De l'extension des langages à objets à la réalisation de modèles métiers : une évolution du développement logiciel

Philippe Lahire

Habilitation à diriger des recherches

PRÉSENTÉE DEVANT :

l'Université de Nice-Sophia Antipolis - Laboratoire I3S

PAR :

Philippe Lahire

De l'extension des langages à objets à la réalisation de modèles métiers : une évolution du développement logiciel

soutenue publiquement le 10 décembre 2004 devant le jury composé de :

Président :

Michel Cosnard
Université de Nice-Sophia Antipolis

Rapporteurs :

Jean Pierre Briot
Université de Paris VI

Pierre Cointe
École des Mines de Nantes

Markku Sakkinen
University of Jyväskylä

Directeur :

Jean-Paul Rigault
Université de Nice - Sophia Antipolis

Examinateurs :

Marianne Huchard
Université de Montpellier II

à 10 h 15 dans les locaux de l'ESSI à Sophia Antipolis

Table des matières

Remerciements

Du point de vue de la taille les remerciements ne représentent qu'une fraction du document mais ils sont pour moi quelque chose d'essentiel qui confère un peu d'âme à la technique et rappelle qu'un parcours d'enseignant-chercheur ne dépend pas que de soi.

Je désire d'abord m'adresser aux collègues qui ont accepté de faire partie de mon jury.

Je remercie *Michel Cosnard* qui malgré les nombreuses obligations liées à ses fonctions a accepté de présider ce jury.

Jean-Pierre Briot, *Pierre Cointe* et *Markku Sakkinen* ont accepté de rédiger un rapport sur ce document et plus généralement sur mon aptitude à diriger des recherches. Je suis d'autant plus sensible à l'honneur qu'ils me font que j'imagine leur emploi du temps et le nombre de sollicitations auxquelles ils doivent répondre.

Au fur et à mesure de nos discussions, des propositions de projet ou de workshop, j'ai pu constater le dynamisme, l'humour, le sérieux et les compétences de *Marianne Huchard*. Le fait qu'elle accepte de participer à mon jury me fait très plaisir.

Je suis très reconnaissant à *Jean-Paul Rigault* pour avoir accepté de superviser mon HDR et m'avoir à la fois guidé et encouragé tout au long de ce long processus. Il y a près de vingt ans, il a influencé mon parcours d'enseignant qui à l'heure actuelle est plutôt orienté « système » ; c'est en effet lui qui m'a fait aimé le système UNIX, quand j'étais un étudiant du DESS ISI (*Informatique et Sciences de l'Ingénieur*).

Un certain nombre de personnes ont particulièrement comptées. J'aimerais citer en premier *Roger Rousseau* en rappelant tout d'abord qu'en 1991, alors qu'il n'avait aucune responsabilité dans le suivi de ma thèse, m'a fait l'amitié de passer plus d'une semaine à relire et annoter mon mémoire. Ses qualités scientifiques ne sont plus à prouver mais ce petit exemple montre aussi son perfectionnisme et ses qualités humaines. Je le remercie pour cela et pour les années que j'ai passé à ses cotés dans le projet *Objets et Composants Logiciels*. Il y a bien sur les autres membres de ce projet : *Robert Chignoli*, *Philippe Collet* et *Pierre Crescenzo* qui a l'image de Roger m'ont toujours prouvé qu'appartenir à une équipe c'était aussi pouvoir compter sur les autres. Je n'oublierai pas non plus *Jean-Claude Boussard* parti aujourd'hui à la retraite mais qui a fait parti des fondateurs de ce projet, ainsi que les différents étudiants que j'ai encadrés en thèse, en DEA ou en stage de fin d'étude et qui ont contribué directement ou indirectement à l'avancée de mes recherches. Je pense en particulier à mes collègues roumains *Dan Pescaru*, *Emanuel Tundrea* et *Ciprian-Bogdan Chirila* qui font ou ont fait preuve d'une grande détermination pour atteindre les objectifs fixés.

De même j'aimerais remercier *Bertrand Meyer* pour m'avoir communiqué son goût pour la recherche et pour la confiance qu'il m'a témoigné tout au long de l'année que j'ai passée auprès de lui en Californie à travailler sur le langage Eiffel. Sans aucun doute mes activités restent influencées par ce langage et son approche du génie logiciel. Je suis heureux aujour-

d'hui d'avoir gardé sa confiance, son amitié et de pouvoir continuer à collaborer avec lui.

Ce document a gagné en qualité par les relectures attentives de *Carine Fédèle* et *Pierre Crescenzo*[1], tous les deux Maîtres de Conférences dans le laboratoire ; je suis très sensible au fait qu'ils aient pris du temps sur leurs loisirs pour m'aider. J'espère pouvoir en faire autant pour eux dans un proche avenir.

Je souhaite aussi remercier *Didier Parigot* qui m'a accueilli pendant ma délégation à l'INRIA et avec qui j'ai eu de nombreuses discussions intéressantes sur des sujets variés.

J'aimerais remercier tous mes collègues du département d'informatique de l'IUT pour les années passées à leur coté et leur soutien. Je pense en particulier à *Louis Andréani, Annie Cavarero*, *Jean-Louis Cavarero, Cécile Chagnon, Robert Chignoli*, *Caroline Fabre*, *Nhanh Le Thanh, Marie-Agnès Péraldi* ainsi qu'à tous les autres membres de l'IUT qui m'ont témoigné leur sympathie et en particulier *Michel Gautero, Florence Nicolau*, *Michel Syska*. J'espère pouvoir bientôt les aider à nouveau dans les tâches pédagogiques et administratives du département ou de l'IUT et ainsi les remercier de leur confiance.

J'ai d'abord partagé un bureau avec *Martine Collard* pendant ma thèse ; après notre nomination nous avons persévéré et nous partageons avec le même plaisir un bureau à l'IUT. Elle a passé avec succès une HDR l'année dernière et je lui adresse mes meilleurs voeux de réussite.

Je remercie l'ensemble de mes collègues du laboratoire I3S pour leur gentillesse au quotidien, leur soutien ou parfois l'expérience dont il m'ont fait profité. Je ne peux pas ici oublier les assistantes de projets et le personnel administratif qui ont toujours répondu présent quand j'en avais besoin. J'aimerais citer ici de nombreuses personnes du laboratoire et notamment celles que j'ai côtoyées dans les divers conseils, comités ou commissions, mais je suis sûr qu'un jour je m'apercevrai que j'ai involontairement oublié quelqu'un ou que tout aussi involontairement j'aurais cité plus de prénoms féminins... Je préfère donc ne pas me risquer à une énumération exhaustive et je suis sûr que beaucoup se reconnaîtront à travers ces quelques lignes.

J'ai ausssi profité de l'expérience de nombreux chercheurs et enseignants-chercheurs français ou étrangers grâce aux discussions et aux collaborations que j'ai eu au fil des années, je souhaite ici les remercier.

Pour terminer et sur un plan plus personnel, il faut bien admettre que si la carrière d'enseignant-chercheur n'est pas de tout repos pour celui qui la conduit, elle ne l'est pas non plus pour son entourage. Je remercie donc ma famille et plus particulièrement mes parents mais aussi *Alain*, *Clothilde* et *Sylvie* à la fois pour leur amour et pour avoir accepté trop souvent une présence en pointillée.

[1]Naturellement je reste le seul responsable des imperfections que ce document peut encore contenir.

Avant-propos

Ce mémoire a pour objectif de donner un aperçu précis et synthétique des activités de recherches que j'ai développées depuis le début de ma carrière d'enseignant-chercheur. Le travail qui va vous être présenté n'est pas le travail d'une seule personne mais le résultat de la collaboration avec plusieurs personnes que j'ai encadrées ou co-encadrées. Ce petit bout de vie dans la recherche doit bien sûr aussi beaucoup aux autres chercheurs ; je pense bien sûr à ceux du projet OCL et en particulier à Roger Rousseau mais aussi à tous ceux que j'ai rencontrés et côtoyés dans le laboratoire ou dans les divers congrès. Vous l'aurez donc compris toutes les idées présentées ci-après ne sont pas issues d'une seule personne, mais je revendique la démarche qui les a guidée.

Je me suis intéressé dès mon stage de DEA aux problèmes liés à la conception de logiciels et en particulier aux aspects concernant la réutilisation, la fiabilité et l'évolution des applications. Ce document retrace mes contributions dans ce domaine.

Page vide

1. Introduction

Contribuer à l'amélioration des techniques mises en œuvre pour le développement d'applications est la principale motivation de mon activité ; elle en constitue le fil conducteur au cours de ces années. Je développerai plus avant cet aspect dans la section 1.1 mais je m'intéresse plus particulièrement à l'étude des différentes techniques qui d'une part, pourraient améliorer la robustesse, la lisibilité, la documentation, les capacités d'adaptation et d'évolution des applications et d'autre part, réduiraient le temps nécessaire à leur mise en œuvre sur des plates-formes logicielles. À ce propos je considère comme essentiel de *i)* s'intéresser à la description de la sémantique des entités *ii)* proposer des solutions qui permettent d'adapter la granularité de ces entités et surtout de leur manipulation en fonction de la phase du processus de développement logiciel dans laquelle on se situe et, *iii)* réduire le fossé entre les phases de conception et de programmation pour obtenir un processus continu de développement, par exemple par transformation de modèles. La manière d'aborder le développement d'applications a évolué au fur et à mesure de l'apparition de nouveaux besoins ou de nouvelles contraintes techniques ou économiques [229]. L'évolution des besoins des applications a suscité l'apparition de nouveaux paradigmes comme par exemple la programmation par séparation des préoccupations, la programmation par composants ou la programmation orienté modèles, pour compléter ou succéder à la programmation orientée objets [84].

Dans la section 1.2, j'évoquerai l'évolution continue des techniques informatiques et son impact sur mes activités et plus généralement sur celles du développement du logiciel. Ma contribution peut être découpée en trois thèmes fortement interconnectés sur lesquels j'ai travaillé tantôt successivement, tantôt en parallèle, au gré des réflexions et des opportunités de collaboration. Il s'agit de *i)* l'intégration de nouveaux services dans une application à travers plusieurs approches, *ii)* la capture de la sémantique opérationnelle des classes et des relations et de ses applications notamment pour améliorer la réutilisation et, *iii)* la description du savoir-faire métier et sa séparation du reste de l'application.

La section 1.3 revient sur chacun de ses thèmes et décrit l'organisation de ce document.

1.1. Fil conducteur de mes activités

L'écriture d'une application est un exercice difficile qui doit bien sûr répondre aux spécifications du problème posé mais aussi mettre tout en œuvre pour que l'implantation soit la plus rapide possible et qu'elle ait un certain nombre de qualités comme par exemple la réutilisabilité de son code, son adaptabilité à différents contextes, mais aussi sa capacité à évoluer au gré de la modification des besoins. Les travaux permettant d'atteindre cet objectif peuvent être entrepris principalement pendant la phase de conception (méthode de conception [235]), ou pendant la phase de programmation (environnement de programmation [121] et langages de programmation à objets [210, 16, 154]). En d'autres termes, on peut s'intéresser aux améliorations à apporter au niveau du programme ou au niveau du modèle mais aussi au passage progressif de la phase de description du modèle à celle de l'écriture du programme. Ma

contribution a porté successivement sur ces trois facettes de la réalisation d'une application ; essayons de la situer par rapport à l'état de l'art.

Activités centrées sur les langages

Pendant la période qui s'étend de mon DEA aux premières années qui ont suivi la fin de ma thèse, je me suis intéressé plus particulièrement aux langages à objets, à classes et en particulier au langage Eiffel. Dans ce cadre j'ai notamment étudié comment introduire un service[1] pour la persistance des objets qui ait une expressivité suffisante, qui soit efficace et surtout qui n'altère pas la réutilisation [183].
Les besoins croissants des applications pour prendre en compte la distribution des données [199], leur stockage ou encore leur évolution [42], m'ont rapidement donné la certitude que les services à ajouter allaient être de plus en plus nombreux et qu'il était donc très important d'ouvrir les langages afin de leur permettre de modifier leur comportement. Les principales approches que l'on retrouve dans la littérature reposent sur un protocole métaobjet, sur la réflexité et plus généralement sur l'identification d'un métaniveau dans les langages [145, 70, 54]. On verra ci-dessous comment cette constatation nous a conduits à la réalisation du modèle OFL dont l'objectif est la modélisation des langages à objets [93].
Pour l'instant, nous avons évoqué l'introduction de services seulement dans le cadre du paradigme objet mais progressivement d'autres paradigmes plus adaptés à l'ajout de nouveaux services sont apparus ; on peut citer par exemple la programmation par aspects ou par composants. C'est ainsi que bien plus tard d'un point de vue chronologique, nous nous sommes intéressés à la généralisation de l'intégration de services (fonctionnels ou non fonctionnels) toujours dans le cadre des langages à objets mais cette fois en utilisant des approches par séparation des préoccupations [261]. Une des principales motivations de ce travail a été d'améliorer la réutilisabilité de hiérarchies de classes en leur permettant de mieux s'adapter au contexte d'utilisation.

Par rapport au cheminement de mes activités on peut retenir quatre idées clés : *i)* augmenter l'expressivité des langages à objets, *ii)* l'ajout d'un service (non fonctionnel) de persistance, *iii)* la préservation de la réutilisabilité des classes d'une application, *iv)* la séparation des préoccupations.

Activités centrées sur une continuité entre la modélisation et la programmation

Ces activités ont débuté comme je l'ai dit plus haut avec la volonté d'ouvrir les langages ou en d'autres termes de paramétrer ou d'adapter leur comportement. Elles induisent implicitement un glissement de mon domaine de recherche vers la réflexivité et la métaprogrammation. Un des principaux objectifs est de réduire le fossé qui existe entre les méthodes de conception et les langages de programmation en utilisant une approche par métaprogrammation ou par métamodélisation, selon le côté vers lequel on porte l'effort.
D'abord basé sur le langage Eiffel, ce travail a ensuite évolué vers la modélisation des langages à objets. Dans l'état de l'art, on constate que l'effort a souvent porté sur l'ouverture des classes afin de permettre d'adapter le comportement de leurs instances [140, 70, 73]. Il nous a paru prometteur d'ajouter le paramétrage des relations entre classes comme par exemple l'héritage

[1] Un service correspond à un ensemble de fonctionnalités dédiées qui sont souvent transversales et orthogonales à la fonctionnalité principale de l'application.

ou l'agrégation, etc. C'est ainsi qu'à partir d'une étude approfondie des langages existants nous avons mis en évidence des *paramètres*, des *propriétés* et des *actions sémantiques* qui permettent ensemble de décrire d'une manière plutôt satisfaisante la sémantique opérationnelle des classes et des relations [93].
Naturellement les caractéristiques qui sont très spécifiques à un langage donné ne peuvent pas être pris en compte par ces paramètres ou propriétés. Dans les langages, ils sont le plus souvent mis en évidence soit explicitement par un qualifieur (ex : *private*, *public*...), soit implicitement comme c'est souvent le cas pour les règles de visibilité. Nous avons proposé une approche qui s'appuie notamment sur des méta-assertions pour définir ces spécificités [260, 96].

De plus, dans la continuité des activités relatives à l'intégration de services dans les langages, il nous a semblé essentiel de chercher à prendre en compte dans la modélisation d'un langage, la sémantique des services dont on voudrait pouvoir disposer. C'était l'objectif de la thèse d'Adeline Capouillez qui a travaillé sur une approche basée sur la séparation des préoccupations pour adapter la sémantique opérationnelle des classes et des relations afin d'y inclure celle des services [58].

Une fois le modèle OFL posé et en nous appuyant sur le savoir-faire que nous avons acquis en le définissant, nous avons étudié et mis en œuvre des applications du modèle OFL qui peuvent notamment s'intégrer dans des environnements de programmation. On peut citer en particulier *i)* une réflexion autour de la possibilité de rendre générique une relation d'héritage (de la même manière que l'on décrit des classes génériques) [162], *ii)* la définition d'un mécanisme d'annotation des relations qui peut s'adapter à de nombreux langages mais qui a plutôt été pensé en se basant sur le langage Eiffel [94], *iii)* la spécification d'une relation d'héritage inverse avec pour objectif principal de favoriser la réutilisation et l'évolution de hiérarchies de classes [74], *iv)* la génération de profils UML à partir de la spécification d'un langage avec OFL afin de permettre au concepteur de bénéficier à la fois des outils dédiés à UML et de la précision de la sémantique offerte par le modèle OFL [260, 97] et, *v)* la définition d'un langage de description de règles de protection entre les entités constituantes d'un langage [13, 14].

Par rapport à ces activités on peut retenir quatre idées clés : *i)* la modélisation des langages à objets, *ii)* le paramétrage des classes et des relations entre classes, *iii)* la modélisation des services par séparation des préoccupations et *iv)* des applications pour les environnements de programmation.

Activités centrées sur les modèles

Le travail de métamodélisation réalisé dans le cadre d'OFL et surtout les applications que l'on voulait pouvoir mettre en œuvre à partir des informations associées à un langage[2] défini à l'aide d'OFL, nous ont fait prendre conscience de la nécessité de bien séparer le langage des applications qui lui sont dédiées. Par ailleurs, les langages de programmation ne sont plus forcément l'axe prépondérant du développement car les besoins deviennent multipolaires, les plates-formes logicielles évoluent et surtout le savoir-faire métier (la connaissance des concepts objets pour OFL) doit être capitalisé. Les motivations pour capitaliser le savoir-faire métier sont multiples, on peut citer par exemple la réactivité face aux évolutions du marché, la diminution des coûts, l'interchangeabilité des interfaces. Ces différentes constatations nous ont

[2]Le mot *langage* doit être compris comme *langage abstrait* ou *modèle de langage*.

conduits à exploiter le savoir-faire acquis pour modéliser les langages pour spécifier des modèles métiers. En d'autres termes, cela signifie que nous nous positionnons au niveau méta pour permettre la définition de modèles métiers et nous offrons des moyens de *i)* choisir l'ensemble des paramètres et des propriétés qui définissent le caractère générique d'une entité[3], *ii)* mettre en évidence un petit nombre d'actions sémantiques qui sont indépendantes des applications[4] et *iii)* définir des contraintes entre ces entités. C'est pour répondre à cette problématique que nous avons décrit une première version du modèle SmartModels [188] et qu'une première implémentation (encore incomplète) [189] a été réalisée à l'aide de SmartTools [28]. La modélisation des modèles métiers est un des thèmes sur lesquels nous désirons travailler dans les prochaines années, en nous appuyant sur les apports des approches MDA, par composants et par séparation des préoccupations.

Par rapport à ces activités on peut retenir quatre idées clés : *i)* la modélisation de modèles métiers, *ii)* l'indépendance entre les modèles métiers et leurs applications, *iii)* la collaboration entre plusieurs applications et/ou modèles métiers et, *iv*) les développements dirigés par le domaine (DDD [101]).

1.2. Mes activités dans un contexte en forte évolution

Je propose ici de revisiter ma contribution par rapport à l'évolution du contexte et des approches pour le développement d'applications. Au début des années 1990, les langages à objets était au centre de beaucoup d'activités de recherche et les services dont on désirait disposer dans un langage étaient relativement limités (persistance, parallélisme, interface graphique, distribution des données) et, surtout, les technologies employées étaient le plus souvent propriétaires. Les méthodes d'intégration allaient de la simple bibliothèque à l'extension syntaxique en passant par la modification de l'exécutif ou correspondaient à une quelconque combinaison de ces trois méthodes. Dans ce contexte, je me suis intéressé à la mise en œuvre d'un couplage fort entre un langage à objets et un système de gestion de bases de données par modification de l'exécutif et le développement d'une bibliothèque dédiée [183]. Dans l'ensemble de ces approches, on constate que le langage est le centre de tout, qu'elles résistent mal à un passage à l'échelle c'est-à-dire à la multiplication des services et, que le coût de l'ajout d'un service est prohibitif.

L'apparition de nouveaux paradigmes

La réutilisation du code écrit est certainement un des chantiers majeurs de l'approche à objets mais la multiplication des besoins des applications accélérée par la montée en puissance d'Internet (distribution des données et des programmes) a mis en évidence ses lacunes. En particulier, la présence d'une seule unité d'encapsulation (la classe) ne permet pas une intégration efficace de services non fonctionnels (gestion de la sécurité, suivi de session, etc.), ce qui amène la plupart des adeptes de l'approche objets à chercher à étendre les langages à objets ou même à introduire de nouveaux concepts au dessus de l'objet. Ainsi plusieurs paradigmes comme la programmation par séparation des préoccupations ou la programmation par

[3] Par exemple dans OFL, les notions de classes et de relations sont des entités génériques dont la généricité est définie par l'ensemble des paramètres et des propriétés.

[4] Par exemple dans OFL, on peut considérer que l'action sémantique qui réalise le lookup (recherche de primitive dans l'arbre des classes) est indépendante des applications qui utiliseront le modèle.

composants viennent respectivement combler des lacunes en matière d'encapsulation et ou de développement par boîtes noires par opposition au développement par boîtes blanches associé aux langages à objets. Dans ce contexte, je me suis intéressé à l'apport de la séparation des préoccupations dans le but d'améliorer la réutilisation et les capacités d'adaptation de services fonctionnels ou non fonctionnels dans une application à objets [261]. Par ailleurs, nous avons cherché à introduire dans le modèle OFL dont la problématique est abordée ci-dessous un mécanisme basé sur la séparation des préoccupations, dont le but est de pouvoir ajouter une description de la sémantique opérationnelle d'un service (persistance, distribution, etc.) à celle qui caractérise un langage [58]. Cet intérêt pour l'utilisation des approches pour la séparation des préoccupations se retrouve dans les travaux que nous menons depuis un an autour de la modélisation des langages métiers et qui seront passés en revue ci-dessous [188].

La prolifération des standards

L'augmentation non seulement du nombre de standards mais aussi de leur influence sur le monde de la recherche est aussi un élément à prendre en compte dans cette évolution. On trouve principalement dans le domaine qui nous préoccupe, à savoir le développement logiciel, deux organismes fortement couplés avec le monde insdustriel : l'OMG et le W3C qui proposent des standards respectivement dans le domaine de la modélisation comme MOF [240] ou UML [237, 239, 236] et dans celui des formats de documents XML et DTD [316], ou XML-SCHEMA [312, 313]. Il faut ajouter que la réalisation de passerelles entre les différents standards de l'OMG et du W3C (comme par exemple XMI [234, 241]) maximise l'impact de ces standards. Dans un premier temps, nos préoccupations nous ont fait nous intéresser principalement à UML. Nous avions constaté en particulier que la description de la sémantique des classes et des relations proposée par UML était pauvre et ne permettait donc une prise en compte efficace, ni des différentes utilisations de l'héritage [213], ni des différentes implémentations (et donc sémantiques) des mécanismes d'héritage proposés par les langages de programmation. C'est pourquoi nous avons proposé un modèle appelé OFL[5] dont l'objectif est de permettre de décrire la sémantique opérationnelle des langages à objets à classes en se basant sur un niveau méta et un ensemble de paramètres hyper-génériques [93]. Lors de la conception de ce modèle, nous nous sommes préoccupés de son intégration dans les standards. Ainsi d'une part il peut se décrire à l'aide de MOF même si l'absence de niveau méta (au sens métaclasses) rend le travail plus complexe. D'autre part, nous avons aussi étudié des passerelles avec UML ; les instances de notre modèle, c'est-à-dire les informations statiques relatives à la description de la sémantique d'un langage peuvent donner lieu à la génération d'un profil UML qui pourra être utilisé pour le manipuler [260].

De même nous adoptons une approche similaire pour rendre visible les modèles construits à partir de SmartModels [188] (voir ci-dessous), dans XML ou UML[6].

L'approche MDA et ses dérivées

Depuis plusieurs années une autre façon de voir le développement du logiciel est en train d'émerger, soutenue par l'OMG, c'est l'approche MDA (Approche dirigée par les modèles) [229]. Elle propose de voir la conception d'une application comme une succession de transformations de modèle, que celles-ci aboutissent à un raffinement du modèle ou à une implantation

[5] *OFL* représente les initiales de *Open Flexible Language*s.

[6] Le prototype que nous développons repose les facilités offertes par SmartTools [255].

de ce dernier sur une plate-forme logicielle. Ce nouveau point de vue sur le développement d'applications conduit à s'intéresser à la notion de modèle métier et plus précisément aux moyens de capturer la logique métier d'une application ou plutôt les logiques métiers. C'est dans cette problématique que se placent nos travaux sur SmartModels [188].

Évoquer plusieurs logiques métiers sous-entend que par la suite il faut pouvoir les composer ; cela nous reporte aux travaux sur la séparation des préoccupations ou sur la métaprogrammation réalisés dans l'univers des langages de programmation. Le fait de vouloir composer différentes facettes d'une application reprend une idée importante de l'évolution des langages de programmation vers le monde des composants. L'approche MDA, la séparation des préoccupations [172], la programmation par composants [295], la programmation générative [100] sont des supports pour ce que nous considérons comme une évolution importante de la manière de concevoir une application : le développement dirigé par le domaine (*Domain Driven Development*) [101].

1.3. Organisation du document

Dans les deux sections précédentes, j'ai montré deux points de vues de mes travaux de recherche : l'un par thème qui suit le plus souvent un ordre chronologique, et l'autre qui reprend certains effets marquants de l'évolution de l'état de l'art. L'organisation de ce document va plutôt s'appuyer sur le premier point de vue afin notamment, d'éviter les références en avant. Le document est découpé en trois grandes parties composées chacune de deux chapitres, et d'une conclusion.

La partie I est intitulée « intégrer de nouvelles préoccupations dans les applications objets ». Elle traite de différentes techniques pour intégrer des nouvelles fonctionnalités dans les langages de programmation. Le chapitre 2 propose une première approche qui est appliquée à la réalisation d'une extension d'Eiffel pour intégrer la persistance. Elle repose sur l'ajout de bibliothèques spécifiques et sur la modification de l'exécutif du langage. Le chapitre 3 propose une seconde approche qui s'appuie sur la séparation des préoccupations. Contrairement à la première approche qui est fortement couplée à un service donné, celle-ci est générique (indépendante des services intégrés) et elle vise plus généralement à améliorer la réutilisation dans les langages à objets (thèse de Laurent Quintian).

La partie II correspond à « la modélisation des concepts objets et ses applications ». Le chapitre 4 a pour objectif de mettre en évidence les travaux autour d'OFL (thèses de Pierre Crescenzo et d'Adeline Capouillez) et de présenter les principaux éléments de notre démarche pour capturer la sémantique opérationnelle des classes et surtout des relations. L'approche utilisée s'appuie sur la métaprogrammation et sur la définition d'un ensemble de paramètres, d'actions et d'assertions.

Le chapitre 5 montre quelques applications relatives à l'exploitation de la sémantique opérationnelle capturée par le modèle OFL. En particulier nous montrons comment exploiter les informations sur les classes et les relations pour réduire le fossé entre les phases de conception et de programmation et pour enrichir les environnements de programmation. D'autres applications principalement dédiées au mécanisme d'héritage sont en cours d'étude. Ainsi nous voulons expliquer comment à partir de la réflexion menée autour de ses diverses sémantiques et usages de l'héritage on peut définir des extensions de langage qui sont intéressantes pour

améliorer la qualité d'une application (réutilisation, évolution, documentation, etc.).

La partie III concerne des travaux qui ont débuté plus récemment et qui sont relatifs à la modélisation de modèles métiers. Le chapitre 6 a pour objectif de faire le point sur les travaux décrits dans les chapitres 3 et 4 pour les généraliser et proposer une approche originale pour le développement dirigé par les modèles. Le chapitre 7 se place dans la continuité et indique de manière détaillée les perspectives de recherche pour les années à venir. Elles reposent d'une part sur une approche pour modéliser les modèles métiers et d'autre part sur la création de modèles métiers qui pourront être par exemple, dédiés à l'extension des langages. Ce document se termine enfin par une conclusion.

Page vide

Première partie .

Intégrer de nouvelles préoccupations dans les applications objets

Page vide

Préambule

On constate que le développement d'applications nécessite l'utilisation de plus en plus de services ou de fonctionnalités indépendantes de l'application elle-même. Ils reposent sur la description de préoccupations fonctionnelles ou non fonctionnelles ou même sur la combinaison des deux. On peut citer par exemple la persistance [183], le parallélisme [211, 120], la mobilité ou la distribution [199] des objets, la sécurité [288], l'intégration *a posteriori* de patrons de conception [133], la possibilité de définir des assertions, de réaliser des traces ou bien encore calculer des métriques [193]. On est passé progressivement d'un petit nombre de services définis une fois pour toutes à un large éventail de préoccupations dont la nature et la quantité évoluent. Selon que l'on se trouve dans la première ou la seconde situation les techniques de mise en œuvre sont différentes ; en particulier, la seconde situation demande une prise en compte du changement d'échelle et surtout du caractère évolutif. Nous proposons dans cette partie de nous intéresser à ces deux situations en portant une attention particulière à la réutilisation des composants de l'application[7].

Dans le chapitre 2, nous décrivons comment un service (la gestion de la persistance des objets) peut être intégré dans un langage à objets. Il y a deux facettes dans ce travail : le contenu du service de persistance (voir section 2.2) et la méthode d'intégration dans un langage de programmation (voir section 2.3). Nous proposons une vue synthétique de chacune de ces facettes et donnons un aperçu de l'état de l'art (voir section 2.1).

Dans le chapitre 3, nous adressons un ensemble plus large de services, à savoir l'ensemble des préoccupations fonctionnelles et non fonctionnelles que l'on veut associer souvent *a posteriori* à une application. Les préoccupations non fonctionnelles sont à rapprocher de services comme la persistance, la mobilité, la sécurité, tandis que les préoccupations fonctionnelles permettent d'étendre les fonctionnalités de l'application et donc par exemple, d'adapter et d'insérer une bibliothèque de classes existante dans l'application (ex : ajout d'une interface graphique ou de patrons de conception). À travers la prise en compte de ces deux grandes familles de préoccupations, nous avons cherché à offrir une approche qui permette d'adapter des bibliothèques de classes au contexte d'une application et ainsi d'en améliorer la réutilisation.

Nous montrerons en particulier que le temps qui s'est écoulé entre ces deux travaux de recherche a particulièrement changé notre vision sur l'intégration d'un service. Elle est passée de l'intégration « en dur » d'un service dans le langage, à la mise en œuvre d'une approche ouverte basée sur un modèle métier et dont l'objectif est d'étendre un langage pour lui permettre d'adapter et d'ajouter des préoccupations définies sous forme de hiérarchie de classes. Voici quelques-unes des raisons qui nous ont amenés à modifier notre point de vue : *i)* l'évolution des besoins des applications, *ii)* l'apparition du paradigme de séparation des préoccupations et, *iii)* l'expérience acquise grâce à nos travaux sur la modélisation des concepts objets et ses applications (voir partie II) et aux premières expériences relatives aux modèles métiers et à l'approche MDA (voir chapitre 6).

[7] Le terme *composant* doit être pris ici au sens large.

Page vide

2. Une préoccupation : la gestion de la persistance

Ce travail a été effectué dans le cadre du projet *ESPRIT* Business Class *[300]*. Il constitue l'essentiel de ma thèse et de mon travail de recherche pendant les premières années qui ont suivi ma nomination comme Maître de Conférences à l'Université de Nice - Sophia Antipolis. La réflexion menée autour des problèmes liés à la persistance des objets nous a amenés à nous intéresser à la définition de points de vues (DEA de Pierre Crescenzo en 1995 [91]).

La section 2.1 fait le point sur l'état de l'art relatif à la persistance au moment de ce travail. Il est intéressant de noter que depuis, les systèmes de gestion de bases de données orientés objets (SGBD-OO) ont presque totalement disparu au profit des systèmes de bases de données objet-relationnelles (*Object-Relational DataBase Management Systems - ORDBMS*), qui sont des bases de données relationnelles incluant des concepts objets [289, 55]. Ainsi plusieurs des principales compagnies telles IBM, Informix, Microsoft, Oracle ou encore Sybase proposent désormais une version objet-relationnelle de leur produit [263]. Celle-ci supporte en particulier les concepts de *type de données*, d'*objet complexe*, d'*héritage* et un langage de requête basé sur la norme SQL3. Par ailleurs, l'apparition de la technologie XML a changé la donne en mettant au centre des préoccupations l'utilisation des standards pour l'échange de données [231]. Une première tentative avec ODL [68] adressait plus la standardisation du langage de requêtes des SGBD-OO que de la représentation des données, mais depuis, l'évolution a suivi son cours pour donner naissance aux bases de données XML. Elles sont classées en plusieurs catégories [49, 48] qui ne concernent pas toutes les très grosses bases de données qui ont les contraintes fortes en matière de performance, d'intégrité ou d'accès partagé, contraintes que nous avons prises en compte dans notre approche[1].

La section 2.2 décrit le modèle de persistance que nous proposons. Il permet entre autres, le chargement et la mise à jour des objets de manière transparente pour l'application et un mécanisme pour réaliser des requêtes qui s'intégre bien dans la philosophie des langages à objets. La section 2.3 décrit notre approche pour intégrer dans les langages à objets le service de persistance. C'est cette approche que nous opposerons à l'approche mise en évidence dans le chapitre 3.

2.1. À propos de la persistance

Les applications requièrent la mémorisation (la persistance) de données qu'elles traitent pendant leurs phases d'activation. De plus, nombreuses sont les situations qui nécessitent que ces données soient partagées et soient disponibles pour plusieurs applications qui se déroulent simultanément.

[1] Selon nous, l'approche la plus intéressante est celle des *bases de données XML natives* qui entre autres repose sur un modèle logique XML.

2. Une préoccupation : la gestion de la persistance

Pendant longtemps (jusqu'à la fin des années 1970), le développeur, pour modéliser et résoudre les problèmes du monde réel, a dû utiliser à la fois un langage de programmation et un système de gestion de bases de données ; le premier permettait la modélisation des structures et des traitements, tandis que le second offrait des fonctionnalités pour la manipulation des données persistantes. La faiblesse des interfaces permettant à des langages de programmation d'accéder aux données manipulées par les SGBD pénalisait le développeur : *i)* besoin de connaître deux modèles, *ii)* problème de compatibilité entre les modèles (langage et SGBD), *iii)* conversion des données d'un modèle à l'autre mélangée au code du programme, *iv)* problème de conception (différentes représentations pour les mêmes données).

