Rapport de DEA

Vers une généralisation du concept d'aspect

Gabriel Leblanc
leblanc@ensm-douai.fr

Encadré par :

Dr. Djamel SERIAI
GIP, École des mines de douai
941, rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France
seriai@ensm-douai.fr

Dr. Noury BOURAQADI
GIP, École des mines de douai
941, rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France
bouraqadi@ensm-douai.fr

École des Mines de Douai
Juin 2004

Résumé

La réutilisation est l'un des principaux objectifs de la programmation par objet. Cependant il n'est pas toujours possible de réutiliser du code existant dans de nouvelles applications notamment lorsqu'il s'agit d'associer sur celles-ci un ensemble de définition de services à des propriétés techniques représentant des mécanismes d'exécution spécifiques. Pour palier à ce type de problème, le paradigme de la programmation par aspect à été mis en place. Il considère que les services d'une application et ses propriétés techniques doivent être découplés. L'application est alors obtenu par composition des différents aspects. Mais la programmation par aspect n'est pas le seul concept utilisé pour modulariser l'application. Il existe au même titre que pour les aspects non fonctionnels des approches considérant le découpage au niveau des aspects "métier" (fonctionnels). Ce découpage est effectué en séparant les fonctionnalités spécifiques et communes du logiciel. L'application est ensuite créée en intégrant l'ensemble des différents modules fonctionnels. Ce type d'approche constitue le paradigme des modèles de représentation multiple. A partir de ces deux concepts, nous avons mis en place un modèle général capable d'intégrer à la fois des aspects fonctionnels et non fonctionnels. Dans ce rapport, nous décrivons l'ensemble de ces caractéristiques et en proposons une implantation en SMALLTALK.

Remerciement

Je tiens à remercier :

Noury Bouraqadi et Djamel Seriai, mes deux encadrants, pour leur accueil au sein de l'équipe, pour leur disponibilité, pour leurs conseils et tout ce qu'ils m'ont apportés au cours de ce stage, mais aussi pour m'avoir offert la possibilité de poursuivre mes travaux. Je remercie particulièrement Noury pour sa patience lors de nos longues séances de codage, et aussi pour m'avoir fait découvrir Smalltalk, ainsi que l'ensemble de la communauté Squeak.

Houssam Fakih et Abdelaziz Gacemi, les deux doctorants de l'équipe, pour leur sympathie leurs conseils avisés, et les longues discussions que nous avons eu ces derniers mois.

Ali Hamadi et Abdallah Sari, avec qui j'ai passé de bons moments.

L'ensemble du département Génie Informatique et Productique de l'école des mines de DOUAI pour leur accueil, et leur sympathie.

Table des matières

Première partie

Introduction

La programmation par objet, en partie grâce à ses nombreux avantages comme entre autre le polymorphisme ou encore la modularité, constitue une technologie importante pour la construction d'applications complexes. Elle favorise notamment la réutilisation de code et permet donc la réduction des délais et des coûts de développement d'applications. Néanmoins, elle n'est cependant pas exempte de défauts. En particulier, l'usage des objets conduit au mélange et à la dispersion des propriétés transversales des applications. De telles propriétés se trouvent mélangées avec le code métier et dispersées entre les différentes classes constituant une application donnée. De ce fait, le développement, la maintenance et l'évolution des applications se trouvent complexifiés.

En réponse à ce problème, la programmation par aspects [Kic96] a été introduite en 1996. Sans remettre en cause les fondations qui ont fait le succès de la technologie objet, l'AOP vise à séparer les définitions des propriétés transversales des logiciels. Chacune de ces propriétés, appelées "aspects", est définie totalement et exclusivement dans un unique module. D'où une simplification du développement et donc moins de risque d'erreurs, puisque les développeurs peuvent focaliser sur un seul problème à la fois. Une fois les différents aspects définis, ils sont assemblés pour construire l'application. Ce processus d'intégration, essentiellement automatique, est appelé "tissage" (weaving). Par ailleurs, si les exemples les plus couramment rencontrés dans la littérature correspondent à des aspects techniques qui décrivent des éléments d'infrastructure (communication distante, transaction, ...), la notion d'aspect peut être vue d'une manière plus générale et peut aussi couvrir également les aspects métiers. Ces derniers représentent des propriétés fonctionnelles qui résultent de la collaboration de différents objets et qui sont, de ce fait, transversales aux applications. Bien que le terme "aspect métier" ne soit pas utilisé en tant que tel, différents travaux traitent du développement sur la base de la séparation des propriétés fonctionnelles. C'est le cas de la programmation par sujets, des travaux sur les points de vues ou de la programmation par contexte. On regroupe l'ensemble de ces approches sous le terme de modèle de représentation multiple.

C'est à partir de ce constat sur les différents types d'aspects, et leur utilisation que nous définissons l'objectif de ce rapport de DEA. Il s'agit de présenter un modèle capable de fédérer à la fois l'intégration des aspects non fonctionnels et celui des aspects fonctionnels. Cette unification nécessitera l'adoption d'une notion la plus générale possible du concept d'aspect. Dans ce qui suit, nous présenterons dans un premier temps différents paradigmes de représentation multiple existants afin de synthétiser les concepts définissant ce type de modèle. Ensuite, nous nous intéresserons aux principes de la programmation par aspect, dans le but d'en dégager les principales caractéristiques. Nous poursuivrons cette étude par l'établissement du modèle d'aspect généralisé et terminerons enfin par son implantation.

Deuxième partie

Etat de l'art

Chapitre 1

Modèles de représentation multiple

Durant cette dernière décennie, le domaine du génie logiciel a vu émergé une multitude de concepts visant à réduire les coûts de développement logiciel. La plupart de ces concepts cherchent principalement à factoriser et synthétiser le code de l'application de manière à le rendre générique, favorisant ainsi la réutilisabilité. L'une de ces approches, vise plus particulièrement à morceler l'application en partie logicielle, afin de faire travailler plusieurs concepteurs spécialisés parallèlement. Cette technique a pour effet de réduire considérablement le temps de développement et par conséquent son coût. Cependant une fois ce morcellement effectué et les parties applicatives créées, le concepteur de l'application se doit d'intégrer chaque partie pour achever le travail. Cette dernière tâche, malheureusement est très complexe et nécessite souvent la modification manuelle de certains éléments, parfois même une réécriture partielle de l'application. Cette approche est représentée par un ensemble de travaux, qui ont conduit à l'élaboration de modèles appelés : ***modèles de représentation multiple***.

1.1 Définitions

Dans cette section, nous empruntons quelques définitions de [Naj99] afin d'établir le vocabulaire de base et de lever les ambiguïtés sur la sémantique des mots *représentation multiple* et *point de vue*.

1.1.1 Qu'est ce qu'une représentation ?

le terme "représenter" est défini dans le dictionnaire ROBERT comme étant le fait *de présenter à l'esprit (un objet absent ou une chose abstraite) au moyen d'un autre objet(signe) qui lui correspond*. Une représentation et plus précisément la représentation à objets (puisque que nous prenons la programmation objet comme point de départ), se définit donc comme étant un formalisme permettant de représenter des connaissances sous forme d'objets organisés en une hiérarchie et fournissant les outils pour les manipuler.

1.1.2 Qu'est ce qu'un point de vue ?

Un point de vue est défini dans le dictionnaire ROBERT comme étant *une manière particulière dont une question peut être abordée* ou encore *un endroit ou l'on doit se placer pour voir un objet le mieux possible*. En informatique, cette notion de point de vue possède un multitude de significations qui divergent selon les travaux et les domaines. En général, on suggère la notion de point de vue dès lors que l'on conçoit des applications multi-utilisateurs qui requiert la coopération d'outils (génie logiciel) ou d'experts avec chacun des connaissances particulière dans des domaines d'applications différents (Conception Assistée par Ordinateur). Nous retiendrons pour notre étude qu'un point vue peut être considéré comme une abstraction qui permet une représentation partielle de l'application. Elle met en relation un concepteur et l'application à modéliser.

1.1.3 Qu'est ce qu'une représentation multiple ?

La représentation multiple peut alors être définie comme étant le fait de conférer à une application plusieurs représentation partielles telle que chacune le décrive dans un point de vue ou une perspective donnée. On note que les différentes représentations partielles se basent généralement pour se définir sur une représentation qui est indépendante de tout point de vue. On nomme cette représentation référentiel.

1.1.4 Qu'est qu'une intégration ?

Le mécanisme d'intégration est un processus permettant la mise en corrélation des différentes entités que sont le référentiel et les points de vues. L'intégration permet de créer de manière effective l'application. Elle ne peut s'exécuter qu'une fois l'intégration terminée.

Dans la suite, nous passons en revue quelques travaux significatifs issus des domaines de la représentation de connaissances et du génie logiciel.

1.2 Les différents paradigmes

Il existe une multitude de modèles objet à représentation multiple. Dans la partie 1.2 de ce chapitre, nous étudions quelques uns de ces paradigmes afin d'en dégager leurs principaux concepts et idées, ce qui constituera l'objet de la partie suivante. Enfin nous étudierons dans le détail deux modèles représentatifs : CEDRE ET CROME.

1.2.1 La programmation par sujets

La programmation par sujets [HO93] est une technique de programmation permettant la réalisation de logiciels à partir de l'assemblage de différents *sujets*. Un sujet est un ensemble de classes représentant une vue particulière de l'application développée. L'une des

caractéristiques essentielles de cette approche réside dans la définition partielle et le partage d'objets. Une classe peut être définie dans un ensemble de sujets, et donc offrir le partage de ses objets aux différents sujets qui la comprennent(les objets définis par cette classe sont accessibles par l'ensembles des sujets, donc partagés). On remarquera de ce fait, que l'intersection de l'ensemble des sujets n'est pas vide, autrement dit, les sujets ne sont pas orthogonaux. Cette caractéristique particulière permet aux développeurs d'avoir une certaine liberté en matière de composition de sujets, et leurs confère la possibilité d'ajouter des extensions au code source original sans le modifier. Elle offre donc la possibilité d'intégrer des fragments logiciels développés indépendamment et suggère par conséquent de nouvelles perspectives en matière de développement logiciel, comme par exemple le fait de pouvoir faire appel à différentes équipes de développement. Ces dernières réalisent la partie logicielle relative au sujet abordé, sans se soucier des différents développements pouvant être réalisés en parallèle. Plus précisément, une équipe peut utiliser certaines définitions de classes sans se préoccuper de la cohérence de ces déclarations avec les autres équipes travaillant sur le logiciel. C'est une fois l'ensemble des sujets implémentés que "l'harmonisation" sera effectuée. Ce type de programmation se révèle donc être un atout majeur dans la réalisation d'applications, que ce soit en matière de rapidité de développement ou de souplesse.

L'élaboration de logiciels à partir de la programmation par sujets repose donc, sur une bonne composition entre les différents sujets de l'application. Cette composition se fait de manière semi-automatique, puisqu'elle est réalisée à partir d'un outil soumis à un ensemble de règles définis par le programmeur. Ces règles ont pour but de spécifier les correspondances existant entre les entités de l'application (classes, attributs, ...) et les sujets qui les définissent. On peut définir deux classes dans des sujets différents, qui restent complémentaires au niveau de l'application. Elles correspondent à des description partielles d'un classe et interviennent par conséquent sur des données communes. Des mécanismes de liaison sont mis en place entre les différentes instances et méthodes de ces classes, afin d'assurer la cohérence des données (définir les variables d'instances qui désignent une même donnée avec des noms différents).

La programmation par sujets s'avère être un outil puissant pour le développement logiciel. Elle est à la fois rapide et souple dans la réutilisation, donc intéressante en matière de coût de développement. Elle possède bien la faculté de pouvoir séparer les différents aspects fonctionnels, notamment à partir des sujets. Cependant, les dépendances existant entre les différents sujets (issues de la composition), pourraient devenir contraignantes lors de la séparation entre les aspects techniques et le code métier. En effet une réification du concept d'aspect, devrait justifier de la mise en place de communications entre aspects techniques qui comme nous le verrons par la suite sont orthogonaux.

1.2.2 La conception par rôle

La conception par rôle encore appelé "role modeling" [AR92], consiste à regrouper des facettes d'une application en *modèle de rôle* pour la création de logiciel. Ce type de conception a lui aussi, comme la programmation par sujets, pour but de développer rapidement

des applications. La caractéristique majeure de cette approche se situe au niveau de la granularité des entités que sont le rôle et le modèle de rôle. Le modèle de rôle décrit les différentes interactions qui existent entre les différents rôles qui le constituent. Pour faire l'analogie avec la programmation par sujets, le modèle de rôle correspond directement au sujet. Par contre, le rôle n'a pas d'entités équivalentes dans la programmation par sujets, mais il correspond à une partie de la description du sujet. Plus précisément il répond à une description des besoins et responsabilités qu'un objet doit satisfaire dans l'exécution de la tâche pour laquelle il est employé. On peut considérer qu'un rôle est un point de vue particulier sur sa participation dans l'application de la tâche. Par conséquent, chaque modèle de rôle se doit de correspondre à une tâche particulière de l'application. Cependant plusieurs modèles de rôle peuvent décrire la même tâche à des niveaux de granularité différents. Cette particularité permet de gérer les visions différentes que pourraient avoir plusieurs spécialistes intervenant au cours de la création logiciel. Par exemple, dans une application de type construction de maison, la tâche « isoler mur » peut avoir plusieurs significations suivant les points de vue d'un spécialiste en isolation thermique et un spécialiste en étanchéité.

L'intégration des rôles, appelée ici la synthèse des modèles de rôle constitue un point crucial dans la définition du modèle. Cette opération significative consiste à regrouper les différents rôles joués par une même entité afin d'établir le raccordement entre plusieurs modèles de rôle. Ce regroupement s'appelle la projection des rôles et s'inscrit dans un nouveau rôle global qui contient l'ensemble des contraintes exprimées lors de cette synthèse. Le but de cette opération consiste principalement à découvrir l'exécution d'une application dans un langage orienté objet.

Cette technique de programmation semble très intéressante sur le plan conceptuel et permet entre autre chose la visualisation des interactions au sein d'applications complexes. Cependant, employer des modèles de rôle au niveau conception et les maintenir à l'exécution exige des techniques plus spécifiques que la projection de rôle décrite ici (voir [VN96] pour le détail de ces techniques). La granularité des entités est dans ce modèle, très intéressante, car elle y définit en partie la notion de points de vue au travers des rôles.

1.2.3 La programmation par points de vue (l'exemple de CEDRE)

La programmation par points de vue a elle aussi pour objectif de construire une application par intégration de point de vue. Dans cette approche de représentation multiple d'objets, on considère similairement au définitions apportées, un point de vue comme une abstraction représentant partiellement un univers du discours contribuant à l'élaboration de la représentation multiple de celui-ci. Cette représentation s'appuie sur un certain nombre de concepts, le premier étant l'élaboration d'une représentation de base servant à l'ensemble des experts pour construire leur propre vue. Le second de ces concepts consiste à fournir aux spécialistes, une liberté relative de spécifications des phénomènes. Il permet de représenter par exemple le même objet, par un ensemble d'attributs différents sans qu'il n'y ait d'ambiguïté sur cet objet. D'autres de ces concepts concernent principalement l'échange d'informations entre experts, la cohérence de représentation par élaboration de contraintes et enfin la coordination.

Dans cette le modèle CEDRE [Naj98], le niveau de granularité par rapport au autres modèles est enrichie. Il possède 4 niveaux : objet, classe, schéma et base. Chaque point de vue sur ces différentes structures, devient alors "structure représentante" de la structure qui lui est associée. Par exemple, le point de vue sur un objet est appelé "objet représentant". On note que tous les niveaux de granularité restent étroitement liés les uns aux autres. La notion de référentiel y est définie pour illustrer le concept du schéma de base qui servira de point de référence à la construction des points de vue. Le référentiel de la représentation multiple est constitué d'un ensemble de propriétés qui sont soient considérées dans leur ensemble, soient partiellement et donc fonctions des points de vues. En ce qui concerne la décentralisation de la représentation, le modèle reprend le concept utilisé dans [San95], s'appuyant sur le morcellement des objets, et l'étend aux classes, schémas et bases. L'échange d'informations entre représentations partielles s'inspire de celui utilisé pour les vues [O295], c'est à dire le mécanisme d'importation et d'exportation de classes. Il s'enrichit d'un échange d'attributs entre classes représentantes par le biais de relations de visibilité définies par les classes. Une relation de visibilité n'est autre qu'un mécanisme d'héritage sélectif entre les classes de cette relation et un mécanisme de délégation entre les objets instances de classes. Enfin, la cohérence est assurée par la mise en place de contraintes, conçues en respectant d'abord la cohérence au sein de chaque représentation et ensuite, les interactions entre les différentes représentations partielles.

L'ensemble des concepts définis dans ce modèle constitue les fondements de la représentation multiple et donc des modèles à base de points de vue. Une étude plus détaillée de ces concepts et du modèle fera l'objet des paragraphes 1.2 et 1.3.1.

1.2.4 La programmation par contexte

La programmation par contexte [GVD97] met particulièrement l'accent sur l'orthogonalité entre deux concepts de programmation que sont les objets et les fonctions. Bien que ces deux approches soient radicalement différentes (d'une part une conception purement objet et de l'autre une structuration fonctionnelle de l'application), elles restent néanmoins intrinsèquement liées. Cette idée appelée "transversalité" réside dans le fait qu'un objet peut utiliser des fonctions communes à plusieurs objets, et que chaque fonction se définit au travers des objets qui l'utilisent. Sur un exemple d'un logiciel de traitement de texte, les objets : document, Texte, Graphique, utilisent plusieurs fonctions comme l'édition ou l'analyse. Symétriquement, on comprend que chacune de ces fonctions n'est définie qu'à partir de la description relative de chaque objet.

Le modèle proposé dans la programmation par contexte, réside dans la conciliation de ces deux approches, à savoir la programmation par objet et la programmation fonctionnelle. Concrètement, il s'agit de définir un cadre de programmation d'objets multifonctionnels. Pour cela on met en place différents plans de conception correspondant chacun à une partie structurelle de l'application. Le premier plan défini, est le plan de base. Il correspond au référentiel de base dans lequel les objets sont définis par des classes, organisées entre elles à partir de relations classiques de composition ou d'héritage. Ce plan de base détient les informations (sur les objets et la structuration) générales communes à l'ensemble des

différents plans fonctionnels, qui sont partagées au travers du référentiel. Ensuite, on crée les différents plans fonctionnels. Ceux-ci sont définis à partir du regroupement d'informations du plan de base et de la fonction à décrire. On considère donc le plan fonctionnel comme un enrichissement et une contextualisation du plan de base. Chaque plan fonctionnel est indépendant par rapport à l'ensemble des autres plans fonctionnels définis sur l'application. Plus précisément, il n'existe pas d'attributs ou de méthodes communes entre deux contextes qui n'appartiennent pas au plan de base. Ceci définit le concept d'orthogonalité. Les classes sont définies de façon générique dans le plan de base et plus précisément dans les plans fonctionnels par un couple (attribut, opération) relatif au contexte. L'un des avantages de la programmation par contexte est de préserver l'héritage à travers les contextes. Les classes "contextualisées" (du plan fonctionnel) possèdent les mêmes liens d'héritage que ceux définis dans le plan de base (elles suivent le schéma établi dans le plan de base). En effet, une classe qui hérite d'une autre d'après le plan de base, hérite de l'ensemble des fonctionnalités qui enrichissent les classes de l'héritage. Cet avantage joue un rôle considérable dans les mécanismes de communication et de partage d'informations. Ainsi, Il est possible de partager des informations entre parties fonctionnelles définies sur une même classe sans même définir de canaux de communications.

La programmation par contexte et surtout le concept d'orthogonalité entre les différents plans, semblent très flexibles en matière de communication, même si l'orthogonalité s'apparente à une contrainte pour certaines applications. Cette flexibilité pourrait devenir un atout lors de l'intégration des aspects techniques. Ce modèle fera l'objet d'une étude approfondie au paragraphe 1.3.2 de ce chapitre.

1.3 Comparaison et classification des approches

Comme nous venons de le constater, il existe de nombreux modèles objets basés sur la représentation multiple. Bien que ces approches soient très proches sur le plans des objectifs, elles restent très différentes dans l'élaboration des modèles. Au travers de ces approches nous essayons de rassembler les différents concepts que peut regrouper un modèle de représentation multiple.

1.3.1 Modèle centralisé versus modèle décentralisé

Il existe deux grandes catégories de modèles objets basés sur la représentation multiple : les modèles centralisés et les modèles décentralisés.

Le modèle centralisé

La distinction entre le modèle centralisé et décentralisé est très simple et s'explique par la présence ou non, d'un référentiel dans l'application. Le modèle centralisé possède donc un référentiel contrairement au modèle décentralisé. Le fait de choisir un modèle plutôt qu'un autre va influer sur deux points : le premier étant sur le mécanisme d'intégration et le second la gestion de la communication entre les différents points de vue.

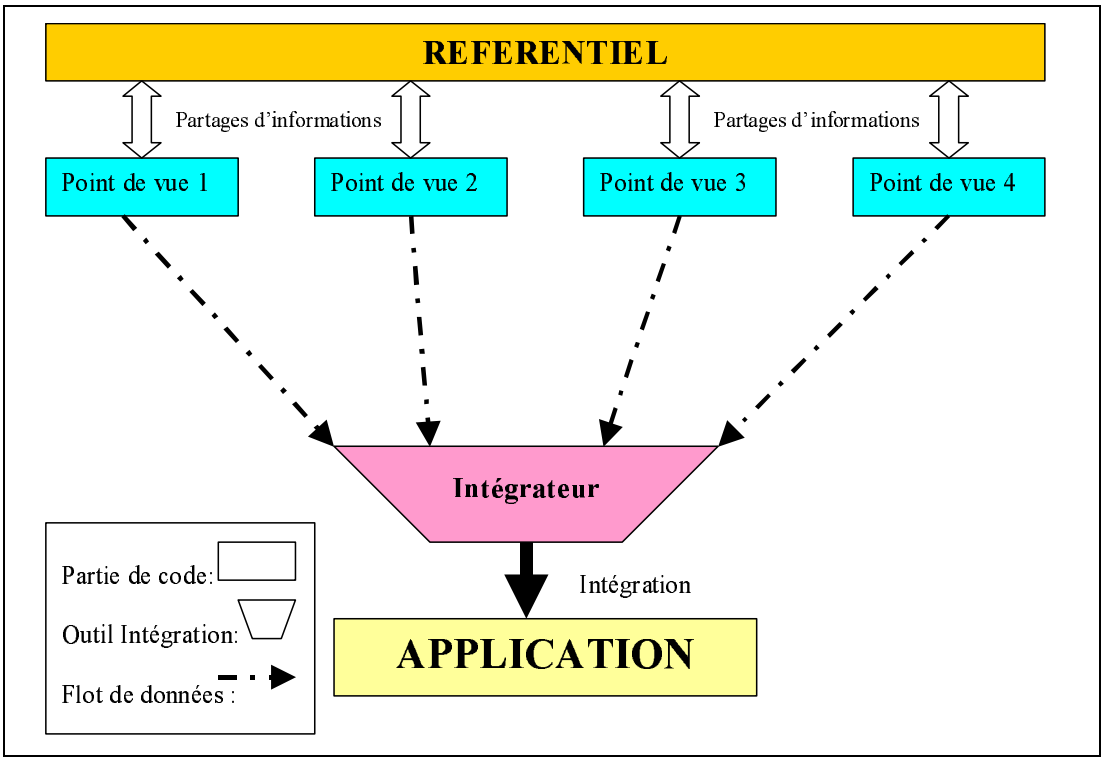

FIG. 1.1 – Intégration d'un modèle centralisé

Clairement, l'intégration d'un modèle centralisé suit le schéma de la figure 1.1. On commence par concevoir le référentiel en y décrivant l'ensemble des classes et objets ainsi que leurs hiérarchies. On raffine ensuite ce référentiel avec les différents points de vue. Une fois l'ensemble de ceux-ci créé, on l'intègre sur le référentiel en respectant les règles d'intégration. L'application est alors fusionnée. Le moteur d'intégration a donc pour mission de vérifier la cohérence des communications existant entre le référentiel et les points de vue, ainsi que de préserver la cohérence des informations partagées. L'avantage d'un modèle centralisé réside essentiellement dans la simplicité de conception du moteur et dans la gestion de la cohérence de l'information. Le référentiel centralise l'ensemble des communications et des données partagées, et de ce fait n'a pas à se soucier des divers problèmes de cohérence qui pourrait intervenir entre deux points de vue distincts sur une donnée (exemple : CROME [Van99]). Il existe cependant des modèles centralisés proposant la mise en place de canaux de communications entre points de vue. Ces canaux ont pour but le partage de données n'appartenant pas au référentiel. Ceux-ci sont construits et déclarés de manière explicite pour que l'intégrateur puisse gérer la cohérence de ces données. On retrouve notamment ce type de modèle dit semi-centralisé, dans CEDRE [Naj99].

Le modèle décentralisé

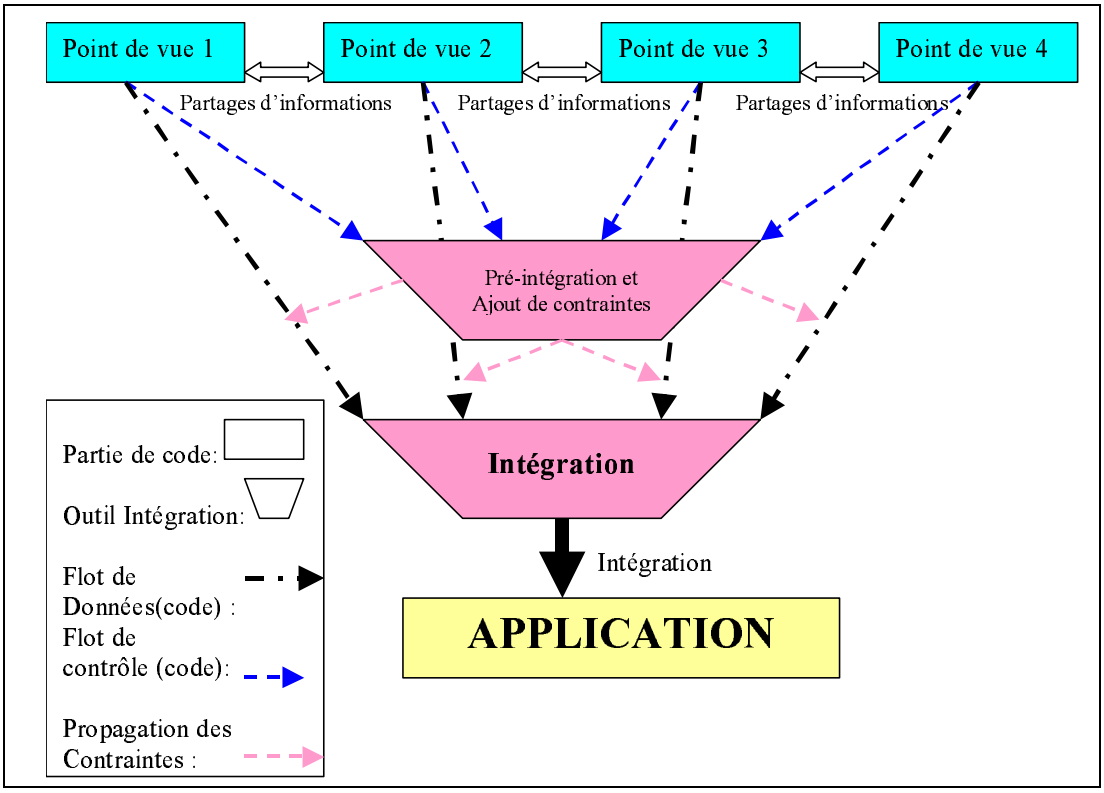

FIG. 1.2 – Intégration d'un modèle décentralisé

Contrairement au modèle centralisé, l'intégration d'un modèle décentralisé est beaucoup plus complexe (voir figure 1.2). L'intégration se déroule en deux étapes. La première consiste en la création de l'ensemble des points de vue. Une fois ceux-ci créés, on réalise une pré intégration qui à pour but de simuler la création d'un référentiel. Cette simulation répertorie l'ensemble des parties communes aux différents points de vue ainsi que l'ensemble des canaux de communications à établir. Une fois la juxtaposition terminée, l'intégrateur transforme les règles recueillies en contraintes d'intégration et les utilise pour construire l'application.

