (1) correspond au modèle instance du métamodèle général Aspect/UML. Le deuxième (2), dénommé modèle général intermédiaire, est basé sur le premier mais doté en plus d'annotations spécifiques à la projection cible. Le troisième (3) est le modèle résultat, instance du métamodèle spécifique cible. Comme la spécification MDA concernant la transformation par marquage reste très informelle, et que les rares expériences de transformations de modèles se basent principalement sur l'approche par métamodèles, nous ne disposons actuellement d'aucun moyen précis permettant l'introduction et l'exploitation de telles annotations.

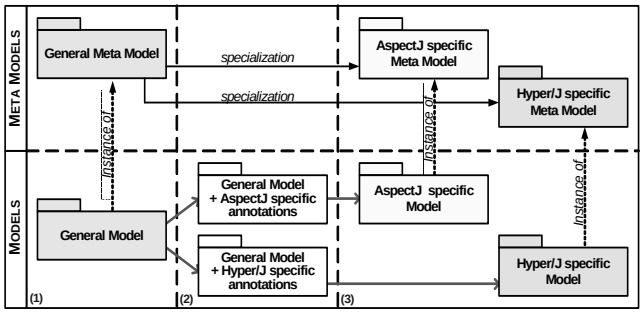


Figure 4.31. *Processus de transformation*

Ceci étant, nous nous concentrons dans une première étape de ce travail de transformation de modèles sur la définition des relations existantes entre les concepts communs des trois métamodèles. Nous avons défini et spécifié des règles de transformation entre les principaux concepts d'Aspect/UML et ceux spécifiques respectivement, à AspectJ/UML et HyperJ/UML. Ces règles doivent être complétées dans des travaux futurs pour prendre en compte les annotations nécessaires. Nous utilisons le langage de relations défini par [OMG-QVT 05] pour la spécification de ces règles. Reste à noter enfin que nous ne considérons pas, dans ce début de travail, l'application concrète de telles règles de transformation, ni l'élaboration d'un moteur de transformation. Nous proposons dans les deux sous-sections suivantes d'expliciter quelques unes des règles de transformation que nous proposons. Nous nous concentrons sur les règles de transformation produisant des éléments de modélisation spécifiques à l'approche Aspect (tel que Aspect et Hyperslice à partir d'un CrosscuttingConcern, etc.) ; les éléments de modélisation Objet (tel que Class, Interface, Operation, etc.) présents dans le métamodèle général restant inchangés. Nous présentons notamment les règles nécessaires à la transformation du patron *Stratégie* [Gamma *et al.*, 95], utilisé comme exemple pour illustrer ces règles.

3.3 Transformation vers AspectJ

Nous présentons ici quelques règles de transformation[20] utiles pour produire un modèle instance du métamodèle spécifique à AspectJ à partir d'un modèle instance du métamodèle général Aspect/UML. Nous illustrons ces règles à travers l'exemple de transformation du modèle général du patron *Stratégie*.

3.3.1 Modèle général du patron *Stratégie*

La figure 4.32 montre la structure par aspects du patron *Stratégie* définie comme étant une instance du métamodèle général Aspect/UML.

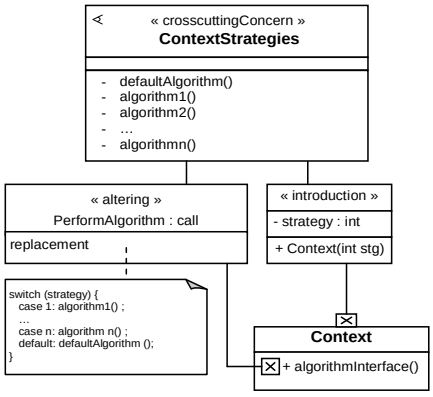


Figure 4.32. *Modèle général par aspects du patron* Stratégie

[20] Le reste des règles de transformation spécifiques à AspectJ sont définies dans l'annexe C.

Nous distinguons principalement une instance de la métaclasse CrosscuttingConcern (ContextStrategies) qui entrecroise une instance de la métaclasse Class (Context), par un élément d'altération instance de la métaclasse AlteringElement (replacement) qui est lié à une spécification d'entrecroisement instance de la métaclasse CrosscuttingElement (PerformAlgorithm) et deux introductions (instances de la métaclasse Introduction). Nous présentons ci-dessous les règles de transformation permettant de projeter ces éléments instances vers des éléments spécifiques à AspectJ.

3.3.2 Règles de transformation

Transformation des préoccupations transversales. Pour chaque instance de CrosscuttingConcern du modèle source, il faut créer une instance d'Aspect dans le modèle cible et appeler toutes les règles de transformation projetant l'ensemble des propriétés de la préoccupation transversale à transformer.

```
top relation CrosscuttingConcernToAspect
{
   ccn, ccp: string ;
   checkonly domain aspectModel cc : CrosscuttingConcern {namespace= p: Package { }, name = ccn,
                                                          isAbstract= 'true', isPrivileged = ccp };
   enforce domain aspectJModel a :Aspect {namespace= tp : Package { }, name = ccn,
                                          isAbstract = 'true', isPrivileged = ccp};
    when { PackageToPackage(p, tp);}
    where {
      AttributToAttribute(cc, a);
      OperationToOperation(cc, a);
      IntroductionToIntroduction(cc, a) ;
      ParentDeclarationToDeclareParents(cc, a);
      CrosscuttingSpecificationToMethodRelatedDesignator(cc, a); }
}
```

Transformation des introductions. La règle ci-dessous transformant les introductions d'une préoccupation transversales fait appel à deux règles : la première est spécifique aux introductions d'attributs, alors que la deuxième est dédiée à la transformation des introductions d'opérations.

```
relation IntroductionToIntroduction
{
   checkonly domain aspectModel cc:CrosscuttingConcern {};
   enforce domain aspectJModel a:Aspect {};
   where {
     AttributeIntroductionToAttributeIntroduction(cc, a,);
     OperationIntroductionToOperationIntroduction(cc, a,);
   }
}
```

- *Transformation des introductions d'attributs*. La règle suivante présente un exemple de transformation d'une introduction d'un attribut de type primitif dans exactement une classe cible.

```
relation AttributeIntroductionToAttributeIntroduction
//introduction d'attribute de type primitif dans une classe
{
   an, tn, cn, pn, av : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = i : Introduction
                                { feature = a : Attribute { name = an, visibility = av, type = t: Primitive {name = tn}},
                                  target = c: Class { namespace = p : Package {name = pn}, name = cn }
                                }};
   enforce domain aspectJModel a: Aspect
       { crosscuttingFeature = i : Introduction
                    { introducedMember = ta: Attribute { name = an, visibility = av, type = tt : Primitive {name = tn}},
                      targetType = tc : Class { namespace = p: Package {name =pn}, name = cn}
                    }};
   when {
       CrosscuttingConcernToAspect(cc, a); }
}
```

– *Transformation des introductions d'opérations.* La règle ci-dessous présente un exemple de transformation d'une introduction d'une opération non abstraite dans une classe cible.

```
relation OperationIntroductionToOperationIntroduction
//introduction d'operation non abstraite dans une classe
{
   os, ov, eb, pn, cn : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
      { crosscuttingElement = i : Introduction
                              { feature = o : Operation { specification = os,
                                                          isAbstract = 'false', visibility = ov,
                                                          method = m: Method
                                                                   {body = mb : ProcedureExpression {body = eb}}
                                                        },
                                target = c: Class {namespace = p: Package {name = pn}, name =cn }
                              } };
   enforce domain aspectJModel a: Aspect
      { crosscuttingFeature = ti : Introduction
                              { introducedMember = to : Operation { specification = os,
                                                                    isAbstract = 'false', visibility = ov,
                                                                    method = tm: Method
                                                                  {body = tmb : ProcedureExpression {body = eb} }
                                                                  },
                                targetType = tc : Class {namespace = p : Package {name = pn}, name =cn }
                              } };
    when {
       CrosscuttingConcernToAspect(cc, a);
    }
}
```

Transformation des spécifications d'entrecroisement. Toute spécification d'entrecroisement non abstraite est transformée en un point de recouvrement primitif nommé (non abstrait) de type MethodRelatedDesignator. La règle transformant une spécification d'entrecroisement fait appel automatiquement aux règles transformant les éléments d'altération liés à cette spécification. Voici ci-dessous un exemple de transformation d'une spécification non abstraite entrecroisant une opération.

```
relation CrosscuttingSpecificationToMethodRelatedDesignator
//transformation vers un point de recouvrement nommé non abstrait (target->size()=1)
{
   csn, csv, cst, os, pn, cn, nps, mrde : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
      { crosscuttingSpecification = cs: CrosscuttingSpecification
           { name = csn, visibility = csv, isAbstract = 'false', type = cst,
             target = o: Operation { specification = os,
                                     owner = c: Class {namespace = p: Package {name = pn}, name =cn } }
           }};
   enforce domain aspectJModel a: Aspect
      {crosscuttingSpecification = np : NamedPointcut
           { specification = nps, visibility = csv, isAbstract = 'false',
             implementation = mrd : MethodRelatedDesignator
                              { identifier = cst, expression = mrde,
                                behavioralFeature = to : Operation
                                   { specification = os,
                                     owner = tc: Class {namespace = tp: Package {name = pn}, name =cn } }
                              }
           }};
    when {
       CrosscuttingConcernToAspect(cc, a);
    }
    primitive domain pass:String;
    where{
        nps = csn+'()';
        mrde = cst+'('+cn+'.'+os);
        AlteringElementToAdvice(cc, cs, a, np) ;
    }
}
```

Transformation des éléments d'altération. Tout élément d'altération est transformé vers une instance de la classe Advice. Voici ci-dessous la relation correspondante.

```
relation AlteringElementToAdvice
//transformation d'un élément d'altération vers un advice
{
   aeb, aet, at : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
       { crosscuttingElement = ae : AlteringElement
            { specification = cs : CrosscuttingSpecification { },
              body = aee: AlteringElementExpression {body = aeb},
              type = aet }
       };
   enforce domain aspectJModel a: Aspect
       { crosscuttingFeature = a : Advice { attachedPointcut = np : NamedPointcut { },
                                            adviceType = at,
                                            body = ae: AdviceExpression {body = aeb} } };
    when {
        CrosscuttingSpecificationToMethodRelatedDesignator(cc, a);
    }
    where{
        at = AlteringElementTypeToAdviceType(aet);
    }
}

function AlteringElementTypeToAdviceType (type: String) : String {
      if (type='before') then 'before'
      else if (type='after') then 'after'
           else 'around'
           endif
      endif; }
```

3.3.3 Modèle AspectJ du patron *Stratégie*

La figure 4.33 montre la structure de *Stratégie* conforme au métamodèle spécifique à AspectJ, obtenue par transformation de celle présentée par la figure 4.32, en appliquant l'ensemble des règles de transformation explicitées ci-dessus.

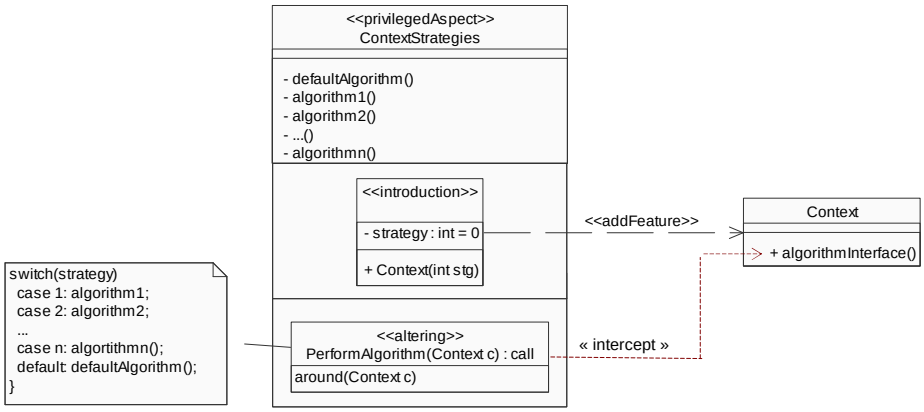


Figure 4.33. *Modèle de* Stratégie *spécifique à AspectJ*

Au-delà du fait que le CrosscuttingConcern est devenu un Aspect, l'AlteringElement un Advice (around(Context c)), on peut remarquer que la CrosscuttingSpecification est devenue un NamedPointcut (PerformAlgorithm(Context c) : call) dont l'implémentation est de type MethodRelatedDesignator (une spécialisation de Pointcut) du fait que la CrosscuttingSpecification soit de type call. On peut également noter le type de l'advice qui a été déterminé par une règle de transformation spécifiant qu'un Advice obtenu par transformation d'une AlteringElement de type replacement doit être de type around.

3.4 Transformation vers Hyper/J

Nous présentons dans ce qui suit quelques règles de transformation[21] utiles pour produire un modèle instance du métamodèle spécifique à Hyper/J à partir d'un modèle instance du métamodèle général Aspect/UML. Nous illustrons ces règles à travers l'exemple de transformation du modèle général du patron *Stratégie* vers un modèle instance d'HyperJ/UML.

3.4.1 Règles de transformation

Création de l'hyperspace, la dimension des préoccupations et l'hypermodule de composition. Toute transformation d'un modèle général vers un modèle spécifique à Hyper/J commence obligatoirement par la construction dans le modèle cible de l'espace (instance d'Hyperspace) et du contexte (instance d'Hypermodule) de composition, mais aussi de la dimension des préoccupations transversales et de base (instance de Dimension). Une règle de transformation correspondante est la suivante. Elle préconise de créer un hypermodule dont la stratégie de composition générale est overrideByName.

```
top relation HyperspaceDimensionAndHypermodule
{
    hn, hmn: String;
    checkonly domain aspectModel p: Package {ownedElement = cc: CrosscuttingConcern {}};
    enforce domain hyperJModel h: Hyperspace { name = hn,
                                               dimension = d: Dimension {name = 'Concern'}};
    enforce domain hyperJModel hm: Hypermodule { name = hmn,
                                                 compositionStrategy = 'overrideByName'};
}
```

Création de la préoccupation de base. Le premier hyperslice à créer dans le cas d'une transformation vers Hyper/J est celui encapsulant les classes et interfaces de base impactées par les préoccupations transversales. Pour ce faire, nous avons défini la relation BaseClassifierToKernelHyperslice qui fait appel à son tour aux règles transformant les classes et/ou interfaces mises en jeu tout en les reliant à l'hyperspace ainsi créé. Cette règle fait appel également aux règles transformant les relations qui existent entre ces classificateurs.

```
top relation BaseClassifierToKernelHyperslice
{
    pn: String;
    checkonly domain aspectModel p: Package {name = pn};
    enforce domain hyperJModel hm : Hypermodule
        { composableHyperslice = kh: Hyperslice { name = 'kernel',
                                                  composablePackage = tp: Package {name = pn},
                                                  owner = d: Dimension {owner = h: Hyperslice { }}
                                                };
    when{
        HyperspaceAndConcernDimension(hs, d, hm); }
    where{
        ClassToClass(p, tp, hs);
        InterfaceToInterface(p, tp, hs);
        //appels aux règles transformant les associations, généralisations, réalisations, etc.}
}

top relation ClassToClass
{
    pn, cn: String;
    ckeckonly domain aspectModel c: Class {namespace = p: Package {name = pn}, name = cn};
    enforce domain hyperJModel tc: Class { namespace = tp: Package {name = pn},
                                           name = cn,
                                           hyperspace = hs: Hyperspace {}} ;
```

[21] Le reste des règles de transformation spécifiques à Hyper/J sont définies dans l'annexe C.

```
    when{
        BaseClassifierToKernelHyperslice(p, tp, hs) }
    where{
       //appels aux règles transformant attributs et opération de classes }
}
```

Transformation des préoccupations transversales. Toute préoccupation transversale (instance de CrosscuttingConcern) est transformée vers un hyperslice et un paquetage lié à cet hyperslice. Le paquetage à créer doit contenir une nouvelle définition pour chaque classe entrecroisée par la préoccupation transversale considérée. Toute règle transformant une préoccupation transversale vers un hyperslice et un paquetage doit faire appel aux règles transformant les introductions, les déclarations de parenté et les éléments d'altération propres à cette préoccupation, comme le montre la règle ci-dessous.

```
top relation CrosscuttingConcernToHyperslice
{
    ccn, pn: String;
    checkonly domain aspectModel p: Package { ownedElement = cc : CrosscuttingConcern {name = ccn},
                                              name = pn };
    enforce domain hyperJModel ch: Hyperslice { name = ccn,
                                              composablePackage = tp: Package {name = pn},
                                              owner = d : Dimension {owner = hs: Hyperspace{ }},
                                              integrationContext = hm: Hypermodule { }
                                             };
    when{
       HyperspaceAndConcernDimension(hs, d, hm); }
    where{
       IntroductionToFeature(cc, tp) ;
       ParentDeclarationToRelationship(cc, tp);
       AlteringElementToOperation(cc, tp);   //utilise les spécifications d'entrecroisement }
}
```

Transformation des introductions. Comme dans le cas des transformations vers AspectJ, les règles transformant les introductions considèrent deux types de sous-règles spécifiques respectivement, aux transformations des introductions d'attributs et opérations. Nous présentons ci-dessous un exemple de règle transformant une introduction d'un attribut de type primitif dans une classe.

```
relation IntroductionToFeature
{
   checkonly domain aspectModel cc : CrosscuttingConcern { };
   enforce domain HyperJModel tp: Package { };
   where {
        AttributeIntroductionToAttribute(cc, tp);
        OperationIntroductionToOperation(cc, tp); }
}

relation AttributeIntroductionToAttribute
{   //introduction d'un attribute de type primitif dans une classe (target->size()=1)
   an, av, tn, pn, cn : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = i : Introduction
                                  { feature = a : Attribute { name = an, visibility = av, type = t: Primitive {name = tn}},
                                    target = c: Class { namespace = p: Package {name = pn}, name = cn }
                                  }};
   enforce domain hyperJModel tp: Package
        {  name = pn,
           ownedElement = tc: Class { name = cn,
                                      feature = ta: Attribute { name = an, visibility = av,
                                                                type = tt : Primitive {name = tn} } }
        }; // Si tc existe déjà ta est tout simplement ajouté à tc
   when {
       CrosscuttingConcernToHyperslice(cc, tp); }
 }
```

Transformation des éléments d'altération. Les éléments d'altération sont transformés vers des opérations. Remarquons que les spécifications d'entrecroisement n'ont pas de correspondants directs dans HyperJ/UML. Elles sont utilisées cependant par les règles transformant les éléments d'altération. Voici ci-dessous un exemple de règle transformant un élément d'altération dans un contexte d'intégration utilisant la stratégie de composition générale overrideByName.

```
relation AlteringElementToOperation
{
  // Transformation d'un élément d'altération vers une opération (ou plusieurs) dans la classe cible d'impact
  // par respect des propriétés de sa spécification (CrosscuttingElement)
  aet, aeb, cst, os, ov, eb, pn, cn, teb : String ;
  checkonly domain aspectModel cc: CrosscuttingConcern
     { crosscuttingElement = ae: AlteringElement
                              { type = aet,
                                body = aee: AlteringElementExpression {body = aeb},
                                specification = cs: CrosscuttingSpecification
                                                { isAbstract = 'false',
                                                  type = cst,
                                                  target = o: Operation
                                                           { specification = os, isAbstract= 'false', visibility = ov,
                                                             method = m: Method
                                                                      {body = mb: ProcedureExpression {body = eb}},
                                                             owner = c: Class
                                                                      { namespace = p: Package {name = pn},
                                                                        name =cn}
                                                           } }
                              } };
   enforce domain hyperJModel tp: Package
      { name = pn,
        ownedElement = tc: Class
                       { name = cn,
                         feature = to: Operation { specification = os, //même spécification que l'op impactée
                                                   isAbstract= 'false', visibility = ov,
                                                   method = tm: Method
                                                            {body = tmb: ProcedureExpression {body = teb}}
                                                 }
                       }
      };
   when {
       CrosscuttingConcernToHyperslice(cc, tp); }
    where{
        cst = 'call' || cst = 'set' || cst = 'get';
        teb = AlteringElementBodyToMethodBody(aet);
       OperationToOperation(cc, tp, ae, tc);  //ajout des opérations appelées par l'ae dans la cn impactée }
}

 OperationToOperation
 {
     sos, ov, eb : String
      checkonly domain aspectModel cc: CrosscuttingConcern
           { crosscuttingElement = ae: AlteringElement
                                   { supplier = so: Operation
                                                { owner = cc: CrosscuttingConcern {},
                                                  specification = sos, isAbstract = 'false', visibility = ov,
                                                  method = m: Method
                                                           {body = mb: ProcedureExpression {body = eb}}
                                                }
                                   } };
      enforce domain hyperJModel tp: Package
           { ownedElement = tc: Class
                            { feature = to: Operation { specification = sos,
                                                        isAbstract= 'false', visibility = ov,
                                                        method = tm: Method
                                                            {body = tmb: ProcedureExpression {body = eb}}
                                                      }
```

```
                                }};
        when{
              AlteringElementToOperation(cc, tp) ;
        }
}

function AlteringElementBodyToMethodBody(type: String) : String {
      if (type='before') then aeb+eb
      else if (type='after') then eb+aeb
            else aeb
           endif
       endif; }
```

3.4.2 Modèle HyperJ du patron *Stratégie*

La figure 4.34 montre la structure de *Stratégie* conforme au métamodèle spécifique à Hyper/J, obtenue par transformation de celle présentée par la figure 4.32, en appliquant l'ensemble des règles de transformation explicitées ci-dessus.

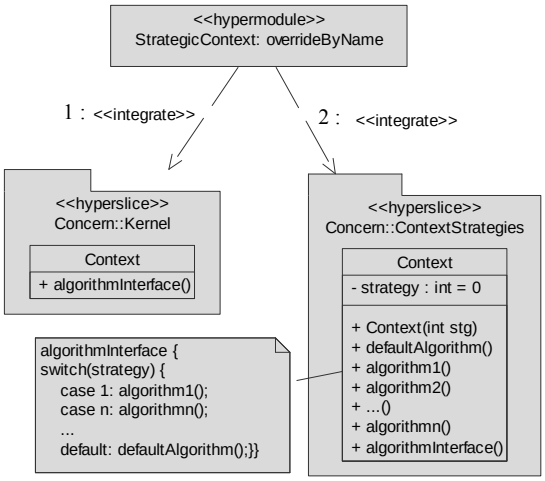


Figure 4.34. *Modèle de* Stratégie*, spécifique à Hyper/J*

L'instance ContextStrategies de CrosscuttingConcern est transformée en une instance d'Hyperslice appelée Concern::ContextStrategies, dont Concern est le nom de la dimension des préoccupations et ContextStrategies représente le nom de la préoccupation transversale. Cet hyperslice consiste en une nouvelle instance de Class (Context) qui détient l'attribut strategy et le constructeur Context(int stg), destinés à être introduits dans la classe Context référencée par l'hyperslice Concern::Kernel représentant la préoccupation de base. Par ailleurs, l'instance de AlteringElement est transformée en l'opération algorithmInterface() détenue aussi par la nouvelle classe Context. Le nom de cette opération est déterminé par le nom de l'opération liée à la spécification d'entrecroisement PerformAlgorithm (cf. figure 4.32). Notons également que la nouvelle classe Context détient les différentes définitions de l'algorithme considéré, qui sont appelées par l'élément d'altération ainsi transformé. Comparativement à la structure de *Stratégie* présentée par la figure 4.32, on peut essentiellement remarquer que peu d'informations liant les préoccupations de base et transversale apparaissent : la règle d'intégration générale overideByName détenue par l'hypermodule StrategicContext suffit dans cet exemple à définir les conditions d'entrecroisement des préoccupations. Cette règle suffit en effet pour retrouver les effets des différentes introductions, mais aussi celui d'une AlteringElement de type replacement dont le corps se retrouve dans cet exemple dans le corps de algorithmInterface() de la classe Context de l'hyperslice Concern::ContextStrategies.

4. Conclusion

Nous avons présenté et analysé dans ce chapitre les principaux travaux qui se sont intéressés à la modélisation des aspects. Ces travaux sont intéressants mais insuffisants, car leurs propositions présentent divers inconvénients ; elles sont souvent partielles et très spécifiques à une technique d'implémentation particulière ou à un domaine d'application particulier. Nous avons ainsi proposé une approche par métamodélisation et transformation de modèles permettant l'expression de conceptions par aspects, non spécifiques à un modèle ou un langage de programmation par aspects particulier. Nous avons basé cette approche sur un métamodèle général (Aspect/UML) dont les modèles instances peuvent être transformés en des modèles instances de métamodèles spécifiques à divers langages de programmation par aspects. Nous avons commencé par proposer deux extensions du métamodèle d'UML : AspectJ/UML et HyperJ/UML spécifiques respectivement à deux des langages de programmation par aspects les plus connus, AspectJ et Hyper/J. Nous avons basé ce travail de métamodélisation sur la version 1.5 d'UML dont la superstructure est identique à celle définie dans la version 1.4.2 d'UML [OMG-UML 04]. Nous avons pu rapprocher ensuite quelques-uns des éléments de modélisation principaux de ces deux métamodèles que nous avons retenus comme base à la définition d'un métamodèle général à l'approche Aspect. Il s'agit d'Aspect/UML, défini également comme étant une extension du métamodèle d'UML 1.5. Nous avons aussi proposé des notations graphiques pour les concepts de chaque métamodèle élaboré. À l'instar de l'approche IDM, un processus et des règles de transformations de modèles sont également proposés, complétant ainsi notre approche de modélisation par aspects. Nous avons montré au travers de l'exemple du patron *Stratégie* que l'application des règles de transformations que nous avons définies, permet de produire à partir d'un modèle instance du métamodèle général un modèle instance de l'un des deux métamodèles spécifiques. Il s'agit d'un travail d'étude de transformation de modèles s'appuyant sur les spécifications d'IDM concernant les techniques de transformation qui sont aujourd'hui en cours de standardisation.

Notre motivation principale dans ce travail était de trouver un moyen d'exprimer des structures de patrons par aspects indépendamment d'un langage de programmation par aspects en particulier, mais cette approche peut aussi bien être appliquée à toute modélisation par aspects. La définition du métamodèle général contribue également à la compréhension approfondie des concepts fondamentaux de l'approche Aspect, comme les métamodèles spécifiques améliorent la compréhension notamment de deux des modèles de programmation par aspects actuellement proposés (modèles symétriques et asymétriques). L'intégration dans notre approche d'autres langages de programmation par aspects que nous n'avons pas encore considérés, tels qu'AspectS (Hirschfeld, 2002), JAC (Pawlak, 2002), ou les filtres de composition (Aksit *et al.*, 1992), par exemple, pourrait éventuellement nous amener à raffiner le métamodèle général, mais devrait certainement permettre à la communauté Aspect de se rapprocher d'un consensus mieux établi sur les concepts fondamentaux de l'approche Aspect. Par ailleurs, nous avons poursuivi notre travail sur de nouvelles expressions des structures des patrons du GoF dans le langage général de modélisation par aspects que nous proposons (et qui est défini par le métamodèle Aspect/UML). Ce travail a confirmé de fortes analogies entre les modèles obtenus pour des patrons décrits comme « apparentés » dans [Gamma *et al.*, 95], et que nous avons déjà classés dans des groupes identiques dans le chapitre précédent (cf. § 3.3, chapitre 3). Ce travail nous a permis d'identifier un ensemble de patrons de conception par aspects plus petit mais, néanmoins, suffisant pour pouvoir appliquer l'ensemble des 23 patrons du GoF dans un contexte de développement par aspects. L'ingénierie de ces nouveaux patrons de conception par aspects constitue l'objectif du chapitre suivant.

Chapitre 5

Ingénierie de patrons de conception par aspects

Afin notamment de valider notre approche de modélisation et de transformation de modèles par aspects présentée dans le chapitre précédent, nous l'avons appliquée à l'ensemble des nouvelles solutions par aspects que nous avions proposées pour les patrons du GoF [Gamma *et al.*, 95]. L'expression en Aspect/UML de ces solutions a confirmé de fortes analogies entre les différents modèles par aspects obtenus pour ces patrons. Par abstraction de ces analogies, nous avons pu identifier et isoler un ensemble restreint de 8 nouveaux patrons de conception par aspects significatifs, à partir desquels les 23 patrons du GoF peuvent être dérivés. Pour une ingénierie complète de ces nouveaux patrons et leur réutilisation (comme énoncé dans la section 1.3.2 du chapitre 1), il convient de s'intéresser non seulement à leur identification, mais aussi à leur spécification. Nous proposons en ce sens, afin de décrire pertinemment les modèles de conception par aspects de ces patrons (à la manière du formalisme de description du GoF [Gamma *et al.*, 95]), d'adapter le formalisme P-Sigma [Conte *et al.*, 01a] défini dans le cadre des activités de recherche de notre équipe, pour obtenir AP-Sigma. Nous avons examiné ensuite les différents types de liens existant entre ces patrons, afin de les organiser et de pouvoir les réutiliser conjointement. Dans une optique de validation de ces nouveaux patrons, nous avons enfin considéré l'ensemble des relations existant entre ceux-ci et les conceptions par aspects que nous proposons pour les patrons du GoF d'une part, et de celles proposées par Hannemann [Web Hannemann] d'autre part. Cette dernière mise en relation avec les solutions de Hannemann nous a permis de plus d'identifier quelques solutions alternatives à certains des patrons de conception par aspects nouvellement introduits.

Nous proposons dans ce chapitre de présenter l'ensemble des 8 nouveaux patrons que nous avons isolés, ainsi que l'étude qui nous a permis de les identifier. La section 1 présente le formalisme de description que nous illustrons à travers l'exemple du patron *Stratégie* [Gamma *et al.*, 95]. La section 2 détaille l'ensemble des 8 nouveaux patrons. La section 3 présente la cartographie des relations existant entre ces patrons. Elle identifie aussi les relations existant entre ces derniers et les patrons du GoF dont ils sont issus, afin de montrer leur large utilisation et de les valider. La section 4 présente quelques travaux connexes et discute notre apport, alors que la section 5 conclut ce chapitre.

1. Formalisme de description de patrons par aspects : AP-Sigma

La réutilisation de patrons est d'autant plus efficace que ceux-ci sont associés à une spécification pertinente qui décrit l'ensemble des informations nécessaires et utiles à leur bonne compréhension et leur bonne utilisation (tels que les indications d'utilisation, les forces de la solution proposée, les conséquences d'utilisation et éventuellement les solutions alternatives, etc.). Notre objectif ici est donc de proposer un moyen nous permettant de décrire efficacement des patrons de conception par aspects.

Nous proposons pour ce faire, d'adapter le formalisme P-Sigma [Conte *et al.*, 01a] défini dans le cadre des activités de recherche de notre équipe [Web Sigma], pour définir AP-Sigma. P-Sigma est le résultat d'un travail ayant principalement pour objectif d'homogénéiser et d'extraire une sémantique commune et significative à divers formalismes de patrons déjà proposés dans la littérature, qui diffèrent principalement par le nombre et le degré de détail plus ou moins élevé des rubriques qu'ils proposent pour décrire les patrons. Dans le formalisme P-Sigma, chaque patron est décrit en trois parties : « Interface », « Réalisation » et « Relations », contenant chacune un ensemble de rubriques définies à l'aide d'un ou plusieurs champs formels ou textuels (cf. Annexe D). Ce formalisme original a été modifié pour renommer les parties, ainsi que les rubriques et leurs champs, supprimer des rubriques et des champs inutiles et adapter d'autres champs nécessaires par redéfinition de leur contenu. D'autres rubriques et champs supplémentaires ont été ajoutés. Nous présentons ci-dessous le formalisme AP-Sigma, et nous détaillons ses rubriques à travers l'exemple de la description du patron *Stratégie* avec une structure par aspects.

1.1 La partie *Interface*

L'interface d'un modèle est décrite en cinq rubriques, sa finalité est de faciliter la recherche et la sélection des patrons de conception. Le tableau 5.1 illustre ces rubriques qui sont détaillées dans ce qui suit.

- *Identifier* spécifie le nom du patron sous la forme d'un texte.
- *Classification* définit l'intention du patron à travers une collection de mots-clés, sous la forme d'une expression logique. Nous proposons que :
  - le premier mot-clé désigne la nature de la cible impactée : classe (*class*) ou objet (*object*) ;
  - le deuxième désigne le type d'impact : ajout de propriétés (*add*) ou modification de comportement (*alter*) ;
  - le troisième donne une précision supplémentaire lorsque le type d'impact est *alter*. En effet, tout comportement d'une classe (ou une instance de classe) peut être modifié de plusieurs façons : en lui ajoutant du code avant (*before*) ou après (*after*), ou avant et après (*combination*), en remplaçant son code (*replacement*), ou en conditionnant son exécution (*narrowing*) ;
  - le quatrième, également et uniquement nécessaire dans le cas d'une altération, précise le contexte dynamique d'impact : au moment de l'instanciation (*instantiation*), mettant donc en jeu le constructeur d'une classe, ou, au moment de l'exécution (*execution*) pour modifier le comportement d'une ou de plusieurs opérations de la classe considérée.

  Les règles de construction des expressions logiques de classification sont les suivantes.

  Expression ::= *class* ^ *add* | *class* ^ *alter* ^ AQ ^ DC | *object* ^ *alter* ^ AQ ^ DC
  AQ ::= aq | (AQ | aq) | ε ; tous les aq constituants un AQ sont différents
  DC ::= dc | (DC | dc) | ε ; tous les dc constituants un DC sont différents
  aq ::= *before* | *after* | *combination* | *replacement* | *narrowing*
  dc ::= *instantiation* | *execution*

- *Problem* détaille le problème considéré par le patron sous la forme d'un texte long.
- *Context* identifie les situations typiques d'application du patron dans un champ textuel *Applicability*. Il peut aussi spécifier, dans un champ formel *Pre-conditions,* les conditions devant être vérifiées avant l'application du patron.
- *Forces* est constitué de deux champs : *Forces* et *Qualifiers*. Le champ textuel *Forces* présente les motivations à résoudre le problème considéré par la solution proposée, et discute les contraintes à prendre en compte lors de l'application de cette solution. Le champ *Qualifiers* contient une expression logique basée exclusivement sur des conjonctions de critères de qualité (tels que *improve reusability*, *reusable code*, *no coupling*, *ease evolution*, etc.) attendus de l'application de la solution.

Tableau 5.1. *Interface du patron* Stratégie *dans sa version Aspect*

Identifier	*Stratégie*
Classification	object ^ alter ^ replacement ^ instantiation
Problem	Définit une famille d'algorithmes, les encapsule, les rend interchangeables, et leur permet d'évoluer indépendamment des clients.
Context	***Applicability***. On utilise le modèle *Stratégie* dans les cas suivants : - plusieurs instances ne diffèrent que par leur comportement. Définir et encapsuler les différentes variantes de ce comportement dans une préoccupation transversale donne le moyen d'appareiller une instance avec une variante idoine parmi plusieurs autres ;

	- on a besoin pour une classe de diverses variantes d'un algorithme qui peut être défini sur une ou plusieurs opérations. Par exemple, on peut définir des algorithmes représentant différents compromis encombrement mémoire / temps d'exécution. *Stratégie* peut être utilisé pour encapsuler ces variantes ; - une classe définit de nombreux comportements, qui figurent dans ses opérations sous la forme de déclarations conditionnelles multiples. *Stratégie* peut être utilisé pour séparer ces variantes tout en les plaçant dans une préoccupation transversale ; - un algorithme utilise des données que les clients n'ont pas à connaître. Utiliser le patron *Stratégie* dispense d'avoir à révéler des structures complexes de données spécifiques des algorithmes.
Forces	***Forces.*** Le patron *Stratégie* possède les avantages et inconvénients suivants : - les différentes variantes de l'algorithme sont encapsulées et séparées dans la préoccupation transversale ; elles sont faciles à comprendre, à adapter et à faire évoluer ; - les définitions des classes impactées sont inchangées, celles-ci se trouvent faciles à appréhender, à maintenir et à faire évoluer. Elles peuvent être d'ailleurs directement réutilisées dans plusieurs autres contextes d'application ; - *Stratégie* présente une solution alternative à l'utilisation du polymorphisme d'inclusion ; - *Stratégie* dispense de l'utilisation de déclarations conditionnelles ; - *Stratégie* a un inconvénient potentiel, du fait que les clients doivent être informés des différentes stratégies disponibles, et doivent avoir compris en quoi les diverses stratégies diffèrent avant de pouvoir choisir la plus appropriée. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability

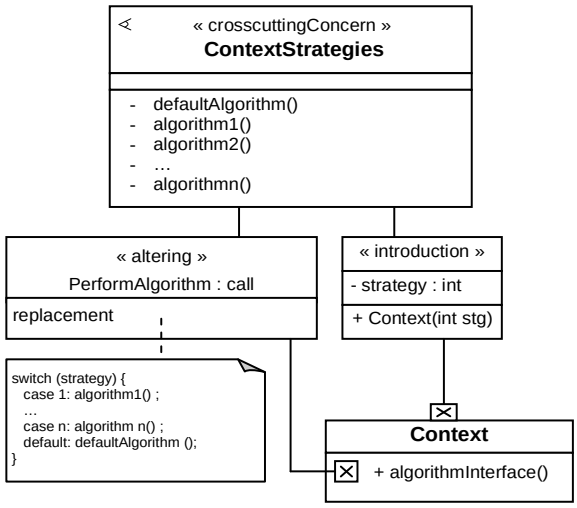
Outre le renommage de l'ensemble des rubriques et leurs champs, les principales adaptations apportées à la partie *Interface* du formalisme P-Sigma pour définir celle de AP-Sigma, concernent le contenu du champ de la rubrique *Classification*, d'une part, et le contenu du champ *Qualifiers* de la rubrique *Forces*, d'autre part. Il ne s'agit pas d'une modification de la signification ou de la nature de ces deux champs, mais de la re-définition des deux ensembles des mots-clés utilisés pour la spécification des expressions logiques constituant ces deux champs. Il est à noter aussi que la rubrique *Classification* contient un seul champ par rapport à sa rubrique correspondante dans P-Sigma, qui contient de plus le champ « classification textuelle » inutile dans notre cas. Les autres rubriques et leurs champs sont inchangés car indépendants d'une technique de programmation particulière et des domaines d'application des patrons.