Avec l'apparition du paradigme objet, et l'accroissement des besoins des applications, on a constaté une évolution. L'approche orientée objet permet de regrouper les structures et les méthodes qui les manipulent, d'augmenter la réutilisation et les capacités d'évolution des composants ; elle laisse ainsi entrevoir des possibilités nouvelles pour la gestion et la manipulation des objets persistants. L'idée de créer des systèmes regroupant à la fois les fonctionnalités des langages et celles des systèmes de gestion de bases de données, a fait son chemin, en suivant deux directions : *i)* introduire la possibilité de décrire des traitements dans les systèmes de gestion de bases de données : SGBD orienté objets, *ii)* introduire une gestion efficace des objets persistants dans les langages de programmation : langage de programmation intégrant la persistance.

Au moment où nous avons réalisé ces travaux, ces deux approches s'opposaient et il est difficile de mettre une frontière nette entre ces deux types de langage. Mais bien qu'il n'y ait jamais eu de consensus à ce sujet, deux différences semblaient se dégager :

- La première, qui semble la plus fondamentale, concerne leur philosophie [202] : les langages de base de données accèdent plutôt aux objets de manière associative (requête sur un ensemble d'objets), tandis que les langages de programmation persistants, privilégient l'accès navigationnel (navigation à travers un graphe d'objets) [23, 165].
- La seconde concerne l'exécution des routines ; souvent les gestionnaires d'objets qui gèrent les entités persistantes manipulées par les applications implémentées par des langages de programmation, n'offrent pas de fonctionnalités permettant l'exécution des routines en mémoire persistante (c'est-à-dire sous contrôle du gestionnaire d'objets). Cela a pour effet de provoquer un important va-et-vient d'objets entre le gestionnaire et les applications clientes en d'autres termes, entre les mémoires persistantes et volatiles.

Comme nous allons le voir dans la section 2.2, notre travail de recherche a concerné l'introduction de la persistance dans les langages de programmation. Pourtant il est important de rappeler les principaux travaux autour des SGBD à objets, ne serait-ce que pour capturer le savoir-faire nécessaire à la mise en œuvre de la persistance qui est le domaine métier associé au monde des bases de données. Pour plus de détails concernant l'état de l'art, voir [183].

Langages de bases de données à objets

Citons d'abord quelques articles qui présentaient les principaux concepts mis en œuvre par les SGBD-OO. Dans [204], les auteurs recensent les fondements des SGBD-OO, alors que dans [24], on trouve une synthèse sur les principaux points à considérer lors de la construction d'un SGBD-OO. L'évolution nécessaire des SGBD est évoquée dans [277], à travers l'apparition des nouvelles applications et de leurs besoins.

Plusieurs comparaisons relatives aux SGBD-OO existants, donnent une vue synthétique

des différentes contributions [156, 174, 31, 32]. Les principaux SGBD se classaient en plusieurs catégories :

- les produits commercialisés : *Gemstone* [203, 51], O_2 [195, 109, 110], *Vbase*[2] [12, 102, 11], *Statice* ;
- les prototypes de recherche en milieu industriel : *Orion* [175, 176, 173, 134, 33], *Iris* [128, 129] ;
- les prototypes de recherche en milieu universitaire : *Encore* [320, 281], *Postgres*, *Avance* [41], *OZ+* [306], *Extra*.

Parmi ces SGBD, plusieurs s'intéressent en particulier aux problèmes relatifs à l'évolution du schéma de la base de données. Il s'agit en particulier de *Gemstone* [259], d'*Orion* [77, 177], d'*Encore* [318, 279, 280] et d'*Avance* [42].

Langages de programmation persistants

Quand on parle de langages persistants, il faut bien distinguer les différents niveaux de persistance [61] :

- Le premier niveau est représenté par le modèle Récupération/Stockage (*Fetch-Store model*), qui ne gère aucune dépendance entre l'objet persistant stocké dans un fichier et sa copie en mémoire volatile. Ce niveau est par exemple caractérisé par la classe *STORABLE* ou *ENVIRONMENT* d'Eiffel [208] ou par la bibliothèque JDOM permettant de stocker des objets en format XML [315, 158] ; ce modèle de persistance est en général le modèle minimum offert par les langages à objets.
- Le deuxième niveau correspond à un modèle de Chargement/Mise à jour (*Load-Dump Model*) ; il gère la dépendance entre un objet persistant et son unique copie dans une application (environnement mono-utilisateur) ; la mémoire volatile agit alors comme un cache par rapport à la mémoire persistante. On peut citer par exemple une extension de Smalltalk [254].
- Le troisième niveau est en fait une extension du précédent dans un environnement où plusieurs applications peuvent accéder aux mêmes données (*Lock-Commit model*) ; ce modèle représente en fait les langages persistants au sens d'Atkinson [21, 22]. Dans la suite, on privilégiera cette dernière catégorie, mais nous mettrons aussi en évidence cette gradation de la persistance.

Certaines extensions à la persistance ont été réalisées à partir de langages compilés comme C/C++. Il s'agit essentiellement du langage *E* [265, 266] utilisé dans le projet Exodus [264, 64, 63], et de *O++* visant à interfacer le système *Ode* [4].

D'autres comme *PS-Algol* introduisent la persistance dans *S-Algol*, lui-même dérivant d'*Algol68*. PS-Algol est certainement le langage persistant le plus souvent évoqué dans la littérature ; on pourra en trouver un historique dans [21]. Il est à noter qu'un autre langage, *Napier 88*, basé sur l'expérience de PS-Algol mais intégrant notamment un type *relation*, est considéré comme l'un de ses successeurs [105]. Parmi les autres contributions, on peut citer :

- *Galileo* [8, 9, 7] qui dérive d'une des premières versions du langage ML [142], ainsi qu'*Amber* [25] qui dérive lui-même de Galileo,
- *Taxis* [222] plutôt orienté vers la gestion des données que vers la programmation,
- des extensions du langage *Ada* [160] telles qu'*Adaplex* [284] qui dérive de Daplex [276], ou *Pgraphite* [310, 297],

[2] *Ontos* succèdera ensuite à *Vbase*.

- *Trellis/Owl* [226, 275, 225] qui est particulièrement adapté pour gérer des objets complexes de grande taille et,
- *OOPS+* [273] qui inclut des possibilités de programmation logique.

Dans l'univers interprété, on trouve essentiellement des langages basés sur Lisp et sur Smalltalk. Pclos [250, 251] introduit la persistance dans le langage Clos (Common Lisp Object System) [44]. Zeitgeist [132] est implanté sur les langages Flavors et Clos. On remarquera que le traitement de la persistance dans Smalltalk est seulement de niveau 1, sauf peut-être pour la gestion des classes qui sont traitées comme des objets persistants.

2.2. Notre approche pour un service de persistance

Notre approche se situe résolument dans le cadre des langages de programmation et notre propos est d'y introduire la gestion de la persistance. Cette approche est basée d'une part sur un couplage fort entre un langage à objets et un serveur d'objets, et d'autre part sur la définition d'une bibliothèque dédiée. Le langage choisi pour valider le modèle est Eiffel [210] tandis que le serveur d'objets qui assure la gestion des objets en mémoire persistante est O_2 [108, 107]. En effet, une des idées directrices concernant la gestion de la persistance est de ne pas essayer de refaire pour le langage Eiffel ou tout autre langage de programmation un nouveau système de gestion de bases de données, mais plutôt d'utiliser un outil de base de données existant et de le coupler fortement avec le langage de programmation, afin qu'il devienne un serveur d'objets qui lui est dédié. Le prototype FLOO[3] ($Eiffel \rightarrow O_2$), est le résultat de l'application de cette idée directrice avec le langage Eiffel et qui utilise comme support de persistance le SGBD O_2.

Dans la suite, nous rappelons les propriétés fondamentales du modèle et dans la section 2.3, nous expliquons les principales caractéristiques de son intégration dans le langage de programmation.

2.2.1. Un modèle tourné vers les principes du génie logiciel

Quand le projet a débuté, au début des années 1990, le langage Eiffel et ses implémentations n'offraient, pour la persistance d'objets, qu'un moyen rudimentaire de stockage/chargement d'objets à partir de conteneurs implémentés directement sur le système de gestions de fichiers du système d'exploitation. Ce mécanisme, s'il offre un premier niveau de persistance suffisant pour des applications individuelles qui manipulent un faible nombre d'objets persistants, n'est pas adapté au développement d'applications de gestion qui nécessitent habituellement : *i)* l'accès à un grand nombre d'objets (de quelques milliers à plusieurs millions), *ii)* des facilités pour sélectionner des objets (langages de requêtes de type SQL), *iii)* des mécanismes de partage d'objets par plusieurs applications et, *iv)* des mécanismes qui garantissent l'intégrité des objets stockés.

Le modèle devait prendre en compte les contraintes suivantes : *i)* être adapté à la variété des cas d'utilisation de la persistance, c'est-à-dire être **général**, *ii)* favoriser la **réutilisabilité** des classes Eiffel qui utilisent la persistance, en permettant à n'importe quel type d'être rendu persistant, et en laissant le moins possible de traces écrites dans les classes qui manipulent

[3] *FLOO* est le nom de notre prototype. En français, *Eiffel* et *FL* sont homophones et *OO* peut aussi s'écrire O_2 comme le SGBD du même nom.

des objets persistants, *iii)* préserver la **fiabilité** (au sens correction, robustesse et sécurité)[4], *iv)* permettre la production d'applications **efficaces** et notamment, minimiser les échanges entre les mémoires volatile et persistante ou ralentir le moins possible le traitement des objets volatiles.

Le modèle a pour objectif de traiter les deux grands types d'accès aux d'objets : **l'accès navigationnel** dans un graphe d'objets, ce qui correspond à l'utilisation des langages de programmation classiques, et **l'accès sélectif** dans une collection d'objets, ce qui est utile pour la gestion de grands volumes de données, et représente donc le mode d'utilisation privilégié des langages de bases de données.

Notre approche s'appuie sur les trois règles d'or définies par M. Atkinson [23, 21, 22], auxquelles nous ajoutons deux autres règles qui permettent de préserver les qualités du langage hôte (ici Eiffel) et de promouvoir des solutions efficaces. Le lecteur trouvera dans [183] une justification détaillée de cet ensemble de règles et plus généralement, une présentation complète du modèle et de son implémentation. Nous rappelons simplement ici les thèmes abordés par les six règles :

Règle N°1 : **Indépendance de la persistance**. Le statut de persistance pour un objet est indépendant des programmes qui le manipule et *vice versa*.

Règle N° 2 : **Orthogonalité des types de données persistants**. Tout objet doit pouvoir devenir persistant, quelque soit son type et être manipulé comme tel, sans limitation.

Règle N° 3 : **Orthogonalité de la gestion de la persistance**. La manière de rendre persistant un objet est indépendante de son type, du modèle d'exécution et des structures de contrôle du langage.

Règle N° 4 : **Accès aux objets persistants**. tout objet persistant doit être accessible par son identificateur (accès navigationnel) et par son contenu (accès sélectif ou associatif).

Règle N° 5 : **Sous-traitance des traitements**. Les traitements réalisés sur un objet persistant doivent être sous-traités de manière transparente à l'outil gérant la persistance ; ils sont appelés traitement en mémoire persistante.

2.2.2. L'essentiel du modèle de persistance

L'ensemble des exemples illustrant les parties qui suivent repose sur l'utilisation d'objets d'une classe *PERSONNE* qui dispose des attributs classiques du concept à modéliser (voir figure 2.1).

Le cœur du modèle En Eiffel 2 [209], toutes les classes héritent de la classe *HERE* qui est donc le lieu naturel d'installation de nouveaux services communs. Les quatre attributs principaux dont nous voulons faire hériter toute classe *via HERE* sont donnés dans la figure 2.2.

L'attribut *pcollection* représente le moyen d'accès, pour tout objet, à la collection des objets persistants de son type. Il faut noter que contrairement à [51], nous avons décidé qu'une collection persistante d'un type contienne non seulement les instances de son type de base, mais aussi toutes celles de tous les types conformes.

[4]Il s'agit en particulier de mettre en œuvre une persistance contagieuse [25], de satisfaire les contraintes usuelles d'intégrité et de cohérence et de sérialiser les accès.

```
class PERSON
feature
  last_name : STRING ;
  first_name : STRING ;
  age : INTEGER
  sex : CHARACTER
  town : TOWN
  husband_or_wife : PERSON ;
  children : LINKED_LIST[PERSON] ;

  create(p_lname : STRING ; p_fname : STRING ; p_age : INTEGER ;
         p_sex : CHARACTER ; p_town : TOWN ; p_code : STRING) is
    do ... end ;

  display is
    do ... end ;

  set_husband_or_wife(p : PERSON) is
    do ... end ;

  add_child(p : PERSON) is
    do ... end ;

  is_named (n : STRING ; p : STRING) : BOOLEAN is
    do ... end ;
end -- class PERSON
```

FIG. 2.1.: Une classe *PERSONNE*

```
class HERE
feature
  pcollection : PCOLLECTION[like Current]
    -- Access to all persistent instances
    -- which conforms to the object type

  type_pcollection(s :STRING) : PCOLLECTION[HERE]
    -- Access to all persistent instances
    -- which conforms to the type of class s

  pw : PERSISTENT_WORLD
    -- Enable access to object databases

  session : SESSION_MANAGER
    -- Provide services for transactional management

  -- NB : All attributes of class HERE are not mentionned
end -- class HERE
```

FIG. 2.2.: Attributs de base de la classe *HERE*

```
...
p1.create("Dupond", jacques",30, 'm', v0, "UNKNOWN") ;
  -- p1 is a PERSON et v0 is an instance of TOWN
pw.put_by_key(p1,'' dupond'') ;
...
```

FIG. 2.3.: Création d'une racine explicite de persistance

L'attribut *type_pcollection* permet de plus de disposer de facon systématique d'un accès à la collection persistante de n'importe quel type (par exemple, la collection persistante des *LINKED_LIST[PERSONNE]*). Il est important de noter que le terme *accès* ne signifie aucunement « chargement » dans la mémoire du processus. Il faut considérer ces attributs comme des poignées gérées de facon paresseuse par l'exécutif. De plus, toutes les primitives déclarées dans *HERE* sont des fonctions *once* [208], ce qui n'augmente pas la taille des objets. Ainsi, *pw* est le moyen d'accéder au monde persistant (c'est-à-dire à une base d'objets contrôlée par O_2) et *session* permet de gérer le dialogue avec le serveur O_2 et en particulier la synchronisation entre le monde persistant et l'application. La classe de réference des primitives *pcollection* et *type_pcollection* est la classe génerique *PCOLLECTION*. Cette classe implémente la notion de collection persistante, c'est-à-dire à la fois le moyen d'accès aux objets du monde persistant pour le type donné mais aussi une batterie d'opérateurs à la SQL : sélection sur critère(s), comptage sélectif, etc.

Insertion d'objets dans une base Dans FLOO, la notion de monde persistant est accessible à travers *pw* disponible dans tout objet. Il correspond en O_2 à un couple base/schéma géré par le serveur d'objets. Le programmeur dispose, *via pw*, de primitives lui permettant de contrôler (connaître, changer) dans son application la base d'objets associée au monde persistant qu'il aura pu fixer au moment du lancement de l'application. Trois techniques sont disponibles pour insérer des objets dans le monde persistant :

1. La première (voir la figure 2.3), permet d'associer un objet (ou une hiérarchie d'objets) à un nom, établissant ainsi une *racine de persistance* explicite. L'objet designé par *p1* dans l'application est désormais accessible par le nom (persistant) *dupond* et appartient à la *pcollection* des *PERSONNE*.
2. La seconde (voir la figure 2.4) permet de réaliser le même travail mais de façon anonyme. Dans ce second cas, une fois l'application terminée, l'objet ne pourra être retrouvé (recherché) que par les attributs d'accès à la *pcollection* des personnes (*via* une requête).
3. La dernière technique (voir la figure 2.5) repose sur le phénomène de contagion : les enfants et l'épouse de Jacques Dupond accèdent au statut persistant par le seul fait qu'ils sont référencés par un objet persistant[5]. La racine de persistance *dupond* permettra donc désormais d'accéder à l'ensemble des membres de la famille Dupond.

Exemples d'accès navigationnels On appelle accès navigationnel un accès à un objet grâce à la notation pointée. C'est dans ce contexte qu'apparaît le mieux le phénomène de transparence, c'est-à-dire le fait que le support d'exécution charge incrémentalement et automatiquement

[5]Ce changement de statut ne signifie pas forcément un va-et-vient immédiat avec le serveur O_2 et c'est l'exécutif prend note des changements de statut et gérera le stockage effectif lors de la validation de la transaction en cours (voir section 2.2.2).

```
p2.create("Durand",jean",40, 'm', v0, "UNKNOWN");
  -- p2 is a PERSON et v0 is an instance of TOWN
pw.put(p);
```

FIG. 2.4.: Création d'un objet persistant anonyme.

```
...
p10.create("Dupond",''Jeanne", 35, 'f', v0, "UNKNOWN");
p1.set_husband_or_wife(p10);
p15.create("Dupond",''Julien", 4, 'm', v0, "UNKNOWN");
p16.create("Dupond",''Claire, 6, 'f', v0, "UNKNOWN");
p1.add_child(p15);
p1.add_child(p16);
...
```

FIG. 2.5.: Création d'objets persistants par contagion.

les objets designés par la notation pointée, au fur et à mesure du besoin. L'implémentation repose sur la mise en œuvre dans le support d'exécution d'un ensemble de mécanismes qui, lors du chargement d'un objet, permettent à la fois, d'éviter le chargement des objets référencés (récursivement) par ses attributs en associant simplement à chacun d'eux un répondeur (qui permettra, si cela devient nécessaire, de le retrouver et de le charger effectivement), et de ne disposer dans la mémoire de l'application, à tout instant, que d'une copie unique de l'objet persistant (voir la figure 2.6). On choisit ici de récupérer un objet persistant qui est une racine explicite de persistance, puis de le traverser pour afficher le nombre d'habitants de la ville où habite le conjoint de la personne décrite par cet objet. Il faut noter, d'une part, qu'il n'y a rien de particulier à ajouter dans le texte Eiffel pour faire ce travail (transparence) et d'autre part que seuls les objets utiles (*p*, *p.epoux* et *p.epoux.ville*) à cet affichage auront été chargés en mémoire (par contagion).

Mécanismes de transaction et de traitement d'erreur Le dialogue entre le gestionnaire d'objets est initialisé et terminé automatiquement par l'exécutif, le va-et-vient et la modification sont aussi à la charge de l'exécutif. En revanche, dans certains cas comme la gestion de l'intégrité et de la cohérence des objets, l'exécutif a besoin pour réagir d'informations sur le contexte de l'application.

L'adoption d'un mécanisme de gestion des transactions dans FLOO permet de garantir l'intégrité des objets en cas de panne logicielle ou matérielle et si un même objet est atteint par plusieurs applications. Selon que l'on se trouve dans l'une ou l'autre de ces situations l'utilisation du mécanisme transactionnel est plus ou moins transparent. Si l'application se déroule dans un univers où elle est seule à utiliser la base d'objets, alors elle n'a pas à se préoccuper de déclencher ou de valider une transaction ; c'est l'exécutif qui s'en charge auto-

```
...
p ?= pw.item_by_key("dupond");
if not p.void then
  io.putint(p.epoux.ville.nb_habitants);
end
...
```

FIG. 2.6.: Accès navigationnel transparent.

```
...
do
   session.start ;
   if p.age > 3 then
     p.set_age(p.age+1) ;
   end ;
   session.validate
rescue
   session.abort
end -- routine
```

FIG. 2.7.: Mise en œuvre explicite des mécanismes transactionnels.

matiquement[6]. Dans un univers où plusieurs applications se partagent les objets persistants, chaque application doit prendre le relais de l'exécutif pour déclencher, annuler ou valider une transaction (voir la figure 2.7) ; elles utiliseront respectivement les primitives de la classe *SESSION_MANAGER* : *start*, *abort* et *validate*, disponibles *via* la primitive *session* de la classe *HERE*.

On notera que dans le cas où l'annulation est le fait de l'application par le déclenchement d'une exception, c'est à l'application de demander l'annulation de la transaction (par exemple dans la clause *rescue* de la routine). Le mécanisme transactionnel a une influence sur la gestion des objets persistants ; en particulier, la synchronisation éventuelle (validation de la transaction) d'un objet persistant avec son image dans la mémoire volatile se fait à la terminaison de la transaction.

Lors de l'accès à un objet, le système doit en particulier pouvoir réagir aux situations suivantes : *i)* l'objet est occupé (utilisé) par une autre application, *ii)* l'objet est obsolète, c'est-à-dire qu'il a été marqué obsolète par une application disposant des droits suffisants sur l'objet[7], *iii)* l'application n'a pas le droit d'accès à l'objet.

Dans le premier cas, l'exécutif réagira en fonction d'un indicateur géré par l'application et accessible à partir de la classe *HERE*. Si un objet est occupé, l'application peut demander à attendre qu'il ne soit plus utilisé ou abandonner sa requête. Dans le second cas, l'exécutif déclenchera une exception signifiant à l'application qu'elle ne peut utiliser un objet déclaré obsolète. Dans le dernier cas, l'exécutif déclenchera aussi une exception.

2.2.3. Illustration des mécanismes de requêtes

Nous proposons deux exemples afin de montrer *i)* l'expressivité du mécanisme, *ii)* l'intérêt de pouvoir voir une méthode comme un objet, c'est-à-dire considérer une méthode comme un objet de première classe et, *iii)* de présenter l'intégration par bibliothèque d'une partie des fonctionnalités liées à la gestion d'objets persistants.

Une requête est décrite dans une classe qui contient à la fois la fonction booléenne de requête (R-fonction) à appliquer à chaque objet de la collection persistante[8] ainsi que des attributs éventuels correspondant aux paramètres de la R-fonction. Le code source de la figure 2.8 est un exemple de classe qui contient une fonction de requête (triviale) qui doit s'appliquer sur

[6]Une transaction est déclenchée dés qu'une routine est activée par un objet volatile sur un objet persistant, et elle se termine avec la fin de cette routine.

[7]Pour plus d'informations au sujet de la gestion des objets obsolètes, voir [183].

[8]Le programmeur peut naturellement regrouper dans une seule classe plusieurs R-fonctions.

```
class BASIC_QUERY1
  -- Query to execute on each person (R-function)
feature
  query_is_named(p : PERSON) : BOOLEAN is
    do
      Result := p.is_named(last_name,first_name);
    end -- query_is_named

  -- Parameters of the R-function
  last_name : STRING is "Dupond";
  first_name : STRING is "Claire"
end -- BASIC_QUERY1
```

FIG. 2.8.: Définition d'une requête.

```
...
local
  context : BASIC_QUERY1
do
  context.create;
  x := p.pcollection.select("query_is_named", one_argument(context));
...
```

FIG. 2.9.: Utilisation d'une requête.

chaque objet de la *pcollection* des personnes. La mise en œuvre de la requête (figure 2.9) pourra être réalisée dans une autre classe comme par exemple la classe racine de l'application.

Le premier argument de la primitive *select* est le nom de la R-fonction à appliquer à chaque objet de la collection persistante pour construire le résultat[9]. Comme ces traitements sont à la charge du serveur d'objets, celui-ci doit donc disposer à la fois de l'image O_2 de la classe représentant la requête et d'un objet persistant de cette classe pour retrouver dynamiquement le nom de la R-fonction à exécuter ; cette information est appelée le contexte. Le deuxième argument est un tableau d'objets (*ARRAY[ANY]*) qui permet la transmission du contexte (toujours) et des paramètres de la R-fonction (éventuellement)[10]. Une variante permet une transmission plus directe de paramètres à la R-fonction (code source 2.10). Les primitives de sélection acceptent ces paramètres et les transmettent (code source 2.11). Tous les autres services de requête disponibles dans FLOO (*cardinal*, *exists*, *all*, *do_ all*, *do_ if*, *count_ if*) reposent sur les mêmes principes d'utilisation.

Une requête emboîtée met en jeu plusieurs collections persistantes et permet ainsi de réaliser des traitements complexes. À titre d'exemple, nous proposons une requête (figure 2.12) qui permet de constituer la collection persistante des villes qui ont au moins *500000 habitants* et qui ont au moins *200 habitants* qui sont *âgés de plus de 90 ans*. Sa mise en œuvre pourra être décrite par le code source de la figure 2.13.

[9] La version 5 d'Eiffel permet sous certaines conditions de passer une méthode en paramètre et donc notre approche utilisant le nom de la routine pourrait être avantageusement remplacée par ce mécanisme.

[10] Les routines *xxx*_argument sont des routines utilitaires qui construisent un tableau avec *xxx* argument(s) puisque Eiffel2.3 (à la différence d'Eiffel3 et suivants) ne dispose pas de routines à liste d'arguments en nombre variable.

```
class BASIC_QUERY2
feature
  -- Query to execute on each person (R-function)

  query_is_named(p : PERSON, last_name :STRING, first_name : STRING) : BOOLEAN is
    do
      Result := p.is_named(last_name, first_name);
    end; -- query_is_named
end -- BASIC_QUERY2
```

FIG. 2.10.: Définition d'une requête paramétrée.

```
...
local
  context : BASIC_QUERY2
do
  context.create;
  x := p.pcollection.select("query_is_named", three_arguments(context, "Dupond", Claire");
...
```

FIG. 2.11.: Utilisation d'une requête paramétrée.

```
class COMPLEX_QUERY
feature
  inhabitants_min : INTEGER is 200;
  town_size : INTEGER is 500000
  age_max : INTEGER is 90

  query_town(v : TOWN) : BOOLEAN is
    do
      Result := type_pcollection("person")
                  .select("person", two_arguments(current, v))
                  .cardinal >= inhabitants_min
                 and
                v.nb_inhabitants >= town_size
    end; -- query_town

  query_person(p : PERSON, v : TOWN) : BOOLEAN is
    do
      Result := p.town.town_name.equal(v.nomville) and p.age > age_max
    end; -- query_person
end -- COMPLEX_QUERY
```

FIG. 2.12.: Définition de requêtes emboîtées.

```
...
local
   context : COMPLEX_QUERY;
   one_town : TOWN;
   towns : PCOLLECTION [TOWN]
do
  towns?= one_town.pcollection;
  context.create;
  sel := towns.select("query_town", one_argument(context));
...
```

FIG. 2.13.: Utilisation de requêtes emboîtées.

2.3. Intégration par couplage fort et définition d'une bibliothèque dédiée

Le couplage fort entre Eiffel et O_2 a été obtenu par l'intégration dans le support d'exécution standard d'Eiffel d'algorithmes de gestion du statut des objets, combiné avec une phase de compilation dont l'objectif est de produire l'image complète des classes Eiffel dans le monde O_2 et de les insérer dans ce monde, ce qui est fondamental pour la réalisation de traitements en mémoire persistante. Lors de la phase de compilation, ce couplage permet de gérer automatiquement (insérer, retrouver, modifier) dans le monde O_2 les images des classes Eiffel et de gérer de façon incrémentale les changements de statut des objets Eiffel. Pour construire une application, le programmeur dispose de la commande *floo* qui se charge d'une part de produire le binaire de l'application et le code O_2C[11] à communiquer au serveur O_2 et d'autre part, de demander au serveur O_2 l'exécution de ce code.

L'enchaîneur de passes *floo* combine les passes de la commande de compilation d'Eiffel avec les passes spécifiques de l'environnement développé. Une compilation revient à enchaîner plusieurs phases qui incluent en particulier : *i)* l'analyse du fichier de configuration d'Eiffel étendu pour déterminer les classes persistantes, *ii)* la traduction de la structure des classes persistantes en O_2, *iii)* la production des routines de conversion entre les représentations internes (EIFFEL / O_2), *iv)* la production du code O_2C des méthodes des classes persistantes, *v)* la production du script O_2 de création/modification de schéma, *vi)* le calcul des instanciations des classes persistantes afin de produire une routine O_2C pour initialiser le graphe d'héritage des collections qui gèrent l'extension des classes, et une routine C initialisant les tables qui représentent le graphe d'héritage des types (gestion du polymorphisme sur les types)[12].

2.4. Bilan

Nous avons proposé une approche qui permet d'introduire un service transparent de gestion de la persistance. Cette transparence a été acquise au prix d'un important travail qui permet d'une part de gérer l'incompatibilité entre deux mondes (Eiffel et O_2)
Le couplage fort que nous avons mis en œuvre permet notamment de garantir le stockage des objets persistants dans la base O_2 et la possibilité de réaliser l'exécution de traitements en mémoire persistante (en particulier l'exécution de requêtes). La définition de requêtes avec la possibilité de pouvoir considérer une méthode comme un objet de première classe, l'accès direct à des racines de persistances et la spécification de transaction, se fait par l'intermédiaire d'une bibliothèque dédiée. Par ailleurs les fonctionnalités essentielles comme l'accès aux instances persistantes ou l'interrogation du statut d'un objet ont été ajoutés dans la classe *HERE* dont hérite toute classe Eiffel. Enfin, la gestion transparente du va-et-vient des objets et de leur mise à jour est implantée par modification de l'exécutif du langage.

Si l'effort à faire pour gérer l'incompatibilité entre deux mondes ne peut être évité, il semble intéressant de s'appuyer sur des approches basées sur la séparation des préoccupations pour intégrer des nouveaux services et des bibliothèques spécifiques. Les travaux décrits dans la section 3 visent à répondre à cette problématique avec l'objectif supplémentaire d'augmenter

[11] C'est le langage qui permet de décrire le corps des méthodes en O_2. Plusieurs phases sont nécessaires.
[12] Ceci est en particulier nécessaire pour prendre en compte (de manière transparente), les différences de gestion de l'héritage d'Eiffel et O_2, et pour pallier à l'absence de classes génériques dans ce dernier.

les capacités d'adaptation et de réutilisation des classes d'une application.

Page vide

3. Description et réutilisation des préoccupations

Nous proposons dans ce chapitre de décrire une deuxième approche qui permette l'ajout de services [261]. Elle diffère par rapport à celle décrite dans le chapitre 2 par notamment son objectif qui n'est pas seulement l'ajout d'un service pour gérer les objets persistants[1], mais qui est plutôt de proposer une généralisation de la problématique et de permettre à un langage de séparer les préoccupations (fonctionnelles ou non) d'une application afin de pouvoir ensuite construire de nouvelles applications en composant différemment ces préoccupations. En d'autres termes on veut : *i)* pouvoir faire évoluer une application au fur et à mesure des besoins par l'ajout de nouvelles préoccupations et *ii)* permettre à une application et plus généralement à une préoccupation de s'adapter afin de pouvoir recevoir ou se composer avec d'autres préoccupations et donc, d'améliorer sa capacité à être réutilisée dans différents contextes.

La programmation par séparation des préoccupations repose sur le constat que la « classe » n'est pas toujours une unité de réutilisation adaptée. Parfois c'est parce que sa granularité est trop faible (on réutilise souvent des groupes de classes collaborantes). Parfois, au contraire, c'est parce que des fonctionnalités de nature très différentes sont réparties entre plusieurs classes et qu'une même classe peut regrouper des éléments participant à plusieurs fonctionnalités ; ceci induit un découpage transversal de l'ensemble des classes.

Nous proposons d'abord de donner un aperçu (section 3.1) des différentes techniques permettant de mettre en œuvre la séparation et la composition des préoccupations. Nous dressons ensuite un bilan des approches les plus prometteuses (section 3.2) et nous en déduisons un cahier des charges. Enfin nous proposons un modèle centré sur la réutilisation de préoccupations (fonctionnelles ou non) et sur leur intégration dans une application (section 3.3).

3.1. Supports pour la séparation des préoccupations

L'approche objet offre un certain nombre de concepts qui sont intéressants pour le support de préoccupations. Parmi ces concepts clés on peut citer par exemple l'encapsulation. Chaque objet possède un type qui est décrit par une classe dont il est instance. Il encapsule un état (ses variables) et les opérations possibles sur cet état (ses méthodes) et sélectionne ce qu'il rend visible [143, 140, 210, 292]. L'héritage est une autre propriété essentielle qui permet en particulier de rendre une classe héritière compatible à la fois sur les plans structurels et comportementaux à toutes les classes dont elle hérite directement ou indirectement. La généricité (template C++, classe générique Eiffel, etc.) représente aussi un apport intéressant.

Par ailleurs, certains langages à objets sont réflexifs ou offrent quelques fonctions réflexives c'est-à-dire qu'ils disposent d'une représentation manipulable (réification) des diverses entités

[1] On rappelle qu'un tel service est non fonctionnel c'est-à-dire qu'il n'enrichit pas la fonctionnalité de base de l'application (ce n'est bien sûr pas vrai si la fonctionnalité de base est justement de gérer la persistance).

qui forment un programme comme les classes ou les méthodes. Selon les langages, ces diverses entités peuvent être observées (introspection) et/ou modifiées (intercession) [53, 145, 140, 169] ; les facilités ainsi offertes doivent aussi être considérées.

Grâce aux concepts mentionnés ci-dessus, le paradigme objet permet *i)* de définir et de construire des objets de manière modulaire, *ii)* de décrire un ensemble d'objets à travers la description d'une classe et, *iii)* d'adapter ou de spécialiser incrémentalement la structure et le comportement des objets et de considérer différents point de vue sur le même objet. Ces propriétés favorisent la réutilisation et l'évolution du code mais la complexité grandissante des systèmes actuels, les rendent parfois insuffisantes. En effet, l'implémentation des préoccupations dans le paradigme objet est souvent transversale à la hiérarchie de classes, et a tendance à la dénaturer. Même l'utilisation (au dessus des concepts objets), de méthodologies évoluées comme la conception par framework [126], n'arrivent pas à intégrer ces préoccupations de manière efficace. En particulier le fait d'avoir le concept de classe comme unique entité d'encapsulation ne permet pas d'inclure les nombreuses préoccupations des applications de manière séparée.

Le besoin d'intégrer de nouvelles préoccupations a suggéré la proposition de nombreuses extensions de langages existants et l'approche décrite dans le chapitre 2 en est un exemple. Comme on l'a vu pour la persistance, ces extensions sont dédiées à un ensemble de préoccupations figées et donc ne peuvent résoudre qu'un sous-ensemble des besoins des applications. La multiplicité et la variabilité de leurs besoins ont provoqué l'apparition d'un nouveau paradigme : « *la séparation des préoccupation*s » ou $ASoC^2$ [200]. Ce paradigme prône la séparation des préoccupations à la fois pendant la conception [67] et l'implémentation. Ainsi, la séparation des préoccupations a provoqué l'émergence de nouveaux langages « généraux » de programmation (dont la plupart ne sont que des extensions de langages existants), capables de séparer les préoccupations pendant l'implémentation. En parallèle, on constate l'apparition de méthodologies de développement [223] et de reengineering [168]. La séparation des préoccupations propose d'identifier les différentes facettes d'un système : ses parties non fonctionnelles (code de synchronisation, affichage de traces, traitement de la persistance, gestion de transaction, etc.) comme ses parties fonctionnelles (structures et comportement qui correspondent à la partie métier de l'application)[3]. Les premières permettent de répondre à un premier objectif qui est de généraliser l'approche d'intégration de services proposée dans le chapitre 2, tandis que la seconde et plus généralement la composition des deux permet d'adresser la problématique plus large de la réutilisation de préoccupations.