Le modèle décentralisé est certainement plus difficile à réaliser, mais ne présente pas de contraintes de conception, puisque toutes les communications y sont explicitées, et reste certainement le meilleur modèle sur le plan de l'utilisation. Le type de programmation qu'il définit, comme par exemple la programmation par sujet, est très souple et encourage le développement parallèle d'applications, ce qui accroît la vitesse de développement par rapport à une conception objet. En effet, dans la programmation par sujet, l'intégrateur se charge de faire la correspondance entre les différents objets représentant la même entité. Il libère ainsi les équipes de développement de toutes contraintes de nommage ou de cohérence, permettant la création parallèle des points de vue de l'application.

Le choix du modèle se résume donc au choix de l'intégrateur : on peut opter pour un intégrateur gérant l'ensemble des communications inter points de vue ou pour un intégrateur contrôlant simplement les communications explicites. Ce choix aura des conséquences quant à la gestion du partage d'information et donc de sa cohérence. Il est donc primordial de détailler l'ensemble des communications pouvant exister entre les différentes entités composant un modèle de représentation multiple.

1.3.2 Le partage d'information

Comme nous venons de le voir dans le paragraphe précédent, le partage de l'information est différent selon le modèle que l'on choisit. Dans un modèle centralisé, l'ensemble des informations partagées entre points de vue, est contenu dans le référentiel [Van99]. Ainsi lorsque l'on veut accéder à un objet partagé, il faut d'abord en faire la demande au référentiel. Ce mécanisme permet entre autre, d'assurer une certaine cohérence de l'ensemble des données partagées. Les canaux de communications établis pour ce modèle, se centralisent donc autour du référentiel, à la manière des réseaux en étoile. Certains modèles centralisés, comme nous l'avons vu précédemment, utilisent des canaux de communications inter-points de vue. En réalité, cette utilisation résulte d'un choix particulier de conception qui est : l'orthogonalité des points de vue. On précise que deux points de vue sont orthogonaux, si les entités qui les composent sont indépendants.

Concrètement, il existe une sous classification des modèles centralisés : les modèles centralisés orthogonaux et non orthogonaux. Dans un modèle orthogonal, seul les communications avec le référentiel sont nécessaires, contrairement au modèle non orthogonal qui doit utiliser des canaux de communications inter points de vue. Ces canaux nécessitent généralement l'utilisation d'un langage de contraintes chargé d'assurer la cohérence des entités partagées.

Dans les modèles décentralisés, l'information appartient au point de vue. Le partage d'information se fait donc d'un point de vue à un autre par la mise en place de canaux de communications directes comme dans l'approche centralisé non orthogonal. Généralement ces modèles utilisent des langages de contraintes qui permettent de contrôler l'information partagée. On construit à partir de ces langages un ensemble de règles, qui seront chargées de limiter et de contrôler les communications d'un point de vue vers un autre. Dans la programmation par sujets, on utilise des règles de composition qui définissent les accords de partage entre les différents sujets.

1.3.3 La gestion de la cohérence

On regroupe sous le terme de *gestion de cohérence* principalement la gestion des accès en lecture et en écriture sur les données partagées, ainsi que la mise à jour de ces valeurs après modification de leurs états.

Le principal problème survient à l'exécution. Lorsque deux points de vue s'exécutant dans deux threads différents veulent accéder à une donnée partagée, il faut pouvoir garantir l'intégrité de la donnée partagée. Il peut parfois s'agir aussi de mettre a jour des données

partagées lorsque l'on utilise la duplication de donnée pour effectué le partage. La gestion de cette cohérence dépend bien donc du modèle de point de vue choisi, puisqu'elle dépend directement du partage de l'information (partage simple ou duplication).

Il existe donc plusieurs techniques pour assurer cette cohérence dont la plus simple est certainement la définition d'accesseurs uniques sur la donnée (cas d'un partage simple). Cette technique est notamment utilisée dans les modèles centralisés, mais peut aussi s'appliquer aux modèles décentralisés. Elle consiste à fournir pour chaque donnée partagée une méthode d'accès en lecture et une méthode d'accès en écriture. Ainsi lorsque deux points de vue cherchent à accéder à une donnée particulière, les envois de messages sont traités séquentiellement. Dans les modèles centralisés ces méthodes sont définies dans le référentiel. Cette technique résout les problèmes d'accès simultanés en écriture, mais reste assez problématique lors de cas de duplication, notamment dans les modèles décentralisés. En effet, son utilisation, dans ce type de modèles mais aussi dans les modèles centralisés non orthogonaux, nécessite l'emploi de mécanismes de propagation de la nouvelle valeur à l'ensemble des protagonistes. Néanmoins, dans les modèles centralisés orthogonaux, la mise à jour s'exécute de manière automatique, puisque toutes les données partagées appartiennent au référentiel.

D'autres techniques existent pour gérer la cohérence de données, mais se généralisent la plupart du temps par l'utilisation d'un langage et d'un gestionnaire de contraintes. Ce gestionnaire contrôle l'intégrité des données à l'aide de mécanismes de violation de contraintes, mais aussi la consistance de l'ensemble des contraintes définies afin d'éviter les contradictions qui pourraient survenir.

1.3.4 Fusion et juxtaposition

Il existe d'autres critères de spécification des modèles à représentation multiple. On peut notamment recenser les modèles utilisant la fusion ou la juxtaposition. La fusion consiste comme nous l'avons vu dans la plupart des approches à intégrer l'ensemble des points de vue et le référentiel s'il existe, pour former l'application. Après fusion, la notion de point de vue, en temps qu'entité de l'application, disparaît. Il en est de même pour le code défini dans chaque point de vue sur un classe particulière, il est fusionné dans une unique classe. Par exemple le modèle réalisé par Harrison [HO93] à partir de la programmation par sujet, est un modèle qui fusionne ces points de vue. En effet, après les opérations de composition, l'application, même si elle peut garder partiellement la notion de point de vue, a perdu les entités représentant les points de vue.

La juxtaposition, au contraire de la fusion, garde cette notion d'entité du point de vue après intégration, ainsi il reste manipulable dans l'application. Pour la juxtaposition, il existe le modèle de représentation multiple CEDRE [Naj99].

1.3.5 Dynamicité et staticité

La notion de dynamicité est elle aussi une caractéristique des modèles de représentation multiple. Un modèle dynamique est un modèle qui permet en cours d'exécution l'ajout et

la suppression de points de vue. Au contraire, un modèle statique n'autorise ces opérations qu'à l'intégration. Le modèle basé sur la programmation par point de vue (CEDRE) fait parti des modèles dynamiques, alors que le modèle d'Harrison [HO93](programmation par sujet) est purement statique.

1.3.6 La granularité

Comme nous avons pu le constaté, la granularité peut elle aussi être un critère de classification. elle correspond au niveau d'application du concept des points de vue. On en distingue de différents types, comme la programmation par sujets qui possède une granularité de niveau applicatif, CROME avec une granularité de niveau classe, ou encore CEDRE, qui lui est multi-niveau puisque sa granularité peut être applicative, de classes ou d'instances.

1.3.7 Synthèse de la classification

Il existe, comme nous venons de le voir, une multitude de critères de spécifications. Même si cette liste n'est pas exhaustive, elle reprend l'essentiel des caractéristiques permettant de définir un modèle de représentation multiple satisfaisant. Le tableau suivant résume l'ensemble des modèles étudiés auparavant, ainsi que les caractéristiques respectives.

	Centralisé Décentralisé	Partage information	Gestion cohérence	Juxtaposition Fusion	Dynamique Statique	Niveau de Granularité	Type de programmation
Modèle sujet [HO93]	Décentralisé	Inter Sujet	Composition	Fusion	Statique	applicatif	Programmation Par sujets
« role modeling » [AR92]	Décentralisé	Inter Role	Synthèse	Fusion	Statique	Classe et instance	Conception par rôle
CEDRE [Naj98]	Semi-centralisé	Inter entité et référentiel	Langage de contraintes	Juxtaposition	Dynamique	Multi-niveau	Programmation par point de vue
CROME [GVD97]	Centralisé	Référentiel	Contrainte d'unicité des accesseurs	Fusion	Dynamique	Classe	Programmation par contexte

FIG. 1.3 – Tableau récapitulatif

1.4 Présentation de deux modèles de référence

Comme nous venons de le préciser, le critère de dynamicité est très important, et fait parti des caractéristiques que notre modèle devra intégrer. C'est pourquoi, nous étudierons dans les deux prochains paragraphes les modèles dynamiques cités précédemment, à savoir, le modèle CEDRE et le modèle CROME.

1.4.1 Le modèle CEDRE

Le modèle CEDRE [Naj99], comme nous venons de le voir, est un modèle dynamique, centralisé dont l'intégration utilise le mécanisme de juxtaposition. Dans ce paragraphe, nous étudierons la description générale du modèle et plus précisément les différentes entités existantes. Ensuite, nous décrirons l'ensemble des mécanismes mis en place pour assurer le partage d'information et la gestion de la cohérence.

Description générale du modèle

CEDRE est donc un modèle centralisé qui se compose de plusieurs entités spécifiques réparties en deux catégories, les entités *multivues* et les entités *vue*. Les entités multivues correspondent aux parties communes de l'application, tandis que les entités vue y décrivent les parties spécifiques. Clairement, une entité multivue contribue à conférer une représentation multiple à l'univers de discours. Elle est spécifique au référentiel. Un point de vue est une notion inhérente au modèle et est défini sur chacune des entités multivues, par un ensemble d'entités vue associé. Généralement, l'entité vue est définie par un enrichissement de l'entité multivue qui lui est associée. Il existe quatre types d'entités différentes qui sont : objet, classe, schéma,ou base.

Définitions

- **Une entité multivue** est morcelée en plusieurs parties :
  - Une partie constituant le référentiel, contenue dans l'entité multivue elle-même.
  - Des parties constituant les représentations partielles qui sont réparties dans les entités vue. Elles sont reliées à l'entité multivue par *une relation R.view-of.*
- **Une classe multivue** (mvc) possède elle aussi plusieurs structures :
  - Une structure contenant un ensemble d'attributs décrivant un ensemble d'entités réelles indépendantes de tout point de vue, constituant le référentiel, et formant la structure de la classe multivue elle-même.
  - Des structures telles que chacune décrive un ensemble d'entités réelles relativement à un point de vue, constituant ainsi les représentations partielles. Elles sont réparties dans des classes appelées classes vue.
- **Un objet multivue**(mvo) est une instance d'une classe multivue(mvc) et possède plusieurs états :
  - Un état décrivant l'objet indépendamment de tout point de vue et constituant le référentiel. il est formé des attributs de la structure de la classe multivue (mvc).
  - Des états qui décrivent l'objet multivue relativement à chaque point de vue. Ces états sont répartis dans des objets appelés objets vue. Ces objets vue sont instances de classes vue.
- **Un schéma multivue** (mvs) est associé à plusieurs hiérarchies de classes (en structure d'arbres) :
  - Une hiérarchie qui constitue le référentiel (composée de classes multivues reliées entre elles par des liens de généralisation / spécialisation).

  - Des hiérarchies réparties en schémas vue représentant partiellement le monde réel (univers de discours).
- **Une base multivue** (mvb) représente plusieurs ensembles d'objets de l'univers de discours :
  - Un ensemble d'objets multivues dont les états véhiculent le référentiel.
  - Des ensembles d'objets telles que chaque ensemble regroupe des objets qui sont des représentations partielles d'entités réelles relative à un point de vue. Ces entités sont réparties dans des bases vue.
- **Une classe vue** (vc) est une représentation partielle associée à une classe multivue par la relation R.view-of. Cette relation implique que la structure de la classe vue hérite de toute la structure de la classe multivue. Les attributs sont hérités eux aussi(ils sont appelés attribut par défaut) et peuvent être redéfinis.
- Il en est de même pour **les schémas vue et bases vue** (vs et vb) qui sont respectivement des représentations partielles de schémas et bases multivues reliées entre elles par des relations R.view-of qui induisent des conditions sur les classes vue et les liens de spécialisations de ces dernières.
- **Un objet vue** (vo) est une représentation partielle associée à un objet multivue par *la relation R.view-of*, et est lié par un lien d'instanciation à sa classe vue. Cette classe vue doit être elle-même associée à la classe d'instanciation de son objet multivue *par un lien R.view-of*.

Il existe un type particulier d'entité, appelé entité mono vue. Cette entité possède une unique représentation selon un point de vue. Ce type particulier permet, entre autre, de définir des entités qui ne seront pas référencées dans le référentiel, autrement dit, qui n'auront pas d'entités multivues associées. Ainsi chaque point de vue a la capacité de définir des entités qui lui sont spécifiques et dont la visibilité n'excède pas le point de vue (l'entité pourrait être qualifiée d'entité privée).

Partage et protection d'informations

Pour une représentation partielle rp1 d'une classe multivue, échanger de l'information avec rp2(une autre représentation partielle) consiste à pouvoir d'une part accéder à des attributs de rp2, et d'autre part transmettre à rp2 des propriétés pertinentes de rp1. Dans ce modèle, l'échange d'attributs se fait explicitement entre classes vue d'une même classe multivue à partir de relations de visibilité (*R.see*). Ces dernières sont basées sur la définition d'un filtre qui sélectionne les attributs de la classe source. Ces mêmes relations permettent aussi d'accéder à l'ensemble des propriétés d'une classe vue.

Il est important que les attributs considérés comme visibles, gardent les mêmes valeurs dans le point de vue qui les définit. Ils peuvent éventuellement s'adapter aux besoins du point de vue accédant à l'attribut partagé. Dans ce cas, on considère cette adaptation comme une redéfinition de l'attribut ou une conversion de celui-ci (au moyen de fonctions de conversion définies dans le modèle). Le fait qu'aucun attribut visible ne pourra être librement évalué par un point de vue externe, garantit donc une cohérence globale, protégeant ainsi l'information.

Gestion de la cohérence des données

Pour assurer une bonne cohérence des données au cours de l'exécution de l'application, CEDRE possède un langage de contraintes. Ce langage permet de définir des contraintes de deux types : les contraintes intra représentations et les contraintes inter représentations.

- Une contrainte intra-représentation contribue à assurer la cohérence dans un schéma donné en posant un ensemble de restrictions sur les objets appartenant à la base associée à ce schéma. Généralement elle est exprimée sur une classe de tout type et peut porter :
  - sur un attribut : elle pose une restriction sur la valeur de l'attribut ou sur sa cardinalité.
  - sur plusieurs attributs : elle pose une restriction sur plusieurs attributs appartenant à la même classe ou à des classes reliées par des liens d'utilisation($R.uses$).
- Une contrainte inter représentations se veut d'assurer une compatibilité entre les représentations des objets multivues. Elle est spécifiée dans une classe multivue à l'aide d'un couple (mvc, vp-fo) où mvc est la classe multivue et vp-fo est une formule primitive du langage de contraintes. Cette contrainte exprime une relation qui lie des représentations différentes d'un objet multivue et peut porter sur :
  - la classe d'appartenance d'une représentation de l'objet multivue dans un point de vue.
  - l'existence d'une représentation dans un point de vue.
  - des attributs relatifs à plusieurs représentations de l'objet multivue (attributs appartenant à des structures de classes vue de points de vue différents).

Ce langage de contraintes permet non seulement d'assurer la gestion de la cohérence des données partagées entre le référentiel et les points de vue, mais aussi la cohérence de l'information échangée entre points de vue (modèle non orthogonal). De ce fait, il permet de garantir la mise à jour de l'information au cours de l'exécution, répondant ainsi à l'un des principaux problèmes de la dynamicité (canaux de communications inter points de vue).

Modèle juxtaposable et dynamique

Comme nous l'avons vu, le modèle s'intègre à partir des mécanismes de juxtaposition. Il conserve donc à l'exécution ses points de vue en tant qu'entité. En effet, chaque entité vue associée à une entité multivue peut être considérer comme une copie enrichie de sa représentation multivue. Ainsi, chaque point de vue possède une certaine autonomie d'exécution, restreinte par le langage de contraintes. Cette autonomie garantit précisément, la notion d'entité exécutable pour chaque point de vue, et donc par conséquent la juxtaposition.

La dynamicité dépend généralement de la plate forme de développement utilisée. Néanmoins, un modèle dynamique doit s'assurer que des mécanismes de mis à jour des données partagées soient définis, afin de garantir leur cohérence au cours de l'exécution. Dans ce modèle, le langage de contraintes assure cette cohérence, à partir notamment, de la création de contraintes sur les données transitant entre les canaux de communications inter points

de vue.

La granularité

Comme nous l'avons spécifié la granularité dans cette approche est multi-niveau. On la retrouve notamment au niveau applicatif avec les schémas et les bases, au niveau classes avec les classes, et aux niveaux instance avec les objets.

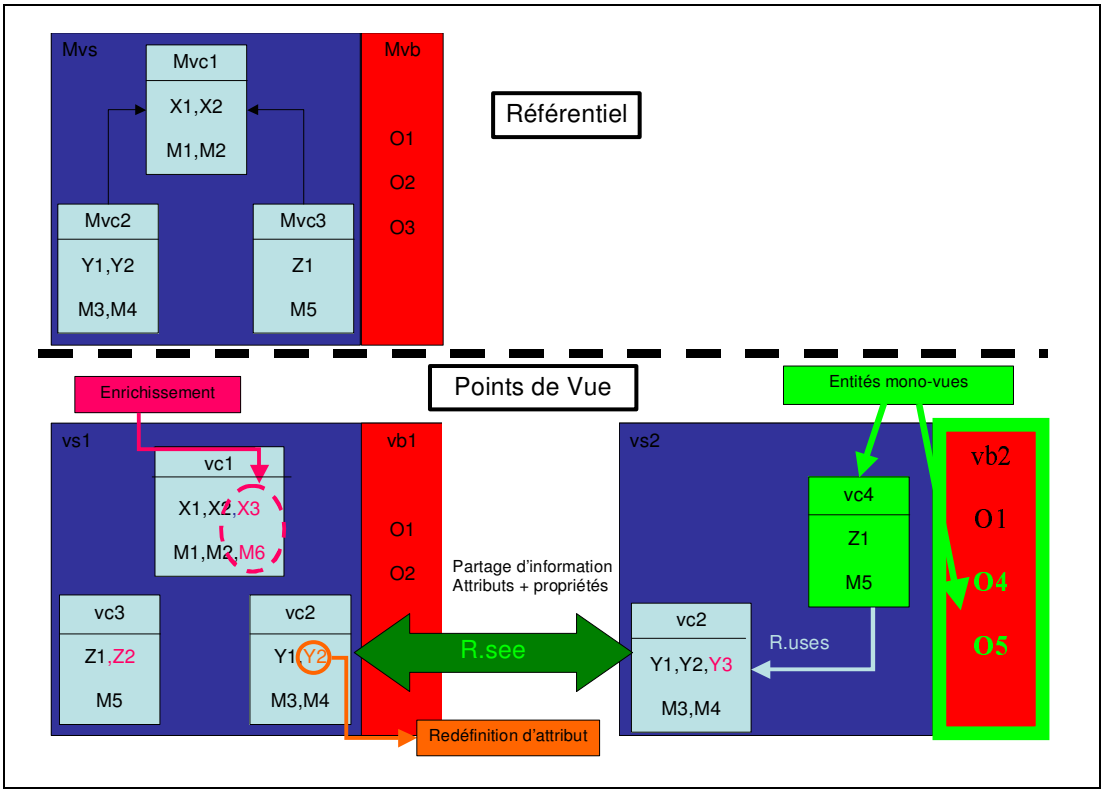

FIG. 1.4 – Exemple d'architecture du modèle CEDRE

La description de l'implémentation du modèle CEDRE est détaillée dans l'annexe A de ce document.

1.4.2 Le modèle CROME

Le modèle CROME [Van99] est lui aussi un modèle dynamique, centralisé mais avec un mécanisme d'intégration utilisant la fusion. Dans ce paragraphe, nous étudierons la description générale du modèle, en discutant notamment des principes de la programmations par plans . Puis, nous décrirons les mécanismes conçus pour garantir le partage d'information et la gestion de la cohérence. Nous porterons l'accent sur l'héritage et sur les communications intra et inter contextes.

Description générale du modèle

CROME est un modèle très intéressant sur le plan architectural. Il introduit une structuration en plan de description, à partir des objets contenus dans le référentiel ; chaque plan correspondant à une fonction (point de vue) de l'application. Il utilise plus précisément une stratégie d'héritage modulaire, basée sur la notion de plan, pour décrire les objets d'une fonction particulière. En ce qui concerne la communication entre objets, elle est circonscrite relativement aux fonctions offrant une modularité de programmation des traitements fonctionnels multiples. L'utilisation de l'héritage modulaire par plan, favorise aussi la factorisation de code au niveau des points de vue, et augmente donc le niveau de granularité par rapport à la programmation objet.

La programmation par plans

La notion de plan est une notion structurante intervenant à la conception, qui systématise transversalement la description des objets et qui permet de rendre compte de la double structuration objets/fonctions. Il existe deux types de plans :

- le plan de base
- les plans fonctionnels

Le premier à être défini, est le plan de base,qui détermine l'identité des objets du référentiel et la description de ces objets partagés par toutes les définitions. Ces objets ainsi introduit sont ensuite décrits spécifiquement dans les plans fonctionnels où ils interviennent. Selon ces plans, l'articulation entre deux axes de structuration se traduit de la façon suivante :

- La participation des objets dans les différentes fonctions est traduite par un ensemble de descriptions, lesquelles sont chacune définies dans un plan fonctionnel différent.
- La participation de plusieurs objets à une même fonction est définie par un ensemble de descriptions respectives localisées modulairement dans le même plan fonctionnel.

1. ***Le plan de base***

   Ce plan détermine la structuration du référentiel en classes : les objets sont identifiés et décrits par un ensemble de classes organisés hiérarchiquement par des relations d'héritage et relations entre objets(Chaque classe contenant la description de base des objets). La description de base contient les caractéristiques (attributs et méthodes) des objets partagées.

   La relation d'héritage entre classes fait intervenir les notions de classes abstraites et classes concrètes qui coexistent dans l'environnement CROME. Cette relation d'héritage est définie classiquement et s'utilise comme les relations d'héritage rencontrées en programmation objet.

   Les relations entre objets des différentes classes sont introduites au niveau du plan de base par utilisation des attributs des descriptions de base et permettent ainsi d'organiser les objets du référentiel selon une structure partagée par toutes les fonctions.

2. ***Les plans fonctionnels***

Les plans fonctionnels sont les plans associés à chaque fonction du système et interviennent une fois le plan de base déterminé. Leur but consiste à enrichir les classes du plan de base pour les contextes fonctionnels correspondant.Cette enrichissement est spécifié par des parties fonctionnelles qui déterminent les caractéristiques(attributs et méthodes) à ajouter, à une classe, pour que ses objets interviennent dans la fonction associée.

Il existe une contrainte d'unicité concernant les parties fonctionnelles : Pour chaque classe introduite dans le plan de base, un plan fonctionnel ne peut détenir qu'une et une seule partie fonctionnelle relative à cette classe.

L'existence de parties fonctionnelles associées aux classes dans le plan correspondant permet de déterminer si certains objets ou classes interviennent dans la fonction ou non. Ce souci de détermination de l'ensemble des participants à une fonction, est très important pour la maintenance de l'application.

La structure d'une partie fonctionnelle se résume à un ensemble d'attributs et de méthodes privées et publiques. Les attributs prennent en compte les données spécifiques à la fonction et les relations inter-objets locales. Ils ne sont accessibles que par les méthodes de la partie fonctionnelle. Les méthodes permettent l'ajout de traitement spécifique pour la fonction. On distingue la notion de méthodes privées et publiques, accessibles ou non de l'extérieur. Il n'existe pas de restriction relative au nommage des caractéristiques.

3. ***Articulation avec la description de base***

   Une partie fonctionnelle entretient une relation d'héritage avec la description de base faite pour la classe qu'elle définit. Cette relation permet entre autre d'avoir accès à l'ensemble des caractéristiques du plan de base et de pouvoir redéfinir les méthodes du plan de base pour les enrichir et les adapter aux spécificités de la fonction associée.

Héritage

Comme nous venons de le voir l'héritage joue un rôle prépondérant dans la conception du modèle, il permet entre autre la structuration de l'application en plan fonctionnel, mais aussi une meilleure factorisation de son code. Dans ce paragraphe, nous étudions les différentes formes d'héritage et leurs effets sur l'application.

1. Héritage local aux plans fonctionnels.

   Le plan fonctionnel préserve les relations d'héritage issues du plan de base. En d'autres termes, la partie fonctionnelle enrichissant une classe est héritée localement par ses sous-classes. Cette préservation d'héritage souligne deux points importants. le premier c'est qu'on dispose d'un héritage contextualisé entre les classes au travers de leurs parties fonctionnelles respectives.Le second concerne l'héritage implicite : même si une classe n'est pas explicitement enrichie, elle bénéficie implicitement de l'enrichissement de ses super-classes.

2. Héritage modulaire

La stratégie d'héritage modulaire amène à dissocier pour chaque contexte et pour chaque classe deux modules : le module de base incluant la description de base et le module spécifique incluant les parties fonctionnelles. Cette stratégie consiste à parcourir dans l'ordre le module spécifique (pour trouver la caractéristique requise) puis le module de base(si elle n'est pas trouvée) garantissant ainsi qu'un objet dispose des caractéristiques définies par le module spécifique et par le module de base, et garantit la règle du plus affiné relatif au contexte(la définition la plus affinée prime sur la moins affinée).

L'héritage modulaire permet entre autre de redéfinir des méthodes du plan de base dans un contexte, d'avoir des références génériques ou masquées au plan de base...

Communications intra et inter contexte

1. **Communication entre objets intra-contexte**

   La communication entre objets de même contexte s'effectue par envoi de message sur les objets référencés. Les messages qu'un objet peut envoyer, sont basés sur leur description dans le même contexte, à savoir les méthodes publiques qui sont déclarées dans le plan fonctionnel. Autrement dit, un objet n'a qu'une connaissance partielle des autres objets du contexte, qu'il référence.

   Il est important de noter que ces envois de messages sont soumis à quelques règles :
   - Dans les méthodes publiques et privées de la description de base, les seules méthodes publiques des autres objets pouvant être activées sont celles issues de leur description de base.
   - Dans les méthodes publiques et privées d'une partie fonctionnelle, l'ensemble des méthodes publiques déterminées pour les autres objets dans le contexte sont accessibles par envoi de messages.

   On notera aussi que le polymorphisme induit par l'envoi de message dans les contextes est conservé. Il permet entre autre la définition de méthodes génériques par rapport aux autres objets intervenant dans ces contextes.

2. **Communication entre objets inter-contexte**

   L'intérêt d'une communication inter-contexte est principalement d'échanger des données entre les fonctions, de coordonner une fonction par rapport à un état fourni par une autre, ou encore de gérer la cohérence entre plusieurs fonctions. Pour cela CROME considère les deux solutions suivantes : le partage entre contexte obtenu à travers la description de base et l'appel de méthodes entre contexte au sein de l'objet.

   (a) *Partage d'objet de base entre contextes*

   Il s'agit du partage de méthode lorsque plusieurs parties fonctionnelles d'une classe font appel à une même méthode de la description de base. Ce cas s'apparente à celui de plusieurs méthodes d'une classe appelant une même méthode de celle-ci dans un langage objet classique. Il s'agit aussi du partage d'attribut lorsque plusieurs parties fonctionnelles d'une classe font appel à un même attribut de la description de base. Ce cas s'apparente aux cas d'une classe dont

plusieurs méthodes appellent toutes une méthode unique, Ce qui implique un partage de la valeur des attributs de la description de base entre les différentes parties fonctionnelles. La modification par une partie fonctionnelle d'un attribut est donc immédiate pour l'ensemble des autres parties fonctionnelles.