1.2 La partie *Realization*

La réalisation d'un patron est définie sur quatre rubriques, spécifiant l'ensemble des éléments qui permettent de résoudre le problème soulevé par le patron. Le tableau 5.2 montre, à titre d'exemple, la rubrique *Solution* du patron *Stratégie*. Nous détaillons dans ce qui suit l'ensemble de ces quatre rubriques.

- *Solution* décrit la solution en détail sur deux champs : un champ textuel *Solution* et un champ formel *Class/Aspect Diagram* contenant un diagramme de classes et aspects exprimé dans la syntaxe concrète d'Aspect/UML (cf. § 2.4, chapitre précédent).
- *Alternatives* est une rubrique optionnelle, qui indique les solutions alternatives et donne les implémentations possibles typiques pour un langage particulier.
- *Application case* présente un exemple d'imitation du patron. Cette rubrique optionnelle est fortement conseillée pour faciliter la compréhension de la solution du patron.
- *Consequences* discute les conséquences induites par l'imitation du patron. Elle peut identifier de plus de nouveaux problèmes conséquents qui nécessitent l'application d'autres patrons.

Tableau 5.2. *Solution par aspects du patron* Stratégie

Solution	***Solution.*** Déclarer dans une préoccupation transversale (*crosscutting concern*) ContextStrategies : - une introduction dans Context[22] d'un attribut strategy dénotant la stratégie interne d'une instance particulière de Context, - une introduction d'un nouveau constructeur paramétré permettant l'initialisation de cet attribut, - les différentes variantes d'algorithmes, - une spécification d'entrecroisement (*crosscutting specification*) et son élément d'altération (*altering element*) correspondant, permettant de remplacer interface() par la variante idoine. ***Class/Aspect Diagram***  **Figure 5.1.** *Structure par aspects de* Stratégie

Concernant cette partie *Realization*, nous avons également renommé toutes les rubriques de P-Sigma ainsi que leurs champs que nous avons retenus. Nous avons par ailleurs supprimé la rubrique « solution démarche » parce que les patrons que nous proposons sont de type produit. Nous avons de plus allégé la rubrique *Solution* tout en supprimant le champ « solution produit formelle séquences », car nous nous intéressons exclusivement à la description statique des solutions des patrons proposés. Pour cette même raison, nous avons également supprimé le champ « cas d'application séquence » de la rubrique « cas d'application ». Enfin, nous avons rajouté une nouvelle rubrique *Alternatives* nous permettant notamment de présenter et de discuter les solutions alternatives à la solution du patron considéré si elles existent.

1.3 La partie *Relations*

La partie *Relations* d'un patron est définie sur une seule rubrique nommée *Relations*. Celle-ci est constituée de plusieurs champs textuels optionnels, correspondant aux différents types de relations existant entre le patron considéré et d'autres patrons de la collection à laquelle appartient ce dernier. En effet, un patron peut constituer une solution alternative à un autre problème soulevé par un autre patron ; il peut aussi raffiner (resp. étendre) ce dernier pour répondre à un problème plus spécifique (resp. plus

[22] La classe Context est la classe destinée à être altérée par la préoccupation transversale ContextStrategies. Dans le même sens, pour le reste des structures par aspects des 8 nouveaux patrons que nous présentons dans ce chapitre, les classes Context et Contextk (k=1..n) représentent en général les classes cibles d'entrecroisements.

général). Un patron peut également être utilisé par un ou plusieurs autres patrons ; il peut lui-même utiliser d'autres patrons pour sa solution. Toutes ces relations (*uses*, *refines*, *extends* et *alternativeOf*) sont définies dans ce qui suit ; elles permettent de positionner le problème et la solution du patron par rapport à d'autres problèmes et solutions existantes. *Stratégie* avec sa structure par aspects utilise, par exemple, le patron *Class Polymorphic Behaviour* (cf. tableau 5.6).

- *Uses*. Un patron P1 utilise un patron P2, si une partie ou tous les problèmes posés par P1 peuvent être résolus en partie ou complètement par P2.
- *Refines/Extends*. Il s'agit de deux relations réciproques. La relation *refines* tend à restreindre le problème d'un patron : un patron P1 raffine un patron P2, si le problème posé par P1 est une spécialisation de celui posé par P2. P2 peut résoudre les problèmes de P1. La force et le contexte de P1 sont enrichis par rapport à ceux de P2. La relation *extends* tend, par contre, à généraliser le problème d'un patron. On dit alors que P2 étend P1.
- *AlternativeOf*. Un patron P1 est une alternative d'un patron P2, si P1 possède le même problème que P2 mais propose une solution différente. Les deux patrons ont le même contexte, la même classification mais des forces différentes.

Il convient de noter ici que nous n'avons pas ajouté de nouveaux types de relations par rapport à ce qui a été déjà défini dans P-Sigma. L'unique adaptation concernant la partie *Relations* de P-Sigma porte plutôt sur la restructuration des différentes rubriques et champs constituant cette partie. Nous avons choisi en effet de définir une seule rubrique principale contenant plusieurs champs dont chacun est spécifique à un type de relations particulières, au lieu de définir une rubrique par type de relations. Il s'agit d'une simplification profitable qui ne contrarie pas l'un des principaux objectifs de P-Sigma : accélérer et faciliter la phase de sélection des patrons. En effet, étant toujours séparés et isolés dans des champs différents, les ensembles des patrons liés au patron considéré par une relation particulière sont aussi rapidement et facilement identifiables.

2. Identification et spécification de 8 nouveaux patrons de conception par aspects

En exprimant les solutions par aspects que nous avions proposées pour les 23 patrons[23] de conception du GoF, conformément à notre métamodèle général Aspect/UML (cf. § 2.4, chapitre précédent), nous avons constaté qu'il existe de fortes analogies entre les modèles obtenus pour différents groupes de patrons comme le montre d'ailleurs les résultats du chapitre 3 (cf. § 3.3). Nous avons ainsi comparé l'ensemble des modèles de chacun des groupes, afin d'isoler et homogénéiser les points communs de ces modèles et d'extraire par abstraction une sémantique commune et significative à ceux-ci. Ce travail nous a permis d'identifier de nouveaux patrons de conception par aspects.

Nous détaillons dans ce qui suit l'ensemble des 8 nouveaux patrons que nous avons isolés, tout en essayant de les mettre en relation. Pour chacun de ces patrons, nous expliquons tout d'abord comment nous l'avons identifié par abstraction des points communs de certaines structures par aspects des patrons du GoF. Nous donnons ensuite sa description tout en utilisant le formalisme AP-Sigma. Les relations identifiées pour chacun de ces patrons sont enfin explicitées dans la rubrique *Relations* de sa description.

[23] La solution par aspects du patron *Façade* est similaire à sa solution par objets ; elle ne présente aucun point commun avec le reste des solutions par aspects des 23 patrons du GoF.

2.1 Le patron Add Features

Le patron *Add Features* est le résultat de l'abstraction des analogies relevées dans les modèles par aspects des trois patrons du GoF : *Visiteur*, *Adaptateur* et *Interpréteur*. Les figures 5.2, 5.3 et 5.4 montrent, respectivement, les structures par aspects de ces trois patrons. Ces patrons proposent, indépendamment de leurs intentions spécifiques, d'étendre une ou plusieurs classes existantes (Element et ses classes dérivées, Adaptee, Expression et ses classes dérivées, Context, selon le patron considéré) tout en leur ajoutant, de manière statique, une ou plusieurs propriétés structurelles et/ou comportementales. Leurs solutions par aspects se basent pour ce faire sur l'utilisation du concept d'*introduction* (cf. § 2.4.2, chapitre précédent) d'Aspect/UML.

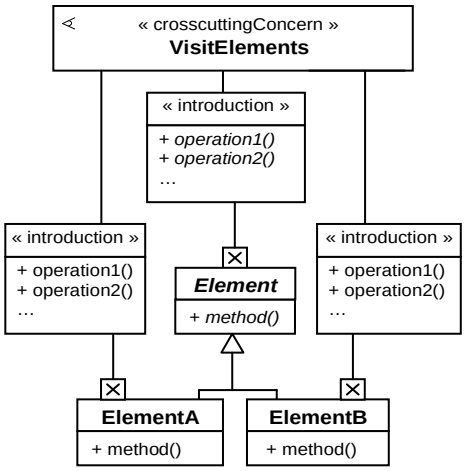


Figure 5.2.*Structure par aspects de* Visiteur

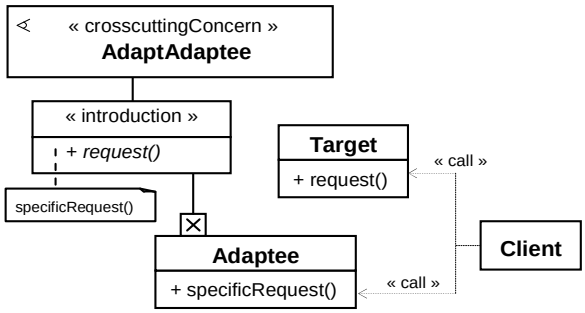


Figure 5.3. *Structure par aspects d'*Adaptateur

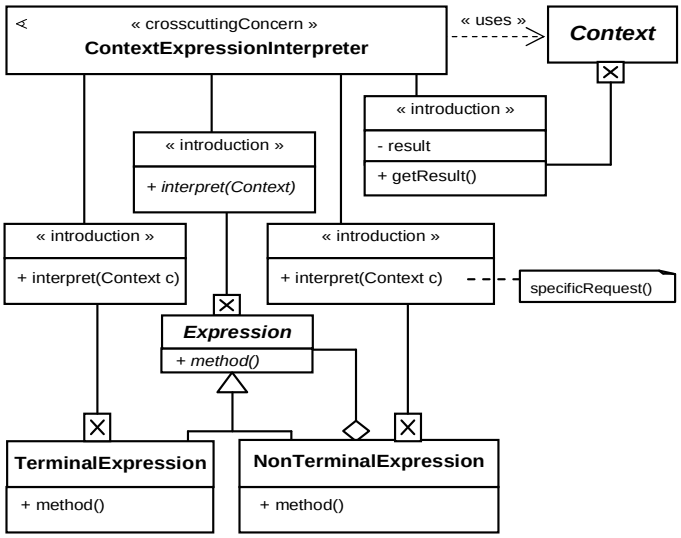
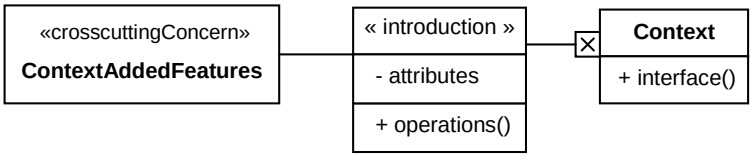


Figure 5.4. *Structure par aspects d'*Interpréteur

Par abstraction des analogies existantes entre ces trois structures par aspects nous proposons le patron *Add Features*, dont la description est donnée dans ce qui suit (cf. tableau 5.3). Il convient de noter ici que, issu notamment des patrons *Visiteur*, *Adaptateur* et *Interpréteur* (traitant exclusivement du même problème que ce nouveau patron de conception par aspects), *Add Features* est également utilisé par la plupart des patrons du GoF pour répondre en partie à leurs problèmes spécifiques.

Tableau 5.3. *Description du patron* Add Features

Identifier	*Add Features*
Classification	class ^ add
Problem	Permet d'ajouter de nouvelles propriétés à une classe concrète ou abstraite (et à ses sous-classes, si nécessaire), devant participer à la réalisation d'une nouvelle préoccupation hors sa préoccupation de base. La définition de la classe mise en jeu doit rester inchangée.
Context	***Applicability*** On utilise *Add Features* dans le cas où on a besoin d'ajouter à une classe Context une ou plusieurs propriétés (attributs et/ou opérations), sans modifier sa définition de base. Ces propriétés sont requises par cette classe, afin qu'elle puisse intervenir dans la réalisation d'une nouvelle préoccupation différente de sa préoccupation de base.
Forces	***Forces*** Les avantages de *Add Features* sont les suivants. - Les nouvelles propriétés augmentant la classe Context sont découplées de celle-ci. La définition de Context est donc inchangée, cette dernière ne traite que de sa préoccupation de base. Elle se trouve ainsi facile à appréhender, à maintenir et à faire évoluer. Elle peut d'ailleurs être directement réutilisable dans plusieurs autres contextes d'application. - Tout le code spécifique à la nouvelle préoccupation transversale est séparé et encapsulé dans une seule unité modulaire, il est plus facile à appréhender, à maintenir, et à faire évoluer indépendamment de la classe Context. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Définir une préoccupation transversale (*crosscutting concern*) regroupant toutes les définitions d'introductions dans Context des attributs et opérations requis. - Dans le cas où les opérations destinées à être ajoutées dans Context doivent utiliser une ou plusieurs propriétés (attributs et opérations) privées de celle-ci, la préoccupation transversale doit être définie comme privilégiée, mais pas nécessairement dans le cas contraire. ***Class/Aspect Diagram*** 
Consequences	Les points suivant sont à prendre en compte lors de l'application de *Add Features* sur une classe contexte. - Dans le cas où les attributs destinés à être ajoutés à Context sont privés, et que celle-ci possède une ou plusieurs sous-classes, il convient de vérifier si ces dernières ont besoin de manipuler directement ces attributs en les héritant. Si besoin est, il faut les déclarer comme étant protégés. - D'ailleurs, si les attributs privés nouvellement ajoutés doivent avoir des accesseurs (en lecture et/ou écriture), il convient également d'introduire ces accesseurs dans Context. - L'introduction d'une opération abstraite, dans une classe abstraite Context, nécessite l'introduction des méthodes correspondantes dans les sous-classes concrètes de celle-ci. - Dans le cas d'une introduction d'une nouvelle opération dans une classe Context (concrète ou abstraite), qui possède une ou plusieurs sous-classes, il convient de vérifier s'il est nécessaire de redéfinir cette opération dans les classes dérivées. - Si les opérations destinées à être introduites dans la classe Context sont privées, et que cette dernière admet des classes dérivées, il convient de vérifier si ces sous-classes doivent hériter ces opérations. Si c'est le cas, il faut les déclarer comme étant protégées.

2.2 Le patron Alter Behaviours

Alter Behaviours est le patron résultat de l'abstraction des trois patrons *Singleton*, *Procuration* et *Décorateur*, dont les structures par aspects sont données respectivement par les figures 5.5, 5.6 et 5.7. Indépendamment de leurs intentions spécifiques, ces trois patrons proposent d'altérer de différentes façons (*replacement*, *before*, *after*… selon le patron considéré) et à différents moments (*instantiation* ou *execution*, selon le patron considéré) le comportement d'une classe Context déjà existante. Pour ce faire, ils utilisent dans leurs solutions les concepts Aspect *crosscutting specification* (cf. § 2.4.2 du chapitre 4) (PerformConstructor, PerformRequest et PerformMethod) et *altering element* (cf. § 2.4.2, chapitre précédent) d'Aspect/UML, qui permettent de modifier dynamiquement les comportements des opérations d'une classe donnée.

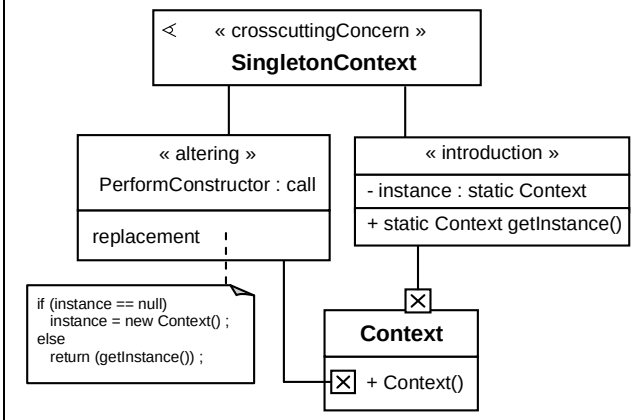


Figure 5.5. *Structure par aspects de* Singleton

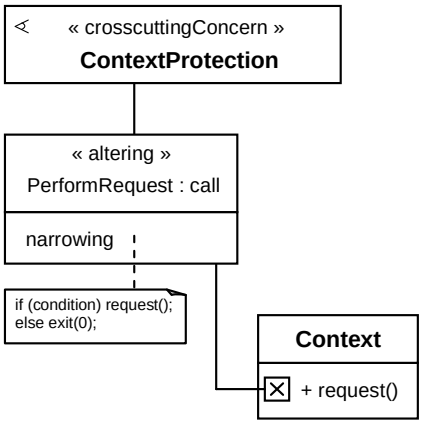


Figure 5.6. *Structure par aspects de* Procuration

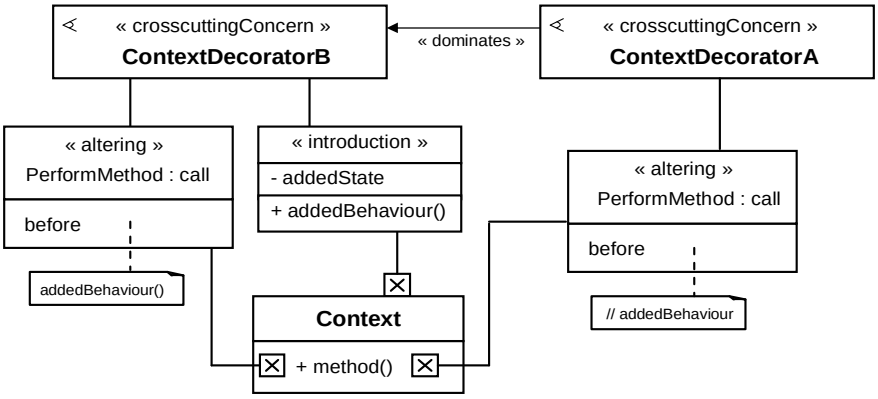
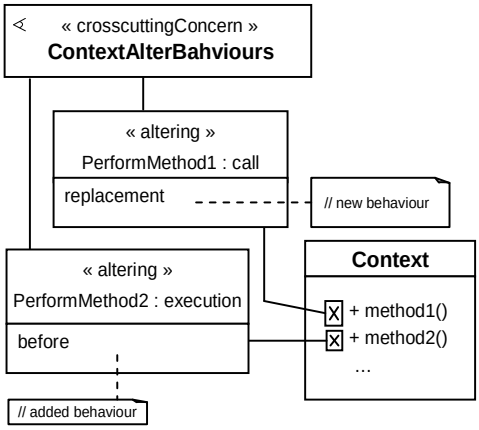


Figure 5.7. *Structure par aspects de* Décorateur

Par abstraction des analogies existantes entre ces trois structures par aspects nous proposons le patron *Alter Behaviours*, dont la description est donnée par le tableau 5.4. Il convient de noter ici que ce nouveau patron est largement utilisé par la plupart des structures par aspects des patrons du GoF, bien qu'issu principalement des patrons *Singleton*, *Procuration* et *Décorateur* qui traitent exclusivement du même problème que celui-ci.

Tableau 5.4. *Description du patron* Alter Behaviours

Identifier	*Alter Behaviours*
Classification	class ^ alter ^ (before | after | replacement | narrowing) ^ (instantiation | execution)
Problem	Permet de modifier le comportement d'une classe devant intervenir dans la réalisation d'une nouvelle préoccupation, hors sa préoccupation de base, sans changer sa définition. Ce comportement peut être défini par une ou plusieurs opérations.
Context	***Applicability*** On utilise *Alter Behaviours* dans le cas où on a besoin pour une classe Context de changer son comportement, afin qu'elle puisse être utilisée dans la réalisation d'une nouvelle préoccupation transversale différente de sa préoccupation de base. La classe Context doit prendre en compte d'autres besoins fonctionnels, sans pour autant que sa définition soit changée. Ces besoins peuvent concerner une ou plusieurs opérations de la classe Context. Selon les nouveaux besoins requis, *Alter Behaviours* peut être utilisé pour : - attacher de nouvelles responsabilités à la classe Context. Les opérations de Context, destinées à être altérées, peuvent être augmentées avant et/ou après leurs exécutions ; - remplacer le comportement des instances de Context. Les définitions des opérations destinées à être altérés sont complètement remplacées par de nouvelles implémentations ; - décider de l'exécution ou non des opérations destinées à être altérées, selon une condition particulière.
Forces	***Forces.*** Les avantages de *Alter Behaviours* sont les suivants. - La définition de la classe impactée est inchangée, elle ne traite que de sa préoccupation de base. La classe Context se trouve ainsi facile à appréhender, à maintenir et à faire évoluer. D'ailleurs, elle est directement réutilisable dans plusieurs autres contextes d'application. - *Alter Behaviours* offre la possibilité d'altérer le comportement de Context sans utiliser le polymorphisme d'inclusion, qui impose de créer de nouvelles sous-classes pour les nouveaux besoins fonctionnels. Ceci engendre de nombreuses classes et accroît la complexité d'un système. Il impose d'ailleurs de mélanger le code relatif aux nouveaux besoins avec celui relatif aux sous-classes. - Tout le code spécifique à la nouvelle préoccupation transversale est séparé et encapsulé dans une seule unité modulaire, il est plus facile à appréhender, à maintenir, et à faire évoluer indépendamment de Context. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Identifier la ou les opérations de Context destinées à être altérées. - Définir une préoccupation transversale (*crosscutting concern*). - Pour chacune des opérations considérées, définir dans la préoccupation transversale, la spécification d'entrecroisement (*crosscutting specification*) et son élément d'altération (*altering element*) correspondant permettant d'altérer le comportement de cette opération. - Les éléments d'altération peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privées de la classe Context. La préoccupation transversale doit être donc définie en tant que privilégiée, mais pas dans le cas contraire.

	Class/Aspect Diagram 
Consequences	Pour certains besoins spécifiques à la préoccupation transversale, il convient d'ajouter de nouveaux attributs et/ou opérations primitives dans la classe Context. Ces propriétés peuvent être ajoutées à l'aide du patron *Add Features*.
Relations	***Uses*** - *Add Features*. Il peut être utilisé pour ajouter, si nécessaire, de nouvelles propriétés à la classe Context.

2.3 Le patron Add New Role

Add New Role est le résultat de l'abstraction des analogies existantes notamment entre les structures par aspects des patrons *Prototype* et *Composite* (cf. figures 5.8 et 5.9). Ces deux patrons proposent d'ajouter dans un but particulier (à travers une ou plusieurs classes « Rôle » communes : Prototype, ConcreteComposite ou Leaf selon le patron considéré) des propriétés communes à plusieurs classes existantes destinées à être utilisées et apparentées pour la réalisation d'une nouvelle préoccupation, sans modifier leurs définitions de base et leurs comportements originaux. Nous remarquons que ce nouveau patron est largement utilisé par l'ensemble des structures par aspects des patrons du GoF.

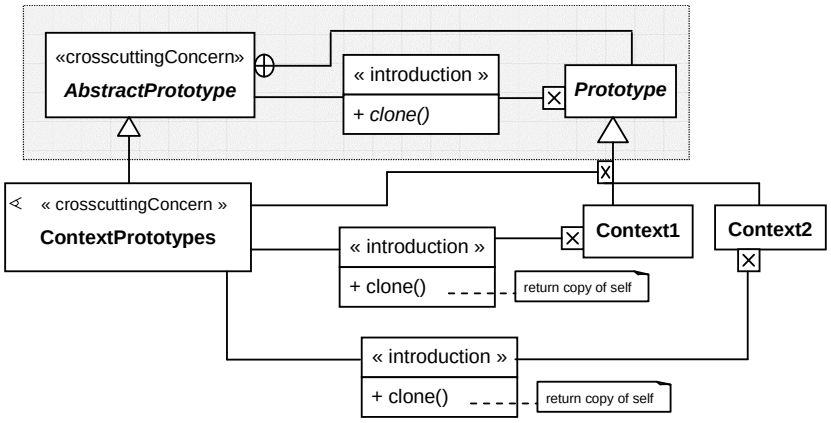
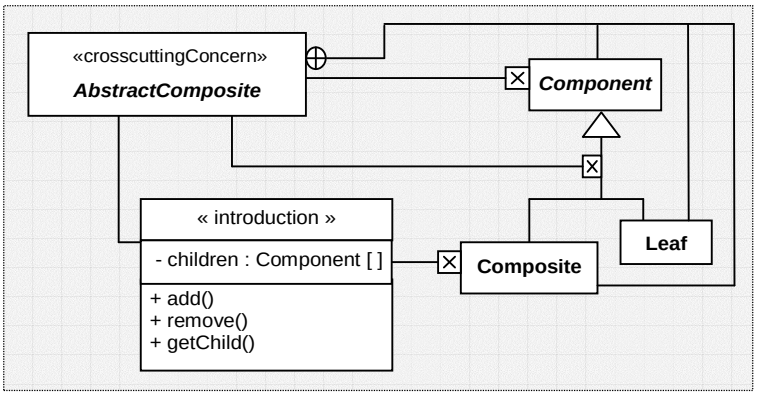
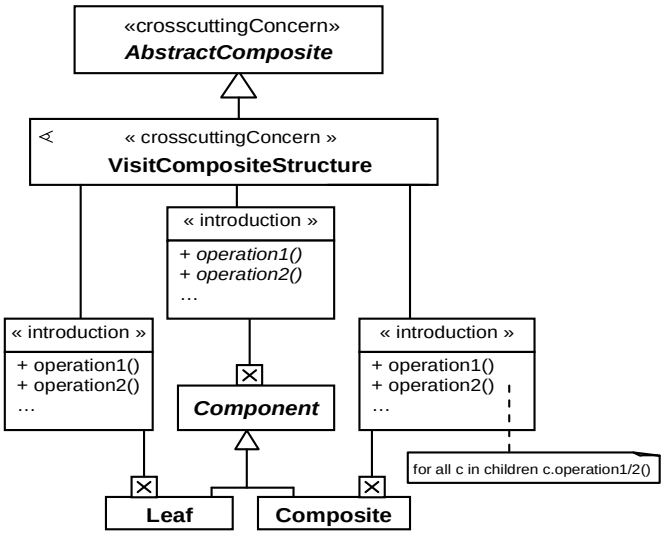


Figure 5.8. *Structure par aspects de* Prototype

Définition des super-classes communes participant dans le patron *Composite*.



Ajout aux classes communes d'opérations spécifiques au contexte d'application (utilisation de *Add Features*).



À la manière de *Prototype*, les classes concrètes jouant les rôles de Leaf et Composite doivent hériter de celles-ci. Ces relations de généralisation sont spécifiées dans un troisième CrosscuttingConcern qui hérite de CompositeStructure, et qui redéfinit et introduit –si nécessaire– les opérations operation1/2/..() dans les classes Leaf concrètes. La figure ci-dessous montre un exemple d'imitation de composite par aspects.

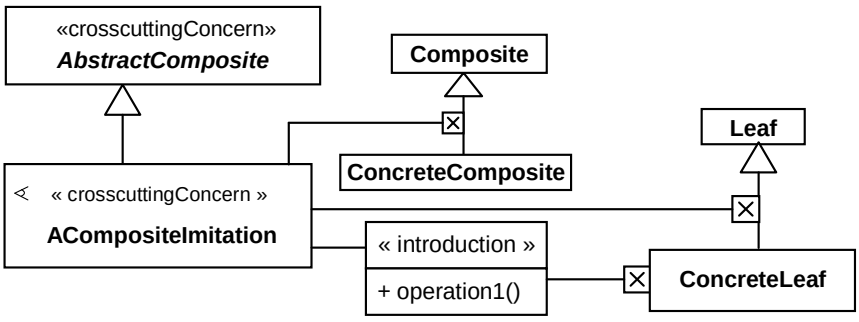


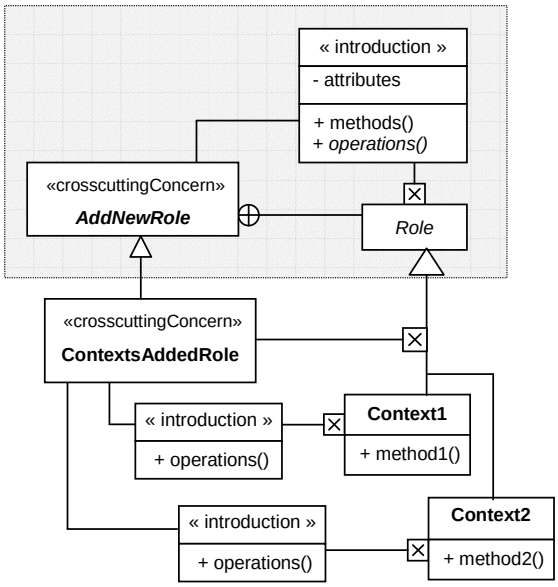
Figure 5.9. *Structure par aspects de* Composite

Par abstraction des analogies existantes entre ces deux structures nous avons identifié le patron *Add New Role*, dont la description est donnée par le tableau 5.5.

Tableau 5.5. *Description du patron* Add New Role

Identifier	*Add New Role*
Classification	class ^ add
Problem	Permet d'apparenter[24] différentes classes indépendantes (ou déjà apparentées) tout en leur ajoutant un nouveau rôle, afin qu'elles puissent participer à la réalisation d'une nouvelle préoccupation (différente de leurs préoccupations de base). *Add New Role* permet donc de définir et d'encapsuler les propriétés (attributs et opérations) communes et invariantes à ces classes dans une super-classe abstraite commune, qui représente le nouveau rôle. Il permet également d'appareiller chacune de ces classes impactées avec un comportement idoine (défini de manière abstraite dans la super-classe commune).
Context	***Applicability*** On utilise *Add New Role* dans les cas où on a besoin : - d'utiliser différentes classes indépendantes dans la réalisation d'une nouvelle préoccupation transversale, différente de leurs préoccupations de base, sans changer leurs définitions de base. Ces classes doivent jouer un même rôle ; elles peuvent cependant différer par leur comportement dans le cadre de ce nouveau rôle. *Add New Role* permet d'apparenter ces classes et de les appareiller avec les comportements idoines. - d'ajouter de nouvelles fonctionnalités à une classe Context (et ses classes dérivées, si elles existent) sans modifier sa définition, tout en lui ajoutant une nouvelle super-classe représentant un nouveau rôle. Context se voit ainsi avoir un type supplémentaire afin de participer dans la réalisation d'une nouvelle préoccupation (différente de sa préoccupation de base).
Forces	***Forces*** Les avantages de *Add New Role* sont les suivants : - les définitions des classes impactées sont inchangées, elles ne traitent que de leurs préoccupations de base. Ces classes se trouvent faciles à appréhender, à maintenir et à faire évoluer. Elles peuvent d'ailleurs être directement réutilisées dans plusieurs autres contextes d'application, du moment où elles sont indépendantes du nouveau rôle (la nouvelle super-classe) ; - tout le code spécifique à la nouvelle préoccupation transversale est séparé et encapsulé dans une seule unité modulaire, il est plus facile à appréhender, à maintenir, et à faire évoluer indépendamment des classes impactées. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Définir une préoccupation transversale abstraite AddNewRole regroupant : ▪ la définition d'une classe interne et abstraite Role, ▪ toutes les définitions d'introductions dans Role des propriétés (attributs et opérations) communes aux classes impactées (Contextk, k=1..n). Le comportement destiné à être polymorphe est défini comme étant abstrait, il peut être défini par une ou plusieurs opérations abstraites, - Définir une préoccupation transversale concrète ContextsAddedRole regroupant : ▪ toutes les définitions des déclarations d'héritage (parent declarations) permettant d'apparenter les classes impactées, ▪ les définitions d'introductions dans les classes impactées des diverses variantes spécifiques concrétisant les opérations abstraites de Role.

[24] Apparenter un ensemble de classes consiste à ajouter à ces dernières une super-classe commune.

	- Les opérations destinées à être ajoutées dans les classes impactées peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privées de celles-ci. La préoccupation transversale doit être dans ce cas définie comme privilégiée, mais pas nécessairement dans le cas contraire. ***Class/Aspect Diagram*** 
Consequences	Le patron *Add New Role* possède les conséquences suivantes. - Si une des classes impactées possède des sous-classes qui ont besoin de manipuler directement toutes les propriétés qu'elles héritent de Role et qui sont destinées à être privées, il convient de déclarer ces propriétés comme étant protégées. - Les sous-classes des classes impactées, si elles existent, doivent également être augmentées des variantes idoines des opérations abstraites de Role. Il convient également de vérifier s'il est nécessaire qu'elles redéfinissent les opérations concrètes de Role. Ces points peuvent êtres traités en appliquant convenablement le patron *Add Features*.
Relations	***Uses*** - *Add Features*. Il est utilisé pour ajouter les propriétés de Role, concrétiser ses opérations abstraites, et introduire leurs diverses variantes dans les classes impactées (et leurs sous-classes, si elles existent).

2.4 Le patron Class Polymorphic Behaviour

Comme le montrent les figures 5.10, 5.11 et 5.12, il existe une forte analogie entre les structures par aspects de *Fabrication, Patron de méthode* et *Fabrique abstraite*. Cette analogie concerne également le patron *Stratégie* dont la structure par aspects est donnée par la figure 5.1 (page 163).

Indépendamment de leurs intentions spécifiques, ces quatre patrons proposent tous de donner un comportement polymorphe aux instances d'une classe contexte (Context, ou Creator… suivant le patron considéré). Ce comportement peut être défini par une ou plusieurs opérations. Les variantes de ces opérations sont regroupées dans un *crosscutting concern*. Un attribut est utilisé pour la représentation interne de la variante idoine associée à chacune des instances de la classe contexte au moment de sa création ; il est ajouté à la classe impactée par une *introduction*, de même que la définition d'un nouveau constructeur destiné à initialiser l'attribut introduit.

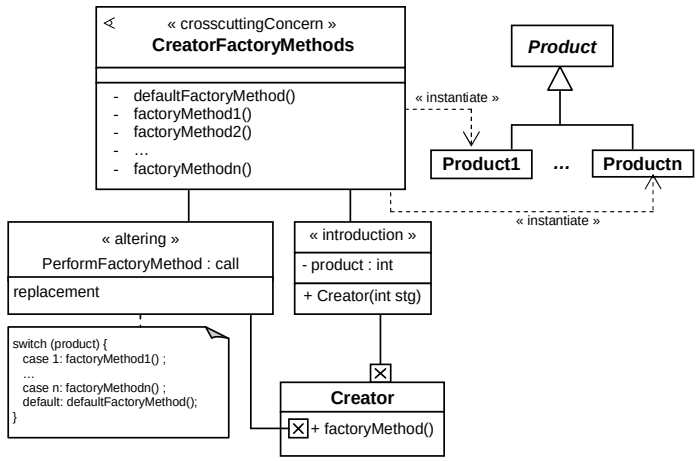


Figure 5.10. *Structure par aspects de* Fabrication

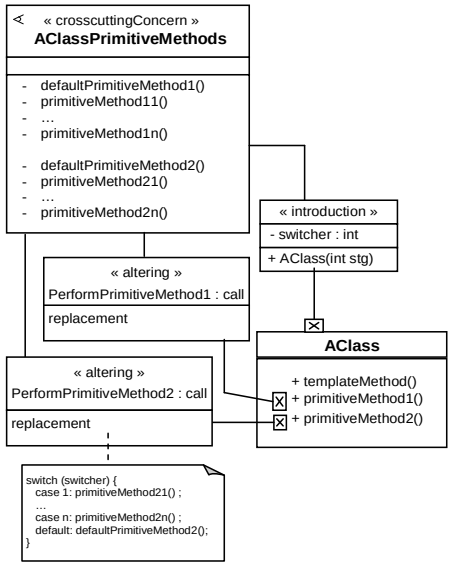


Figure 5.11. *Structure par aspects de* Patron de méthode

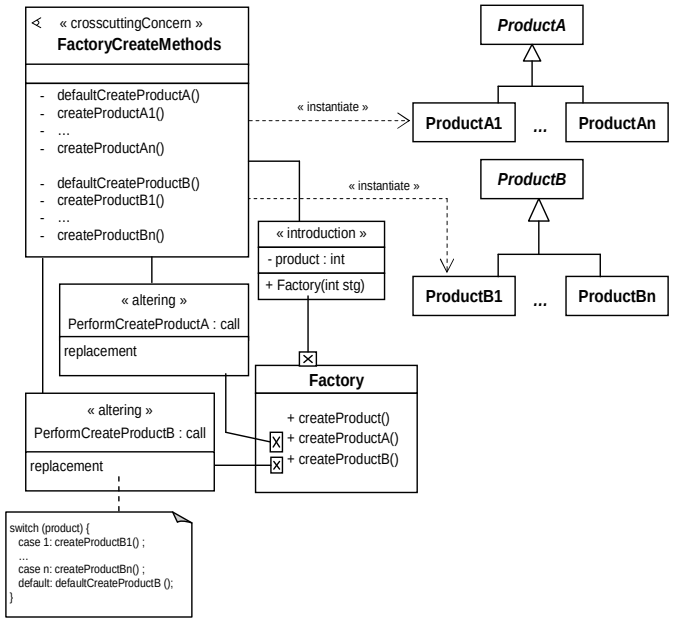


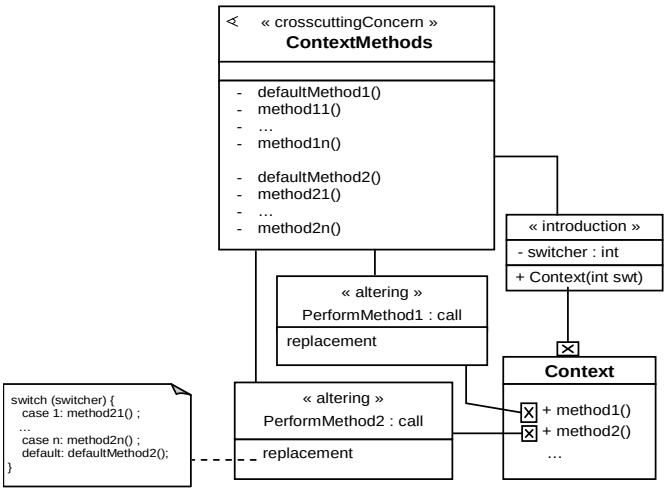
Figure 5.12. *Structure par aspects de* Fabrique abstraite

Par abstraction de ces trois modèles et de celui de *Stratégie* (figure 5.1, page 163) nous proposons celui du patron *Class Polymorphic Behaviour* présenté dans la rubrique *Solution* de la description de ce patron (cf. tableau 5.6). Ce nouveau patron est utilisé par l'ensemble des quatre patrons desquels il est issu, et qui répondent à des problèmes plus spécifiques que celui de *Class Polymorphic Behaviour*.