Une préoccupation est un concept générique décrivant une entité homogène composant le logiciel. La notion de préoccupation induit de nouvelles questions lors de la conception et de l'implémentation : *i)* quelles sont les entités qui ont la vocation à devenir une préoccupation ? *ii)* par quoi une préoccupation est-elle formée ?
Les préoccupations doivent être séparées les unes des autres pour permettre leur réutilisation dans différents contextes et une des difficultés est de s'affranchir des problèmes de couplages inter préoccupations. Par rapport au paradigme de l'objet, l'ASoC permet de gagner en mo-

[2] Advanced Separation of Concerns.

[3] Naturellement, si la fonction principale de l'application est par exemple de manipuler des traces, alors elle devient une préoccupation fonctionnelle. Ceci est applicable à n'importe quelle préoccupation non fonctionnelle.

dularité et en expressivité pour la conception et l'implémentation.

Pour être réutilisées (et participer à la construction d'un logiciel), les préoccupations sont composées entre-elles. Cette phase de composition nécessite le plus souvent des adaptations mutuelles entre les différentes préoccupations, ce qui suppose le transfert d'une partie du travail lié à la réutilisation au moment de la composition. Ce « tissage » des différentes préoccupations implique une composition le plus souvent non orthogonale, c'est a dire que des préoccupations peuvent modifier la sémantique d'autres préoccupations. Elle s'oppose à une composition de préoccupations orthogonales ou l'intersection de la sémantique entre les différentes préoccupations est nulle (elles sont donc toutes indépendantes les unes des autres).

Il est intéressant de noter que la séparation des préoccupations peut être étendue à la notion de séparation multidimensionnelle des préoccupations. L'objectif est de pouvoir classer les préoccupations dans plusieurs dimensions arbitraires ; une dimension représente un ensemble de préoccupations ayant des caractéristiques communes [247]. Dans la suite, nous donnons un aperçu des différents modèles qui peuvent s'intégrer au paradigme objet et permettent plus ou moins le support de la séparation des préoccupations.

Programmation générique et mixins. Un langage qui supporte la programmation générique permet de paramétrer les classes, c'est le cas de CLOS, de C++ et d'Eiffel [169, 210, 292] mais aussi de Java à partir de la version 1.5 [29]. Une classe ainsi paramétrée est dite générique, ses paramètres génériques formels définissent son degré d'ouverture. La programmation générique permet donc de séparer (découpler) la préoccupation liée à la définition d'une structure de données, de la spécification du contenu qu'elle permet de mémoriser et manipuler. La composition est effectuée lors de l'instanciation de la classe générique (ce sont les paramètres génériques qui fournissent l'information complémentaire qui est nécessaire à l'instanciation). Les *Mixins* [50, 99] et les *Mixins Layers* [283] sont deux techniques dérivées de la programmation générique qui améliorent la séparation des préoccupations. Un *mixin* est une classe générique dont le paramètre générique fixe sa superclasse, il permet d'implémenter *a priori* une extension des fonctionnalités existantes ou de nouvelles fonctionnalités qui sont orthogonales à un ensemble de classes.

Métaprogrammation. La métaprogrammation est née du besoin d'ouverture des langages ; les protocoles à métaobjets [169] formalisent le comportement d'un langage à objets à l'intérieur de métaobjets et implémentent sa sémantique [80, 54, 83, 82]. Ces métaobjets sont associés à une ou plusieurs classes qu'ils décrivent de manière fonctionnelle et structurelle et un métalangage qui peut être vu comme l'extension d'un langage à objets, est nécessaire pour les exprimer. Une métaclasse est une abstraction qui permet de décrire le comportement d'une ou plusieurs classes (ses instances) [267] ; elle est associée à une classe par une relation d'instanciation ou *lien méta*. De même que pour les classes, une relation d'héritage est définie pour les métaclasses, ce qui permet d'étendre et d'adapter le langage aux besoins de l'utilisateur de manière incrémentale et modulaire. Les métaobjets ont une vocation plus générale que d'autres modèles en ce qui concerne l'adaptabilité, car ils agissent directement au niveau du langage, ils identifient les points d'accès (ou points de jointure) du langage pour permettre de modifier son comportement.

Programmation par rôles ou points de vue. La programmation par rôles ou par point de vue [167] sont deux paradigmes assez proches ; ils permettent de déstructurer la notion d'objet

par l'introduction de vues subjectives : les rôles qu'ils jouent ou les points de vues qu'ils représentent. Notons par exemple, qu'un système comme Jiazzy [207] permet de programmer par rôles ou par points de vue.

Filtres de composition Les approches par filtres de composition [6] s'intégrent de manière satisfaisante dans le paradigme objet [35] ; elles associent des filtres aux objets ce qui permet d'intercepter les envois et réceptions de message qui sont eux-mêmes réifiés[4]. Les messages entrants (respectivement sortants) sont soumis à une collection ordonnée de filtres d'entrée (respectivement de sortie) qui ont la capacité d'accepter ou de rejeter un message et d'exécuter des *actions*. À chaque filtre correspond une expression de filtrage qui repose en particulier sur la vérification d'une expression booléenne mais il n'y a pas d'héritage entre filtres.

Les filtres, par leur impact sur les messages, modifient le comportement des objets sur lesquels ils sont appliqués, ce qui permet de remonter certaines préoccupations associées aux objets au niveau des filtres [36]. Parmi les systèmes qui proposent un support pour les filtres de composition, on trouve essentiellement des extensions de Java comme *ConcernJ* [65] ou *ComposeJ* [309], une extension pour Smalltalk [181], ou encore une extension de C++ [137].

Programmation orientée aspects. La programmation par aspects [172] est un modèle de séparation des préoccupations qui a dans un premier temps été basé sur le paradigme objet. Il permet de formaliser l'entrelacement des préoccupations dans les applications à partir d'une opération de tissage appelée aussi *crosscutting*[5]. Les éléments à entrelacer sont nommés *aspects* et sont des abstractions de préoccupations qui doivent être intégrées à un ensemble de classes. En d'autres termes, les préoccupations (en particulier celles qui sont transversales) sont implémentées par des aspects tandis que les comportements qui sont naturellement hiérarchisés le sont par des classes. Un aspect peut lui-même hériter d'autres aspects et encapsuler des variables et des méthodes d'instance ou de classe. Son instanciation peut être guidée par des politiques différentes (une instance par objet, une instance par classe d'objets, etc.). Parmi les principaux systèmes mettant en œuvre la programmation par aspects, on peut citer *Aspect/J* [150, 304, 17], *Caesar* [219, 218, 248, 220], Java Aspect Component (*JAC*) [258, 257, 256], *Aspect Moderator Framework* [85] ou encore, un système basé sur des diagrammes de séquence de messages (MSC) [303] et un métalangage basé sur la logique et les aspects [104].

Les endroits où les aspects s'intègrent dans l'application sont les points de jointure ou *joinpoint* dont la granularité peut être assez variable. Il peut désigner par exemple une méthode particulière, un ensemble de méthodes, toutes les méthodes d'une classe ou d'un ensemble de classes. Chaque point de jointure possède une information contextuelle associée qui est utilisable par l'aspect. Une fois attachée à un point de jointure, la méthode est exécutée chaque fois que le flot d'exécution du programme atteint ce point[6].

Programmation orientée sujets La programmation orientée sujets [153] s'intègre dans l'approche objet un peu de la même manière que la programmation par aspects. Ainsi, un sujet est un programme ou un fragment de programme décrit en suivant une approche par objets.

[4] Un filtre est lui-même un objet qui évalue et manipule des messages réifiés.

[5] Le tissage est réalisé, soit à travers une nouvelle étape de compilation qui va composer les aspects et le programme « normal » avant de l'envoyer au compilateur natif du langage, soit par une modification de l'interprète qui assure la composition des aspects à l'exécution.

[6] Il est possible aussi de préciser le moment d'exécution par rapport au point de jointure : *avant*, *après*, *autour*, *après exception* ou encore, *après retour de valeur*.

C'est la composition d'une collection de sujets qui produit l'application finale[7] ; on nomme *intégration* ce processus de composition. Les sujets sont vus de manière égalitaire et aucun sujet n'est prédominant par rapport à un autre. La composition est décrite par des règles [245] qui impliquent chacune au moins deux sujets et reposent sur des opérations simples (fusion, redéfinition et séquencement). L'*intégration* est encapsulée dans des modules de composition qui mettent en relation plusieurs sujets à intégrer ensemble et qui permet de développer des classes de manière décentralisée [244]. L'évolution de la programmation par sujets se place dans la continuité des travaux sur la séparation multidimensionnelle des préoccupations et se nomme programmation par *HyperSpace* [246, 159] ; sa principale particularité est d'imposer la propriété de complétude[8] aux sujets ou *HyperSlice* ce qui permet d'éliminer le couplage entre elles (les morceaux manquants sont obtenus lors de la phase d'intégration).

Bilan. Le survol des modèles proposé ci-dessus montre que :

- L'apport de la programmation générique est limité par rapport aux objectifs qui sont fixés.
- L'utilisation seule de la métaprogrammation pour séparer les préoccupations est restrictive et surtout très complexe à mettre en œuvre ce qui est très gênant si un des objectifs est d'augmenter la réutilisation.
- Même si conceptuellement la programmation par rôle ou par points de vue permet de supporter une forme de séparation des préoccupations, leurs différentes implémentations ne permettent pas de réconcilier les problèmes de partage d'information, et de recouvrement entre les différentes préoccupations.
- Les limitations de l'approche par filtre de composition ne permettent de considérer efficacement que les préoccupations qui sont orthogonales entres elles, même si on a introduit la super-imposition [37] afin de pouvoir exprimer de manière modulaire l'association de comportements à un ensemble de classes et limiter ainsi l'entrelacement des préoccupations.
- La programmation par sujets et la programmation par aspects offrent le support le plus intéressant ; nous résumons leurs apports et leurs limitations dans la section suivante.

3.2. Aspects et sujets : apports et limitations

Dans la section 3.1, nous avons donné un aperçu des différentes approches permettant de séparer et de composer des préoccupations ; les plus adaptées à la séparation des préoccupations s'appuient sur les sujets et sur les aspects. Dans [261], l'utilisation d'*AspectJ* et d'*Hyper/J* pour mettre en œuvre trois exemples qui reposent successivement sur des préoccupations fonctionnelles, non fonctionnelles et une combinaison des deux a permis de dresser un bilan sur les apports et limitations de chacune des approches.
On constate que, même si on réussit à séparer les préoccupations et enlever les dépendances qui peuvent exister entre elles, la principale difficulté reste de les composer pour produire une application, que ce soit dynamiquement ou statiquement. De plus, la complexité des préoccupations empêche la composition d'être entièrement automatisée.

[7] Cette composition peut être réalisée par étape et donner naissance à des sujets composites qui peuvent être à leur tour composés.

[8] Chaque méthode utilisée doit être décrite mais pas forcément implémentée dans le sujet ou HyperSclice.

N°	Description	Type d'adaptation	Cible de l'adaptation
1	Implémentation de nouvelles interfaces	fonctionnelle	Classe et ensemble de classes
2	Fusion de classes et de méthodes	fonctionnelle	Classe et ensemble de classes
3	Ajout et redéfinition de méthodes dans les classes	fonctionnelle	Classe et ensemble de classes
4	Ajout de variables de classe et d'instance à des classes	fonctionnelle	Classe et ensemble de classes
5	Interception *avant*, *après*, *autour* et *sur exception* des méthodes	non fonctionnelle	Méthode et ensemble de méthodes
6	interception des accès aux variables de classe et d'instance	non fonctionnelle	Variable et ensemble de variables

TAB. 3.1.: Adaptations nécessaires à la réutilisation de préoccupations.

Une autre constatation que l'on retrouve d'ailleurs dans [150, 304] montre que la composition d'une préoccupation est facilitée si elle est accompagnée de son protocole de composition. Ce protocole représente une abstraction de l'ensemble des adaptations qu'il est nécessaire de réaliser pour pouvoir composer la préoccupation à l'intérieur d'une application. Il doit pouvoir être spécialisé et concrétisé par héritage afin de proposer une solution qui s'adapte au contexte de chaque cas d'utilisation en complétant au besoin les définitions proposées par le protocole abstrait.
Tout ceci montre qu'un modèle adapté à la réutilisation des préoccupations doit permettre d'une part d'encapsuler le protocole de composition et d'autre part de décrire un protocole indépendant du ou des futurs contextes d'utilisation. Or les approches par objets et sujets ne permettent ni l'un, ni l'autre et la programmation par aspects n'offre qu'un support limité, du fait d'un modèle de description des adaptations incomplet.
Le modèle de description des adaptations est donc un élément clé pour tout langage qui voudrait favoriser la réutilisation de préoccupations. Ces adaptations sont synthétisées dans le tableau 3.1 qui est tiré de [261].

Concernant les adaptations présentées dans le tableau 3.1, les langages à objets existants ne permettent aucune de ces adaptations : *Hyper/J* implémente les types adaptations de *1* à *4* et partiellement *5*, *AspectJ* supporte tous les types d'adaptation sauf le *2*.

On vient de voir que pour pouvoir réutiliser une préoccupation (et donc pouvoir la composer facilement) il ne suffit pas de pouvoir décrire une adaptation il faut aussi pouvoir la décrire de manière indépendante du contexte (description abstraite) et de pouvoir ensuite la compléter (description concrète, adaptée au contexte d'utilisation). Or *Hyper/J* ne permet pas de décrire de manière abstraite n'importe laquelle des adaptations que le langage supporte et, *AspectJ* offre cette facilité seulement pour les adaptations *5* et *6*. De plus, il ne suffit pas de pouvoir abstraire la description d'une adaptation, il faut aussi pouvoir spécifier le plus tard possible, c'est-à-dire au moment de la description concrète, quelles sont les cibles de l'adaptation. *Hyper/J* n'offre pas cette possibilité quel que soit le type de la cible, tandis qu'*AspectJ* le propose

uniquement pour les méthodes et les ensembles de méthodes. Par ailleurs, sans même parler d'abstraction, aucun des deux langages ne permet de représenter un ensemble de classes par des expressions régulières et de son côté *Hyper/J* ne l'offre pas non plus pour les méthodes.
Un autre point concerne les objectifs d'*Hyper/J* et d'*AspectJ* : le premier est plutôt dédié aux préoccupations fonctionnelles alors que le 2ème est particulièrement adapté aux préoccupations non fonctionnelles, même s'il s'adresse aussi aux préoccupations fonctionnelles. Un des effets de bord de la priorité affichée par ces langages pour l'une ou l'autre des catégories de préoccupation est que la composition de préoccupations se fait *in situ* dans *AspectJ* et *ex situ* dans *Hyper/J*. En d'autres termes, cela signifie que la composition donne lieu à la création d'une nouvelle entité (le plus souvent une classe) dans *Hyper/J* alors qu'en ce qui concerne *AspectJ*, la modification est faite dans l'entité elle-même.
Les résultats décrits ci-dessus montrent clairement que malgré des avancées par rapport à la programmation par objets qui sont indiscutables, *Hyper/J* et *AspectJ* n'offrent pas toutes les qualités nécessaires à une bonne réutilisation des préoccupations. Il y a donc matière à décrire un modèle dédié à la séparation et à la composition des préoccupations dans le cadre du développement d'application à objets. Ce qui aura pour conséquence d'offrir un mécanisme pour intégrer de manière flexible de nouveaux services correspondant à des préoccupations fonctionnelles ou non. Les principaux éléments de ce modèle sont proposés dans la section 3.3.

3.3. Vers un modèle métier dédié à la réutilisation des préoccupations

Cette section propose une synthèse du modèle proposé par Laurent Quintian dans sa thèse, revu pour certains points dans [191]. Il supporte l'ensemble des adaptations décrites dans la section 3.2 ainsi que les capacités d'abstraction requises mentionnées dans la même section et fournit les deux types de composition.

Approche MDA et modèle métier L'approche qui est proposée pour décrire et mettre en œuvre le modèle proposé s'accorde parfaitement avec les travaux et les perpsectives proposés dans la partie III. En effet, la description de ce modèle suit l'approche MDA [229] et correspond à un modèle métier dédié à la description de la séparation et de la composition des préoccupations dans une application. Plus précisément, le modèle repose sur une réification des principales entités d'un modèle objets et sur de nouvelles entités dont la principale décrit le concept d'adaptateur. Les instances de ce modèle permettent donc de décrire *i)* les différentes préoccupations d'une application suivant une approche par objets et *ii)* le protocole de composition qui décrit les adaptations. L'application sera ensuite transformée de manière *in situ* ou *ex situ* en fonction des besoins, pour que la réification ainsi modifiée corresponde à l'application que le concepteur veut obtenir. Ces transformations seront appliquées sur l'ensemble des entités de l'application en s'appuyant sur les adaptateurs qui leur sont associés et sur les adaptations qu'ils décrivent. En d'autres termes, cette approche est centrée sur le modèle qui sera associé à un langage dédié (Domain-Specific Language - DSL) [301, 30].

Modélisation des préoccupations Une préoccupation est encapsulée dans un conteneur (*package*, *cluster*)[9], lui-même composé de classifieurs (*class*, *interface*, etc.) ou d'autres conteneurs. La figure 3.1 propose une vision synthétique de la description UML [233] d'un langage ou plutôt des entités de ce langage qui sont nécessaires pour décrire les adaptations qu'il faudra implémenter afin de composer les préoccupations. On constate que les adaptations à réaliser reposent sur des concepts qui sont présents dans la plupart des langages et il est donc raisonnable de proposer une réification commune de l'ensemble de ces propriétés (il est bien sûr possible d'augmenter cette réification à tout moment sans *casser* l'architecture du modèle. Un classifieur (*class*, *abstract class*, *deferred class*, *interface*, etc.) peut hériter d'un ou plusieurs autres classifieurs. Il est toujours défini dans un conteneur et il a des qualifieurs (*abstract*, *deferred*, *final*, *freeze*, *public*, etc.). Il contient des attributs (constantes, variables d'instance ou de classe), des méthodes (procédures, fonctions ou constructeurs) qui nécessitent ou pas la création d'instance. Les méthodes ont une signature (nom, paramètres, type de retour, etc.), des qualifieurs et un corps qui correspond à un ensemble d'instructions.
Il est à noter que le concepteur de l'application n'aura pas à manipuler cette réification directement puisqu'il disposera d'un langage de programmation par objets pour décrire son application. La réification sera ensuite construite à partir du source de l'application en utilisant des outils de transformation de programme. Un aperçu d'une mise en œuvre possible sera donné ci-dessous.

Modélisation de la composition Après la description de la réification des concepts du langage à étendre, nous nous intéressons aux adaptations à réaliser sur les préoccupations afin de les composer entre elles. La description d'une adaptation repose sur les concepts réifiés et sur une ou plusieurs règles de transformation et de typage. Elle est située à l'extérieur des classes à adapter ; on dit que ces adaptations sont *non invasives*.

La figure 3.2 présente les principales entités qui permettent de décrire la composition de préoccupations et donc la mise en œuvre d'une application. Cette composition est représentée par un ou plusieurs *adaptateurs*[10]. Un adaptateur a un nom unique qui permet de l'identifier, il peut être concret ou abstrait et il peut hériter d'un autre adaptateur. Ce mécanisme d'héritage est simple et il ne permet que deux des usages de l'héritage définis par B. Meyer [213] : l'héritage de concrétisation (*reification inheritance*) et l'héritage de redéfinition de comportement (*functional variation inheritance*). Un adaptateur contient des *cibles d'adaptation* concrètes ou abstraites[11] selon que les entités sur lesquelles portent l'adaptation sont spécifiées dans la déclaration ou pas. Une cible d'adaptation est typée et concerne soit un classifieur, soit une méthode, soit un attribut. Lors de la déclaration d'une cible d'adaptation abstraite, il est possible de fixer des contraintes à respecter. Une cible d'adaptation concrète contient soit une liste explicite d'identificateurs (selon son type, des noms de classifieur, de méthode ou d'attribut), soit une expression régulière[12] dont le calcul produira une liste d'identificateurs. Un adaptateur contient aussi des *opérations d'adaptation* typées par la cible de l'adaptation. Si l'adaptateur est défini *ex situ*, un nouveau *conteneur* (un *package* dans l'implémentation actuelle) sera créé et les préoccupations concernées par les adaptations y seront placées. S'il est

[9] Nous utilisons des termes de différents langages de programmation (Java, Eiffel, etc.) pour mettre en évidence le fait que notre approche n'est pas limitée à un langage particulier.
[10] À l'heure actuelle le mécanisme pour composer des adaptateurs est très limité et garantit seulement que les adaptateurs *ex situ* (qui peuvent créer de nouvelles préoccupations) sont traités en premier.
[11] S'il contient au moins une cible d'adaptation abstraite, l'adaptateur est lui-même abstrait.
[12] Son expressivité est tout à fait similaire à celle que l'on peut trouver dans AspectJ [171].

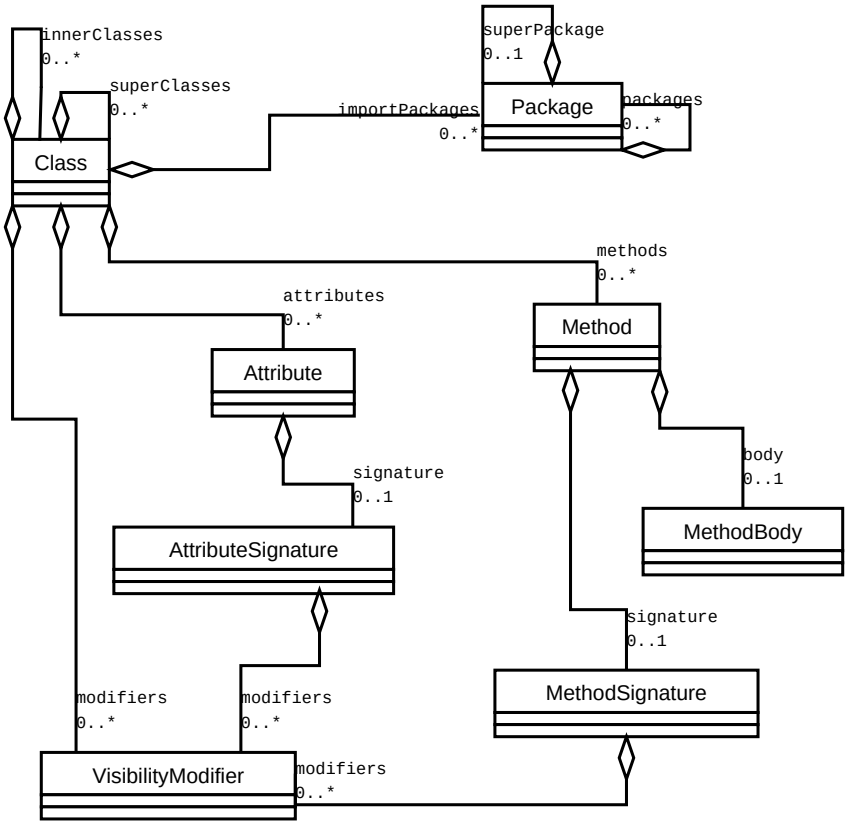

FIG. 3.1.: Concepts d'une préoccupation pris en compte pour la composition

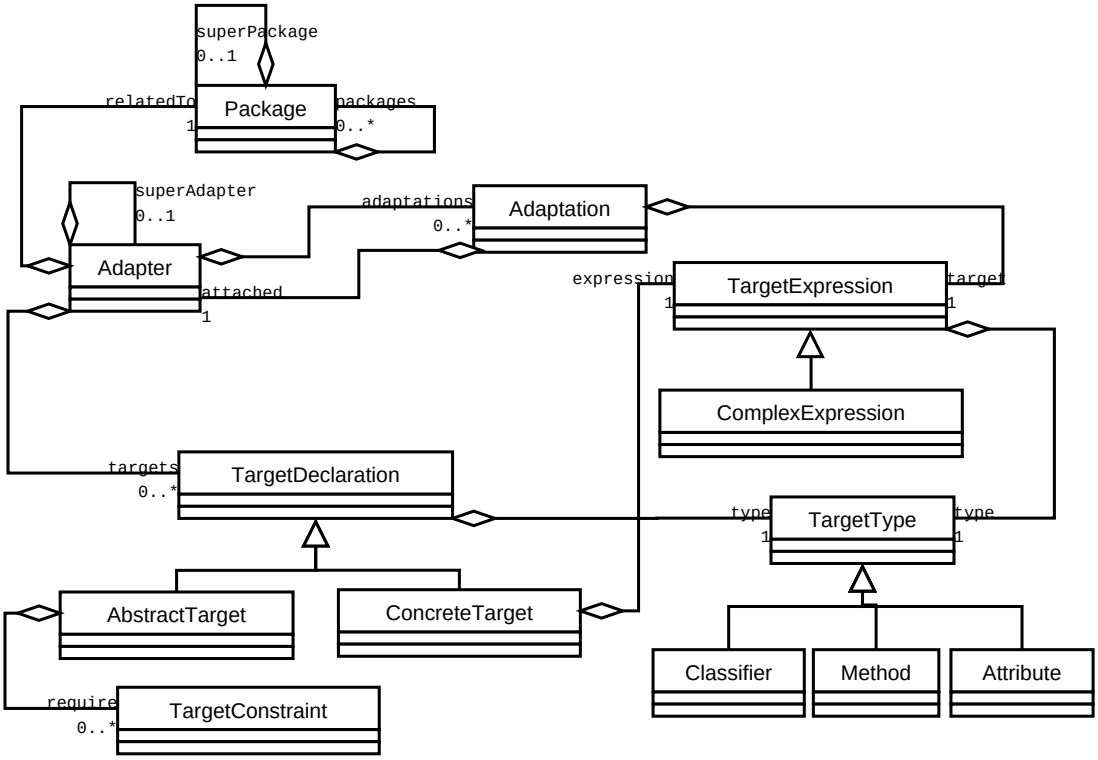

FIG. 3.2.: Réification de la composition de deux préoccupations

in situ, alors le conteneur existe déjà et le résultat de l'adaptation modifiera la préoccupation qui est associée à celui-ci.

Types d'adaptation supportés Notre modèle supporte cinq grandes catégories d'adaptation qui regroupent chacune un ou plusieurs types d'adaptation ; leur nombre pourra être étendu par la suite si de nouvelles capacités d'adaptation se révèlent utiles. La figure 3.3 montre que les choix de conception du modèle permettent d'intégrer à tout moment de nouvelles adaptations sans remettre en cause l'architecture du modèle ou de son implémentation. Les cinq grandes catégories d'adaptation sont synthétisées dans le tableau 3.1 qui indique en particulier si une adaptation introduit une nouvelle fonctionnalité (adaptation fonctionnelle) ou si elle complète la description de fonctionnalités existantes afin de leur associer de nouveaux services (adaptation non fonctionnelle).

Les adaptations proposées par notre modèle s'intègrent ainsi dans ces catégories en conservant l'ordre du tableau 3.1 : *i)* implémentation de nouvelles interfaces ou ajout d'une superclasse (*SuperClassifierIntroduction*) ; *ii)* fusion de méthodes (*MethodMerging*) avec plusieurs variantes permettant de spécifier que le corps de la première méthode est placé avant (*MergingBefore*), après (*MergingAfter*) ou si l'une des deux méthodes est abstraite (*MergingWithAbstract*), et fusion de classes (*ClassMerging*) avec la possibilité, soit de seulement ajouter des méthodes ou des attributs (*FeaturesOnlyadded*), soit de réellement fusionner le corps des méthodes (*CustomizedMerging*) ; *iii)* ajout (*MethodIntroduction*) ou redéfinition de méthode (*MethodRedefinition*) ; *iv)* ajout de nouvelles variables d'instance (ou de classe) à des

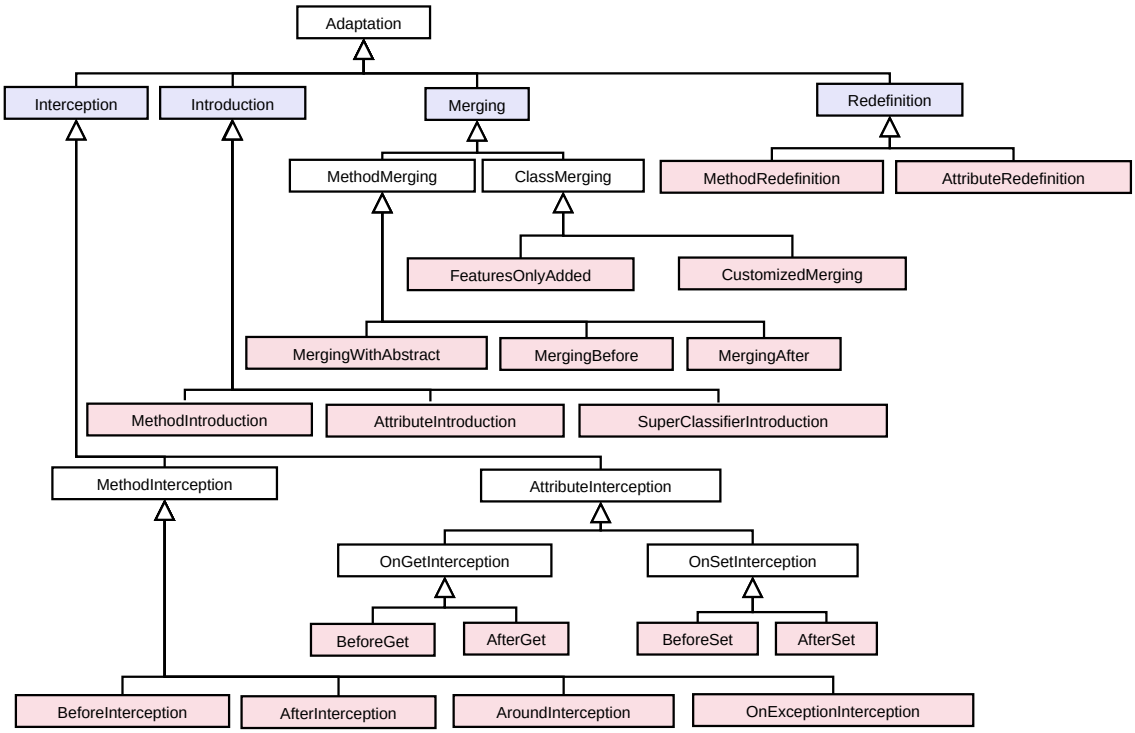

FIG. 3.3.: Classification des adaptations offertes actuellement

classes (*AttributIntroduction*) ; *v)* interception avant (*BeforeInterception*), après (*AfterInterception*), autour (*AroundInterception*) ou sur exception des méthodes (*OnExceptionInterception*), qu'elles soient d'instance ou de classe ; *vi)* interception des accès aux variables d'instance ou de classe (*OnGet/OnSet*). Là aussi nous autorisons l'intégration d'un traitement avant et après la consultation (*BeforeGet* et *AfterGet*) ou avant et après la modification de la variable (*BeforeSet* et *AfterSet*).

Éléments de mise en œuvre Le modèle qui a été présenté a été implémenté ; nous donnons un aperçu de l'implémentation qui a été réalisée[13]. L'implémentation de l'extension que nous proposons est réalisée à l'intérieur de la plate-forme Eclipse [121] en étendant le *plug-in* Java qui est inclus dans la distribution. Cette extension a été développée comme un précompilateur qui est appelé avant que le compilateur Java ne soit lui-même exécuté. Les principales tâches réalisées par ce précompilateur sont : *i)* l'extraction des informations contenues dans les arbres de syntaxe abstraite correspondant au code source et aux adaptateurs, *ii)* la vérification de la cohérence des informations, l'aplatissement de la hiérarchie des adaptateurs et la composition des préoccupations et, *iii)* la génération du code source après la composition.

Nous avons choisi d'utiliser la technologie XML et en particulier les *XML-Schema* [305] comme modèle pivot pour décrire les adaptateurs associés à une préoccupation pour plusieurs raisons : *i)* la description des adaptateurs est ainsi indépendante de tout langage et elle est donc accessible à tous et la réutilisation du protocole de composition est donc facilitée, *ii)* il est possible de bénéficier du grand éventail d'outils dédiés à la technologie XML comme

[13]Une description plus détaillée est disponible dans [261].

par exemple l'éditeur de fichier XML d'Eclipse ou la bibliothèque JAXB [293] qui permet de générer automatiquement un ensemble de classes Java à partir de l'information contenue dans des fichiers XML, *iii)* il est facile de produire une vue des adaptateurs avec une syntaxe adaptée [191]. Il est ensuite aisé de proposer un analyseur pour cette syntaxe qui produit le fichier XML et permet donc au programmeur de bénéficier d'un accès plus convivial pour décrire un adaptateur.

Deuxième partie .

Modélisation des concepts objets et ses applications

Page vide

Préambule

Le point de départ de ce travail repose sur plusieurs constatations :

- L'intégration de la persistance dans les langages à objets [183] fait apparaître tout un ensemble de problématiques qui ne cessent de se diversifier (gestion de l'évolution, distribution sur le réseau, mobilité, interopérabilité, etc.). Elles sont de plus en plus nécessaires à l'écriture et à la maintenance d'applications dites modernes. Il semble donc intéressant d'étendre les langages de telle manière à pouvoir les inclure tout en ne polluant pas le code (métier) de l'application.
- La sémantique des concepts de classes (*classifiers* en UML) et de relations (notamment la relation d'héritage), sont souvent spécifiques à un langage donné et relativement de bas niveau du point de vue conceptuel. De son côté, UML ne propose pas une sémantique claire mais offre des mécanismes d'extension pour spécifier des relations adaptées au contexte [239]. Cependant, cette information conceptuelle est le plus souvent perdue lors de la projection vers les langages de programmation. Il semble donc intéressant d'une part, de réduire le fossé qui existe entre les phases de conception et de programmation et d'autre part, de pouvoir comparer la sémantique associée aux différents concepts présents dans des langages à objets de l'état de l'art.
- Les mécanismes associés aux classes et aux relations dans les langages de programmation ont souvent une expressivité qui n'est pas adaptée aux besoins : elle peut être soit trop grande, soit au contraire trop faible ; dans les deux cas, cela nuit à la réutilisabilité. Pour dire cela, nous nous appuyons sur une expérience de développement important réalisé pour le compte de la Food Agriculture Organization [187, 186, 185]. Ce projet a permis à la fois de tester en vraie grandeur le bien-fondé des techniques à objets soutenues par Bertrand Meyer [209, 213], dans un projet où les contraintes de coût, d'évolutivité et de portabilité étaient particulièrement sévères[14], et de montrer certaines limites des concepts mis en œuvre. Quelques-unes de ces limitations sont décrites dans [164].
- Il peut être intéressant d'offrir un support pour expérimenter de nouvelles relations ou de nouvelles sortes de classes et pour adapter l'expressivité de ces concepts au contexte d'utilisation.