Ce type de partage soulève néanmoins deux problèmes. Le premier concerne les interférences entre méthodes intervenant sur un même attribut, le second, la cohérence des données partagées. Pour cela, CROME utilise des méthodes d'accès uniques en lecture et en écriture sur chaque attribut, synchronisant ainsi les accès de façon séquentielle (voir le paragraphe sur la gestion de la cohérence) et garantissant les données partagées.

(b) *Appel de méthodes inter-contextes*

Le principe consiste à appeler une méthode d'une autre partie fonctionnelle(autre que celle appelante) au sein de l'objet. Toute méthode d'une partie fonctionnelle, quelle soit publique ou privée est visible par l'ensemble des parties fonctionnelles liées à l'objet qu'elles partagent.L'objet, ainsi décomposé en parties fonctionnelles reste donc l'unité d'encapsulation du système.

Cet appel de méthode d'un autre contexte provoque donc un changement de contexte qui conduit l'objet à ce comporter suivant le nouveau contexte(contexte d'appel).

Contrairement à CEDRE, CROME ne nécessite pas l'introduction d'un langage de contraintes pour assurer la cohérence des données partagées. En fait, cela s'explique par le fait que CROME est un modèle centralisé orthogonal, et donc qu'il ne sollicite pas de communications entre contexte. Néanmoins, on a vu que sur un même objet, CROME permettait l'accès à d'autres méthodes que celles contenues dans le référentiel et dans le contexte initial. Conscient que certaines applications nécessitent l'ouverture de canaux de communications entre contextes (modèle non orthogonaux), Crome propose à partir de son modèle la possibilité de partager de l'information entre plusieurs objets de contextes différents. Pour cela il utilise la combinaison des deux solutions précédentes, à savoir le partage d'objets de base entre contexte et les appels de méthodes inter contextes. La combinaison de ces deux solutions conduit à une résolution des problèmes de communications inter-objets et inter-contextes.

On dispose donc de deux approches :

- La première consistant à effectuer l'envoi de message d'abord au sein du contexte et d'effectuer ensuite le changement de contexte.
- La seconde consistant à faire l'inverse de la première idée, c'est à dire effectuer le changement de contexte puis l'envoi de message.

Cependant, CROME n'assure que partiellement la cohérence des données dans ce type de partage, puisqu'il n'existe pas de langage de contraintes spécifique pour ce genre de communications. Au final, il est important de retenir que l'envoi de messages entre contexte est possible en CROME et que cela peut se résoudre par l'utilisation des deux solutions proposées.

Modèle fusionnable et dynamique

Comme nous l'avons vu, le modèle s'intègre à partir des mécanismes de fusion. En effet, une fois l'application compilée, les plans fonctionnels représentants les points de vue n'existent plus. En fait, le modèle CROME, peut être considérer comme "un style de programmation", "une façon" d'écrire l'application, et de l'organiser en un ensemble de plans. Le caractère "automatique" (utilisant des concepts de la programmation objet) de résolution des problèmes de la programmation par points de vue, contribue à cette affirmation.

La dynamicité dépend principalement de la plate forme de développement utilisée. Néanmoins, un modèle dynamique doit s'assurer que des mécanismes de mis à jour des données partagées soient définis, afin de garantir leur cohérence au cours de l'exécution. Dans ce modèle, les différents mécanismes de gestion de la cohérence sont principalement issus de la programmation objet et suffisent amplement pour l'exécution de modèle orthogonaux. Cependant dans le cas où l'utilisateur devrait avoir recours à l'utilisation de canaux de communications, CROME ne garantit pas la mise à jour de l'ensemble des données modifiées. La vérification reste à la charge de l'utilisateur.

La granularité

la granularité est ici de niveau classe. En effet, le niveau d'application du concept de contexte se fait ici au niveau des classes et de la hiérarchie de celles-ci. On note quand même un léger degré dans cette granularité avec la définition de parties fonctionnelles qui composent les différents plan. Néanmoins ces parties ne constituent pas un niveau assez riche pour apparaître comme un niveau à part entière.

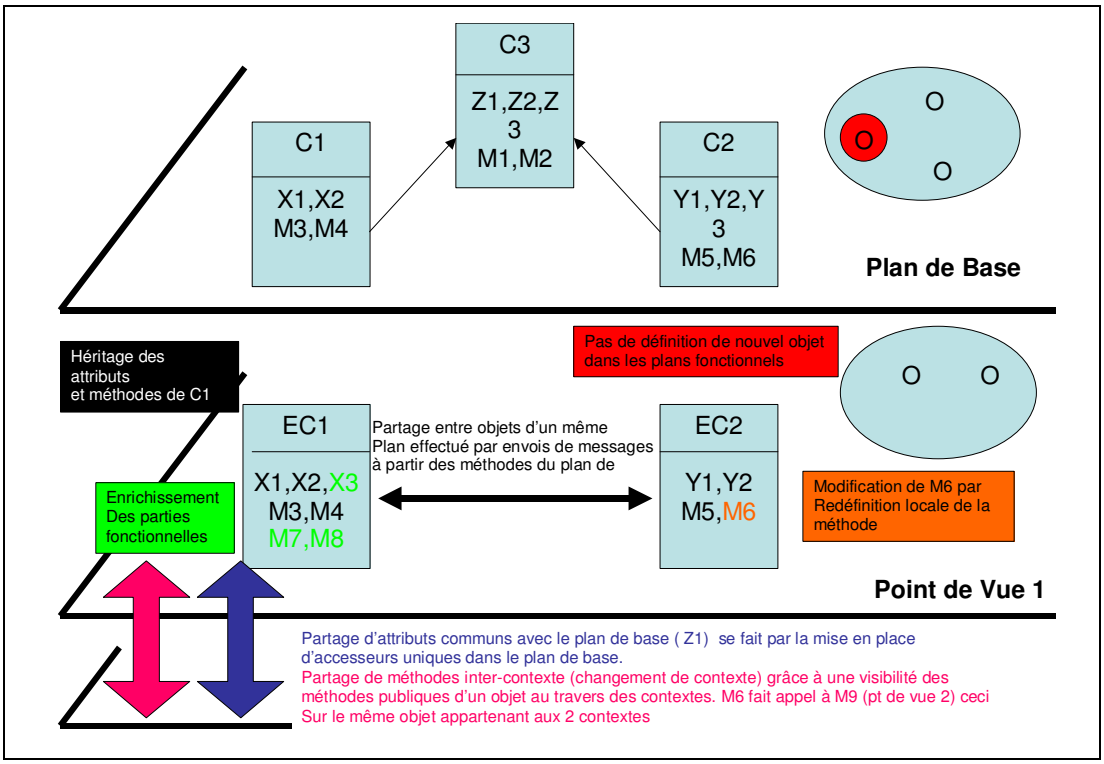

FIG. 1.5 – Exemple d'architecture du modèle CROME

La description de l'implémentation du modèle CROME est détaillée dans l'annexe B de ce document.

1.4.3 Bilan des deux approches

Comme nous avons pu le constater, les deux modèles ont chacun leurs avantages et leurs inconvénients. On peut notamment citer d'un coté la simplicité de gestion de la cohérence pour CROME et de l'autre la rigueur et le contrôle de l'ensemble des communications par le biais du langage de contraintes. L'intégration par juxtaposition se révèle être aussi un atout pour la dynamicité (offrant la possibilité de retirer et ajouter des point de vue en cours d'exécution), contrairement à la fusion qui restreint cette dynamicité par perte de la notion de point de vue. Cette juxtaposition par contre, nécessite de lourds mécanismes de contrôle qui ralentissent l'exécution.

Il est difficile de choisir un modèle, plus qu'un autre comme point de départ. Néanmoins le simplicité de CROME et surtout sa capacité d'évolution, nous pousse pourtant à opter pour ce modèle. En effet, contrairement à CEDRE, le modèle CROME est beaucoup plus accessible en matière de modification que le modèle CEDRE qui avec son langage de contraintes limite les possibilités d'extension. Un des avantages de CROME, c'est qu'il possède aussi des propriétés de conception assez proches de celles que nous rencontrerons

dans l'état de l'art sur les aspects. On peut citer à titre d' exemple l'orthogonalité entre les plans fonctionnels qui est une notion similaire à l'indépendance entre les aspects.

Le fait que CROME ne gère pas effectivement les canaux de communications inter contextes, n'est pas réellement un problème pour le moment. La mise en place de contraintes pourra être effectuée ultérieurement sur le modèle pour contrôler la cohérence des données.

	CEDRE	CROME
Architectures des Modèles Entités	Bases Schémas Classes Objets	Plans Classes Objets
Présentation	Entités Multivues (référentiel) ↓ *Rview-of* Entités-vue (point de vue)	Plan de base (référentiel) ↓ *Héritage* Plans fonctionnels (point de vue)
Liens entre Référentiel & Points de vue	Lien d ' instanciatiation : *Rview-of*	Parties fonctionnelles Héritent du plan de base
Partage d ' information	Rsee (Filtre) pour les vues appartenant à la même multivue Redéfinition d'attributs (Retypage)	*Intra contexte :* Visibilité des méthodes publiques *Inter contexte*: Visibilité de l'ensemble des méthodes d'un objet Au travers de toutes ces parties fonctionnelles
Cohérence Intra-point de vue	Contraintes intra-représentations portant sur 1 ou plusieurs attributs	Insertion d'une méthode unique dans la description de base, utilisée pour déterminer l'attribut (contrainte pour accéder à la valeur)
Inter-point de vue	Contraintes inter-représentations portant sur les objets d'une classe multivue	Résolu par la visibilité, sinon d'autres méthodes à approfondir

FIG. 1.6 – Tableau récapitulatif des concepts des modèles CEDRE et CROME

Chapitre 2

La programmation par aspects

2.1 Introduction

La POA (Programmation Orientée Aspect, AOP - Aspect-Oriented Programming) a été définie par Gregor Kickzales (du Xerox) en 1996. Ce type de programmation apporte une nouvelle approche dans la conception de nos programmes. Le but majeur est d'offrir la possibilité aux programmeurs de produire un code plus lisible et réutilisable. La POA se présente dans son approche, plus comme une philosophie de programmation. Elle tente de résoudre des problèmes de façon plus élégante que les méthodes classiques. L'esprit de cette philosophie repose sur trois points essentiels.

Un projet informatique tient son efficacité de la capacité à bien séparer les préoccupations de nature différentes. Ainsi, lorsqu' un des programmeurs veut faire une modification, celle-ci ne doit être communiquée qu'à son voisin direct, on ne doit pas introduire de bugs supplémentaires. Donc le premier point essentiel est de faire **une séparation des préoccupations**.

Lorsque l'on se trouve dans un système informatique complexe on rencontre toujours des préoccupations qui ne peuvent être modularisées comme on le souhaiterait. Certains services vont être généraux car utilisés par plusieurs modules. On observe donc **une dispersion des préoccupations**.

Une solution pour éviter cette dispersion serait de ne pas utiliser ces services communs et d'**inverser les dépendances**. A ce moment c'est le service qui utilise le programme et non plus l'inverse.

2.2 Motivation : les limites de la programmation objet

L'un des principaux arguments de la programmation par aspects est de séparer l'ensemble des préoccupations d'une application. Pour cela, la programmation objet possède un nombre d'outils comme le polymorphisme et l'héritage, qui lui permettent de modulariser au mieux une application. Prenons l'exemple de la fonctionnalité Parsing qui consiste à parser des fichier XML. Comme nous pouvons le constater sur le schéma, cette fonction-

nalité représentée par les traits rouge horizontaux, se retrouve bien dans une seule classe et constitue un bloc compact. Le programme, dans ce cas, bénéficie d'une bonne modularité.

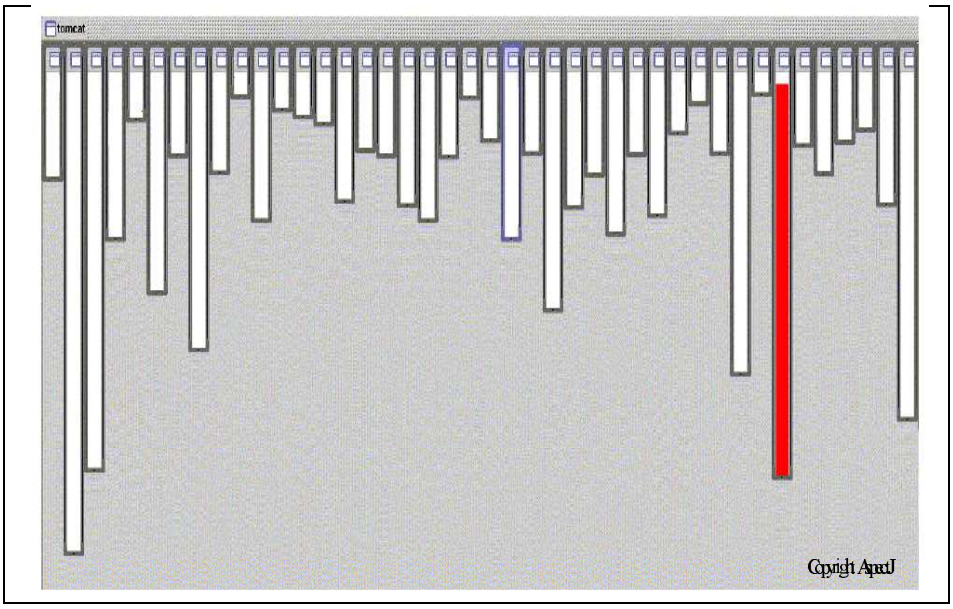

FIG. 2.1 – exemple d'une bonne modularité

Néanmoins certaines parties, intervenant déjà dans plusieurs modules, ne peuvent être modulariser. Concrètement, prenons l'exemple de la fonctionnalité de logging sur une application quelconque. On observe sur le schéma une multitude de traits rouge horizontaux apparaissant dans plusieurs des classes de l'application. Chaque trait correspond à un appel de la fonction logging dans une des méthodes de la classe. On remarque bien la dispersion de code relative à la fonction de logging. Cette application ne jouit pas d'une bonne modularité, pour ce qui est de cette fonction.

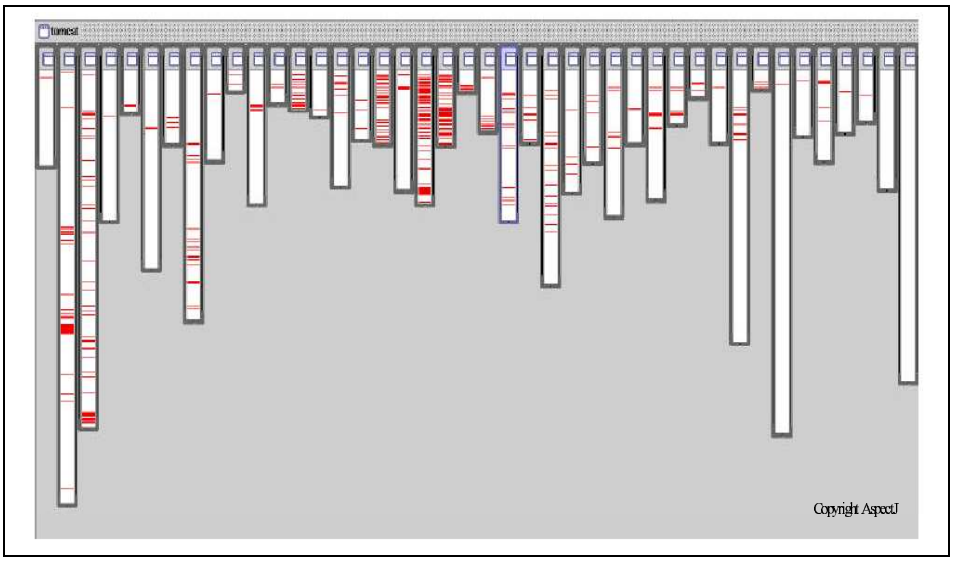

FIG. 2.2 – exemple d'une mauvaise modularité

Ce type de fonctionnalité est dite technique ou non fonctionnelle. La réutilisation d'une propriété non fonctionnelle est d'autant plus difficile que sa définition se trouve dispersée dans l'application. Il est donc difficile de l'isoler pour la réutiliser. En fait une propriété peut concerner plusieurs objets, et le code symbolisant la fonctionnalité pour un objet peut être différent pour un autre. Il est donc difficile de modifier ou de réutiliser ce type de code bien que la politique de cette fonctionnalité soit la même.

2.3 Principes de la programmation par aspects

2.3.1 Les concepts de la programmation par aspects

Une des solutions avancées pour résoudre le problème du mélange et de la dispersion de code des propriétés non fonctionnelles rencontrées avec la programmation par aspects, consiste à découpler leur définition en suivant le principe de la séparation des préoccupations. Partant de ce principe , Kiczales et al. ont formalisé cette séparation ainsi que les différentes étapes de réalisation de l'application pour aboutir au paradigme de la programmation par aspects (AOP : Aspect Oriented Programming)[Kic96]. Ce paradigme de programmation consiste à structurer une application en modules indépendants représentant chacun les définitions des différents aspects ou propriétés non fonctionnelles.

Une application se décompose en deux parties. La première correspond à l'aspect de base. celui ci définit l'ensemble des fonctionnalités offertes par l'application. Spécifiquement il décrit l'application. la seconde partie représente un ensemble d'aspects techniques où chacun définit les mécanismes qui régissent l'exécution de l'application. Ils sont tous indépendants les uns des autres et définissent chacun un mécanisme d'exécution particulier.

Ce type de construction modulaire d'application, nécessite bien évidemment une étape d'assemblage. En effet, chaque module étant indépendant d'un autre, il est nécessaire de les assembler pour construire l'application. Ce mécanisme d'assemblage s'appelle **la composition** (figure 2.3) ou encore "le tissage" (weaving).

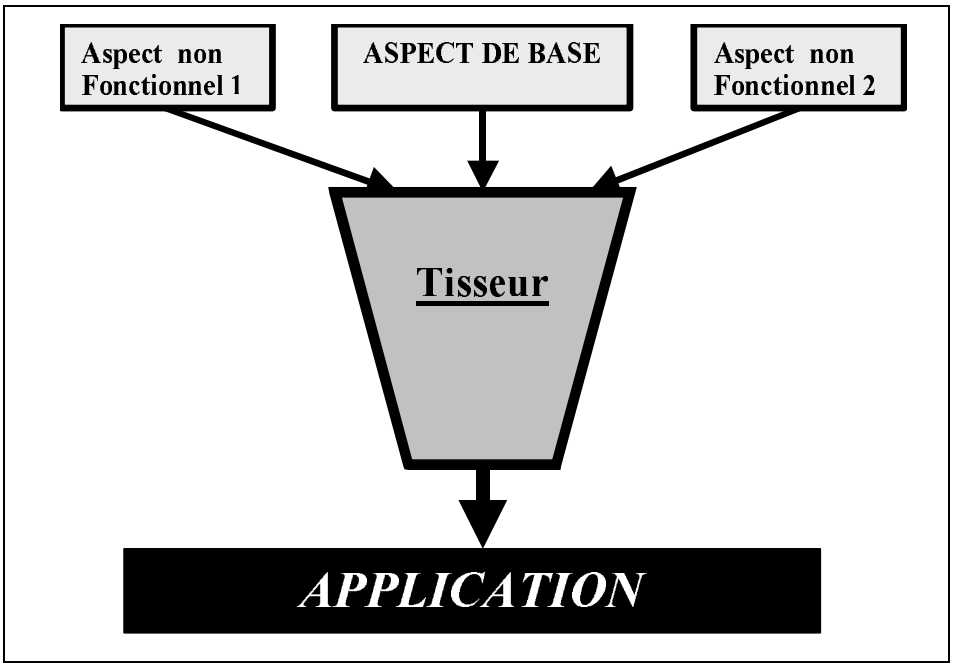

FIG. 2.3 – construction d'une application avec l'AOP

Techniquement, la composition d'aspects se traduit par l'établissement d'une jonction entre les différents aspects de l'application. Cette jonction est établie à partir d'un ensemble de points du flots d'exécution de l'aspect de base appelés **points de jonction** [Kic96]. Tous les points du flot d'exécution constituent des points de jonction potentiels. Clairement, un invocation de méthode, une boucle ou une affectation peuvent être considérer comme des points de jonctions.

La figure 2.4 illustre parfaitement le principe d'une ligne de coupe dans une application. les classes 2, 4 et 5 possèdent toutes un ou plusieurs points de jonctions qui serviront à introduire le code supplémentaire généré par l'aspect après composition.

Pour réaliser cette opération de tissage, les développeurs doivent avoir connaissance de l'ensemble des points de jonction qui seront utiliser pour chaque aspect. Ce processus de définition qui consiste à associer à chaque aspect un ensemble de points de jonction s'appelle **la configuration de l'aspect**.

Le tissage d'un aspect technique et de l'aspect de base peut utiliser plusieurs points de jonction (voir figure 2.4). Inversement un point de jonction peut servir au tissage de plusieurs aspects non fonctionnels. Le problème qui se pose alors, est de pouvoir concilier l'ensemble des aspects sur le même point de jonction. En effet, il s'agit de préserver le

principe de l'exécution de l'application en harmonisant la superposition des aspects, et en respectant la sémantique de chaque aspect. Cette opération de superposition et de composition des aspects constitue l'un des principaux problèmes de la programmation par aspects. Ce problème, aujourd'hui reste ouvert et n'a toujours pas trouver de solution satisfaisante.

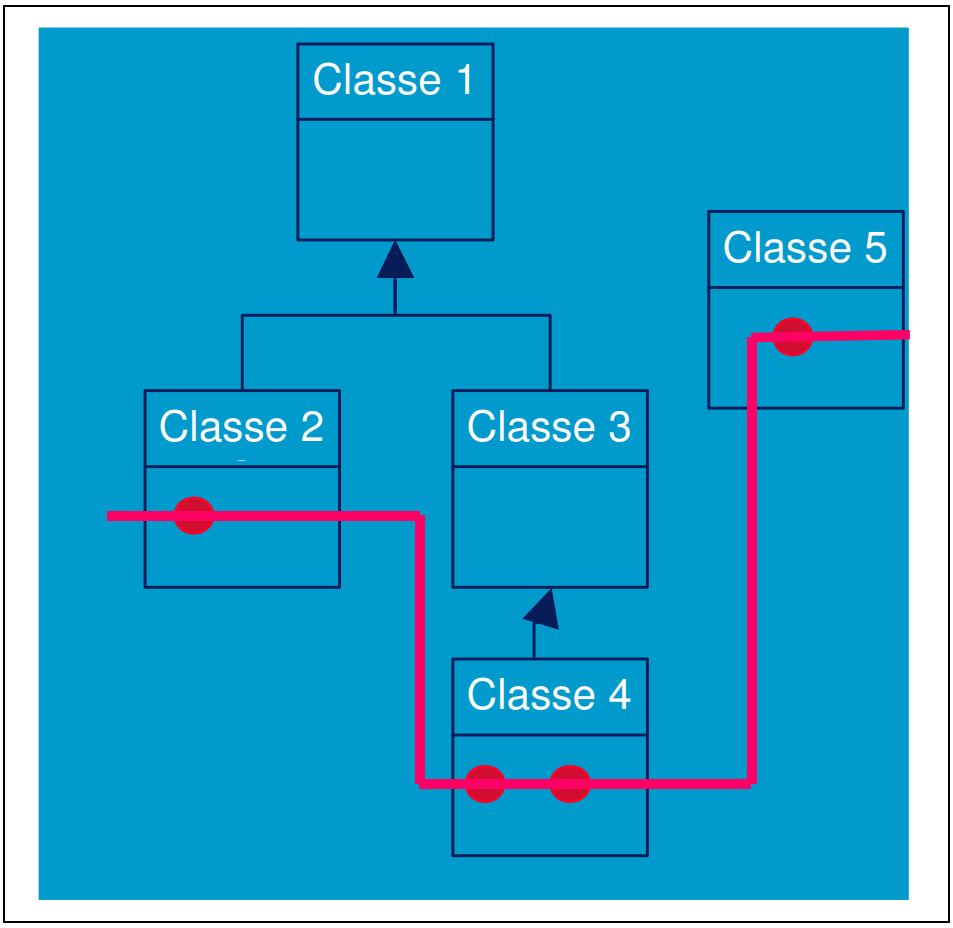

FIG. 2.4 – illustration d'une ligne de coupe dans une hiérarchie de classes

2.3.2 Quelques critères de classification

Le premier critère de classification choisis reprend celui de [BL01], qui consiste à classer les différentes techniques de mise en oeuvre de la programmation par aspects à partir du concept séparant les types d'intervention possibles définis sur le couple <Programme, Interprète> de chaque application. Ce critère consiste à concevoir l'application en temps que *programme* informatique, et à *interpréter* cette notion de programme comme une séquence d'instructions destinée à produire un *résultat*. L'exécution du programme est effectuée par une plate-forme qui interprète la séquence d'instructions. Dans notre cas,

l'aspect de base décrit donc, à partir de cette définition, les services d'une application. Par contre, les aspects techniques interviennent sur la manière de réaliser l'ensemble de ces services.

Décrivons sur un exemple le critère de classification. On peut considérer l'aspect de base comme un programme P en l'absence d'aspects non fonctionnels. Ce programme P est écrit pour son interprète I par défaut. L'exécution de P sur I produit un résultat R. Les aspects techniques interviennent sur l'application de façon à produire un résultat R1 différent de R. l'obtention de ce résultat peut provenir de plusieurs modifications différentes du couple (P,I) : soit en modifiant P, soit en modifiant I ou encore en modifiant les deux. Chacun de ces types d'interventions constituent une catégorie de mise en oeuvre de la programmation par aspects.

Dans la suite nous ne nous intéresserons qu'aux deux premières, qui sont les plus répandues.

2.3.3 Les différentes approches

Description de l'approche par transformation de programme

Si l'on considère une application comme un couple (Programme, Interprète), l'approche par transformation de programme consiste à modifier le programme P1 en un programme P2, pour qu'il fournisse le résultat escompté, à savoir R1. Cette approche consiste à définir un aspect technique comme un ensemble de règles à appliquer à l'aspect de base. Par exemple, si l'on veut écrire un aspect de trace, il suffit d'ajouter des règles de transformation qui se traduiront par l'insertion de code de production de trace au début ou à la fin de chaque méthode, désirant être tracer.

La difficulté majeure de cette approche réside dans la production de ces règles de transformation. Cette modification de programme doit être générique, c'est à dire indépendante de l'aspect de base, afin qu'il puisse être réutiliser. Cet objectif peut être réalisé en paramétrant l'ensemble des règles de transformation. Clairement, les aspects techniques sont définis par un ensemble de points de jonction "abstraits". En effet, ces points de jonction permettent de référencer, sans couplage, des éléments de l'aspect de base. La configuration de l'aspect fonctionnel consiste à lier l'ensemble de ces points de jonction abstraits à l'ensemble des éléments concrets de l'aspect de base. Dès lors que la configuration est terminée, la composition peut avoir lieu, puisque désormais les règles de transformation possèdent "le mapping" leur permettant de s'appliquer sur l'aspect de base.

Le processus de composition des aspects fusionne alors l'ensemble des aspects non-fonctionnels à l'aspect de base. Une fois la composition achevée, le programme est dit fusionné et l'on ne peut plus distinguer de différence entre les lignes de code représentant l'aspect de base et les lignes de code représentant les aspects techniques. Ces lignes sont enchevêtrées de manière à ce que le code de l'application ne forme qu'un bloc rigide, conventionnel à celui qu'il aurait formé si la programmation objet avait été utilisée. Ce processus s'apparente à un outil d'automatisation du mélange de code métier et technique.