Tableau 5.6. *Description du patron* Class Polymorphic Behaviour

Identifier	*Class Polymorphic Behaviour*
Classification	object ^ alter ^ replacement ^ instantiation
Problem	Offre un comportement polymorphe aux instances d'une classe, sans changer sa définition. Ce comportement peut être défini par une ou plusieurs opérations, dont les diverses variantes sont définies et encapsulées dans une préoccupation transversale. La variante idoine associée à une instance est définie une fois pour toutes au moment de sa création.

Context	***Applicability*** On utilise *Class Polymorphic Behaviour* dans les cas où : - on a besoin de diverses variantes pour une classe Context d'un comportement qui peut être défini par une ou plusieurs opérations ; - les instances de Context ne diffèrent que par leur comportement. *Class Polymorphic Behaviour* donne le moyen d'appareiller chaque instance avec une variante parmi plusieurs autres ; - les variantes idoines doivent être associées une fois pour toutes aux instances au moment de leur création.
Forces	***Forces*** *Class Polymorphic Behaviour* présente les avantages suivants : - la définition de la classe impactée est inchangée, celle-ci se trouve facile à appréhender, à maintenir et à faire évoluer ; - il offre une solution alternative au codage explicite d'expressions conditionnelles dans le code de Context, qui nuit à la lisibilité de ce code ; - il offre aussi d'une solution alternative au polymorphisme d'inclusion, et donc à la dérivation en sous-classes qui présente l'inconvénient de coder explicitement le comportement dans la classe Context et ses dérivées, donc de mélanger les diverses variantes du comportement polymorphe avec l'implémentation des différentes classes Context. Ceci évite d'ailleurs d'augmenter le nombre de classes du système ; - il offre également une solution alternative à la délégation et la composition d'objets qui présentent l'inconvénient d'augmenter le nombre de classes/objets du système et leur communication, et rend ainsi plus difficile la compréhension, la maintenance et l'évolution du code de celui-ci ; - Les diverses variantes du comportement polymorphe sont définies et encapsulées dans une préoccupation transversale ; leurs fonctionnalités communes peuvent ainsi être factorisées. *Class Polymorphic Behaviour* permet d'ailleurs à ces variantes d'évoluer indépendamment des clients qui les utilisent. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Identifier la ou les opérations de Context destinées à être polymorphes. - Définir une préoccupation transversale (*crosscutting concern*) privilégiée regroupant : ▪ les définitions des différentes variantes des opérations polymorphes, ▪ la définition d'une introduction dans la classe Context d'un attribut switcher, utilisé pour la représentation interne de la variante idoine associée à chacune des instances de Context au moment de sa création, ▪ la définition d'une introduction dans la classe Context d'un nouveau constructeur paramétré destiné à initialiser l'attribut switcher, ▪ les définitions des spécifications d'entrecroisement (*crosscutting specifications*) et leurs éléments d'altération (*altering elements*) correspondants, permettant de remplacer les opérations polymorphes au moment de leur appel par l'appel de leurs variantes idoines (suivant la valeur de l'attribut switcher de l'instance de Context invoquée). - Les opérations polymorphes peuvent utiliser une ou plusieurs propriétés (attributs et opérations privés ou non) de la classe Context. Les éléments d'altération doivent donc fournir aux diverses variantes de ces opérations toutes les données qu'elles requièrent, lorsqu'elles sont appelées. Chaque élément d'altération crée les paramètres nécessaires à l'opération polymorphe qu'il altère, et les initialise avant d'appeler ses définitions idoines. D'ailleurs, si besoin est, il doit carrément communiquer l'instance Context comme un argument à ces diverses variantes. Ceci permet à ces dernières de rappeler le contexte si nécessaire.

	Class/Aspect Diagram 
Consequences	Le patron *Class Polymorphic Behaviour* possède les conséquences suivantes : - les clients de Context doivent être informés des différentes valeurs possibles de l'attribut switcher, et des diverses variantes correspondantes. Ils doivent d'ailleurs avoir compris en quoi ces variantes de comportement diffèrent avant de pouvoir choisir la plus appropriée ; - la solution proposée ne permet pas de changer la variante associée à une instance au moment de son exécution.
Relations	***Uses*** - *Add Features*. Il est utilisé pour introduire l'attribut switcher et le nouveau constructeur paramétré dans Context. - *Alter Behaviours*. Il est utilisé pour altérer les opérations destinées à être polymorphes.

2.5 Le patron Class Polymorphic Behaviour with Standalone Classes

Les figures 5.13 et 5.14 montrent respectivement les structures par aspects des patrons *Monteur* et *Commande*. Chacun de ces deux patrons proposent d'adapter une ou plusieurs opérations d'une classe (Director dans *Monteur* et Invoker dans *Commande*), tout en donnant un comportement polymorphe aux instances de celle-ci.

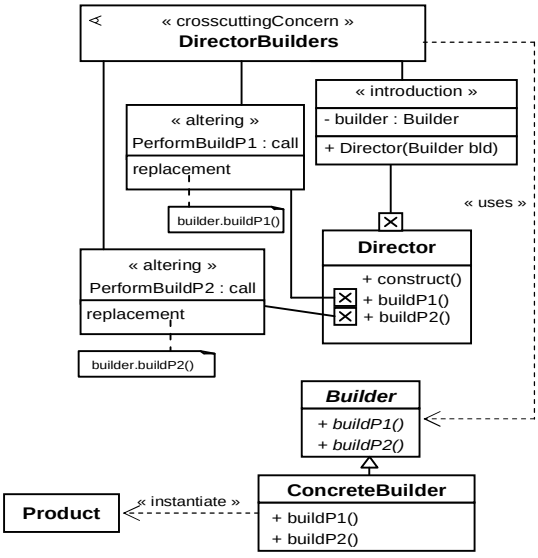


Figure 5.13. *Structure par aspects de* Monteur

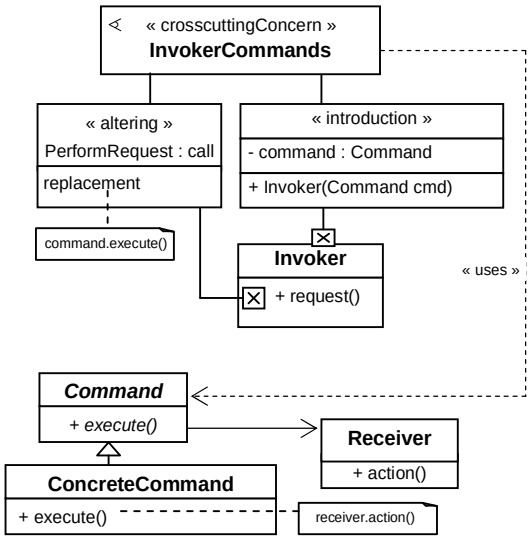


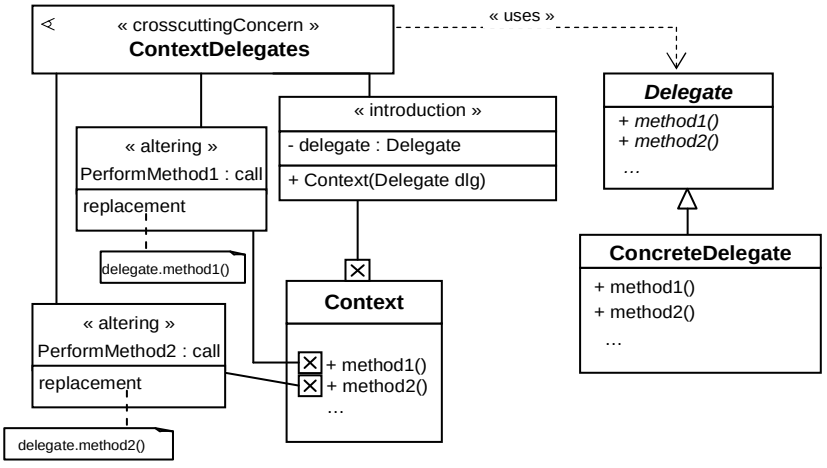
Figure 5.14. *Structure par aspects de* Commande

Pour ce faire, ils proposent, à la différence des quatre patrons précédents, de regrouper les différentes variantes des opérations dans des classes de comportement (Hiérarchie des Builder pour *Monteur* et hiérarchie des Command pour *Commande*). À chaque instance de la classe impactée est associé un objet délégué, défini une fois pour toutes au moment de sa création. Les requêtes envoyées à ces instances sont déléguées à leurs objets de comportement associés, grâce à l'utilisation des concepts d'altération de comportement (*crosscutting specification* et *altering element*) d'Aspect/UML.

L'abstraction des analogies existantes entre ces deux modèles par aspects donne lieu au patron *Class Polymorphic Behaviour with Standalone Classes*, dont la structure par aspects est donnée dans la rubrique *Solution* de la description de ce patron (cf. tableau 5.7). Ce patron constitue une alternative au patron *Class Polymorphic Behaviour*, tout en proposant une autre solution avec des forces et des conséquences différentes. Cette solution favorise, par exemple, la réutilisation des différentes variantes de comportement qui sont externalisées de la préoccupation transversale (*crosscutting concern*). Elle augmente cependant le nombre d'objets participants et accroît la communication entre ceux-ci.

Tableau 5.7. *Description du patron* Class Polymorphic Behaviour with Standalone Classes

Identifier	*Class Polymorphic Behaviour with Standalone Classes*
Classification	object ^ alter ^ replacement ^ instanciation
Problem	Offre un comportement polymorphe aux instances d'une classe, sans changer sa définition. Le comportement peut être défini par une ou plusieurs opérations, dont les différentes variantes doivent être définies, et encapsulées séparément dans des classes de comportement apparentées. L'objet de comportement approprié à une instance lui est associé et défini une fois pour toutes au moment de sa création. Le lien entre les classes de comportement et la classe impactée doit être extrinsèque à cette dernière. Il est assuré par l'utilisation d'une préoccupation transversale qui se préoccupe de déléguer les requêtes envoyées aux instances de la classe impactée à leurs objets associés.
Context	***Applicability.*** On utilise *Class Polymorphic Behaviour with Standalone Classes* dans les cas où : - on a besoin, pour une classe Context, de diverses variantes d'un comportement qui peut être défini par une ou plusieurs opérations. Ce patron permet de séparer et d'encapsuler chacune de ces variantes, dans des classes de comportement apparentées ; - les instances de Context ne diffèrent que par leur comportement. Ce patron donne le moyen d'appareiller chaque instance avec une variante parmi plusieurs autres, tout en lui associant un objet de comportement approprié ; - un seul objet de comportement doit être agrégé une fois pour toutes à une instance au moment de sa création ; - la classe Context ne doit pas être directement et étroitement liée aux classes de comportement, afin qu'elles puissent être réutilisées séparément.
Forces	***Forces.*** Les avantages de *Class Polymorphic Behaviour with Standalone Classes* sont les suivants. - La définition de la classe impactée est inchangée, celle-ci se trouve facile à appréhender, à maintenir et à faire évoluer. - Le couplage entre la classe Context et les classes de comportement est extrinsèque à la définition de Context. Le lien entre toutes ces classes est plutôt défini dans le contexte de la préoccupation transversale. Du fait que Context et les classes de comportement ne sont pas directement liées, elles peuvent ainsi être indépendamment et directement réutilisées dans d'autres contextes d'application. - Il offre une solution alternative au codage explicite d'expressions conditionnelles dans le code de Context, qui nuit à la lisibilité de celui-ci. - Il offre aussi une solution alternative au polymorphisme d'inclusion qui présente l'inconvénient de coder explicitement le comportement dans le contexte et ses classes dérivées, et donc de mélanger les différentes implémentations du comportement polymorphe avec l'implémentation des différentes classes Context. Ce qui évite de plus d'augmenter le nombre de sous-classes.

	- Les différentes variantes du comportement polymorphe sont définies et encapsulées séparément dans plusieurs classes de comportement apparentées ; leurs fonctionnalités communes peuvent ainsi être factorisées. Ces variantes peuvent d'ailleurs évoluer séparément et indépendamment des clients qui les utilisent. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Identifier la ou les opérations de Context destinées à être polymorphes. - Définir dans une hiérarchie de classes de comportement les différentes variantes d'implémentation de ces opérations. - Définir une préoccupation transversale (*crosscutting concern*) privilégié qui contient : ▪ la définition d'une introduction dans la classe Context d'un attribut delegate représentant l'objet de comportement associé à une instance particulière de Context, ▪ la définition d'une introduction dans la classe Context d'un nouveau constructeur paramétré qui initialise l'attribut delegate, ▪ la définition des spécifications d'entrecroisement (*crosscutting specifications*) et leurs éléments d'altération (*altering elements*) correspondants, permettant d'altérer les opérations impactées, tout en délégant leur exécution aux objets agrégés définis par l'attribut delegate. - Les diverses variantes des opérations polymorphes, définies dans les classes de comportement, peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privés ou non de la classe Context. Les éléments d'altération doivent donc fournir à ces variantes toutes les données nécessaires, lorsqu'elles sont appelées. Chaque élément d'altération crée les paramètres nécessaires à l'opération polymorphe qu'il altère, et les initialise avant d'appeler ses variantes à travers l'objet de comportement défini par l'attribut delegate. D'ailleurs, si nécessaire, les instances de Context doivent être communiquées, en tant qu'un argument, à ces diverses variantes au moment de la délégation de leurs requêtes. Ceci permet aux objets de comportement de rappeler le contexte si nécessaire. ***Class/Aspect Diagram*** 
Consequences	L'application du patron *Class Polymorphic Behaviour with Standalone Classes* possède les conséquences suivantes : - les clients de Context doivent être informés des différentes classes de comportement disponibles. D'ailleurs, ils doivent avoir compris en quoi les différents comportements de ces classes diffèrent avant de pouvoir choisir l'objet de comportement délégué ; - la solution proposée ne permet pas de changer l'objet délégué à une instance de Context au moment de son exécution ; - augmentation du nombre d'objets du système et surcharge de communication.

Relations	***Uses*** - *Add Features*. Il est utilisé pour introduire l'attribut delegate et le nouveau constructeur paramétré dans Context. - *Alter Behaviours*. Il est utilisé pour altérer les opérations destinées à être polymorphes. ***AlternativeOf*** - *Class polymorphic Behaviour*. *Class Polymorphic Behaviour with Standalone Classes* constitue une alternative à ce patron.

2.6 Le patron Instance Polymorphic Behaviour with Standalone Classes

Les patrons *État* et *Pont*, dont les structures par aspects sont respectivement montrées par les figures 5.15 et 5.16, proposent également de donner un comportement polymorphe aux instances d'une classe contexte (Context pour *État* et Abstraction pour *Pont*) par délégation à des objets détenant ce comportement (de type State pour *État* et Implementor pour *Pont*). Ils permettent, cependant, de rendre interchangeables ces objets de comportement : chaque instance de la classe contexte peut être appareillée à plusieurs objets délégués, au cours de son exécution. Pour ce faire, ils proposent d'utiliser une opération supplémentaire (setState(State s) pour *État* et setImplementor(Implementor imp) destinée à être introduite dans la classe Context. Il appartient ensuite à l'utilisateur de définir les critères de changement d'objets délégués. Si ces critères sont prédéterminés, par exemple, ils peuvent être définis intégralement et encapsulés dans la préoccupation transversale (comme c'est le cas pour *État*).

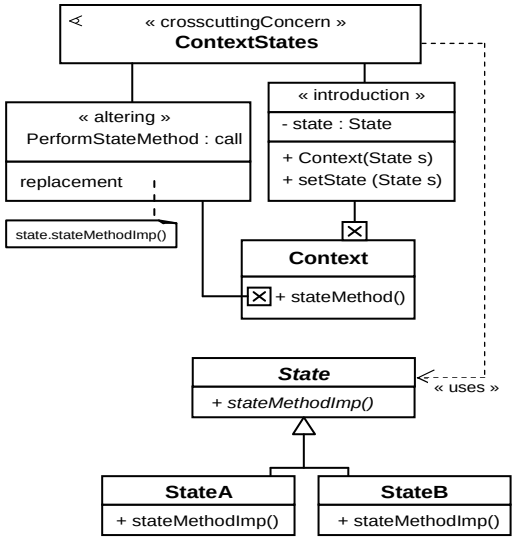


Figure 5.15. *Structure par aspects de* État

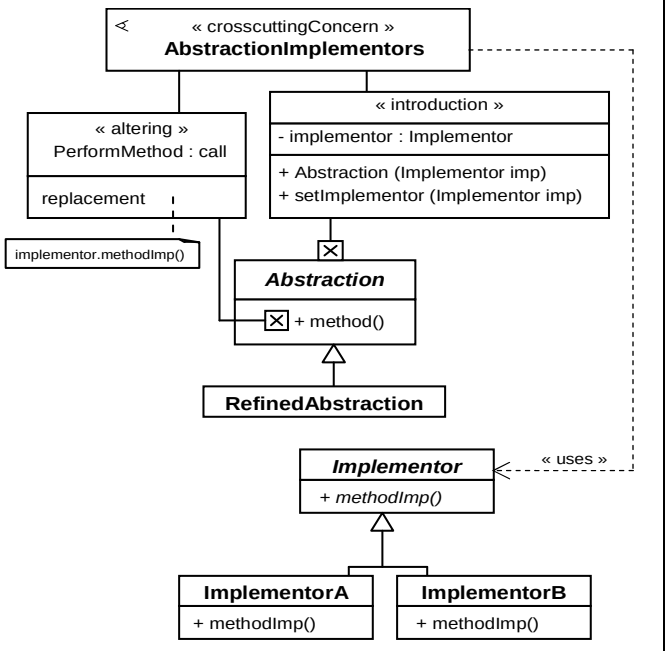
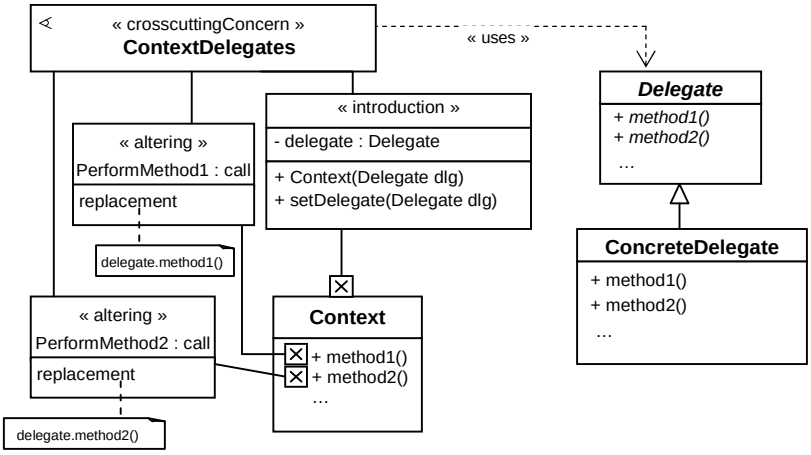


Figure 5.16. *Structure par aspects de* Pont

Par abstraction de ces analogies existantes entre les modèles Aspect de ces deux patrons du GoF nous proposons le patron *Instance Polymorphic Behaviour with Standalone Classes*, dont la structure par aspects est donnée par la rubrique *Solution* du tableau 5.8 qui décrit ce patron.

Tableau 5.8. *Description du patron* Instance Polymorphic Behaviour with Standalone Classes

Identifier	*Instance Polymorphic Behaviour with Standalone Classes*
Classification	object ^ alter ^ replacement ^ execution
Problem	Offrir un comportement polymorphe aux instances d'une classe, sans changer sa définition. Le comportement associé à une instance (au moment de sa création) peut être changé au moment de son exécution. Il est défini par une ou plusieurs opérations, dont les différentes variantes doivent être définies et encapsulées séparément dans des classes de comportement apparentées. *Instance Polymorphic Behaviour with Standalone Classes* rend ces différentes variantes interchangeables et leur permet d'évoluer indépendamment des clients qui les utilisent. Le couplage entre les classes de comportement et la classe impactée doit être extrinsèque à cette dernière. Il est cependant assuré par la préoccupation transversale.
Context	***Applicability*** On utilise *Instance Polymorphic Behaviour with Standalone Classes* dans les cas où : - on a besoin de diverses variantes pour une classe Context d'un comportement qui peut être défini sur plusieurs opérations. Ces variantes, interchangeables pour les instances de cette classe, doivent être définies et encapsulées séparément ; - les instances de Context ne diffèrent que par leur comportement modifiable au moment de leur exécution. Ce patron donne le moyen d'appareiller chaque instance avec une variante parmi plusieurs autres, tout en lui associant plusieurs objets de comportement au moment de sa création et au cours de son exécution ; - la classe Context ne doit pas être directement et étroitement couplée aux classes de comportement, afin de pouvoir les réutiliser séparément.
Forces	***Forces.*** Les avantages de *Class Polymorphic Behaviour with Standalone Classes* sont les suivants : - la définition de la classe impactée est inchangée, celle-ci se trouve facile à appréhender, à maintenir et à faire évoluer ; - les classes de comportement et la classe Context peuvent être facilement étendues par dérivation. Ce patron permet de combiner les différentes instances de type Context et les différents objets de comportement ; - le couplage entre Context et les classes de comportement est extrinsèque à la définition de Context, il n'est défini que dans le contexte du *crosscutting concern*. Du fait que Context et les classes de comportement ne sont pas directement liées, elles peuvent être directement réutilisées indépendamment dans d'autres contextes d'application ; - il offre une solution alternative au codage explicite d'expressions conditionnelles dans le code de Context, qui nuit à la lisibilité de son code ; - il offre aussi une solution alternative au polymorphisme d'inclusion qui présente l'inconvénient de coder explicitement le comportement dans le contexte et ses dérivés, donc de mélanger les différentes implémentations du comportement polymorphe avec l'implémentation des différentes classes Context ; - les différentes variantes du comportement polymorphe sont définies et encapsulées séparément dans plusieurs classes de comportement apparentées ; leurs fonctionnalités communes peuvent ainsi être factorisées. Ces variantes peuvent d'ailleurs évoluer indépendamment des clients qui les utilisent. ***Qualifiers***. no coupling ^ ease evolution ^ improve reusability
Solution	***Solution*** - Identifier la ou les opérations de Context destinées à être polymorphes. - Définir dans une hiérarchie de classes de comportement les différentes variantes d'implémentation de ces opérations. - Définir une préoccupation transversale (*crosscutting concern*) privilégiée qui contient : ▪ la définition d'une introduction dans Context d'un attribut delegate représentant l'objet de comportement associé à une instance particulière de cette classe, ▪ la définition d'une introduction dans la classe Context d'un nouveau constructeur paramétré qui initialise cet attribut,

	▪ la déclaration d'une méthode setDelegate(Delegate dlg) permettant de changer la valeur de delegate au moment de l'exécution d'une instance de Context, ▪ la définition des spécifications d'entrecroisement (*crosscutting specifications*) et leurs éléments d'altération (*altering elements*) correspondants, permettant d'altérer les opérations impactées, tout en délégant leur exécution aux objets agrégés. - Les diverses variantes des opérations polymorphes, définies dans les classes de comportement peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privées ou non de la classe Context. Les éléments d'altération doivent donc fournir à ces variantes toutes les données nécessaires, lorsqu'elles sont appelées. Chaque élément d'altération crée les paramètres nécessaires à l'opération polymorphe qu'il altère, et les initialise avant d'appeler ses variantes à travers l'objet de comportement défini par l'attribut delegate. D'ailleurs, si nécessaire, les instances de Context doivent être communiquées, en tant qu'argument, à ces diverses variantes au moment de la délégation de leurs requêtes. Ceci permet aux objets de comportement de rappeler le contexte si nécessaire. - Il appartient à l'utilisateur de définir les critères de changement d'objets agrégés. Si ces critères sont prédéterminés, par exemple, ils peuvent être implémentés intégralement dans la préoccupation transversale. ***Class/Aspect Diagram*** 
Consequences	Le patron *Instance Polymorphic Behaviour with Standalone Classes* possède les conséquences suivantes : - les clients de Context doivent être informés des différentes classes de comportement disponibles. D'ailleurs, ils doivent avoir compris en quoi les différents comportements de ces classes diffèrent avant de pouvoir choisir l'objet délégué ; - augmentation du nombre d'objets du système et surcharge de communication.
Relations	***Uses*** - *Add Features*. Il est utilisé pour introduire l'attribut delegate, le nouveau constructeur paramétré, ainsi que l'opération setDelegate(Delegate dlg) dans Context. - *Alter Behaviours*. Il est utilisé pour altérer les opérations destinées à être polymorphes. ***Refines*** - *Class polymorphic Behaviour*. Ce patron est raffiné par *Instance polymorphic Behaviour with Standalone Classes*, qui permet de plus de changer dynamiquement le comportement associé à une instance de Context au moment de son exécution. - *Class polymorphic Behaviour with Standalone Classes*. Ce patron est également raffiné par *Instance polymorphic Behaviour with Standalone Classes*.

2.7 Le patron Add New Functionalities

Les figures 5.17 et 5.18 montrent, respectivement, les structures par aspects obtenues pour *Mémento* et *Itérateur*. Indépendamment de leurs intentions spécifiques, ces deux patrons proposent d'ajouter de nouvelles fonctionnalités à une classe Context existante, et à ses sous-classes si elles existent, sans changer leurs comportements initiaux. Nous remarquons que les structures par aspects de ces deux patrons présentent des analogies. Elles se basent sur la définition de classes supplémentaires (ContextMemento pour *Mémento* et ContextIterator pour *Itérateur*) détenant les nouvelles fonctionnalités pour le compte de Context. Ces classes supplémentaires sont définies comme étant internes aux préoccupations transversales. Le lien entre ces classes internes (qui peuvent admettrent des classes dérivées, pour encapsuler les différentes variantes des fonctionnalités destinées à être ajoutées dans les sous-classes de Context si elles existent) et les classes pour lesquelles elles détiennent les fonctionnalités ajoutées est ensuite établi grâce, d'une part, à la définition de classes abstraites rôles qui sont internes à la préoccupation transversale, et à l'utilisation de *Add New Role* d'autre part, pour concrétiser et introduire les différentes variantes des opérations nécessaires à la définition de ces fonctionnalités.

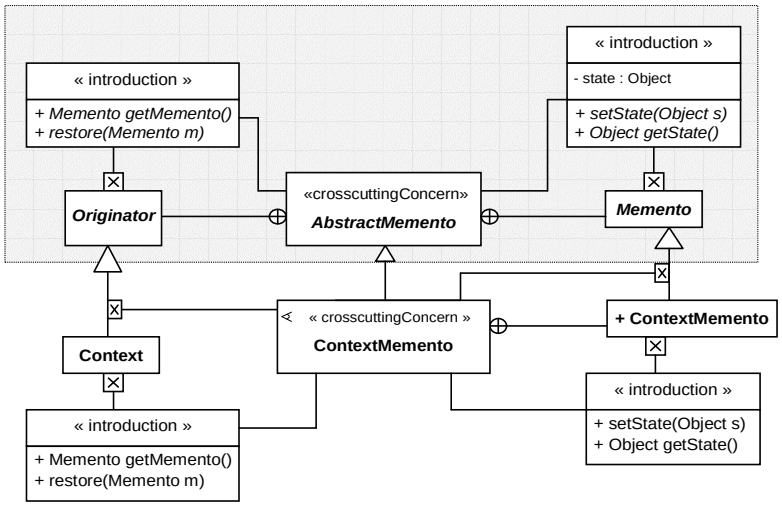


Figure 5.17. *Structure par aspects de* Mémento

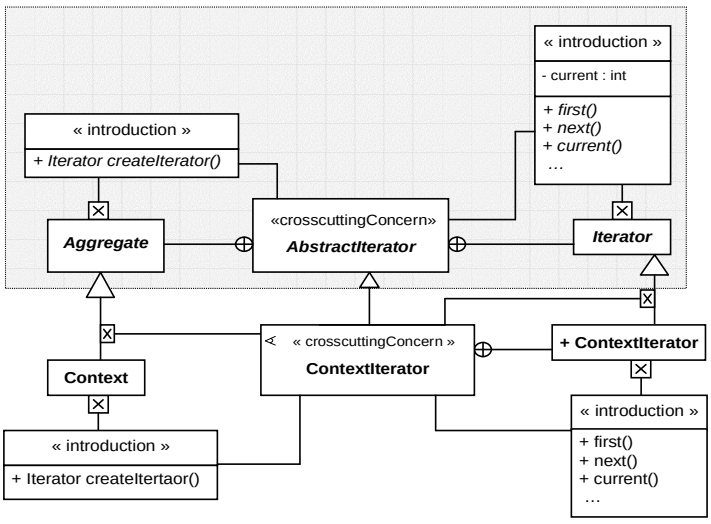
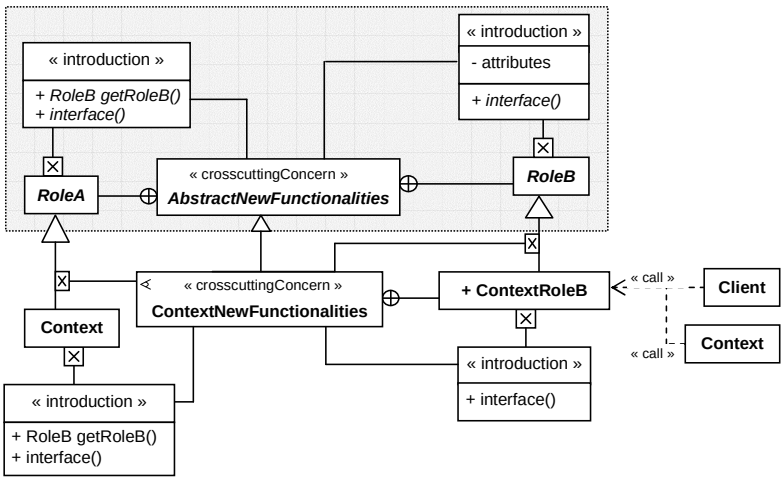


Figure 5.18. *Structure par aspects d'*Itérateur

Par abstraction des analogies existantes entre ces deux patrons nous avons défini le patron par aspects *Add New Functionalities*, dont la description est donnée par le tableau 5.9.

Tableau 5.9. *Description du patron* Add New Functionnalities

Identifier	*Add New Functionalities*
Classification	class ^ add
Problem	Permet d'étendre les fonctionnalités d'une classe (et ses sous-classes, si elles existent), sans modifier sa définition de base et sans changer son comportement d'origine. Les éléments (attributs et opérations) nécessaires à la réalisation de ces nouvelles fonctionnalités doivent être définies et encapsulées dans une classe additionnelle, dont on a souvent besoin de créer plusieurs instances.
Context	***Applicability*** On utilise *Add New Functionalities* dans le cas où on a besoin d'étendre l'interface d'une classe Context et ses sous-classes (si elles existent), tout en leur ajoutant de nouvelles fonctionnalités, sans modifier leurs définitions de base. Le comportement d'origine de ces classes doit être également inchangé. Les nouvelles fonctionnalités doivent être définies et encapsulés dans une autre classe supplémentaire, dont on peut avoir besoin de créer plusieurs instances. Cette classe peut admettre des sous-classes, détenant les diverses variantes des fonctionnalités, pour le compte des sous-classes de Context si elles existent.
Forces	***Forces.*** Les avantages de *Add New Functionalities* sont les suivants : - les définitions de la classe Context et de ses dérivées sont inchangées. Elles se trouvent faciles à appréhender, à maintenir et à faire évoluer. Elles peuvent d'ailleurs être directement réutilisées dans plusieurs autres contextes d'application, du moment où elles sont indépendantes de leurs nouvelles préoccupations ; - tout le code spécifique à la nouvelle préoccupation transversale est séparé et encapsulé dans les classes détenant les nouvelles fonctionnalités, il est plus facile à appréhender, à maintenir, et à faire évoluer indépendamment des classes contextes impactées. ***Qualities*** no coupling ease evolution ^ improve reusability ^ reusable code
Solution	***Solution*** - Définir une préoccupation transversale abstraite (*crosscutting concern*) regroupant : ▪ la définition d'une classe interne et abstraite RoleA représentant le rôle joué par la classe Context (et éventuellement par ses dérivées), ▪ la définition d'une classe interne et abstraite RoleB représentant le rôle joué par la classe supplémentaire (et éventuellement par ses dérivées), détenant pour le compte de Context, les fonctionnalités destinées à être ajoutées. ▪ toutes les définitions d'introduction des propriétés (attributs et/ou opérations) nécessaires à la réalisation des nouvelles fonctionnalités, respectivement dans RoleA et RoleB. Les opérations de ces rôles doivent être définies comme étant abstraites. - Définir une préoccupation transversale concrète regroupant : ▪ la définition d'une classe concrète et interne ContextRoleB, qui spécialise RoleB. Cette classe doit être déclarée comme étant publique pour les clients qui l'utilisent (notamment pour la classe Context qui a la charge de créer une instance de cette classe et la retourner au Client, en utilisant getRoleB()). Cette classe concrétise toutes les opérations nécessaires à la réalisation des nouvelles fonctionnalités appropriées à la classe Context. ▪ les définitions des déclarations d'héritage (*parent declarations*) permettant d'apparenter, d'une part, la classe Context qui se voit avoir le type RoleA, et d'autre part, la classe ContextRoleB qui se voit avoir le type RoleB. ▪ les définitions d'introduction dans la classe Context des concrétisations des opérations abstraites de RoleA. - Les opérations destinées à être ajoutées dans la classe Context peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privées de celle-ci. La préoccupation transversale concrète doit être définie dans ce cas comme privilégiée, mais pas nécessairement dans le cas contraire.

	- ContextRoleB est considérée comme étant une extension de la classe Context. ContextRoleB peut donc requérir quelques uns des attributs de Context. Dans ce cas, l'opération getRoleB() nouvellement introduite dans Context doit fournir à ContextRoleB toutes les données essentielles à la réalisation des nouvelles fonctionnalités. Il est souvent nécessaire, d'ailleurs, que ContextRoleB fait appel aux opérations de Context. Il conviendra dans ce cas de passer toute l'instance de Context comme un argument à sa nouvelle opération getRoleB() devant créer l'instance de ContextRoleB appropriée. ***Class/Aspect Diagram*** 
Consequences	Le patron *Add New Functionalities* possède les conséquences suivantes : - l'interface de ContextRoleB dispense du besoin d'une interface similaire dans Context, simplifiant de ce fait celle de Context ; - un ContextRoleB peut être défini comme une extension de Context. Context et ContextRoleB se trouvent ainsi étroitement liée. *Add New Functionalities* permet d'éviter ce couplage, qui se trouve abstrait au niveau des rôles définis dans le contexte de la préoccupation transversale abstraite.
Relations	***Uses*** - *Add New Role.* il est utilisé pour la définition des rôles RoleA et RoleB, et l'introduction des opérations concrètes respectivement dans la classe Context et la classe ContextRoleB.

2.8 Le patron Encapsulate Complex Functionality

Le patron *Observateur* a une structure par aspects (cf. figure 5.19) similaire à celle de *Chaîne de responsabilité* (cf. figure 5.20). Indépendamment de leurs intentions spécifiques, ces deux patrons proposent en fait de définir et d'encapsuler une nouvelle fonctionnalité complexe, par collaboration de plusieurs classes existantes (ConcreteSubject et ConcreteObserver dans *Observateur*, et Handlerk dans *Chaîne de responsabilité*) jouant des rôles différents (Subject et Observer dans *Observateur*) ou non différents (Handler dans *Chaîne de responsabilité*) dans le cadre de cette fonctionnalité. Ces rôles définissent, de manière abstraite, toutes les propriétés nécessaires à l'accomplissement de la nouvelle fonctionnalité. Leurs opérations abstraites doivent ensuite être concrétisées et introduites dans les classes concrètes concernées. Pour ce faire, ces deux patrons proposent d'utiliser le patron *Add New Role*.

Dans ces deux patrons, le comportement coopératif des classes en collaboration est défini comme étant unidirectionnel : il s'agit, en fait, d'une collaboration amenée par une classe particulière (ConcreteSubject ou Handler1, selon qu'il s'agit, respectivement, d'*Observateur* ou de *Chaîne de responsabilité*) et complétée ensuite par le reste des autres classes (ConcreteObserver dans *Observateur* et Handlerk (k=2..n) dans *Chaîne de responsabilité*). Pour établir convenablement la nouvelle fonctionnalité, ces

deux patrons proposent donc d'altérer les comportements originaux des classes contextes, afin de pouvoir établir la collaboration.