Pour répondre à ces constatations, nous proposons un modèle paramétrable (voir chapitre 4) pour décrire la sémantique opérationnelle des concepts objets que l'on peut trouver à la fois dans les méthodes de conception et les langages de programmation. Dans un second temps, nous présentons dans le chapitre 5 un certain nombre de travaux qui dérivent des utilisations possibles de ce modèle ou des idées sous-jacentes.

[14] Un haut niveau de lisibilité du code, de sa structure et de sa documentation interne était donc requis.

Page vide

4. Modélisation des concepts objets

Les raisons pour capturer la sémantique opérationnelle d'une application sont diverses. Dans la phase de conception, l'objectif peut être de pouvoir faire des contrôles ou de générer le code source correspondant le plus automatiquement possible et ainsi réduire le fossé entre modèle de conception et langages de progammation. Si cette capture intervient pendant la phase de programmation, elle peut *i)* fournir les informations nécessaires à un environnement de programmation correspondant, *ii)* enrichir ou modifier le comportement de l'application lorsque cette possibilité est offerte par le langage. Capturer la sémantique opérationnelle peut permettre aussi d'introduire une certaine forme d'interopérabilité.
Nous nous sommes donc intéressés à plusieurs langages de programmation dont Java [16] et Eiffel [210, 212] afin d'étudier et de comparer leur sémantique ; il est à noter que Gilles Ardourel, avec qui nous avons et nous continuons à collaborer a réalisé dans le cadre de sa thèse un comparatif détaillé de la sémantique des langages par rapport aux règles de visibilité et de protection [13]. Concernant ces langages, nous nous sommes attachés à étudier les deux types d'entités phares des langages à objets : les classes et les relations entre classes (héritage, agrégation, etc.).
Dans un premier temps, dans la section 4.1, nous donnons un aperçu de l'état de l'art, puis nous présentons les principaux éléments de l'approche (section 4.2), c'est-à-dire le cœur du modèle et enfin dans la section 4.3, nous évoquons les extensions sur lesquelles nous avons travaillé.

4.1. Survol de l'état de l'art

Pour couvrir les aspects évoqués ci-dessus, il est nécessaire d'examiner les contributions suivantes : *i)* les langages métaobjets et les métamodèles, *ii)* les modèles et méthodes de conception.

Langages métaobjets et métamodèles Nous avons déjà évoqué dans la section 3.1 la réflexivité et la métaprogrammation pour séparer les préoccupations, nous nous attardons ici sur son apport pour l'ouverture et la flexibilité des langages. Ainsi, un protocole métaobjet peut être vu comme un système, un ensemble d'opérations ou de composants logiciels qui permet de représenter et d'adapter le comportement d'un langage orienté objets. Nous nous sommes intéressés à plusieurs systèmes et parmi eux, certains proposent des moyens syntaxiques[1] pour redéfinir dans le programme le comportement de l'application, c'est le cas de Clos [169, 166] et de Smalltalk [138, 140] qui proposent un protocole métaobjet. Ces langages sont largement réflexifs et proposent au minimum une ouverture des classes et des méthodes, mais celle des relations n'est pas vraiment privilégiée. En effet, même si dans le cas de Clos, il est possible de redéfinir le comportement associé aux relations inter-classes, il est très difficile de modifier en profondeur ces relations. Pour cela il faudrait réécrire, ou tout au moins compléter, des

[1] Il y a un seul langage à la fois pour la métaprogrammation et la programmation.

algorithmes méta qui sont loin d'être évidents[2]. Smalltalk va plus loin dans ses capacités de réflexion car les exceptions, les processus, la mémoire et même les arbres syntaxiques, voire la pile d'exécution ou du compilateur sont aussi réifiés. De nombreux travaux se sont ainsi basés sur Smalltalk tels Classtalk [83, 81], Neoclasstalk [267] et Metaclasstalk [46]. La principale approche qui est proposée pour modifier le comportement des applications est que chacune des entités d'un programme pour laquelle on veut réaliser une ouverture est représentée par une métaclasse qui peut être spécialisée ; dans le cas de Clos, ceci est complété par la notion de méthode générique qui permet de réaliser la liaison dynamique pour pallier la non encapsulation des méthodes dans la classe. Smalltalk et Clos sont non typés et donc reposent sur une philosophie de contrôle exclusif à l'exécution ; même si Clos en particulier reste simple à manipuler, cela a des conséquences sur la robustesse des applications. Un autre aspect important de ces deux langages est que par nature il n'y a pas de séparation des phases de métaprogrammation et de programmation, ce qui nous a semblé être une cause importante de problèmes. En effet, la capacité de modifier un langage pendant son utilisation fournit certes une liberté motivante mais reste néanmoins une énorme source d'erreurs et de complexité.
Au contraire, d'autres langages comme OpenC++ [70, 71] ou Openjava [298, 73] agissent à la compilation pour créer un nouveau langage. C'est-à-dire que le niveau méta ne se retrouve pas lors de l'exécution ; si cela limite leur flexibilité, cela augmente d'autant les performances des programmes qu'il décrivent.

Javassist [299, 72], Iguana [144, 113] et Metaxa (initialement appelé MetaJava) [179, 141] sont un peu à la frontière entre les approches OpenC++ et Smalltalk. Tous interviennent à la compilation mais aussi à l'exécution en maintenant la réification d'au moins une partie du niveau méta ; le choix des entités réifiées est soit codé en dur (notamment Metaxa), soit fixé par le programmeur comme dans Iguana. Leur capacité d'ouverture est assez similaire à celle d'OpenC++ ou de Clos. Il est intéressant de noter que Javassist se présente comme un assistant mettant à la disposition du programmeur un système d'aide à la programmation. Il est conçu pour produire facilement une nouvelle classe sur une requête, plutôt que pour adapter le comportement d'une classe existante, ce qui est souvent l'apanage des systèmes métaobjets. L'approche de Metaxa est originale : il possède un niveau méta qui est activé par des événements déclenchés par le programme lui-même.

Le langage Flo [116, 115] mérite d'être considéré à part car il est le seul à proposer des métaobjets pour les relations. Il utilise les caractéristiques réflexives de Clos pour assurer le maintien de la cohérence de dépendances entre objets. Les dépendances elles-mêmes sont réifiées et disposent donc d'un métaobjet correspondant, ce qui se trouve être parfaitement dans le fil de notre approche sauf que nous nous intéressons plus aux relations interclasses. Il existe aussi d'autres protocoles métaobjets que ceux cités ci-dessus. Rien que pour Java, citons par exemple Guarana [228], Kava [308] (anciennement Dalang [307]) et Proactive [66].

Par ailleurs on peut voir aussi apparaître dans plusieurs de ces langages tels que Clos, OpenC++, en complément de la présence de métaclasses, la possibilité d'ajouter des bouts de code au moyen de routines qui s'exécutent avant, après ou autour (seulement pour Clos). Ceci est complété dans OpenC++ par la notion de *wrapper* modélisée par une métaclasse.

La plupart de ces protocoles métaobjets offrent des points d'entrée à des moments clés de

[2]Surtout que la notion de relation n'est pas réifiée et que les algorithmes sont largement dispersés.

l'exécution du programme modélisé. Nous retrouvons notamment parmi ces derniers l'*appel d'une méthode*, l'*appel d'un constructeur*, l'*accès à un attribut* ou la *création d'un objet*. Il est également parfois possible d'intervenir lors du *chargement d'une classe*. Par rapport à l'ouverture des relations entre classes, il est possible, par un détournement de l'appel de méthode, de métaprogrammer un comportement particulier simulant une relation spécifique. Cependant, cette technique reste particulièrement lourde à mettre en œuvre car il est alors nécessaire de réécrire bon nombre d'algorithmes du langage (structure du schéma de types, liaison dynamique, masquage, redéfinition, etc.).

La classe est et reste la structure de base à métamodéliser pour tous ces protocoles métaobjets, l'appel de méthode étant le moment privilégié où les algorithmes de niveau méta sont activés. Nous pensons, pour notre part, que si la classe est importante, les relations entre classes le sont encore plus. Aussi le modèle OFL dont les grandes lignes seront présentés dans la section 4.2 met ces relations au centre de son système.

Autres systèmes et modèles À partir du moment où nous voulons pouvoir décrire les concepts objets présents dans les langages dans le but de proposer, par exemple, des extensions, des outils d'aide à la programmation ou d'introduire de nouveaux services, il faut évoquer UML (*Unified Modeling Language*) [45, 268, 161, 235]. Par rapport à nos préoccupations, il est intéressant par plusieurs aspects : *i)* il fournit un formalisme graphique qui est utilisé et compris par l'ensemble de la communauté *objets*, *ii)* la présence de relations entre classes ne se limite pas à celles présentes dans les langages de programmation (par exemple il y a aussi la relation d'*association*) et il est possible de signaler un type de relation spécifique en nommant la flèche qui unit les classes entre elles. Cependant la définition de la sémantique des concepts graphiques utilisés reste pauvre et fait apparaître un besoin pour réduire le fossé entre méthode de conception et langage de programmation même si des solutions à base de patrons de conception [133] peuvent apporter des solutions intéressantes.

MOF (*Meta-Object Facility*) [90, 232, 240] est un format de description de métamodèles, en quelque sorte un métamétamodèle. MOF apporte principalement deux avantages : *i)* Il est un moyen standard de décrire des capacités et structures de niveau méta. En cela, il permet de reposer sur un formalisme reconnu et adapté à la description de métamodèle. *ii)* Il garantit que tous les métamodèles qui l'utilisent comme format de description peuvent stocker leurs données au sein d'une base commune. MOF améliore donc également l'interopérabilité entre métamodèles ; d'ailleurs, depuis la spécification de MOF 2.0 et UML 2.0 [237, 239, 238] il y a une partie qui est commune aux deux métamodèles.

XML (*eXtensible Markup Language*) [231] n'a pas vocation à être utilisé comme format de base pour chaque application mais plutôt comme support d'exportation et d'importation rendant les données indépendantes des programmes qui les traitent. Ce format n'est pas forcément adapté à tous les usages mais le fait que le format soit ouvert contrebalance avantageusement cet inconvénient. Par exemple bien que dédié à la manipulation de données, XML peut mémoriser des algorithmes (au niveau méta, les programmes et les algorithmes peuvent être considérés comme des données).

Signalons aussi l'existence de XMI (*Xml Metadata Interchange*) [234, 241], système dont le but est de permettre l'échange de métadonnées XML entre des outils de modélisation basés sur UML ou sur MOF.

4.2. Le modèle OFL

L'approche qui a été choisie pour capturer la sémantique d'une relation ou d'une classe est basée sur l'hyper-généricité, c'est-à-dire le paramétrage des classes et des relations inter-classes afin d'obtenir le comportement souhaité par le programmeur.

Nous avons proposé un ensemble de paramètres généraux qui représentent les principales caractéristiques des relations entre classes dans les langages à objets courants. Pour mettre en œuvre un tel système, il a fallu modéliser la notion de *relation*. Ces relations sont posées entre des *classes*. Nous nous sommes donc également intéressés à celles-ci. Enfin, les relations et les classes sont généralement associés à un langage précis. Nous avons donc défini un concept de *langage*.

Nous appelons *concept-relation* l'entité représentant un type de relation : un concept-relation est donc une métarelation. Sa description représente un de nos principaux objectifs, mais pour mieux appréhender la totalité du contexte, il est nécessaire d'évoquer les deux autres concepts importants du modèle : le *concept-description* et le *concept-langage*.

Un *concept-description* permet de définir la notion de classe telle qu'elle existe dans les langages à objets, c'est donc une sorte de métamétaclasse. Notre objectif est de pouvoir faire coexister plusieurs concepts-descriptions à la sémantique différente, comme c'est le cas dans des langages comme Java (où nous décrivons notamment les notions d'interface et de classe comme deux concepts-descriptions). Chaque concept-description définit un ensemble de concepts-relations avec lesquels il est compatible et gère leurs interactions.

Le *concept-langage* est une notion importante et simple. Il modélise, comme son nom l'indique à l'évidence, un langage. Chaque langage est constitué en particulier d'un ensemble de concepts-descriptions et d'un ensemble de concepts-relations, chacun étant compatible avec au moins un des concepts-descriptions sélectionnés.
Notre volonté, tout au long de l'élaboration du modèle OFL, a été de permettre au métaprogrammeur de modifier le comportement, la sémantique opérationnelle, des composants de son langage. Cependant, réaliser tout cela par métaprogrammation serait une tâche particulièrement rebutante. Nous avons donc choisi de mettre en œuvre le principe d'hyper-généricité [106] et nous avons sélectionné d'une part un ensemble de paramètres dont la valuation permet de décrire le comportement d'un composant et d'autre part, un système d'actions prédéfini qui exploite la valeur de ces paramètres (voir les définitions ci-dessous).

Définition de l'hyper-généricité : L'hyper-généricité est le paramétrage du comportement d'un système informatique selon des critères simples. Le terme *hyper-généricité* est utilisé par extension du terme *généricité* qui, dans les langages à objets décrit la capacité d'une classe à être paramétrique.

Définition d'une action : Une action, dans le modèle OFL, est une routine qui met en œuvre un comportement des langages décrits par le modèle. L'algorithme d'une action tient compte de la valeur des paramètres hyper-génériques. La recherche de la méthode adéquate en exploitant la liaison dynamique, l'envoi de message, ou encore l'affectation d'un objet à un attribut sont des exemples d'action.

Chaque instance de concept possède donc des paramètres mais seuls les concepts-relations et concepts-descriptions sont équipés d'actions (les paramètres et les actions sont partagés

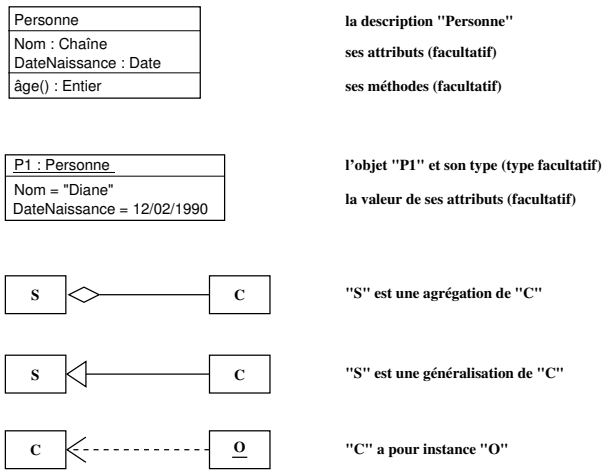

FIG. 4.1.: Légende générale.

par toutes leurs instances, ces derniers pouvant les redéfinir localement pour les affiner) et d'un protocole pour les métaprogrammer. Le travail du métaprogrammeur peut prendre deux formes : soit il attribue simplement une valeur aux paramètres décrits dans les concepts et définit ainsi les composants de son langage, soit il souhaite modifier la sémantique associée à au moins un de ces paramètres. Dans le premier cas, son travail est terminé et c'est là l'un des intérêts de notre approche. Dans le second cas, qui ne doit s'appliquer qu'en dernier ressort, il lui faut redéfinir les actions dont l'algorithme ne convient pas.

Les actions ont été classées en six thèmes et pour chacun nous donnons un ou plusieurs exemples : *i)* la recherche de primitive (recherche d'une primitive dans le cadre de la liaison dynamique), *ii)* l'exécution de primitive (exécution d'une méthode, lecture d'un attribut ou envoi de message), *iii)* le contrôle (vérification des paramètres effectifs d'un appel de primitive par rapport aux paramètres formels), *iv)* la gestion d'instance de description (création d'une instance), *v)* la gestion d'extension de description (action avant ou après la création d'une instance) et, *vi)* les opérations de base (l'affectation ou la copie d'objet).

Les actions de concepts-descriptions font un usage intensif de celles de concepts-relations. Il n'est donc pas étonnant de constater que les actions de concepts-descriptions sont distribuées en six thèmes reprenant ceux des concepts-relations. De manière générale, le même nom d'action est utilisé pour les concepts-descriptions et concepts-relations, celles se trouvant dans les premiers appelant souvent celles qui se trouvent dans les seconds, le tout pour réaliser une opération unique (telle la liaison dynamique par exemple).

4.2.1. L'architecture du modèle

Nous présentons tout d'abord, dans la figure 4.1, la légende qui s'applique à tous les schémas concernant le modèle OFL. La notation choisie s'apparente à celle d'UML.

La figure 4.2 illustre l'utilisation du modèle OFL pour décrire une application ; elle représente à la fois une hiérarchie méta représentée par des liens d'instanciation mais aussi une hiérarchie de descriptions liées par spécialisations. Dans l'arbre d'instanciation, chaque niveau n'est relié directement qu'au niveau supérieur, par contre ce n'est pas le cas pour l'arbre de

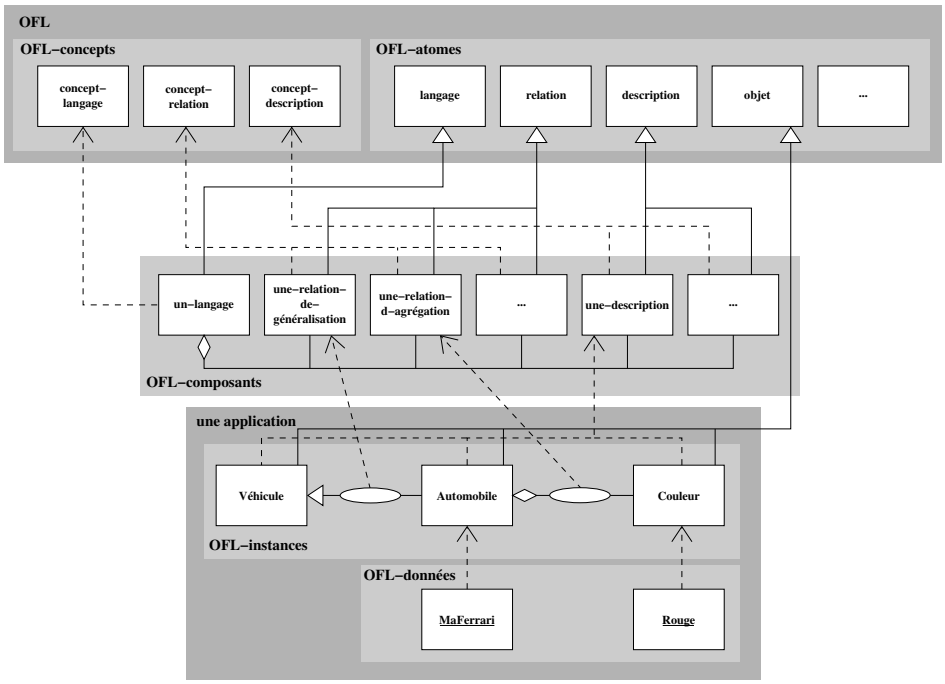

FIG. 4.2.: Architecture d'OFL

spécialisation dont l'objectif est structurel : modéliser la réification des informations qu'elles soient méta ou pas[3].

Nous montrons dans cette figure les trois niveaux de modélisation nécessaires : *i)* le niveau application regroupe les descriptions et les objets du programme (*OFL-instances* et *OFL-données*), il est donc créé par le programmeur, *ii)* le niveau langage décrit les composants du langage de programmation (*OFL-composants*), il est sous la responsabilité du métaprogrammeur et *iii)* le niveau OFL représente la réification de ces composants (*OFL-concepts* et *OFL-atomes*), c'est-à-dire notre modélisation de la sémantique opérationnelle des descriptions et relations. L'introduction d'un vocabulaire nouveau a pour objectif de minimiser les quiproquos concernant les termes employés. Le lecteur pourra trouver par exemple qu'une description s'apparente à une classe et qu'un *composant-description* se rapproche de la notion de métaclasse.

Le niveau application Pour décrire une application, le programmeur utilise les services offerts par le niveau langage. Il crée, au niveau application, des OFL-instances, qui sont les descriptions et les relations de son application, par instanciation des OFL-composants. À l'exécution, les objets de l'application, nommés OFL-données, sont des instances des OFL-instances représentant les descriptions :

– **Les *OFL-instances***. Chaque description ou relation décrite par le programmeur est modélisée par une OFL-instance. La figure 4.2 propose un exemple d'application qui comprend cinq OFL-instances : *i)* trois descriptions : *Véhicule*, *Automobile* et *Couleur*, *ii)* une relation de généralisation : *Automobile* hérite de *Véhicule* et *iii)* une relation

[3]Nous revenons sur cette distinction au fur et à mesure de l'examen de la figure 4.2.

d'agrégation : *Automobile* a un attribut de type *Couleur*.
- **Les *OFL-données***. Dans l'application, chaque instance de description est réifiée à l'exécution par une OFL-donnée. La figure 4.2 en présente deux : *i) MaFerrari*, instance de la description *Automobile* et *ii) Rouge*, instance de la description *Couleur*. Remarquons que les OFL-instances qui représentent des descriptions spécialisent l'OFL-atome ***objet***. En effet, ***objet*** est la réification des données de l'application (OFL-données) et constitue donc la racine de l'arbre de spécialisation des OFL-instances représentant des descriptions. Il contient par exemple la collection des attributs d'une instance de description.

Le niveau langage Le niveau langage décrit les différentes sortes de relation et de description qu'il est possible d'utiliser dans le langage modélisé. Les relations sont des instances de concept-relation, les descriptions des instances de concept-description. Le langage lui-même est une instance de concept-langage. Ces instances sont appelées *OFL-composants*.

La figure 4.2 recense : *i)* des composants-descriptions dont ***une-description***, *ii)* des composants-relations dont ***une-relation-de-généralisation*** et ***une-relation-d-agrégation*** et *iii)* un composant-langage ***un-langage***. Il est possible de se représenter un composant-description sous la forme d'une métaclasse, un composant-relation comme une métarelation et, de la même manière un composant-langage comme un métalangage. Les entités méta, en plus de l'aspect comportemental qui leur est associé, contiennent un ensemble d'informations fixe. Ces informations sont importées des OFL-atomes au travers d'une spécialisation.

Le niveau OFL Le niveau OFL constitue un métamodèle pour le langage de programmation (niveau langage) et un métamétamodèle pour les programmes (niveau application). Comme cela a déjà été évoqué, nous avons choisi de paramétrer trois notions essentielles : les relations, les descriptions et les langages. Cependant, il est nécessaire de réifier bien d'autres composants, tels les objets, les méthodes, les assertions, etc. pour modéliser complètement un langage. Le niveau OFL contient donc deux sortes d'entités : *i)* les *OFL-concepts* qui décrivent la partie paramétrable (la sémantique opérationnelle) des relations, descriptions et langages et *ii)* les *OFL-atomes* qui décrivent la partie non-paramétrable de ces trois concepts ainsi que tous les autres éléments. Ajoutons enfin que des assertions sont décrites dans chaque OFL-concept et OFL-atome pour garantir la cohérence du modèle.

La figure 4.3 montre l'intégralité de la classification des OFL-concepts. Parmi les concepts-relations, on retrouve en particulier les relations interdescriptions pour lesquelles, nous distinguons les relations d'importation (généralisation du mécanisme d'héritage) de celles d'utilisation (généralisation du mécanisme d'agrégation). OFL prend également en compte les relations entre objets et descriptions qui permettent notamment de modéliser le lien d'instanciation qui existe entre un objet et sa description. Nous pouvons aussi modéliser les relations entre objets. Cependant, notre principale préoccupation reste les relations interdescriptions.
La tâche du métaprogrammeur consiste à créer un OFL-composant, instance d'OFL-concept, en donnant une valeur à chacun de ses paramètres. Il définit par ce biais le comportement de chaque future instance de l'OFL-composant. Si les actions prévues n'offrent pas au métaprogrammeur la sémantique opérationnelle qu'il souhaite associer à un OFL-composant, il doit alors modifier le code de ces actions. Cette possibilité laisse le modèle OFL ouvert mais ne doit être utilisée que dans des contextes très spécifiques. En effet, le travail du métaprogrammeur

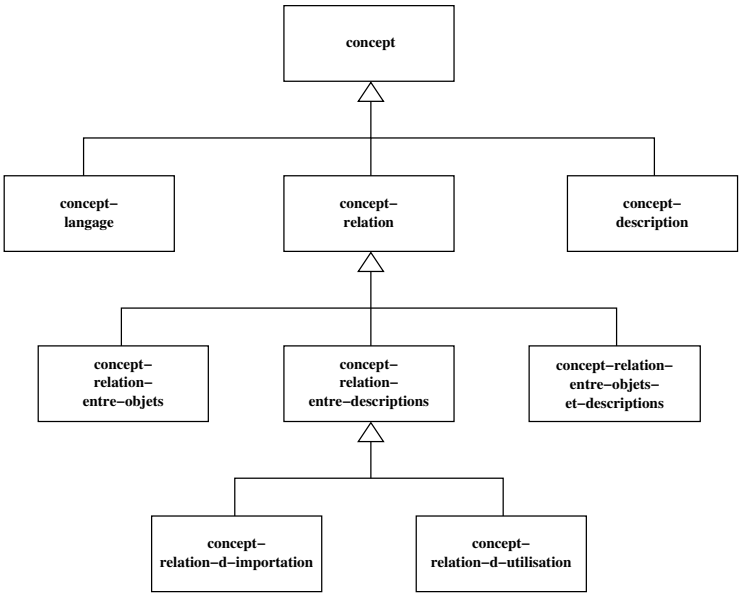

FIG. 4.3.: Les OFL-concepts

est dans ce cas beaucoup plus lourd que la simple valuation de paramètres.

À partir des informations présentes dans la figure 4.2 nous donnons quelques informations complémentaires concernant les trois types d'OFL-concepts :

- **Les *concepts-relations***. Parmi les sortes de relation présentes dans de nombreux langages à classes et méthodes de conception à objets, nous pouvons citer par exemple l'héritage, l'agrégation, la composition, la généralisation, etc. Cependant une méthode ou un langage donné possède rarement toutes ces relations et en utilise certaines pour en simuler d'autres. Par exemple, la généralisation en UML décrit aussi bien une généralisation, qu'un héritage, qu'un sous-typage strict[4]. Une trentaine de paramètres définissent la sémantique de chaque concept-relation du modèle OFL[5]. La figure 4.2 montre un exemple d'instance de concept-relation d'importation (***une-relation-de-généralisation***) et un exemple d'instance de concept-relation d'utilisation (***une-relation-d-agrégation***).
- **Les *concepts-descriptions***. Un concept-description permet de définir la notion de classe et de tout ce qui ressemble à une classe, comme les interfaces en Java. Nous pouvons remarquer, par exemple, que les classes d'Eiffel [213], de C++ [291, 292] ou de Java [143], même si elles se ressemblent, présentent des différences notables. La figure 4.2 donne, à titre d'exemple, une seule instance de concept-description appelée ***une-description***. Une vingtaine de paramètres sont nécessaires pour décrire le comportement d'une description dans le modèle OFL. Chaque description peut initier ou être la cible d'une ou plusieurs sortes de relation selon la sémantique qu'on veut lui donner ; les sortes de relations qui sont acceptées par (autrement dit, compatibles avec) un composant-description font partie des informations qui lui sont associées. Par exemple en Java, l'instance de concept-description ***interface*** peut être la cible d'une relation d'*implémentation* mais

[4]Ces trois relations ont une sémantique différente même si elles sont suffisamment proches pour être parfois confondues.

[5]À titre d'exemple, la section 5.1.1 propose plusieurs relations et en particulier celles du langage Java.

pas d'*héritage interclasses*.
- **Les *concepts-langages***. Chaque langage est constitué en particulier d'un ensemble de composants-descriptions et d'un ensemble de composants-relations, chacun étant compatible avec au moins un des composants-descriptions sélectionnés. Dans la figure 4.2, nous avons une seule instance de concept-langage (***un-langage***) qui représente le langage modélisé. Les concepts-langages sont très peu paramétrés et leur principale fonction est de fédérer des composants-relations et des composants-descriptions compatibles entre eux.

Les OFL-atomes représentent la réification des entités non paramétrées du modèle. Les relations, descriptions et langages possèdent également leur OFL-atome qui décrit la partie de leur structure et de leur comportement qui n'est pas paramétrée[6]. Sur la figure 4.2, nous pouvons noter que l'OFL-composant ***une-relation-d-agrégation*** est une spécialisation de l'OFL-atome ***relation*** qui spécifie par exemple la liste des descriptions sources et la liste des descriptions cibles. Dans une application, toutes les primitives d'une description sont instances d'un descendant de ***primitive***, toutes les expressions sont instances d'***expression*** ou d'un de ses descendants et toutes les instances de descriptions sont aussi instances d'un descendant d'***objet***. OFL offre donc une réification complète des entités présentes à l'exécution d'une application.

4.2.2. Aperçu des paramètres d'un *concept-relation*

Un important travail a été réalisé afin de dégager les paramètres que nous proposons pour décrire la sémantique opérationnelle d'une relation ou d'une classe ; nous les avons classifiés en plusieurs catégories. Même si énumérer les différents paramètres qui modélisent une relation entre classes est fastidieux pour le lecteur, il semble important d'en fournir quand même un aperçu parmi les plus significatifs, sachant qu'un nombre à peu près identique a été mis en évidence pour les classes. Une description complète et détaillée de l'ensemble de ces paramètres se trouve dans [93].

Caractéristiques générales

- *Name*. C'est simplement le nom du concept-relation.
- *Kind*. Ce paramètre est important, il définit le genre principal de la relation : *i)* importation inter-classes (`import`), *ii)* utilisation inter-classes (`use`), *iii)* entre classe(s) et objet(s) (`class-object`), *iv)* entre objets (`objects`). Une relation peut donc être une importation telle l'héritage, mais une relation peut également décrire une utilisation : la clientèle en est un exemple ou une liaison entre une classe et un objet (telle l'instanciation), voire même une liaison entre objets[7].
- *Cardinality*. Il exprime la cardinalité du concept-relation sous la forme *1-n*[8] qui signifie que les relations issues de ce concept-relation peuvent être créées entre une classe source de la relation et *1* à *n* classes cibles (*n* est ∞ ou un entier naturel supérieur ou égal à *1*).

[6] Il est donc naturel que les trois niveaux de notre architecture fassent référence directement aux OFL-atomes.
[7] Le modèle OFL se focalise sur les relations entre classes, nous ne reviendrons plus sur ces deux dernières formes.
[8] Le modèle peut supporter une cardinalité *m-n* (en vue de modéliser une association d'UML) mais les LOO considèrent toujours une relation d'héritage par rapport à une seule classe source, c'est pourquoi nous nous limitons ici à *1-n*.

Par exemple, un concept-relation d'héritage simple aura une cardinalité *1-1* alors que, pour un héritage multiple, elle sera de *1-*∞.

- *Circularity.* Il exprime la possibilité de créer des cycles avec des relations issues de ce concept-relation : `allowed` signifie que des cycles sont permis, `forbidden` qu'ils sont interdits. Par exemple, l'héritage et la clientèle non référencée interdisent les cycles alors que la clientèle les permet [209].
- *Repetition.* Il indique si une répétition de classe est permise dans les classes sources et dans les classes cibles (exemple : héritage répété) des relations issues de ce concept-relation. C'est une paire de valeurs `allowed` ou `forbidden`.
- *Symmetry.* Il indique si le concept-relation est symétrique. Un concept-relation est symétrique s'il fournit lui-même une relation sémantique symétrique. Il est possible d'imaginer ainsi un concept-relation `is-a-kind-of` pour lequel la classe-cible et la classe-source sont les termes d'une relation symétrique.
- *Opposite.* Il indique, s'il existe, le concept-relation inverse. Un concept-relation est inverse d'un autre (et *vice versa*), s'ils décrivent une sémantique inverse. Par exemple, un concept-relation de spécialisation est inverse d'un concept-relation de généralisation. Ces définitions entraînent le fait suivant : un concept-relation symétrique est son propre inverse.

Gestion du polymorphisme

- *Polymorphism_ direction.* Il est spécifique aux relations d'importation et indique si le polymorphisme (la capacité pour un objet d'être considéré selon plusieurs classes) est autorisé par le concept-relation et, si c'est le cas, dans quel(s) sens il est possible. S'il prend la valeur `up`, cela signifiera que toutes les instances de la classe source sont aussi instances de la classe cible. S'il est égal à `down`, c'est que l'on souhaite que toute instance de la classe cible soit également instance de la source. Si l'on choisit la valeur `both`, les deux comportements sont cumulés, ce qui signifie que les extensions des classes source et cible sont identiques. Enfin la valeur `none` indiquerait que les deux extensions sont indépendantes. Prenons plusieurs exemples : la *clientèle* interdit le polymorphisme (`none`), la *spécialisation* le permet dans le sens source vers cible (`up`), la *généralisation* dans le sens inverse (`down`) et nous pouvons imaginer un concept-relation de *version* qui l'autorise dans les deux sens (`both`).
- *Polymorphism.* Ce paramètre (spécifique aux relations d'importation) ne prend de sens que si *Polymorphism_ direction* est différent de `none`. Il précise si le polymorphisme sur les méthodes et sur les attributs (c'est donc une paire de valeurs) doit se faire comme un masquage (comme c'est le cas pour les attributs en Java : `hiding`) ou comme une redéfinition (cas des méthodes, toujours en Java : `overriding`).
- *Primitive_ variance.* Ce paramètre (spécifique aux relations d'importation), signale le type de variance de la relation par un triplet de valeurs, la première pour les paramètres de méthode, la seconde pour les résultats de fonction, la dernière pour les attributs. Chaque membre du triplet peut donc être :
  - `covariant` : le type des paramètres de méthode (ou résultats de fonction ou attributs)[9] de la classe source doit être identique à ou être une source (pour le même concept-relation) du type des paramètres de méthode correspondants dans la classe cible.
  - `contravariant` : le type des paramètres de méthode de la classe source doit être iden-

[9] Cette parenthèse est valable pour les autres explications.

tique à ou être une cible (pour le même concept-relation) du type des paramètres de méthode correspondants dans la classe cible.
  - `nonvariant` : le type des paramètres de méthode de la classe source doit être identique à celui des paramètres de méthode correspondants dans la classe cible.
  - `non_applicable` : aucune contrainte ne s'applique entre le type des paramètres de méthode de la classe source et celui des attributs correspondants dans la classe cible.
- *Assertion_ variance.* Ce paramètre (spécifique aux relations d'importation) signale le type de variance de la relation pour les assertions. Il est en fait composé de trois valeurs, une première pour les invariants, une deuxième pour les préconditions et une dernière pour les postconditions. Chacune des valeurs peut être : *i)* `weakened` : L'assertion de la classe source doit être identique à ou plus large que celle de la classe cible. C'est-à-dire que la première doit être impliquée par la seconde. *ii)* `strengthened` : L'assertion de la classe source doit être identique à ou plus restreinte que celle de la classe cible. C'est-à-dire que la première doit impliquer la seconde. *iii)* `unchanged` : L'assertion de la classe source doit être identique à celle de la classe cible. C'est-à-dire qu'elles doivent être équivalentes. *iv)* `non_applicable` : Aucune contrainte ne s'applique entre l'assertion de la classe source et celle de la classe cible.