Il peut parfois arriver, comme nous l'avons souligné dans la section 2.3.1 que plusieurs

aspects "cohabitent" sur le même point de jonction. Dans ce cas, il faut résoudre le problème de composition d'aspects relatif à ce point de jonction. En effet, l'ordre de réalisation des aspects n'influe pas forcement de la même manière sur le résultat. Il est important de pouvoir spécifier l'ordre d'appel des différents aspects concernés. Pour remédier à cet impératif sémantique, certaines mises en oeuvre, proposent des techniques d'ordonnancement des aspects non fonctionnels [XER00]

Nous étudierons en détail l'une de ces mises en oeuvre, AspectJ [Kic96], dans la section 2.3 de ce chapitre.

Description de l'approche par transformation d'interprète

En considérant un programme informatique comme un couple <Programme, Interprète>, l'approche par transformation d'interprète consiste à modifier l'interprète I1 en un Interprète I2 de sorte que le programme produise le résultat R1. Il s'agit donc de modifier l'interprétation du programme P1 pour qu'il prenne en compte les aspects techniques.

Les interprètes en général, sont représentés par des structures complexes et difficiles à modifier. En effet, choisir les mécanismes internes qui permettront de modifier l'interprète pour qu'il intègre des aspects non fonctionnels constitue une tâche ardue. Il est donc essentiel de définir un cadre qui permettra de guider les transformations. La plupart des approches rencontrées utilise pour cela la réflexion. La réflexion selon Brian Smith [Smi] est *la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même, de la manière qu'elle représente et manipule son domaine d'application premier.* Elle caractérise plus particulièrement la capacité qu'un système a, à raisonner et agir sur lui même. Un Système est donc réflexif, s'il est capable de contrôler son propre fonctionnement et de l'adapter en fonction des besoins du système.

Un système réflexif est capable de séparer les aspects de base et les aspects techniques parce qu'il possède deux niveaux : *le niveau de base et le niveau méta.* le niveau de base représente toujours la description des services à réaliser, et le niveau méta décrit la manière d'interpréter le niveau de base , soit donc les aspects non fonctionnels et l'ensemble des mécanismes utilisés pour la réflexion. Le méta niveau peut être structuré de façon à ce que chaque aspect soit représenté par une classe particulière de ce niveau.

Le niveau méta se compose de plusieurs objets, appelés *méta-objets.* Chacun de ces objets peut être relier à des objets du niveau de base par *un lien méta.* Une fois l'objet de base lié à un méta-objet, celui-ci voit son activité contrôlée par le méta-objet qui lui est associé. Par exemple, l'invocation d'une méthode m sur un objet o, est interprété par le ou les méta-objets correspondants.

Tout comme dans l'approche par transformation de programme, l'ensemble des aspects non-fonctionnels doit être configuré. Cette configuration consiste à définir pour chaque objet du niveau de base, le ou les méta-objets chargés de contrôler son exécution. On enrichit cette définition par la déclaration des points de jonction sur lequel les méta-objets doivent intervenir. On remarquera que cette opération n'affecte pas le niveau de base, et peut être définis indépendamment de celui-ci, hormis pour la désignation des points de jonction, qui appartiennent au niveau de base.

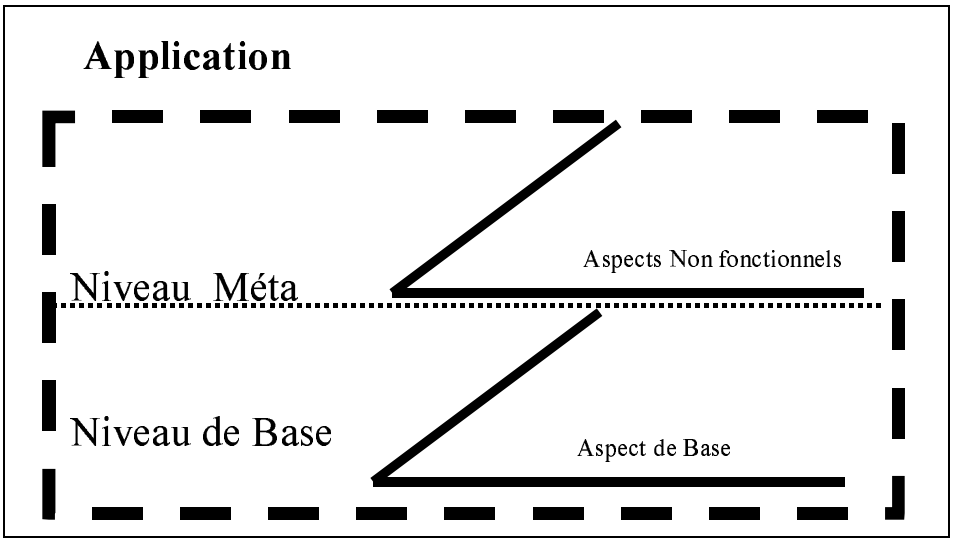

FIG. 2.5 – Correspondance entre aspects et niveaux d'une application réflexive

Le problème principal de cette approche réside dans la composition des méta-objets qui s'inscrivent sur le même point de jonction. Tout comme dans l'approche précédente (i.e. par transformation de programme), il s'agit d'ordonner l'ensemble des méta-objets ou l'ensemble de leurs classes de façon à ce qu'ils ne dénaturent pas l'exécution. Pour cela, les concepteurs proposent plusieurs techniques dont notamment la linéarisation, qui consiste à ordonner les méta-objets dans l'ordre précis d'exécution.

Il existe plusieurs mises en oeuvre de ce type d'approches comme l'extension réflexive de SMALLTALK, MetaclassTalk [Bou99] ou encore JAC [Paw02], un "'FrameWork"' pour l'AOP basé sur le langage JAVA.

2.4 AspectJ : un modèle de mise en oeuvre par transformation de programme

AspectJ, comme nous venons de la voir , est une mise en oeuvre de l'approche par transformation de programme. Dans cette section, nous expliquerons brièvement l'historique d'aspectJ, puis nous nous pencherons sur la traduction en AspectJ, des concepts clés du modèle par transformation de programme. Enfin nous détaillerons l'ensemble des outils fournis dans AspectJ, pour faire de la programmation par aspect.

2.4.1 Historique d'aspectJ

AspectJ est issus des travaux menés sur la programmation par aspects au Xerox-PARC [Kic96]. Il est l'un des langages à expliciter clairement la notion d'aspect tel que nous l'avons définis. Concrètement AspectJ est une extension du langage java qui utilise une hiérarchie

de classes pour définir l'aspect de base. Pour la définition des aspects non fonctionnels, AspectJ utilise une syntaxe spécifique définie dans un langage dédié à la définition et la configuration de ces aspects. A partir de ces constructions, les développeurs peuvent définir les différents aspects techniques par un ensemble de règles de transformation simples.

Initialement, AspectJ a été conçu sans prendre en compte la séparation explicite de la configuration des points de jonction et des aspects techniques. Clairement, les règles de transformation qui représentent les aspects non fonctionnels référencent explicitement les classes ou les méthodes du code de base. Les concepteurs conscients de cette limitation, proposent aujourd'hui, une solution qui passe par la création d'un aspect abstrait permettant de découpler les définitions des aspects, de leur configuration. Ainsi par héritage simple d'aspect, les opérations de configuration et de définitions pourront être effectuées séparément. Cependant, cette solution n'étant pas automatisée, elle dépend essentiellement de la rigueur des programmeurs à l'utiliser.

2.4.2 Le projection des concepts clés d'AspectJ

L'aspect

Dans AspectJ, le mot clé **aspect** représente une entité définie par une classe étendue avec la définition des points de jonction utilisés dans l'aspect, ainsi que les transformations réalisées. Il contient donc la définition du code non fonctionnel (*Advice + introductions*), ainsi que la configuration qui lui est associé (*pointcut + before/after*).

Les pointCuts

La traduction littérale de pointCut signifie *"point de coupe"*. Cette entité est utilisée pour rassembler et identifier l'ensemble des points de jonction relatifs à un aspect donné. Ces ensembles permettent de retrouver rapidement l'endroit ou les parties de code spécifiques devront être ajoutées.

Les règles d'exécution *des Advices* les mots clé before et after

Les règles de transformations sont introduites à partir des mots clé **before et after**. Chaque mot clé désigne le type de transformation à réaliser. Il est suivis du pointcut spécifique à l'action devant être réalisée. Utiliser le mot clé **before** signifie que le code spécifique de l'aspect sera ajouté avant le pointcut et utiliser le mot clé **after** signifie qu'il sera effectué après.

2.5 Conclusion

La programmation par aspects, comme nous avons pu le constater, constitue un outil puissant quelque soit le type d'approche utilisée. Le choix d'une approche particulière, dépend surtout des besoins de l'application. Par exemple, vouloir une gestion dynamique

des aspects impose généralement le choix d'une approche par transformation d'interprète et par conséquent le choix d'une plate-forme réflexive.

Dans la suite de ce document, nous porterons notre attention plus particulièrement à la fois sur les approches par transformation d'interprète pour essayer de conserver une certaine dynamicité dans notre modèle et sur l'approche par transformation de programme d'AspectJ, ce dernier étant à notre avis le modèle le plus abouti.

Troisième partie

Un modèle pour la programmation par aspects généralisés

Chapitre 3

Le modèle

3.1 Motivation : Les points communs entre la programmation par aspects et la représentation multiple

Dans cette section, nous détaillerons l'ensemble des points communs relatifs aux deux concepts étudiés dans la partie précédente, à savoir les similitudes existants entre les approches de représentation multiple et de la programmation par aspects. En fait ces deux paradigmes partagent souvent les mêmes objectifs, les mêmes démarches pour y parvenir, la même structuration de modèle adjacent (concepts, lien entre les concepts), et aussi les techniques utilisées pour mettre en oeuvre ces modèles.

3.1.1 Objectifs identiques

L'étude des deux paradigmes, que sont la représentation multiple et la programmation par aspects, conduit à un premier constat : les deux paradigmes ont le même objectif, à savoir la réutilisation. En effet, le but premier de la programmation par aspects est de produire un code suffisamment générique pour qu'il puisse être réutilisé. Il en est de même pour les modèles de représentation multiple, même si leur objectif principal reste la simplification et la vitesse de construction des applications.

En plus de la réutilisabilité, la programmation par aspects comme les modèles de représentation multiple vise aussi à produire du code plus rapidement. Le fait, de séparer les fonctions métier du code de base a pour effet tout comme la séparation des préoccupations, de simplifier la création de code et donc d'accélérer sa production.

La modularité fait aussi partie des objectifs communs. En effet, l'éclatement du code métier en programmation par point de vue et la séparation des aspects en programmation par aspects on le même but, qui est de modulariser l'application. Cette modularisation dans les deux cas vise à augmenter la réutilisabilité , mais surtout assure une meilleure maintenance de l'application. Il est plus facile d'intervenir sur de petites portions de code que sur un programme compact.

3.1.2 Modèles conceptuels semblables

La liste des points communs entre ces deux paradigmes ne s'arrête pas aux objectifs, elle concerne aussi les modèles correspondants(concepts et structuration des concepts).

En effet, comme on a pu le constater dans la première partie, les points de vue sont constitués de plusieurs entités distinctes. Suivant le niveau de granularité que l'on choisit, le nombre d'entités varient. Si l'on se place au niveau le plus haut, on remarque deux entités spécifiques qui sont le référentiel et le point de vue. Le référentiel rappelons le, correspond aux parties communes de l'ensemble des point de vue, et le point de vue correspond à un traitement spécifique de l'application. Similairement, en programmation par aspect, on retrouve cette granularité entre l'aspect de **base** et les aspects non **fonctionnels**. L'aspect de base définit l'ensemble des services, alors que les aspects techniques, définissent les mécanismes qui régissent l'exécution. En prenant l'exemple de CROME, la similitude s'accentue puisque le vocabulaire employé est similaire : le référentiel est appelé plan de **base** et les points de vue, plans **fonctionnels**.

Les points communs architecturaux ne s'arrête pas à ce niveau de granularité, si l'on reprend l'exemple de CROME, on s'aperçoit que le choix d'architecture à savoir centralisé orthogonal, a son équivalent en programmation par aspects. Elle considère notamment que chaque aspect doit être indépendant des autres aspects de l'application. L'indépendance signifie que la définition d'un aspect ne dépend pas de la définition des autres aspects de l'application, autrement dit elle équivaut à l'orthogonalité.

Les mécanismes d'intégration pour les points de vue et de tissage pour les aspects, sont similaires. Plus précisément, l'intégration consiste à assembler les points de vue et le référentiel pour former l'application. Identiquement, le tissage compose les aspects non fonctionnels et l'aspect de base pour créer l'application.

3.1.3 Techniques similaires

Il existe d'autres similitudes entre les modèles de représentation multiple et la programmation par aspects. Comme nous l'avons vu précédemment, les modèles à représentation multiple peuvent être dynamiques ou statiques, ceci en fonction de la considération dans le modèle de la notion de point de vue. En effet, si le modèle préserve cette notion après intégration, il offre la possibilité d'être dynamique, à condition que proposer les outils adéquates (ajout-suppression de point de vue en cours d'exécution...). Il en est de même pour les différentes approches de la programmation par aspects. Si l'on reprend les deux approches étudiées précédemment, on comprend que la transformation de programme est une approche statique et que l'approche par transformation d'interprète se traduit généralement par une approche dynamique. En effet, le fait de transformer le programme source avant l'étape d'interprétation, ne permet pas de retrouver en cours d'exécution les aspects, puisqu'ils sont "fondus" dans un nouveau programme. Alors que l'approche par transformation d'interprète, autorise cette dynamicité, puisqu'elle est basée sur une modification de l'interprète et que l'ajout ou la suppression d'aspects ne modifie pas l'interprète. La dynamicité constitue donc une similitude supplémentaire entre les modèles de représentation

multiple et la programmation par aspects.

Synthèse et motivation

Ces deux approches (i.e la programmation par aspect et la représentation multiple) possèdent pratiquement les même objectifs, à savoir la réutilisabilité, la vitesse de développement et la modularité. Elles sont similaires sur le plan structurelle. On retrouve notamment le concept de granularité relatif au entité des modèles (base-méta et référentiel-point de vue), la contrainte d'orthogonalité et surtout un mécanisme d'intégration ou de tissage comparable. Les techniques employées se recoupent aussi. La modularité et la dynamicité en sont les preuves.

C'est à partir de ce constat que nous avons envisagé de regrouper ces deux concepts que sont la représentation multiple et la programmation par aspects. Néanmoins même si ces deux types d'approches sont similaires sur bien des points, leur rapprochement engendre quelques questions auxquels il nous faudra répondre au cours de l'établissement du modèle.

La première de ces questions, concerne l'unification des deux concepts et plus particulièrement, il nous faudra faire la lumière sur les liens existant entre les différents niveaux de description (fonctionnels et non fonctionnel). Comment effectuer l'intégration d'aspect fonctionnels tout en prenant en compte les aspects non fonctionnels ?

Les questions suivantes reposent principalement sur la gestion de la cohérence et plus précisément se résume par l'assurance de la gestion de la cohérence entre les différents aspects de niveau de base, sachant que leurs données pourront être modifier par les aspects de niveaux méta. Ces questions constituent les problèmes majeures qu'il nous faudra résoudre dans l'établissement du modèle.

3.2 Notions de base et modèles de référence

3.2.1 Les méta objets : support pour la programmation par aspects

Avant de décrire et d'expliquer l'ensemble des choix que nous avons réalisé pour faire notre modèle, il est important de définir les notions qui nous ont servis de base pour la construction de celui-ci.

Introduction

Pour comprendre la notion de méta-objet, il faut avant tout définir la notion de langage réflexif, ainsi que la notion de langage réflexif à objet. Une fois ces deux notions définies, nous donnerons la définition du méta objet.

Comme nous avons pu le constater au cours de l'état de l'art sur les aspects, la structure d'un système informatique est composé d'un ensemble de données, du programme et de l'exécutant et il s'exécute en agissant et raisonnant sur une "partie du monde". On rappelle aussi que la réflexion selon Brian Smith [Smi] est *la capacité d'une entité à*

s'auto-représenter et plus généralement à se manipuler elle-même, de la manière qu'elle représente et manipule son domaine d'application premier. Un système réflexif est donc un système qui a la capacité de s'auto-observer et donc de pouvoir raisonner sur lui même et de s'auto-modifier, donc d'altérer son comportement et sa structure. Il est dit réifié, lorsque l'on peut représenter sous forme explicite les éléments constituant le programme et l'exécutant. On définit donc un langage réflexif comme un langage dont les constructions et l'interprète sont réifiés.

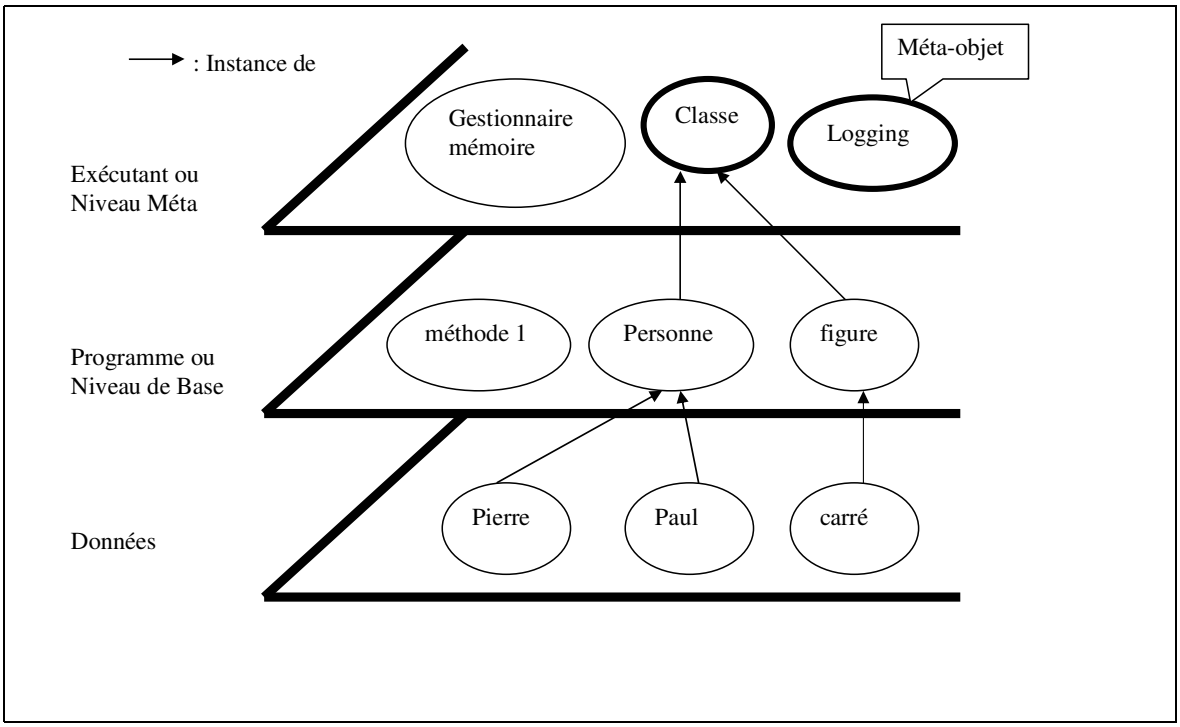

FIG. 3.1 – Système dans un langage réflexifs à objets.

La différence entre un système réflexif "classique" et un système réflexif à objets se situe principalement dans la séparation, sur deux niveaux de programmation, du programme et de l'exécutant. Dans ce type de système, chaque objet est une instance d'une classe. Chaque classe est elle aussi considérée comme un objet et instance de classes appelées méta-classes. Ainsi comme on peut le voir sur le schéma 3.1, il existe deux niveaux dans les systèmes réflexifs à objets le niveau de base et le niveau méta.

Définition et principes de fonctionnement

Dans ces systèmes, l'exécution est à la charge du méta-objet. Un méta-objet est définis comme un objet du niveau méta qui contrôle l'exécution, comme par exemple les messages et les accès au champs du programme. Un langage réflexif à objets offre la possibilité de lier les objets de base à des méta-objets particuliers.

Ainsi comme le montre la figure 3.2, le fait de lier l'objet Pierre au méta-objet de log définis, produit une double exécution à la fois sur la console, exécution relative au programme de base, et dans le fichier de log correspondant à la classe du méta-objet de log : *LogMetaObject*. En fait, en recevant le message *ditBonjour* contenu dans la classe *Personne*, le méta-objet *méta-objet de log* relatif à l'objet Pierre exécute par le biais de sa méthode *receive* contenu dans la classe de méta objet *LogMetaObject* , le traitement d'ajout dans un fichier puis libère la capture du message en exécutant le message d'origine, ce qui a pour effet d'écrire bonjour sur la console.

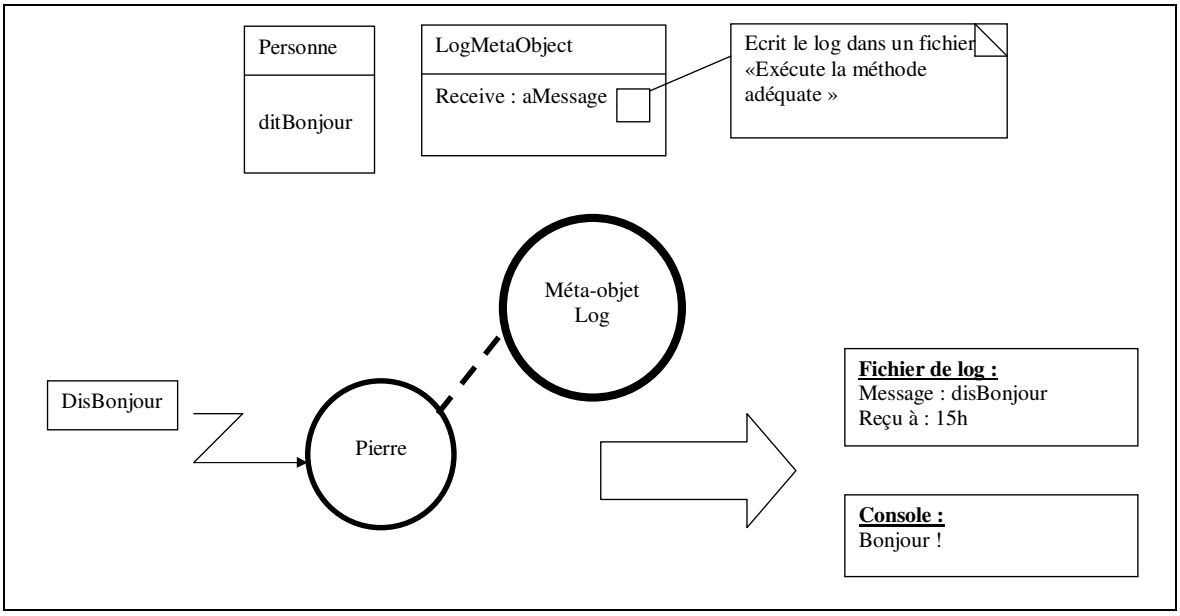

FIG. 3.2 – Exemple du méta-objet de log

L'utilisation de langages réflexifs apporte de nombreux avantages comme la flexibilité, l'adaptabilité et la simplicité. En effet, ils offrent par exemple la capacité d'évolution aux systèmes qui les utilisent, mais aussi la possibilité de produire du code plus générique et de séparer ainsi les différents types de code (fonctionnels et techniques), ce qui est précisément le but recherché.

3.2.2 Les mixins : outil pour l'héritage multiple

La notion de mixin est elle aussi une notion capitale qu'il faut comprendre avant d'aborder le modèle, puisque c'est à partir de cette notion et plus précisément de sa faculté à concevoir l'héritage multiple qui a été utilisée pour la réalisation de notre modèle.

Définition

Un mixin est une sous classe paramétrable par une super-classe, ce qui lui confère la possibilité d'être réutilisée avec différentes hiérarchies de classe. Le concept de mixin est apparu dans CLOS [Kee89] comme un style de programmation. Il a été introduit pour résoudre les problèmes causés par l'héritage multiple et la linéarisation automatique des hiérarchies de classes.

Principes de fonctionnement

Nous nous sommes principalement inspirés du modèle de mixin proposé par [Bou03]. Pour comprendre et illustrer le principe de l'héritage à base de mixin nous utiliserons l'exemple suivant :

Supposons que l'on veuille définir une sous classe S qui hérite à la fois de C une classe non mixin et deux classes M1 et M2 mixins. La figure 3.3 illustre les relations entre les classes et les mixins. Chaque classe possède respectivement un ensemble de champs et de méthodes.

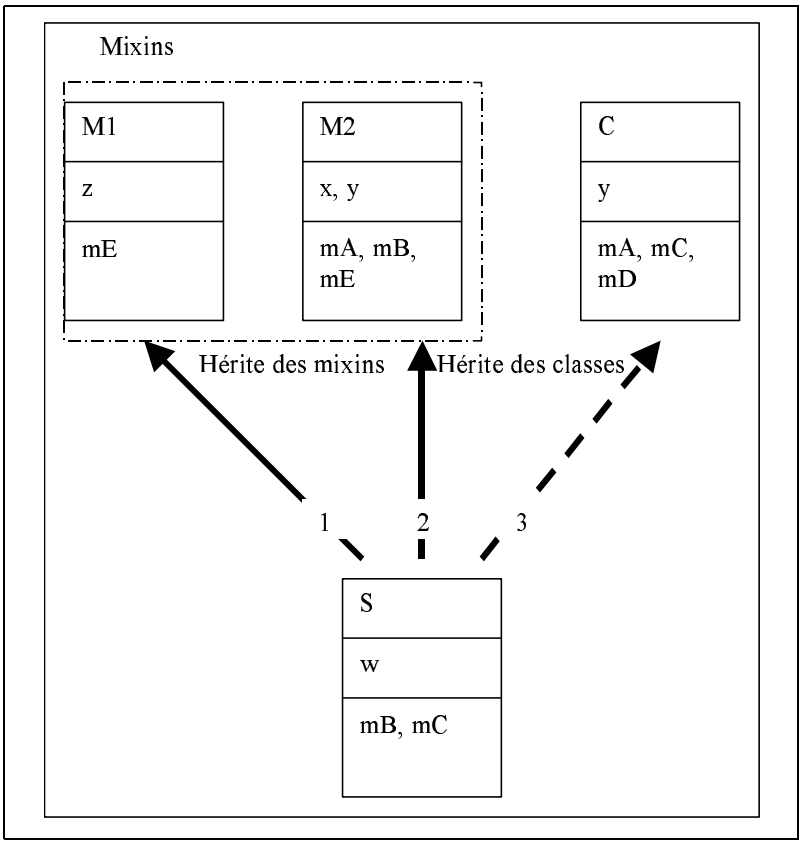

FIG. 3.3 – Exemple d'héritage multiple à base de mixins

Les flèches représentent les relations d'héritage et sont numérotés pour montrer explici-

tement le principe de linéarisation. Clairement, S hérite successivement de M1 puis de M2 et enfin de C. L'évaluation d'un message "super" dans un mixin, a pour effet d'appeler le mixin suivant de la chaîne de linéarisation. Au cas ou le mixin serait le dernier de la chaîne, effectuer un super revient à envoyer le message super à la super classe de S, soit donc C. Supposons que la méthode mB contienne le message super mB, le traitement sera le suivant le super à pour effet de chercher la méthode dans la chaîne de mixins. M1 ne contenant pas la méthode, il passe la main à M2 qui lui contient la méthode et qui l'effectue.

L'utilisation de ce style de programmation pourrait provoquer des problèmes de masquage de champs ou de méthodes, qui deviendraient inaccessible. Pour palier à cette éventualité, l'héritage à base de mixins proposé par [Bou03] est régit par trois règles :

1. Les méthodes définies par une classe précèdent les méthodes définies par l'ensemble de ces super classes.
2. Les méthodes définies par une super classe mixin précèdent celles définit par les super classes non mixins.
3. Dans une liste de mixins, les méthodes définies par les mixins du début de liste précédent les méthodes définies par les mixins en fin de liste.