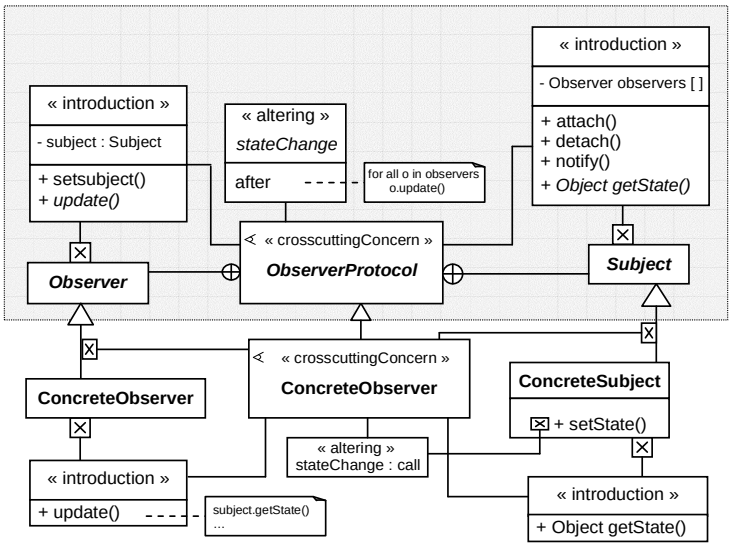


Figure 5.19. *Structure par aspects d'*Observateur

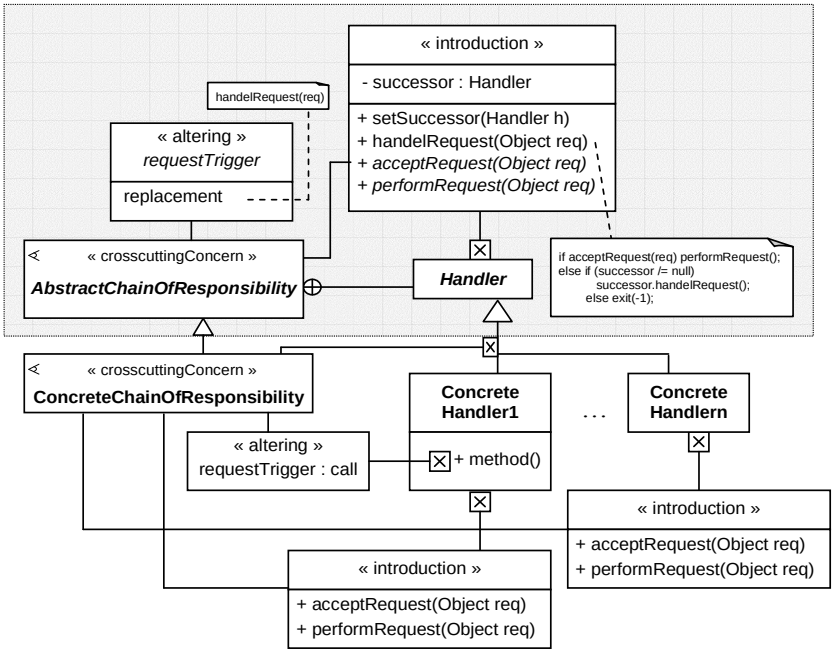


Figure 5.20. *Structure par aspects de* Chaîne de responsabilité

Il convient de noter ici que, le protocole de collaboration nécessaire à la réalisation de la nouvelle fonctionnalité est en partie défini de manière abstraite (i.e. indépendamment de la spécification du comportement d'origine concret de la classe contexte menant la collaboration) au niveau de la préoccupation transversale détenant les déclarations des rôles. Ce protocole est spécifié par un élément d'altération lié à une spécification d'entrecroisement abstraite (stateChange dans *Observateur* et requestTrigger dans *Chaîne de responsabilité*). Celle-ci est ensuite concrétisée par une préoccupation transversale concrète, afin d'instaurer complètement le protocole de collaboration, et donc d'altérer le comportement d'origine de la classe contexte menant cette collaboration.

Les analogies existantes entre les structures par aspects de ces deux patrons concernent également celle du patron *Poids mouche* (cf. figure 5.21), à la différence de la manière dont ce dernier définit son protocole de collaboration. En effet, le protocole de collaboration de *Poids mouche* ne peut être défini en partie de manière abstraite au niveau de la préoccupation AbstractFlyweight, car étroitement lié au comportement d'origine de la classe contexte (ConcreteFactory) menant la collaboration. Il se trouve ainsi complètement défini, de manière concrète, dans la préoccupation ConcreteFlyweight.

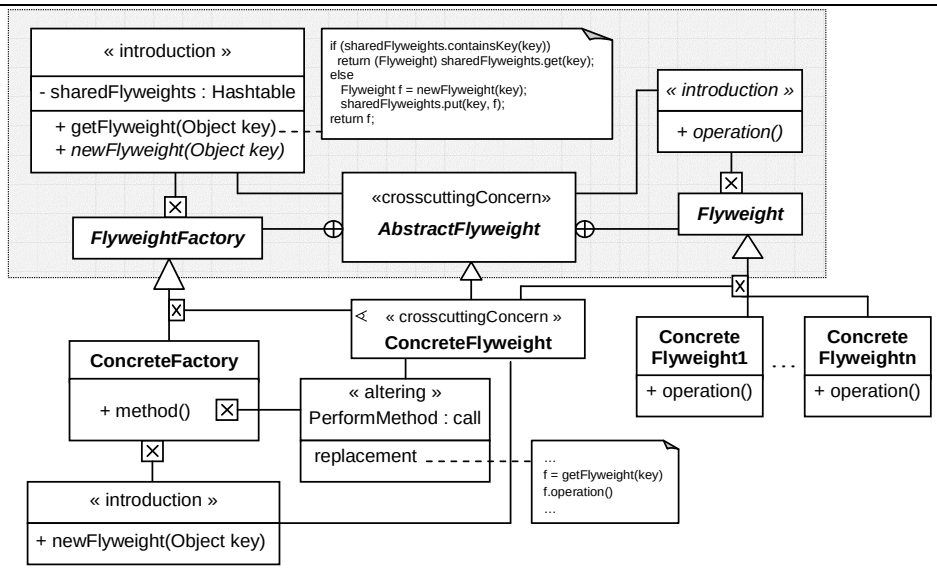


Figure 5.21. *Structure par aspects de* Poids mouche

La structure par aspects du patron *Médiateur* (cf. figure 5.22) est concernée également par ce dernier problème, son protocole de collaboration est défini donc au niveau de la préoccupation transversale concrète ConcreteMediator. Toutefois, à la différence des comportements coopératifs spécifiques aux trois patrons précédents, celui concernant les classes en collaboration dans *Médiateur* est multidirectionnel. Aucune classe particulière n'est en fait définie comme étant celle qui mène la collaboration. Toutes les classes impliquées voient, cependant, leurs comportements originaux altérés pour définir complètement la nouvelle fonctionnalité.

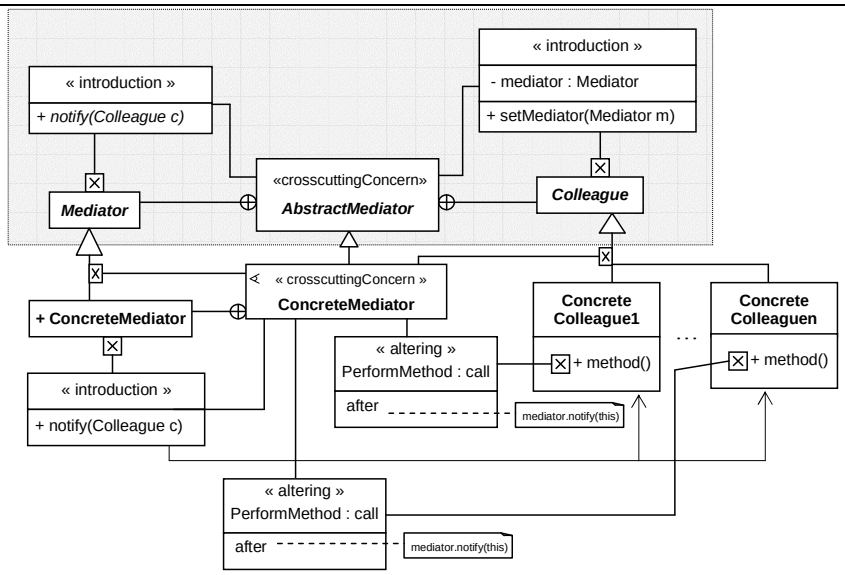


Figure 5.22. *Structure par aspects de* Médiateur

Par abstraction des analogies existantes entre les structures par aspects notamment des patrons *Observateur* et *Chaîne de responsabilité*, nous avons défini celle de *Encapsulate Complex Functionality* présentée dans la rubrique *Solution* de la description de ce patron (cf. tableau 5.10). Nous avons ensuite raffiné cette structure, de sorte qu'elle puisse également être utilisée pour répondre aux problèmes soulevés par *Poids mouche* et *Médiateur*. Nous n'avons pas pour autant défini deux nouveaux patrons par aspects supplémentaires, car le problème de fond de ces patrons est le même. Toutefois, nous avons choisi de spécifier, dans la rubrique *Alternatives* de la description du patron *Encapsulate Complex Functionality*, toutes les modifications nécessaires à l'adaptation de sa structure pour prendre en compte les besoins spécifiques de *Poids mouche* et *Médiateur*.

Tableau 5.10. *Description du patron* Encapsulate Complex Functionality

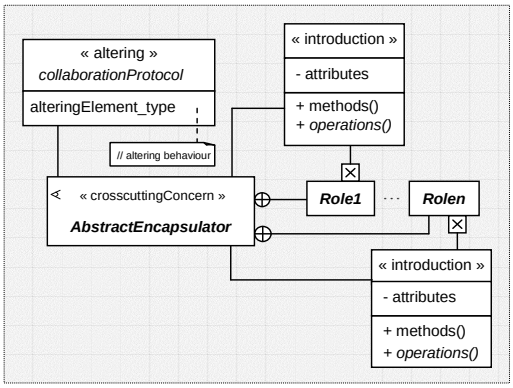
Identifier	*Encapsulate Complex Functionality*
Classification	object ^ alter ^ (after || replacement) ^ execution
Problem	Permet de définir et d'encapsuler une fonctionnalité complexe par la collaboration de plusieurs classes existantes, jouant des rôles différents dans le cadre de cette fonctionnalité. Le comportement coopératif peut être unidirectionnel ou multidirectionnel. Les définitions des classes mises en jeu ne doivent pas être modifiées.
Context	***Applicability*** On utilise *Encapsulate Complex Functionality* dans le cas où on a besoin de réaliser une nouvelle fonctionnalité complexe, mettant en collaboration plusieurs objets de différents types, donc de différentes classes. Le comportement coopératif peut être unidirectionnel ou multidirectionnel. *Encapsulate Complex Functionality* permet de définir et d'encapsuler une telle fonctionnalité, sans modifier les définitions des classes mises en jeu dans le cadre de cette collaboration. Il propose d'ailleurs deux solutions très proches, appropriée chacune à un type de comportement donné (uni/multidirectionnel).
Forces	***Forces*** Les avantages d'*Encapsulate Complex Functionality* sont les suivants : - les définitions des classes en collaboration sont inchangées. Celles-ci se trouvent faciles à appréhender, à maintenir et à faire évoluer. Elles peuvent d'ailleurs être directement réutilisées dans plusieurs autres contextes d'application, du moment où elles sont indépendantes les unes des autres ; - tout le code spécifique à la nouvelle fonctionnalité est séparé et encapsulé à part, il est plus facile à appréhender, à maintenir, et à faire évoluer indépendamment des classes (en collaboration) réalisant cette fonctionnalité. ***Qualifiers*** no coupling ^ ease evolution ^ improve reusability ^ reusable code
Solution	La solution suivante est appropriée à un comportement coopératif unidirectionnel. L'adaptation de celle-ci pour la réalisation d'un comportement coopératif multidirectionnel est discutée dans la rubrique *Alternatives*. Un cas particulier du problème soulevé par ce patron est traité également dans cette même rubrique, pour savoir adapter la structure présentée ci-dessous. ***Solution*** - Définir une préoccupation transversale (*crosscutting concern*) abstraite AbstractEncapsulator regroupant : ▪ la définition des classes internes et abstraites Rolek (k=1..n) représentant les différents rôles joués par les classes en interaction ;

- toutes les définitions d'introduction des propriétés (attributs et/ou opérations) nécessaires à la réalisation de la nouvelle fonctionnalité. Les opérations destinées à être polymorphes sont définies dans ces rôles comme étant abstraites ;
- la définition d'une spécification d'entrecroisement (*crosscutting specification*) abstraite collaborationProtocol et son élément d'altération (*altering element*) correspondant, permettant de définir complètement le protocole d'interaction à un niveau abstrait (indépendamment d'une application donnée, donc de ses classes concrètes en interaction).

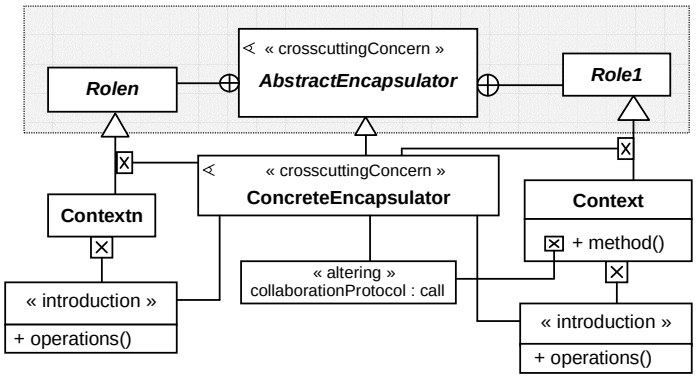
- Définir une préoccupation transversale concrète ConcreteEncapsulator regroupant :
  - Les déclarations d'héritage (parent declarations) permettant d'apparenter, les différentes classes participantes avec leurs rôles appropriés.
  - les définitions d'introduction dans les classes en interaction des concrétisations des opérations abstraites des différents rôles.
  - la concrétisation de la spécification d'entrecroisement abstraite collaborationProtocol, déterminant la méthode de la classe Context destinée à être altérée pour appliquer le protocole de collaboration (complètement défini par l'élément d'altération associé à collaborationProtocol) à l'ensemble des classes en interaction. La classe Context est définie ici comme étant la classe qui mène la collaboration.
- L'élément d'altération associé à la spécification d'entrecroisement abstraite peut accéder à une ou plusieurs propriétés privées des classes abstraites Rolek (k=1..n). Dans ce cas, la préoccupation abstraite AbstractEncapsulator doit être privilégiée.
- Les opérations destinées à être ajoutées dans les classes en interaction peuvent utiliser une ou plusieurs propriétés (attributs et opérations) privées de celles-ci. La préoccupation transversale concrète doit être définie dans ce cas comme privilégiée, mais pas nécessairement dans le cas contraire.

Class/Aspect Diagram

AbstractEncapsulator



ConcreteEncapsulator



Consequences	*Encapsulate Complex Functionality* offre une solution flexible à la réalisation de comportements coopératifs, à travers la définition des rôles en interaction.
Alternatives	Le patron *Encapsulate Complex Functionality* possède les alternatives suivantes. **1.** La définition du protocole de collaboration se trouve souvent, exclusivement et étroitement, liée au contexte concret de l'application. Il est ainsi impossible d'instaurer le protocole de collaboration à un niveau abstrait, dans la préoccupation AbstractEncapsulator. Dans ce cas, il convient de définir complètement le protocole de collaboration, de manière concrète, au niveau de la préoccupation ConcreteEncapsulator. Pour ce faire, il suffit de définir la spécification d'entrecroisement et son élément d'altération correspondant, requis pour adapter les comportements des classes en interaction, au niveau de ConcreteEncapsulator. Les définitions de la spécification d'entrecroisement abstraite et son élément d'altération correspondant dans AbstractEncapsulator deviennent inutiles. **2.** Dans le cas d'un comportement coopératif multidirectionnel, il n'y a pas une classe Context particulière qui conduit la collaboration. À l'inverse, les différents types de classes en interaction voient leurs comportements de base altérés, pour instaurer complètement le protocole de collaboration. Dans ce cas, une adaptation de la solution proposée répondant à ce problème plus spécifique consiste à : - définir complètement le protocole de collaboration au niveau de la préoccupation ConcreteEncapsulator. Les définitions de la spécification d'entrecroisement collaborationProtocol et son élément d'altération correspondant ne sont plus utiles. - définir toutes les spécifications d'entrecroisement nécessaires (et leurs éléments d'altération correspondants) permettant d'altérer les comportements des différentes classes en interaction multidirectionnelle.
Relations	***Uses*** - *Add New Role*. Ce patron est utilisé pour la définition des rôles Rolek (k=1..n), et l'introduction des opérations concrètes dans les différentes classes en collaboration. - *Alter Behaviours*. Ce patron est utilisé pour altérer le comportement des classes en interaction pour instaurer le protocole de collaboration.

3. Organisation et validation des patrons de conception par aspects

Pour compléter l'ingénierie des huit nouveaux patrons de conception par aspects ainsi identifiés et spécifiés, nous proposons, dans ce qui suit, de les organiser en fonction de l'ensemble de leurs inter-relations. Nous essayons ensuite, afin de valider ces huit nouveaux patrons, d'étudier la qualité des problèmes et solutions qu'ils proposent.

3.1 Organisation

Afin de faciliter la réutilisation des huit nouveaux patrons proposés, et pouvoir utiliser conjointement et correctement plusieurs patrons pour la conception d'un système d'information complexe, il convient de s'intéresser à leur organisation, et considérer donc l'ensemble des relations existant entre ces patrons. La figure suivante (cf. figure 5.23) présente la cartographie de ces huit nouveaux patrons. Nous avons conçu cette cartographie par la synthèse de l'ensemble des relations indiquées dans les différentes rubriques *Relations* de l'ensemble des spécifications des huit patrons par aspects.

Il convient de noter, au regard de cette cartographie, que les trois patrons *Add Features*, *Alter Behaviours* et *Add New Role* sont clairement utilisés par le reste de l'ensemble des huit nouveaux patrons par aspects que nous proposons.

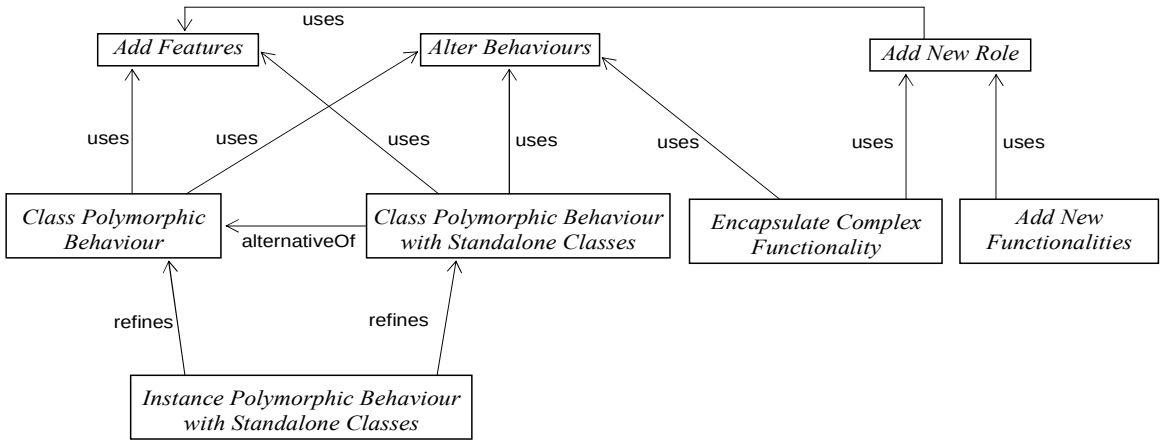


Figure 5.23. *Système de patrons de conception par aspects*

3.2 Validation des 8 nouveaux patrons et solutions alternatives

Les huit nouveaux patrons par aspects que nous proposons, en l'absence d'une réelle étude de cas empirique aboutissant à la conception d'un nouveau système d'information à base d'aspects, n'ont pu être véritablement validés. Cependant, afin de vérifier la qualité de ces patrons du point de vue de leur réutilisation, les exemples d'utilisation fournis dans ce qui suit permettent à la fois de valider leur utilité, mais aussi partiellement les problèmes identifiés et les solutions proposées. Il s'agit en fait de considérer toutes les relations existant entre les structures des huit nouveaux patrons et les structures par aspects que nous avons proposées pour les patrons de conception du GoF (cf. table 11, 1ère ligne de chaque cellule), mais aussi celles proposées par Hannemann [Web Hannemann] (2ème ligne de chaque cellule).

Cette mise en relation nous a permis, en effet, de déterminer jusqu'à quel point les 8 nouveaux patrons par aspects sont appliqués dans des exemples de conception par aspects. Elle nous a permis, également, d'identifier des solutions alternatives pour certains de ceux-ci. Dans ce qui suit, nous proposons de montrer l'utilité de ces patrons, à travers l'analyse du tableau précédent. Nous présentons également l'ensemble des solutions alternatives que nous avons pu identifier, afin de discuter leur qualité vis-à-vis des solutions que nous proposons pour ces huit nouveaux patrons par aspects.

Nous pouvons remarquer de nouveau que les trois patrons *Add Features*, *Alter Behaviours* et *Add New Role* sont largement utilisés par la plupart des structures par aspects des patrons de conception du GoF, bien qu'ils soient issus de l'abstraction de deux ou, au plus, de trois patrons du GoF :

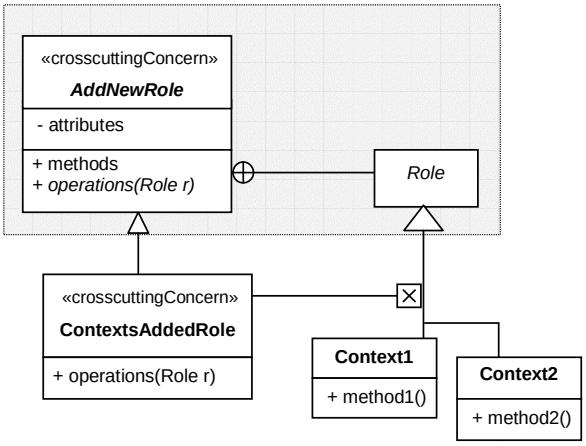
- plus particulièrement, le patron *Add Features* est utilisé par l'ensemble des conceptions par aspects des patrons du GoF ;
- quant à *Alter Behaviours*, il est utilisé par 15 de nos structures et uniquement par 9 des structures de Hannemann ;
- le patron *Add New Role* est utilisé par 10 de nos structures, il n'est utilisé, cependant, que par 5 des structures de Hannemann. Le reste des structures par aspects proposées par Hannemann pour les patrons du GoF, concernés par le problème de *Add New Role*, proposent et utilisent par contre une solution alternative à ce patron. Nous avons pu identifier cette solution alternative par l'abstraction notamment des analogies présentes entre les structures par aspects proposées par Hannemann pour les patrons *Prototype* et *Composite*. Cette alternative est utilisée également par les patrons *Mémento*, et *Poids mouche*, par exemple. Le tableau 5.12 décrit cette solution alternative, devant figurer dans la rubrique « Alternatives » de la description de *Add New Role*.

Tableau 5.11. *Relations entre les 8 nouveaux patrons et les solutions par aspects des patrons du GoF*

Patrons	*Add Features*	*Alter Behaviours*	*Add New Role*	*Class Polymorphic Behaviour*	*Class Polymorphic Behaviour with Standalone Classes*	*Instance Polymorphic Behaviour with Standalone Classes*	*Add New Functionalities*	*Encapsulate Complex Functionality*
Fabrique abstraite	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Monteur	uses	uses			uses			
	uses			alternativeOf	alternativeOf			
Fabrication	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Prototype	uses		uses					
	uses		alternativeOf					
Singleton	uses	uses						
		uses	uses					
Adaptateur	uses/may refine							
	refines							
Pont	uses	uses				Uses		
	uses							
Composite	uses		uses					
	uses		alternativeOf					
Décorateur	may use	uses						
	may use	uses						
Poids mouche	uses	uses	uses					uses
	uses							alternativeOf
Procuration	may use	uses	uses					
	may use	uses	uses					
Chaine de responsabilité	uses	uses	uses					uses
	uses	uses						alternativeOf
Commande	uses	uses	uses		uses			
	uses	uses	uses			alternativeOf		
Interpréteur	uses							
	uses							
Itérateur	uses		uses				uses	
	uses						alternativeOf	
Médiateur	uses	uses	uses					uses
	uses	uses						alternativeOf
Mémento	uses		uses				uses	
	uses						alternativeOf	
Observateur	uses	uses	uses					uses
	uses	uses						alternativeOf
État	uses	uses				Uses		
	uses	uses						
Stratégie	uses	uses		uses				
	uses	uses	uses			alternativeOf		
Patron de méthode	uses	uses		uses				
	uses			alternativeOf	alternativeOf			
Visiteur	uses							
	uses		uses					

Tableau 5.12. *Une solution alternative à* Add New Role

Une solution alternative à celle proposée par le patron *Add New Role*, dans la rubrique *Solution* de sa description (cf. tableau 5.5), consiste à déclarer un rôle vide, qui sert uniquement pour le typage. Les propriétés communes aux différentes classes Contextk (k=1..n) cibles d'entrecroisement, sont à déclarer comme étant des propriétés de la préoccupation transversale abstraite AddNewRole. Celle-ci détient pour le compte des classes contextes, tous les nouveaux services destinés à étendre ces dernières. Les opérations dépendantes des différentes classes contextes sont à déclarer comme étant abstraites, elles sont concrétisées ensuite dans la préoccupation transversale concrète ContextsAddedRole. La structure par aspects de cette nouvelle solution alternative est montrée ci-dessous.

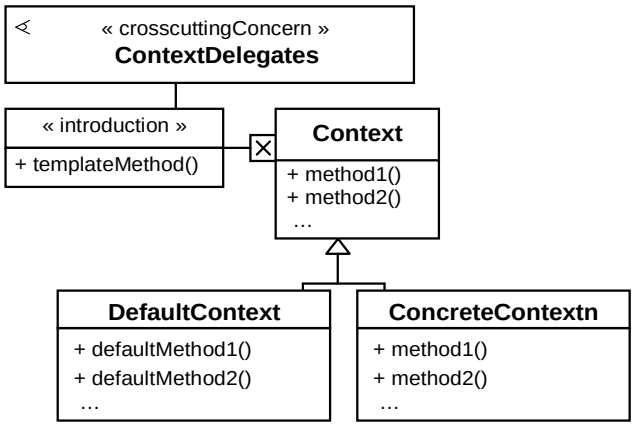


Les cinq autres patrons sont utilisés par les conceptions par aspects des patrons du GoF à partir desquels ils sont issus. Nous remarquons toutefois que les solutions proposées par Hannemann pour les patrons du GoF, concernés également par les problèmes soulevés par ces cinq patrons, proposent et utilisent des solutions alternatives à ces derniers :

- les structures des patrons *Monteur*, *Patron de méthode*, *Fabrication* et *Fabrique Abstraite* proposent et utilisent, par exemple, une solution alternative aux patrons *Class Polymorphic Behaviour* et *Class Polymorphic Behaviour with Standalone Classes*, que nous décrivons dans le tableau 5.13.

Tableau 5.13. *Une solution alternative à* Class Polymorphic Behaviour *et à* Class Polymorphic Behaviour with Standalone Classes

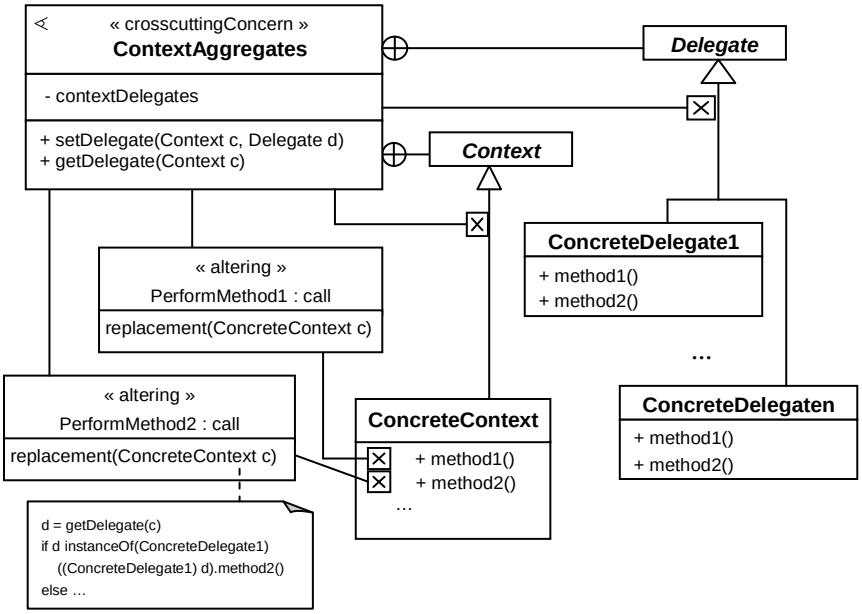
Une solution alternative à celles proposées, respectivement, par *Class Polymorphic Beahviour* (cf. rubrique *Solution* du tableau 5.6) et *Class Polymorphic Behaviour with Standalone Classes* (cf. rubrique *Solution* du tableau 5.7) consiste à dériver la classe Context en plusieurs sous-classes. Chacune de celles-ci définit une variante (ensemble des méthodes method1(), method2(),…) du comportement polymorphe dont les méthodes correspondantes sont définies et introduite dans Context comme étant vides. Cette solution propose ensuite de définir et d'introduire dans Context la méthode templateMethod() correspondant à ce comportement. Aucun élément d'altération et aucune spécification d'entrecroisement ne sont utiles dans ce cas. Ceci est d'ailleurs valable pour l'attribut delegate et le nouveau constructeur de Context. La structure par aspects de cette solution est la suivante.



- une solution alternative au patron *Instance Polymorphic Behaviour with Standalone Classes* est celle utilisée par les structures par aspects proposées par Hannemann pour les patrons *Stratégie* et *Commande*. Le tableau 5.14 décrit cette nouvelle alternative, devant apparaître dans la rubrique *Alternatives* de la description de *Instance Polymorphic Behaviour with Standalone*.

Tableau 5.14. *Une solution alternative à* Instance Polymorphic Behaviour with Standalone Classes

Une solution alternative à celle du patron *Instance Polymorphic Behaviour with Standalone Classes*, présentée dans la rubrique *Solution* de sa description (cf. tableau 5.8), consiste à définir deux classes Context et Delegate, représentant respectivement les rôles joués par la classe ConcreteContext (qui correspond à la classe Context définie dans la solution proposée par ce patron) et les classes de comportement (ConcreteDelegatek, (k=1..n)). Ces deux classes sont définies comme étant des classes abstraites et internes à la préoccupation transversale. Le lien entre ConcreteContext et les classes de comportement est assuré par l'utilisation d'un attribut contextDelegates. Cet attribut appartient également à la préoccupation transversale, ainsi que deux autres opérations supplémentaires : setDelegate(Context c, Delegate dlg) et getDelegate(Context c) qui permettent, respectivement, d'ajouter une instance de ConcreteContext et son objet de comportement associé dans le tableau contextDelegates, et de retourner l'objet agrégé à une instance particulière de ConcreteContext. Le diagramme de classes/aspects de cette solution est donné ci-dessous.

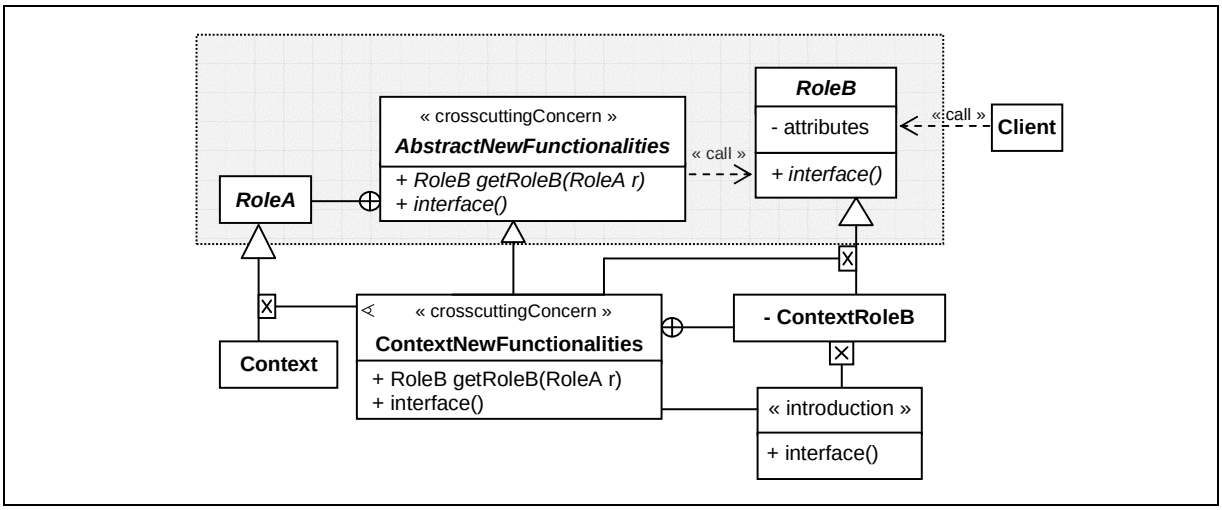


Avec cette manière de faire, il devient inutile d'introduire l'attribut delegate, le nouveau constructeur paramétré et la méthode setDelegate(Delegate dlg) dans la classe ConcreteContext. De plus, pour déléguer les requêtes des instances de ConcreteContext à leurs objets de comportement associés, les éléments d'altération doivent utiliser tout d'abord getDelegate(Context c) pour déterminer ces objets.

- par ailleurs, les structures par aspects proposées par Hannemann pour les patrons *Itérateur* et *Mémento* utilisent une solution alternative (cf. tableau 5.15) à celle proposée pour le patron *Add New Functionalities* ;

Tableau 5.15. *Une solution alternative à* Add New Functionalities

Une solution alternative à celle proposée par le patron *Add New Functionalities*, dans la rubrique *Solution* de sa description (cf. tableau 5.9), utilise l'alternative de *Add New Role* pour étendre l'interface de la classe Context à travers le rôle vide RoleA. Elle propose d'ailleurs de définir la classe abstraite RoleB comme étant externe à la préoccupation transversale AbstractNewFunctionalities. Elle définit ensuite, au niveau de la préoccupation ContextNewFunctionalities une classe concrète privée ContextRoleB qui hérite de RoleB. En effet, comme ContextNewFunctionalities détient les propriétés nécessaires à la réalisation des nouvelles fonctionnalités, les clients externes ainsi que la classe Context n'ont plus besoin d'accéder à la classe ContextRoleB (qui n'a plus besoin donc d'être déclarée comme publique). La structure par aspects de cette alternative est présentée ci-dessous.



- les solutions par aspects proposées par Hannemann à l'ensemble des patrons du GoF concernés par le problème du patron *Encapsulate Complex Functionality*, utilisent et proposent également différentes alternatives à la solution de celui-ci (présentée dans la rubrique Solution du tableau 5.10). Ces alternatives se basent en partie dans leurs solutions, le plus souvent, sur l'utilisation de l'alternative de *Add New Role*. D'ailleurs, dans les deux types de comportement coopératif unidirectionnel et multidirectionnel, et pour réaliser le lien entre les instances des classes en interaction, ces alternatives adoptent la même idée utilisée dans l'alternative du patron *Instance Polymorphic Behaviour with Standalone Classes*.

Il reste enfin à discuter la qualité de toutes ces solutions alternatives, par rapport à la qualité des solutions que nous proposons pour les huit nouveaux patrons par aspects considérés. Cette comparaison nous permet, en effet, de valider relativement nos solutions. Trois remarques importantes sont à signaler.

- *Encapsulation et séparation des préoccupations transversales sont souvent non respectées*. Il est à noter que, certaines des solutions alternatives proposées par Hannemann à nos patrons aspects n'assurent pas l'encapsulation et la séparation des préoccupations transversales qu'elles traitent. Par exemple, la solution alternative à celles proposées pour *Class Polymorphic Behaviour* et *Class Polymorphic Behaviour with Standalone Classes* ne garantit pas l'encapsulation et la séparation de la préoccupation transversale concernant le comportement polymorphe.
- *Solutions difficiles à mettre en œuvre dans le cas d'Hyper/J*. La solution alternative à celle proposée par *Add New Role* est particulièrement adaptée pour être mise en œuvre dans des langages tels qu'AspectJ. Elle est cependant complexe à appliquer dans le cas d'Hyper/J, par exemple. Elle nécessite, en effet, la définition et l'ajout d'une nouvelle classe supplémentaire (accessible par le client d'une part et les classes contextes, d'autre part). Par ailleurs, tout en se basant en partie sur l'utilisation de l'alternative de *Add New Role*, la solution alternative à celle proposée par *Add New Functionalities* est également bien adaptée pour des langages tel que AspectJ, mais complexe à mettre en œuvre pour le cas d'Hyper/J. De même, la solution alternative à celle proposée par *Instance Polymorphic Behaviour with Standalone Classes* est directement réalisable en AspectJ. Toutefois, dans le cas d'Hyper/J, il convient de simuler le rôle de mappage (joué par la préoccupation transversale) par une classe supplémentaire, définie comme étant une classe d'association entre les classes abstraites Context et Delegate. Une telle classe, doit déclarer l'attribut contextDelegates et les deux opérations setDelegate(Context c, Delegate dlg) et getAggregate(Context c). Elle est utilisée, d'une part, par le client qui crée l'ensemble des classes du système, et d'autre part, par l'image de la classe ConcreteContext (appartenant à l'hyperslice représentant la préoccupation transversale) qui doit

déléguer ses requêtes à son objet agrégé. Pour conclure, force est de noter donc que, bien adaptée pour AspectJ, ces solutions alternatives sont plus difficiles à mettre en œuvre dans le cas d'Hyper/J.

- *Augmentation du nombre de classes participantes dans le patron*. Il est à remarquer que toutes les solutions alternatives présentées auparavant, pour être appliquées en Hyper/J, nécessitent la définition et l'ajout de classes supplémentaires par rapport aux solutions que nous proposons pour les patrons par aspects considérés. Elles présentent ainsi l'inconvénient d'augmenter le nombre de classes participantes dans les patrons imités, mais aussi le nombre de communications entre objets du système.

4. Discussion et travaux connexes

La mise en relation des huit nouveaux patrons par aspects que nous avons identifiés et les modèles de conception proposés (par nous-même ou par Hannemann) pour l'ensemble des patrons du GoF, s'apparente à une première forme de validation de cet ensemble de patrons. Elle nous a permis, en effet, de vérifier la qualité de ces patrons d'un point de vue de leur réutilisation possible par des conceptions par aspects, et de qualifier en conséquence les problèmes soulevés par ces patrons. Par ailleurs, l'analyse des solutions alternatives que nous avons pu identifier nous a permis de plus de valider les solutions proposées pour ces 8 nouveaux patrons de conception par aspects. Dans cette même optique de validation, nous considérons, dans ce qui suit, plusieurs autres travaux connexes qui proposent également d'identifier de nouveaux problèmes, et donc de nouvelles solutions de conception par aspects. Nous proposons, notamment, d'identifier les relations existantes entre les problèmes et solutions qu'ils proposent et ceux des 8 nouveaux patrons que nous avons isolés.