Gestion de l'accès et du partage

- *Direct_ access.* Ce paramètre permet de préciser si la relation offre un accès direct aux primitives de la classe cible dans la classe source. Si un accès direct est présent pour une primitive et que cet accès ne présente aucune ambiguïté (cas de la présence d'un autre accès direct ou d'une primitive locale homonyme), alors l'utilisation de la primitive distante peut se faire comme si elle était locale. Naturellement, les relations d'importation privilégient souvent les accès directs. Les valeurs possibles pour ce paramètre sont `mandatory`, `allowed` ou `forbidden`.
- *Indirect_ access.* Ce paramètre est similaire au précédent mais pour la gestion des accès indirects. Par accès indirect, nous entendons la nécessité de nommer explicitement la classe cible ou l'une de ses instances pour accéder à la primitive. Les relations d'utilisation, par définition, ont plutôt tendance à mettre en œuvre des accès indirects.
- *Dependence.* Ce paramètre (spécifique aux relations d'utilisation) signale si les instances de la classe cible sont dépendantes (`dependent`) ou non (`independent`) de celle de la classe source. Le fait d'être dépendant signifie en particulier que l'instance de la classe cible ne peut survivre à l'instance de la classe source.
- *Sharing.* Il indique si les instances de la classe cible sont spécifiques (`specific`) à une instance de la classe source ou si elles sont partagées (`shared`) entre toutes les instances de la classe source (c'est l'idée de primitive de classe). Il est spécifique aux relations d'utilisation.
- *Read_ accessor.* Ce paramètre (spécifique aux relations d'utilisation) spécifie si l'accès en lecture aux attributs peut se faire directement (`optional`) ou s'il est nécessaire de passer par des accesseurs (`obligatory`).
- *Write_ accessor.* Ce paramètre est le pendant du précédent (spécifique aux relations d'utilisation), pour la modification des attributs : l'affectation d'un attribut est-elle possible directement (`optional`) ou exclusivement par un accesseur (`obligatory`)

Gestion de l'adaptation des primitives

- *Removing.* Ce paramètre indique si la suppression de primitive est permise (`allowed`) ou interdite (`forbidden`) au travers de cette relation.
- *Renaming.* Ce paramètre précise si le renommage de primitive est autorisé (`allowed`) ou interdit (`forbidden`) au travers de cette relation.
- *Redefining.* Ce paramètre (plutôt adapté aux relations d'importation) indique si la redéfinition de primitive est autorisée ou interdite au travers de cette relation. Au contraire des précédents, ce paramètre est multi-valué. On peut donner la valeur `allowed` ou `forbidden` pour autoriser ou interdire la redéfinition des assertions, de la signature, du corps et des qualifieurs (visibilité, protection, constance) de la primitive.
- *Hiding.* Ce paramètre est présent pour notifier la possibilité ou l'interdiction de masquer une primitive et non supprimer (cf. *Removing*), elle peut être démasquée (cf. *Showing*) par une autre relation.
- *Showing.* C'est l'inverse de *Hiding.* Il exprime s'il est possible (`allowed`) ou non (`forbidden`) de rendre de nouveau visible une primitive préalablement masquée. S'il est possible de masquer une primitive, il est intéressant de pouvoir la démasquer.
- *Abstracting.* Ce paramètre (plutôt adapté aux relations d'importation) permet de signifier s'il est possible de rendre une primitive abstraite au travers de cette relation (alors qu'elle est concrète dans la classe source).
- *Effecting.* C'est le pendant d'*Abstracting* (et donc plutôt adapté aux relations d'importation) ; il décrit l'autorisation (`allowed`) ou non (`forbidden`) de rendre concrète une primitive (qui est abstraite dans la classe source).

Il est à noter que la valeur d'un paramètre peut être liée à la valeur d'un autre. Par exemple, il nous semble normal qu'*Opposite* soit dépendant de *Symmetry* puisqu'une relation symétrique est son propre opposé. De la même façon, il existe des combinaisons de valeurs de paramètres auxquelles nous n'avons pas trouvé de sens. Cependant nous n'avons pas souhaité mettre en place un mécanisme de contrôle de la cohérence du système de paramètres pour laisser toute liberté au métaprogrammeur.

En plus des paramètres, le modèle propose des ***éléments de réification***. Parmi eux certains définissent des propriétés communes à l'ensemble des instances d'une sorte de relation, alors que d'autres représentent des propriétés propres à chaque relation :

- **Propriétés associées à un composant-relation** : *i)* les composants-descriptions valides en tant que source directe, c'est-à-dire ceux que la relation accepte en tant que source, *ii)* les composants-descriptions valides en tant que cible directe, *iii)* les modifieurs valides pour ce composant-relation et, *iv)* l'extension du composant-relation, c'est-à-dire l'ensemble de ses instances.
- **Propriétés associées à une instance de composant-relation** : Chaque relation a connaissance des ensembles de descriptions (en fait des types qui sont définis par ses descriptions) qui lui servent de sources et de cibles. De même, chaque relation décrit également la manière dont elle gère les primitives de ses descriptions cibles. Il est ainsi possible d'avoir connaissance des primitives supprimées, masquées, démasquées ou rendues abstraites ou concrètes par la relation. Pour les primitives renommées, l'ancienne signature et le nouveau nom sont conservés. Pour celles qui sont redéfinies, l'ancienne et la nouvelle primitive sont disponibles.

4.2.3. Aperçu des actions

À titre d'exemple, nous présentons l'action ***submit*** qui appartient à la catégorie 2 (exécution de primitive)[10] et dont nous résumons les principales caractéristiques :

définition. *submit* soumet un message à un objet. *submit* ne peut prendre en compte un message que si celui-ci décrit une procédure car *submit* ne gère pas de résultat.

signature. *void submit(M : Message)*
submit a pour charge de répartir le travail entre :
- *execute* (voir [93]) si la procédure décrite dans le message M doit être exécutée localement ou
- *send* (voir [93]) si cette procédure ne doit pas être exécutée localement.

paramètres hypergénériques impliqués. (pris en compte) dans une :
- description (cf. [93]) ceux de *execute* et *send*.
- relation (cf. [93]) ceux de *execute* et *send*

remarque. Tous les appels à *execute* ou *send* doivent transiter par *submit* pour fournir une interface unifiée d'exécution de tous les messages.

On notera que *i)* la signature peut comprendre aussi des préconditions qui sont basées sur la réification de l'ensemble des entités du modèle, *ii)* une action peut avoir une partie qui agit statiquement (vérification ou traitement lors de la compilation) ou dynamiquement (vérification ou traitement pendant l'exécution) et *iii)* l'algorithme associé à une action peut lui-même être découpé en plusieurs sous-algorithmes.

4.3. Une extension du modèle OFL : les *modifieurs*

Nous avons vu dans la section 4.2, qu'OFL offrait un moyen de capturer la sémantique des langages de programmation à travers des paramètres et des actions dont l'exécution est conduite par ces derniers. Cependant les paramètres proposés ne peuvent concerner que des aspects ou des fonctionnalités qui sont assez généraux pour s'appliquer à une famille significative de langages [93]. Or l'expérience montre que la sémantique proposée par chaque langage contient une partie spécifique qui reste importante qui est le plus souvent associée à des mots-clés[11]. Il est donc nécessaire de proposer une approche qui permette de la capturer. Pour cela, nous proposons d'étendre le modèle OFL à travers l'introduction d'un nouveau concept appelé ***modifieur*** ; ce travail de recherche s'inscrit dans le cadre du travail de thèse de Dan Pescaru [260]. Il s'inspire aussi de l'important travail réalisé par Gilles Ardourel dans sa thèse [13] et avec qui nous avons collaboré sur ce domaine (voir section 5.1.3), ainsi que sur d'autres travaux concernant les mécanismes de contrôle d'accès [1, 285, 62]. On constate par exemple que même UML, qui a été conçu pour être indépendant des langages, présente des lacunes en terme de protection des entités de programmes [131].

La définition de modifieurs permettra en outre : *i)* d'augmenter l'expressivité et la flexibilité du niveau méta proposé par OFL afin de permettre d'étendre facilement des langages existants avec de nouvelles fonctionnalités, *ii)* de limiter le nombre de OFL-composants quand c'est

[10] Voir au début de la section 4.2 la classification des actions.
[11] Souvent ces mots-clés sont relatifs à des règles de visibilité ou de droits d'accès par rapport à des structures du programme.

possible. En effet, une sorte de relation ou de classe (voir celles de Java dans la section 5.1.1) est définie par une combinaison de valeurs des paramètres qui lui est associée. Parfois, la valeur d'un seul paramètre diffère, nous voudrions dans ce cas pouvoir créer un modifieur qui nous évite la création de cette nouvelle sorte de classe ou de relation[12] en permettant de prendre en compte cette diffèrence de sémantique.

Survol du concept de *modifieur* Dans le but de satisfaire les objectifs mentionnés, nous proposons une approche générique pour définir les règles qui mettent en œuvre la sémantique qui doit être ajoutée aux OFL-composants. Ces règles, dans la philosphie du modèle OFL doivent permettre de satisfaire des besoins en matière de contrôle (droits d'accès, visibilité, etc.), d'assistance du programmeur et plus généralement d'environnement de programmation (collecte de métrique, rapport d'erreur, aide à la conception et à la mise au point, etc.) . Elles permettent d'ajouter des contraintes aux entités qui participent à la description d'un langage donné et donc d'améliorer ainsi la capacité d'expression du langage lui-même. Ces règles sont encapsulées dans un nouveau concept (appelé ***OFL-modifier***) qui possède entre autres les capacités de réification nécessaire, et les OFL-composants évoluent pour pouvoir intégrer des modifieurs, en fonction des besoins exprimés par leur sémantique.

Ainsi les OFL-modifiers doivent être utilisés en complément des autres entités d'un langage afin de changer leur visibilité, leurs droits d'accès ou d'autres aspects de leur sémantique. Les principales entités concernées sont les *classes*, les *relations* mais aussi les *méthodes* et les *attributs*. Comme cela a déjà été mentionné, un certain nombre de modifieurs correspondent à des mots-clés présents dans certains langages ; d'autres pourront être ajoutés dans le but d'étendre les capacités du langage ou de simplifier la tâche du programmeur.

Si nous considérons à nouveau nos travaux sur la persistance (voir section 2), cette approche permet de modéliser à un niveau méta ce que certains langages définissaient en dur. Par rapport à des approches comme celle de [272], nous nous démarquons d'un langage précis en proposant une approche générique. L'approche proposée dans LAMP [14] est plutôt dirigée vers la réalisation d'un langage dédié aux protections en se servant d'un niveau méta, que vers l'augmentation du support méta afin qu'il capture mieux la sémantique dédiée aussi bien à la visibilité et à la protection des structures des entités de programmation, qu'à l'intégration d'un nouveau service comme la persistance ou la distribution des données.

Principes de mise en œuvre La figure 4.4 décrit l'architecture du modèle OFL, comme la figure 4.2 mais elle contient en plus l'extension des OFL-Modifiers. Nous proposons quatre sortes de modifieurs (un pour chaque type d'entité concernée par la définition de modifieurs) : *description-modifier*, *relationship-modifier*, *method-modifier* et *attribute-modifier*. Un OFL-modifier est défini par :

- *un nom*. Il est unique et permet son identification ;
- *un type*. Il peut être « terminal » (il se suffit à lui-même) ou « composite » et dans ce cas, il est composé de plusieurs modifieurs terminaux.
- *un contexte*. C'est l'entité sur laquelle il s'applique ; il peut représenter : une *description*, une *relation*, un *attribut* ou une *méthode* ;

[12] C'est par exemple le cas pour le concept de classe Java "normale" ou "abstraite" ou entre une classe "privée" ou "publique" pour lesquels très peu de paramètres diffèrent (voir section 5.1.1).

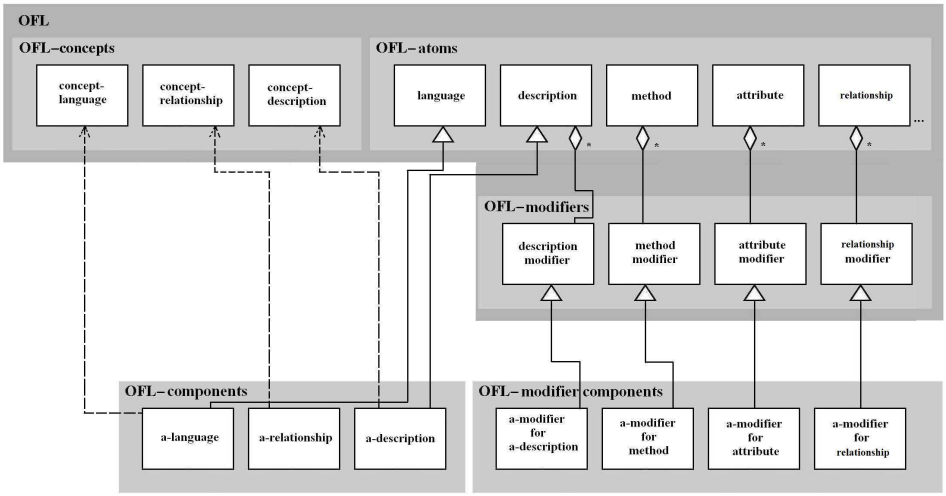

FIG. 4.4.: Architecture OFL incluant les modifieurs

- *un mot-clé* ou *une collection de modifieurs terminaux*. Si c'est un mot-clé, il représente, dans la syntaxe concrète du langage ou dans UML, la chaîne de caractères qui permet d'y faire référence. Si c'est une collection de modifieurs, ces derniers représentent la définition d'un modifieur plus complexe (par exemple le modifieur *droits d'accès d'une primitive* par opposition au modifieur *public*).
- *un ensemble d'assertions*. Nous utilisons le langage OCL [230] associé à UML pour les spécifier. Chaque assertion peut utiliser n'importe quelle propriété réifiée dans le modèle ; elle est soit associée à une action sous forme de précondition ou postcondition, soit à des OFL-composants (voir l'exemple ci-dessous).
- *un ensemble d'actions associées*. Ce sont toutes les actions qui sont touchées par l'intégration du modifieur dans le langage. Elles sont mémorisées avec leurs modifications. Ces modifications peuvent être : *i)* un ajout de précondition(s) ou de postcondition(s), *ii)* la redéfinition du corps de l'action. Ce dernier point pourrait être affiné par l'introduction d'un mécanisme de séparation des préocupations qui permette d'ajouter du comportement à des endroits différents du corps d'une action.
- *un ensemble de modifieurs incompatibles*. Aucun des modifieurs cités ne peut être associé à son *contexte* lorsque lui-même lui est associé ; par exemple un attribut *att* d'une classe Java ne pourra être à la fois *private* et *public*[13]. D'une manière générale, deux modifieurs définis sur le même *contexte* sont incompatibles si leurs actions associées ou leurs assertions ne peuvent pas être exécutées sans que leur sémantique n'interfère. Du point de vue de la mise en oeuvre, l'ensemble des modifieurs incompatibles peut exister physiquement et être garanti par une assertion globale au langage, mais dans ce cas, il force un modifieur à connaître les autres modifieurs, ou il peut être représenté par une métaassertion dans le *contexte* lui-même.

Exemple de modifieur À titre d'exemple, prenons le cas très classique d'un modifieur *composite* appelé *JavaAccessRights* qui définit les protections d'une primitive en Java qui sont

[13]Voir l'exemple de modifieur qui est présenté ci-dessous.

représentés par les modifieurs *terminaux* : *public*, *protected*, *private*, et les droits par défaut (nous prendrons *package* pour simplifier)[14]. La réification d'un attribut est représentée par l'entité *AtomAttribute*. Les modifieurs *protected*, *private*, *package* et *public* sont incompatibles entre eux, ce qui est traduit par l'invariant suivant :

```
context AtomAttribute
inv : self.modifiers->includes('public') XOR  self.modifiers->includes('private') XOR
      self.modifiers->includes('package') XOR self.modifiers->includes('protected')
```

Pour définir les protections relatives à une primitive Java, il faut considérer trois situations : *i)* la primitive appartient à la classe courante et elle est utilisée par une autre de ses primitives en utilisant *this* ou pas, *ii)* la primitive est héritée à travers une relation d'héritage et appartient à un ancêtre direct ou non, *iii)* la primitive est accédée à travers une relation d'agrégation par une classe cliente de celle qui contient la primitive. Dans le premier cas, il n'y a rien à faire puisqu'une primitive ne se protège pas contre la classe qui l'a définie. Pour les deux autres, la protection est mise en œuvre à travers des invariants dans les OFL-composants qui décrivent la relation d'héritage entre classes[15] et l'agrégation, respectivement *ComponentJavaExtends* et *ComponentJavaExtends.*

```
context ComponentJavaExtends
inv : self.showedFeatures->forall(f :Feature |
            f.modifiers->includes('public') OR f.modifiers->includes('protected'))
inv : self.redefinedFeatures->forall(f :Feature |
            f.modifiers->includes('public') OR f.modifiers->includes('protected'))
inv : self.hiddenFeatures->forall(f :Feature | f.modifiers->includes('private'))
```

Cet invariant est relatif à la relation d'héritage *ComponentJavaExtends*. Il indique que toutes les primitives visibles (*showed*Features) et redéfinies (*redefinedFeatures*) à travers la relation modélisée par le mot-clé Java *extends* (*ComponentJavaExtends*) doivent être associées à l'un des modifieurs suivants : *public* ou *protected*. Toutes les primitives cachées (*hiddenFeatures*) doivent être associées au modifieur *private*.

```
context ComponentJavaAggregation
inv : self.showedFeatures->forall(f :Feature |
           f.modifiers->includes('public')  OR
           (( f.modifiers->includes('package') OR f.modifiers->includes('protected'))
             AND self.source.package = self.target.package)))
inv : self.hiddenFeatures->forall(f :Feature |
           f.modifiers->includes('private') OR
           (( f.modifiers->includes('package') OR f.modifiers->includes('protected'))
             AND self.source.package <> self.target.package)))
```

Les assertions ci-dessus concernent la relation d'agrégation *ComponentJavaAggregation*. Pour ce type de relation, il faut aussi considérer le paquetage (*package*) des descriptions représentant la source (*source*) et la cible (*target*) de la relation d'agrégation représentée par *self*[16]. La source est celle qui déclare le mot-clé *extends* en Java tandis que la cible est mentionnée après ce même mot-clé. On notera pour finir que les différents invariants qui mettent en œuvre le modifieur *JavaAccessRights* doivent être vérifiés chaque fois qu'une relation qui concerne une primitive associée au modifieur est créée.

[14] D'autres exemples plus recherchés pourront être trouvés dans [190, 260].

[15] Nous simplifions le problème en ne considérant que les classes et en laissant de côté les interfaces.

[16] Afin de ne pas compliquer inutilement les assertions de cet exemple, nous ne prenons pas en compte le cas où une classe décrit un attribut dont le type est le même que la classe parente.

5. Applications de la modélisation des concepts objets

En complément du travail de modélisation mené dans le cadre des thèses de Pierre Crescenzo pour ce qui concerne le cœur du modèle [93], d'Adeline Capouillez pour la mise en œuvre d'un protocole d'extension basé sur la séparation des préoccupations [58], et de Dan Pescaru pour l'introduction de nouvelles capacités de description [260, 96], nous avons initié plusieurs travaux qui sont terminés ou qui ont déjà produit des résultats. Dans ce chapitre, nous en proposons un rapide descriptif.

Nous nous intéressons d'abord à montrer l'expressivité du modèle OFL pour capturer la sémantique des classifieurs et des relations qui les lient. Pour cela, nous décrivons ou discutons la ou les sémantiques possibles. Après cette phase d'expérimentation, nous décrivons deux utilisations possibles du modèle. La première vise à améliorer l'enchaînement entre la phase de conception et la phase de programmation [97], tandis que l'autre s'intéresse à la description d'un langage dédié aux protections.

Dans un deuxième temps, nous proposons de montrer comment utiliser la réflexion que nous avons menée autour des concepts objets et de l'héritage en particulier pour faciliter la maintenance, l'évolution et la réutilisation du logiciel. Pour cela, nous avons d'une part défini une approche pour annoter les relations d'héritage avec pour objectif de les utiliser par exemple dans un environnement de programmation [162, 94] et d'autre part, entamé la définition d'une relation d'héritage inverse dédiée à l'amélioration de l'adaptabilité des classes en vue d'une meilleure réutilisation [75, 74, 76].

5.1. Application à la description des langages

Nous proposons d'abord (section 5.1.1) de montrer des exemples de sémantique de relation et de classe, décrits à l'aide du formalisme proposé par OFL. Ils permettront au lecteur de se faire une opinion sur l'apport du travail réalisé. Il trouvera entre autres *i)* des éléments de description d'une relation de réutilisation et d'une relation de point de vue, *ii)* un aperçu des différentes sortes de classes et relations qui sont présentes dans le langage Java.

Dans un deuxième temps, nous décrivons dans la section 5.1.2 une approche pour générer, à partir des métainformations d'OFL, des profils UML. L'objectif visé est d'adapter le comportement d'UML pour un langage donné.

Dans un dernier temps (section 5.1.3), nous montrons comment il est possible d'exploiter l'approche d'OFL pour spécifier un langage « universel » dédié à la description des protections pour les entités d'un langage de programmation.

5.1.1. Relations et classifieurs dans les langages

Au cours de nos différents travaux, nous avons décrit des classifieurs et des relations de langages connus comme Eiffel ou Java [92, 59] mais aussi des exemples de relations aussi

```
class Person {
  integer yearOfBirth;
  integer age() {
    return currentYear() - yearOfBirth;
   }
 }

class Car
    reuses Person (
       // Reuse of yearOfBirth which is renamed in yearOfFirstRegistration
      yearOfBirth -> yearOfFirstRegistration,
       // Reuse of age() without any renaming
      age()
     )
  {
   ...
   Color externalColor;
   ...
  }
```

FIG. 5.1.: Exemple de relation de réutilisation

bien traditionnelles comme la spécialisation, ou l'implémentation [93], que moins courantes telle une relation de réutilisation ou de point de vue [57] ou encore une généralisation [56]. Parmi les buts recherchés, nous pouvons citer *i)* l'obtention d'un code plus précis et donc plus lisible, *ii)* la posssibilité d'activer des contrôles et des actions automatiques qui facilitent la maintenance des applications. Nous présentons successivement les principaux éléments d'une relation de réutilisation de code, une discussion sur le choix des paramètres à associer à une relation de point de vue en fonction de l'usage que l'on en attend et, un résumé des différentes sortes de relations et de classifieurs mis en œuvre dans Java.

Aperçu d'une relation de réutilisation L'héritage est souvent utilisé pour mettre en œuvre une spécialisation. Pourtant, combien de fois n'en avons nous pas fait usage à d'autres fins, par exemple dans le but de réutiliser du code. C'est une amélioration particulièrement significative par rapport au copier/coller du code car elle ouvre la voie à une réutilisation accrue. Cependant, utiliser l'héritage dans ce but, c'est notamment : *i)* laisser la place au polymorphisme là où il n'a pas de raison d'être, *ii)* ne pas pouvoir choisir les méthodes à réutiliser mais devoir toutes les importer, *iii)* être contraint par les règles d'adaptation des primitives (renommage, redéfinition, etc.) qui sont souvent une gêne dans ce contexte.

Pour mieux comprendre ce que pourrait donner une capacité de réutilisation de code, imaginons les classes *Person*, *Car* de la figure 5.1[1]. Nous voulons réutiliser des primitives de la classe *Person*, notamment la méthode *age* dans la classe *Car*. L'extension proposée dans la figure 5.1 permet d'éviter la duplication de *age* tout en renommant l'attribut *yearOfBirth*. Pour préciser un peu la sémantique associée au mot-clé reuses, on peut examiner le tableau 5.1[2] qui révèle en particulier que : *i)* l'on peut réutiliser autant de classes que l'on veut (relation multiple), *ii)* la suppression et le renommage de primitive sont autorisés, *iii)* deux classes peuvent réciproquement réutiliser des primitives distinctes leur appartenant et surtout, *iv)* que la relation n'est pas polymorphique (sans quoi *ii)* ne serait pas possible).

[1] Dans un souci de simplicité, nous utilisons ici une syntaxe *à la Java* enrichie du mot-clé *code-reuses*.
[2] Pour plus d'information concernant la signification des différents paramètres, voir la section 4.2.2.

Paramètres	Valeurs requises
Name	code-reuse
Cardinality	1-∞
Repetition	<ignored, allowed>
Circularity	true
Opposite	none
Polymorphism_direction	none
Removing	allowed
Renaming	allowed
Redefining	<mandatory, allowed, forbidden, allowed>

TAB. 5.1.: Quelques-uns des principaux paramètres d'une relation de réutilisation de code

Il existe d'autres approches qui sont dédiées à la réutilisation de code. À titre d'exemple, on citera les approches basées sur les *traits* [270, 271]. Un *trait* ne représente pas un type mais un module et il est vu comme l'entité de réutilisation. Contrairement à notre approche qui favorise la réutilisation *a posteriori*, un *trait* est défini *a priori* et il spécifie les méthodes qu'il fournit et celles qui sont utilisées par les premières. En fonction de ses besoins, une classe s'associe à un ou plusieurs *traits*.

Discussion autour de la relation de point de vue La relation *Is-a-view-of* définit des vues sur les instances d'une classe, dans un sens proche des vues en base de données. Une vue permet de présenter des données sous une forme différente de celle proposée par leur classe d'origine. On peut par exemple cacher certaines parties des objets ou en présenter d'autres sous un nouvel angle. Nous discutons ici l'influence de la valeur des paramètres de *concept-relation* sur l'usage du lien.

Une relation de vue peut être considérée de deux manières, soit comme une relation d'utilisation, soit comme une relation d'importation. Le paramètre *Name* aura alors comme valeur la chaîne de caractères *« Is-a-**u**view-of »*, pour l'utilisation, et *« Is-a-**i**view-of »* pour l'importation. De même, le paramètre *Kind* a la valeur *use* pour le premier tandis qu'il prend la valeur *import* pour le second. Chacune des deux sortes de relation implique des paramètres spécifiques qui mettent en évidence des différences de sémantique et d'utilisation (voir figure 5.2).

Pour expliquer l'intérêt de *Direct_access* et *Indirect_access* (voir section 4.2.2), nous proposons des valeurs différentes selon le *concept-relation* de vue qui est considéré. Nous rappelons que le choix d'une relation d'utilisation a pour effet d'introduire dans la classe source une référence à une instance de la classe cible (par exemple à travers un attribut), ce qui n'est pas le cas dans une relation d'importation. Positionner le paramètre *Direct_access* du *concept-relation Is-a-**u**view-of* à *allowed* est plutôt inhabituel pour une relation d'utilisation. La procédure *test1* de la classe *GARAGE* décrite dans la figure 5.3 montre l'intérêt d'une telle démarche avec le deuxième appel à *print*. De manière plus traditionnelle, nous proposons de positionner, *Direct_access* à *allowed* et *Indirect_access* à *forbidden* pour le *concept-relation Is-a-**i**view-of* et *Indirect_access* à *allowed* dans le *concept-relation Is-a-**u**view-of*.

Si la valeur de *Cardinality* indique une relation multiple, notamment dans le cas d'une relation d'importation (*Is-a-**i**view-of*), cela permettrait entre autres de simuler l'opération

class CAR-VIEW-USE	class CAR-VIEW-IMPORT	class GARAGE

aCar : Is-a-uview-of CAR
rename
maximal-price as price
remove cost-price
end Is-a-uview-of
 	
function promotional-price : REAL {
 	
return 0.9 * price()
 	
}
 	
end class
	Is-a-iview-of CAR
	rename
	maximal-price as car-price
	remove cost-price
	end Is-a-iview-of
	
	function promotional-price : REAL {
	
	return 0.9 * car-price()
	
	}
	
	end class
		car1 : client-of CAR-VIEW-USE
		
		car2 : client-of CAR-VIEW-IMPORT
		
		oneCar : client-of CAR
		
		procedure test1() { }
		procedure test2() { }
		
		end class

FIG. 5.2.: Exemple de relation exprimant un point de vue

```
procedure test1 {
  print(car1.aCar.maximal-price) ;
   // appel de la primitive maximal-price de l'attribut aCar de l'objet car1
  print(car1.price) ;
   // appel de la primitive price de l'objet car1 :
   // intérêt du paramètre Direct_access qui permet d'appeler maximal-price
   // avec le nom price et sans passer par aCar
  print(car2.car-price) ;
   // maximal-price a été importé sous le nom car-price
   // (effet similaire à l'héritage avec renommage)
}
```

FIG. 5.3.: Illustration du paramètre *Direct_ access*

```
procedure test2 {
  oneCar <- car2 ; // non valide : le polymorphisme est descendant
  car2 <- oneCar ; // valide
    // l'instance de CAR est vue comme une CAR-VIEW-IMPORT
  print(car2.car-price) ; print(oneCar.maximal-price) ;
    // la primitive maximal-price a été renommée dans CAR-VIEW-IMPORT
  print(oneCar.cost-price) ; print(car2.cost-price) ;
   // seul le premier appel est valide : cost-price a été supprimé
   // dans CAR-VIEW-IMPORT
}
```

FIG. 5.4.: Illustration du sens du polymorphisme

de jointure des bases de données relationnelles. Par contre, il faudrait alors gérer tous les problèmes habituels des relations multiples, tels que les conflits de noms.

Pour le *concept-relation Is-a-**i**view-of* et pour lui seulement, le paramètre *Polymorphism-_direction* (voir section 4.2.2) a un sens. Indépendamment de la sémantique qu'on veut donner à la vue, si celle-ci ne fait que supprimer des primitives, il est alors acceptable de proposer la valeur *down*. Si on ne fait qu'en ajouter, *up* est plus adapté. Si on enlève et n'ajoute rien, alors *both* est raisonnable (ce dernier cas n'a souvent d'intérêt que dans une relation multiple). Dans le contexte des applications de base de données relationnelles, il n'est pas possible d'ajouter un attribut dans une vue. On peut par contre en supprimer, donc la valeur la plus raisonnable dans ce contexte serait *down*. L'usage de l'héritage pour simuler une telle relation présenterait l'inconvénient d'avoir un polymorphisme *up*, ce qui est inadapté pour une relation de vue. Ainsi, la procédure *test2* de la classe *GARAGE* décrite dans la figure 5.4 décrit un certain nombre d'expressions valides et invalides. Dans le cas d'un *concept-relation* comme *is-a-**u**view-of*, il est raisonnable que le paramètre *Dependence* soit fixé à *independent* car il n'y a naturellement aucune raison à ce que l'instance de *CAR* disparaisse dans le cas où sa vue est supprimée.

Une clause d'adaptation est un mécanisme permettant de modifier (redéfinir, renommer, etc.) une primitive importée. Dans OFL, nous modélisons, toujours à l'aide de paramètres, le fait que l'on peut ou non faire usage de ces clauses.

Ces dernières représentent un aspect important de la définition d'un *concept-relation* de vue, comme ceux présentés dans la figure 5.2. D'elles vont dépendre la valeur d'un certain nombre de paramètres mentionnés ci-dessus. De plus, la sélection des clauses d'adaptation valides influence directement l'usage de ces relations. Par exemple, si on souhaite construire des vues qui représentent des projections (au sens des bases de données relationnelles), on va naturellement autoriser la suppression de primitives et le paramètre *Removing* aura la valeur *allowed*. De même, autoriser le renommage des primitives permet d'adapter au plus juste la signification de la primitive ou de résoudre les conflits. Les paramètres *Hiding* et *Showing* sont particulièrement intéressants lorsqu'on introduit la possibilité d'avoir des vues de vues et donc d'avoir un arbre de relations *Is-a-**i**view-of* ou *Is-a-**u**view-of*.

Pour finir, l'abstraction, la réalisation ou la redéfinition de primitive (paramètres *Redefining*, *Abstracting* et *Effecting*) n'ont *a priori* aucune raison d'être autorisés.

Représentation OFL de Java Nous avons voulu utiliser l'expressivité d'OFL pour décrire la sémantique des différentes sortes de classe (*description* en OFL) et de relation présentes dans le langage Java. Chacune de ces sémantiques est représentée par un OFL-composant. Le lecteur peut se référer à la figure 5.5 pour avoir la liste complète des OFL-composants de Java.