C'est trois règles suffisent à résoudre l'ensemble des problèmes qui pourraient survenir lors de leur utilisation.

3.2.3 Nos modèles de référence : les modèles de programmation par contextes et aspects

Le constat réalisé dans la première partie de ce chapitre à montrer que les modèles à représentation multiple et la programmation par aspects avaient beaucoup de points communs, tant à la fois au niveau des objectifs, qu'au niveau conceptuel. Au cours de cette étude, nous avons relevé que le modèle de programmation par contexte (i.e CROME) avait bien plus de points communs avec les aspects que les autres approches à représentation multiple. Comme nous l'avons vu précédemment, CROME possède l'orthogonalité, qui est un concept majeur dans la programmation par aspects. De plus son vocabulaire très proche de celui des aspects, montre cette volonté de similitude entre ces deux concepts. C'est pourquoi, il nous est apparu logique de partir de cette approche comme point de départ pour la représentation multiple de notre modèle.

Pour ce qui est de la programmation par aspects, il nous est apparu évident de nous tourné vers une approche par transformation d'interprète puisque nous voulions garder une certaine dynamicité. Pour cela nous avons étudier plus particulièrement une des implémentations réflexives de SMALLTALK : **MetaclassTalk**. Néanmoins cette plate-forme ne disposant pas d'un modèle complet pour la programmation par aspects, nous avons garder AspectJ comme modèle de référence. Et c'est à partir de ce modèle que nous avons dégagé l'ensemble des outils nécessaires à l'établissement d'un modèle de programmation par aspects.

Les caractéristiques imposées par les deux modèles

L'ensemble des caractéristiques qui suivent, constituent les concepts essentiels qui font respectivement de ces approches (contexte et aspect), un modèle de représentation multiple et un modèle de programmation par aspect.

- **La projection du modèle de programmation par contexte**
  - Pour commencer notre modèle devrait posséder une structure similaire au modèle de programmation par contexte. Concrètement le modèle devrait être centralisé orthogonal. Bien que cette caractéristique ne soit pas primordial au fonctionnement du modèle, il est important dans un premier temps de suivre ce concept, et si nécessaire de le faire évoluer. En effet, un modèle orthogonal peut devenir non orthogonal par la mise en place de canaux de communications inter points de vue (inter contextes). Un modèle centralisé peut évoluer en un modèle décentralisé en supprimant la notion de référentiel et en l'introduisant dans les différents points de vue. La gestion de la cohérence est alors effectué par la mise en place de canaux de communications.
  - Notre modèle devrait posséder ensuite un outil d'intégration qui permettrait de réaliser l'opération de fusion entre les points de vue et le référentiel. Cet outil d'intégration devra évoluer en fonction de l'évolution du modèle.
  - Ce modèle devrait gérer le partage d'informations entre les différentes entités définies. Il devrait notamment considérer :
    - le partage d'attributs (champs) du plan de base (référentiel).
    - le partage de méthodes du plan de base.
    - le partage d'information entre objets de même plan fonctionnel (contexte - point de vue).
    - le partage d'information entre objets de plans fonctionnels différents.
  - Il devrait aussi gérer l'ensemble de la cohérence des données partagées entre les contextes (réalisé en CROME par contraintes et traitements compensatoires).
- **La projection du modèle de programmation par aspects**
  - Le modèle devrait bien évidemment contenir la notion d'aspect, et tout ce qui ci rapporte, comme les points de coupe et les points de jonction. La notion d'advice est elle aussi très importante puisqu'elle définit spécifiquement le code de l'aspect. L'ensemble de ces notions constitue les entités de base qu'il nous faudra définir pour montrer que notre modèle intègre la programmation par aspects.
  - Ce modèle devrait aussi tenir compte des principes de ce type de programmation comme l'héritage d'aspect et plus précisément le souci de réutilisabilité des aspects. Nous avons vu que l'un des buts de la programmation par aspects est de pouvoir réutiliser les aspects existants afin de gagner du temps lors de la phase de développement.
  - Il devrait aussi prendre en considération la possibilité d'enrichissement du code de base et du code technique. Cet enrichissement se traduit principalement par l'utilisation d'un script de configuration et est utilisé pour spécialiser et paramétrer.
  - Enfin notre modèle devrait intégrer la notion d'aspects partagés. Plus précisément, il devrait fournir la fonctionnalité de partage d'un ou plusieurs aspects entre plusieurs

entités de l'application. En clair, le partage d'un aspect correspond au partage des données relative à l'exécution de l'aspect (l'équivalence des clauses "perX" en aspectJ). Par exemple partager un aspect de trace, c'est offrir la possibilité de tracer l'ensemble des messages reçus ou envoyés des différentes instances , dans un même fichier ou de séparer ces traces dans des fichiers spécifiques pour chacune d'elles. Le schéma 3.4 résume cet exemple.

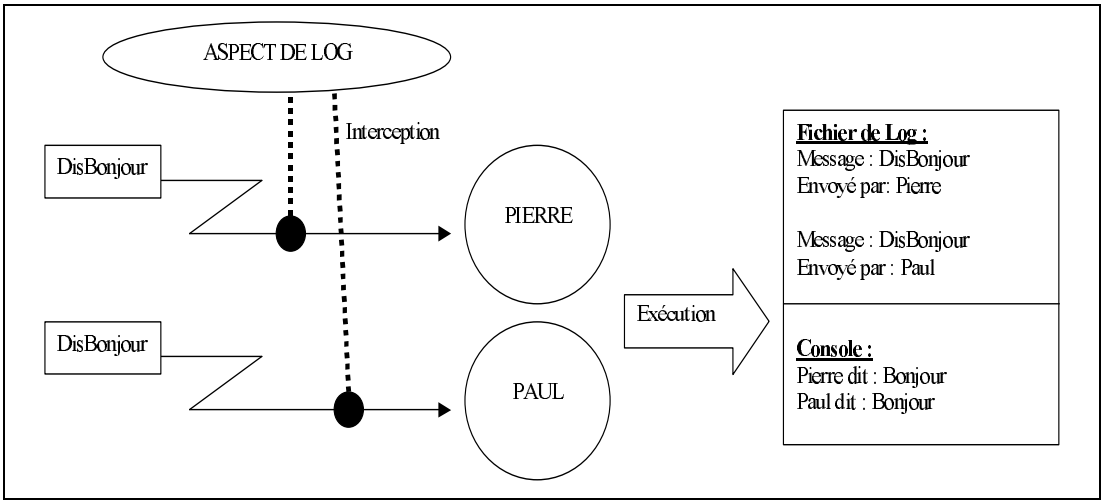

FIG. 3.4 – Exemple de partage d'aspects

Si Pierre et Paul partagent les données relative à l'aspect de trace, lorsque Pierre et Paul envoient le message ditBonjour, le fichier de trace doit contenir la trace des deux envois de message. S'ils ne les partagent pas, les traces sont dans deux fichiers différents.

3.3 Description du modèle

L'ensemble des bases de la programmation par aspects et des concepts de la représentation multiple à base de contexte étant posé, nous pouvons maintenant définir notre modèle. Dans un premier temps, nous définissons l'ensemble des entités nécessaires à l'élaboration d'un modèle de généralisation du concept d'aspect et décrirons son fonctionnement. Ensuite, nous comparons le modèle et ses propriétés avec les modèles étudiés auparavant. Enfin nous discutons les différents avantages de l'approche choisie.

3.3.1 Définitions

Représentation

Pour commencer nous définissons l'entité la plus générale de notre modèle qui est **la représentation**. Une représentation correspond à une vue de l'univers de discours (application). Cette vue peut être particulière, dans ce cas elle correspond à un point de

vue, mais elle peut aussi être générale auquel cas elle désigne le référentiel de l'application. Une application est donc constituée d'un ensemble de représentations.

Référentiel

La seconde entité que nous définissons est **le référentiel**. Il regroupe l'ensemble du code générique, commun à l'ensemble des points de vue de l'application. Il correspond au code métier de l'application. On représente ce code par la hiérarchie de classes, relative à la définition objet de l'application.

L'aspect de base lui n'a pas de correspondance directe dans notre modèle, néanmoins il correspond au regroupement des aspects fonctionnels et du référentiel, comme le montre la figure 3.5. Cette figure répond en partie à la question posée sur le rapprochement des deux concepts. Elle montre comment s'imbrique les entités des deux modèles de référence au travers des niveaux de base et méta.

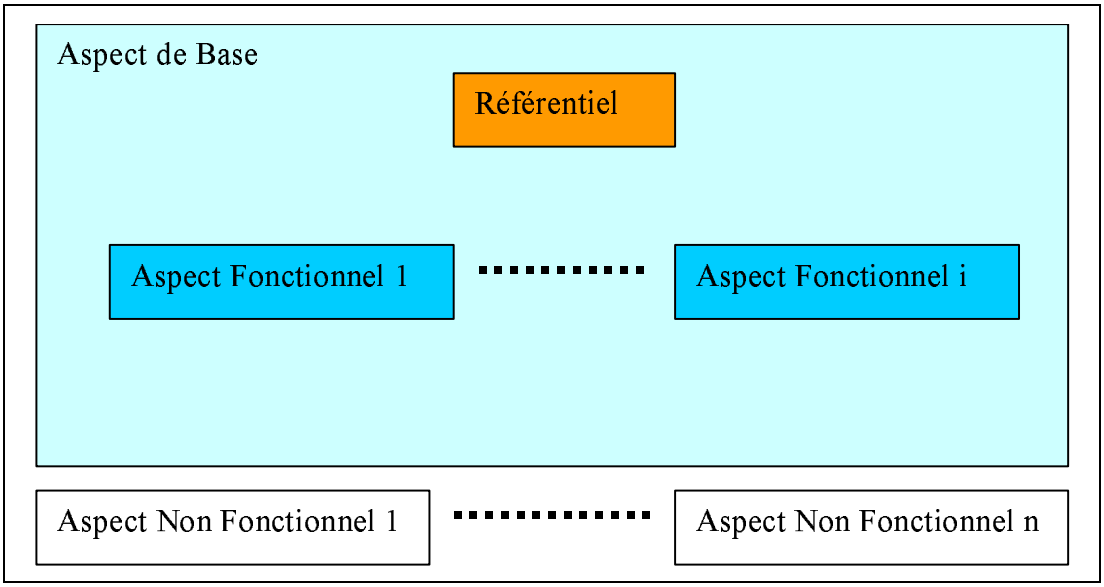

FIG. 3.5 – Correspondance des concepts d'aspects fonctionnels et non fonctionnels avec ceux de la programmation par aspects

Aspect Généralisé

Les notions de points de vue et d'aspects que nous avons rencontrés au cours de l'état de l'art ont été regroupés autour d'une seule entité appelée **aspect généralisé**. Similairement à CROME et aux *Advices* définis dans la programmation par aspects, il est composé d'un ensemble de parties fonctionnelles (CROME) ou non fonctionnelles (Advice) correspondant respectivement à du code métier ou du code technique.

La figure 3.6 montre qu'une intégration regroupe l'ensemble des parties fonctionnelles d'une classe ainsi que son code de base. Les aspects non fonctionnels s'appliquent alors sur ce regroupement.

Dans cette exemple en intègre la classe C1 avec ses parties fonctionnelles A1 et B1. Sur ce code généré, on déploie l'aspect généralisé 3 relatif au code méta. Chaque instance de l'application (O1 et O2) possède un méta-objet (MO1 et MO2) qui permet d'appliquer l'aspect généralisé 3 à l'ensemble de la classe C1. Ces méta-objets sont instances d'une méta-classe contenant la description de l'aspect et sont référencés dans la classe C1. Cette méta-classe est donc spécifique à la classe C1. Le lien défini en pointillé sur notre schéma constitue le lien méta, il permet essentiellement d'associer à un objet, un méta-objet qui contrôlera son exécution. Ce lien méta sert de correspondance entre le niveau fonctionnel (et les mixins de base) et le niveau non-fonctionnel (et les mixins méta).

Cet exemple traduit parfaitement le regroupement des deux concepts et répond spécifiquement au premier problème posé (i.e regroupement des deux approches).

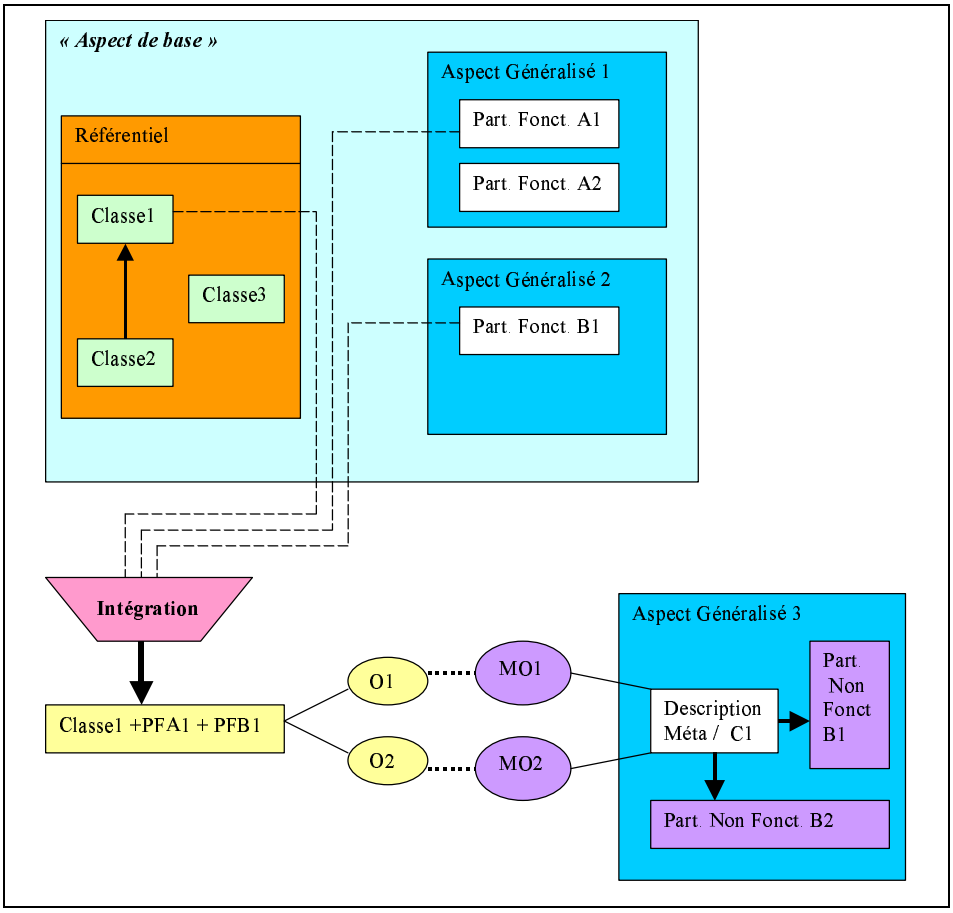

FIG. 3.6 – Représentation générale des concepts de base du modèle

Mixins

Chaque partie, quelle soit fonctionnelle ou non, est représentée par un ou plusieurs **mixins**. Il possède précisément un ensemble de mixins relatif au points de vue (mixins de niveau de base) et un ensemble de mixins relatifs aux aspects (mixins de niveau méta).

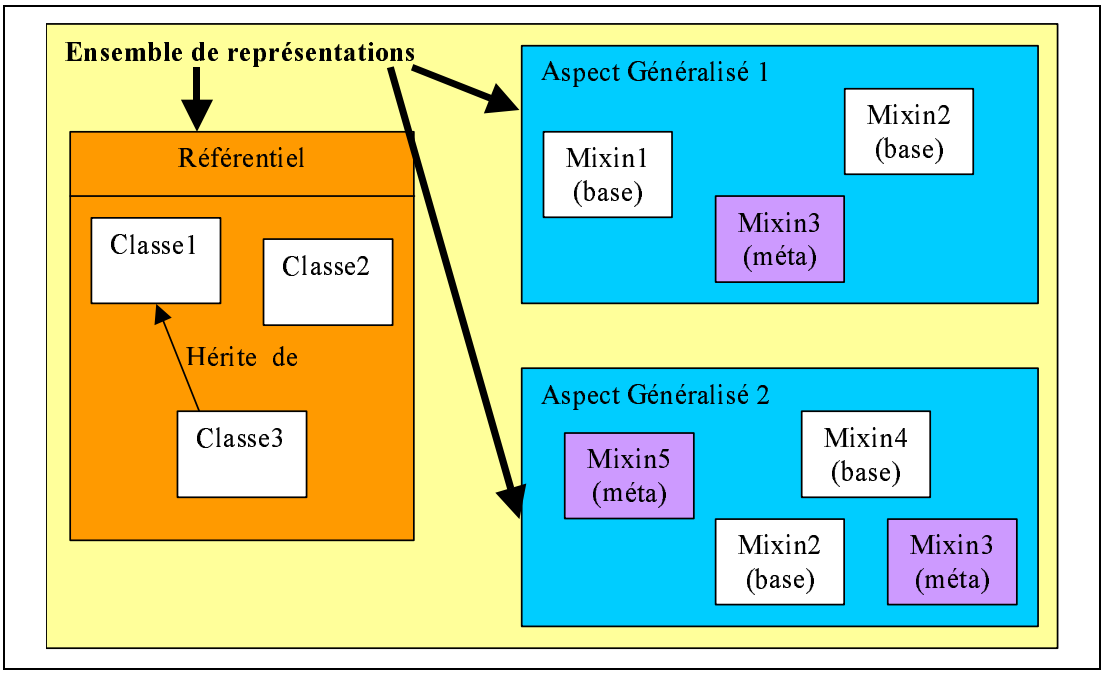

FIG. 3.7 – Structure de base de notre modèle

Ainsi comme on peut le constater sur la figure 3.7, notre modèle est composé d'un ensemble de représentations contenant un référentiel et un ensemble d'aspects généralisés. Chaque aspect généralisé est constitué de deux ensembles de mixins : les mixins de niveau de base et les mixins de niveau méta. On remarquera sur le schéma que l'aspect généralisé contient aussi bien du code métier, que du code technique, ce qui renforce l'idée de généricité de l'aspect.

Intégrateur généralisé

Pour construire une application à partir des éléments définis, notre modèle utilise un outil d'intégration appelé : **intégrateur généralisé**. C'est à partir de cet outil que la fusion est réalisée. Il est chargé de mettre en correspondance l'ensemble des classes du référentiel et les mixins appartenant aux aspects généralisés. En fait, il utilise un dictionnaire d'intégration définit par l'utilisateur. Ce dictionnaire contient un ensemble de couple (Classe-Mixin).

3.3.2 Principes et mécanisme général d'intégration

Comme on peut le constater sur le schéma 3.8, pour pouvoir intégrer une application il faut fournir un certain nombre d'entités à l'intégrateur généralisé.

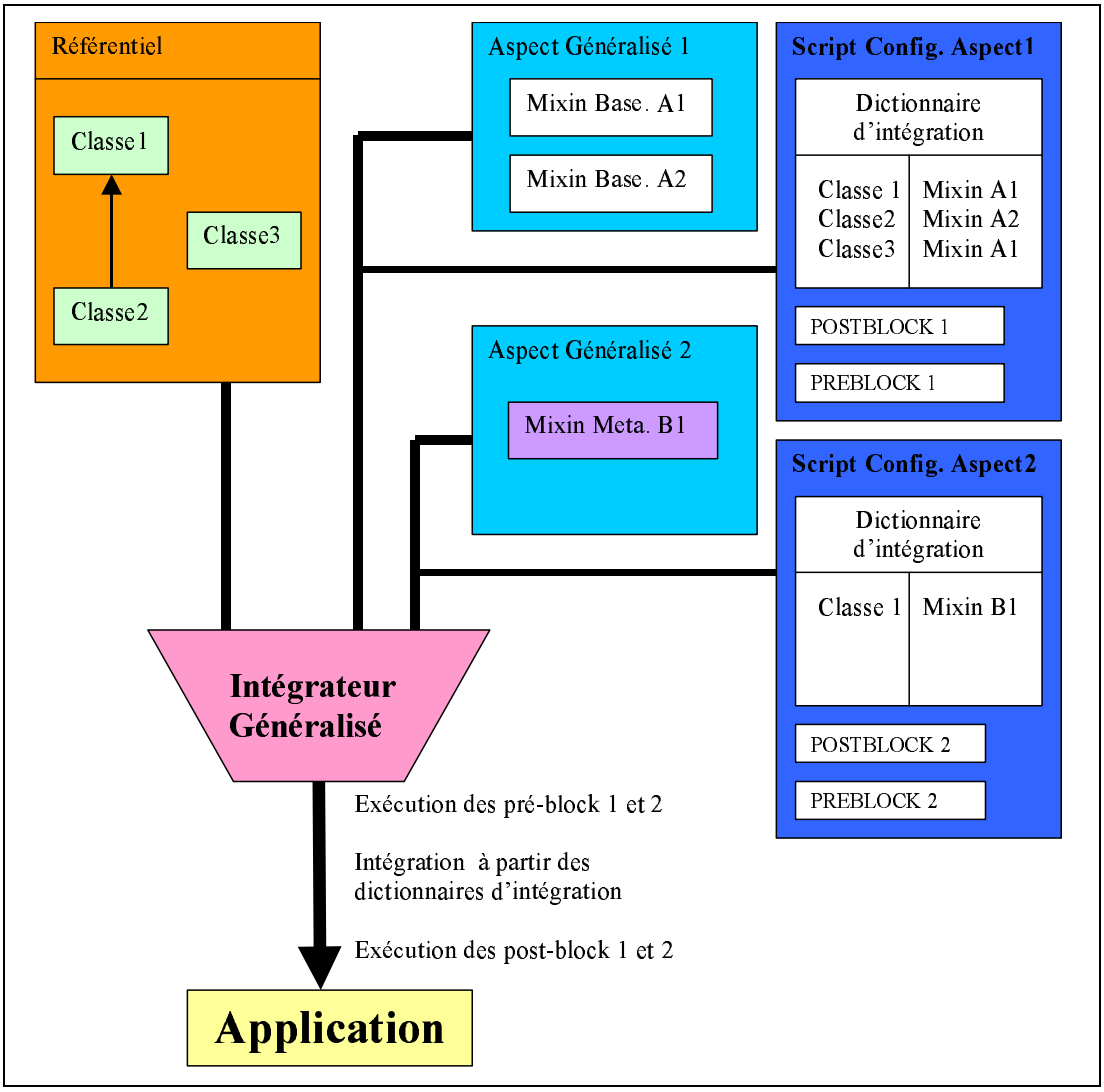

FIG. 3.8 – Schéma général du mécanisme d'intégration

Tout d'abord il faut lui fournir la hiérarchie de classes représentant le référentiel. On rappelle que cette hiérarchie contient uniquement les classes génériques (communes à tous les aspects) de l'application. Il faut ensuite lui fournir un ou plusieurs aspects généralisés. Chacun de ces aspects contient un ensemble de mixins correspondant chacun à une partie de code spécifique de l'aspect généralisé. A chaque aspect généralisé, on doit fournir un dictionnaire d'intégration, contenant un ensemble de couple (classe-mixin) correspondant à l'intégration spécifique du mixin sur la classe dans l'aspect généralisé. Ce dictionnaire

d'intégration fait partie d'une entité plus générale appelée **script de configuration**. Un script de configuration est spécifique à un aspect généralisé. il contient généralement une partie de code propre à l'application développée. Il est contenu dans des blocs s'exécutant avant ou après l'exécution (pré ou post bloc). Un pré-bloc s'exécute juste avant l'intégration de l'aspect sur l'application, un post-bloc après. Ces deux entités bloc permettent de spécifier le tissage de l'aspect.

Une fois l'ensemble de ces entités créées, on peut commencer le processus d'intégration. Pour pouvoir être intégrer un aspect généralisé doit valider un certain nombre de tests qui garantissent principalement la conformité d'intégration. L'ensemble de ces tests sera détaillé dans le paragraphe suivant. Pour commencer, l'intégrateur exécute l'ensemble des préconditions d'intégration contenues dans les pré-blocks, puis démarre les différents tests. Une fois ces tests validés, on poursuit le processus en ajoutant pour chaque classe appartenant au référentiel (classe générique), le mixin (une partie de code spécifique) qui lui est associé dans le dictionnaire d'intégration. On termine le tissage d'un aspect en exécutant les post-conditions d'intégration contenu dans les post-blocks. On renouvelle se processus pour chaque aspect de l'application. L'intégration terminée, chaque classe contient son code générique ainsi qu'un ensemble de parties de codes spécifiques relatives aux mixins ajoutés.

3.3.3 Les règles d'intégration

Pour s'assurer que le processus d'intégration se déroule correctement, il doit tout d'abord répondre à un ensemble de critères contrôlés chacun par une règle d'intégration. Notre modèle, similairement à celui de CROME ne comporte qu'une seule contrainte spécifique au processus d'intégration. En effet, il doit s'assurer de **l'orthogonalité des aspects généralisés** définis dans l'application. Pour cela, il doit précisément vérifier que deux mixins associés à une même classe de base n'ont pas de méthodes ou de champs de même signature ou de nom. La vérification de cette contrainte constitue le test d'orthogonalité.

Il existe deux autres règles à vérifier pour valider la phase de tests précédant le processus d'intégration. Il s'agit des règles **d'intégrité et de compatibilité**. Le test de compatibilité permet de vérifier l'ensemble des entités passées en paramètre à l'intégrateur. Il contrôle la cohérence des données présentes dans les dictionnaires d'intégration avec celles fournis dans les différentes représentations. Le test d'intégrité vérifie la validité des ensembles de données fournies. Il contrôle principalement la cohérence des tailles des ensembles passés en paramètre à l'intégrateur.

Notre modèle comporte donc 3 règles spécifiques que sont l'intégrité, la compatibilité et l'orthogonalité. De ces 3 règles, seule la dernière constitue un contrainte d'intégration relative au modèle, les autres sont indépendantes du modèle.

3.3.4 Les mécanismes d'intégration : l'ajout de mixins

La description qui vient d'être faite, constitue le principe général de l'intégration. Il faut maintenant expliquer pourquoi l'ajout d'un mixin sur une classe, permet l'ajout d'un code spécifique au code générique de la classe du référentiel. Pour cela on se doit de distinguer

les deux types de mixins existants : les mixins de niveau de base et les mixins de niveau méta.

les mixins de base

L'intégration d'un mixin de niveau de base a pour effet de créer une relation de multi-héritage entre la classe, ses super-classes et le mixin. Cette relation permet donc d'hériter au niveau de la classe du code du mixin. Comme nous l'avons vu dans la paragraphe dédié aux mixins, ce sont des sous classes paramétrables par des super classe. Ajouter un mixin sur une classe c'est le définir clairement comme une super-classe de cette classe.

Comme nous l'avons constaté précédemment, l'ajout de plusieurs mixins nécessite une organisation particulière afin de respecter l'ordre d'héritage et plus précisément dans l'ordre d'ajout des mixins. Dans notre modèle, cet ordre n'a pas d'importance. En effet, étant régis par l'orthogonalité, les mixins définis sur une classe générique sont indépendants deux à deux. Cette affirmation garantit l'unicité des signatures et des noms des entités (méthodes et champs) des mixins définis sur cette classe.

L'ajout de mixins de base sur un classe provoque donc l'ajout de code spécifique au code source de la classe.‘

les mixins méta

Pour ce qui est des mixins méta, le principe est presque le même, à la différence que le mixin de niveau méta ne s'ajoute pas sur la classe générique mais sur une classe de méta-objet relative à celle-ci. L'introduction de la classe de méta-objet s'explique simplement par le fait que les mixins de niveau méta doivent pouvoir intervenir sur les mixins du niveau de base pour remplir pleinement leur rôle. Pour comprendre cette distinction entre mixins de base et mixins méta, regardons la figure 3.9.