- [Chavez *et al.*, 04] proposent quelques principes et directives préliminaires pour une bonne conception de systèmes par aspects, basée sur le langage de modélisation aSideML [Chavez 04]. Les directives de conception qu'ils proposent ont été notamment dérivées de leurs expériences d'implémentation de différentes applications (comme the Portalware MAS [Garcia *et al.*, 02]) avec AspectJ, mais également de différents exemples d'implémentation canoniques fournis par l'équipe d'AspectJ et plusieurs autres praticiens de ce langage. Parmi les directives qu'ils proposent, nous pouvons citer, par exemple, « *Aspects for Collaboration* ». Cette directive propose d'isoler et de localiser un comportement coopératif (mettant en jeu plusieurs classes en collaboration) dans un aspect, tout en définissant et utilisant des rôles pour les différents participants. Ce patron traite du même problème que le patron *Encapsulate Complex Functionality* que nous proposons, il propose cependant une solution alternative à celle de ce dernier. Une autre directive que nous pouvons considérer ici est « *Aspects for Evolution* », dont l'intention est d'étendre une classe de base existante tout en lui fournissant de nouvelles propriétés, et tout en lui permettant de jouer un nouveau rôle. Le patron *Add New Role* que nous avons identifié raffine l'intention de cette directive, afin de pouvoir attribuer le nouveau rôle à plusieurs classes de base à la fois. Il peut donc être utilisé pour répondre à l'intention de cette directive. Par ailleurs, la directive « *Aspects for Views* » propose d'offrir plusieurs interfaces, et donc différentes vues, à une classe de base qui peut être utilisée par plusieurs clients. Elle propose de définir un aspect par vue, tout en encapsulant l'ensemble des propriétés structurelles et comportementales nécessaires à la réalisation de cette vue. Elle utilise donc la solution du patron *Add Features* que nous proposons. Enfin, la directive « *Aspects and Obliviousness* » propose une solution pour un problème très spécifique. Il est remarquable toutefois, qu'elle utilise en partie dans sa solution les patrons *Alter Behaviours* et *Add Features*.
- [Monteiro 04] propose un catalogue de patrons de refactorisation permettant de faire évoluer un code Objet vers un code Aspect. Les patrons *Add Features*, *Alter Behaviours*, *Add New Role* et *Add*

New Functionalities, que nous avons identifiés, se retrouvent dans cet ensemble de patrons de refactorisation. Ils sont d'ailleurs utilisés par plusieurs de ces patrons du catalogue pour répondre totalement ou partiellement à leurs problèmes. « *Move Field from Class to Inter-type* » et « *Move Method from Class to Inter-type* » utilisent, par exemple, le patron *Add Features* qui traite à la fois des deux problèmes soulevés par ces deux patrons. Par ailleurs, le patron « *Extract Fragment into Advice* » utilise le patron *Alter Behaviours*, alors que « *Extract Feature into Aspects* » utilise *Alter Behaviours* et *Add Features* pour répondre à son problème. Quant à « *Extract Inner Class to Standalone* », il utilise la même solution que *Add New Functionalities*. Un autre exemple est celui de « *Generalise Target Type with Marker Interface* » qui utilise la même solution que *Add New Role*.

Le tableau suivant (cf. tableau 5.16) fait la synthèse des relations existant entre certains des patrons que nous proposons et quelques unes des directives proposées par [Chavez *et al.*, 04], mais aussi quelques uns des patrons de refactorisation de [Monteiro 04].

Tableau 5.16. *Relations existant entre 5 des 8 nouveaux patrons, les directives de [Chavez* et al.*, 04] et les patrons de refactorisation de [Monteiro 04]*

	Patrons	*Add Features*	*Alter Behaviours*	*Add New Role*	*Add New Functionalities*	*Encapsulate Complex Functionality*
Directives de conception par aspects de [Chavez* et al.*, 04]	*Aspects for Collaboration*					alternativeOf
	Aspects for Evolution			extends		
	Aspects for Views	uses				
	Aspects and Obliviousness	uses	uses			
Patrons de refactorisation de [Monteiro 04]	*Move Field from Class to Inter-type*	uses				
	Move Method from Class to Inter-type	uses				
	Extract Fragment into Advice		uses			
	Extract Feature into Aspects	uses	uses			
	Extract Inner Class to Standalone				uses	
	Generalise Target Type with Marker Interface			uses		

Un autre travail que nous pouvons considérer comme connexe à notre travail d'ingénierie de patrons de conception par aspects est celui de [Baumgartner *et al.*, 96], bien qu'il ne soit pas en relation directe. Ce travail étudie l'impact des mécanismes existants dans les langages de programmation par objets sur le nombre et la nature de patrons de conception. Sous l'hypothèse de l'extension effective de ces langages par des mécanismes qui sont proposés, il prévoit une réduction et une simplification du nombre de patrons significatifs dans l'ensemble des patrons du GoF. Il est remarquable que certains mécanismes proposés sont semblables à ceux qui ont depuis été introduits dans les langages de programmation par aspects, et que les problèmes soulevés peuvent être résolus en partie par les patrons que nous avons

identifiés. D'ailleurs, certains regroupements des patrons du GoF qu'ils considèrent dans leur travail de recherche sont similaires à quelques-uns de ceux que nous obtenons.

Enfin, partant du constat qu'il existe de fortes similitudes entre les patrons de conception par objets décrits dans la littérature, plusieurs travaux ont proposé d'abstraire ces derniers en s'appuyant sur différents critères, aboutissant ainsi à de nouveaux ensembles de patrons plus réduits. Les critères de rapprochement des patrons de base portent, par exemple, sur les différents types de relations liant les patrons entre eux, comme c'est le cas dans [Zimmer 95]. Notre approche est plus proche de [Diaz 02] qui base le rapprochement de patrons sur les concepts et mécanismes utilisés dans leur structure, mais ne considèrent pour autant que des mécanismes et concepts strictement Objet et n'aboutit donc pas aux mêmes groupements que nous obtenons.

5. Conclusion

Afin de valider le métamodèle général que nous avons proposé dans le chapitre précédent pour l'expression de conceptions par aspects indépendantes d'un langage de programmation par aspects considéré en particulier, nous l'avons appliqué à l'ensemble des patrons du GoF. Ce travail nous a permis, par étude d'analogies existant entre les expressions des structures de patrons que nous avons obtenues à partir des solutions par aspects proposées dans notre contribution initiale ainsi que celles proposées par [Web Hannemann], d'isoler huit nouveaux patrons de conception permettant d'utiliser les patrons du GoF dans le cadre d'un développement par aspects. Par ailleurs, pour une description complète et une meilleure réutilisation conjointe de ces huit patrons originaux, nous avons proposé une adaptation aux aspects d'un formalisme de description de patrons de conception que nous avons utilisé pour décrire un nouveau système de patrons.

Il n'est pas étonnant que le nombre de patrons significatifs s'en trouve réduise, dans la mesure où la technologie Aspect a été introduite pour repousser les limites de la technologie Objet, mais l'identification des problèmes récurrents et de leurs possibles solutions en conception par aspects reste un enjeu important. On peut en effet noter que nous avons abouti à cet ensemble de huit patrons en nous basant exclusivement sur l'étude de patrons par objets et non celle de développements logiciels par aspects. Nous pensons en conséquence que d'autres patrons de conception par aspects restent à découvrir, en particulier du fait que le nombre de classes entrecoupant un aspect est généralement supposé plus important que le nombre de classes considérées par un patron Objet. Nous pensons que cet ensemble de huit patrons peut toutefois être considéré comme un premier pas dans l'identification et la définition de patrons de conception par aspects qui ne soient pas de simples traductions des patrons de conception par objets. Ils peuvent de plus s'avérer utiles, dans une phase transitoire, dans l'évolution d'une conception par objets vers une conception par aspects, comme le laisse supposer les concordances avec les résultats des travaux de [Monteiro 04] sur la refactorisation de code Objet vers Aspect.

Conclusion générale et perspectives

Les patrons de conception par objets de type produit facilitent l'ingénierie de systèmes d'information de bonne qualité. Toutefois, étant définis le plus souvent sur plusieurs classes, en dépit de leurs nombreuses qualités ces patrons ont le défaut de généralement « disparaître » dans les implémentations des applications. Toute imitation d'un patron retrouve ainsi son code mélangé et disséminé dans les classes de l'application sur lesquelles elle s'applique, ce qui diminue la qualité des programmes et nuit fortement à l'évolution et la réutilisation des systèmes. Nous avons présenté dans cette thèse un état de l'art de nombreux modèles et langages de programmation par aspects actuellement introduits dans le cadre d'une nouvelle approche de programmation, l'approche Aspect. Nous avons particulièrement présenté une étude approfondie de la programmation par aspects (*AOP*) et du langage AspectJ d'une part, et de l'approche *Hyperspace* et du langage Hyper/J d'autre part, deux extensions par aspects du langage Java. Les mécanismes et concepts Aspect offerts par ces deux langages offrent une meilleure structuration des programmes, notamment au travers de l'encapsulation et de la séparation des préoccupations transversales au découpage des applications en termes de classes. En palliant les problèmes de dispersion et d'enchevêtrement de code de telles préoccupations, ils présentent ainsi des avantages reconnus quant à l'évolution et à la réutilisation des systèmes logiciels, et permettent par conséquent de s'affranchir de certaines limites de l'approche Objet.

Partant de ces constats, notre travail de recherche a eu plus précisément pour premier objectif de pouvoir localiser le code relatif à l'imitation d'un patron dans une application, et donc d'améliorer la traçabilité des patrons de conception jusqu'à la phase d'implémentation et augmenter en conséquence leur réutilisation, en tirant profit des concepts et mécanismes Aspect. Pour décrire cette première contribution, nous avons rappelé tout d'abord la notion de patron de conception et présenté un cas d'étude d'un système simple utilisant divers patrons dans sa solution. À travers cet exemple d'application, nous avons défini et détaillé ensuite quatre problèmes liés à l'utilisation des patrons de conception par objets issus notamment de [Gamma *et al.*, 95]. En effet, étant basées principalement sur les mécanises d'héritage, de composition et d'indirection explicite, les imitations des patrons posent plusieurs problèmes dont, principalement, les problèmes de confusion et de traçabilité, les problèmes d'indirection et de surcharge d'implémentation, les problèmes de rupture d'encapsulation et les problèmes liés à l'héritage. Ces problèmes sont essentiellement dus à la dispersion et à l'enchevêtrement du code de ces imitations dans celui relatif aux différentes classes constituant le système. Ainsi avons-nous proposé de nouvelles réalisations pour l'ensemble des patrons du GoF, en tirant parti des nouvelles techniques introduites dans le cadre de l'approche Aspect, et plus précisément dans les langages AspectJ et Hyper/J, pour éviter ces problèmes. Nous avons adopté une ligne de conduite qui consiste à obtenir des solutions aboutissant à l'identification d'une imitation de patron par une seule unité modulaire d'encapsulation dans l'ensemble du code d'une application. Une telle correspondance nous a permis de pallier l'ensemble des problèmes et donc des limites que posent les patrons, facilitant ainsi l'identification et la traçabilité de ces derniers jusqu'à la phase d'implémentation et améliorant par conséquent leur réutilisation mais aussi l'évolution et la réutilisation des systèmes. Cette étude nous a permis par ailleurs de définir une nouvelle classification des 23 patrons du GoF que nous considérons.

Si l'approche Aspect permet d'améliorer l'implémentation des patrons de conception par objets, aucune structure de patron indépendante d'un langage de programmation par aspects n'a été cependant proposée dans le cadre de cette contribution initiale, ainsi que dans le cadre de plusieurs autres travaux traitant également de la réalisation des patrons objets à l'aide de l'approche Aspect. Nous pensons que ceci contredit l'objectif fondamental des patrons qui proposent des solutions génériques pouvant être réutilisées et appliquées dans plusieurs techniques. En effet, un certain manque de consensus sur les concepts et mécanismes fondamentaux de l'approche Aspect et la diversité des modèles et langages de programmation proposés dans ce courant rendent difficile l'élaboration de telles structures. Rares sont donc les travaux qui proposent d'intégrer les aspects au niveau de la phase de conception, d'autant plus que les langages de modélisation qu'ils proposent sont généralement spécifiques à un modèle ou un langage de programmation par aspects particulier. Ainsi la deuxième contribution de notre thèse est de décrire un cadre conceptuel pour la modélisation de structures par aspects de patrons de conception, ou de toute autre conception dans un cadre plus général, indépendamment d'une technique de programmation par aspects particulière. Nous avons proposé pour ce faire une approche par métamodélisation et transformation de modèles. Nous avons défini trois extensions du métamodèle d'UML intégrant des éléments permettant d'exprimer des modélisations par aspects, et proposé des syntaxes concrètes pour ces extensions pour la représentation graphique de telles modélisations. La première de ces extensions correspond à un métamodèle général pour la modélisation par aspects, alors que les deux suivantes sont chacune respectivement spécifiques à deux des langages de programmation par aspects les plus connus, AspectJ et Hyper/J. Nous avons défini et proposé ensuite des règles de transformation de modèles, permettant de produire à partir d'un modèle instance du métamodèle général un modèle instance de l'un des deux métamodèles spécifiques. Nous avons adopté un processus de transformation de modèles basé sur l'utilisation conjointe des approches de transformation par métamodèles et par annotations.

Nous avons utilisé le métamodèle général que nous proposons pour l'expression de nouvelles structures par aspects des patrons du GoF. L'analyse des structures que nous avons obtenues à partir, d'une part, des solutions d'implémentation par aspects proposées dans cette thèse, et d'autre part, des solutions proposées par [Hannemann *et al.*, 02] pour l'ensemble des patrons du GoF, a confirmé de fortes analogies existant entre ces structures. Cette étude nous a permis d'identifier un ensemble de huit nouveaux patrons de conception par aspects par abstraction des analogies existant entre les structures considérées. Nous avons proposé ensuite une adaptation aux aspects d'un formalisme de description de patrons que nous avons utilisé pour la définition des huit nouveaux patrons tout en les mettant en relation, constituant ainsi un système de patrons de conception par aspects. Ce système de patrons constitue la troisième contribution significative de notre thèse, proposant un ensemble de patrons qui permet notamment l'exploitation des patrons du GoF dans un cadre de développement par aspects. Cet ensemble de patrons peut par ailleurs être utile dans l'évolution de toute conception par objets vers une conception par aspects. Une telle évolution permet de pallier les problèmes d'enchevêtrement et de dispersion du code concernant les systèmes Objet, afin de faciliter leur évolution et augmenter leur réutilisation.

Bilan du travail réalisé

En résumé, les principaux apports de notre travail sont les suivants.

- Nous proposons une évolution des 23 patrons de conception par objets de type produit dotée de nouvelles solutions par aspects, apportant des améliorations à la traçabilité de tels patrons jusqu'à la phase d'implémentation, et augmentant de même leur réutilisation. Il s'agit d'une nouvelle vision pour appliquer au mieux les savoirs capitalisés par ces patrons, et aboutir à la construction de systèmes de qualité qui sont faciles à faire évoluer et à réutiliser.

Cette pratique offre de nouvelles perspectives quant à la réingénierie de patrons utilisés dans la conception d'une application donnée.

- Nous proposons une nouvelle vision des 23 patrons de conception du GoF, considérés comme étant des préoccupations transversales apportant différents types d'adaptations sur les classes d'une application. Cette étude a permis une nouvelle classification de ces patrons selon trois axes : nature de la cible impactée, type d'impact, contexte dynamique d'altération.
- Nous contribuons à la compréhension approfondie à la fois des langages AspectJ et Hyper/J et leurs modèles respectifs de programmation, mais aussi de l'approche Aspect en général, en adoptant une approche descriptive. Il s'agit de la proposition, d'une part, de deux métamodèles spécifiques explicitant et formulant respectivement les principaux concepts et relations d'AspectJ et Hyper/J, et d'autre part, d'un métamodèle général qui identifie et définit un ensemble de concepts et relations fondamentaux à l'approche Aspect.
- Nous avons défini un cadre conceptuel permettant la modélisation et la représentation de structures par aspects indépendantes d'une technique de programmation particulière, pouvant être facilement transformées vers des structures exprimées et représentées dans des langages de modélisation spécifiques. Ce cadre est basé sur une approche de développement par transformation de modèles dite hybride, utilisant conjointement les techniques de transformation par annotations et par métamodèles.
- Nous avons établi un système de patrons pour l'ingénierie de systèmes d'information à base d'aspects. Ce système capitalise des savoirs spécifiques à la conception par aspects, mais aussi des savoirs mis en œuvre dans l'ingénierie de systèmes à base d'objets par des patrons de conception par objets déjà existants. L'utilisation de ces patrons Objet pose plusieurs problèmes et limites préjudiciables à l'évolution et la réutilisation des systèmes, les patrons nouvellement proposés permettent en revanche de réutiliser les savoirs déjà capitalisés par les patrons Objet tout en améliorant la qualité des logiciels produits. Ils permettent également de faire évoluer des conceptions objets vers des conceptions aspects, dans le but de faciliter leur évolution et augmenter leur réutilisation.
- Nous avons adapté aux aspects un formalisme de description de patrons Objet. Cette adaptation porte notamment sur la proposition d'un nouveau système de classification permettant la définition de manière formelle des intentions des patrons aspects.

Perspectives

Ce travail de recherche ouvre la voie vers plusieurs perspectives qui se situent sur deux plans : un plan d'approfondissement du travail proposé et un plan d'élargissement de notre domaine de recherche.

Pour ce qui est de l'approfondissement, une perspective immédiate de notre première contribution est la généralisation à d'autres langages de programmation par aspects. En effet, nous n'avons pour l'instant considéré que les langages AspectJ et Hyper/J, mais il est souhaitable d'étudier dans quelles mesures nos propositions sont applicables dans d'autres langages tels que AspectS [Hirschfeld 02], AspectC++ [Spinczyk *et al.*, 02], etc. Par ailleurs, il serait notamment intéressant d'élargir rapidement le champ des patrons considérés à plusieurs autres patrons de conception par objets déjà existants. Cette étude permettrait, d'une part, de faire évoluer ces patrons en tirant profit de l'approche Aspect, et de confirmer les apports de cette nouvelle approche.
De la même manière, l'évolution aux aspects d'autres patrons de conception permettrait d'apporter des extensions à notre système de patrons. Il serait intéressant en fait de raffiner dans un premier temps

l'ensemble des patrons que nous proposons, et d'étendre ensuite ces patrons par d'autres patrons venant en complément. Notons en effet qu'un patron n'est jamais vraiment achevé dans le sens ou il s'agit d'une forme déclarative d'une certaine expertise, qui peut s'affiner et se modifier avec le temps. Ainsi, comme nous l'avons déjà indiqué, les patrons proposés dans cette thèse nécessitent également un travail de validation et de réutilisation sur plusieurs applications à base d'aspects, afin d'être raffinés et confrontés à l'usage de la communauté de l'approche Aspect.
En ce qui concerne notre approche de modélisation par aspects, il convient tout d'abord d'affiner les règles de transformation de modèles que nous proposons en intégrant les annotations nécessaires pour les projections cibles. Une étude approfondie des approches de transformation de modèles par annotations est donc d'un grand intérêt. De plus, il serait notamment intéressant d'étendre les trois extensions du métamodèle d'UML que nous proposons en permettant de plus la modélisation de comportements. En effet, les patrons de conception considérés traitent en général de collaborations entre plusieurs classes en interaction, il serait intéressant donc de compléter la description de leurs solutions par des diagrammes dynamiques tels que les diagrammes de séquences par exemple. Enfin, comme nous l'avons déjà indiqué, nous avons basé notre travail de métamodélisation sur la version 1.5 d'UML. Une perspective immédiate de notre proposition est l'adaptation à UML 2.0. Une telle adaptation devrait nous permettre d'implémenter les trois métamodèles proposés et d'automatiser la transformation d'un modèle général vers des modèles spécifiques, offrant ainsi un premier prototype opérationnel.

L'extension de notre travail concerne notamment l'intégration dans notre approche de métamodélisation et transformation de modèles d'autres modèles et langages de programmation par aspects tels que AspectS [Hirschfeld 02], JAC [Pawlak 02], ou les filtres de composition [Aksit *et al.*, 92], par exemple. Cette intégration est d'un grand intérêt. Elle devrait évidemment permettre de se rapprocher d'un consensus mieux établi sur les concepts et mécanismes fondamentaux de l'approche Aspect, et d'étendre en conséquence le métamodèle général que nous proposons par raffinement ou par ajout de nouveaux concepts et relations venant en complément.

Pour ce qui est des patrons d'ingénierie, nos perspectives à long terme s'articulent notamment autour de deux axes :

- L'ingénierie de patrons spécifiques aux aspects, répondant à des problèmes récurrents posés par les limites de la conception par aspects qui sont encore mal connus, plutôt que par les seules limites de la conception par objets. En effet, les patrons par aspects proposés sont identifiés et définis exclusivement sur la base d'études de patrons Objet, l'identification de patrons spécifiques aux problèmes de conception par aspects reste donc un enjeu important. La découverte de tels patrons Aspect nécessite une démarche différente de celle que nous avons adoptée, telle que l'analyse détaillée et exhaustive de systèmes Aspect de taille conséquente, aujourd'hui encore difficile étant donné le faible nombre d'applications existantes.
- L'intégration des patrons proposés dans une démarche d'ingénierie des systèmes d'information à base de patrons. Un système de patrons de conception par aspects se doit, en effet, d'être le plus complet possible si l'on veut pouvoir s'en servir comme aide à la conception et au développement de systèmes d'information utilisant au mieux les nouvelles techniques de programmation par Aspect. Il convient donc d'identifier et de définir de nouveaux patrons de conception de type processus étendant notre système, afin de définir un système de patrons plus complet contenant, d'une part, des solutions génériques à des problèmes de conception, et d'autre part, des solutions génériques de guidage de l'ingénierie de systèmes.

Les enjeux à venir en matière de systèmes d'information sont plus en termes de démarches que de modèles de solutions.

Bibbliographie

[Abiteboul *et al.*, 91] S. Abiteboul and A.J. Bonner — *Objects and Views*. In Proceedings of the ACM SIGMOD International Conference on Management of Data, 1991.

[Adams *et al.*, 95] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve and K. Nicodemus — *Fault-Tolerent Telecommunication System Patterns*. AT&T Bell Laboratories, 1995.

[Affer 95] J. Mc Affer — *Meta-Level Programming with CodA*. In Proceedings of 9th European Conference on Object-Oriented Programming (ECOOP'95), Lecture Notes in Computer Science, Vol. 952, pages 190-214, Springer-Verlag, Aarhus, Danemark, Août 1995.

[Aksit 89] M. Aksit — *On the Design of the Object-Oriented Language Sina*. Ph.D. Thesis, departement of computer science, University of Twente, the Nethetlands, 1989.

[Aksit *et al.*, 88] M. Aksit and A. Tripathi — *Data Abstraction Mechanisms in Sina/ST*. In Procceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'88), ACM Sigplan Notices, vol. 23, no. 11, pp. 265-275, 1988.

[Aksit *et al.*, 92] M. Aksit, L. Bergmans and S. Vural — *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*. In Proceedings of the ECOOP'92 Conference, LNCS 615, Springer-Verlag, 1992.

[Aksit *et al.*, 94] M. Aksit, J. Bosch, W. v.d. Sterren and L. Bergmans — *Real-Time Specification Inheritance Anomalies and Real-Time Filters*. In Proceedings of the ECOOP'94 Conference, LNCS 821, Springer-Verlag, pp. 386-407, July 1994.

[Aksit *et al.*, 96] M. Aksit, B. Tekinerdogan and L. Bergmans — *Achieving adaptability through separation and composition of concerns*. In Proceedings of the ECOOP Workshop on Adaptability in Object-Oriented Software Development, Linz, Austria, July 1996.

[Albin-Amiot *et al.*, 01] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien — *Instantiating and detecting design patterns: Putting bits and pieces together*. In proceedings of the 16th conference on Automated Software Engineering, pages 166–173. IEEE Computer Society Press, November 2001.

[Alcatel *et al.*, 03] Alcatel, Softeam, Thales and TNI-Valiosys — *Response to the MOF 2.0 Query/Views /Transformations*. Initial Submission, mars 2003.

[Aldawud *et al.*, 01] O. Aldawud, T. Elrad and A. Bader — *A UML Profile for Aspect-Oriented Modeling*. Workshop on Aspect-oriented Programming in OOPSLA 2001, Tampa Bay, Florida, USA, October 14-18, 2001.

[Aldawud *et al.*, 03] O. Aldawud, T. Elrad and A. Bader — *UML Profile for Aspect-Oriented Software Development*. 3rd International Workshop on Aspect-Oriented Modeling (in AOSD 2003), Boston, USA, Mars 2003.

[Alexander 79] C. Alexander — *The Timeless Way of Building*, Oxford University Press, New York, NY, 1979.

[Alexander *et al.*, 77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-king and S. Aangel — *A Pattern Language*. Oxford University Press, New York, NY, 1977.

[Ambler 97] S.W. Ambler — *An introduction Introduction to Process Pattern*. AmbySoft White paper, http://www.Ambysoft.com, 1998.

[Ambler 98] S.W. Ambler — *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[Antoniol *et al.*, 98] G. Antoniol, R. Fiutem and L. Cristoforetti — *Using Metrics to Identify Design Patterns in Object-Oriented Software*. In Proceedings of the Fifth International Symposium on Software Metrics (METRICS '98). IEEE Computer Society, pp. 23–34, 1998.

[Apple 89] Apple — *Macintosh Programmers Workshop Pascal 3.0 Referenc*e. Cupertino, California, 1989.

[Appleton 97] B. Appleton — *Pattern and Software: Essential Concepts and Technology.* http://www.enteract.com/~bradpp/docs/patterns-intro.html, 1997.

[Arévalo 05] G. Arévalo — *High Level Views in Object Oriented Systems using Formal Concept Analysis.* Ph.D. Thesis, University of Berne, January 2005.

[Arnaud *et al.*, 04] N. Arnaud, A. Front et D. Rieu — *Une approche par méta-modélisation pour l'imitation des patrons.* Inforsid'04, Biarritz, Mai 2004.

[AspectJ 02] AspectJ Team – *The AspectJ Programming Guide.* http://www.eclipse.org/aspectj/, 2002.

[Asplund *et al.*, 73] K. Asplund, G. Swift — *Downstream Water Volume.* http://www.designmatrix.com/pl/ecopl/dswv.html, 1973.

[Austin 04] C. Austin — *J2SE 1.5 in a Nutshell: the Source for Developers.* http://java.sun.com/developer/technicalArticles/releases/j2se15/, May 2004.

[Barbier 94] F. Barbier — *Traceability in object-oriented software life cycle.* In Proccedings of TOOLS EUROPE 94, Versailles, mars 1994.

[Bardou 98] D. Bardou — *Étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue.* Thèse de doctorat, Université Montpellier II, avril 1998.

[Bardou *et al.*, 96] Bardou D. and Dony C. — *Split Objects: a Disciplined Use of Delegation within Objects.* In Proceedings of OOPSLA'96, ACM Sigplan Notices, Vol. 31, N°10, p. 122-137, 1996.

[Basch *et al.*, 03] M. Basch and A. Sanchez — *Incorporating aspects into the UML.* 3rd International Workshop on Aspect-Oriented Modeling (in AOSD 2003), Boston, USA, Mars 2003.

[Baumgartner *et al.*, 96] G. Baumgartner, K. Läufer and V.F. Russo — *On the interaction of Object-oriented Design Patterns and Programming Languages.* Technical Report CSD-TR-96-020, Purdue University (USA), 1996.

[Beck 96] K. Beck — *Smalltalk Best Practice Pattern*s. Vol. 1: Coding, 1996.

[Beck 97] K. Beck — *Best Practice Patterns.* Vol. 1: Coding, Prentice Hall, NJ, 1997.

[Beck *et al.*, 87] K. Beck and W. Cunningham — *Using pattern languages for object-oriented programs.* Technical Report CR-87-43, Tektronix Inc. September 1987.

[Belaunde *et al.*, 02] M. Belaunde and M. Peltier — *From EDOC Components to CCMComponents: a precise mapping specification.* In FASE'2002 (Fundamental Approaches to Software Engineering), 2002.

[Bergmans 94] L. M. J. Bergmans — *Composing Concurrent Objects*, *Applying Composition filters for the Development and Reuse of Concurent Object-Oriented programs.* Ph.D. Thesis, departement of computer science, University of Twente, The Netherlands, 1994.

[Bézivin 01] J. Bézivin — *FromObject-Composition to Model-Transformation with MDA.* In proceedings of TOOLS'USA, Volume IEEE TOOLS-39, Santa Barbara, August 2001.

[Bézivin *et al.*, 02] J. Bézivin and S. Gérard — *A Preliminary Identification of MDA Components.* In OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture, Seattle, Etas-unis, Nov. 2002.

[Booch 97] G. Booch — *Ingénierie du logiciel avec ADA: de la conception à la realisation.* InterEditions, Décembre 1997.

[Borland 94] Borland — *A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5.* Scotts Valley, CA, 1994.

[Borne *et al.*, 99] I. Borne et N. Revault — *Comparaison d'outils de mise en oeuvre de design patterns.* L'Objet 5(1), 1999.

[Bosch 98] J. Bosch — *Design Patterns as Language Constructs.* Journal of Object-Oriented Programming (JOOP), p. 18-32, May 1998.

[Bosch, 96a] J. Bosch — *Relations as Object Model Component*s. Journal of Programming Languages, Vol. 4, 1996.

[Bosch, 96b] J. Bosch — *Langage Support for Design Patterns*. In Proccedings of *TOOLS Europe'9*6, 1996.

[Bracha *et al.*, 90] G. Bracha and W. Cook — *Mixin-Based Inheritance*. In Proceedings of OOPSLA 90, pp. 303-311, 1990.

[Brooks 95] F.P. Brooks — *The Mythical Man-Month, Essays on software architecture*. Addison-Wesley, anniversary edition, 1995.

[Brown *et al.*, 98] W.J. Brown, R. C. Malveau, W.H. Brown and T.J. Mowbray — *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998.

[Bruneton 04] E. Bruneton — *Developping with Fractal*. The ObjectWeb Consortium, http://fractal.objectweb.org/tutorial/, 2004.

[Bruneton *et al.*, 04a] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema and J.B. Stefani — *An Open Component Model and Its Support in Java*. In Proceedings of the International Symposium on Component-based Software Engineering, Edinburgh, Scotland, mai 2004.

[Bruneton *et al.*, 04b] E. Bruneton, T. Coupaye and J.B. Stefani — *The Fractal Component model specification*. Object Web Consortium, France Telecom and INRIA. http://fractal.objectweb.org, 5 février 2004.

[Budinsky et al., 96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides and P.S. Yu — *Automatic code generation from design patterns*. IBM Systems Journal 35(2), 1996.

[Burke 04] B. Burke, A. Chau, M. Fleury, A. Brock, A. Godwin and H. Gliebe — *Jboss Aspect Oriented Programming*. http://www.jboss.org/, février 2004.

[Buschmann 96] F. Buschmann — *What is a pattern?*. Object Expert, vol. 1, n°3, p.17-18, 1996.

[Buschmann *et al.*, 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal — *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley and Sons Ltd., New York, 1996.

[Castano *et al.*, 94] S. Castano, V.D. Antonellis, C. Francalanci and B. Pernici — *A reusability-based comparison of requirement specification methodologies*. In Methods and Associated Tools for the Information Systems Life Cycle (A-55), A. A. Verrijn-Stuart and T. W. Olle (Editors), Elsevier Science B. V. (North-Holland), 1994.

[Cauvet *et al.*, 01] C. Cauvet, D. Rieu, A. Front-Conte et P. Ramadour — *Réutilisation dans l'ingénierie des SI*. Chapitre 5 du livre *Ingénierie des SI*, Editions Hermès, pp. 115-147, 2001.

[Cauvet *et al.*, 99] C. Cauvet et F. Semmak — *La réutilisation dans l'ingénierie des systèmes d'information*. Chapitre 1du livre *Génie Objet, analyse et conception de l'évolution*, Edition Hermès, pp. 25-55 1999.

[CCM 02] Object Management Group (OMG) — *Corba Component Model (CCM) Specification 3.0*. http://www.omg.org/technology/documents/formal/components.htm, 2002.

[Chavez 04] C. Chavez — *A Model Driven Approach to Aspect-Oriented Design*. PhD thesis, PUC-Rio, Brazil, 2004.

[Chavez *et al.*, 02] C. Chavez and C. Lucena — *A Metamodel for Aspect-Oriented Modeling*. 1st International Workshop on Aspect-Oriented Modeling with UML in AOSD 2002, Enschede, the Netherlands, April 2002.

[Chavez *et al.*, 04] C. Chavez and C. Lucena — *Guidelines for Aspect-Oriented Design*. Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'04), Brasília. Anais do Primeiro, 2004.

[Chiba 95] S. Chiba — *A Metaobject Protocol for C++*. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Sigplan Notices vol.30, no.10, pp. 285-299, Octobre 1995.

[Clarke 01] S. Clarke — *Composition of Object-Oriented Software Design Models*. PhD Thesis, Dublin City University, January 2001.

[Clarke *et al.*, 01] S. Clarke and R.J. Walker — *Composition Patterns: An Approach to Designing Reusable Aspects*. In Proceedings of ICSE 2001, IEEE Computer Society, pp. 5-14, 2001.

[Coad 92] P. Coad — *Object-Oriented Patterns*. Communications of the ACM 35(9):152-159, 1992.

[Coad 95] P. Coad — *Object Models – strategies, patterns and applications*. Yourdon press computing series, 1995.

[Coad *et al.*, 97] P. Coad, D. North and M. Mayfield — *Object Models: Strategies, Patterns, and Applications*. Prentice Hall, 1997.

[Coady *et al.*, 01] Y. Coady, G. Kiczales, M. Feeley and G. Smolyn — *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code*. In FSE 2001.

[Cohen *et al.*, 04] T. Cohen and J. Yossi Gil — *AOP + J2EE: Towards an aspect based, programmable and extensible middlewareframework*. In M. Odersky editor, ECOOP 2004- Object-oriented Programming, 18th european Conference, Oslo, Norway, number 3086 in Lecture Notes in Computer Science, Springer-Verlag, June 2004.

[Cointe 87] P. Cointe — *Meta-classes are First Class : the ObjVlisp Model*. In Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), ACM Press, pp. 156-167, Orlando Floride, Octobre 1987.

[Conte *et al.*, 01a] A. Conte, M. Fredj, J.P. Giraudin et D. Rieu — *P-Sigma : un formalisme pour une représentation unifiée de patrons*. Inforsid'01, Genève, Mai 2001, pp. 67-86.

[Conte *et al.*, 01b] A. Conte, I. Hassine, J.P. Giraudin et D. Rieu — *AGAP : un Atelier de Gestion et d'Application de Patrons*. Inforsid'01, Genève, Mai 2001, pp. 142-159.

[Conte *et al.*, 01c] A. Conte, J.P. Giraudin, I. Hassine et D. Rieu — *Un environnement et un formalisme pour la définition, la gestion et l'application de patrons*. Revue ISI, numéro spécial *Formalismes et Modèles pour les Systèmes d'Informations*, Vol. 6, n° 2, Hermès, 2001.

[Coplien 96] J.O. Coplien — *Patterns, the Patterns White Paper*. 1996.

[Coplien 98] J.O. Coplien — *The patterns Handbook: Techniques, Strategies and Applications*. Cambridge University Press, New York, 1998.

[Coplien *et al.*, 95] Coplien, J.O. and D.C. Schmidt — *Pattern Languages of Program Desig*n. Addison-Wesley, Reading, MA, 1995.

[Corriveau 96] J.P CORRIVEAU — *Traceability Process for large OO Projects*. IEEE Computer, septembre 1996.

[Coupaye *et al.*, 03] T. Coupaye, E. Bruneton and J.B. Stefani — *The Fractal Composition Framework, Proposed Final Draft of Interface Specification version 0.9*. The ObjectWeb Consortium, Jun 2002.

[Demighiel *et al.*, 87] L. Demighiel and R. Gabriel — *The Common Lisp Object System: An overview*. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP 87), pp. 151-170, 1987.

[DeRemer *et al.*, 76] F. DeRemer and H. Kron — *Programming in the large versus programming in the small*. IEEE Transactions on software Engineering, SE-2(2) pp. 80-86, 1976.

[Diaz 02] T. M.D. Diaz — *Meta-Patterns: a new approach to design patterns*. http://www.moisesdaniel.com/wri/metapatterns.html, 2002.

[Dijkstra 76] E. W. Dijkstra — *A discipline of programming*. Prentice-Hall, 1976.

[Douence *et al.*, 04] R. Douence, P. Fradet and M. Südholt — *Trace-based aspects*. In M. Aksit, S. Clarke, T. Elrad and R. E. Filman (Editors), *Aspect-Oriented Software Developement*. Addition-Wesley, 2004.

[D'Souza *et al.*, 99] DF. D'Souza and AC.Wills — *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, Reading, MA, 1999.

[Eden *et al.*, 97] Eden, A.H., A. Yehudai and J. Gil — *Patterns of the Agenda*. In *LSDF97: Workshop in conjunction with ECOOP'9*7, 1997.

[EJB 03] Sun Microsystems — *Enterprise JavaBeans (EJB) Specification 2.1*. http://www.sun.com/, 2003.

[El-Boussaidi *et al.*, 04] G. El Boussaidi et H. Mili — *Les patrons de conception : représentation et mise en œuvre*. Laboratoire de recherche sur les technologies du commerce électronique (LATECE), Université du Québec à Montréal, Rapport technique, 2004.

[Elrad *et al.*, 01] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr and H. Ossher — *Discussing aspects of AOP*. Communication of the ACM, Vol.44 No. 10, October, 2001.

[Ettinger *et al.*, 04] E. Ettinger and M. Verbaere — *Untangling: a slice extraction refactoring*. In Proceeding of AOSD 04, pages 93-101, 2004.

[Fakih *et al.*, 04] H. Fakih et N. Bouraqadi — *Les aspects et les composants logiciels : Etude de cas avec le modèle de composant Fractal*. Dans actes de la 1ère journée francophone de développement de logiciels par aspects *(JFDLPA 2004)*, Paris, France, 14 Septembre 2004.

[Falkerhainer *et al.*, 89] B. Falkenhainer, K. Forbus and D. Gentner — *The Structure-Mapping Engine: Algorithm and Examples*. Artificial Intelligence 41, pp 1-63, 1989.

[Ferber 89] J. Ferber — *Computational reflection in class based object oriented languages*. In Proceedings of the 4th Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'89), volume 24 of Sigplan Notices, pages 317-326. ACM Press, October 1989.