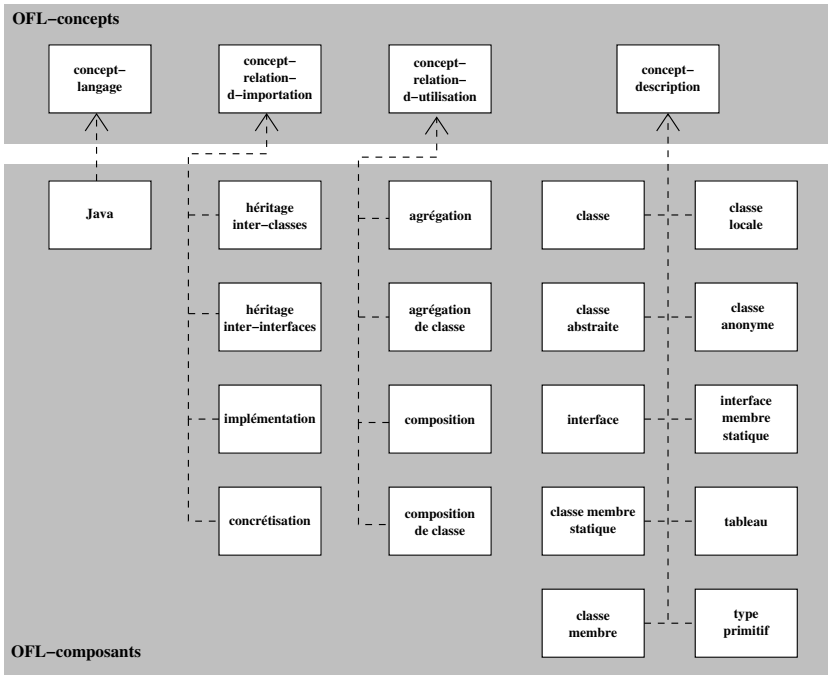

FIG. 5.5.: Les OFL-composants de Java

Nous avons ainsi dénombré pour Java : *i)* huit composants-relations où quatre sont d'importation et quatre d'utilisation et *ii)* dix composants-descriptions. Le nombre d'OFL-composants peut sembler élevé au programmeur Java. Il est dû à la précision de notre système de paramètres qui offre une granularité relativement fine[3]. Les différences sémantiques entre relations ou descriptions sont souvent masquées au programmeur par l'usage d'un même mot-clé dans un contexte différent. La présentation que nous donnons des OFL-composants de Java ne donne pas la valeur de chacun des paramètres mais plutôt une présentation de leurs principales caractéristiques. Nous signalons entre parenthèses les mots-clés associés à chaque OFL-composants. Les quatre premiers composants-relations sont des importations, les quatre suivants des utilisations :

- L'héritage inter-classe (*extends*). Cette relation est utilisée pour affiner l'implémentation de la spécification d'un type de données. L'implémentation d'une spécification est réalisée dans une classe. L'héritage inter-classe spécialise donc une classe. Il s'agit d'un héritage simple pour lequel les cycles sont interdits. Les primitives de la classe héritée sont importées dans la classe héritière. Il est possible de remplacer les attributs et de redéfinir les méthodes. Le polymorphisme s'applique de manière ascendante, c'est-à-dire que toute instance de l'héritière peut être vue comme instance de l'héritée.
- L'héritage inter-interface (*extends*). Cette relation permet d'affiner la spécification d'un type de données. Elle est posée entre interfaces, l'héritière spécialisant les héritées. Au contraire de l'héritage inter-classes, cette relation est en effet multiple. Le polymorphisme fonctionne de manière similaire à l'héritage inter-classe.
- La concrétisation (*extends*). Cette relation permet de concrétiser l'implémentation de la

[3]Lorsque les différences sont minimes, on préférera l'utilisation de modifieurs (voir section 4.3).

spécification d'un type de données. Elle est posée entre une classe abstraite (héritée) et une classe non abstraite (héritière). Cette relation est identique à l'héritage inter-classe si ce n'est qu'elle impose de donner, dans l'héritière, un corps aux méthodes abstraites de l'héritée et éventuellement à celles de ses superclasses lorsque celles-ci n'en possèdent toujours pas.
- L'implémentation (*implements*). Cette relation modélise l'implémentation de la spécification d'un type de données. Une classe peut ainsi implémenter une ou plusieurs interfaces. L'implémentation est donc une relation multiple. Si la classe est concrète, elle doit donner un corps à toutes les méthodes spécifiées dans les interfaces. Si elle est abstraite, elle peut donner un corps à certaines méthodes et conserver les autres abstraites. Le polymorphisme est identique à celui des relations d'héritage.
- L'agrégation. Cette relation modélise l'utilisation des services d'une description. Pour réaliser une telle utilisation, il suffit de déclarer et d'initialiser un attribut du type de la description utilisée[4]. Contrairement aux quatre relations d'importation précédentes, les cycles sont autorisés pour l'agrégation. L'accès aux attributs de la description utilisée est direct, c'est-à-dire que la présence d'accesseurs n'est en rien obligatoire[5]. La durée de vie de l'objet utilisé est indépendante de celle de l'objet utilisateur et le même objet peut être utilisé par plusieurs objets utilisateurs.
- L'agrégation de classe (*static*). Cette relation modélise la notion bien connue dans les langages à objets d'*attribut de classe*. Elle est identique à l'agrégation si ce n'est le fait que l'objet utilisé est associé à la classe utilisatrice et non à ses instances.
- La composition. Cette relation modélise l'utilisation forte des services d'une description. Dans la littérature, on emploie le terme *composition* entre deux classes au lieu d'*agrégation* pour dire qu'une instance d'une des deux classes est incluse dans l'instance qui l'utilise, et donc sa durée de vie est dépendante de celle de l'objet qui la contient/l'utilise. En Java, cette notion de composition ne s'applique qu'aux descriptions qui utilisent un type primitif (par exemple : *int*).
- La composition de classe (*static*). Elle est similaire à la composition mais définit un *attribut de classe* comme le fait l'agrégation de classe.

Nous donnons maintenant un aperçu des *composants-descriptions* de Java. De manière générale, nous nommons *classes internes* les classes membres statiques ainsi que les classes membres, les classes locales, les classes anonymes et les interfaces membres statiques. Nous avons recensé dix composants-descriptions[6] :
- La classe (*class*). La classe est une implémentation concrète d'un type de données. C'est une description non générique qui peut contenir des méthodes[7] et des attributs. Elle est visible au sein de son paquetage mais cette visibilité peut être étendue ou restreinte par un qualifieur (*public* ou *private* par exemple). Elle a la capacité de créer des instances mais pas de les détruire explicitement. Enfin, la classe autorise la surcharge sans prendre en compte le type du retour des fonctions.
- La classe abstraite (*abstract class*). La classe abstraite est une implémentation abstraite d'un type de données. Cette description possède les mêmes propriétés qu'une classe mais elle peut décrire des méthodes abstraites (sans corps) et ne peut pas posséder d'instance

[4] L'usage d'un paramètre de méthode, d'un résultat de fonction ou d'une variable locale du type de la description utilisée s'apparente à une agrégation.
[5] Sauf si la visibilité de l'attribut est restreinte, par exemple en le déclarant *private*.
[6] Nous ne tenons pas compte des classes internes abstraites dans ce document.
[7] Ainsi que des constructeurs, initialiseurs et destructeurs.

propre.
- L'interface (*interface*). Il s'agit de la spécification d'un type de données. Au contraire d'une classe, une interface ne peut pas définir d'attribut (sauf les *constantes de classe*). De plus, ses méthodes sont toutes abstraites et elle ne peut donc pas créer d'instance.
- La classe membre statique (*static class*). C'est une implémentation, locale à une classe, d'un type de données. Sa particularité, par rapport à une classe, est d'être définie à l'intérieur d'une classe et non au plus haut niveau. Elle n'est d'ailleurs accessible qu'au travers de sa classe encapsulante.
- La classe membre (*class*). Il s'agit également d'une implémentation, locale à une classe, d'un type de données. Mais, à la différence de la classe membre statique, une instance de la classe membre est automatiquement associée à chaque instance de la classe encapsulante.
- La classe locale (*class*). Elle représente une implémentation, locale à une méthode, d'un type de données. Elle n'est visible qu'à l'intérieur de sa méthode encapsulante. En dehors de cela, elle est équivalente aux autres *composants-descriptions* de classe.
- La classe anonyme (*class*). La classe anonyme est une implémentation, locale à une expression, d'un type de données. Elle est équivalente à une classe locale mais n'est visible qu'au sein de son expression encapsulante mais n'ayant pas de nom, il n'est pas possible de la référencer et donc d'en hériter. De par sa structure syntaxique, si elle implémente une interface, elle ne peut en implémenter qu'une. De plus, si elle hérite directement d'une classe, elle ne peut pas implémenter directement une interface.
- L'interface membre statique (*static interface*). Il s'agit d'une spécification, locale à une classe, d'un type de données. C'est l'équivalent de la classe membre statique sous la forme d'une interface.
- Le tableau. Il représente la structure de données bien connue. C'est donc une collection indexée et de taille fixe d'entités d'un type défini. Le tableau est un cas particulier en Java. Chaque tableau est une instance d'une classe virtuelle[8] représentant son type (exemple : un tableau d'entiers est de type *int[]*).
- Le type primitif. Le type primitif est la représentation d'un type de base du langage. Il permet de décrire les éléments essentiels des applications : booléens, caractères, octets, entiers courts, entiers, entiers longs, flottants et flottants doubles. Remarquons que chaque type primitif décrit une valeur et non un objet mais qu'une classe existe pour représenter chacun d'eux. Par exemple, la classe *Integer* permet de considérer un *int* comme un objet.

En ce qui concerne les composants-relations d'importation nous pouvons, par exemple, signaler les contraintes suivantes :

- L'héritage inter-classe ne peut avoir pour cible ou pour source ni une interface ni une interface membre statique. De plus, les classes anonymes ne peuvent être cibles d'un tel héritage.
- L'héritage inter-interface est posé entre interfaces (qu'elles soient membres statiques ou pas).
- La concrétisation a forcément lieu entre une classe abstraite et une classe non abstraite.
- Enfin, l'implémentation est une relation qui a toujours comme cible une ou plusieurs interfaces (quelles qu'elles soient) et comme source une classe (quelle qu'elle soit). Le même genre de contrainte s'applique aux *composants-relations* d'utilisation.

[8] Cette classe n'existe pas mais tout se passe comme si elle existait vraiment.

5.1.2. Génération de profils UML

Nous rappelons que l'un des objectifs que nous avions lorsque nous avons entrepris la spécification du modèle OFL était de réduire le fossé qui existe entre les langages et les méthodes de conception. Il est donc naturel de vouloir proposer une passerelle entre le modèle OFL et UML [97] et de produire les profils UML à partir d'une description OFL de la sémantique. L'intérêt est double. D'une part, on veut pouvoir décrire précisément la sémantique opérationnelle des différents types de classifieur ou de relations associés à un langage à objets pour ensuite les utiliser pendant la phase de conception. D'autre part il s'agit de définir de manière précise dans UML des concepts de classes ou de relations issus des langages de programmation existants et ainsi, de faciliter la projection de schémas de conception vers les langages de programmation grâce aux informations mémorisés pour chacun des concepts. Du point de vue de la mise en œuvre, nous désirons réutiliser l'éditeur OFL déjà implémenté [274] pour générer des profils UML.

Aperçu du métamodèle et des profils UML UML propose un métamodèle pour décrire l'ensemble de ces entitiés ; les éléments de ce métamodèle sont mémorisés dans plusieurs paquetages. Nous nous intéressons plus particulièrement aux paquetages UML standard appelés *Core* et *Model Management*[9]. On notera qu'en particulier *Core* contient lui-même deux sous-paquetages : le premier représente les éléments qui forment la colonne vertébrale structurelle du métamodèle (espace de noms, type de données, classifieur, primitive, contrainte, etc.) tandis que le second montre des éléments qui participent à la définition des relations. À titre d'exemple, la figure 5.6 donne une description des entités de ce dernier paquetage.

Les profils d'UML [235] fournissent un mécanisme générique pour construire des applications ou des modèles adressant des domaines spécifiques (voir figure 5.7). Pour plus d'informations à propos de la méthodologie proposée pour réaliser des profils qui adaptent le modèle UML aux besoins d'un domaine spécifique, il pourra être intéressant de lire [286]. Il faut cependant constater que même si des profils commencent à apparaître, le fait d'utiliser un tel mécanisme reste controversé [114, 19]. Ce mécanisme est basé sur l'adjonction de *stereotypes*, *tagged values*, *elements*, *attributes*, *methods*, *links*, *assertions*, etc. On rappelle ci-dessous les principales entités qui participent à la définition d'un profil :

- Un profil (*Profile*) est un paquetage auquel on a associé un stéréotype et qui contient tous les éléments du modèle qui ont été adaptés à un domaine spécifique en utilisant des stéréotypes, des définitions étiquetées, et des contraintes.
- Un stéréotype (*Stereotype*) est un élément du modèle qui définit de nouvelles valeurs (basées sur la définition d'étiquettes), de nouvelles contraintes et éventuellement d'une nouvelle représentation graphique. Tous les éléments d'un modèle qui sont associés à un stéréotype « héritent » de ces valeurs et contraintes en supplément des attributs, associations, et superclasses que l'élément a dans UML non étendu.
- La définition d'étiquettes (*Tag definition*) spécifie de nouvelles sortes de propriétés qui peuvent être attachées à des éléments du modèle. Les propriétés concrètes des éléments individuels du modèle sont spécifiées en utilisant des valeurs étiquetées (*Tagged values*). Elles peuvent être soit des valeurs d'un type simple ou des références à d'autres éléments du modèle. La définition d'étiquettes peut se comparer à la définition de méta-attributs tandis que les valeurs étiquetées correspondent à des valeurs attachées aux éléments du modèle.

[9] Ce travail s'appuie sur la version 1.5 d'UML [235].

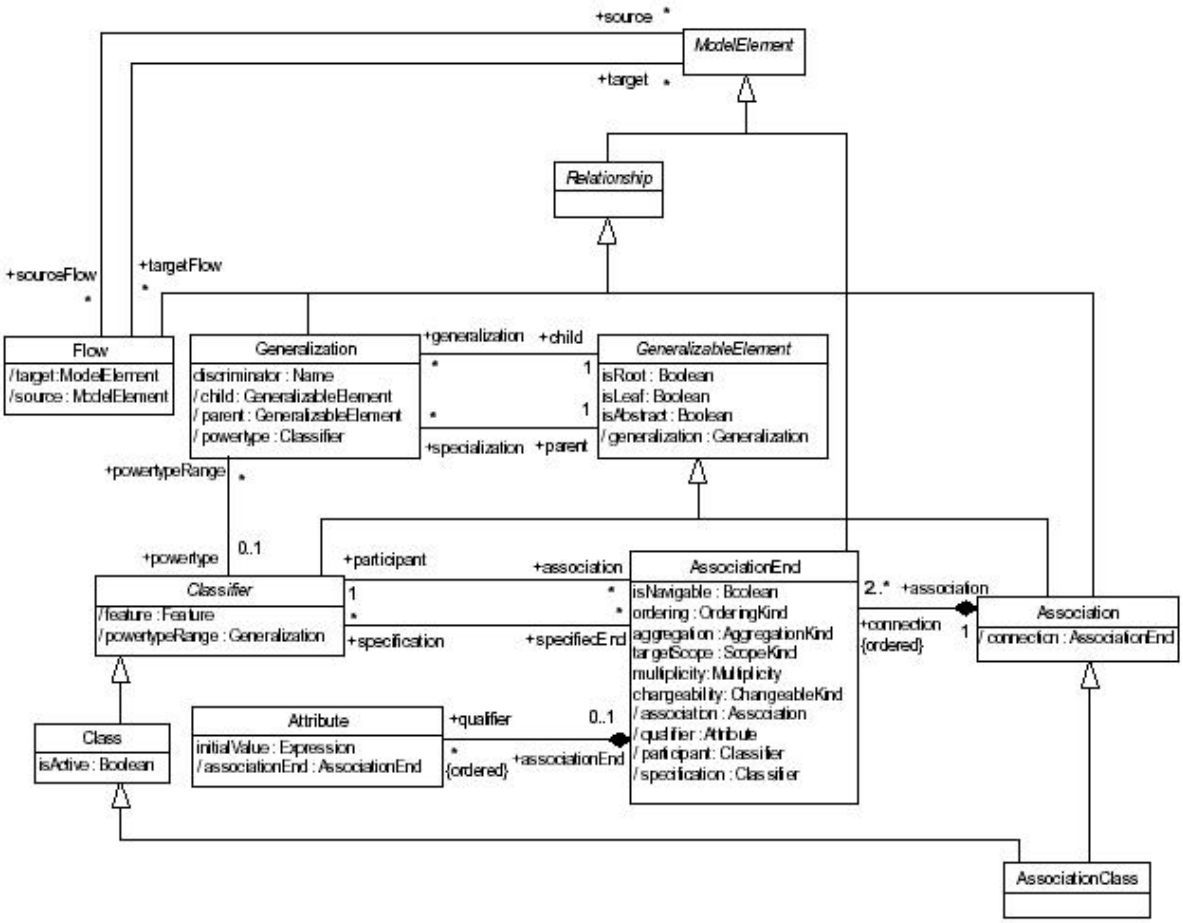

FIG. 5.6.: Le paquetage *Core* - *Relationships* d'UML

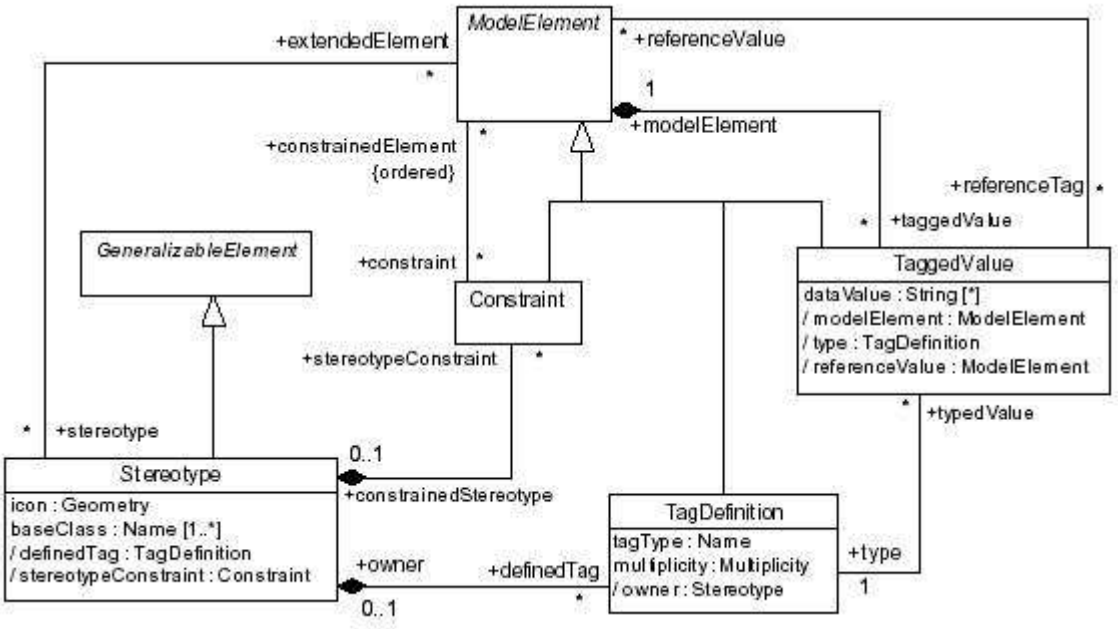

FIG. 5.7.: Le paquetage *Extension mechanisms* d'UML

– Des contraintes (*Constraints*) peuvent aussi être attachées à n'importe quel élément du modèle pour affiner sa sémantique. Elles sont généralement décrites en OCL. Les contraintes attachées à un stéréotype doivent être satisfaites par tous les éléments du modèle qui sont associés à ce stéréotype.

Survol de l'approche L'approche que nous avons développée consiste à générer des profils UML à partir des informations recueillies par la description d'entités du modèle OFL, c'est-à-dire de spécifier un métaprofil ; nous l'appelons OFL-ML. Il s'appuie sur la définition de diagrammes de classes d'UML[10] et notamment sur une extension des paquetages UML standard appelés *Core* et *Model Management* (voir ci-dessus). On notera que pour exprimer les règles d'accès aux classifieurs et aux primitives, UML utilise les marqueurs standard de visibilité (correspondant à *private*, *public*, etc.)[11].

Un des objectifs importants d'OFL-ML est de générer automatiquement un profil propre et lisible. Les règles suivantes sont destinées à favoriser cet objectif : *i)* chaque *OFL-composant* est représenté par son propre stéreotype, *ii)* chaque combinaison de valeurs associées à un élément OFL qui n'est pas paramétré (c'est-à-dire réifié par un OFL-*atom*) produit un stéréotype différent, *iii)* des OFL-éléments supplémentaires (comme les *OFL-modifieurs* ou les *OFL-assertions*) sont associés à des valeurs étiquetées ou à des contraintes qui sont créées dans le profil généré.

[10] C'est-à-dire sur des graphes de classifeurs connectés entre eux par des relations.
[11] Ces marqueurs n'ont pas de sens pour OFL-ML qui propose une définition plus précise de ces droits d'accès, à travers la définition de modifieurs (voir section 4.3).

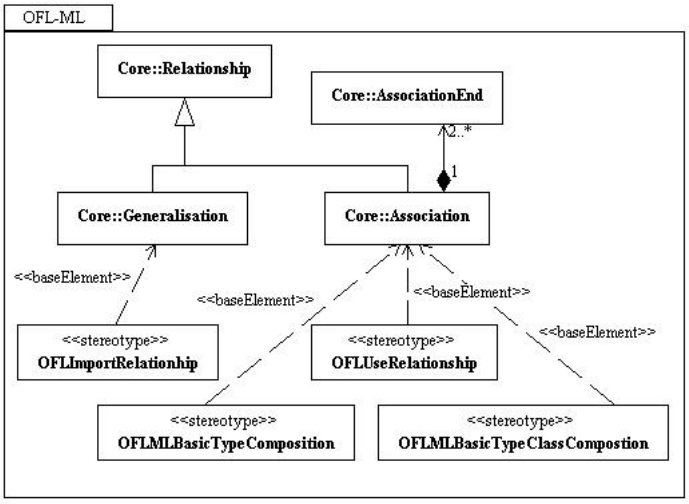

FIG. 5.8.: OFL-ML : Les relations

Typiquement les stéréotypes pourront être exploités pour d'une part générer une réification OFL qui correspond à une application et d'autre part, l'initialiser avec les traitements et les informations adéquates pour répondre à un besoin particulier. L'application ainsi équipée pourra être utilisée par exemple pour réaliser des métriques, des tests [163] ou pour générer un code adéquat pour un langage donné [286].

Aperçu de la génération de profils La définition d'un profil UML nécessite la définition d'un métamodèle virtuel. Ce dernier correspond à un modèle formel associé à un ensemble d'extensions d'UML lui-même exprimé en UML. Le métamodèle virtuel d'OFL-ML est présenté dans [97] comme un ensemble de diagrammes de classes, cependant on trouvera des informations complémentaires dans [237, 239, 238]. À titre d'exemple, la figure 5.8 présente le métamodèle virtuel associé aux principaux *concepts-relations* définissant des relations entre classes (voir figure 4.3). Nous donnons un aperçu des résultats obtenus à travers la description des stéréotypes, valeurs étiquetées, contraintes et éléments du modèle qu'il faut introduire dans OFL-ML pour supporter le concept OFL dédié aux relations d'héritage (*OFL-ImportRelationship*).

Comme on l'a vu dans le chapitre 4.2, la relation d'importation est une généralisation du mécanisme d'héritage présente dans les langages à objets. Le métaprogrammeur a la responsabilité de créer un *composant-relation* OFL pour chaque sémantique de relation d'importation du langage considéré. OFL-ML générera les éléments nécessaires de manière à rendre accessible ces relations dans UML. Ainsi, le stéréotype ≪`OFLImportRelationship`≫ (voir figure 5.8) est abstrait. Il est la base de tous les stéréotypes concrets associés à une relation d'importation du langage. Le nom du stéréotype généré est directement dépendant de celui associé au composant OFL correspondant[12]. Il sera associé à un ensemble de valeurs étiquetées. Leurs valeurs correspond à quelques-unes des caractéristiques (au sens de OFL) définies pour les relations d'importation[13]. Par exemple, la valeur étiquetée *redefinedFeatures* contiendra la liste

[12]Par exemple, le composant *ComponentJavaExtends* produira le stéréotype ≪`JavaExtends`≫.
[13]En fait, on ne trouvera que les caractéristiques qui ont un sens dans la description d'un diagramme de classifieurs.

des noms de primitives qui sont redéfinies (à travers cette relation), dans la classe source (voir section 4.2).

Une valeur est associée à chacun des paramètres et des caractéristiques qui participent à la définition d'un composant ; elle participe à la description de la sémantique de celui-ci. Une contrainte sera générée pour chaque couple (paramètre/caractéristique, valeur) d'un composant-relation. Par exemple, le paramètre indiquant s'il est possible ou non de redéfinir des primitives (*redefining*) à partir d'une relation R peut prendre les valeurs *mandatory* ou *forbidden* (voir [93]). Si le *composant-relation* correspondant associe à ce paramètre la valeur *mandatory*, la contrainte suivante sera générée :

```
context ComponentRelationship (OFLImportRelationship)
inv : self.parent.features->forAll( f : Feature |
          self.stereotype.taggedValue ->forAll(t |
               t.name='redefinedFeatures' implies t.value->includes(f)))
```

On rappelle qu'une des extensions du modèle OFL (voir section 4.3) permet de définir des modifieurs pour un composant-relation donné. Chaque modifieur associé à un composant donnera lieu à la génération de contraintes dans le profil [260].

5.1.3. Langage LAMP

LAMP est une proposition de langage pour définir les règles de protection et de visibilité mises en œuvre dans les différents langages de programmation. Son étude a été initiée dans le cadre de la thèse de Gilles Ardourel [13] et a été poursuivie dans le cadre, d'abord d'une collaboration informelle, puis d'une action COLOR de l'INRIA. Les travaux réalisés dans le cadre de cette collaboration se situe dans la continuité de ceux qui ont amené la définition du modèle OFL. Nous présentons d'abord le contexte et les motivations, puis nous donnons un aperçu des résultats.

Motivation L'encapsulation constitue l'un des principes fondamentaux des langages de programmation à objets. Indépendamment des critiques que nous avons formulées dans le chapitre 3, elle offre la capacité de localiser une information là où elle est pertinente et peut être traitée. Elle participe donc à la modularité mais également à une plus grande séparation entre l'interface et l'implémentation des classes.

Les langages à objets proposent, pour mettre en œuvre l'encapsulation, des techniques qui consistent à associer à certaines entités du programme (tels les objets, les méthodes ou les classes) des directives de protection comme *public* ou *protected* en Java [143] ou *export* en Modula [152]. Les mécanismes du langage interprètent ces directives pour permettre ou limiter l'accès à certains aspects des objets suivant le profil de l'entité qui y accède. Une utilisation correcte de ces mécanismes de protection permet de réduire le couplage entre les composants et assure ainsi de bonnes conditions de maintenance ou de réutilisation [78, 45].

Les moyens d'expression de la protection dans les langages existants sont bien souvent trop faibles pour les concepteurs, programmeurs et documenteurs de logiciels. Ceux-ci ne disposent trop souvent que de *public*, *protected* ou *private* et de leurs correspondants UML $+$, $\#$ et $-$ [192, 235] qui n'ont pas toujours la même signification d'un langage à l'autre, voire d'une version d'un même langage à la suivante.

Nous pensons que les mécanismes de protection devraient être plus expressifs, et qu'un langage de définition de mécanismes de protection pourrait bénéficier à la fois aux concepteurs

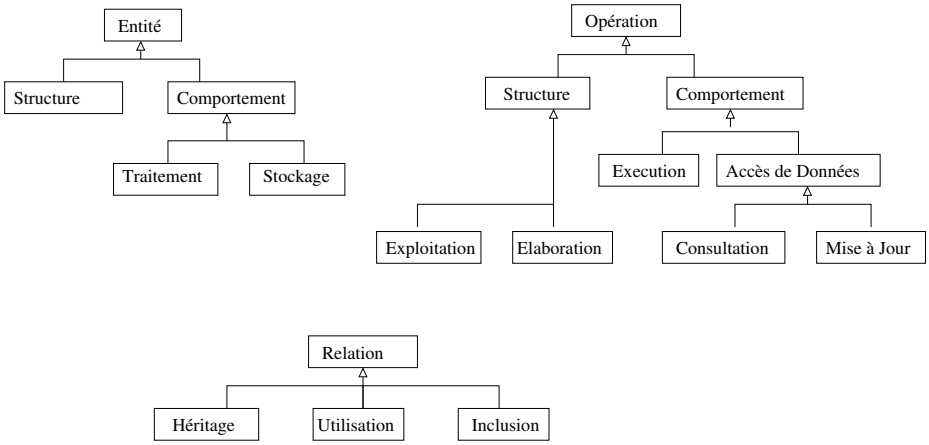

FIG. 5.9.: Partie *built-in* des métainformations

et programmeurs d'applications et aux métaprogrammeurs qui définissent des langages, en décrivant clairement une sémantique et en assurant la continuité des spécifications de protection entre la conception et l'implémentation. Une telle explicitation de ces mécanismes entraînerait aussi une meilleure compréhension des concepteurs, qui les utiliseraient donc plus souvent et à meilleur escient, et des programmeurs qui traduiraient mieux les spécifications des concepteurs.

Ce travail a pour objectif de proposer un langage simple, nommé *LAMP*, de définition de mécanismes de protection. Nous proposons d'abord un ensemble de métainformations ainsi que les contraintes d'utilisation associées[14], qu'il faut recenser dans le langage de programmation cible ; nous nous appuyons pour cela sur les besoins de langages existants, en particulier Java. Dans le paragraphe suivant, nous évoquons principalement l'apport d'un niveau méta tandis qu'une description plus complète se trouve dans [14].

Résultats et apport des aspects méta Pour obtenir un langage de définition des protections qui s'applique à n'importe quel langage à objets, nous avons choisi de le faire reposer sur un méta niveau qui propose une réification des entités d'un langage. Ce métaniveau a une partie fixe (voir figure 5.9), mais il est conçu de manière à pouvoir être enrichi au fur et à mesure des besoins requis par les langages.

Ces métainformations contiennent donc trois hiérarchies qui représentent *i)* les entités manipulées par un langage de programmation, *ii)* les opérations possibles sur ces entités et, *iii)* les relations qui peuvent exister entre les différentes entités. En fonction du langage pour lequel on veut définir et vérifier les règles de visibilité, on placera dans chacune des trois hiérarchies les entités, relations et opérations adéquates comme le montre à titre d'exemple la figure 5.10.

Il faut ensuite définir les entités qui sont acceptées comme source et/ou comme cible d'une relation, ainsi que les opérations qui peuvent leur être appliquées. L'ensemble de ces informations est suffisant pour ensuite définir les règles de protection et visibilité à l'aide d'un mini-langage [14, 13] qui s'appuie sur ce métaniveau. Enfin se pose le problème de la mise en œuvre de ces règles ; pour cela, nous avons commencé à définir un prototype à l'aide de SMARTTOOLS [88, 28, 255]. Nous utilisons son langage de description (appelé *absynt*) pour

[14]Pour cela, nous nous appuyons sur l'étude menée dans le cadre de la thèse de Pierre Crescenzo [93].

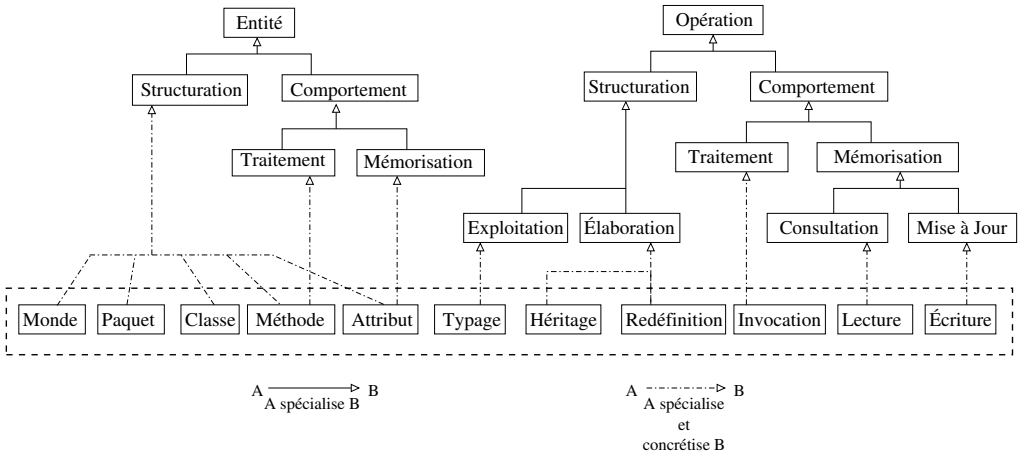

FIG. 5.10.: Métainformations relatives aux entités d'un programme

décrire le langage cible ainsi que quelques informations complémentaires qui permettent de mémoriser *i)* le nom et la localisation de l'élément du langage, *ii)* les opérations valides quand c'est une entité et les entités sources et cibles compatibles lorsque c'est une relation[15] et *iii)* les règles de protection mises en œuvre pour une entité. Ensuite, il faut décrire un ensemble de visiteurs qui, en fonction des éléments du langage vérifiera que les règles sont satisfaites et produira le cas échéant, un message d'erreur ou un traitement approprié. À l'aide de ce prototype, nous envisageons d'étudier et de valider de nouvelles extensions de langage relativement aux protections.

5.2. Application à la description de logiciel

Une seconde catégorie d'applications du modèle OFL a été d'étudier d'une part les possibilités d'améliorer la description de l'usage de l'héritage [94][16] et d'autre part de proposer des relations qui dérivent de la relation d'héritage, à savoir *i)* une relation d'héritage inverse que nous comptons exploiter pour adapter et faire évoluer les classes et, *ii)* une relation d'héritage générique dont l'objectif est de voir comment on peut apporter des solutions pour introduire dans un langage à objets des techniques de séparation des préoccupations [162].

5.2.1. Annotation de l'héritage

Nous partons de la constatation partagée par [13] et [213] qu'il est important de permettre à un classifieur de se protéger des autres entités du programme afin de ne pas leur permettre de briser la cohérence de ses instances. L'expressivité des langages de programmation comme Eiffel, Smalltalk, Java, C# ou C++ permet de prendre en compte de multiples situations sans pour autant que les choix du concepteur de l'application soient toujours spécifiés de manière explicite ou même qu'ils puissent l'être. Même si des tentatives de clarification sont faites en

[15] Cette information est en fait une métainformation. Elle est en effet associée à l'ensemble des instances des entités ou relations et non pas à une instance donnée comme c'est le cas pour *i)* et *iii)*.

[16] Ce travail comme celui sur l'héritage générique a été réalisé dans le cadre du stage de DEA de Christophe Jalady.

ce sens, c'est par exemple le cas pour l'usage de l'héritage, il reste encore beaucoup à faire dans ce domaine. D'ailleurs de nombreux articles discutent des utilisations justifiées (ou non) de la relation d'héritage [149, 296].