Pour chaque classe Ci du référentiel, l'intégrateur se charge de créer une classe MOCi de méta-objet associée. Chaque classe Ci possède donc une référence vers sa classe de méta-objet MOCi. La création d'une instance de la classe Ci noté Obj instancie automatiquement la classe de méta-objet et ajoute le lien méta entre ces deux instances nouvellement créées. Ainsi chaque instance du niveau de base est relié par un lien méta avec un méta-objet qui lui est propre. Comme nous l'avons précisé, l'ajout de mixin méta ce fait au niveau de la classe de méta-objet. Parallèlement au mixin de niveau de base, l'ensemble du code spécifique relatif au mixin méta s'applique donc sur les instances des classes de méta-objets. On sait par ailleurs que chaque méta-objet associé à un objet du niveau de base, contrôle l'exécution de celui ci. Ainsi, le fait d'ajouter les mixins à la classe de méta-objets offre le contrôle de l'exécution à l'ensemble des mixins, soit donc à l'aspect généralisé.

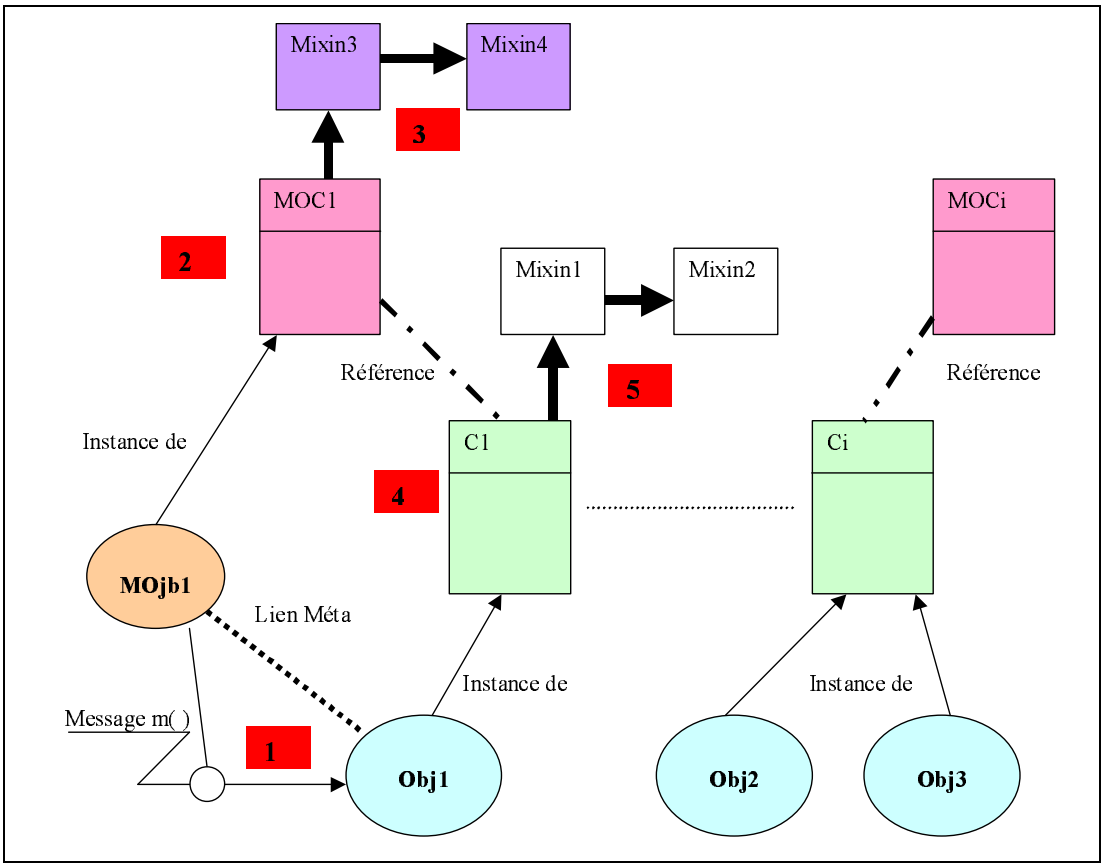

FIG. 3.9 – Simulation d'exécution regroupant les mixins méta et les mixins de base

Prenons l'exemple du message m() envoyé à Obj1 une instance de C1, le message m() est capturé par le méta-objet MObj1 (étape 1 et 2). On regarde donc si le traitement de m() ne déclenche pas de traitement particulier. Pour cela on parcourt l'ensemble des méthodes définies dans les mixins (étape 3), susceptible d'apporter un traitement supplémentaire (traitements relatif à l'aspect technique). Une fois les traitements effectués, on relâche l'exécution en transmettant le message à l'objet Obj1 qui à son tour procède comme nous l'avons vu précédemment au traitement de la méthode m() grâce aux mixins de base (étape 4 et 5).

3.4 Comparaisons avec les modèles de référence

Nous comparons maintenant notre modèle avec les modèles références.

3.4.1 Programmation par contexte : le modèle CROME

Le modèle et sa structure

Tout comme CROME, notre modèle est un modèle centralisé orthogonal, puisque d'une part, il possède un référentiel centralisant le partage de données et d'autre part, l'intégrateur est soumis à une contrainte d'orthogonalité. La granularité au niveau entitaire est elle aussi similaire et comparable. CROME utilise un plan de base et des plans fonctionnels. Dans notre modèle, ces deux entités ont leur équivalent nommé respectivement référentiel et aspect généralisé. Comme nous l'avons vu précédemment, les parties fonctionnelles de CROME sont représentées par les mixins de niveau de base. Bien que CROME n'utilise pas explicitement un mécanisme de fusion pour réaliser l'intégration, elle s'apparente néanmoins à ce type de mécanisme. Dans notre modèle, l'intégration utilise principalement ce processus à la différence qu'elle garde partiellement la notion d'aspect généralisé à l'exécution, ce qui lui confère la particularité d'être dynamique. Clairement en cours d'exécution, il est possible de retirer un aspect généralisé de l'application. La structure du modèle proposé, peut donc s'apparenter à celle réaliser dans CROME.

Le partage d'information

Le partage d'informations constitue un caractéristique majeure des modèles centralisés. Il existe dans ces modèles différents types de partage, le premier étant le partage d'informations appartenant au référentiel. Similairement à CROME, nous imposons la contrainte d'unicité des accesseurs en écriture et en lecture sur un objet. De ce fait, le partage d'informations relatives au plan de base, est réalisé par l'appel de ces méthodes particulières. Concrètement, lorsqu'un mixin souhaite accéder à un objet appartenant au référentiel il exécute la méthode correspondante à l'objet. La question qui se pose, est de savoir comment une super classe (mixin) peut-elle accéder à une sous classe de sa hiérarchie d'héritage ? Il s'agit en fait d'une des particularités des mixins. Comme nous l'avons vu précédemment les mixins sont régis par 3 règles fondamentales spécifiant l'ordre d'héritage. Le fait d'exécuter une méthode du mixin sur lui même (envoi de message à self) provoque l'exécution au niveau de la sous classe du mixin et non au niveau du mixin, afin de respecter l'ensemble de ces règles.

Le second type de partage concerne le partage d'informations entre objets d'un même contexte. Dans notre cas, il s'agit de partager des informations entre différents objets d'un même aspect. Le fait d'accéder à partir d'un mixin à une autre classe que celle sur lequel il est déployé, se traduit par un envoi de message du mixin vers la classe correspondante. Ce message transitera par la sous-classe du mixin avant d'être acheminer vers la classe correspondant au destinataire.

De la même manière, on comprend que l'on peut réaliser le troisième type de partage, rencontré dans CROME, à savoir la communications entre objet et entre contexte. Il s'agit de partager de l'information entre des objets différents appartenant à des aspects différents. Sur le même schéma que précédemment il est possible d'effectuer ce partage, puisqu'en aucun cas, dans l'approche précédente nous n'avons distingué si la classe destinataire était

un classe appartenant à l'aspect ou non. Il est donc possible de réaliser l'ensemble des différents partages d'informations avec notre modèle.

La gestion de la cohérence

Le dernier point de comparaison que nous aborderons avec le modèle CROME, concerne la cohérence des données partagées. Bien que tous les types de partage peuvent être effectués avec notre modèle, la gestion de la cohérence n'est pas totalement définie sur chacun d'eux. Le modèle actuel gère similairement à CROME, les deux premiers : le partage d'informations appartenant au référentiel et le partage d'informations entre objets de même aspect. Il utilise particulièrement la contrainte d'unicité qui transforme l'accès à l'information en un accès séquentiel.

Notre problème actuel concerne le partage d'informations entres objets d'aspects différents. Dans la majeure partie des autres approches, comme CEDRE, il se résout par la mise en place à la fois de canaux de communications spécifiques entre les différents points de vue et la création d'un langage de contraintes chargé d'assurer la cohérence de l'information partagée. Le problème que nous rencontrons, n'est pas un problème de gestion ou de partage , mais un problème de restriction. En effet, tout comme CROME, il n'est pas assez restrictif. Il offre la capacité d'accéder à partir d'un mixin à l'ensemble des informations de l'application, ce qui peut engendrer des évolutions de comportement non désirés. Cependant, comme nous l'avons préciser lors de l'étude de CROME, ce problème ne se rencontre que lorsque l'on considère qu'il est nécessaire d'établir des canaux de communications entre aspects et donc de casser l'hypothèse d'orthogonalité. La mise en place de restriction sur ce type de partage n'est donc pour l'instant pas nécessaire, mais il est bon de préciser que notre modèle offre cette possibilité sans garanties. La gestion des différents effets de bord qu'il pourrait occasionner reste à la charge du programmeur.

Conclusion

Cette comparaison montre l'ensemble des similitudes entre notre modèle et CROME. Cependant, si notre modèle remplit la majorité des fonctionnalités de l'approche par contexte, il possède certains avantages supplémentaires. La vérification de l'orthogonalité est certainement le plus intéressant. Contrairement à CROME, notre modèle possède un outil d'intégration vérifiant explicitement cette orthogonalité. Dans CROME, elle est imposé par le modèle et la vérification reste à la charge du concepteur. La vérification de la validité des règles d'intégration constitue un second avantage. En effet, notre modèle possède la capacité de vérifier la cohérence des informations fournies avant intégration et d'agir sur le processus d'intégration en conséquence. Par exemple si le référentiel et le dictionnaire d'intégration ne correspondent pas, l'intégration sera annulée. Enfin, le troisième avantage est certainement la dynamicité du modèle. Notre intégrateur est capable d'ajouter et de supprimer des aspects généralisés en cours d'exécution.

L'ensemble des points de comparaison est résumé dans le tableau de la figure 3.10.

CROME	**Notre Modèle**
Plan de base	Référentiel
Plans fonctionnels	Aspects Généralisés
Parties fonctionnelles	Mixins
Partage d'informations du plan de base	Dans les classes du référentiel, il y a des méthodes de lecture/écriture (accesseurs) qui peuvent être appelées dans les mixins Ces méthodes peuvent aussi être appelées par envoi de message à ***self*** dans les mixins.
Partage d'informations entre objets de même contexte	L'information transite par la sous-classe du mixin (dans le référentiel) avant d'être acheminer vers la classe correspondant au destinataire et si besoin au mixin correspondant.
Partage d'information entre objets entre Contextes	Réalisable, mais pas assez restrictif. Contraintes à la charge du programmeur. (hors contexte par rapport à l'orthogonalité)
Gestion de la cohérence des données	Contraintes d'unicité et d'orthogonalité gèrent l'ensemble de la cohérence
Avantages de notre Modèle :	Un intégrateur explicite qui vérifie la cohérence et la validité des directives d'intégration.
	La dynamicité apportée par l'intégrateur.

FIG. 3.10 – Tableau comparatif de notre modèle et de CROME

3.4.2 La programmation par aspects et les caractéristiques du modèle AspectJ

le modèle de programmation par aspect d'aspectJ est un modèle riche, qui est en constante évolution. Pour cette raison, il nous est difficile de comparer l'ensemble des spécificités d'aspectJ avec notre modèle. Néanmoins à partir de [Asp04], on peut dégager un ensemble de caractéristiques primordiale qui caractérise la programmation par aspect.

Le modèle et sa structure

Il existe un certain nombre d'entités génériques à l'ensemble des modèles de programmation par aspects. Nos nous devons d'expliciter ces entités et de donner leurs équivalents dans le modèle que nous proposons. Commençons par l'entité la plus générale : **l'aspect**. Comme on peut sans douter, l'aspect est représenté dans notre modèle par l'aspect généralisé. Il possède un ensemble de mixins et plus précisément des mixins de niveau méta qui lui permettent d'interagir avec l'application. On précise que **l'Advice** représentant le code de l'aspect correspond au mixin méta. La notion de "**pointcut**" (point de coupe) est elle aussi très importante. On rappelle qu'un poincut est constitué d'un ensemble de **points de jonction**. Dans notre modèle cette notion en tant qu'entité de l'application n'existe pas. En fait elle est implicite au méta-objet et s'applique sur deux niveaux de granularité différents. Le premier niveau est un niveau par défaut définit dans le méta-objet. Celui-ci intercepte tous les messages en lecture sur les méthodes et les variables. Il n'est donc pas nécessaire de préciser l'ensemble des méthodes et variables devant être intercepter. Néanmoins il peut s'avérer nécessaire de cibler les méthodes devant être intercepter, pour cela on dispose d'un niveau de granularité plus fin, capable de se focaliser sur des points de jonction précis. Notre modèle permet donc de définir des ensembles de points de jonction qui seront paramétrés dans **le script de configuration** de l'aspect afin de réaliser le tissage.

Pour être en effet complet avec les concepts de la programmation par aspect, chaque aspect doit pouvoir être configurer de manière spécifique en fonction des besoins de l'application. Pour cela on dispose d'un entité script de configuration, contenant l'ensemble des conditions pré et post-intégration.

L'héritage

La première notion rencontrée est une notion générale qui concerne l'héritage. Elle permet entre la réutilisation d'aspects génériques. Dans AspectJ, un aspect peut hériter d'un et un seul autre aspect à partir du mot clé **extends**. On parle dans ce modèle d'héritage simple. Notre modèle offre aussi la possibilité de faire de l'héritage, mais à la différence d'aspectJ, un aspect peut hériter de plusieurs aspects. Il cautionne donc l'héritage multiple. Pour cela l'aspect dispose d'un attribut contenant l'ensemble des super aspects dont il hérite. Cet attribut spécifique est initialisé lors de la construction de l'aspect. Comme chaque aspect n'est composé que d'un ensemble de mixins, l'héritage consiste donc pour

notre modèle, à récupérer l'ensemble des mixins contenu dans les super classes, afin de les tenir à disposition de la classe en héritant. Cette méthode de récupération permet de définir dynamiquement l'héritage d'aspect.

"Le partage d'aspect"

La seconde notion que nous avons dégagé, concerne le partage d'aspect. Il s'agit de pourvoir partager un aspect entre plusieurs objets. Comme nous l'avons expliqué dans la section 3.2.3, ce partage concerne les données relatives à l'exécution de l'aspect. Le modèle AspectJ utilise une approche descendante pour réaliser ce partage. Par défaut, il considère les données comme communes à l'ensemble des entités utilisant l'aspect. A partir de la méthode percflow("poincut"), il est possible de spécifier des données particulières pour chaque poincut relatif au percflow. Notre modèle, à la différence d'aspectJ utilise une approche ascendante. Par défaut, il spécifie les données relatives à l'aspect pour chaque entité. Pour prendre en compte le partage d'aspect, nous avons introduit la notion de partage dans le script de configuration. Pour qu'un aspect soit partagé, il faut qu'il soit déclaré comme partagé par l'utilisateur. Une fois cette déclaration effectuée, l'intégrateur au cours de la pré-intégration, construit un classe de méta-objet associée à l'aspect. Il définit ensuite une instance de cette classe qui sera référencé par les méta-objets relatifs aux objets de base nécessitant le partage d'aspect. Ainsi, ce méta-objet particulier sera intégrer à la chaîne de linéarisation, ce qui lui procurera le pouvoir d'intercepter les messages envoyés aux objets nécessitant cet "aspect partagé" (voir figure 3.11).

Les caractéristiques d'AspectJ

Les mots clé *before, after et around*, jouent un rôle prépondérant dans la composition d'aspects. Il est donc utilise de préciser leurs équivalents dans notre modèle de généralisation. Actuellement, il est possible de simuler les mots clé *before* et *after*. En effet en redéfinissant dans les mixins, les méthodes d'interception de messages, il est possible d'ajouter du code avant ou après le relâchement de l'interception. En ce qui concerne le mot clé *around*, le respect de l'exécution imposé par aspectJ, ne nous permet pas de le définir. Néanmoins, nous avançons l'idée qu'une modification du MOP de MetaclassTalk nous permettrait de redéfinir *before* et *after* et d'intégrer *around* . Pour cela il faudrait modifier la méthode **send** en y introduisant les méthodes **beforeDestinataire**, **aroundDestinataire** et **afterDestinataire**. Ces méthodes pourrait ainsi modifier le fil d'exécution de la méthode initiale. Dans chacune de ces nouvelles méthodes on aurait respectivement un appel à *super beforeDestinataire, super aroundDestinataire, et super afterDestinataire*, permettant ainsi de respecter la séquence d'exécution de l'ensemble des aspects définis dans l'application. Cette idée permet donc d'intégrer la notion de composition d'aspect dans notre modèle. La méthode *proceed()* utilisée dans *around* pour libérer l'exécution, est équivalente à l'oubli du super **aroundDestinataire** contenu dans **aroundDestinataire**. Cet oublie aura pour effet de bloquer l'exécution des *around* contenus dans les mixins suivants. Ces mots clé constituent les principes de déclaration des advices et par conséquent les bases de la

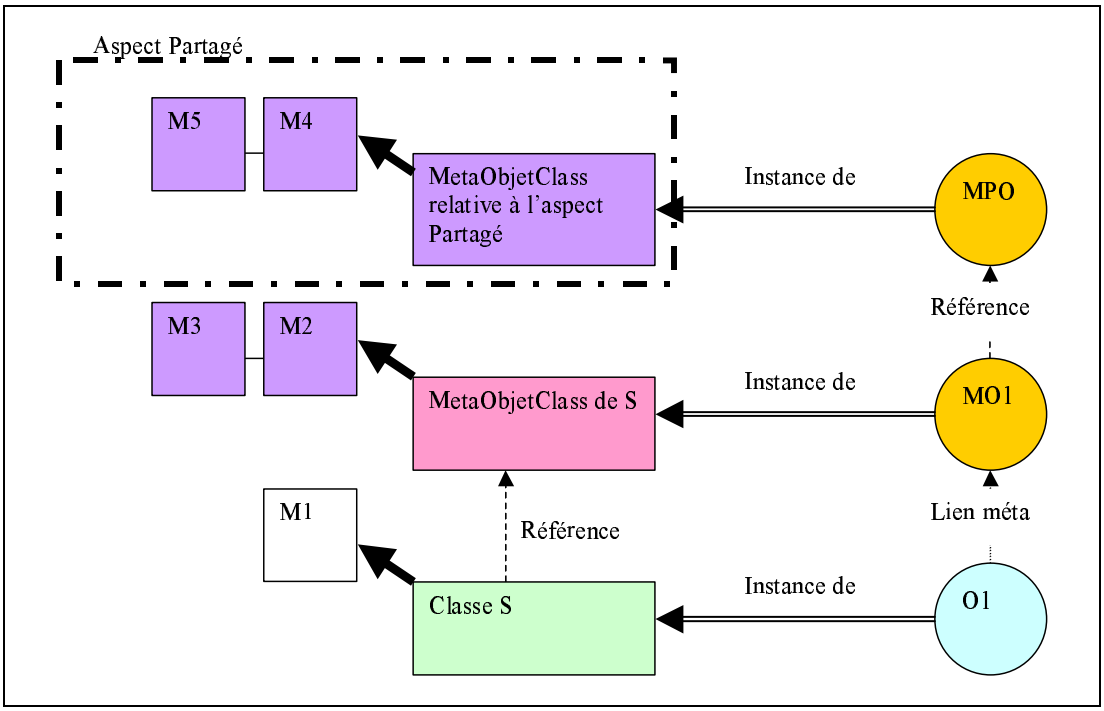

FIG. 3.11 – Schéma de partage d'aspect

programmation par aspect.

Les poincuts jouent eux aussi un rôle primordial dans la déclaration d'aspect. Pour répondre aux attentes des programmeurs, il existe en AspectJ une série de méthodes primitives applicables sur ces pointcuts. Nous donnerons dans le tableau de la figure 3.12 quelques exemples et leurs équivalents dans notre modèle.

Notre modèle actuel, ne peut pas encore être considéré comme un outil complet pour la programmation par aspect. Comme on peut le constater en regardant AspectJ [Asp04], il manque notamment l'établissement d'une grammaire spécifique servant à la définition des pointcut. Néanmoins il offre déjà de multiples possibilités de configuration et propose les fonctionnalités de bases nécessaires pour réaliser une application avec ce style de programmation.Bienqu'il ne soit pas complet, il offre en plus d'AspectJ un héritage multiple d'aspects, mais surtout la dynamicité, ce qui est un avantage considérable par rapport aux approches statiques.

3.5 Conclusion

Le modèle développé intègre donc bien l'ensemble des concepts essentiels de la programmation par contexte et de la programmation par aspect. Il offre notamment l'ensemble des fonctionnalités du modèle CROME et même plus en fournissant un outil de contrôle d'intégration. Il possède aussi l'ensemble des concepts rencontrés dans la programmation par

AspectJ Primitive Poincut	définition	Equivalent de notre modèle (basé sur MétaclassTalk)
Call (method)	Un appel à la méthode en paramètre	**send:** selector **from:** sender **to:** receiver **arguments:** args **superSend:** superFlag **originClass:** orginCl
Execution (method)	Une exécution de la méthode de la méthode en paramètre	**receive:** selector **from:** sender **to:** receiver **arguments:** args **superSend:** superFlag **originClass:** originCl
Get (field)	Un accès au champ en lecture	**atIV:** iVName
Set (field)	Un accès au champ en écriture	**atIV:** iVName **put:** value

FIG. 3.12 – Quelques exemples de Primitive Poincuts et leurs équivalents

aspect, cependant il n'offre pas encore la même richesse de fonctionnalité. Néanmoins il permet l'héritage multiple d'aspects et surtout il dispose d'une propriété essentielle qui est la **dynamicité**.

Notre modèle offre donc la possibilité de construire des aspects hybrides, à la fois fonctionnels et non fonctionnels. Cette notion a déjà été abordée dans [Bou99] en perspective des travaux réalisés sur MetaclassTalk. L'approche effectuée dans cette dernière souligne essentiellement le problème de composition d'aspect, mais ne prend pas en considération les problèmes relatifs à l'intégration des deux concepts(représentation multiple et programmation par aspect), parce qu'elle est surtout focaliser sur la programmation par aspect. Dans notre cas, nous avons rencontrés deux problèmes majeurs, le rapprochement des approches et la gestion de la cohérence à tout les niveaux (niveaux de base et niveau méta).

Le rapprochement des approches s'est résolus par l'introduction de méta-objets qui séparent dans l'aspect le niveau méta relatif à la programmation par aspect et le niveau de base de la programmation par contexte. La gestion de la cohérence entre les niveau méta et les niveau de base est principalement contrôlé par la contrainte d'unicité des champs et méthodes dans l'application. L'efficacité de cette gestion dépend essentiellement de la mise à jour dynamique des nouvelles valeurs, assuré par l'héritage classe-mixin. En ce qui concerne le problème de composition des aspects évoqué dans MetaclassTalk, nous proposons au développeur l'ordonnancement des aspects avant l'intégration ce qui permet d'assurer un processus composition raisonnable. Pour améliorer cette opération on pourrait, puisque notre processus d'intégration se déroule en plusieurs étapes, prévoir un fichier de configuration spécifique à l'application qui ordonnerait l'ensemble des aspects Sur ces différentes étapes.

Chapitre 4

L'implantation

4.1 Smalltalk et MetaclassTalk : des outils pour la programmation par aspect

4.1.1 Smalltalk : Historique, description et particularité

La programmation orientée objet consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel (que l'on appelle domaine) en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objets. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, ...).

L'approche objet est une idée qui a désormais fait ses preuves. Simula a été le premier langage de programmation à implémenter le concept de classes en 1967. Smalltalk fondé par Alan Key apparaît en 1976 comme le langage de programmation orientée objet le plus pur.Il implémente notamment les concepts principaux de l'approche objet comme l'encapsulation, l'agrégation, et l'héritage. Il sera standardisé en 1980 par le Xerox Parc (apparition de Smalltalk80).

Dans Smalltalk, toutes les entités du langage sont des objets. Les objets possèdent une mémoire locale, ainsi que la possibilité d'effectuer des opérations, de communiquer avec d'autres objets en échangeant des messages, et d'hériter d'autres objets.

Bien que Smalltalk fournissent des opérations de niveau méta comme accéder à la structure des objets en passant outre l'encapsulation (instVarAt : instVarAt : put :) ou encore envoyer des messages construit à l'exécution (perform : withArguments :), il n'offre pas la possibilité de contrôler les traitements réalisés. En effet il n'existe pas de méta-objet en SMALLTALK. Pourtant pour construire un modèle manipulant les aspects non fonctionnels, il nous fallait une plate forme réflexive, ou les méta classes sont explicites et si possible disposant d'une API ou d'un framework simple pour les extensions. C'est pourquoi nous nous sommes tournés vers MetaclassTalk.

4.1.2 MetaclassTalk : l'extension réflexive

MetaclassTalk [Bou99] est une extension réflexive de SMALLTALK, dans laquelle les classes jouent le rôle de méta-objets. Il permet non seulement de définir de nouveaux types de classes (méta classes), mais aussi d'étendre le processus d'évaluation des programmes. Ces méta classes offrent principalement le contrôle de la structure de l'application et de son exécution. elles sont capables notamment de contrôler les accès en lecture et en écriture des champs, mais aussi d'intercepter les envois et réceptions de messages. Elles possèdent aussi la faculté de choisir les méthodes à évaluer, en particulier grâce à l'existence de la notion d'héritage dans Smalltalk. Elles peuvent aussi évaluer les méthodes si c'est utile.

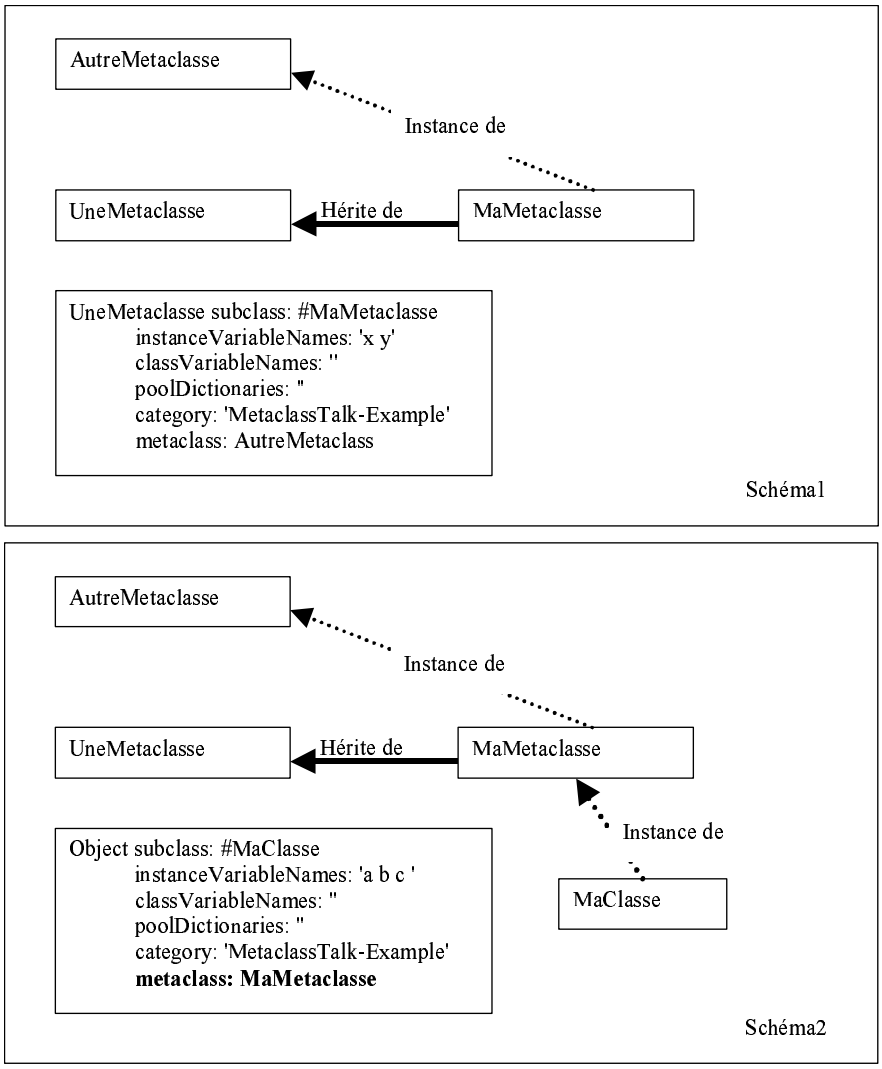

FIG. 4.1 – Une utilisation de MetaclassTalk

La figure 4.1 explique comment utiliser MetaclassTalk. On commence par définir la

méta classe relative à la classe que l'on veut crée (schéma 1), ensuite on construit la classe de niveau de base et on indique le lien d'instanciation entre la classe et la méta classe (schéma2) en remplissant le champ **metaclass :** (au bas de la déclaration de la classe).