[Filman *et al.*, 00] R.E. Filman and D.P. Friedman — *Aspect-Oriented Programming is Quantification and Obliviousness*. Workshop on Advanced Separation of Concerns (OOPSLA 2000), Octobre 2000.

[Finch 98] L. Finch — *So Much OO, So Little Reuse*. Dr. Dobb's Journal, http://www.ddj.com/articles/1998/9875/, mai 1998.

[Florijn *et al.*, 97] G. Florijn, M. Meijers and P.V. Winsen — *Tool support for object-oriented patterns*. 1997.

[Fowler 97] M. Fowler — *Analysis Patterns – Reusable Object Models*. Addison-Wesley, 1997.

[Fowler 97a] M. Fowler — *Role Patterns*. Proceedings of PLoP'97, 1997.

[France *et al.*, 04] R.B. France, D-K. Kim, S. Ghosh et E. Song — *A UML-Based Pattern Specification Technique*. IEEE Transactions on Software Engineering, vol. 30, No. 3, pp. 193-206, mars 2004.

[Front 97] A. Front — *Développement de systèmes d'information à l'aide de patrons – Applications aux bases de données actives*. Thèse de doctorat, Université Grenoble 1, décembre, 1997.

[Front *et al.*, 99] A. Front, J-P. Giraudin, D. Rieu et C. Saint-Marcel — *Réutilisation et patrons d'ingénierie*. Chapitre 4 du livre *Génie Objet - Analyse et Conception de l'Evolution*, Editions Hermès, 1999, pp. 91-136.

[Gamma 91] E. Gamma — *Object-Oriented Software Developments based on ET++: Design Patterns, Class Library, Tools*. PhD thesis, University of Zürich, 1991.

[Gamma *et al.*, 93] E. Gamma, R. Helm, R. Johnson and J. Vlissides — *Design Patterns: Abstraction an Reuse of Object-Oriented Design*. In Proceedings of *ECOOP'9*3, 1993.

[Gamma *et al.*, 95] E. Gamma, R. Helm, R. Johnson and J. Vlissides — *Design Patterns, Elements of reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[Garcia *et al.*, 02] A. Garcia, V. Silva, C. Chavez and C. Lucena — *Engineering Muti-Agent Systems with Patterns and Aspects*. Journal of the Brazilian Computer Society, 8(1):57-72, 2002.

[Garlan 00] D. Garlan — *Software architecture and object oriented systems*. In Proceedings of the IPSJ Object oriented symposium 2000. Tokyo, Japan, August 2000.

[Gauthier *et al.*, 70] R. Gauthier and S. Pont — *Designing Systems Programs*. Prentice-Hall, Englewood Cliffs, N.J., 1970.

[Gentner 83] D. Gentner — *Structure Mapping: A Theoretical Framework for Analogy*. Cognitive Science 5, 1983.

[Glandrup 95] M. Glandrup — *Extending C++ Using the Concepts of Composition Filters*. Master's Thesis, Departement of Computer Science, University of Twente, The Netherlands, November 1995.

[Goldberg *et al.*, 83] A. Goldberg and D. Robson — *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.

[Gottlob *et al.*, 96] G. Gottlob and M. Schrefl — *Extending Object-Oriented Systems with Roles*. ACM Transactions on Information Systems (TOIS), 1996.

[Groher *et al.*, 03] I. Groher and S. Schulze — *Generating aspect code from UML models*. 3rd International Workshop on Aspect-Oriented Modeling (in AOSD 2003), Boston, USA, Mars 2003.

[Grosz *et al.*, 96] G. Grosz, S. Si-Said et C. Rolland — *MENTOR : un environnement pour l'ingénierie des méthodes et des besoins*. Congrès INFORSID, Bordeaux, juin 1996.

[Guéhéneuc 03] Y.G. Guéhéneuc — *Un cadre pour la traçabilité des motifs de conception*. Thèse de doctorat de l'université de Nantes. 23 juin, 2003.

[Gzara 00] L. Gzara — *Les patrons pour l'ingénierie des Systèmes d'Information Produit*. Thèse de doctorat de l'INPG, spécialité Génie Industriel, Décembre 2000.

[Hachani 02] O. Hachani — *Apports de la programmation par aspects dans l'implémentation des patrons de conception par objets*. Rapport de DEA de l'UJF Grenoble, Juin 2002.

[Han *et al.*, 04a] Y. Han, G. Kniesel and A. B. Cremers — *A Metamodel for AspectJ*. Technical report IAI-TR-2004-3, ISSN 0944-8535, Computer Science Department III, University of Bonn, Germany, http://www.cs.uni-bonn.de/~gk/papers/IAI-TR-2004-3.pdf, October 2004.

[Han *et al.*, 04b] Y. Han, G. Kniesel and A. B. Cremers — *A meta model and modelling notation for AspectJ*. In the 5th Aspect-Oriented Modeling Workshop in conjunction with UML 2004, Lisbon, Portugal, October 11-15, 2004.

[Hanenberg *et al.*, 01] S. Hanenberg and R. Unland — *Concerning AOP and Inheritance*. Workshop Aspekt-Orientierung der GI-Fachgruppe 2.1.9, Paderborn, May, 2001.

[Hannemann *et al.,* 02] J. Hannemann and G. Kiczales — *Design Pattern Implementation in Java and AspectJ*. In Proceedings of OOPSLA 2002, ACM SIGPLAN Notices, Vol. 37, n° 11, p. 161-173, 2002.

[Harrison *et al.*, 02] W. Harrison, H. Ossher and P. Tarr — *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. IBM Research Report RC22685 (W0212-147) December 30, 2002.

[Heineman *et al.*, 01] G. Heineman and W. T.Councill — *Component-Based Software Engineering : Putting the Pieces Together*. Addison-Wesley, 2001.

[Herrmann 02] S. Herrmann — *Composable Designs with UFA*. 1st International Workshop on Aspect-Oriented Modeling with UML in AOSD 2002, Enschede, The Netherlands, April 2002.

[Herrmann *et al.*, 01] S. Herrmann and M. Mezini — *Combining composition styles in the evolvable language LAC*. In Proceedings of ASoC workshop at the 23nd ICSE, 2001.

[Hirschfeld 02] R. Hirschfeld — *AspectS – Aspect-oriented Programming with Squeak*. In Proceedings of the International Conference NetObjectDays (NODe 2002), LNCS, vol. 2591, Springer, pp. 213-232. Erfurt, Germany, October, 2002.

[Hirschfeld *et al.*, 03] R. Hirschfeld, R. Lämmel and M. Wagner — *Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration*. In the 3rd German Workshop on AOSD, p. 25-32, 2003.

[Holyoak *et al.*, 89] K. Holyoak and P. Thagard — *Analogical Mapping by Constraint Satisfaction*. Cognitive Science, pp 295-355, 1989.

[Hursch *et al.*, 95] W. Hursch and C. Lopes — *Separation of Concerns*. Rapport NU-CCS-95-03, College of Computer Science, Northeastern University, 1995.

[Jézéquel 96] J. M. Jézéquel — *Object-oriented software engineering with Eiffel*. Eiffel in practice. Addison-Wesley, 1996.

[Johnson 92] R.E. Johnson — *Documenting Frameworks using Patterns*. In Proccedings of *OOPSLA'92*, Vancouver BC, 1992.

[Johnson *et al.*, 92] R.E. Johnson, C. McConnell and J.M. Lake — *The RTL system: A framework for code optimisation*. In International Workshop on Code Generation, Dagstuhl, Germany, 1992.

[Kandé *et al.*, 02] M. M. Kandé, J. Kienzle and A. Strohmeier — *From AOP to UML – A Bottom-Up Approach*. 1st International Workshop on Aspect-Oriented Modeling with UML in AOSD 2002, Enschede, The Netherlands, April 2002.

[Keene 89] S. Keene – Object-oriented programming in Common Lisp: a Programmer's Guide to CLOS. Addison Wesley, 1989.

[Khammaci *et al.*, 05] T. Khammaci, A. Smeda and M. Oussalah — *Coexistence of architectural description and object oriented modelling*. Advance in Software Engineering and knowledge Engineering, World Scientific Publishing Co, 2005.

[Khayati 05] O. Khayati — *Modèles formels et outils génériques pour la gestion et la recherché de composants*. Thèse de doctorat de l'INPG, Décembre 2005.

[Kiczales *et al.*, 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold — *An Overview of AspectJ*. In Proceedings of ECOOP 2001, European Conference on Object-Oriented Programming, Budapest, Hungary, LNCS, vol. 2072, Springer, pp. 327-353, June 2001

[Kiczales *et al.*, 91] G. Kiczales, J. Rivières and D.G. Bobrow — *The art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, ISBN 0-262-11158-6, 1991.

[Kiczales *et al.*, 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier and J. Irwin — *Aspect-Oriented Programming*. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). LNCS vol.1241, Springer-Verlag, Juin 1997.

[Kim *et al.*, 95] J.J Kim and K.M. Benner — *A Design Patterns Experience: Lessons Learned and Tool Support*. Position Paper, Workshop on Patterns, ECOOP'95, Aarhus, Denmark, 1995.

[Kristensen *et al.*, 96] B. B. Kristensen and K. Osterbye — *Roles: Conceptual Abstraction Theory & Practical Language Issues*. Special Issue of Theory and Practice of Object Systems on Subjectivity in Object-Oriented Systems, 1996.

[Kulkarni *et al.*, 03] V. Kulkarni and S. Reddy — *Supporting Aspects in MDA*. Workshop on Software Model Engineering in the 6th International Conference UML 2003, October 2003.

[Larman 02] C. Larman — *UML et les designs patterns*. Campus press, 2002.

[Le Guennec *et al.*, 00] A. Le Guennec, G. Sunyé et J.M. Jézéquel — *Precise modeling of design patterns*. In Proceedings of UML 2000, volume 1939 of *LNCS*, pages 482-496. Springer Verlag, 2000.

[Lemesle 98] R. Lemesle — *Transformation rules based on meta-modeling*. In EDOC'98, San Diego, November 1998.

[Lieberherr 94] K. J. Lieberherr — *The Art of Growing Adaptive Object-Oriented Software*. PWS Publishing Compagny, Boston, 1995.

[Lieberherr 95] K. Lieberherr — *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. May 23, 1995.

[Lieberherr *et al.*, 01] K. Lieberherr, D. Orleans and J. Ovlinger — *Aspect-Oriented Programming with Adaptive Methods*. Communications of the ACM, 44(10), pp. 39-41, Octobre 2001.

[Lieberherr *et al.*, 89] K. Lieberherr, and I. Holland — *Assuring Good Style for Object-Oriented Programs*. In Proceedings of IEEE Software, pp. 38-48, 1989.

[Lieberherr *et al.*, 94] K. J. Lieberherr, I. Silva-Lepe and C. Xiao — *Adaptive object-oriented programming using graphbased customisation*. Communications of the ACM, 37(5):94-101, May 1994.

[Lieberherr *et al.*, 97] K. Lieberherr and D. Orleans — *Preventive Program Maintenance in Demeter/Java*. In Proceedings of the 19th International Conference on Software Engineering, *(ICSE'97)*, pp. 604-605, Boston, 1997.

[Lieberherr *et al.*, 99] K. Lieberherr, D. Lorenz, and M. Mezini — *Programming with aspectual components*. In Technical Report, Northeastern University, April 1999.

[Linton *et al.*, 89] M.A. Linton, J.M. Vlissides and P.R. Calder — *Composing user interfaces with Interviews*. In Computer, pp. 8-22, 1989.

[Lions *et al.*, 02] M.J. Lions, D. Simoneau, G. Pitette and I. Moussa — *Extending OpenTool/UML Using Metamodeling: An Aspect-Oriented Programming Case Study*. 2nd International Workshop on Aspect-Oriented Modeling with UML (UML 2002), September 2002.

[Lopes *et al.*, 94a] C. Lopes and K. J. Lieberherr — *Abstracting Process-to-Process Relations in concurrent Object-Oriented Applications*. In Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94), Lecture Notes in Computer Science Vol. 821, Springer-Verlag pp. 81-99, Bologne, Italie, Juillet 1994.

[Lopes *et al.*, 94b] C. Lopes and K. Lieberherr — *Generative Patterns*. In the 8th European Conference on Object-Oriented Programming (ECOOP'94) Workshop on Patterns, Bologne, Italie, Juillet 1994.

[Lorenz 98] D.H. Lorenz — *Visitor Beans: An Aspect-Oriented Pattern*. In ECOOP'98 Workshop on Aspect-Oriented Programming, 1998.

[Low 02] T. Low — *Designing, Modeling and Implementing a Toolkit for Aspect-Oriented Tracing (TAST)*. 1st International Workshop on Aspect-Oriented Modeling with UML in AOSD 2002, Enschede, The Netherlands, April 2002.

[Lowy 03] J. Lowy — *Programming .NET Components*. 1st edition, O'Reilly, USA, 2003.

[Maes 87] P. Maes — *Concepts and experiments in computationnal reflection*. In proceedings of the 2nd Conference on Object-Oriented Programming: Systems, Languages and applications (OOPSLA'87), volume 22 of SIGPLAN Notices, pages 147-155. ACM Press, December 1987.

[Martin *et al.*, 98] R. Martin, D. Riehle and F. Buschmann — *Pattern Languages of Program Design* 3. Addison-Wesley, Reading, MA, 1998. [Next, 1994] Next, *NEXTSTEP General reference: Release* 3. Vol. 1,2. Addison-Wesley, Reading, 1994.

[McDirmid *et al.*, 03] S. McDirmid and W.C. Hsieh — *Aspect-Oriented Programming with Jiazzi*. In Proceedings of AOSD 2003, 2003.

[McIlory 76] M.D. MCILORY — *Mass-produced software components*. Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering). In J.M Buxton, P. Naur, and B. Randell (Editors), pp. 88-89, 1976.

[Medvidovic *et al.*, 02] N. Medvidovic, D. Rosenblum and R. Taylor — *A Language and Environment for architecture based software development and evolution*. In Proceedings of the 21 Internationnal Conference on software Engineering, Los Angeles, CA, pp. 44-53, May 1999.

[Meijers, 96] M. Meijers — *Tool Support for Object-Oriented Design Patterns*. Utrecht University, Utrecht, 1996.

[Meijler *et al.*, 96] T.D. Meijler and R. Engel — *Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment*. In Preliminary EuroPLoP ConferenceProceedings, 1996.

[Meijler *et al.*, 97] T. Meijler and O. Nierstrasz — *Beyond objects : Components*. Cooperative Information Systems: Current Trends and Directions. Academic Press, 1997.

[Mellor 03] S. J. Mellor — *A Framework for Aspect-Oriented Modeling*. 4th International Workshop on Aspect-Oriented Modeling with UML in UML 2003, October 2003.

[Meyer 92] B. Meyer — *Eiffel : The Language*. Object-Oriented Series. Prentice Hall, 1992.

[Meyer 97] B. Meyer — *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[Mezini *et al.*, 03] M. Mezini and K. Ostermann — *Conquering aspects with Caesar*. In Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press, pp. 90–99, 2003.

[Monteiro 04] M.P. Monteiro — *Catalogue of Refactorings for AspectJ*. Technical Report UM-DI-GECSD-200402, Departamento de Informática Universidade do Minho, Portugal, December 2004.

[Moon 86] D. A. Moon — *Object-Oriented Programming with Flavors*. In Proceedings of the 1st ACM Conference on Object-Oriented Programming Languages ans Applications (OOPSLA'86). ACM SIGPLAN Notices, vol 21. no 11. pp. 1-8, 1986.

[Mordhorst *et al.*, 95] J. Mordhorst and W. van Dijk — *Composition Filters in Smalltalk*. Master's Thesis, Departement of Computer Science, University of Twente, The Netherlands, July 1995.

[Net 05] Microsoft .Net web site http://www.microsoft.com/net, 2005.

[Next 94] Addison-Wesley, Reading, MA — *NextStep General Reference: Release 3*, Volumes 1 and 2, 1994.

[Noble 98a] J. Noble — *Towards a Pattern Language for Object Oriented Design*. http://citeseer.nj.nec.com/47369.html, 1998.

[Noble 98b] J. Noble — *Classifying relationships between object-oriented design patterns*. In Australian Software Engineering Conference (ASWEC), 1998.

[Noda *et al.*, 01] N. Noda and T. Kishi — *Implementing Design Patterns Using Advanced Separation of Concerns*. In OOPSLA 2001 Workshop on ASoC in OOS, 2001.

[Nordberg 01a] M.E. Nordberg — *Aspect-Oriented Dependency Inversion*. In OOPSLA 2001 Workshop on ASoC in OOS, 2001.

[Nordberg 01b] M.E. Nordberg — *Aspect-Oriented Indirection - Beyond Object-Oriented Design Patterns*. In OOPSLA 2001 Workshop "Beyond Design: Patterns (mis)used", 2001.

[Noyé *et al.*, 05] J. Noyé, R. Douence et M. Südholt — *Composants et aspects*. Chapitre 6 du livre *Ingénierie des composants – Concepts, techniques et outils*, édition Vuibert, Paris, pp. 169-195, 2005.

[OMG-CCM 02] Object Management Group (OMG) — *CORBA Components, version 3.0*. http://www.omg.org/technology/documents/corba_spec_catalog.htm#CCM, June 2002.

[OMG-MDA 03] Object Management Group (OMG) — *Model Driven Architecture (MDA) guide v 1.0.1*. http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003.

[OMG-OCL 03] Object Management Group (OMG) — *UML 2.0 Object Constraint Language (OCL) Specification*, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, 2003.

[OMG-QVT 05] Object Management Group (OMG) — *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification*, http://www.omg.org/cgi-bin/doc?ptc/2005-11-01, 2005.

[OMG-UML 04] Object Management Group (OMG) — *Unified Modeling Language (UML) Specification 1.4.2*. http://www.omg.org/cgi-bin/doc?formal/04-07-02, 2004.

[Orleans *et al.*, 01] D. Orleans and K. Lieberherr — *DJ: Dynamic Adaptive Programming in Java*. In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Springer-Verlag, Kyoto, Japan, September 2001.

[Ossher *et al.*, 00] H. Ossher and P.L. Tarr — *Hyper/J™: Multi-dimensional separation of concerns for Java™*. In Proceedings of the ICSE 2000, International Conference on Software Engineering, Limerick, Ireland, June 2000.

[Ossher *et al.*, 93] H. Ossher and W. Harrison — *Subject-Oriented Programming (A Critique of Pure Objects)*. In Andreas Paepcke (Editor) Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93), pp. 411-428, washington, DC, USA, October 1993. Published as ACM SIGPLAN Notices 28(10).

[Ossher *et al.*, 94] H. Ossher, W. Harrison, F. Budinsky and I. Simmonds — *Subject-Oriented Programming: Supporting Decentralized Development of Objects*. In Proceedings of the 7th IBM Conference on Object-Oriented Technology, July 1994.

[Ossher *et al.*, 95] H. Ossher, M. Kaplan, W. Harrison, A. Katz and V. Kruskal — *Subject-Oriented Composition Rules*. In Proceedings of OOPSLA'95, 1995.

[Ossher *et al.*, 96] H. Ossher, M. Kaplan, A. Katz and W. Kruskal — *Specifying Subject-Oriented Composition*. In Theory and Practice of Object Systems, vol. 2, no. 3, 1996.

[Oussalah *et al.*, 05] M. Oussalah, T. Khammaci et A. Smeda — *Les composants : définitions et concepts de base*. Chapitre 1 du livre *Ingénierie des composants – Concepts, techniques et outils*, édition Vuibert, Paris, pp. 169-195, 2005.

[Paakki *et al.*, 00] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen and I. Verkamo — *Software Metrics by Architectural Pattern Mining*. In Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), pp. 325–332, 2000.

[Pagel *et al.*, 96] B.U. Pagel and M. Winter — *Towards Pattern-Based Tools*. In Proceedings of EuropLop, Munchen, 1996.

[Palsberg *et al.*, 95] J. Palsberg, C. Xiao and K. Lieberherr — *Efficient Implementation of Adaptive Software*. ACM Transactions on Programming Languages and Systems, 17(2):264-292, Mars 1995.

[ParcPlace 90] ParcPlace (Editor) — *ObjectWorks\Smalltalk Release 4 Users Guid*e. Mountain View, CA, 1990.

[Parnas 72] D. Parnas — *On the criteria to be used in decomposing systems in modules*. Communication of the ACM, 15(2): 1053-1058, 1972.

[Pawlak 02] R. Pawlack — *La programmation par aspects interactionnelle pour la construction d'applications à préoccupations multiples*. Thèse de doctorat, Conservatoire National des Arts et Métiers, Paris, 2002.

[Pawlak *et al.*, 01] R. Pawlak, L. Seinturier, L. Duchien and G. Florin — *JAC: A Flexible Framework for AOP in Java*. In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Springer-Verlag, Kyoto, Japan, Septembre 2001.

[Peltier 02] M. Peltier — *Transformation entre un profil UML et un méta-modèle MOF. Application du langage Mtrans*. In Proceedings of LMO'02, Janvier 2002.

[Peltier 03] M. Peltier — *Techniques de transformation de modèles basées sur la méta-modélisation*. Thèse de l'Université de Nantes, 2003.

[Pessemier *et al.*, 04a] N. Pessemier, L. Seinturier and L. Duchien — *Components, ADL & AOP: Towards a common approach*. Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP'04, June 2004.

[Pessemier *et al.*, 04b] N. Pessemier, L. Seinturier, L. Duchien and O. Barais — *Partage de composants Fractal pour l'AOP*. Dans actes de la 1ère journée francophone de développement de logiciels par aspects *(JFDLPA 2004)*, Paris, France, Septembre 14 2004.

[Phillipow *et al.*, 03] I. Phillipow, M. Riebisch and K. Boellert — *The Hyper/UML Approach for Feature Based Software Design*. 4th International Workshop on Aspect-Oriented Modeling with UML in UML 2003, October 2003.

[Pree 95] W. Pree — *Design patterns for object-oriented software development*. Addison-Wesley Publishing, ACM Press, 1995.

[Reenskaugh *et al.*, 92] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, 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), pp. 27-41 October 1992.

[Rieu *et al.*, 99] D. Rieu, J.P. Giraudin, C. Saint-Marcel et A. Front-Conte — *Des opérations et des relations pour les patrons de conception*. Inforsid'99, Toulon, Juin 1999.

[Rising 00] L. Rizing — *The Pattern Almanac 2000*. Software Pattern Series, Addison-Wesley, 2000.

[Schmidt 95] D.C. Schmidt — *Using Design Pattern to develop Reusable Object-Oriented Communication Software*. Communications of the ACM, vol. 38, n°10, 1995.

[Schmuker 87] K. J. Schmuker — *Object Oriented Programming for the Macintosh*. Addison-Wesley, pp. 271-300, 1987.

[Ségura-Devillechaise *et al.*, 03] M. Ségura-Devillechaise, J. M. Menaud, G. Muller and J. Lawall — *Web cache prefetching as an aspect: Towards a dynamic weaving based solution*. In proceedings of AOSD 03, pages 110-119, 2003.

[Semmak 98] F. Semmak — *Réutilisation de composants de domaine dans la conception des systèmes d'information*. Thèse de doctorat, Université Paris I, 1998.

[Shtern 04] V. Shtern — *On Object-Oriented Approach to program Modularization*. In Proceedings of Software Engineering and Applications, MIT Cambridge, USA, November 2004.

[Smaragdakis *et al.*, 98] Y. Smaragdakis and D. Batory — *Implementing Layered Designs with mixin Layers*. ECOOP 98, 1998.

[Snyder 86] A. Snyder — *Encapsulation and inheritance in object-oriented languages*. In Proceedings of OOPSLA'86, ACM SIGPLAN Notices, Vol. 21, n° 11, pp. 38-45, 1986.

[Soukup, 95] J. Soukup — *Implementing Pattern*s. In Pattern Languages of Program Design, J.O. Coplien and D.C. Schimidt (Editors), 1995.

[Spanoudakis *et al.*, 96] K. Spanoudakis and P. Constanpopoulos — *Elaborating analogies from conceptual models*. Internation Journal of Intelligent Systems, Vol. 11, No. 11, novembre 1996.

[Spinczyk *et al.*, 02] O. Spinczyk, A. Gal and W. Schröder-Preikschat — *AspectC++: An Aspect-Oriented Extension to C++*. In proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, February 18-21, 2002.

[Sriplakich 03]P. Sriplakich — *Techniques des transformations de modèles basées sur la méta-modélisation*. Mémoire de DEA. http://modfact.lip6.fr/ModFactWeb/qvt/SimpleTRL.pdf, Septembre 2003.

[Stein 02] D. Stein — *An Aspect-Oriented Design Model Based on AspectJ and UML*. Master Thesis, University of Essen, Germany, 2002.

[Stroustrup 97] B. Stroustrup — *The C++ Programming Language*. Addison-Wesley Publishing Co., 3rd edition, 1997.

[Sun-EJB 04] Sun Microsystems — *Entreprise JavaBeans specification, version 3.0*. Early draft. http://java.sun.com/products/ejb/docs.html, 2004.

[Sunyé, 99] G. Sunyé — *Génération de code à l'aide de patrons de conception*. In Langages et Modèles à Objets - LMO'99, Villefranche sur mer, 1999.

[Sunyé *et al.*, 00] G. Sunyé, A. Le Guennec et J.M. Jézéquel — *Design pattern application in UML*. In E. Bertino, editor, ECOOP'2000 proceedings, number 1850, pp. 44-62, Lecture Notes in Computer Science, Springer Verlag, Juin 2000.

[Suzuki *et al.*, 99] J. Suzuki and Y. Yamamoto — *Extending UML with Aspects: Aspect Support in the Design Phase*. ECOOP'99 Workshop on Aspect-Oriented Programming, 1999.

[Szyperski 02] C. Szyperski — *Component software beyond object-oriented programming*. Addison Wesley, États-Unis, 2002.

[Tarr *et al.*, 00] P.L. Tarr and H. Ossher — *Hyper/J™ User and Installation Manual.* http://www.research.ibm.com/hyperspace, 2000.

[Tarr *et al.*, 99] P.L. Tarr, H. Ossher, W. Harrison and S. Sutton — *N Degrees of Separation: Multi-Dimension Separation of Concerns*. In Proceedings of the ICSE 99, International Conference on Software Engineering, Los Angeles, CA, USA, pp. 107-119, May 1999.

[Tatsubori *et al.*, 00] M. Tatsubori, S. Chiba, M. O. Killijian and K. Itano — *OpenJava: A Class-Based Macro System for Java*. In Reflection and Software Engineering, sous la direction de Walter Cazzola, Robert J. Stroud, Francesco Tisato, pp.117-133, Lecture Notes in Computer Science 1826, Springer- Verlag, 2000.

[Turner *et al.*, 03] J. Turner and K. Bedell — *Strataus*. Edition CampusPress, Juin 2003.

[Vasseur 04] A. Vasseur — *Dynamic AOP and Runtime Weaving for Java - How does Aspect-Werkz Address it ?* . AOSD'04, 2004.

[Villalobos 03] J. Villalobos — *Fédération de composants : une architecture logicielle pour la composition par coordination*. Thèse en Informatique à l'Université Joseph Fourier (Grenoble), juillet 2003.

[Vlissides *et al.*, 88] J.M. Vlissides and M.A Linton — *Applying object oriented design to structures graphics*. In USENIX C++, Denver, CO, 1988.

[Vlissides *et al.*, 90] J.M. Vlissides and M.A. Linton — *Unidraw: A framework for building domain-specific graphical editors*. ACM Transactions on Information Systems 9(33): pp. 104-124, 1990.

[Vlissides *et al.*, 96] J.M. Vlissides, J.O. Coplien and N.L. Kerth — *Pattern Languages of Program Design* 2. Addison-Wesley, Reading, MA, 1996.

[Walker *et al.*, 03] D. Walker, S. Zdancewik and J. Ligatti — *A theory of aspects*. In proceedings of the 8th ACM SIGPLAN international conference on Functional programming, pages 127-139. ACM Press, 2003.

[Wartik *et al.*, 92] S. Wartik and R. Pietro-Diaz — *Criteria for comparing Domain Analysis Approaches*. International Journal of Software Engineering and Knowledge Engineering, pp 403-431, 1992.

[Web Hachani] http://www-lsr.imag.fr/Les.Personnes/Ouafa.Hachani/works.html

[Web Hannemann] http://www.cs.ubc.ca/~jan/AODPs/

[Web Sigma] http://www-lsr.imag.fr/Les.Groupes/sigma/

[Weinand *et al.*, 88] A. Weinand, E. Gamma and R. Marty — *ET++ An Object-Oriented Application Framework in C++*. In Proceedings of *OOPSLA'88*, 1988.

[Weinand *et al.*, 89] A. Weinand, E. Gamma and R. Marty — *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*. Structured Programming, pp. 63- 87, 1989.

[Wegner 87] P. Wegner — *Dimensions of object-based language design*. In Proceedings of the 2nd ACM Conference on OOPSLA'87 (Orlando, Florida, October 4-8, 1987). ACM SIGPLAN Notices, 22, 12, pp. 168-182, December 1987.

[Wirth 84] N. Wirth — *Programmer en Modula-2*. Traduit de l'anglais par J. André. Presses polytechniques romandes, Lausanne, 1984.

[Zakaria *et al.*, 02] A. A. Zakaria, H. Hosny and A. Zeid — A *UML Extension for Modeling Aspect-Oriented Systems*. 2nd International Workshop on Aspect-Oriented Modeling with UML (UML 2002), September 2002.

[Zimmer 95] W. Zimmer — *Relationships Between Design Patterns*. In Coplien, J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison-Wesley, pp. 345-364, 1995.

Annexe A

UML — Concepts de base

Cette annexe donne les syntaxes abstraites des principaux concepts d'UML que nous considérons comme briques de base pour la définition des trois métamodèles que nous proposons dans le chapitre 4.

Le paquetage *Foundation* du métamodèle d'UML définit principalement l'ensemble des éléments de modélisation nécessaires à la représentation de la structure statique des systèmes à concevoir. Il contient trois sous-paquetages (cf. Figure 6.1) : *Core*, *Data Types* et *Extension Mechanisms*. Nous proposons dans ce qui suit de donner les définitions précises de quelques uns, mais pas tous, les éléments des paquetages *Core* et *Data Types*, que nous considérons dans la définition des trois métamodèles que nous proposons. Nous rappelons également les éléments du paquetage *Extension Mechanisms* que nous utilisons principalement dans la définition des notations graphiques que nous proposons.

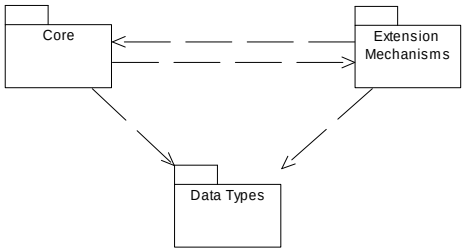


Figure 6.1. *UML Foundation Package*

1. UML Core package

Du paquetage *Core*, nous considérons notamment les éléments de modélisation définis par les modèles *Backbone*, *Classifier*, *Relationships* et *Dependencies*.

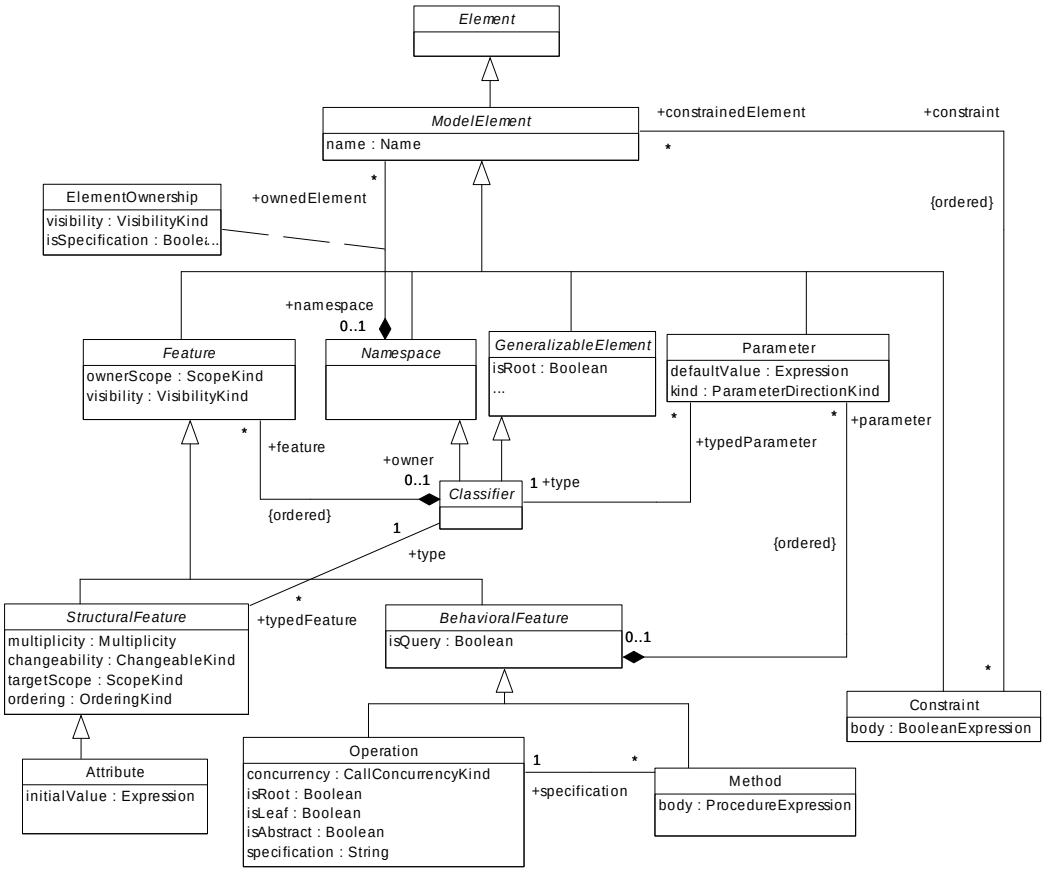


Figure 6.2. *UML Core Package — Backbone*

Un élément de construction fondamental en UML est le concept de classificateur (cf. figure 6.2). UML définie un classificateur (Classifier) comme étant un espace de nommage (NameSpace) dont les propriétés (Feature) possèdent des noms uniques. Un classificateur peut avoir des propriétés structurels (Attribut) et des propriétés comportementales (Operation et Method) qui peuvent admettrent à leur tour des paramètres typés (Parameter). UML définie différents types de classificateur (cf. figure 6.3). Nous distinguons et considérons principalement les Classes (Class), Interface (Interface) et les types de données (DataType). Une classe décrit un ensemble d'objets du monde réel qui sont caractérisés par un certain nombre de propriétés (attributs et opérations), alors qu'une interface décrit le comportement visible d'une classe qui est défini par une liste d'opérations ayant une visibilité publique. Les types de données sont décrit dans ce qui suit (cf. § 3).

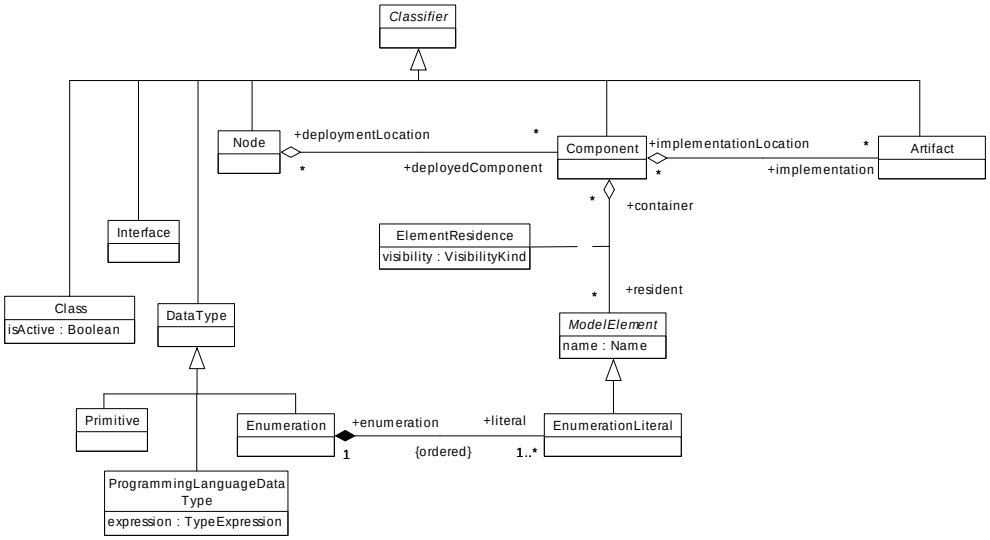


Figure 6.3. *UML Core Package — Classifiers*

UML définie par ailleurs différents types de relations (cf. figure 6.4) dont nous distinguons principalement la relation de généralisation (Generalization) et la relation d'association (Association).