A propos de ces tentatives de clarification, on peut citer par exemple le cas de Java ou d'Eiffel. Le mot-clé *abstract* pour le premier et *deferred* pour le second permet de souligner que le classifieur contient des définitions de routines abstraites ou retardées, selon le vocabulaire employé dans chacun des langages.

En Java, plutôt que d'utiliser une seule relation d'héritage on dispose de deux mots-clés : *extends* qui décrit une relation d'héritage simple entre classes ou multiple entre interfaces, et *implements* qui décrit une relation d'héritage multiple entre une classe et des interfaces. Par ailleurs, on trouve aussi en C++ la possibilité de faire un héritage « privé » (mot-clé *private*), qui interdit en particulier l'application du polymorphisme sur les instances comme c'est normalement le cas lorsqu'on utilise le symbole *« : : »*. Ces approches constituent à notre sens une avancée car elles permettent au développeur d'applications de mieux spécifier l'usage qu'il fait de l'héritage mais elles nous semblent encore insuffisantes.

Notre proposition consiste à permettre au développeur d'application (de manière facultative) d'ajouter un ensemble d'annotations à la description des classifieurs ou des relations d'héritage afin de mieux en préciser l'usage. Il est important de mentionner que ces annotations sont amenées à être utilisées par des compilateurs, des interprètes ou des environnements de programmation plutôt que par l'exécutif du langage ; elles n'ont pas vocation à modifier la sémantique des relations d'héritage ou plus généralement du langage. Nous défendons l'idée que grâce à cette approche, un développeur se voit offrir des moyens supplémentaires en vue d'obtenir des applications plus documentées, réutilisables, maintenables et robustes.

On retrouve des préoccupations similaires dans ClassTalk [196] qui propose de spécifier la nature des classes en s'appuyant sur un *Meta-Object Protocol* (MOP). On retrouve en particulier des propriétés relatives au fait d'être abstraite ou de ne pas pouvoir posséder de sous-classes. D'autres caractéristiques sont introduites comme la possibillité de spécifier l'ensemble de méthodes à redéfinir par les sous-classes ou d'empécher la modification de l'interface. Elles sont encapsulées dans des métaclasses et influent sur les futures relations d'héritage entre une classe (instance d'une métaclasse donnée) et de ses (futures) sous-classes. D'une certaine manière, l'auteur propose à une classe de pouvoir contraindre les futures relations d'héritage dont elle sera la cible dans le but d'améliorer l'organisation structurelle des classes plutôt que pour envisager comme nous, des contrôles supplémentaires sur les hiérarchies.

Une approche basée sur des classifications Nous désirons faire reposer notre ensemble d'annotations sur des taxinomies de l'héritage ; nous en avons trouvées deux dans l'état de l'art. Il s'agit d'une part de la taxinomie de Bertrand Meyer [213] qui a pour principal objectif d'expliciter les différents cas où l'utilisation de l'héritage peut se justifier (voir figure 5.11), et d'autre part, de celle proposée par Xavier Girod [135] dont l'objectif est plutôt de recenser les différentes sortes d'utilisation de l'héritage en prenant le point de vue du concepteur d'applications.

Nous proposons en complément une troisième classification qui est orthogonale aux précédentes et dont l'objectif est de recenser les différentes limitations que l'on peut appliquer aux nœuds de ces classifications sans en dénaturer le sens.

La figure 5.12 représente seulement un extrait de cette classification, dans lequel nous nous

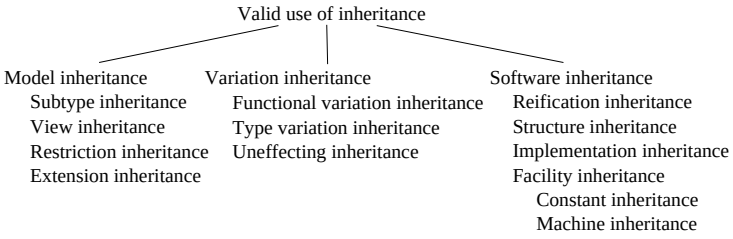

FIG. 5.11.: Taxinomie de l'héritage de Bertrand Meyer

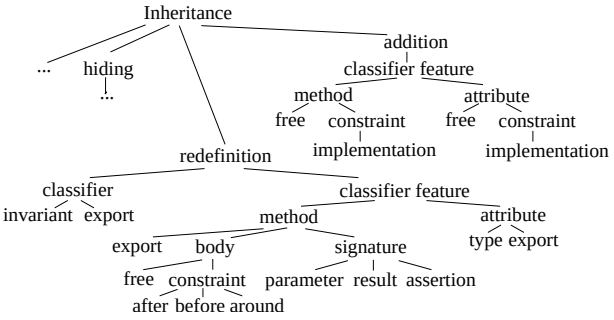

FIG. 5.12.: Extrait d'une classification « de référence » des adaptations réalisées par héritage.

intéressons principalement aux adaptations relatives à l'ajout et à la redéfinition de caractéristiques. Parmi les catégories d'adaptation non citées dans le schéma, on peut recenser par exemple, le renommage de caractéristiques ou l'inactivation d'un corps de méthode. La classification proposée veut être indépendante des langages ; il est donc probable que certaines adaptations n'aient pas de sens dans un langage donné. Par exemple, la redéfinition de signature n'existe pas en Java alors qu'elle fait partie des facilités offertes par le langage Eiffel.

La définition d'une annotation se définit en plusieurs étapes, qui nécessitent de disposer d'une réification adaptée des types d'entités (métaentités) qui sont mentionnées dans la classification dite de référence.

Définition par étapes d'une annotation Il s'agit dans un premier temps d'attribuer des règles qui définissent chaque nœud de cette classification[17]. Ces règles s'appuient sur la réification des entités et sur le fait que l'adaptation associée au nœud considéré est obligatoire (*Mantatory*), interdite (*Forbidden*) ou simplement permise (*Allowed*)[18]. Voici plus précisément ce que signifient ces valeurs :

- **Mandatory** : l'adaptation associée au nœud doit être appliquée pour chaque occurrence de la métaentité représentée (en général un classifieur, un attribut ou une méthode). Par exemple, « *toutes les méthodes de la classe doivent être redéfinies* ».
- **Forbidden** : aucune occurrence de la métaentité représentée ne peut effectuer cette adaptation.
- **Allowed** : l'adaptation associée au nœud est possible, c'est-à-dire qu'elle peut ou pas, être appliquée sur une occurrence donnée de la métaentité.

[17]Ces règles peuvent être comprises comme étant des assertions de niveau méta pour l'application.
[18]Il faut au moins que chaque nœud puisse disposer d'une règle pour *Mandatory* et d'une autre pour *Forbidden*.

Ainsi nous citons l'exemple suivant de règle[19] qui participe à la définition du nœud « *inheritance/...method/body/free* » :

```
∀R, une relation d'héritage,
   soit R.S, son classifieur source et
        R.T, son classifieur cible,
   soit R.M_ST, l'ensemble des méthodes de R.S redéfinies dans R.T,
constraint = forbidden implies R.M_ST = ensemble vide
```

Dans un deuxième temps, il faut choisir la classification qui est adaptée à l'objectif considéré, par exemple celle de [213] ou celle de [135]. Puis, il faut décrire chacun des nœuds terminaux de la classification[20] (c'est la *définition initiale* qui correspond à un ensemble d'informations). Chacune de ces informations correspond à un *nom de nœud* de la classification de référence, et une *valeur* parmi *Allowed* / *Mandatory* / *Forbidden*.

D'un point de vue plus général, on peut dire que la plupart du temps les informations, qui décrivent les types d'héritage (nœuds de la classification), sont données sous forme d'un texte ou d'exemples explicatifs sujets à interprétation. Par l'approche que nous sommes en train de décrire, nous proposons à la fois, d'en extraire, quand c'est possible, des règles plus formelles qui définissent leur sémantique, et d'organiser toutes les informations recueillies en trois catégories[21] : *i)* les informations d'adaptation qui décrivent les capacités d'adaptation du classifieur source et qui peuvent donc être décrites par un ensemble de nœuds de la classification de référence (c'est la *définition initiale* mentionnée ci-dessus), *ii)* les informations sur la structure de la hiérarchie qui dépendent de la hiérarchie globale et qui pourront être définies à l'aide de métainvariants, *iii)* les informations intuitives qui correspondent à une description textuelle ou à des exemples de la sémantique de l'héritage.

Dans un troisième temps, il s'agit de définir l'annotation proprement dite. De manière intuitive, une annotation va nous permettre d'une part d'associer un usage U (par exemple *variation inheritance* [213]) à une relation d'héritage R qui pourrait exister entre deux classifieurs d'une bibliothèque, et d'autre part, de mémoriser une redéfinition éventuelle de la définition initiale de U afin de spécifier pour cette relation d'héritage l'existence d'une situation particulière. Ainsi une annotation va contenir les informations suivantes : *i)* le nom associé à un nœud d'une taxonomie (par exemple *variation inheritance*) qui permet d'accéder aux informations mentionnées plus haut et éventuellement, *ii)* un ensemble de nœuds de la classification dite de référence qui représente une redéfinition de la définition initiale de U (elle est appelée *définition redéfinie*).

La manipulation des définitions initiale et redéfinie nécessitera de disposer d'opérateurs ensemblistes mais aussi d'opérateurs permettant d'ajouter ou d'enlever des éléments à ces ensembles, de modifier la valeur (*Allowed*, *Mandatory* ou *Forbidden*) associée à un de leurs éléments. L'utilisation de ces annotations donnera lieu à la définition d'actions à réaliser en cas de succès ou d'échec de la vérification des règles ; ces actions dépendent bien sûr des fonctionnalités à introduire (par exemple dans un environnement de programmation), à travers l'utilisation des annotations.

Intégration dans le langage Eiffel À partir de l'approche décrite ci-dessus, nous avons proposé une intégration dans le langage Eiffel [210, 212] ; elle permet d'illustrer cette approche.

[19] Sa description repose naturellement sur une réification des classifieurs et des relations.

[20] L'ensemble des règles, qui seront recensées dans un nœud donné d'une taxonomie ou classification, est égal à l'union des règles associées aux nœuds ancêtres.

[21] Il est à noter qu'un nœud de taxonomie peut éventuellement être composé par d'autres nœuds.

Le langage Eiffel propose déjà une clause d'indexation (*indexing*) décrite principalement avant la description d'une classe ; elle permet de donner des informations complémentaires sur la classe qui sont destinées à être utilisées par des outils externes ou par l'environnement de programmation. La liste des types d'information est libre et peut être étendue à volonté par le développeur de la classe mais il est naturellement très important qu'il y ait une uniformisation car elle permet d'envisager la mise en œuvre de recherches systématiques d'information.

Notre approche intègre la notion d'annotation à travers l'extension de la clause d'héritage (*inherit*) avec une clause d'indexation (*indexing*). La déclaration ci-dessous définit un exemple d'utilisation. On peut noter que le choix d'ajouter cette information dans la clause *indexing* plutôt que par l'intermédiaire d'autres ajouts syntaxiques insiste sur notre volonté de ne pas modifier la sémantique du langage et de faire en sorte que ces informations soient utilisées par des outils externes. De même que pour la clause *indexing* de classe en Eiffel, la clause *indexing* d'héritage est facultative.

Dans cet exemple, la clause *indexing* précise que la classification utilisée est celle décrite dans la figure 5.11 et que le nœud choisi dans la classification représente un héritage de concrétisation.

```
class LINKED_LIST [G]
inherit DYNAMIC_LIST [G]
        rename
          ....
        redefine
          ....
        indexing
          taxonomy : "valid_use_inherit"
          use : "reification_inheritance"
        end
 ...
end -- LINKED_LIST
```

On désire maintenant indiquer dans cette classe plusieurs limitations ponctuelles en se servant du contenu de la classification de référence (figure 5.12). Pour éviter d'alourdir la classe avec le chemin complet des nœuds, on a préféré donner ici un nom qui se veut le plus significatif possible. Il est à noter que les opérateurs $+$, **-** et $=$ signifient respectivement ajouter, enlever et remplacer un élément de la définition initiale. Dans le schéma ci-dessous, le programmeur dit à travers leur évocation *i)* qu'il permet de redéfinir de manière contrainte le corps des routines (il était libre auparavant, il doit comporter désormais avant le code ajouté, l'appel à la version originale de la méthode), *ii)* qu'il est possible d'ajouter des attributs ou des méthodes d'implémentation qui ne doivent pas être exportées vers d'autres classes et que désormais il faut rendre effectives toutes les méthodes (auparavant c'était facultatif). Les lettres **A**, **F** et **M** correspondent respectivement à *Allowed*, *Forbidden* et *Mandatory* (voir ci-dessus).

```
indexing
   ...
  taxonomy_inherit : "valid_use_inherit"
class LINKED_LIST [G]
inherit DYNAMIC_LIST [G]
        ....
        indexing
          use : "reification_inheritance"
          redefinition : "- method_redefinition_body
                         + method_redefinition_after (A),
                         + add_implementation_feature (A),
                         = method_body_definition (M)"
        end
 ...
end -- LINKED_LIST
```

On notera que dans le cas où la clause *indexing* associée à chaque relation d'héritage qui est annotée est relative à la même classification, il pourra être intéressant, comme c'est fait dans l'exemple ci-dessus, de donner cette information dans la clause *indexing* de la classe afin de ne pas répéter plusieurs fois la même information. Le lecteur pourra trouver des fonctionnalités complémentaires dans [94]. En particulier il est possible *i)* de considérer qu'une annotation concerne plusieurs relations d'héritage (héritage composé [135]) et, *ii)* que des classifieurs puisse se protéger contre d'autres et ainsi, spécifier qu'ils ne peuvent être hérités que sous certaines conditions.

5.2.2. Une relation d'héritage générique

La réflexion que nous avons menée sur l'héritage depuis le début de nos travaux sur OFL nous a amenés à nous intéresser à nouveau à la problématique abordée par le chapitre 3 mais en l'abordant sous un angle différent. D'un point de vue historique, ces travaux se sont déroulés en parallèle, leurs motivations sont proches et naturellement les constatations faites par rapport aux apports et limitations du paradigme objet sont similaires.
Le travail réalisé a donc porté sur la pertinence de l'ajout d'une nouvelle relation, dite relation d'héritage générique, en vue d'améliorer la réutilisabilité, la maintenance et la lisibilité du code d'une application. Nous sommes restés dans le cadre des langages à objets classiques auxquels nous souhaitions simplement ajouter un type de relation d'héritage. L'objectif était donc d'une part, d'étudier les différentes possibilités permettant de modifier la cible[22] d'héritage d'une classe en fonction de son contexte d'utilisation, c'est-à-dire suivant les différentes applications qui l'utilisent ou même suivant les différentes parties d'une même application, et d'autre part, le cas échéant de proposer une approche complète.

Rappels sur le contexte Les relations d'héritage proposées par les langages à objets permettent une structuration des classes sous forme de hiérarchies pouvant aboutir à la définition de bibliothèques. Ces dernières doivent *i)* pouvoir être réutilisées dans des *contextes* variés, par exemple dans des applications différentes, ou pour construire de nouvelles bibliothèques *ii)* pouvoir évoluer : une fois réalisée, une bibliothèque doit être maintenue, mais aussi doit pouvoir être étendue pour permettre de résoudre de nouveaux besoins. Dans ce contexte, la disponibilité des sources est généralement admise, facilitant la résolution de ces problèmes par les concepteurs.

Cependant l'utilisation dans différents contextes d'une bibliothèque réalisée par un tiers nécessite aussi de pouvoir l'adapter à ses propres besoins [155, 243, 194, 269, 249]. En particulier, il pourra s'agir de l'étendre pour une application spécifique ; ou encore de lui ajouter des fonctionnalités, ou même de la modifier en partie. Toutes ces adaptations devront pouvoir se faire par des utilisateurs sans modifier le code source.

On a déjà constaté dans le chapitre 3 que la relation d'héritage ne permet d'étendre une hiérarchie que par l'ajout de nouvelles classes dérivant d'une ou plusieurs autres classes[23] et qu'elle ne répond ainsi qu'à une partie seulement des besoins. Or un développeur, utilisateur d'une bibliothèque, doit pouvoir, dans une certaine mesure, manipuler les hiérarchies de classes qui composent la bibliothèque. En particulier, il doit pouvoir ajouter des entités implémentant

[22] La cible (respectivement source) d'une relation d'héritage représente la super-classe (respectivement la sous-classe).
[23] Suivant le type d'héritage possible, simple ou multiple.

de nouvelles fonctionnalités, modifier une implémentation, ou généraliser deux hiérarchies indépendantes. Ces possibilités sont nécessaires notamment pour l'ajout de code orthogonal. Ce dernier représente la mise en œuvre de propriétés non spécifiques à une classe particulière (ou à un type particulier) et qui en général concerne un ensemble de classes qui ne sont pas reliées par des relations d'héritage ou d'agrégation.

Comme on l'a vu dans la section 3.1, plusieurs approches sont envisageables pour adresser la problématique décrite ci-dessus. Une première approche consiste à développer ces propriétés de manière externe à une application pour ensuite les intégrer ; c'est le principe proposé par le paradigme de la séparation des préoccupations [172]. D'autres travaux ont été réalisés pour accroître l'expressivité des langages ou, du moins, améliorer leurs capacités à réutiliser l'existant ou à factoriser les comportements. Comme ils ont été évoqués longuement dans la section 3.1, nous ne reviendrons pas dessus sauf quand les solutions reposent plus précisément sur l'héritage ou la généricité :

- La généricité : Plusieurs langages ont proposé des solutions, il s'agit *i)* d'Eiffel [213] qui implémente la notion de classe générique et la généricité contrainte[24], *ii)* de C++ [290] qui s'appuie sur un ajout syntaxique et la réécriture pour définir de fait des patrons de classe, *iii)* de Pizza qui est une extension de Java [227] et, *iv)* de Java depuis la version 1.5 [29].
- Les mixins : ils contribuent à un assouplissement de la relation d'héritage. En effet, un mixin ne spécifie pas lors de sa déclaration de cible d'héritage. Il pourra ainsi s'attacher, suivant son utilisation, à différentes superclasses permettant ainsi de spécialiser plusieurs entités sans avoir à impliquer des duplications de code. Les principales implémentations des mixins ont été réalisées pour CLOS [50], C++ [278] et Java [10].
- Les classes virtuelles : Cette notion a été définie pour le langage Beta [201, 38], qui généralise les notions de procédure et de classe, sous la forme de *pattern*. Beta a un comportement spécifique relativement à ces procédures virtuelles : c'est la *super-méthode* qui définit, *via* le mot-clé *inner*, où les futures spécialisations de l'algorithme seront intégrées.
- Les langages de configuration : ils permettent de modifier le comportement d'une application de manière externe au langage et, dans une certaine mesure, de paramétrer le comportement d'une bibliothèque pour l'adapter à des besoins spécifiques. On peut par exemple citer le cas de *Lace*[25] [212] qui permet entre autre de renommer une classe lorsqu'il y a conflit de noms entre deux bibliothèques utilisées par un même système.

Résultats de l'étude L'étude d'exemples basés sur les systèmes mentionnés ci-dessus qui est proposée dans [162] a montré que des travaux significatifs étaient menés dans ce domaine. Ainsi il semble intéressant de s'inspirer des approches basées sur :

- La généricité horizontale (classe générique). Celle-ci doit s'intégrer de manière homogène avec l'héritage générique qui concerne la généricité verticale.
- Les métainformations. La définition du mécanisme d'annotations proposé dans la section 5.2.1 repose sur cette technologie, nous verrons plus loin comment l'exploiter.
- Les *mixins* et les *templates* C++ où l'idée de paramétrer la cible de l'héritage est fortement présente. Par ailleurs, la notion de patron de relation par analogie avec la notion de patron de classe (*template*) nous semble intéressante.

[24] Certains cas d'utilisation de la généricité contrainte sont critiqués par [125].
[25] Lace signifie : Language for Assembling Classes in Eiffel.

- La programmation par aspects : le fait de pouvoir adapter une hiérarchie de classes de manière à ce qu'elle intègre désormais une nouvelle fonctionnalité orthogonale est particulièrement important.
- Les *classes virtuelles* (Beta) et les *before/after* de CLOS qui permettent d'assurer que la redéfinition du corps d'une méthode pourra être contrainte. Nous avons d'ailleurs intégré cette facilité dans la classification de référence mentionnée dans la section 5.2.1 qui est associée au mécanisme d'annotations. Par ailleurs, les idées developpées semblent un bon support pour décrire des patrons de classes et de méthodes.

En parallèle à la mise en évidence de quelques-uns des principaux éléments d'une approche pour une relation d'héritage générique, nous avons montré que l'adjonction d'un mécanisme permettant de modifier la cible de l'héritage induit un certain nombre de problèmes. Nous pouvons citer :

- des conflits de noms qui sont introduits notamment lorsqu'une classe cible étend l'interface de la classe initiale ;
- la redéfinition covariante des méthodes n'est pas sûre du point de vue des types mais est nécessaire pour une bonne modélisation [119]. Cependant, avec l'ajout de l'héritage générique, la redéfinition covariante des méthodes peut par exemple provoquer une redéfinition contravariante alors même qu'elle n'est pas prévue dans le langage.
- La cohérence des assertions. Si nous nous appuyons sur les concepts mis en évidence dans [213], les contraintes imposées sont très importantes et restreignent beaucoup l'usage de l'héritage générique.

Pour répondre à ces problèmes, nous proposons le mécanisme d'annotation mentionné dans la section 5.2.1. Il veut offrir un support pour faire plus de contrôle et ainsi prévenir les cas qui posent problème en vue par exemple de les interdire[26]. Même si cette solution peut ne pas apparaître idéale (elle restreint l'utilisation de la généricité), elle a l'avantage de garantir la fiabilité et la robustesse de l'application. Cependant, un certain nombre de questions restent ouvertes et n'ont été que partiellement abordées ou étudiées :

- Est-il possible d'utiliser à plusieurs endroits d'une même application des instanciations différentes d'une même classe générique ? Dans ce cas, comment règle-t-on la compatibilité des objets ?
- Est-il préférable d'opter pour l'introduction d'un mécanisme permettant de redéfinir la cible d'une relation d'héritage ou bien est-il opportun d'étendre la généricité aux relations d'héritage ?
- À quel endroit doit être décrit le choix de la classe cible, que ce soit par instanciation ou redéfinition ? À l'intérieur de l'application ou bien à l'extérieur, par exemple à l'aide d'un langage de configuration ?
- Quelles doivent être les capacités d'adaptation mises à disposition de l'architecte de l'application lorsque ce dernier choisit la cible effective d'une relation d'héritage ?

La réponse à ces questions nécessite en particulier de réfléchir à la plate-forme de mise en œuvre mais aussi d'étendre l'examen de l'état de l'art aux langages de configuration afin de voir comment ces derniers pourraient intégrer des capacités d'adaptation. Une perspective que nous comptons mener à bien est d'étudier une solution complète dans le cadre d'un langage à objets comme Eiffel. En particulier, il pourra être intéressant de proposer *i)* une

[26] Un tableau détaillé des types de problèmes en fonction de la classification de l'héritage proposé par B. Meyer (voir figure 5.11) est proposé dans [162].

solution qui étende Lace, *ii)* une intégration de notre mécanisme d'annotations comme nous avons commencé à le faire dans [94], *iii)* un moyen de définir des patrons de méthodes ce qui permettra d'augmenter l'expressivité de la solution et ainsi de la rapprocher de celles basées sur la séparation des préoccupations.

5.2.3. Une relation d'héritage inverse

À travers l'étude d'une relation d'héritage inverse, nous nous intéressons à nouveau à la problématique abordée par le chapitre 3. L'état de l'art fait référence à plusieurs travaux qui concernent des formes d'héritage inverse. On peut citer en particulier [269] pour le langage Java[27], [194] pour le langage Eiffel ou d'autres travaux plus proches d'UML [98, 103]. Plus précisément, nous désirons intégrer quelques-unes des techniques issues de la séparation des préoccupations avec une relation d'héritage inverse pour augmenter les capacités d'adaptation et de réutilisation des classes. Comme pour l'approche développée dans la section 3.3, nous désirons atteindre notre objectif par la réalisation d'un modèle métier dont une des implémentations pourra correspondre à l'extension d'un langage existant. Ce travail correspond à la thèse de Ciprian Chirila qui est actuellement à mi-parcours [75, 74, 76] et nous reviendrons donc sur certains aspects dans les perspectives (section 7.1.2).

Par rapport à d'autres approches comme la programmation par aspects ou par sujets, la programmation par composants ou bien encore la programmation par modèles (voir chapitre 6), l'héritage inverse n'a surement pas l'expressivité la plus riche mais c'est une extension de la programmation par objets qui a l'avantage de s'intégrer parfaitement à la philosophie objet et qui peut rendre de nombreux services comme nous allons le montrer dans les lignes qui vont suivre. Introduire l'héritage inverse est particulièrement intéressant pour les langages qui ne proposent que l'héritage simple parce qu'il permet de capturer une partie de l'expressivité de l'héritage multiple. Nous décrivons d'abord trois utilisations potentielles (insertion d'une classe dans une hiérarchie, construction de types dérivés, composition de hiérarchies), puis nous proposons une sémantique convenable.

Utilisations potentielles Un des avantages cités le plus souvent dans la littérature pour convaincre le lecteur de l'utilité de l'héritage inverse est l'adaptation d'une hiérarchie de classe existante qui n'est pas assez générale. L'exemple qui est proposé dans la figure 5.13 présente l'exemple très classique d'une hiérarchie de classes décrivant des objets géométriques qui n'intégre pas la classe rectangle[28] et qu'il faut donc adapter lorsque cette dernière est nécessaire. Le code source 5.14 décrit une partie du code qui lui est associé.

Pour réaliser une telle adaptation nous avons besoin de pouvoir : *i)* créer une nouvelle classe (*Rectangle* dans cet exemple), *ii)* insérer cette dernière dans la hiérarchie, c'est-à-dire modifier la superclasse d'une classe (*Parallelogram* dans cet exemple), *iii)* factoriser des primitives (*h*, *heighten*, *area* dans cet exemple), *iv)* ajouter de nouvelles primitives (aucune dans cet exemple).

Contrairement à la relation d'héritage classique, qu'elle soit simple ou multiple, l'héritage inverse permet au programmeur de modifier le contenu et le comportement d'une classe sans

[27] Nous lui empruntons notamment le mot-clé exherits et les principaux choix seront discutés avec son auteur.

[28] Le propos n'est pas ici de savoir si la hiérarchie proposée dans la figure 5.13 doit utiliser l'héritage d'extension [213] comme c'est le cas ici entre *Square* et *Parallelogram* ou l'héritage de spécialisation comme nous l'avons proposé dans [76].

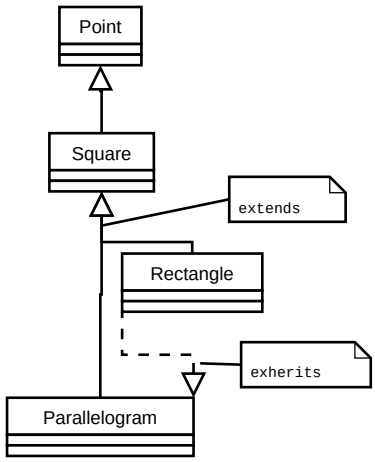

FIG. 5.13.: Amélioration d'une hiérarchie de classes de la bibliothèque Java

```
public class Square extends Point  {
    protected int w = 0;
    public void widen(int W) { w = W; }
    public double area() { return w*w; }
}
public class Parallelogram extends Square {
    protected int h = 0;                          // should be in Rectangle if exists
    public void heighten(int H) { h = H; }      // should be in Rectangle if exists
    protected double a = 0;
    public void skew(double A) { a = A; }
    public double area() { return w*h*sin(a);} // should be in Rectangle if exists
                                                // but with another implementation
}
public class Rectangle extends Square exherits Parallelogram {
    protected factored int h = 0;                    // factored from parallelogram
    public factored void heighten(int H) { factored } // factored from parallelogram
    public factored double area() { return w*h; }     // factored from parallelogram
                                                     // but with its own body
}
```

FIG. 5.14.: Code source des classes correspondant à la figure 5.13

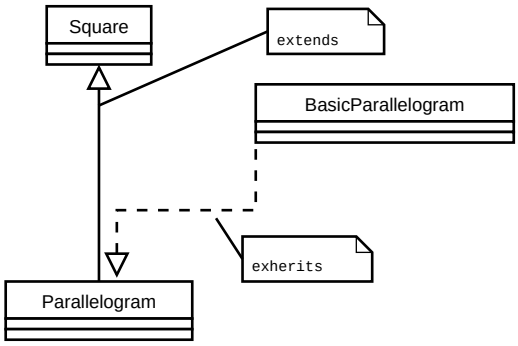

FIG. 5.15.: Un exemple de type dérivé

```
public class BasicParallelogram exherits Parallelogram {
    protected factored int w = 0;                    // factored from square
    protected factored int h = 0;                    // factored from parallelogram
    public factored void widen(int w) { factored }    // factored from square
    public factored void heighten(int h) { factored } // factored from parallelogram
    public Parallelogram toParallelogram () {         // feature which is added
        // Code to be defined as usual
       }
}
```

FIG. 5.16.: Code source d'une classe correspondant à la figure 5.15

être obligé de créer une classe héritière ni de toucher à une ligne de son code source. Dans le code source de la figure 5.14 toutes les primitives factorisées sont déclarées dans la classe *Parallelogram* elle-même et non dans une de ses superclasses alors que théoriquement ce serait possible. Le fait que la classe *Rectangle* hérite elle-même de *Square* lui donne automatiquement accès à toutes les primitives de ce dernier ainsi qu'à celles de ses ancêtres et donc vouloir les factoriser dans *Rectangle* n'aurait pas de sens.

La figure 5.15 décrit un exemple dans lequel, contrairement au cas précédent, cela a un sens de vouloir factoriser des primitives d'un ancêtre de la classe déclarée comme cible de l'héritage inverse. Cela permet en effet de construire des classes qui correspondent à une combinaison de plusieurs classes (ici *Square* et *Parallelogram*).

Supposons ici que la classe *Parallelogram* contienne des informations spécifiques à la bibliothèque graphique (par exemple *X-Window*) ; ces informations peuvent par exemple correspondre à des pointeurs qui n'ont de sens qu'à l'intérieur d'une même session et qui sont donc dépendant du contexte d'exécution. La classe *BasicParallelogram*, construite par héritage inverse, qui peut être abstraite ou concrète selon les besoins, contient toute l'information relative à un parallélogramme mais sans ces informations spécifiques. L'existence de cette classe permettra donc de sauvegarder dans un fichier seulement les informations que l'on sait indépendante du contexte d'exécution. On notera qu'une instance de *Parallelogram* peut être associée à une primitive de type *BasicParallelogram* mais pas l'inverse ce qui nécessitera sûrement l'ajout d'une méthode de conversion, pour permettre l'utilisation de ces instances lors du rechargement (voir la figure 5.16). On n'aurait pas pu utiliser l'héritage normal pour mettre en œuvre cet exemple, car on ne veut pas conserver à la fois toutes les primitives de *Square* et de *Parallelogram*.

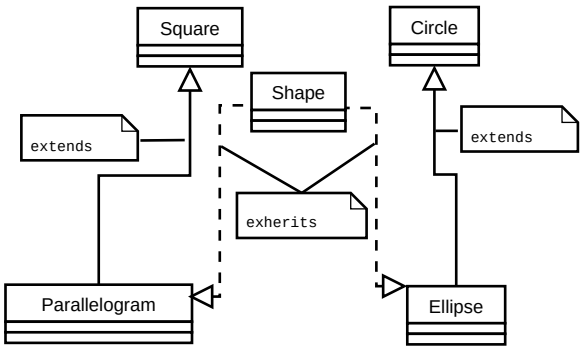

FIG. 5.17.: Un exemple de composition de hiérarchies

Dans le code source décrit dans la figure 5.16, la classe *BasicParallelogram* factorise les méthodes aussi bien que les attributs car nous voulons pouvoir mettre à jour ses instances.

La figure 5.17 propose une troisième utilisation de l'héritage inverse qui prend en compte plusieurs cibles. C'est particulièrement intéressant quand deux hiérarchies sont construites par différentes personnes ou indépendamment l'une de l'autre et que plus tard elles doivent être combinées afin d'en extraire les parties communes. Ici la classe *Shape* permet de factoriser les parties communes d'*Ellipse* et de *Parallelogram* afin de pouvoir utiliser le polymorphisme et construire par exemple une collection de *shapes* sur laquelle un traitement doit être exécuté.

Éléments de définition de la sémantique En nous appuyant sur les exemples présentés ci-dessus et sur des langages existants et en particulier Java, nous proposons que l'héritage inverse suive les règles suivantes :

- Il doit être possible de factoriser tous les types de primitive (attribut, fonction, procédure, etc.).
- Une méthode peut être factorisée dans une classe C avec son corps, uniquement si tous les attributs et toutes les méthodes manipulés par ce corps sont déclarés (ou factorisés) dans cette classe ou dans une de ses superclasses.
- Si une primitive f d'une classe *C1* est factorisée dans une classe *C2* (car *C2 exherits C1*), alors la visibilité de f dans *C1* doit être la même ou plus contrainte que la visibilité de f dans *C2*.
- Une méthode qui ne peut pas être redéfinie (*final* en Java ou *freeze* en Eiffel) peut quand même être factorisée.
- Des variables ou des méthodes de classe ne peuvent être factorisées (en particulier si la sémantique du langage ne permet pas qu'elles soient héritées).
- Si une méthode de la classe dont il faut changer la superclasse (*Parallelogram* dans notre exemple) utilise le mot-clé *super* (en Java) ou *precursor* (en Eiffel), alors ce dernier référencera désormais la version se trouvant dans la classe insérée (*Rectangle* dans notre exemple) si elle existe ou la version présente dans ses parents (*Square* pour nous), récursivement. Quand c'est un constructeur qui utilise ce mot-clé, il doit être présent dans la classe insérée (*Rectangle* dans l'exemple).
- La définition de la sémantique concernant le polymorphisme doit respecter les règles mises en évidence dans le code source se trouvant dans les figures 5.18 ou 5.19, suivant que nous

```
public class Example  {
    private Square s = new Square();
    private Parallelogram p = new Parallelogram();
    private Rectangle r = new Rectangle();
    public void test() {
       s = p; // Parallelogram is a descendant of Square
       s = r; // Rectangle is a direct descendant of Square
       r = p; // Parallelogram is a direct descendant of Rectangle
       p = r; p = s; r = s // these assignments are forbidden
    }
}
```

FIG. 5.18.: Règles pour le polymorphisme correspondant à la figure 5.13

```
public class Example  {
    private Square s = new Square();
    private Parallelogram p = new Parallelogram();
    private BasicParallelogram b = new BasicParallelogram();
    public void test() {
       s = p; // Parallelogram is a descendant of Square
       s = b; //  NOT ALLOWED
       b = p; // Parallelogram is a descendant of BasicParallelogram
       p = b; p = s; b = s // These assignments are also forbidden
    }
}
```

FIG. 5.19.: Règles pour le polymorphisme concernant la figure 5.15

voulons insérer une classe dans la hiérarchie ou dériver un nouveau type.