4.1.3 Les mixins de MetaclassTalk

Dans cette sous section, nous détaillons comment implémenter des mixins avec MetaclassTalk. Nous renvoyons le lecteur à la sous section 3.2.1 pour la définition des mixins. Trois méta-classes sont implémentées pour définir le concept de mixin. La méta-classe mixin définit le mixin en temps qu'entité, elle définit un ensemble de méthodes permettant de contrôler la structure et l'exécution de la classe sur laquelle il est ajouté. La méta-classe GeneratedClass regroupe l'ensemble des classes générées par les mixins et enfin la méta-classe CompositeClass définit l'ensemble des classes qui hériteront des mixins.

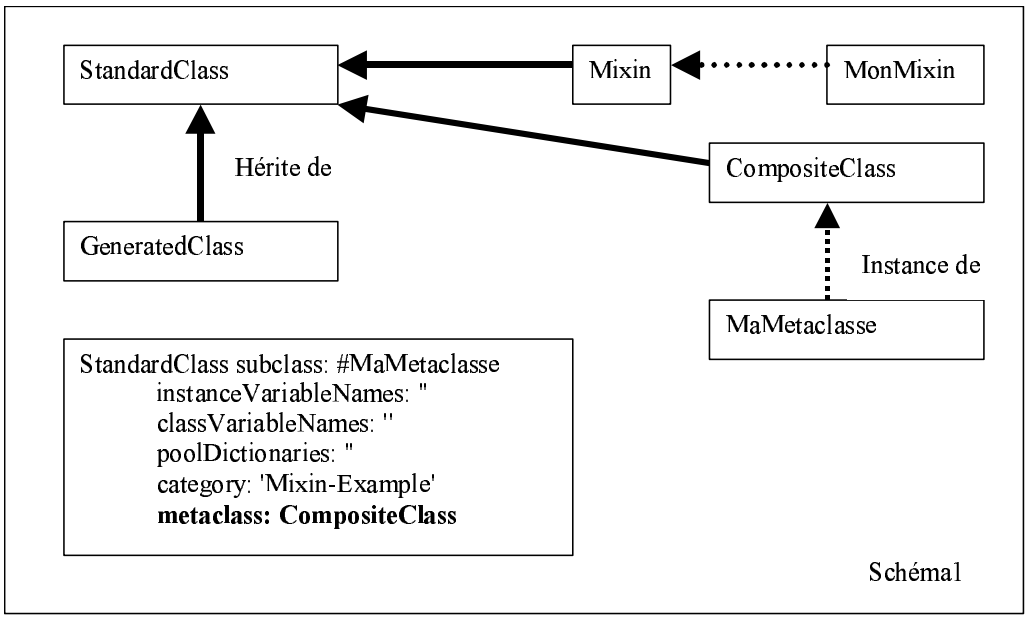

FIG. 4.2 – Procédure d'utilisation des mixins - schéma 1

La procédure de construction d'une application à base de mixins est représentée aux figures 4.2 et 4.3. On commence par générer la méta-classe relative à la classe de l'application qui nécessite l'ajout de mixins. la méta-classe de cette classe se doit d'être CompositeClass (schéma 1). Ensuite on crée la classe de base et on lui ajoute sa méta-classe propre (schéma 2). Une fois l'opération réalisée, on ajoute le mixin sur la méta-classe de la classe de base, à partir de la méthode **addMixin :** .

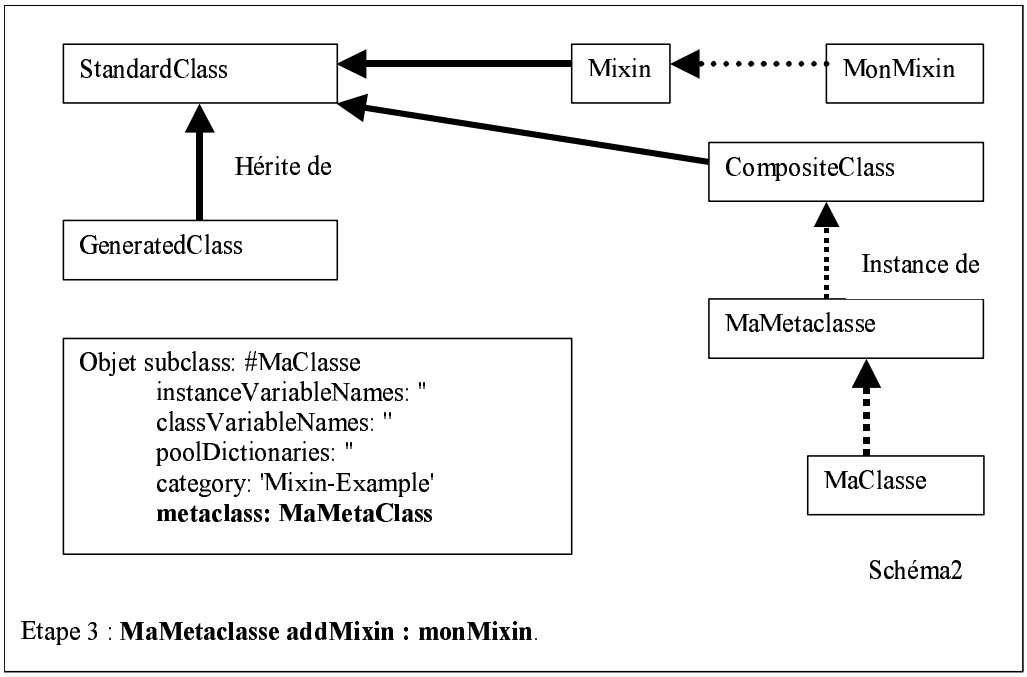

FIG. 4.3 – Procédure d'utilisation des mixins - schéma 2

4.2 L'implantation

4.2.1 Les classes implantées

L'implémentation des différents concepts a nécessité la création de classes spécifique aux entités définies. Dans cette section, nous ne redéfinirons pas ces entités, mais nous exposerons l'ensemble des classes qui ont été créées et nous décrirons les propriétés qui les caractérisent.

Nous commencerons l'énumération de ces classes, par la classe **Représentation** . Comme son nom l'indique elle caractérise l'entité représentation. Cette classe est la super classe des classes **Referentiel** et **AspectGeneralise**. Elle possède notamment des méthodes de contrôle qui servent à gérer les ensembles de classes et de mixins contenus respectivement dans le référentiel et dans les aspects généralisés.

La classe **Referentiel** représente donc l'entité référentiel.elle hérite comme nous venons de le voir de l'ensemble des méthodes de contrôle de sa super classe représentation. On note que cette classe n'effectue aucun traitement spécifique, cette sous classe est vide.

La classe **AspectGeneralise** est elle beaucoup plus riche. Elle possède notamment, comme nous avons pu le voir dans la description du modèle, deux ensembles de mixins correspondant aux mixins de niveau de base et au mixins de niveau méta. Elle dispose aussi d'une variable contenant l'ensemble des aspects dont l'instance de cette classe héritera. C'est notamment grâce à cette variable et aux traitements réalisés dessus, que l'héritage

multiple pourra être effectué.

La suivante est la classe **Script** qui comme elle l'indique symbolise le script de configuration de l'aspect. elle est composée de 3 variables spécifiques. la première, **integrationDictionary**, symbolise le dictionnaire d'intégration devant être fournis à l'intégrateur pour qu'il puisse assembler les aspects. Les deux autres, représente des blocs de code pouvant être effectuer en pré ou en post intégration, la première symbolisant le **preblock** et la seconde le **postblock**.

Enfin la dernière classe que nous décrirons et la classe **IntegrateurGeneralise**, qui caractérise notre outil d'intégration. Elle constitue le moteur de notre implantation du modèle et regroupe l'ensemble des propriétés que nous avons décris auparavant. Dans la section suivante nous aborderons sa composition dans le détail.

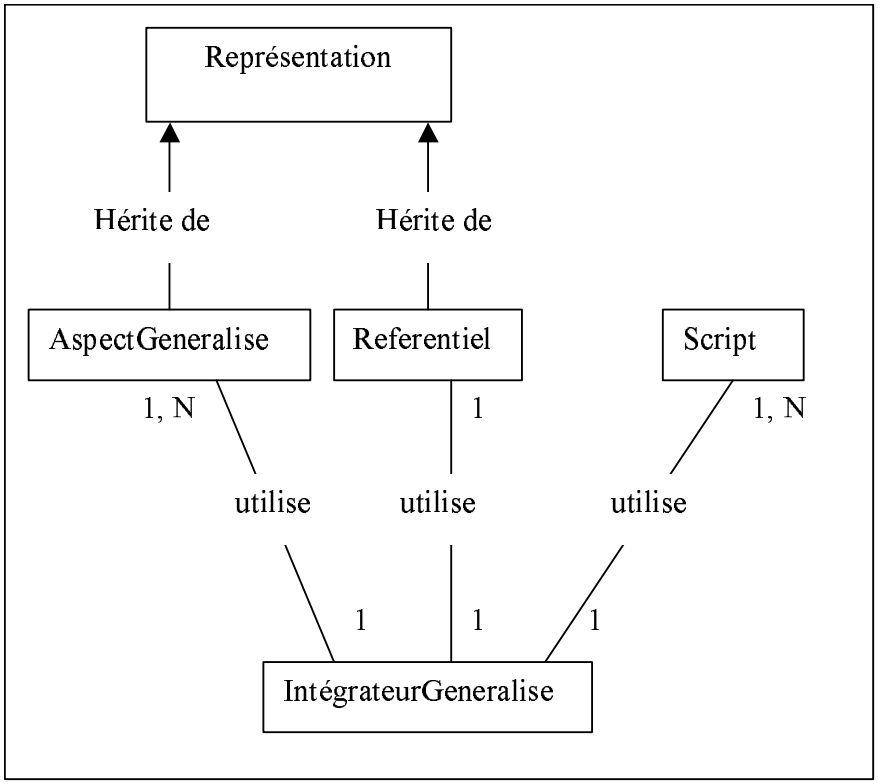

FIG. 4.4 – Diagramme des classes implémentées

le schéma 4.4 représente la hiérarchie des classes implémentées pour la réalisation de notre modèle.

4.2.2 L'intégrateur et ses outils

Comme on peut le constater sur la figure 4.4, L'intégrateur joue un rôle central dans la réalisation d'application. Comme nous l'avons vu précédemment, pour fonctionner, il

doit disposer d'un référentiel, d'une collection d'aspects généralisés et d'une collection de script. Ces entités, définies comme variable dans cette classe, constituent les hypothèses de travail de notre outil. Il possède donc une série de méthodes associées à ces variables, qui permettent d'accéder et de modifier la valeur de celles-ci.

Avant d'intégrer nous avons constater dans l'établissement du modèle, qu'il était nécessaire de vérifier un certain nombre de propriétés concernant les hypothèses fournies. Pour cela, notre intégrateur dispose d'un ensemble de méthodes de vérification. Elles sont contenues dans différentes catégories représentant chacune une propriété à vérifier. Il y a donc 3 catégories : **verify Integrity, verify Compatibility, verify Orthogonality** (figure 4.5), qui correspondent respectivement à la vérification des tests d'intégrité, de compatibilité et d'orthogonalité.

Nous avons vu que pour mettre en place les mécanismes de programmation par aspect, nous devions disposer de classes de méta-objets définies sur chaque classe du référentiel. L'intégrateur se charge d'effectuer ce travail. A partir de la méthode **createClass :**, on génère automatiquement pour chaque classe appartenant au référentiel une classe de méta-objets associée. on ajoute sur cette classe deux mixins par défaut : ClassWithInstanceMutableMetaObjects et MixinCreation. Le premier permet d'assurer la création d'une et une seule instance de cette classe par instance de la classe de base qui lui est associée, le second assure l'automatisation de l'opération d'association définie par le premier.

Une fois l'ensemble des hypothèse mises en places et les vérifications effectuées, l'intégration peut avoir lieu. Elle peut s'effectuer de deux manières différentes, la premier consiste à faire une intégration de masse en assemblant plusieurs aspects simultanément. la seconde est une intégration individuelle. Elle consiste à intégrer un seul aspect et un seul script de configuration sur l'application. Cette méthode permet entre autre d'ajouter des aspects en cours d'exécution (Dynamiquement). On remarquera que la désintégration suit aussi un processus individuelle permettant ainsi le "détissage" dynamique.

Le déroulement du processus d'intégration s'effectue donc dans cet ordre :

1. On commence par définir l'ensemble des classes du référentiel.
2. On construit ensuite les aspects généralisés ainsi que les sous ensembles qui les composent.
3. On définit les scripts de configuration associés aux aspects précédemment créés.
4. L'intégration peut alors commencer, en construisant dans un premier temps l'ensemble des classes de méta-objets relatives aux classes du référentiel.
5. On exécute ensuite l'ensemble des informations de pré-intégration contenus dans les scripts des aspects.
6. On vérifie l'intégrité, la compatibilité et l'orthogonalité des données fournies.
7. On intègre l'ensemble des aspects.
8. On exécute la seconde partie des scripts relatives aux mécanismes de post-intégration.
9. l'application est prête à être exécutée.

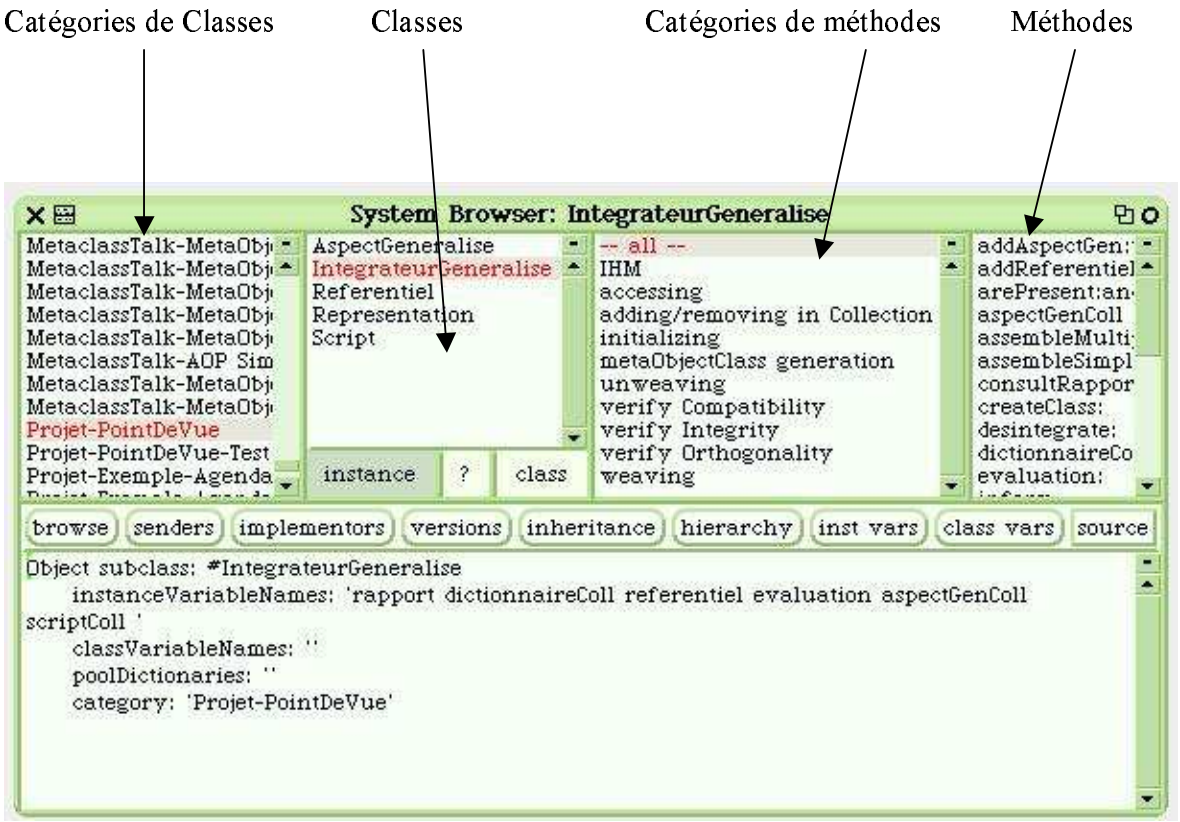

FIG. 4.5 – Les catégories de méthodes de la classe IntegrateurGeneralise

4.2.3 Application : l'exemple de l'agenda collaboratif

Afin d'expliciter les concepts que nous avons définis, nous avons réalisés une petite application, utilisant le modèle. Nous avons choisis l'exemple de l'agenda collaboratif.Dans cette section, nous commencerons par définir les objectifs de cette applications. Ensuite nous discuterons de l'étape de conception en expliquant les différents aspects dégagés. Enfin nous détaillerons un cas d'utilisation.

l'établissement de cette application doit répondre à 3 attentes : la prise de rendez vous, la planification de congés et l'organisation de réunion. Bien évidemment puisqu'il s'agit d'un agenda collaboratif, il doit bien évidemment être partager entre plusieurs utilisateurs.

La prise de rendez vous

elle consiste pour un utilisateur à bloquer une plage horaire de son agenda. les paramètres devant apparaître dans ce type d'opérations sont la date l'heure et le libellé qui constituent ensemble un événement. La prise de rendez vous se résume donc par la création d'un événement et son écriture dans l'agenda. La suppression de rendez vous associé effectue l'opération inverse qui consiste à supprimer l'événement de l'agenda et à détruire l'entité correspondante. La prise de rendez-vous s'effectue en trois temps : l'utilisateur envoie sa requête à l'agenda collaboratif, l'agenda traite les requêtes en vérifiant les disponibilités et renvoie la réponse. (voir figure 4.6)

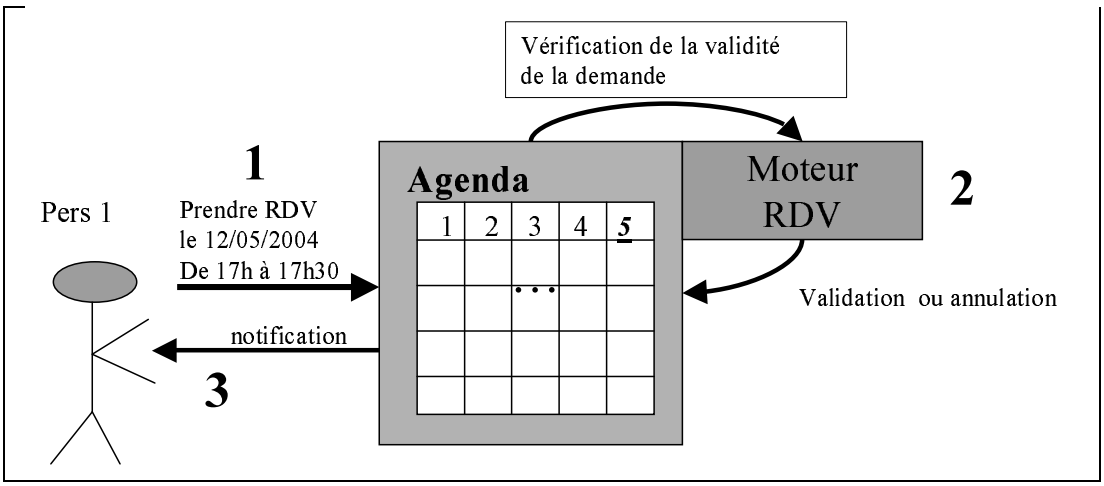

FIG. 4.6 – Cas d'utilisation de la prise de Rendez vous

La planification de congés

Parallèlement au processus de prise de rendez vous, il s'agit de créer ou de supprimer un événement en fonction d'une date de début et d'une date de fin (suivant l'opération de prise ou de suppression de congés). Le processus de planification s'effectue en plusieurs étapes.

l'utilisateur doit d'abord effectuer sa demande auprès de l'agenda. S'il y a concordance avec des plages vides, l'agenda pré-réserve les congés et transmet la requête au supérieur hiérarchique. Ce dernier valide ou refuse la requête et renvoie la réponse à l'agenda qui à son tour réserve ou annule et renvoie à l'utilisateur.

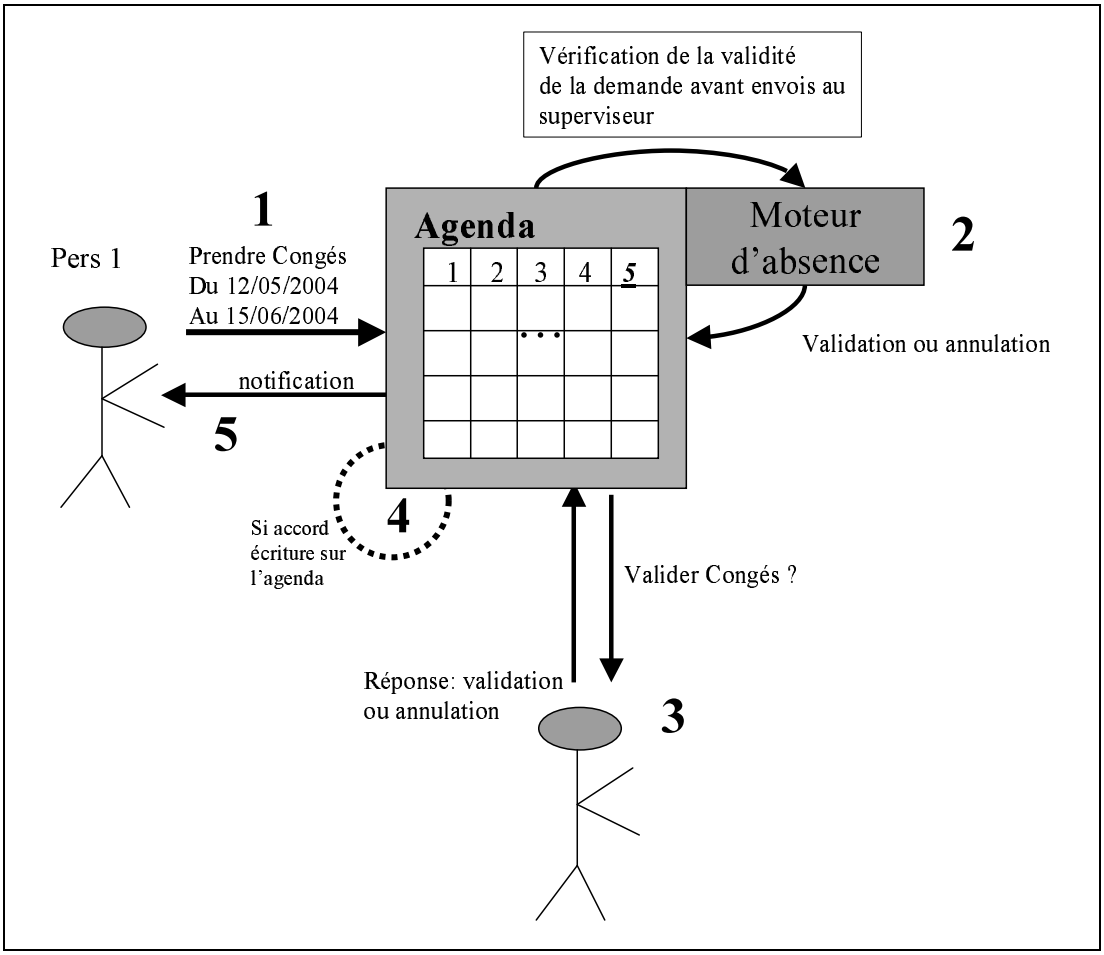

FIG. 4.7 – Cas d'utilisation de la planification de congés

Organisation de réunion

Le mécanisme d'organisation de réunion doit permettre de réunir sur une même plage horaire un ensemble de participants. Il doit aussi avoir la possibilité de proposer une date de réunion à partir d'une échéance précise. L'organisation de réunion s'effectue en différentes étapes. Un des utilisateurs demande l'organisation d'une réunion en fonction d'une liste de participants. L'agenda traite la demande en vérifiant l'ensemble des agendas des personnes concernés. Une fois l'événement définis, il notifie l'ensemble des participants et attends leur réponse. Si accord général, il valide la réunion, sinon il notifie l'échec à l'ensemble des

participants.

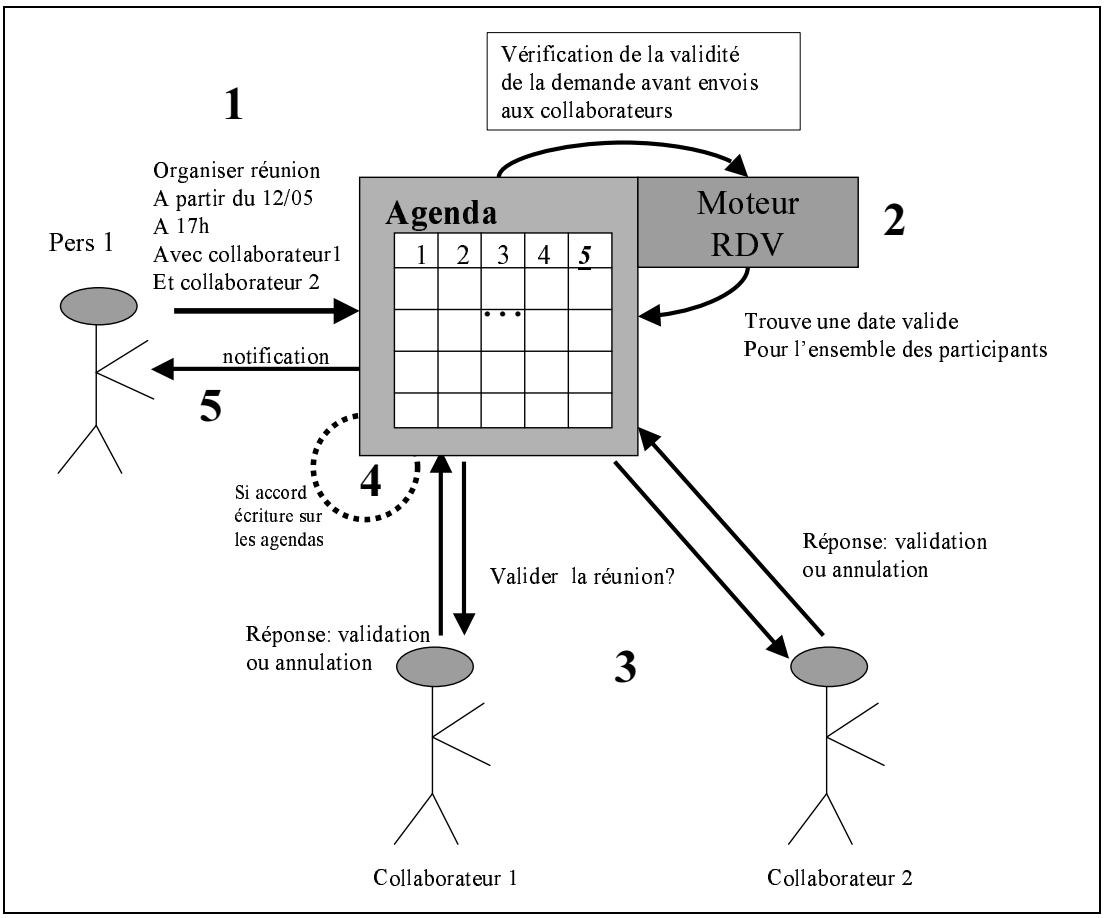

FIG. 4.8 – Cas d'utilisation de l'organisation d'une réunion

les services

Il est important que chaque utilisateur de l'agenda soit authentifié avant de pouvoir exécuter une requête. Pour cela notre modèle doit posséder une structure permettant la gestion de la sécurité. Il serait aussi intéressant de pouvoir lister l'ensemble des connexions et des différentes opérations réalisé sur l'agenda. Pour cela, nous devrions mettre en place un fichier de trace pour chaque utilisateur.

la conception des différents aspects

Nous avons donc, pour répondre à ces différentes attentes, dégagé 4 aspects distincts : gestionRDV, gestionAbsence, Trace et Sécurité. Avant de les commenter nous décrivons brièvement l'ensemble des classes appartenant au référentiel qui ont servies de base pour la conception de cette application.