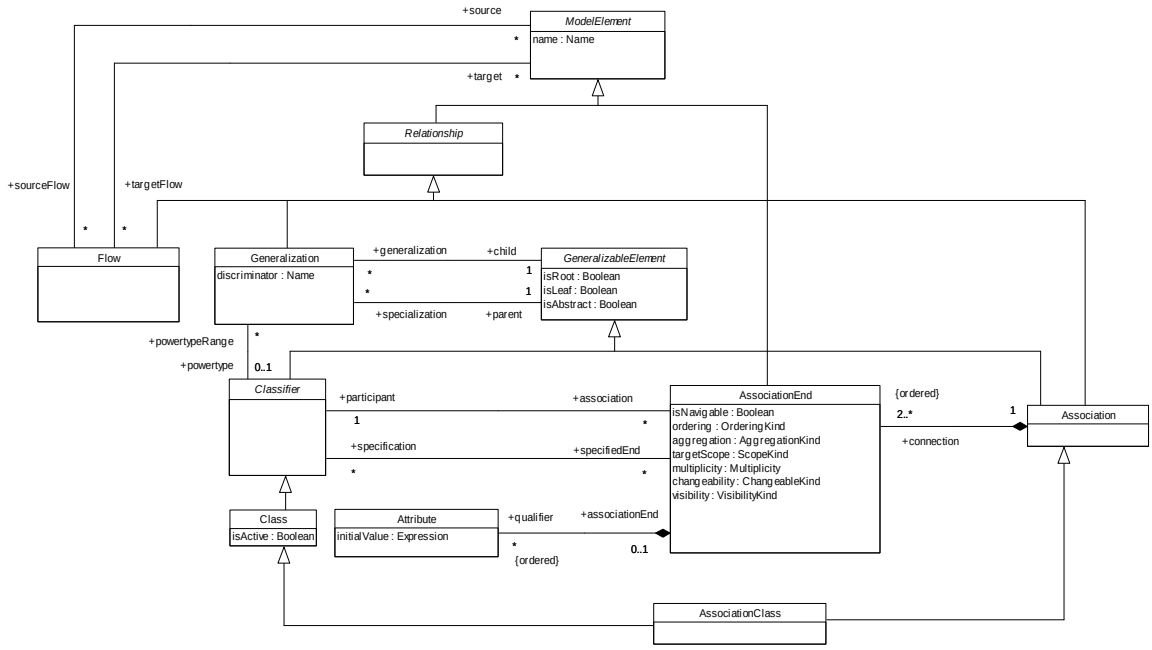


Figure 6.4. *UML Core Package — Relationships*

La relation de généralisation est la relation de classification entre un élément plus général et un élément plus spécifique. Par exemple, un animal est un concept plus général qu'un chat, un chien, etc. Inversement, un chat est un concept plus spécialisé qu'un animal. La généralisation s'applique à différent types d'éléments généralisables (GeneralizableElement), tels que les classes (définies comme étant un sous type particulier de Classifier), les paquetages, etc. Toute relation de généralisation entre classes indique, par exemple, que les propriétés d'une classe spécifique (child) sont aussi décrites par une autre classe plus générale (parent). Une classe plus spécifique peut cependant contenir des informations ou des propriétés qui lui sont propres.

Les associations représentent des relations structurelles sémantiques entre classes d'objets. Une association compte au moins deux extrémités d'association (AssociationEnd), reliées à des classificateurs (Classifier). La plupart des associations sont dites binaires car elles relient deux classes. Des associations n-aires peuvent cependant exister. Les associations se représentent en traçant une ligne entre les classes associées. Parmi les associations nous distinguons les agrégations et les compositions. Une agrégation représente une association non symétrique dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. Elle permet de modéliser une contrainte d'intégrité et de désigner l'agrégat comme garant de cette contrainte. L'agrégation se représente en ajoutant un petit losange vide du coté de l'agrégat. La composition est un cas particulier d'agrégation avec un couplage plus important. La composition indique, en plus des propriétés d'agrégation, une coïncidence des durées de vie des composants et du composite (la classe jouant le rôle dominant) : la destruction du composite implique automatiquement la destruction de tous ces composants. Elle se représente dans les diagrammes par un losange plein, placé du côté de la classe composite.

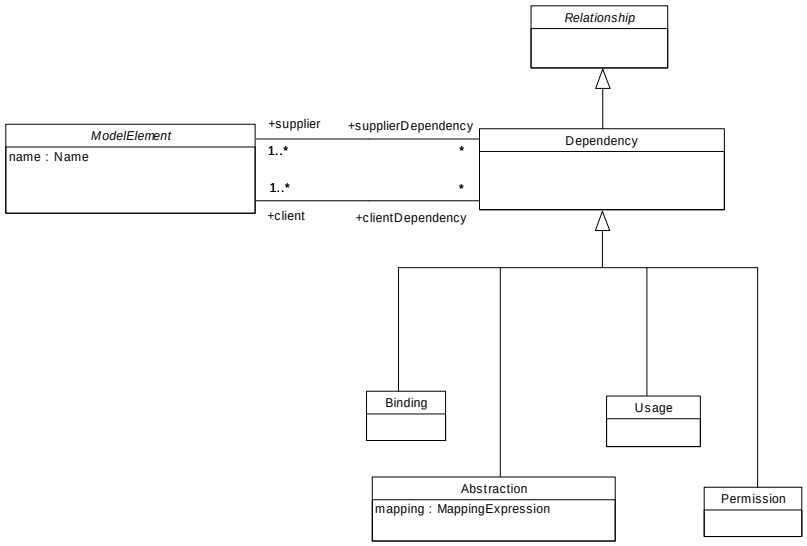


Figure 6.5. *UML Core Package — Dependencies*

En plus de ces deux relations, UML définit les relations de dépendance (cf. figure 6.5). Les relations de dépendance sont utilisées lorsqu'il existe une relation sémantique entre plusieurs éléments qui n'est pas de nature structurelle. Une relation de dépendance définit une relation unidirectionnelle entre un élément (ou des éléments) source(s) et un élément (ou des éléments) cible(s) ; elle indique qu'un changement au niveau de la cible implique un changement au niveau de la source. Les éléments source et cible sont aussi appelés respectivement client et fournisseur. UML définit quatre types particuliers de relation de dépendance : Binding, Abtraction, Usage et Permission. Nous considérons notamment ici la relation d'utilisation, il s'agit d'une relation de dépendance entre un élément source qui requiert la présence d'un autre élément

cible pour son bon fonctionnement ou son implémentation. UML distingue entre quatre relation d'utilisation stéréotypées : « call », « create », « instanciate » et « send » [OMG-UML 04]. Call représente une relation de dépendance entre une opération qui invoque une opération d'une autre classe. Create met en relation deux classificateur, elle indique que le classificateur source crée une instance du classificateur cible, alors qu'Instanciate indique qu'une instance du classificateur cible est créée par une opération donnée du classificateur source. Nous ne considérons pas le stéréotype « send » étant donné que nous nous concentrons seulement sur les éléments de représentation structurelle.

2. UML Extension Mechanisms package

UML définit un ensemble de mécanismes communs qui assurent l'intégrité conceptuelle de la notation. Parmi ces mécanismes nous distinguons principalement les mécanismes d'extension (cf. figure 6.6) : Stereotype, Constraint et TaggedValue (valeurs marquées) qui permettent l'extension et la spécialisation d'UML.

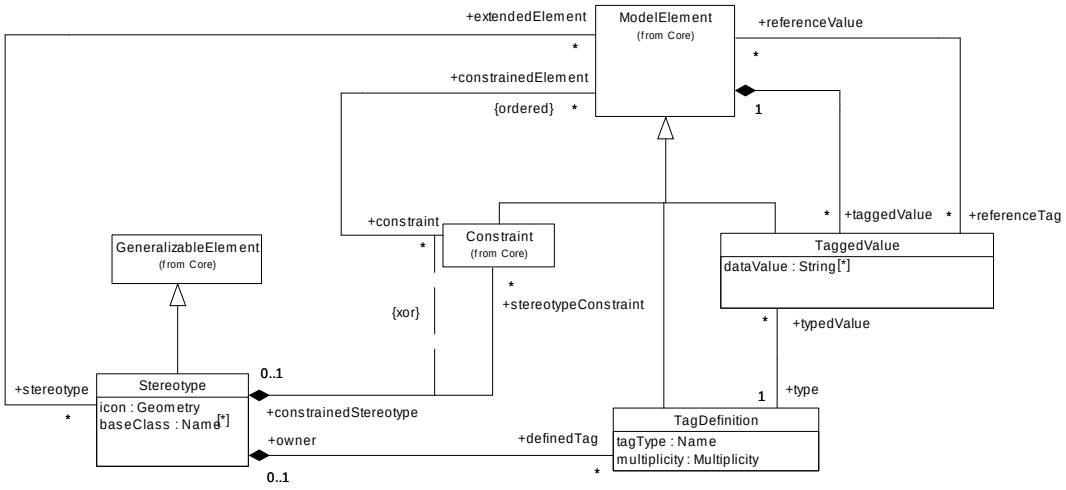


Figure 6.6. *UML Extension Mechanisms Package*

Les stéréotypes spécialisent les classes du métamodèle, les valeurs marquées étendent les attributs des classes du métamodèle et les contraintes sont des relations sémantiques entre éléments de modélisation qui définissent des conditions que doit vérifier le système.

3. UML DataTypes package

Le standard UML définit différents types de données définis par le paquetage *DataTypes*. Parmi les types de donnée nous distinguons (cf. figure 6.3) les types primitifs (Primitive), les énumérations (Enumeration) et les types de données spécifiques aux langages de programmation (ProgrammingLanguageDataType), mais aussi les expressions (Expression), etc. Integer et String sont des types primitifs par exemple, alors que Boolean est un stéréotype d'énumération. Parmi les expressions nous distinguons principalement les expressions de procédure (ProcedureExpression) et les expressions booléennes (BooleanExpression), par exemple.

Annexe B

Spécifications des nouveaux métamodèles

Cette annexe complète la description des syntaxes abstraites des trois métamodèles que nous proposons ; elle présente d'autres règles de bonne formation et précise la sémantique exacte de certains éléments de modélisation. Elle présente, par ailleurs, les syntaxes textuelles des langages AspectJ et Hyper/J.

Les langages de modélisation AspectJ/UML et HyperJ/UML, introduits dans le chapitre 4, permettent la modélisation de structures statiques des programmes écrits, respectivement, en AspectJ et Hyper/J. Comme nous l'avons déjà précisé dans le chapitre mentionné, pour définir ces deux langages nous avons adopté une approche « *bottom-up* » : nous avons commencé par identifier et examiner les notations utilisées pour définir les syntaxes textuelles des différents concepts clés des deux langages, pour définir ensuite leurs métamodèles spécifiques. Dans cette annexe nous présentons la syntaxe textuelle de chaque langage de programmation considéré. Nous complétons ensuite la description de la syntaxe abstraite des éléments de modélisation spécifiques aux trois métamodèles AspectJ/UML, HyperJ/UML et Aspect/UML, présentons d'autres règles de bonne formation complémentaires et précisons la sémantique de certains de ces éléments de modélisation.

1. AspectJ/UML

1.1 Syntaxe textuelle du langage AspectJ

Aspect. Un aspect est défini en AspectJ de la forme suivante :

```
aspect ::= <access> [abstract] [privilege] [static] aspect <identifier> <classIdentifier> <instantiation>
{ <aspectBody> }
<access> ::= [public|protected|private]
<identifier> ::= letter {letter|digit}
<classIdentifier> ::= [dominates]|[extends]|[implements]
<instantiation> ::= [issingleton | perthis |pertarget | percflow | perflowbelow]
<aspectBody> ::= {<aspectFeature>}
<aspectFeature> ::=  <features>|<introduction>| <declareParents>| <declarePrecedence>
                        |<pointcut>|<advice>|<declareSoft>|<customCompilation>
```

<features> représente une propriété classique : un attribut, une opération ou une méthode. Nous ne détaillons pas les syntaxes de ces éléments comme ils sont déjà définis en UML par la métaclasse Feature. En revanche, nous considérons dans ce qui suit le reste des propriétés possibles en AspectJ.

Introduction. AspectJ distingue quatre types d'introductions dont les syntaxes textuelles sont les suivantes :

- introduction d'une opération concrète ;

```
<modifiers><returnType><targetType>"."<id>"("[formals]")"[throwsClause]"{"<methodBody>"}"
<modifiers> ::= [public|protected|private]
<returnType> any legal AspectJ type.
<targetType> any legal AspectJ type. Defines the type to add the the feature to.
<id> ::= letter {letter | digits}
 formals ::= {formal (as java formal parameter) [","]}
 throwsClause (as Java throws clause)
 <methodBody> as java method body
```

- introduction d'une opération abstraite ;

```
abstract <modifiers><returnType><abstractTargetType>"."<id>"("[formals]")"[throwsClause] ";"
<abstractTargetType> any legal AspectJ type that must be an interface or an abstract class.
```

- introduction d'un constructeur ;

```
<modifiers><constructorTargetType>"." new "("[formals]")"[throwsClause]"{"<constructorBody>"}"
<constructorTargetType> any concrete or abstract class
```

– introduction d'un attribut.

<modifiers><typeOfField><targetType>"." <id> [fieldInitialization] ";"
<typeOfField> any any legal AspectJ type.
fieldInitialization ::= "=" Expression ";"

DeclareParents. AspectJ distingue deux types de déclarations de parenté dont les syntaxes textuelles sont les suivantes :

– déclaration d'héritage ;

declare parents ":" <typePattern> extends <typeList> ";"
<typePattern> AspectJ type pattern (Java class types)
<typeList> ::= Type {"," Type (Java class types)}

– déclaration de réalisation.

declare parents ":" <typePattern> implements <typeList> ";"

DeclarePrecedence. AspectJ permet de déclarer des précédences entre aspects de la forme :

declare precedence ":" <typePatternList> ";"
<typePatternList> ::= typePattern {"," typePatern (AspectJ aspect type)}

Pointcut. AspectJ définit deux types de points de recouvrement : nommés et anonymes dont les syntaxes textuelles sont les suivantes :

– points de recouvrement anonymes ;

<pointcut> ::= <designator>{"&&" | "||" <designator> }";"
<designator> ::= <designatorIdentifier> (<signaturePattern> | <typePattern> | <pointcut>| <namedPointcut>)
<designatorIdentifier> ::= call | execution | initialisation | handler | get | set | this | target | args | cflow | cflowbelow
| staticInitialization | within | withincode | if | adviceexecution | perinitialization
<signaturePattern> as java operation signatures

– points de recouvrement nommés.

namedPointcut ::= [<accesType>] pointcut <pointcutName> "(" [formals] ")" ":" <pointcut>
abstractNamedPointcut ::= abstract [<accesType>] pointcut <pointcutName> "(" [formals] ")" ";"
<accessType> ::= [public | protected | private]
<pointcutName> ::= letter {letter | digits}
formals ::= {formal (as java formal parameter) [","]}

Advice. La syntaxe textuelle des perfectionnements en AspectJ est de la forme suivante :

[returnType] <adviceType> "(" [formals] ")" [afterQualifier] [throwsClause] ":" <attachedpointcut>
"{"<adviceBody>"}"
returnType applies only to around advice
<adviceType> ::= before | after | around
afterQualifier ::= throwing | returning (applies only to after advice)
<attachedPointcut> ::= <pointcut> | <namedPointcut> | <abstractNamedPointcut>
<adviceBody> as a java method body (with a specific method: proceed())

Declare soft. La syntaxe textuelle des déclarations d'exception est la suivante :

declare soft ":" <exceptionType> ":" <staticallyDeteminablePointcut> ";"
<exceptionType> as Java exception type
<staticallyDeteminablePointcut> ::= <pointcut> | <namedPointcut> (pointcuts with designators this, target, args, cflow, cflowbelow and if are non static).

Custom Compilation. La syntaxe textuelle des déclarations d'erreur ou d'avertissement de compilation est la suivante :

declare [error | warning] ":"<staticallyDeteminablePointcut> ":" <message> ";"
<message> a Java string literal

1.2 Syntaxe abstraite et règles de bonne formation

L'analyse de la syntaxe textuelle du langage AspectJ nous a permis d'identifier un ensemble de concepts et relations fondamentaux que nous avons définis dans le métamodèle AspectJ/UML, et présenté partiellement dans le chapitre 4 de ce mémoire. Nous proposons dans ce qui suit de compléter la présentation de la spécification de ces concepts et relations tout en détaillant leurs propriétés, leurs associations et leurs règles de bonne formation.

Aspect

Règles de bonne formation

- Tous les points de recouvrement nommés d'un aspect concret doivent admettre des implémentations, de même que toutes ses opérations doivent avoir des méthodes.

```
context Aspect inv:
        (not self.isAbstract) implies (self.allNamedPointcuts->forAll (np |
         np.allPointcuts->exists( p | p.specification->includes(np)))
context Aspect inv:
        (not self.isAbstract) implies (self.allOperations->forAll (op |
         op.allMethods->exists( m | m.specification->includes(op)))
```

- Une association entre un aspect et une classe est uniquement navigable dans le sens aspect vers classe, et jamais dans le sens contraire.

```
context Aspect inv:
        self.association->forAll (ae | ae.isNavigable = #false)
```

- Un aspect de type singleton n'admet aucun point de recouvrement comme base.

```
context Aspect inv:
        (self.instantiation = "singleton") implies (self.base->isEmpty())
```

- Un point de recouvrement base d'un aspect est toujours anonyme.

```
context Aspect inv:
        (self.base.specification->isEmpty())
```

- Étant un espace de nommage, une classe peut contenir en plus des aspects.

```
context Class inv:
        self.allContents->forAll (c | c.oclIsKindOf(Aspect) or
                                      …)
```

- Un aspect spécifique à une classe donnée est obligatoirement statique et privé.

```
context Class inv:
        self.allContents->forAll (c | c.oclIsKindOf(Aspect))
                ->forAll(a | a.visibility = #private and a.isStatic=#true)
```

- Une classe ne peut jamais implémenter un aspect.

```
context Realization inv:
   self.client.oclIsTypeOf(Class) implies not (self.supplier.oclIsTypeOf(Aspect))
```

Opérations additionnelles

- L'opération allCrosscuttingFeatures donne l'ensemble des propriétés d'entrecroisement de l'aspect lui même et toutes ses propriétés d'entrecroisement héritées.

```
allCrosscuttingFeatures: Set(crosscuttingFeature);
allCrosscuttingFeatures = self.crosscuttingFeature->union (
                                self.parent.oclAsType(Aspect).allCrosscuttingFeatures)
```

- L'opération allCrosscuttingSpecifications donne l'ensemble des spécifications d'entrecroisement de l'aspect lui même et toutes ses spécifications d'entrecroisement héritées.

```
allCrosscuttingSpecifications : Set(crosscuttingSpecification);
allCrosscuttingSpecifications = self.crosscuttingSpecification->union (
                                self.parent.oclAsType(Aspect).allCrosscuttingSpecification)
```

- L'opération allNamedPointcuts donne l'ensemble des points de recouvrement nommés d'un aspect.

```
allNamedPointcuts : Set(namedPointcut);
allNamedPointcuts = self. allCrosscuttingSpecifications->select(np |
                                                np.oclIsKindOf(NamedPointcut))
```

CrosscuttingFeature

Les propriétés d'entrecroisement possibles en AspectJ sont représentées de manière abstraite par la métaclasse CrosscuttingFeature.

Association

owner Définit l'aspect contenant la propriété d'entrecroisement.

Règles de bonne formation

- Les propriétés d'entrecroisement sont anonymes.

```
context CrosscuttingFeature inv: self.name = ""
```

- Les propriétés d'entrecroisement n'ont pas de visibilité

```
context CrosscuttingFeature inv: self.visibility = #false
```

Introduction

Règles de bonne formation

- Une introduction peut ajouter des opérations concrètes dans des classes, des aspects et même des interfaces.

```
context Introduction inv:
    (self.introducedMember.oclIsTypeOf(Operation) and
     self.introducedMember.isAbstract = #false)
    implies  (self.targetType.oclIsKindOf(Class) or
                self.targetType.oclIsKindOf(Aspect) or
                self.targetType.oclIsKindOf(Interface))
```

- Une introduction peut ajouter des attributs à une ou plusieurs classes et aspects, mais pas à des interfaces.

```
context Introduction inv:
    (self.introducedMember.oclIsTypeOf(Attribut))
        implies  (self.targetType.oclIsKindOf(Class) or
                self.targetType.oclIsKindOf(Aspect))
```

Advice

Règles de bonne formation

- L'attribut afterQualifier d'un perfectionnement de type before ou around doit obligatoirement avoir la valeur par défaut undefined.

```
context Advice inv:
    (self.adviceType = #before or self.adviceType = #around)
    implies (self.afterQualifier = #aak_undefined)
```

- Les paramètres d'un perfectionnement doivent avoir des noms différents.

```
context Advice inv:
self.parameter->forAll( p1, p2 | p1.name = p2.name implies p1=p2)
```

CustomCompilation

Attributs

type	Indique s'il s'agit d'une déclaration d'erreur ou d'avertissement. Les valeurs possibles sont : - error. Indique qu'il s'agit d'un message d'erreur qui doit arrêter la compilation ; - warning. Indique qu'il s'agit tout simplement d'un message d'avertissement. La compilation n'est pas interrompue.
message	Spécifie le message à afficher par le compilateur.

Association

attachedPointcut	Spécifie le point de recouvrement auquel est attachée la déclaration d'erreur ou d'avertissement.

Règles de bonne formation

- Une déclaration d'erreur ou d'avertissement n'a pas de nom.

```
context CustomCompilation inv:
    self.name = ""
```

- L'attribut visibility hérité par CustomCompilation de sa super-classe Element n'a aucun sens dans le contexte d'une déclaration d'erreur ou d'avertissement.

```
context CustomCompilation inv:
    self.visibility = #false
```

Soft

Attribut

exceptionType	Spécifie le type de l'exception à traiter par la déclaration.

Association

attachedPointcut	Spécifie le point de recouvrement auquel est attachée la déclaration d'exception.

Règles de bonne formation

- L'attribut visibility hérité par Soft de sa super-classe Element n'a aucun sens dans le contexte d'un traitement d'exception.

```
context Soft inv: self.visibility = #false
```

– Un point de recouvrement attaché à une déclaration de type Soft doit être statiquement déterminable.

```
context soft inv: self.attachedPointcut.isStaticallyDeterminable() = #true
```

NamedPointcut

Attributs

isAbstract	Indique si le point de recouvrement nommé est abstrait (true) ou pas (false). La valeur par défaut est false.
isLeaf	Indique si le point de recouvrement nommé peut avoir (false) d'autres implémentations descendantes ou non (true). Dans ce dernier cas, son implémentation ne peut être remplacée par une autre dans le context d'un sous-aspect. La valeur par défaut est false.
isRoot	Spécifie si le point de recouvrement nommé est un super-point (i.e. ne peut avoir d'ascendants) (true) ou non (false). Dans le premier cas l'aspect contenant ce point de recouvrement ne peut hériter un point de recouvrement nommé de même nom. La valeur par défaut est false.
specification	Représente la signature du point de recouvrement nommé.

Règles de bonne formation

– Les paramètres d'un point de recouvrement nommé doivent avoir des noms différents.

```
context NamedPointcut inv:
    self.parameter->forAll( p1, p2 | p1.name = p2.name implies p1=p2)
```

Opération additionnelle

– L'opération allPointcuts donne l'ensemble des points de recouvrement anonymes implémentant le point de recouvrement nommé.

```
allPointcuts: Set(Pointcut);
allPointcuts = self.implementation
```

Pointcut

Opération additionnelle

– L'opération isStaticallyDeterminable détermine si le point de recouvrement est statiquement déterminable ou non (i.e. expose le context d'exécution). Il s'agit d'une opération abstraite.

```
isStaticallyDeterminable: Boolean;
```

Crosscut

AddFeature — *Règles de bonne formation*

– Une relation d'entrecroisement stéréotypée AddFeature admet un seul fournisseur de type Introduction.

```
context AddFeature inv:
    (self.supplier->size()=1) and (self.supplier.oclIsTypeOf(Introduction))
```

– Les clients d'une relation d'entrecroisement stéréotypée AddFeature peuvent être de type interface, classe ou aspect. De plus, selon qu'il s'agit d'une introduction d'un attribut, d'une opération concrète ou abstraite, d'autres contraintes sur ces clients existent.

```
context AddFeature inv:
    (self.supplier.introducedMember.oclIsTypeOf(Operation) and
                                    (self.supplier.introducedMember.isAbstract = #true)
    implies (self.client->forAll(c | c.oclIsKindOf(Interface) or
                                    c.oclIsKindOf(Class) and (c.isAbstract = #true) or
                                    c.oclIsKindOf(Aspect) and (c.isAbstract = #true))
    (self.supplier.introducedMember.oclIsTypeOf(Attribut))
    implies (self.client->forAll( c | c.oclIsKindOf(Class) or c.oclIsKindOf(Aspect))
    (self.supplier.introducedMember.oclIsTypeOf(Operation) and
                                    (self.supplier.introducedMember.isAbstract = #false)
    implies (self.client->forAll(c | c.oclIsKindOf(Interface) or
                                    c.oclIsKindOf(Class) or
                                    c.oclIsKindOf(Aspect))
```

AddParent — *Règles de bonne formation*

- Une relation d'entrecroisement stéréotypée AddParent admet un seul fournisseur de type DeclareParents.

```
context AddParent inv:
    (self.supplier->size()=1) and (self.supplier.oclIsTypeOf(DeclareParents))
```

- Les clients d'une relation d'entrecroisement stéréotypée AddParent peuvent être de type Generalization ou Realize (i.e. une relation d'abstraction stéréotypée).

```
context AddParent inv:
    (self.client.oclIsTypeOf(Generalization) or self.client.oclIsTypeOf(Abstraction))
```

Intercept — *Règles de bonne formation*

- Une relation d'entrecroisement stéréotypée intercept admet un seul fournisseur de type MethodBasedDesignator ou TypeBasedDesignator.

```
context Intercept inv:
   (self.supplier->size()=1) and (self.supplier.oclIsTypeOf(MethodBasedDesignator) or
                                    self.supplier.oclIsTypeOf(TypeBasedDesignator))
```

- Les clients d'une relation d'entrecroisement stéréotypée intercept peuvent être de type BehavioralFeature ou Class, selon le type du fournisseur de la relation.

```
context Intercept inv:
    (self.supplier.oclIsTypeOf(MethodBasedDesignator))
     implies (self.client.oclIsTypeOf(BeahvioralFeature))

    (self.supplier.oclIsTypeOf(TypeBasedDesignator))
     implies (self.client.oclIsTypeOf(Class))
```

Les types de données (cf. figure 4.9, chapitre 4)

Nous expliquons ici les types nouvellement introduits. InstanciationKind est le type de l'attribut instanciation d'un Aspect. L'attribut body d'un Advice est de type AdviceExpression. Il s'agit d'une nouvelle métaclasse spécialisant la métaclasse Expression du métamodèle d'UML. Par ailleurs, Les valeurs possibles de ses attributs adviceType et afterQualifier sont spécifiées respectivement, par les métaclasses AdviceKind et AfterAdviceKind. Les valeurs possibles de l'attribut type d'une CustomCompilation sont déterminées par la métaclasse CustomCompilationKind. Le type de l'attricbut expression d'un point de recouvrement (Pointcut) est spécifié par la matéclasse PointcutExpression. Le type de l'attribut identifier d'un Advice est déterminé par la matéclasse PointcutDesignatorKind.

2. HyperJ/UML

2.1 Syntaxe textuelle du langage Hyper/J

Les préoccupations tarnsversales et de base sont spécifiées et composées en Hyper/J grâce à trois fichiers : le fichier spécifiant l'hyperspace, le fichier organisant les différentes unités intégrables et le fichier correspondant à l'hypermodule qui précise les règles d'intégration. Nous présentons dans ce qui suit les syntaxes textuelles de ces trois fichiers, identifiant ainsi les principaux concepts du langage Hyper/J.

Spécification de l'hyperspace. L'hyperspace spécifie l'ensemble des classes Java à composer. La syntaxe textuelle utilisée pour définir un tel fichier est la suivante :

```
hyperspace <hyperspaceName>
   <classFileSpecification> ";"  { <classFileSpecification> ";"  }
<hyperspaceName> ::= letter {letter | digits}
<classFileSpecification> ::= [composable] class <classes> [expect <classes>] [including subclasses]
                           | composable file <classFiles>
                           | uncomposable class <classes>
<classes> ::= class { "," class (as Java class)}
```

Organisation des unités integrables. Les unités à intégrer doivent être organisées dans l'hyperspace considéré, et donc affectées à des préoccupations particulières. Cette affectation est définie grâce à un fichier dont la syntaxe textuelle est la suivante :

```
<unitMapping> ";" { <unitMapping> ";"}
<unitMapping> ::= <unitKind> <unitPattern> ":" <dimensionName>"."<hypersliceName>
<unitKind> ::= package | class | interface | operation |field
<unitPattern> as hyper/J integrables units (constructors and staticInitializers can be mapped to concerns, their names are <init> and <clinit>)
<dimensionName> ::= letter {letter | digits}
<hypersliceName> ::= letter {letter | digits}
```

Spécification de l'hypermodule de composition. Les règles composant deux ou plusieurs préoccupations (hyperslice) sont spécifiées dans un fichier représentant l'hypermodule. La syntaxe textuelle d'un tel fichier se présente comme suit :

```
hypermodule <hypermoduleName>
   hyperslices ":"
      <hyperslice> { "," <hyperslice> },<hyperslice> ;
   relationships ":"
      <generalStrategy> ";"
      [<integrationRules>]
end hypermodule ";"
<hypermoduleName> ::= letter {letter | digits}
<hyperslice> ::= <dimensionName>"."<hypersliceName>
<generalStrategy> ::= mergeByName | nonCorrespondingMerge | overrideByMerge
<integrationRules> ::= {<ruleKind> <unitKind> <ruleBody>}
<ruleKind> ::= equate | order | rename | merge | noMerge | override | match | bracket | summary
<unitKind> ::= hyperslice | class | interface | operation | action | field
<ruleBody> Syntaxes depend on the rules
```

Règles d'intégration. Chacune des règles d'intégration possibles en Hyper/J possède une syntaxe textuelle particulière. Nous présentons dans ce qui suit l'ensemble de ces syntaxes et proposons ensuite une synthèse des caractéristiques de l'ensemble des règles possibles.

– *Equate*

```
equate <unitKind> <unitName> {“,” <unitName>} “;”
<unitKind> ::= class | interface | operation | action | field
```

– *Order*

```
Order <unitKind> <unitName> {“,” <unitName>} <orderKind>
      <unitKind> <unitName> {“,” <unitName>} “;”
<unitKind> ::= hyperslice | class | interface | operation | action
<orderKind>  ::= befor  | after
```

– *Rename*

```
rename <unitKind> <unitName> to <newUnitName> “;”
<unitKind> ::= class | interface | operation | action | field
```

– *Merge*

```
merge <unitKind> <unitName> “,” <unitName>{“,” <unitName>} “;”
<unitKind> ::= class | interface | operation | action | field
```

– *NoMerge*

```
noMerge <unitKind> <unitName> “,” <unitName> {“,” <unitName>} “;”
<unitKind> ::= class | interface | operation | action | field
```

– *Override*

```
override <unitKind> <unitName> {“,” <unitName>}with
         <unitKind> <unitName> {“,” <unitName>}“;”
<unitKind> ::= class | interface | operation | action
```

– *Match*

```
match <unitKind> <unitName> with  <unitPattern> “;”
<unitKind> ::= class | interface | operation | action | field
```

– *Bracket*

```
bracket <classPattern>“.”<operationPattern>
         [ from  <unitKind> <unitName> {“,” <unitName>}] (Specify the call site)
         [ with ]
         [ before <fullyQualifierMethodName> “,”]
         [ after  <fullyQualifierMethodName>] “;”
<unitKind> ::= hyperslice | class  | operation | action
```

– *Summary function*

```
set summary function for <unitKind> <unitName> to <summaryFunction> “;”
<unitKind> ::= action | operation
<summaryFunction> ::= [external] <unitName> | <unitKind> <unitName>
```

2.2 Synthèse des règles d'intégration spécifiques

Afin de pouvoir abstraire et modéliser l'ensemble des règles d'intégration spécifiques possibles en Hyper/J, nous proposons dans ce qui de faire la synthèse de celles-ci en mettant en avant leurs propriétés caractéristiques (cf. tableau 7.1).

Tableau 7.1. *Synthèse des règles d'intégration offertes par Hyper/J*

Relationships	Exception to, specialization of		unit Kind	input units	output units	Constraints, optional specifications
	MergeByName, nonCorrespondingMerge	*Override*				
Equate	×	×	all*	2..*	0	- Equated units must be of the same type - OS: Can specifies an optional output name (*into* specification)
Match	×	×	all	1 – 1..* (2..*)	0	- Matching only occur for same-typed units
Rename	×	×	all	1	0	- It is not legal to rename units in the input hyperslices with this directive - Input element = Output element
NoMerge	×	×	all	2..*	0	- NoMerge only occur for same-typed units
Bracket	×	×	operation	1..* – 1..2 (2..*)	0	- Only apply to operation - OS: can include a *callsite* specification (all/Interface and field + hyperslice) - One before and/or one after specified methods
Merge		×	all	1..* – 1 2..*	1-primitive *-composite (1..*)	- Merged units must be of the same kind
Override	×		all/field	1 – 1..* (2..*)	1-primitive *-composite (1..*)	- Overridden and overriding units must be of the same type
Order	×	×	all/field + Hyperslice	2..*	0	- Ordered units must be of the same kind
Summary function	×	×	action| operation	1– 1 (2)	0	- The summary function must be static (same kind) - If the summary function is not external she must be in the hyperspace

* all := class|interface | operation | action |field

2.3 Syntaxe abstraite, règles de bonne formation et sémantique des éléments de modélisation d'HyperJ/UML

En partant de la syntaxe textuelle du langage Hyper/J, ainsi que de la synthèse des règles d'intégration ci-dessus, nous avons définit AspectJ/UML que nous avons commencé à présenter dans le chapitre 4 de ce mémoire. Nous propososn dans ce qui suit de compléter la présentation de la spécification des éléments de modélisation d'HyperJ/UML, en détaillant leurs propriétés, leurs associations et leurs règles de bonne formation. Nous présentons également la sémantique de quelque uns de ces éléments.

IntegrableUnit

Règle de bonne formation

- Dans le cadre d'un hyperspace donné, une unité intégrable peut appartenir à deux hyperslices à la fois à condition que les dimensions de ceux-ci soient differentes.

```
context IntegrableUnit inv:
        self.addressedHyperslice->forAll(h1, h2 | h1.owner = h2.owner implies h1 = h2)
```

Opération additionnelle

- L'opération unitKind retourne le type d'une unité integrable.

```
unitKind: oclType;
unitKind = if self.oclTypeOf(Class) then Class
                else if self.oclTypeOf(Interface) then Interface
                    else if self.oclTypeOf(Operation) then Operation
                        else if self.oclTypeOf(Method) then Method
                            else if self.oclTypeOf(Attribut) then Attribut
                                else Hyperlice
```

Package

Règle de bonne formation

- Dans le cadre d'un hyperspace donné, un paquetage peut appartenir à deux hyperslices à la fois à condition que les dimensions de ceux-ci soient differentes.

```
context Package inv:
        self.addressedConcern->forAll(h1, h2 | h1.owner = h2.owner implies h1 = h2)
```

Hyperslice

Règle de bonne formation

- Un hyperspace ne peut posséder ou référencier que des dimensions, des classes et des interfaces.

```
context Hyperspace inv:
        (self.allContents->forAll(c | c.oclIsKindOf(Dimension) or
                                      c.oclIsKindOf(Class) or
                                      c.oclIsKindOf(Interface))
```

Opération additionnelle

- L'opération allContents retourne l'ensemble des éléments contenus dans un hyperspace : ses dimensions ainsi que les classificateurs qu'il référencie.

```
allContents: Set(ModelElement);
allContents = self.ownedElement->union(self.composableClassifier)
```

Dimension

Règles de bonne formation

- Une dimension de préoccupations ne peut contenir que des hyperslices

```
context Dimension inv:
        (self.contents->forAll(c | c.oclIsKindOf(Hyperslice))
```

– Les hyperslices d'une dimension donnée ne peuvent avoir des unités intégrables ou des paquetages en commun.

```
context Dimension inv:
   (self.contents->forAll (c1, c2: Hyperslice |
                           c1.composableUnit->excludesAll(c2.composableUnit) and
                           c1.composablePackage->excludesAll(c2.composablePackage))
```

Opération additionnelle

– L'opération contents retourne l'ensemble des éléments détenus par une dimension.

```
contents: Set(ModelElement);
contents = self.ownedElement
```

Hyperslice

Règle de bonne formation

– Un hyperslice ne peut référencier que des paquetages, des classes, des interfaces, des opérations, des méthodes et des attributs.

```
context Hyperslice inv:
      (self.allContents->forAll(c | c.oclIsKindOf(Package) or
                                    c.oclIsKindOf(Class) or
                                    c.oclIsKindOf(Interface) or
                                    c.oclIsKindOf(Operation) or
                                    c.oclIsKindOf(Method) or
                                    c.oclIsKindOf(Attribut))
```

Opération additionnelle

– L'opération allContents retourne l'ensemble des éléments détenus par un hyperslice.

```
allContents: Set(ModelElement);
allContents = (self.ownedElement->union(self.composableUnit))
                                                ->union(self.composablePackage)
```

Hypermodule

Règle de bonne formation

– Un hypermodule ne peut contenir ou référencier que des règles d'intégration et des hyperslices.

```
context Hypermodule inv:
      (self.allContents->forAll(c | c.oclIsKindOf(IntegrationRule) or
                                    c.oclIsKindOf(Hyperslice))
```

Opération additionnelle

– L'opération allContents retourne l'ensemble des éléments détenus par un hypermodule.

```
allContents: Set(ModelElement);
allContents = self.ownedElement->union(self.composableHyperslice)
```

IntegrationRule

Associations

hypermodule	Spécifie l'hypermodule propriétaire de la règle d'intégration.
integrableUnit	Donne la liste ordonnée de l'ensemble des unités intégrables concernées par la règle d'intégration.

Règle de bonne formation

– Les unités intégrables liées par une règle d'intégration spécifique doivent être de même type.

```
context IntegrationRule inv:
        (self.integrableUnit->forAll(iu1, iu2 | iu1.unitKind = iu2.unitKind)
```

Opération additionnelle

– L'opération unitKind retourne le type des unités adressées par la règle d'intégration.

```
unitKind: oclType;
unitKind = self.integrableUnit->iterate (iu : IntegrableUnit;
                                         kind : oclType =ModelElement
                                         | kind = iu.unitKind)
```

Integration

Association

composedUnit	Donne la liste des unités intégrables résultats de la composition par la règle d'intégration considérée.