Notre référentiel est composé de 4 classes : *Agenda, Personne, Superviseur, Evenement.* Un agenda est défini par un ensemble d'événements. Sa classe comporte donc un attribut *evenementSet* ainsi qu'un ensemble de méthodes permettant l'ajout et la suppression d'éléments de cet ensemble. Un événement est composé de deux dates enrichies par la notion d'heures et de minutes, et d'un intitulé. Chaque date correspond soit à une date de début ou une date de fin d'événement. La classe *Personne* décrit les utilisateurs. Une personne est définie par son nom, son prénom, son département et une référence vers son superviseur. La classe *Superviseur* est une sous classe de *Personne* et possède des attributs supplémentaires liées aux tâches supplémentaires qu'il peut effectué comme la validation de congés.

L'aspect généralisé de gestion de rendez vous (gestionRDV), regroupe l'ensemble des requêtes pouvant être effectués lors de la prise d'un rendez vous, comme l'ajout et la suppression dans l'agenda. Il dispose pour cela de deux mixins. L'un est défini sur la classe *Personne*, permettant la définition des différentes requêtes et l'autre sur la classe *Agenda* effectuant ainsi les opérations relatives à ces requêtes. Cette aspect prend aussi en considération l'organisation de réunion, en définissant de la même manières des requêtes dans le mixin de la classe *Personne* et des méthodes effectuant ces requêtes dans le mixin de la classe *Agenda*.

L'aspect généralisé de gestion d'absence (gestionAbsence) est définit pour prendre en considération l'établissement de congés dans l'Agenda. Il dispose de 3 mixins. le premier est défini sur la classe *Personne* et décrit l'ensemble des méthodes de prise de congés. le second est défini sur la classe *Superviseur*. Il permet au superviseur de valider ou de refuser une demande de congés. Et le dernier est défini sur l'agenda permettant d'effectuer l'ensemble des requêtes définies dans les classes *Personne* et *Superviseur*.

L'aspect généralisé de sécurité s'occupe de gérer l'authentification des utilisateurs pour une requête, il est composé d'un mixin validant ou refusant l'accès à l'agenda suivant la requête effectuée.

L'aspect généralisé de Trace est lui aussi composé d'un seul mixin chargé d'intercepter l'ensemble des messages envoyés à l'agenda. Il écrit chaque accès dans un fichier spécifique définis pour un utilisateur particulier.

Le schéma de la figure 4.9 retrace l'ensemble des aspects, classes et mixins abordés.

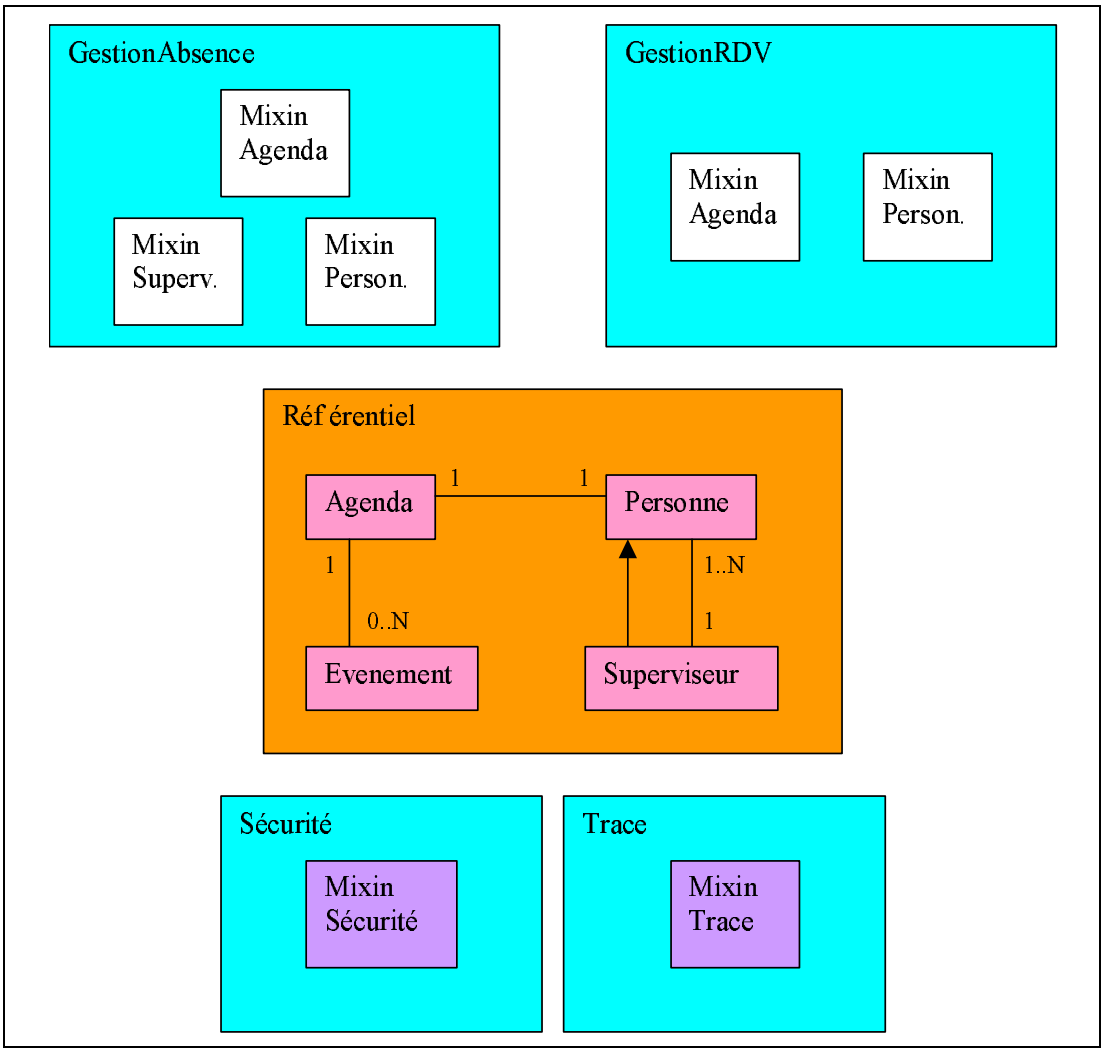

FIG. 4.9 – schéma de l'application agenda partagé

Exemple : Déroulement d'une prise de rendez vous (figure 4.10)

L'utilisateur commence donc par effectuer une requête prendreRDV(événement). Le message est aussitôt intercepté par le méta objet relatif à l'utilisateur et transmet le message au mixin relatif à l'authentification (étape 1). Si l'utilisateur peut effectuer cette requête le processus continue, sinon il s'arrête. Une fois l'authentification réalisée le message est transmis à l'agenda (étape 2). Mais avant que le message n'arrive il est intercepté par l'aspect de Trace, qui exécute le traçage de la requête dans sur la console (étape 3). Le message peut alors être traité par l'agenda. Lors de ce traitement (étape 4), l'agenda vérifie la validité de la requête et dans le cas d'une réponse positive inscrit le rendez-vous sur l'agenda (étape 5). Il notifie ensuite l'utilisateur que sa requête a été accepté (étape6).

Le message de notification envoyé de l'agenda vers l'utilisateur est de nouveau intercepté par l'aspect de Trace qui affiche la validation de la prise de rendez-vous sur la console. Dans le cas où la validation s'avère négative l'agenda notifie l'utilisateur en envoyant un message qui sera lui aussi intercepté par l'aspect de Trace, qui indiquera l'échec de la requête sur la console. La figure suivante résume ce cas d'utilisation.

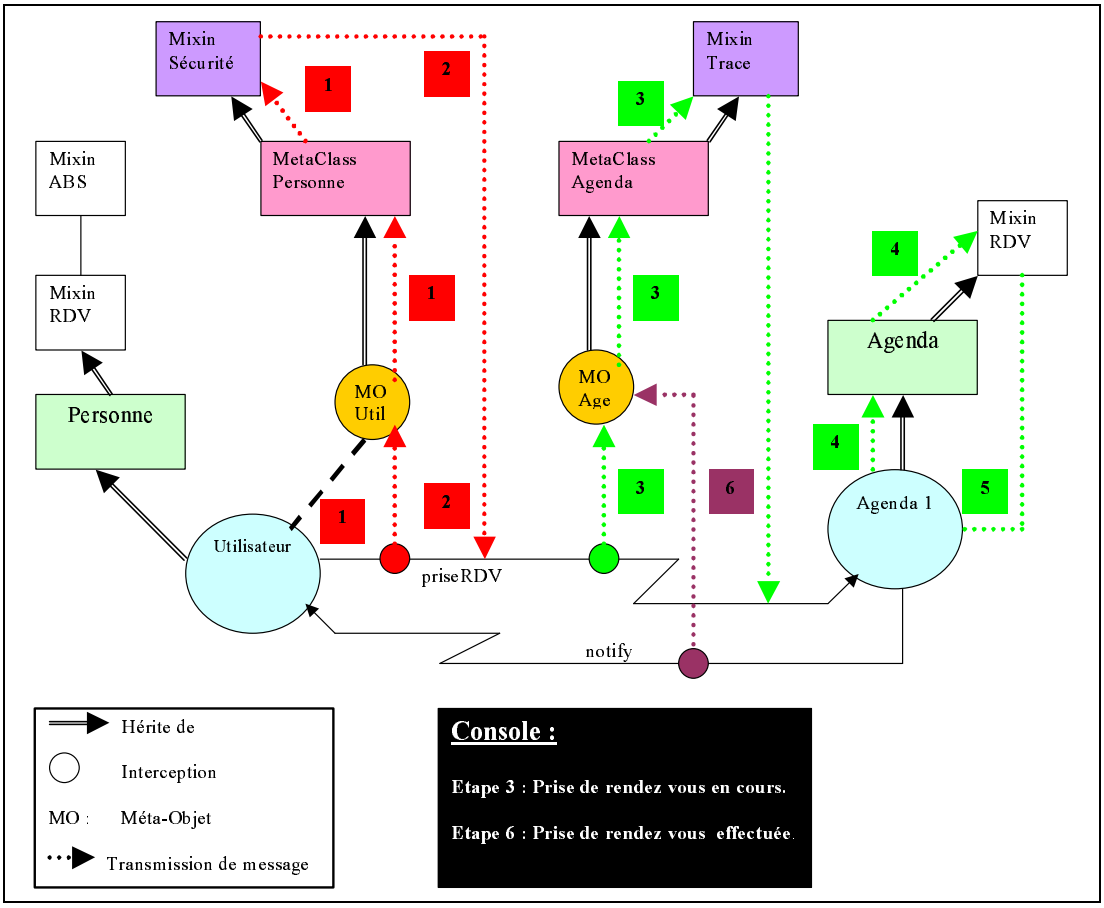

FIG. 4.10 – schéma d'application d'une prise de rendez vous

Quatrième partie

Conclusion et perspectives

Afin de palier les lacunes de la programmation par aspects en matière de réutilisation, différents paradigmes de programmation ont été élaborés. Le principal objectif de ce travail était de réifier l'ensemble des approches fonctionnelles et non fonctionnelles sous un unique concept. Il fallait prendre en considération les différents problèmes liés aux approches mais aussi les problèmes liés à l'unification.

L'étude des deux premiers chapitres, nous a montré que les modèles de représentation multiple et la programmation par aspects avaient de nombreux points communs que ce soit au niveau conceptuel, structurel ou technique. On peut notamment rappeler le but de ces deux approches : la réutilisabilité. C'est à partir de ce constat que nous avons développé au cours du Chapitre 3 notre modèle intégrant à la fois le concept de représentation multiple et celui de la programmation par aspects. Il unifie clairement sous l'entité **aspect généralisé** à la fois les points de vue rencontrés dans la représentation multiple et la notion d'aspect technique relative à la programmation par aspects. Notre modèle dispose aussi d'un outil d'intégration capable d'effectuer les différents processus de tissage ou d'intégration correspondant aux paradigmes de référence(i.e la programmation par contexte et la programmation par aspect). Comme nous l'avons définis, notre modèle est un modèle

centralisé orthogonal similaire à celui implémenté dans la programmation par contexte. Parallèlement à la programmation par aspect, il offre aussi la possibilité de séparer les aspects non-fonctionnels du code métier. Cependant il dispose d'un certain nombre d'avantages.

- Il possède un outil d'intégration, muni d'un ensemble de propriétés qui permettent de vérifier et d'assurer le bon déroulement d'une intégration
- Il autorise l'héritage multiple d'aspect contrairement à AspectJ qui ne propose que de l'héritage simple d'aspect.
- Il est dynamique, notamment grâce à son outil d'intégration
- Enfin il dispose d'un mécanisme de partage de données d'exécution d'un aspect.

Cependant, Bien que ce modèle soit complet conceptuellement, il n'offre pas la souplesse de conception des modèles décentralisés. Cet inconvénient souligné lors du choix des modèles de référence, pourrait être comblé par une évolution de notre modèle centralisé orthogonal, vers un modèle décentralisé. Pour cela nous avançons l'idée suivante : l'intégrateur devrait à partir d'un modèle centralisé orthogonal modifier le référentiel ainsi que les aspects pour les faire évoluer vers un modèle décentralisé. Il devrait donc dupliquer les données du référentiel nécessaires à chaque aspect, dans ceux-ci et entreprendre la création d'un outil de gestion des communications inter-aspects permettant ainsi l'assurance de la cohérence des données partagées. Cette solution envisageable pourrait par exemple s'inspirer des travaux sur la programmation par points de vue et plus précisément du modèle CEDRE [Naj98] pour l'établissement d'un langage de contraintes et l'assurance de la cohérence des données.

La comparaison de notre modèle face à un modèle riche comme celui d'AspectJ, nous a permis de dégager l'ensemble des fonctions offertes par notre modèle, ainsi que quelques

une des lacunes de celui-ci. Plus précisément, les mots clé de configuration d'aspects (before, after, around) devraient être explicités clairement dans notre modèle. Pour cela nous effectuons actuellement des modifications du MOP de MetaclassTalk afin que celui intègre ces notions. Ces modifications passent essentiellement par une transformation des messages *send et receive* qui devront être scindés en deux,afin d'offrir l'équivalent des outils de configuration d'AspectJ. D'autres méthodes concernant les *Primitive Poincuts*, n'ont pas d'équivalent dans notre modèle. Cependant, il est possible par composition des méthodes de base implémentées de les reproduire.

En résumé, même si certaines caractéristiques ne sont pas encore présentes dans le modèle, des pistes de solutions sont néanmoins avancées pour répondre aux différentes attentes des concepteurs. Cette étude particulière de généralisation du concept d'aspect à permis de résoudre certains des problèmes liés au différents style de programmation, mais aussi des problèmes liés à la généralisation des deux concepts. On peut notamment citer à titre d'exemple les problèmes de gestion de cohérence des modèles de représentation multiple, résolus essentiellement par l'orthogonalité et le multi-héritage. Cependant certaines questions restent en suspend. Et plus particulièrement le problème de composition d'aspects qui s'accentue avec la généralisation (même si l'utilisation d'un script de configuration général pour l'intégration résout en partie l'ordonnancement).

Enfin, la programmation par aspects est un paradigme en pleine effervescence qui laisse entrevoir de nouvelles perspectives de réutilisation. Même si de nombreuses questions restent ouvertes, nous sommes enclins à croire que ce type de programmation et plus particulièrement cette généralisation des concepts connaîtra dans les années à venir un essor semblable à celui de la programmation par objets.

Annexe A

Le prototype CEDRE développé avec SMALLTALK

Comme on peut le constater sur le schéma récapitulatif, le développement du prototype à nécessité la création de plusieurs entités représentatives du modèle. Dans le paragraphe suivant, nous décrivons brièvement le rôle de chaque entité, ainsi que les liens qui les unissent.

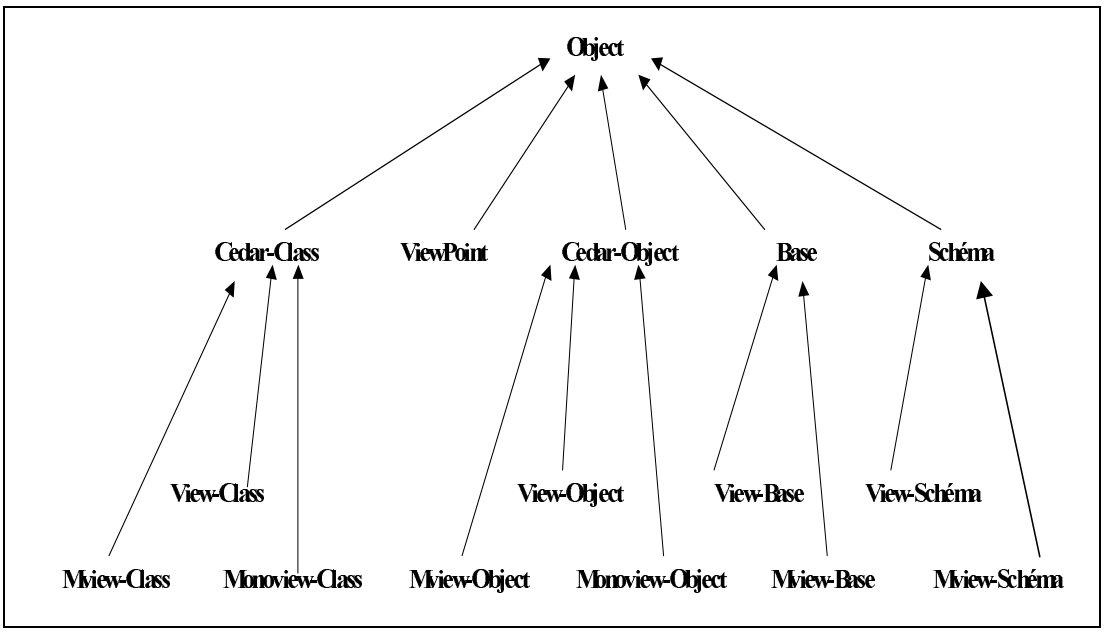

FIG. A.1 – schéma récapitulatif de l'architecture des classes du modèle CEDRE

1. ***Les classes***
   – La classe **View-class** est décrite par l'attribut mvClass qui référence la classe multivue associée à une classe-vue par une relation Rview-of.

- La classe **Mview-class** est décrite par l'attribut viewClasses qui référence l'ensemble des classes vues.
- Le lien de visibilité est implémenté par la classe **See-link** qui contient trois attributs :
  - *fromclass* qui désigne la classe source.
  - *toclass* qui désigne la classe destinataire.
  - *filter* qui décrit le filtre associé au lien de visibilité.
- La classe **Filter** représente les objets filtres et comporte deux attributs :
  - *input* qui définit les attributs constituant l'entrée du filtre.
  - *output* qui définit les attributs constituant le résultat du filtrage.

2. ***Les objets***
   - La classe **View-object** est décrite par deux attributs :
     - *mvObject* qui reférence l'objet multivue de l'objet vue.
     - *delegation* correspondant à un ensemble de couple(o,attrs), où o est un objet délégué et attrs une liste d'attributs viewObjects.
   - La classe **Mview-object** est décrite par l'attribut viewObject qui référence l'ensemble des objets vue associés à un objet multivue dans les différents points de vues.
3. ***Les bases***
   - La classe **Base** est décrite par deux attributs :
     - *itsObjects* qui est l'ensemble des objets la constituant.
     - *itsSchema* qui est le schéma associé à la base.
   - La classe **View-base** hérite de la classe Base et est décrite par l'attribut mvBase qui référence la base multivue associée.
   - La classe **Mview-base** est décrite par l'attribut viewBases qui définit l'ensemble des bases vue qui représentent une base multivue dans les différents points de vue.
4. ***Les schémas***
   - La classe **Mview schéma** est décrite par l'attribut viewSchemas qui définit l'ensemble des schémas vue qui représentent un schéma multivue dans les différents points de vue.
   - La classe **View schema** est décrite par l'attribut mvSchema qui référence la classe multivue associée.
5. ***Points de Vue***
   - Un point de vue est implémenté par la classe **ViewPoint** qui est décrite par les deux attributs schéma et base qui sont le schéma vue et la base vue définissant le point de vue.

Annexe B

Le prototype CROME développé avec SMALLTALK

Comme on peut le constater sur le schéma récapitulatif, le développement du prototype ne nécessite pas de création d'entités particulières. Les classes créées sont des sous classes de classes existantes dans SMALLTALK. Mais avant d'expliquer le travail réalisé, expliquons brièvement le cadre de développement du langage SMALLTALK.

L'environnement de développement SMALLTALK est composé d'un navigateur permettant de parcourir l'ensemble des classes et des méthodes qui les composent. Chaque classe Smalltalk appartient à une catégorie de classe, que l'on peut comparer au package Java.Dans chacune de ces classes, on trouve un ensemble de catégorie qui regroupe des méthodes par fonctionnalité. Par exemple Une classe "A", possédant un attribut "a", pourrait avoir deux méthodes geta() et seta(valeur) rangées dans une même catégorie "accessing". Cette catégorisation permet d'ordonner les méthodes, afin d'augmenter la visibilité du code. Le navigateur SMALLTALK (Browser) est donc composé de cinq fenêtres relatives aux différentes catégories, classes et méthodes.

La mise ne oeuvre de CROME à SMALLTALK a donc nécessité :

1. L'extension du concept des catégories de méthodes en introduisant les variables d'instances dans ces catégories pour obtenir des parties fonctionnelles.
2. L'extension du concept général pour relâcher la contrainte de mono-appartenance des classes aux catégories.
3. L'extension du concept de catégories de classes vers un contextualisation. Les catégories sont ainsi devenues des contextes par couplage des différentes catégories de classe.

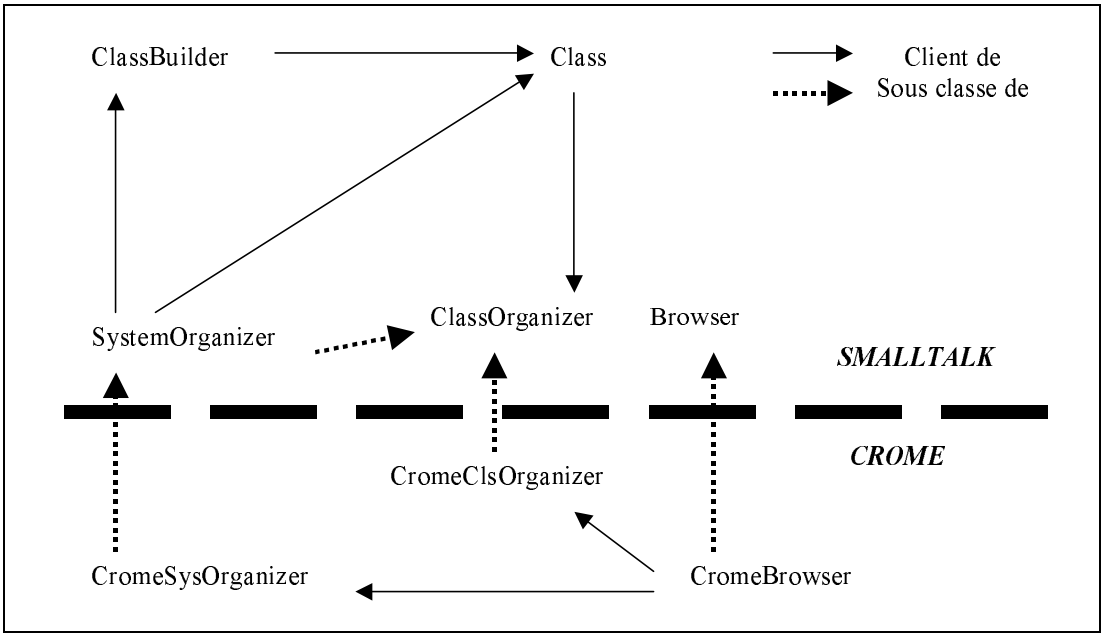

FIG. B.1 – architecture des classes du modèle CROME

Bibliographie

[AR92] Egil P. Andersen and Trygve Reenskaug. System design by composing structures of interacting objects. *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, 615 :133–152, 1992.

[Asp04] Aspectj 1.1 quick reference. *"aspectJ.org Site", http ://www.aspectJ.org*, 2004.

[BL01] N. Bouraqadi and T. Ledoux. Le point sur la programmation par aspects (aspect-oriented programming - in french). *Technique et Science Informatique*, 20(4) :505–528, 2001.

[Bou99] N. Bouraqadi. *Un MOP Smalltalk pour l'étude de la composition et de la compatibilité des métaclasses. Application à la programmation par aspects.* Thèse de doctorat, Université de Nantes, Nantes, France, 1999.

[Bou03] N. Bouraqadi. Efficient support for mixin-based inheritance using metaclasses. In *Workshop on Reflectively Extensible Programming Languages and Systems at The International Conference on Generative Programming and Component Engineering (GPCE'03)*, Erfurt, Germany, September 2003.

[GVD97] Bernard Carre Gilles Vanwormhoudt and Laurent Debrauwer. Programmation par objets et contextes fonctionnels. application de crome à smalltalk. *LMO'97*, pages 223–239, 1997.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *ACM SIGPLAN Notices*, 28(10) :411–428, October 1993.

[Kee89] Sonya E. Keene. Object-oriented programming in common lisp : A programmer's guide toclos. *Addison-Wesley,Reading,Massachsetts,USA*, 1989.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es) :154, 1996.

[Naj98] Hala Naja. La représentation multiple d'objets pour l'ingénierie. *L'objet*, Volume 4-2 :173–191, 1998.

[Naj99] Hala Naja. Cedre : un modèle pour une représentation multi-points de vue dans les bases d'objets. *Doctorat de l'Université Henri Poincaré, Nancy I*, 1999.

[O295] Technology O2. The o2 system administration guide. *Technical Report*, Version 4.5, 1995.

[Paw02] Renaud Pawlak. *"aopsys.com Site",http ://www.aopsys.com*, 2002.

[San95] Dos Santos. Un mécanisme de vues dans les systèmes de gestion de bases de données. *Thèse de Doctorat, Université Paris Sud, Orsay*, 1995.

[Smi] Brian Smith. What do you mean, meta ? *Workshop on reflection and metalevel Architectures in OO Programming, ECOOP/OOPSLA'90, Ottawa, Ontario, Canada.*

[Van99] Gilles Vanwormhoudt. Crome : un cadre de programmation par objets structurés en contextes. *PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I*, 1999.

[VN96] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. *ACM SIGPLAN Notices*, 31(10) :359–369, October 1996.

[XER00] XEROX. *"AspectJ.org Site", http ://www.aspectJ.org*, 2000.