Règles de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type abstrait Integration peut être de type Class, Interface, Operation, Method ou Attribut.

```
context Integration inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method) or
                                                           iu.oclIsKindOf(Attribut))
```

– Le type des unités intégrables adressées par une règle d'intégration spécifique de type abstrait Integration, et celui des unités intégrables résultats de cette même règle sont identiques.

```
context Integration inv:
    (self.composedUnit->forAll(cu1, cu2 | cu1.unitKind = cu2.unitKind and
                                          cu1.unitKind = self.unitKind)
```

Override

Règles de bonne formation

– Les unités integrables adressées par une règle d'intégration de type Override correspondent à l'ensemble de ses unités remplacées et ses unités remplaçantes.

```
context Override inv:
        self.integrableUnit = self.overridenUnit->union(self.overridingUnit)
```

– Une unité intégrable adressée par une règle d'intégration de type Override peut être de type Class, Interface, Operation ou Method.

```
context Override inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method))
```

Sémantique détaillée

Bien que les relations de type Override soient conçues pour être appliquées à des méthodes, elles peuvent également être appliquées à d'autres types d'unités intégrables : classes, interfaces et opérations. Ainsi lorsqu'une classe remplace une autre, par exemple, toutes les méthodes de la première classe doivent remplacer les méthodes qui leurs correspondent dans la deuxième classe. Dans le cas d'opérations, toutes les méthodes implémentant la première opération doivent remplacer celles implémentant la deuxième opération. Enfin, si une interface remplace une autre, toutes les opérations de la dernière interface doivent être remplacées par celles de la première.

Order

Règles de bonne formation

- Une unité intégrable adressée par une règle d'intégration de type Order peut être de type Class, Interface, Operation, Method ou Hyperslice.

```
context Order inv:
    (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                       iu.oclIsKindOf(Interface) or
                                                       iu.oclIsKindOf(Operation) or
                                                       iu.oclIsKindOf(Method) or
                                                       iu.oclIsKindOf(Hyperslice))
```

- Les unités integrables adressées par une règle d'intégration de type Order correspondent à l'ensemble de ses orderedUnit et orderingUnit.

```
context Order inv:
    self.integrableUnit = self.orderedUnit->union(self.orderingUnit)
```

Sémantique détaillée

Bien que les relations de type Order soient conçues pour être appliquées à des méthodes, elles peuvent également être appliquées à d'autres types d'unités intégrables : classes, interfaces, opérations et hyperslices. Ainsi lorsque deux classes (ou deux hyperslices) sont liées (liés) par une relation Order, par exemple, toutes les méthodes de la dernière classe (du dernier hyperslice) doivent précéder (ou suivre, selon la valeur de l'atribut orderingType) les méthodes qui leurs correspondent dans la première classe (le premier hyperslice). D'une manière générale, une relation de type Order ne spécifie pas un ordre complet, surtout lorsqu'il s'agit de plusieurs méthodes qui sont mises en jeu à la fois. Hyper/J choisie un ordre d'exécution respectant au mieux toutes les relations de type Order si elles existent. Ainsi, si un ordre exact est nécessaire, il convient de définir autant de relations Order que nécessaires.

Summary

Règle de bonne formation

- Une unité intégrable adressée par une règle d'intégration de type Summary peut être de type Operation ou Method.

```
context Summary inv:
    (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Operation) or
                                                       iu.oclIsKindOf(Method))
```

Sémantique détaillée

Une relation Summary mettant en jeu une opération, s'applique en réalité sur l'ensemble des méthodes implémentant cette opération.

Match

Règles de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type Match peut être de type Class, Interface, Operation, Method ou Attribut.

```
context Match inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method) or
                                                           iu.oclIsKindOf(Attribut))
```

– Les unités integrables adressées par une règle d'intégration de type Match correspondent à l'ensemble de ses matchedUnit et matchingUnit.

```
context Match inv:
        self.integrableUnit = self.matchedUnit->union(self.matchingUnit)
```

Sémantique détaillée

Les relations de type Match servent notamment à définir des correspondances entre unités intégrables de noms différents. Les unités mises en jeu par de telles relations ne sont pas toutefois réellement intégrées, elles correspondent tout simplement et sont donc sujettes à la stratégie de composition générale.

Equate

Règle de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type Equate peut être de type Class, Interface, Operation, Method ou Attribut.

```
context Equate inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method) or
                                                           iu.oclIsKindOf(Attribut))
```

Sémantique détaillée

Comme dans le cas de la règle Match, les relations de type Equate servent aussi à définir des correspondances entre unités intégrables de noms différents. Ainsi, les unités mises en jeu par une relation Equate ne sont pas réellement intégrées, elles correspondent et sont sujettes à la stratégie de composition générale. Une relation Equate peut spécifier éventuellement une clause into, indiquant le nom qui doit être donné à l'élément composé résultat de l'intégration par la stratégie de composition générale. Dans le cas où aucun nom n'est spécifié, Hyper/J synthétise un nom à partir des différents noms des unités mises en jeu.

NoMerge

Règles de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type NoMerge peut être de type Class, Interface, Operation, Method ou Attribut.

```
context NoMerge inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method) or
                                                           iu.oclIsKindOf(Attribut))
```

– Les unités intégrables adressées par une règle d'intégration de type NoMerge soivetn avoir des noms identiques.

```
context NoMerge inv:
        (self.integrableUnit->forAll(iu1, iu2 | iu1.name = iu2.name))
```

Sémantique détaillée

Les relations de type NoMerge sont typiquement utilisées dans le cas où la stratégie de composition générale est mergeByName ou overrideByName, et que certaines unités intégrables de noms identiques ne doivent pas être intégrées.

Rename

Règles de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type Rename peut être de type Class, Interface, Operation, Method ou Attribut.

```
context Rename inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Interface) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method) or
                                                           iu.oclIsKindOf(Attribut))
```

– La règle d'integration Rename adresse exactement une seule unité intégrable

```
context Rename inv:
        (self.integrableUnit->size = 1)
```

Bracket

Règles de bonne formation

– Une unité intégrable adressée par une règle d'intégration de type Bracket peut être de type Class, Hyperslice, Operation ou Method.

```
context Bracket inv:
        (self.integrableUnit->forAll(iu : IntegrableUnit | iu.oclIsKindOf(Class) or
                                                           iu.oclIsKindOf(Hyperslice) or
                                                           iu.oclIsKindOf(Operation) or
                                                           iu.oclIsKindOf(Method))
```

– Les unités integrables adressées par une règle d'intégration de type Bracket correspondent à l'ensemble de ses barcketUnit, ses callSite, sa beforeMethod et sa afterMethod.

```
context Bracket inv:
  self.integrableUnit = ((self.bracketUnit->union(self.callSite))->union(self.beforeMethod))
                                ->union(self.afterMerhod)
```

3. Aspect/UML

Nous propososn dans ce qui suit de compléter la spécification des éléments de modélisation d'Aspect/UML, déjà présentée dans le chapitre 4. Nous détaillant les propriétés, les associations et présentons d'autres règles de bonne formation des éléments de modélisation de ce métamodèle général à l'approche Aspect.

Concern

Une préoccupation est un élément généralizable qui peut participer à des associations.

Associations

concernElement	Donne la liste ordonnée des éléments de la préoccupation. Il s'agit d'une relation de possession qui implique la destruction de tous les éléments d'une préoccupation donnée lorsuq'elle est détruite.
association	Donne les extrimités d'associations auxquelles participe la préoccupation transversale ou de base.

Règles de bonne formation

- Une association entre une préoccupation transversale et une préoccupation de base (i.e. classe) est uniquement navigable dans le sens préoccupation transversale vers classe, et jamais dans le sens contraire.

```
context Concern inv:
    self.association->forAll (ae | ae.participant.oclIsTypeOf(CrosscuttingConcern)
                                   implies ae.isNavigable = #false)
```

- Une classe ne peut pas implémenter une préoccupation transversale.

```
context Realization inv: self.client.oclIsTypeOf(Class) implies
                    not (self.supplier.oclIsTypeOf(CrosscuttingConcern))
```

- Une classe ne peut pas étendre une préoccupation transversale.

```
context Generalization inv: self.child.oclIsTypeOf(Class) implies
                    not (self.parent.oclIsTypeOf(CrosscuttingConcern))
```

CrosscuttingConcern

Attributs

isAbstract	Spécifie si la préoccupation transversale est abstraite (true) ou non (false).
isPrivileged	Indique si la préoccupation transversale à droit d'accéder (true) ou non (false) aux propriétés privées ou protégées des préoccupations de base ou transversales qu'elle impacte.

Associations

crosscuttingElement	Donne la liste ordonnée de l'ensemble des éléments d'entrecroisement d'une préocuupation transversale donnée.
crosscuttingSpecification	Donne la liste ordonnée des spécifications d'entrecroisement d'une préoccupation transversale donnée.
dominance	Donne l'ensemble des relations de précédences d'une préoccupation, dans lesquelles elle participe en tant que dominante.
submission	Détermine l'ensemble des relations de précédence d'une préoccupation, dans lesquelles elle participe en tant que dominée.

Règles de bonne formation

- Entant qu'un espace de nommage (*namespace*), une préoccupation transversale ne peut contenir que des classes, des associations, des généralisations, des dépendences, des types de données ou des interfaces.

```
context CrosscuttingConcern inv:
    self.allContents->forAll( c | c.oclIsKindOf(Class) or
                                  c.oclIsKindOf(Interface) or
                                  c.oclIsKindOf(Generalization) or
                                  c.oclIsKindOf(Association) or
                                  c.oclIsKindOf(Dependency) or
                                  c.oclIsKindOf(DataType))
```

- Les attributs d'une préoccupation transversale doivent avoir des noms différents.

```
context CrosscuttingConcern inv:
    (self.concernElement->select (a | a.oclIsKindOf(Attribute))->forAll(a1, a2 |
                                                    a1.nom = a2.nom implies a1=a2)
```

- Les propriétés comportementales d'une préoccupation transversale ne peuvent avoir des signatures identiques.

```
context CrosscuttingConcern inv:
    self.concernElement->select( f, g |
          ( (f.oclIsKindOf(Operation) and g.oclIsKindOf(Operation)) or
            (f.oclIsKindOf(Method) and g.oclIsKindOf(Method)) )
          and  (f.oclIsKindOf(BahavioralFeature).matchesSignature(g))
          implies f = g )
```

- Si une préoccupation transversale est non abstraite, toutes les spécifications d'entrecroisement de cette préoccupation doivent l'être également.

```
context CrosscuttingConcern inv:
    not self.isAbstract implies self.allCrosscuttingSpecifications->forAll(cc |
                                                    cc.isAbstract = #false)
```

Opérations additionnelles

- L'opération allContents détermine et retourne l'ensemble des éléments détenus par une préoccupation transversale donnée.

```
allContents: Set(ModelElement);
allCrosscuttingSpecifications = self.contents
```

- L'opération allCrosscuttingSpecifications retourne l'ensemble des spécifications d'entrecroisement d'une préoccupation transversale.

```
allCrosscuttingSpecifications: Set(CrosscuttingSpecification);
allCrosscuttingSpecifications = self.crosscuttingSpecification
```

Classifier

Règle de bonne formation

- Étant un type particulier de Concern, un Classifier ne peut avoir des éléments d'entrecroisement (CrosscuttingElement).

```
context Classifier inv:  self.concernElement->forAll(ce | ce.oclIsKindOf(Feature))
```

CrosscuttingElement

Associations

owner	Spécifie la préoccupation transversale propriétaire de l'élément d'entrecroisement considéré.
crosscutting	Donne l'ensemble des relations d'entrecroisement auxquelles participe ll'élément d'entrecroisement en question.

AdditiveElement

Association

target	Donne la liste ordonnée des préoccupations de base (i.e. classificateurs) concernées par l'élément considéré.

Règle de bonne formation

– Les cibles d'un élément d'addition ne peuvent être que des classes ou des interfaces.

```
context AdditiveElement inv:
    self.allTargets->forAll(c | c.oclIsKindOf(Class) or
                                c.oclIsKindOf(Interface))
```

Opération additionnelle

– L'opération allTargets détermine et retourne l'ensemble des classificateurs impactés par l'élément d'addition.

```
allTargets: Set(Classifier);
allTargets = self.target
```

Introduction

Opération additionnelle

– L'opération introducedFeature retourne la propriété destinée à être ajoutée.

```
introducedFeature: Feature;
introducedFeature = self.feature
```

ParentDeclaration

Opération additionnelle

– L'opération parent retourne la classe ou l'interface destinées à être, respectivement, étendue ou réalisée par les classificateurs cible d'impact de l'élément d'addition.

```
parent: Classifier;
parent = (self.generalization.parent)->union(self.abstraction.parent)
```

AlteringElement

Attributs

body	Spécifie le corps de l'élément d'altération.
type	Spécifie la manière dont l'élément d'altération doit altérer la ou les propriétés de comportement qu'il impacte à tarvers sa spécification d'entrecroisement. Les valeurs possibles sont : - before indique que le code de l'élément d'altération doit être exécuté avant celui de la ou des propriétés comportementales impactées par cet élément ; - after indique que le code de l'élément d'altération doit être exécuté juste après celui de la ou des propriétés comportementales impactées par cet élément ; - combination indique que le code de l'élément d'altération et celui de la ou des

propriétés comportementales destinées à être altérées doivent se combiner d'une façon ou d'une autre, pour définir le nouveau comportement attendu.

- replacement indique que le code de l'élément d'altération remplace celui relatif aux propriétés comportementales entrecroisées par cet élément.
- Narrowing indique que les propriétés comportementales destinées à être altérées ne peuvent être exécutées que sous conditions. De telles conditions sont spécifiées par le code de l'élément d'altération.

Association

specification	Désigne la spécification d'entrecroisement qui conditionne l'exécution de l'élément d'altération.

Opération additionnelle

– L'opération targetBehavioralFeatures retourne la liste des propriétés comportementales destinées à être altérées par l'élément d'altération.

```
targetBehavioralFeatures: Set(BehavioralFeature);
targetBehavioralFeatures = self.specification.allBehavioralFeatures
```

CrosscuttingSpecification

Règles de bonne formation

– Si une spécification d'entrecroisement est abstraite, elle n'est liée à aucune propriété de comportement.

```
context CrosscuttingSpecification inv:
    self.isAbstract implies self.allBehavioralFeatures->isEmpty
```

– Si une spécification d'entrecroisement est abstraite, son type est obligatoirement indéfini.

```
context CrosscuttingSpecification inv:
    self.isAbstract implies self.type = #undefined
```

Opération additionnelle

– L'opération allBehavioralFeatures retourne l'ensemble des propriétés de comportement impactées par une spécification d'entrecroisement donnée.

```
allBehavioralFeatures: Set(BeahvioralFeature);
allBehavioralFeatures = self.target
```

Add

Règle de bonne formation

– Les relations d'entrecroisement stéréotypées Add ne peuvent porter que sur des éléments d'entrecroisement de type AdditiveElement.

```
context Add inv:
    self.crosscuttingElement.oclIsKindOf(AdditiveElement)
```

Alter

Règle de bonne formation

– Les relations d'entrecroisement stéréotypées Alter ne peuvent porter que sur des éléments d'entrecroisement de type AlteringElement.

```
context Alter inv:
    self.crosscuttingElement.oclIsKindOf(AlteringElement)
```

Types de données

La figure 7.1 montre l'ensemble des métaclasses étendant le paquetage *Data Types* d'UML. Ces métaclasses spécifient les types de donnée utilisés dans la modélisation des concepts nouvellement introduits.

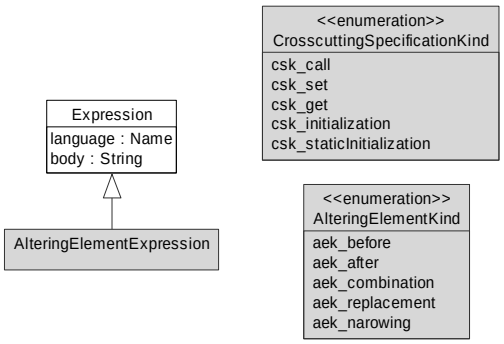


Figure 7.1. *Aspect/UML — Types de données*

AlteringElementExpression est le type de l'attribut body d'un élément d'altération, il s'agit d'une spécialisation de la métaclasse Expression d'UML. CrosscuttingSpecificationKind est un type énuméré définissant les différentes valeurs possibles de l'attribut type d'une spécification d'entrecroisement, alors que AlteringElementKind définit les différentes valeurs possibles de l'attribut type d'un élément d'altération.

Annexe C

Règles de transformation de modèles

Cette annexe propose d'autres règles de transformation de modèles complétant celles déjà introduites en chapitre 4. Elle explique par ailleurs la syntaxe textuelle utilisée pour écrire de telles règles.

Nous proposons dans cette annexe de présenter brièvement les notations utilisées, par le langage de relations de la spécification QVT, pour définir la syntaxe textuelle d'une transformation de modèles donnée. Nous présentons ensuite d'autres règles de transformation complétant celles déjà proposées dans le chapitre 4.

1. Notations utilisées dans la définition de la syntaxe textuelle des transformations

Une transformation entre deux modèles candidats est définie, dans le langage de relations (*Relations Language*) de la spécification QVT [OMG-QVT 05], comme étant un ensemble de :

- relations de correspondance entre les éléments constituant ces modèles ;
- déclarations de fonctions utiles pour la transformation des données des éléments mis en jeu ;
- spécifications d'un certain nombre de clés, par type d'éléments à créer, permettant d'identifier d'une façon unique un élément donné du modèle cible de la transformation.

Les deux modèles (source et cible) étant nommés et bien conformes à leurs métamodèles spécifiques, une déclaration de transformation d'un modèle aspectModel (instance de Aspect/UML) vers un modèle spécifique aspectJModel (instance d'AspectJ/UML) se présente par exemple comme suit :

```
transformation aspectModelAspectJModel (aspectModel: Aspect/UML, aspectJModel: AspectJ/UML)
{
  // spécification des clés par type d'éléments à créer
  // déclaration des relations de transformation
  // déclaration des fonctions
}
```

1.1 Notation des clés d'éléments à créer

Les éléments à créer dans le modèle cible sont identifiés, à travers leurs types spécifiques, par une ou plusieurs clés. Par exemple, une instance particulière d'Introduction appartenant au modèle cible aspectJModel est identifiée de façon unique par l'aspect qui la détient et par la propriété qu'elle introduit. Toute instance du type Introduction est ainsi déterminée par les valeurs de ces deux propriétés introducedFeature et owner. On note alors :

```
key Introduction(introducedFeature, owner) ;
```

Ainsi, si le modèle cible contient déjà une introduction conforme à une instance d'Introduction destinée à être créée, cette dernière ne sera jamais créée mais seulement complétée.

1.2 Notation des relations

Chaque relation est définie par deux ou plusieurs domaines (domain) et, éventuellement, une paire de prédicats when et where. Un domaine est une variable typée appartenant soit au modèle source, soit au modèle destination. Un domaine d'une relation peut être déclaré comme étant checkonly ou enforce. Un checkonly domain n'est pas destiné à être modifié, mais tout simplement à être vérifié. Contrairement, un enforce domain est destiné à être modifié. Une relation peut être de type top ou simple. Une top relation concerne la transformation d'un élément composite (tels qu'un paquetage, une classe ou un aspect). Elle utilise obligatoirement la clause where pour regrouper les relations transformant les composants (tes que les attributs et les opérations d'une classe) de l'élément composite qu'elle considère. Par ailleurs, une relation transformant un élément propre à un élément composite (un attribut d'une classe par exemple) utilise la clause when pour indiquer sa *top relation* correspondante. Nous donnons ci-dessous un exemple

d'une relation de type top transformant une préoccupation transversale (i.e. *un crosscutting concern*) vers un aspect.

```
top relation CrosscuttingConcernToAspect
{
    ccn, pn : string ;
    checkonly domain aspectModel cc : CrosscuttingConcern {namespace= p: Package {name = pn},
                                                           name = ccn};
     enforce domain aspectJModel a :Aspect {namespace= tp : Package {name = pn},
                                            name = ccn};
    when {
        PackageToPackage(p, tp);
    }
    where {
        IntroductionToIntroduction(cc, a) ;
        ParentDeclarationToDeclareParents(cc, a);
        CrosscuttingSpecificationToMethodRelatedDesignator(cc, a);
       //…
    }
}
```

Deux domaines sont déclarés dans cet exemple, représentant respectivement un élément du modèle source et son correspondant à créer dans le domaine cible. Il s'agit d'une préoccupation transversale de nom ccn appartenant à un espace de nommage (paquetage) p de nom pn, devant être transformés vers un paquetage tp et un aspect de mêmes noms (pn, respectivement, ccn). Remarquons ici que si le modèle destination aspectJModel contient déjà un paquetage de nom pn l'élément tp ne sera pas crée, il sera tout simplement augmenté par un nouveau aspect.

1.3 Notation des fonctions

Dans l'exemple précédent, le nom de l'aspect à créer est égal à celui de la préoccupation transversale à transformer. Il s'agit ici d'une transformation de données des éléments sources dite triviale et directe. Dans d'autres cas, le calcul des données des éléments à créer à partir de celles spécifiques aux éléments sources est plus complexe. Il convient alors de définir des fonctions permettant de déterminer de telles données. Il est nécessaire ainsi de définir, par exemple, une fonction calculant la valeur de la propriété adviceType d'un perfectionnement (i.e. *advice*) à partir de celle de la propriété type d'un élément d'altération. Nous présentons ci-dessous cette fonction.

```
function AlteringElementTypeToAdviceType (type: String) : String {
        if (type='before') then 'before'
        else if (type='after') then 'after'
             else 'around'
             endif
endif; }
```

2. Règles de transformation spécifiques à AspectJ

Nous proposons dans ce qui suit de compléter l'ensemble des règles de transformation que nous proposons pour une projection d'un modèle instance du métamodèle général Aspect/UML vers un modèle instance du métamodèle spécifique AspectJ/UML.

```
transformation aspectModelToAspectJModel (aspectModel : Aspect/UML, aspectJModel : AspectJ/UML)
{
        key Package(name) ;
        key Class(namespace, name);    //une classe est identifiée par son espace de nommage et son nom
        key Operation(specification, owner);  //une operation est identifiée par sa spécification et sa classe
        key Introduction(introducedFeature, owner);
        key DeclareParents(introducedGeneralization || introducedAbstraction, owner);
```

```
key NamedPointcut(specification, owner);
key MethodRelatedDesignator(identifier, owner);

top relation PackageToPackage { .. }
top relation ClassToClass { … }
top relation InterfaceToInterface { … }
top relation CrosscuttingConcernToAspect{ … }
relation AttributToAttribut{ … }
relation OperationToOperation{ … }
relation IntroductionToIntroduction{ … }
relation AttributeIntroductionToAttributIntroduction{ // target->size()=1 }
relation OperationIntroductionToOperationIntroduction{ // target->size()=1 }
```

```
relation ParentDeclarationToDeclareParents
{
   checkonly domain aspectModel cc:CrosscuttingConcern {};
   enforce domain aspectJModel a:Aspect {};
   where {
     GeneralizationDeclarationToDeclareGeneralization(cc, a);
     AbstractionDeclarationToDeclareAbstraction (cc,a);
   }
}
```

```
relation GeneralizationDeclarationToDeclareGeneralization
//declaration d'héritage entre deux classes (target->size()=1)
{
   ppn, pcn, cpn, ccn, pn, cn : String
   checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = pd :  ParentDeclaration
                                    { generalization = g : Generalization
                                        {parent = pc : Class { namespace = pp : Package {name = ppn}, name =pcn},
                                         child = cc : Class { namespace = cp : Package {name = cpn}, name =ccn}},
                                      target = c: Class {namespace = p : Package {name = pn}, name = cn }
                                    } };
    enforce domain aspectJModel a : Aspect
        { crosscuttingFeature = dp : DeclareParents
                                   { introducedGeneralization = ig : Generalization
                                      {parent = tpc : Class { namespace = tpp : Package {name = ppn}, name =pcn},
                                       child = tcc : Class { namespace = tcp : Package {name = cpn}, name =ccn}},
                                     targetType = tc : Class { namespace = tp : Package {name = pn}, name =cn}
                                   } };
    when {
      CrosscuttingConcernToAspect(cc, a);
     }
 }
```

```
relation AbstractionDeclarationToDeclareAbstraction
//declaration d'abstraction entre une classe et une interface (target->size()=1)
{
   ipn, sin, cpn, ccn, pn, cn : String
   checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = pd :  ParentDeclaration
                                    {abstraction = a : Abstraction
                                       {supplier = si: Interface { namespace = ip : Package {name = ipn}, name =sin},
                                        client = cc: Class { namespace = cp : Package {name = cpn}, name =ccn}},
                                     target = c: Class {namespace = p : Package {name = pn}, name = cn }
                                     }};
    enforce domain aspectJModel a : Aspect
        { crosscuttingFeature = dp : DeclareParents
                                   {introducedAbstraction = ia : Abstraction
                                       {supplier = tsi: Interface {namespace = tip: Package {name = ipn}, name =sin},
                                        client = tcc: Class {namespace = tcp : Package {name = cpn}, name =ccn}},
                                    targetType = tc : Class {namespace = tp : Package {name = pn}, name =cn}
                                   }} ;
    when {
      CrosscuttingConcernToAspect(cc, a);
```

```
    }
  }
```

```
    relation CrosscuttingSpecificationToMethodRelatedDesignator{ // target->size()=1 }
    relation AlteringElementToAdvice{ … }
    //…

    function AlteringElementTypeToAdviceType (type: String) : String { .. }
    //…
}
```

3. Règles de transformation spécifiques à Hyper/J

Nous proposons dans ce qui suit de compléter l'ensemble des règles de transformation que nous proposons pour une projection d'un modèle instance du métamodèle général Aspect/UML vers un modèle instance du métamodèle spécifique HyperJ/UML.

```
    transformation aspectModelToHyperJModel (aspectModel : Aspect/UML, hyperJModel : HyperJ/UML)
    {
        key Package(name) ;
        key Class(namespace, name) ;          //une classe est identifiée par son namespace et son nom
        key Operation(specification, owner) ;  //une operation est identifiée par sa spécification et sa classe
        key Hyperspace(name) ;
        key Dimension(owner, name) ;
        key Hyperslice(owner, name) ;

        top relation HyperspaceDimensionAndHypermodule{ … }
        top relation BaseClassifierToKernelHyperslice { … }
```

```
top relation ClassToClass {
   pn, cn: String;
   ckeckonly domain aspectModel c: Class {namespace = p: Package {name = pn}, name = cn};
   enforce domain hyperJModel tc: Class { namespace = tp: Package {name = pn},
                                          name = cn,
                                          hyperspace = hs: Hyperspace {}} ;
   when{
      BaseClassifierToKernelHyperslice(p, tp, hs)
   }
   where{
     //appels aux règles transformant les attributs, opérations, etc.
   }
}
```

```
        top relation CrosscuttingConcernToHyperslice{ … }
        relation IntroductionToFeature{ … }
        relation AttributIntroductionToAttribut{//target->size()=1}
```

```
 relation OperationIntroductionToOperation
 //introduction d'une operation dans une classe (target->size()=1)
 {
   os, ov, eb, pn, cn : String ;
   checkonly domain aspectModel cc: CrosscuttingConcern
       { crosscuttingElement = i : Introduction
                                { feature = o : Operation { specification = os,
                                                            isAbstract = 'false', visibility = ov,
                                                            method = m: Method
                                                                     {body = mb : ProcedureExpression {body = eb}}
                                                          },
                                  target = c: Class {namespace = p: Package {name = pn}, name =cn }
                                }
       };
   enforce domain hyperJModel tp: Package
       {  name = pn,
```

```
        ownedElement = tc: Class { name = cn,
                                   feature = to: Operation { specification = os,
                                                             isAbstract = 'false', visibility = ov,
                                                             method = m: Method
                                                             {body = mb : ProcedureExpression {body = eb}}
                                                           }
                                   }
      };
   when {
      CrosscuttingConcernToHyperslice(cc, tp);
   }
}
```

```
relation ParentDeclarationToRelationship
{
   checkonly domain aspectModel cc:CrosscuttingConcern { };
   enforce domain hyperJModel tp: Package {};
   where {
     GeneralizationDeclarationToGeneralization(cc, tp);
     AbstractionDeclarationToAbstraction (cc, tp);
   }
}
```

```
relation GeneralizationDeclarationToGeneralization
//declaration d'héritage entre deux classes (target->size()=1)
{
    ppn, pcn, cpn, ccn, pn, cn : String
    checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = pd :  ParentDeclaration
                    { generalization = g : Generalization
                            {parent = pc : Class { namespace = pp : Package {name = ppn}, name =pcn}},
                      target = c: Class {namespace = p : Package {name = pn}, name = cn } }
        };
    enforce domain hyperJModel tp: Package
        { name = pn,
          ownedElement = tc: Class
                        {name =cn,
                         generalization = tg: Generalization
                              {parent = tpc: Class {namespace = tpp: Package {name = ppn}, name =pcn}}
                        }
        };
     when {
       CrosscuttingConcernToHyperslice(cc, tp);
     }
 }
```

```
relation AbstractionDeclarationToAbstraction
{//declaration d'abstrcation entre une classe et une interface (target->size()=1)
    ipn, sin, cpn, ccn, pn, cn : String
   checkonly domain aspectModel cc: CrosscuttingConcern
        { crosscuttingElement = pd :  ParentDeclaration
                                {abstraction = a : Abstraction
                                   {supplier = si: Interface { namespace = ip : Package {name = ipn}, name =sin},
                                    client = cc: Class { namespace = cp : Package {name = cpn}, name =ccn}},
                                 target = c: Class {namespace = p : Package {name = pn}, name = cn }
                                }};
    enforce domain hyperJModel tp: Package
        { name = pn,
          ownedElement = tc: Class
                        { name =cn,
                          abstrcation = ta: Abstraction
                                { supplier = tsi: Interface {namespace = tip: Package {name = ipn}, name =sin}}
                         }};
     when {
       CrosscuttingConcernToHyperslice(cc, tp);
```

```
    }
  }
```

```
        relation AlteringElementToOperation{ … }
        //…

        function AlteringElementBodyToMethodBody(type: String) : String { … }
        //…
}
```

Annexe D

Le formalisme P-Sigma

Cette annexe présente la structure générale du formalisme de description de patrons P-Sigma.

Le formalisme P-Sigma est constitué de trois parties : « Interface », « Réalisation » et « Relations ». Chaque partie regroupe un certain nombre de rubriques. Chacune des rubriques est composée d'un ou de plusieurs champs typés. Les différentes rubriques et divers champs de P-Sigma peuvent être obligatoires ou optionnels. Nous proposons dans ce qui suit de présenter la structure générale de ce formalisme de description de patrons, sous la forme de trois diagrammes de classes.

1. Partie Interface

La figure 9.1 montre l'ensemble des rubriques et champs constituant la partie « Interface » de P-Sigma. Elle illustre également le caractère facultatif (dénoté par une multiplicité égale à 0..1) ou non de ces rubriques et champs.

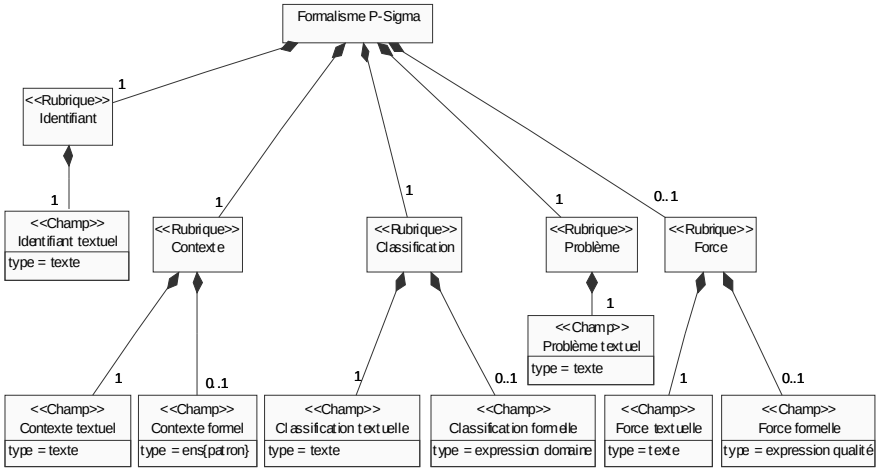


Figure 9.1. *Modèle de la partie « Interface » du formalisme P-Sigma*

2. Partie Réalisation

La figure 9.2 montre l'ensemble des rubriques et champs constituant la partie « Réalisation » de P-Sigma. Elle illustre également le caractère facultatif (dénoté par une multiplicité égale à 0..1) ou non de ces rubriques et champs.

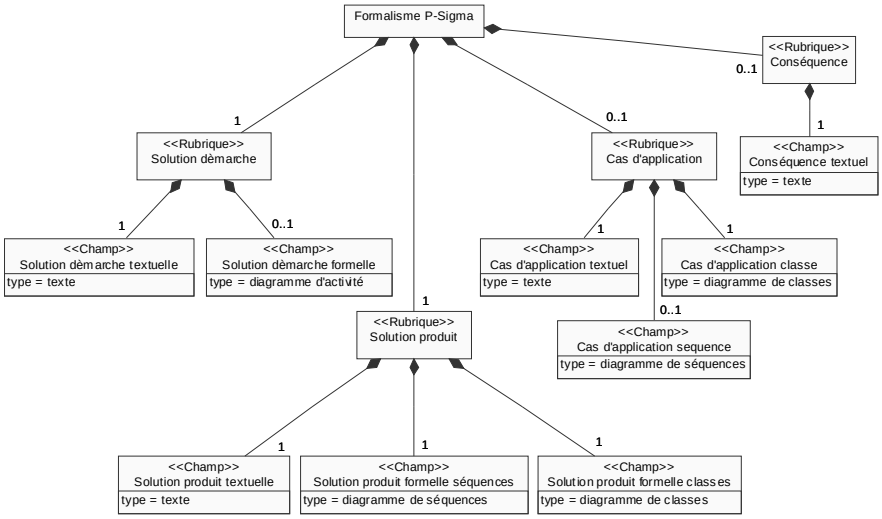


Figure 9.2. *Modèle de la partie « Réalisation » du formalisme P-Sigma*

3. Partie Relations

La figure 9.3 montre l'ensemble des rubriques et champs constituant la partie « Relations » de P-Sigma. Elle illustre également le caractère facultatif (dénoté par une multiplicité égale à 0..1) ou non de ces rubriques et champs.

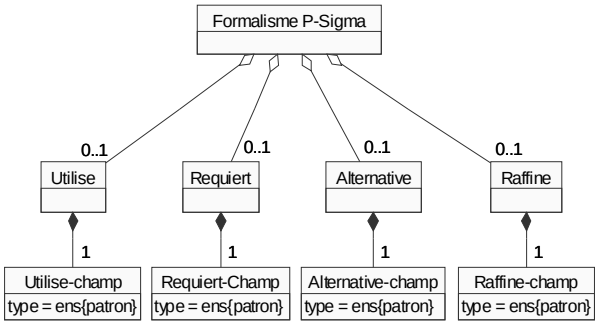


Figure 9.3. *Modèle de la partie « Relations » de P-Sigma*

Résumé

Ce travail de recherche concerne l'apport de l'approche Aspect à l'ingénierie des systèmes d'information (SI) en général et aux patrons de conception en particulier. L'objectif principal de ce travail est de développer des patrons à base d'aspects afin de faciliter et de guider l'ingénierie de SI par réutilisation de patrons.

Les patrons de conception par objets améliorent et accélèrent le développement en favorisant l'évolution, l'adaptation et la réutilisation de SI. Leur utilisation dans une approche strictement Objet pose cependant plusieurs problèmes et limites qui sont principalement liés à la dispersion et à l'enchevêtrement du code de leurs imitations dans l'implémentation des applications. L'approche Aspect permet de nouvelles solutions pour ces patrons contribuant à garder visible et isolée l'imitation de chaque patron dans le code des applications, afin de pallier à leurs problèmes d'utilisation et d'améliorer leur traçabilité et leur réutilisation.

Toutefois, un manque de consensus sur les concepts et mécanismes fondamentaux de l'approche Aspect et la diversité des modèles et langages de programmation proposés dans ce courant de recherche rendent difficile l'expression de structures par aspects de patrons indépendamment d'une technique de programmation par aspects particulière. Pour aborder cette difficulté, nous avons adopté une approche par métamodélisation et transformation de modèles. Nous avons basé cette approche sur un métamodèle général intégrant les particularités de l'approche Aspect et deux métamodèles spécifiques à AspectJ et Hyper/J. Les trois métamodèles proposés sont définis comme étant des extensions du métamodèle d'UML. Des règles de transformation sont également proposées.

Nous avons utilisé le métamodèle général pour l'expression de nouvelles solutions par aspects des patrons de conception par objets que nous considérons. Cette étude nous a permis de définir un système de huit nouveaux patrons originaux capitalisant des expertises en matière de conception par aspects. Les patrons proposés sont coordonnés et hiérarchisés ce qui permet d'offrir un cadre pour une démarche pour réaliser une conception et une programmation par aspects de qualité.

Mots clés

Systèmes d'information, réutilisation, patrons de conception, approche Aspect, métamodélisation, transformation de modèles, UML, AspectJ, Hyper/J.

Abstract

Our research work concerns the contribution of the Aspect approach to the engineering of the information systems (IS) in general, and to design patterns in particular. The main goal of this work is to develop aspect-oriented patterns in order to facilitate and to guide the engineering of IS by pattern reuse.

Object-oriented design patterns improve and accelerate the development while supporting SI evolution, adaptation and reuse. However, their use in a strict Object approach poses several problems and limits which are mainly related to the scattering and the tangling of the code of their imitations in the applications. The Aspect approach allows new solutions for these patterns that can keep visible and isolated the imitation of each pattern in the whole code of the applications, in order to mitigate their use problems and to improve their traceability and their reuse.

However due to a certain lack of consensus on what are the basic aspect-oriented concepts and mechanisms and the diversity of the suggested models and programming languages related to the Aspect approach, make it uneasy to express aspect-oriented pattern structures in a way that is not dependent from a specific aspect-oriented programming technique. To mitigate this difficulty we have adopted a meta-modeling and model transformation approach. We based this approach on a general meta-model which integrates particularities of the Aspect approach, and two meta-models that are respectively specific to AspectJ and Hyper/J. The three proposed meta-models are defined as extensions of the UML meta-model. Transformation rules are also proposed.

We used the general meta-model for expressing the aspect-oriented structures of the patterns that we consider in our work. This research led us to define a system of eight original patterns that capitalize expertises related to aspect-oriented design. The proposed patterns are coordinated and treated on a hierarchical basis what makes it possible to offer a method to carry out an aspect-oriented design and programs with good quality.

Keywords

Information systems, software reuse, design patterns, Aspect approach, meta-modeling, model transformation, UML, AspectJ, Hyper/J.