- Une classe C qui *exherits* d'une classe B (qui hérite elle-même d'une classe A) n'est pas autorisée à factoriser une méthode m de A (respectivement de A' un ancêtre de A), si la classe C hérite de A (respectivement de A').
- Quand la cible d'un héritage inverse ou l'un de ses descendants contient une méthode qui a exactement la même signature (en incluant le type de retour si c'est une fonction) que dans la source, alors ce n'est pas une insertion mais une factorisation et les règles ci-dessus s'appliquent, la version du descendant étant considérée comme une redéfinition (sauf si c'est une méthode de classe).
- Quand la cible d'un héritage inverse ou l'un de ses descendants contient une méthode qui a le même nom mais pas la même signature alors les primitives sont indépendantes l'une de l'autre (aucun conflit). Si la méthode est une fonction, alors les descendants n'ont pas le droit d'avoir une fonction avec la même signature mais avec un type de retour qui est différent.
- Quand la primitive ajoutée est un attribut et que la cible de l'héritage inverse contient elle aussi un attribut avec le même nom (qu'il ait ou non le même type) alors il n'y a pas de conflit mais l'attribut qui est visible est celui qui est le plus près (en montant dans la hiérarchie) du type de l'instance considérée. Bien sûr, s'il n'existe pas d'attribut de même nom dans le descendant, alors le nouvel attribut est visible dans toute la hiérarchie.

Page vide

Troisième partie .

Modélisation de modèles métiers : avancées actuelles et perspectives

Page vide

Préambule

Cette dernière partie correspond à une nouvelle facette de mes activités de recherche qui a débuté en 2004[29] et aux perspectives de recherches associées aux travaux en cours développés dans les parties I et II.

Les activités décrites dans le chapitre 6 correspondent à la fois à une évolution dans notre manière de voir le développement d'application et à une prolongation des travaux autour du modèle OFL décrits dans les chapitres 4 et 5. Ces activités se placent aussi dans la continuité des travaux sur la définition de modèles métiers[30] comme ceux proposés dans les chapitres 3 et 5, puisqu'il s'agit de définir une collection de modèles métiers dédiés à la modélisation des modèles métiers eux-mêmes.

Évolution du développement des applications ? En parallèle aux approches classiques de développement d'application plutôt centrées sur les langages à objets, il y a depuis le début des années 2000 de nombreuses tentatives destinées à faire évoluer ces approches. Elles s'appuient en particulier autour de trois idées clés :

- *La séparation des préoccupations.* Ce paradigme a été longuement évoqué dans le chapitre 3 ;
- *La programmation par composants.* Une application est réalisée à partir de composants logiciels indépendants et exécutables qui sont vus l'un par l'autre comme des boîtes noires offrant des points d'entrées pour se composer les uns avec les autres ; chaque composant peut être développé avec le paradigme objet ou avec un autre.
- *L'approche dirigée par les modèles (MDA).* Le savoir-faire métier est décrit par des modèles indépendants du contexte d'utilisation (Platform Independent Model - PIM) c'est-à-dire indépendant de la plate-forme logicielle ou des traitements à réaliser. Les applications sont ensuite réalisées par transformation du modèle métier vers des plates-formes logicielles spécifiques (Platform Specific Model - PSM). Les deux principales idées sous-jacentes sont la capitalisation du savoir-faire et la prise en compte rapide de l'évolution de la technologie.

Ces idées clés participent à la mise en œuvre d'une nouvelle manière de développer des applications : le développement dirigé par les modèles (*Domain-Driven Development* ou *DDD*). Ce nouveau point de vue du développement du logiciel considère qu'une application s'appuie sur un modèle qui décrit un savoir-faire spécifique à un domaine donné et qui est réalisé indépendamment des applications futures. Le chapitre 6 présente d'abord un bref état de l'art sur les travaux en cours autour du DDD. Il décrit ensuite un certain nombre de règles destinées à poser le cadre de ce nouvel axe de recherche. Enfin il propose une description pas à

[29] Cela correspond au début de la période de ma délégation à l'INRIA.

[30] Un modèle métier répond à un problème ou matérialise un savoir-faire spécifique dans le cadre d'un domaine donné. On peut citer par exemple la gestion de la persistance pour des applications à objets ou la description d'interfaces graphiques ou textuelles pour des environnements de programmation.

pas de l'approche que nous proposons pour modéliser et implémenter des modèles métiers et leurs applications. Cette approche veut réutiliser le savoir-faire que nous avons acquis pour décrire la sémantique opérationnelle des concepts objets (chapitre 4), pour l'appliquer à la description de modèles métiers exécutables. Dans ce cadre, le modèle OFL (modélisation des concepts objets) devient un modèle métier particulier (parmi d'autres) qui permet de valider notre approche et d'offrir des exemples d'utilisation.

Dans le chapitre 7 et plus précisément dans la section 7.2, nous proposerons plusieurs perspectives pour approfondir et étendre cette approche.

Que deviennent les approches plus classiques ? Par rapport aux approches mentionnées ci-dessus, les travaux autour des langages à objets et plus généralement l'approche classique de développement se situent clairement dans la programmation du ou des composants. La conception et l'implémentation du contenu des composants restent bien sûr des aspects importants du développement d'application. Les modèles métiers ou extensions de langages sur lesquels nous travaillons à l'heure actuelle (voir la section 5.2) veulent favoriser la réutilisation et l'évolution de ce contenu. Ils montrent que nous nous intéressons plus que jamais à cette problématique ; ils s'appliquent indifféremment aux applications par objets classiques et à celles qui engendrent la création de composants. Les perspectives de recherche qui concernent cette partie de nos travaux sont décrites dans la section 7.1 et elles représentent pour l'instant notre priorité à court terme.

6. Collection de modèles métiers pour modéliser les modèles métiers

Pour reprendre ce qui a déjà été dit ci-dessus, un développement centré sur les modèles repose sur plusieurs paradigmes et il doit les combiner de manière originale afin d'avoir les meilleures qualités possibles pour assurer la modélisation de modèles métiers. Au début de ce chapitre, nous nous positionnerons par rapport à plusieurs approches importantes de l'état de l'art. Puis nous proposerons un ensemble de propriétés que nous jugeons essentielles pour la définition et la mise en œuvre d'une bonne approche DDD (section 6.1) et enfin nous présenterons les grandes lignes de notre solution (section 6.2).

Nos travaux se situent entre les approches dirigées par les modèles et la programmation orientée aspects appliquée aux langages dédiés à un domaine (*Domain-Specific Language* ou *DSL*). Plus généralement, nous nous rapprochons de la notion de fabrique logicielle qui inclut d'autres notions relatives à la conception d'applications comme les composants logiciels pour les applications distribuées.

Les travaux relatifs à la programmation orientée aspects ont pour objectif de proposer des mécanismes puissants pour décrire la sémantique des DSLs [34]. Tous ces travaux [197, 198, 253, 242] visent à l'obtention d'une meilleure séparation entre la structure de données et la sémantique des traitements associés.

Il est bien connu que le support d'une programmation orientée aspects peut être plutôt complexe et peut introduire des situations à peine contrôlables [47]. Pour résoudre ce problème, des langages orientés aspects dédiés à un contexte donné sont proposés [282]. Néanmoins, presque dans tous les cas, le mécanisme introduisant la réflexivité joue un rôle majeur [200, 205]. Par conséquent, de notre point de vue, il y a alors une dépendance étroite entre l'approche et les techniques d'implémentation (qui devraient être aussi peu visibles que possible au niveau du modèle). Les mécanismes réflexifs *i)* sont inefficaces et *ii)* imposent une importante dépendance avec la sémantique du langage cible ; les mécanismes réflexifs sont différents suivant l'implémentation du langage choisi.

Dans l'hypothèse où l'on choisit d'utiliser la programmation générative[1] pour mettre en œuvre le modèle, la programmation orientée aspects qui est associée est spécifique et n'a pas besoin de mécanismes réflexifs ; c'est la génération de code qui prend en compte les problèmes qui auraient été gérés par de tels mécanismes. De plus, dans ce contexte, la notion d'aspects est directement introduite au niveau du modèle et il est ainsi possible de réduire intentionnellement le nombre de points de coupe (*pointcut* en anglais), ce qui diminue la complexité.

Par rapport à des approches qui se concentrent sur des problèmes liés à la modularité ou à la réutilisation de la sémantique des composants [287, 34], L'approche qui est proposée dans

[1] La programmation générative est un paradigme centré sur la modélisation de familles de systèmes logiciels. Il permet, à partir d'un modèle qui représente le cahier des charges, de fabriquer automatiquement (au moyen de générateurs) des produits fortement personnalisés.

la section 6.2 est plutôt orientée vers la spécification d'une collection de modèles métiers ou de langages dédiés. D'autres approches s'intéressent sur la façon d'introduire des mécanismes puissants pour réutiliser les composants d'un langage ; leur objectif est d'être capable de concevoir un langage dédié à partir de la composition de composants existants [34].

Par rapport aux approches mentionnées, nous pensons qu'il est essentiel de proposer un métaniveau qui est indépendant des applications et qui contient la définition des différents concepts du modèle. Ce métaniveau permet la factorisation du savoir-faire relatif à un domaine donné qui est utilisé par différentes applications ou préoccupations. De plus, comme ce métaniveau est indépendant des applications, il est moins sensible aux changements structurels et pourrait être réutilisé dans d'autres contextes.

Par rapport aux approches par modèles, nous nous sentons plus proches de celles qui prennent en compte la spécificité d'un domaine [20] que des approches qui permettent de définir des extensions (*profiles* dans UML) d'un modèle standard universel. En effet, évoquer la réalisation d'un modèle universel ne semble plus être la piste privilégiée par l'OMG qui se tourne plutôt vers le MDA. De notre point de vue, des approches comme *Action Semantics* [294] semblent avoir les mêmes inconvénients puisque cette approche unifie dans un même cadre (la notation UML) des notions très différentes.

Par ailleurs, nous pensons qu'une bonne approche doit proposer des mécanismes simples pour décrire des traitements (par exemple la possibilité d'adapter des entités en fonction du domaine ou des squelettes d'actions sémantiques ou de préoccupations), basés sur l'utilisation de la sémantique décrite. Celle-ci peut être elle-même décrite à l'aide d'un langage de programmation ou d'un autre formalisme. En d'autres termes, nous pensons qu'il est important que le métaniveau autorise la description d'entités génériques qui favoriseront la factorisation de la sémantique.

Par rapport aux approches par transformation de modèle (MDA) [20], le mécanisme de génération que nous pensons qu'il faut mettre en œuvre est beaucoup plus complexe que ceux fournis par de simples transformations entre modèles. De notre point de vue, à cause de la pauvreté sémantique des modèles, une transformation au niveau du modèle [39, 20] ne réalisera jamais d'opérations complexes. Mais nous pensons qu'il est important de prévoir la possibilité de transformer un traitement décrit initialement pour un modèle donné pour qu'il puisse s'appliquer à un autre modèle cible [229]. D'ailleurs, nous défendons l'idée que la séparation entre le métaniveau et le niveau structurel des entités et la présence de mécanismes pour séparer de diverses manières les préoccupations doivent favoriser à la fois les transformations et la réutilisation.

Finalement, il semble intéressant que l'approche choisie à l'image de [127, 146, 86] soit intégrée dans un contexte plus global : les fabriques logicielles. SmartTools qui sert de support à la définition des modèles métiers est une fabrique logicielle et elle prend en compte beaucoup des concepts mis en évidence dans [146].

6.1. Systèmes centrés modèles : quelles propriétés ?

La programmation centrée sur les modèles est donc une nouvelle approche pour le développement de logiciels, qui s'appuie non seulement sur les paradigmes des objets et des aspects mais aussi sur des concepts mis en œuvre dans les systèmes d'information. La programmation orientée modèles introduit un nouveau niveau d'abstraction (le modèle) qui agit comme une

entité autonome qui peut recevoir des requêtes d'applications satellites. La spécification à la fois des modèles et des applications peut utiliser par exemple des approches orientées objets ou aspects.

Chaque application est construite autour d'un modèle métier. Nous nous intéressons au cas le plus fréquent où un modèle métier est prédominant. Ce cas est à rapprocher des applications construites en utilisant la programmation orientée aspects mais on rappelle que dans ce cas les préoccupations sont attachées au modèle métier et non pas tissées dans des applications qui peuvent être exécutées avec ou sans ces préoccupations[2]. Un modèle possède un aspect comportemental (une sémantique), mais il n'invoque pas lui-même un quelconque traitement spécifique à une application. Au contraire, la sémantique du modèle n'intervient que lorsqu'une application interroge le contenu du modèle métier dans le but de mettre en œuvre les traitements auxquels elle s'identifie. Dans le contexte du DDD, un modèle métier peut participer à la mise en œuvre de deux grandes catégories d'application , celles dédiées : *i)* au calcul et/ou à la mise à jour des informations contenues dans des instances de modèles ; elles se rapprochent de celles des systèmes d'information et, *ii)* à la transformation de modèles métiers ; elles sont particulièrement intéressantes dans le contexte de l'approche MDA.

La programmation orientée modèles diffère des autres techniques de programmation et en particulier de la programmation orientée objets (OOP). En particulier, elle va à l'encontre de la suprématie des langages de programmation : le modèle est maintenant le point central tandis que le formalisme utilisé pour décrire ses instances joue un rôle mineur. La composition de l'approche MDA et de la programmation générative favorise le couplage du modèle et du ou des formalisme(s), et donc l'apparition de langages spécifiques à un domaine (DSL) présentés comme des modèles/langages métiers.

Une nouvelle approche pour le développement de logiciel doit garantir que les propriétés mises en évidence dans le cadre du génie logiciel soient garanties et même améliorées par rapport aux approches par objets ou par aspects. La réutilisabilité, la capacité d'évolution et la robustesse à la fois des modèles métiers et des applications doivent être prises en compte soigneusement par la programmation orientée modèles. Pour atteindre cet objectif, nous proposons neuf règles qui concernent à la fois la conception et l'implémentation d'une approche orientée modèle.

6.1.1. Règles relatives à la conception

Règle N°1 : Un modèle métier est une entité de première classe pour le processus de développement. Un modèle métier repose sur un modèle de données et sur un modèle qui décrit sa sémantique. Le modèle de données contient la description des entités impliquées dans le modèle métier tandis que le modèle sémantique décrit les interactions et les contraintes qui existent entre ces entités et le comportement qui leur est associé dans le cadre du savoir-faire métier. Un modèle métier est considéré par les applications comme un tout ou pour son contenu. La notion de modèle constitue un niveau supplémentaire d'abstraction qui favorise les opérations globales telles les transformations ou l'introspection.

Règle N°2 : Une triple indépendance entre le modèle, l'application et la technologie. Un modèle métier n'est pas une application. Il encapsule la description de son comportement (sa

[2]On verra dans la section 6.2 qu'il est aussi possible d'ajouter des préoccupations à l'application elle-même.

sémantique), qui doit être indépendant d'une quelconque future application. Cette propriété garantit que le modèle métier est réutilisable indépendamment des applications qui peuvent l'utiliser[3]. De plus, une application ou un modèle métier doit être conçu indépendamment de la plate-forme logicielle sur laquelle l'application sera exécutée. C'est seulement le plus tard possible que l'association de la technologie utilisée par la plate-forme[4] doit être faite. Cette seconde propriété permet à la logique métier d'être utilisée quelles que soient les technologies qui peuvent apparaître dans le futur.

Règle N°3 : Support d'entités génériques. Typiquement, les modèles métiers doivent pouvoir adresser des lignes de produit et plus généralement des familles d'entités qui peuvent avoir des aspects en commun et des différences mais dont la sémantique est relativement proche ; Elles doivent être modélisées comme des entités génériques qui peuvent être facilement dérivées (instanciées). Une situation somme toute assez classique est que le modèle métier s'appuie sur un petit nombre d'entités-clés dont la définition repose sur des entités de base ; ces entités-clés peuvent souvent être vues comme des entités génériques. Ainsi il est particulièrement important que ces entités génériques offrent une vision claire de leur comportement car elles représentent elle-mêmes une partie significative de la sémantique du modèle. Un langage à objets comme Eiffel illustre bien l'intérêt de disposer d'un support d'entités génériques pour garantir une meilleure réutilisabilité et des capacités d'évolution plus importantes.

Règle N°4 : Une séparation claire entre la description de la sémantique et de la réification des entités. Le savoir-faire métier est encapsulé dans des modèles métiers à travers le modèle de données (réification et structuration des entités) et le modèle sémantique (comportement de ces entités). Pouvoir réutiliser la sémantique quand le modèle de données évolue est un objectif d'autant plus important que, dans le contexte de la transformation de modèles, la sémantique doit s'adapter à l'évolution du modèle de données (le plus automatiquement possible). La programmation orientée modèle doit donc fournir une séparation claire entre la description des modèles de données et sémantique.

Règle N°5 : Une approche orthogonale pour la prise en compte de préoccupations. La règle N°2 sous-entend une séparation des préoccupations entre le modèle métier et les applications. Le premier est sous la responsabilité d'un expert qui capture le savoir-faire métier, tandis que les secondes sont gérées par les programmeurs. Mais la séparation des préoccupations doit aussi exister à l'intérieur même du modèle métier et des applications.

Relativement au modèle métier, les besoins sont de deux ordres : *i)* la sémantique peut être suffisamment complexe et nécessiter plus de modularité, et *ii)* des bouts de sémantique qui sont orthogonaux à la sémantique de départ (par exemple relatifs à des services non fonctionnels), doivent pouvoir être mis en œuvre directement.

Par rapport aux applications, les besoins sont encore plus importants. Une application peut contenir plusieurs facettes (des *sujets* au sens de la programmation orientée sujets) qui doivent se composer simplement pour constituer l'application. De plus, une application doit être capable de s'adapter à l'évolution potentielle de l'environnement (qui ne peut pas être prévue à l'avance et en particulier pas au moment de la conception), sans que cela n'affecte le c œur de l'application.

[3]Naturellement il est aussi important de garantir la réutilisation du comportement d'une application.
[4]On appelle parfois ceci : le modèle dépendant de la plate-forme (*Platform Dependent Model* ou *PDM*).

6.1.2. Règles relatives à la mise en œuvre

Règle N°6 : Un équilibre subtil entre les programmations déclarative et impérative. La sémantique des modèles métiers doit être décrite autant que possible de manière déclarative afin de spécifier ce qui est demandé (le *« quoi »*) mais pas comment c'est réalisé (le *« comment »*). C'est un des objectifs les plus importants de l'approche MDA. Cependant, il n'est pas raisonnable de pousser cette approche jusqu'au point où la description du *«quoi»* aboutit à l'utilisation de formalismes difficiles à lire et à comprendre. Il est nécessaire de trouver un compromis entre le « *tout déclaratif* » et le « *tout impératif* ».

Règle N°7 : Support de langages métiers. Une distinction claire doit être faite entre l'expressivité d'un modèle métier et celle du langage (qu'il soit textuel, graphique, etc.) utilisé par le concepteur pour spécifier les différentes pièces du modèle métier. De plus la programmation orientée modèles tend à se rapprocher du grand public (programmation ubiquitaire), et donc le besoin d'offrir des langages dédiés à un modèle métier et même à une application devient de plus en plus crucial. La programmation générative combinée à l'approche MDA est un bon support pour atteindre cet objectif.

Règle N°8 : Ouverture du processus de développement. Fournir un métamodèle et un ensemble de mécanismes associés qui répondent aux besoins de n'importe quel modèle métier est selon nous une utopie. Nous défendons l'idée qu'il est préférable de proposer une approche unifiée avec très peu de mécanismes *« built-in »*, pouvant facilement être adaptés aux besoins que des applications modernes auront dans le futur. En particulier, il est important d'être capable de paramétrer la façon de rechercher l'information en fonction du contexte d'utilisation. En d'autres termes, la génération et la gestion d'un modèle métier exécutable doivent être paramétrables. Considérer chaque concept-clé du métamodèle comme des entités de première classe apparaît comme un point de départ intéressant pour l'ouverture du processus de développement.

Règle N°9 : Un métamodèle minimal avec des capacités d'auto-extension. Comme cela a déjà été mentionné dans plusieurs règles, la programmation orientée modèles nécessite un métamodèle qui permet de décrire à la fois le modèle métier et ses applications. Ce métamodèle peut être considéré comme un modèle métier particulier. La règle n°7 évoque le fait que la spécification des différentes parties de ce métamodèle peut reposer sur un langage dédié[5]. Cependant beaucoup d'autres besoins peuvent être nécessaires au développement d'applications. En particulier, les applications modernes doivent être disponibles comme des composants pouvant interagir les uns avec les autres. Il est donc important de rendre le métamodèle auto extensible, c'est-à-dire capable d'inclure d'autres applications ou modèles métiers construits à l'aide du métamodèle lui-même. Par exemple, pour prendre en compte la notion de composant d'application, on pourra par exemple concevoir un modèle métier dédié à cette problématique.

Les neufs règles proposées ci-dessus représentent une première tentative pour définir le cadre de la programmation orientée modèles. Nous défendons l'idée qu'il faut suivre ces règles le plus possible si on veut mettre en œuvre une programmation orientée modèles. Dans la section suivante, nous présentons les grandes lignes de notre approche.

[5]Il peut être construit comme un pseudo-langage ou il peut utiliser l'approche graphique d'UML relative aux diagrammes de classes ou d'activités, à *Action Semantics*, etc.

6.2. SmartModels : Un métamodèle pour modèles métiers

Il est maintenant temps de poser les fondements de notre approche qui s'intègre parfaitement dans l'axe de recherche *DDD*. Celui-ci s'appuie en particulier sur *i)* les langages à objets et à composants, *ii)* l'approche MDA, *iii)* la séparation des préoccupations, et *iv)* la programmation générative. Nous appelons le style de programmation associé : la *programmation orientée modèles*[6].

Cette approche s'appuie d'une part, sur un métaniveau, pour identifier clairement la sémantique des concepts utilisés dans la modélisation d'un métier particulier, et d'autre part, sur les approches par séparation de préoccupation et sur la programmation générative pour permettre une instrumentation modulaire des applications associées au métier (ou domaine) considéré. Cette approche est originale et se différencie des autres approches sur plusieurs points : *i)* elle associe aux entités qui structurent un modèle (entités de réification), un niveau sémantique qui permet de préciser et de factoriser les fonctionnalités de base du domaine, *ii)* elle permet une instrumentation (conception rapide des applications associées au modèle) qui s'appuie fortement sur les deux niveaux du modèle (niveau structurant et sémantique), *iii)* elle assure une séparation nette entre la modélisation (du métier), et les technologies utilisées pour rendre le modèle exécutable (c'est-à-dire implémenter les applications pour ce modèle).

L'intérêt de notre approche (appelée SmartModels), réside dans le fait qu'elle propose un moyen de décrire la sémantique des entités utilisées dans un modèle qui est indépendent d'une application donnée. En général, la sémantique d'un modèle est dispersée dans les diverses applications qui le manipulent. Notre approche ne fait pas de différence entre la modélisation du métier et son instrumentation[7]. En effet, puisque la sémantique du modèle est associée aux entités elles-mêmes, les diverses applications peuvent utiliser directement cette connaissance sans passer par une phase d'implémentation[8]. Les apports de la programmation générative et par séparation des préoccupations sont utilisés pour permettre à la partie « instrumentation du modèle », une expressivité, une capacité d'évolution et une modularité accrues. SmartModels réutilise nos travaux sur OFL. Parmi les aspects importants réutilisés, on trouve en particulier :

- la capacité de décrire n'importe quel élément ou entité d'un modèle de données,
- la possibilité de décrire des paramètres métiers qui permettent de capturer une partie de la sémantique du modèle,
- la modélisation de concepts génériques (la généricité est décrite par les paramètres métiers), permettant par exemple de représenter des « lignes de produits »,
- la spécification d'actions sémantiques associées aux concepts et aux entités du modèle, dont l'exécution est dirigée par des paramètres métiers,
- un support pour la définition de méta-assertions,
- un protocole métaobjets permettant de mettre en œuvre tous les aspects importants du modèle (actions, paramètres, préoccupations, assertions, etc.) comme des entités de première classe du modèle et d'offrir automatiquement à chaque entité un ensemble de services comme par exemple la gestion de ses instances.

Comme on le verra dans la section 6.3, la mise en œuvre de SmartModels dans un pre-

[6] En anglais, *Model-Oriented Programming*.
[7] Les fonctionnalités ou la glue qui rend le modèle exécutable.
[8] Ce sont les générateurs qui, en proposant une implémentation générique des concepts du modèle, prendront en compte cette phase d'implémentation.

mier prototype de « fabrique logicielle », que nous avons baptisé SmartFactory[9], s'inspire naturellement largement des idées mises en œuvre dans SmartTools [88, 87] comme par exemple *i)* les techniques de génération de code permettant d'implanter les modèles décrits avec SmartModels sur des plates-formes logicielles, *ii)* la description de traitements par visiteurs et par aspects, et *iii)* les techniques de transformations de modèles, notamment vers les technologies issues du *W3C*.

6.2.1. Description des principaux éléments de l'approche

Deux des aspects fondamentaux de SmartModels sont relatifs *i)* à la sémantique des entités du modèle métier (commune à l'ensemble de leurs instances), qui est encapsulée dans un métaniveau, de manière à ce qu'il soit facile de la distinguer de la réification des instances de ces mêmes entités et, *ii)* aux familles d'entités qui peuvent être spécifiées facilement, grâce à l'existence dans le métamodèle de la notion d'entités génériques (voir la figure 6.1).

Avant d'aller plus avant, il est bon de nous positionner par rapport aux standards comme MOF ou UML. Que dire d'utiliser MOF directement pour construire des modèles métiers ? MOF [232, 240] possède les principales entités nécessaires pour réifier un modèle métier mais il n'offre pas la notion de classe paramétrée (ou générique), ni la notion de métaclasse (dont les instances seraient des classes)[10]. UML possède ces différentes notions et pourrait parfaitement être utilisé mais il faudrait pouvoir (par la création d'un profil) empêcher l'utilisation de certains éléments, en définir certains autres et proposer un guidage du concepteur de modèles[11].

Il est par contre tout à fait possible de créer SmartModels en utilisant MOF ou UML. Nous avons d'ailleurs choisi d'expliquer les différentes parties de notre métamodèle en utilisant UML, même si SmartTools que nous avons utilisé pour implémenter SmartModels utilise le langage *absynt*.

La figure 6.1 décrit les grandes lignes de l'architecture du métamodèle. La sémantique d'un modèle métier est prise en compte notamment à travers la spécification de *paramètres hyper-génériques* [106], de *caractéristiques,* d'*assertions* et d'*actions*. Tous participent à la définition des entités d'un modèle métier[12] (dans le cas d'entités non génériques seules les assertions et les actions sont considérées), mais ils ne concernent pas la description de leurs instances. Parce que les applications sont décrites en dehors des modèles métiers, les méthodes qui sont relatives à la gestion des instances des atomes sont seulement des accesseurs[13] ; ils peuvent être générés automatiquement en tenant compte du type (par exemple, si c'est une collection ou pas)[14]. Les *actions* sont des méthodes qui ont des propriétés particulières ; par exemple, elles peuvent prendre en compte la définition d'assertions (voir la section 6.2.2), d'aspects (voir la section 6.2.3). On notera que les actions sont des entités de première classe.

Le métaniveau d'une entité est encapsulé dans un *concept* qui peut être associé à un ou plusieurs atomes[15]. Cette séparation claire entre les parties « sémantique » et « réification »

[9] Un premier prototype partiel est déjà opérationnel. Il permet, entre autres, de générer la plupart des classes de la plate-forme OFL/J qui était au préalable écrites à la main.

[10] Les principales facilités fournies par MOF pour décrire des métainformations sont les variables et les méthodes de classe.

[11] La présence d'entités définies à des niveaux différents (méta ou non) rend l'utilisation d'UML très complexe.

[12] Nous les appelons *atom* - voir la section 6.2.2.

[13] C'est la principale différence avec les actions qui concernent les entités mais pas leurs instances.

[14] Cette facilité est offerte par l'entité qui encapsule le modèle métier (voir la section 6.2.3).

[15] Cest une approche très similaire à celle mise en œuvre par les classes et métaclasses du langage Smalltalk.

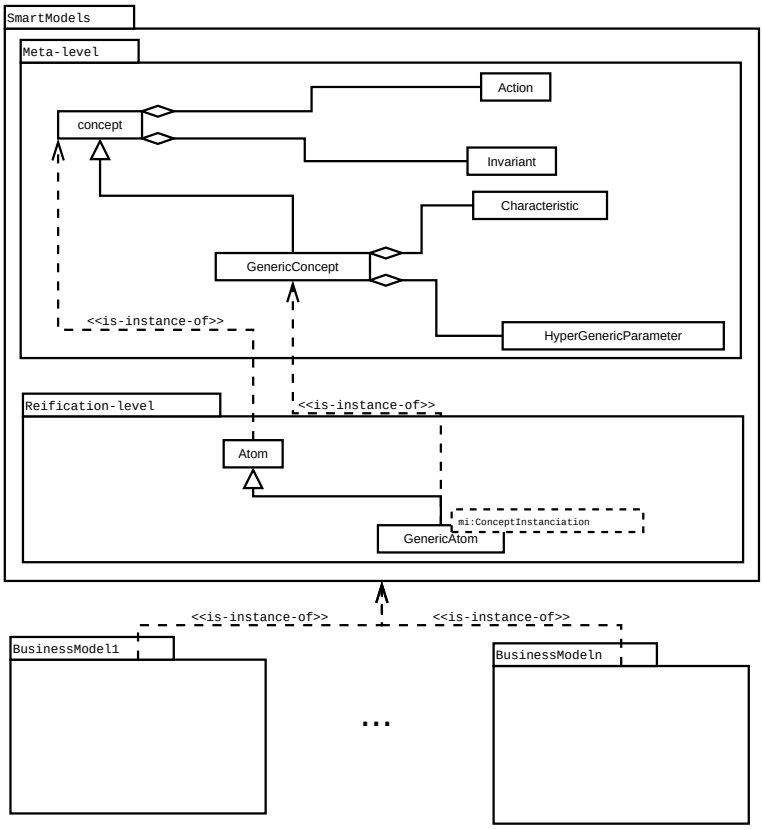

Fig. 6.1.: Principaux éléments de SmartModels

```
Formalism of SmartModels is
    Root is Top;
    Operator and type definitions {
    Top = businessModel(Name name,
                        GeneralInformation generalInformation,
                        Concept[] concept,
                        GenericConcept[] genericConcept,
                        Atom[] atom,
                        GenericAtom[] genericAtom,
                        ConceptInstantiation[] ConceptInstanciation,
                        DerivedGenericAtom[] derivedGenericAtom,
                        ParameterDefinition[] parameterDefinition,
                        CharacteristicDefinition[] characteristicDef;
                        Action[] action;
                        ...) // etc
    }
    Attribute definitions { ... } // not shown
```

Fig. 6.2.: Racine de la description de SmartModels avec le langage *absynt*

de ces entités est très importante car elle favorise *i)* la maintenance de la sémantique (la redéfinition d'un aspect sémantique ne devrait concerner que les *concepts*), *ii)* la réutilisation de la sémantique dans d'autres modèles métiers (proches), et *iii)* la transformation de modèles.

SmartFactory qui est une première implémentation de SmartModels a été réalisé avec SmartTools [16] et donc le langage qui a été utilisé pour le décrire est *absynt*. La figure 6.2 propose avec ce formalisme un bref aperçu des informations déjà proposées dans la figure 6.1 en utilisant UML[17] .

6.2.2. Spécification d'un modèle métier pas à pas

Nous proposons ici une visite guidée de SmartModels qui débute par deux points de vues macro de ce métamodèle et qui sera poursuivie par une succession de points de vue qui portent chacun sur une partie du métamodèle. Concrètement, nous nous plaçons dans l'hypothèse où nous désirons modéliser avec SmartModels un modèle métier (*OFL*) pour décrire les concepts mis en œuvre par les langages à objets. Les applications que l'on peut envisager de réaliser en s'appuyant sur le modèle concernent par exemple la mise en œuvre de fonctionnalités relatives aux environnements de programmation (métriques, assistants divers ou éditeurs spécialisés, etc.). Parmi les entités qui pourraient participer à la description du modèle, on peut citer par exemple, les *attribut*s, les *méthod*es, les *paramètres de méthodes*, les *modifieurs*, etc. Mais les plus intéressantes sont sûrement celles qui participent à la description des différentes sortes de *classifieurs* (classes abstraites ou non, interfaces, etc.) et des *relations* (telle l'agrégation ou l'héritage).

La figure 6.3 propose sans les détailler (cela sera fait par la suite), quelques-uns des éléments qui interviennent dans la description du modèle métier *OFL*. Le formalisme utilisé pour présenter ce modèle métier, c'est-à-dire une instance de notre métamodèle, est le langage XML, qui est le mécanisme de base de SmartTools pour exprimer une instance du modèle de données.

La figure 6.3 montre donc quelques-unes des entités qui interviennent dans la modélisation d'OFL ; la description de ces entités sera montrée dans les prochains paragraphes, au fur et

[16] Nous montrerons dans la section 6.3 une vision plus détaillée de cette impléméntation.

[17] Par souci de simplicité, certaines informations comme l'instanciation de métainformation ne sont pas présentées.

```
<BusinessModel>
  <name>ObjectModel</name>
  <generalInformation> ...<version>0.1</version> ... </generalInformation>
  <concept> <name>ConceptAttribute</name> ... </concept>
  <genericConcept> <name>ConceptClassifier</name> ... </genericConcept>
  <genericConcept> <name>ConceptRelationship</name> ... </genericConcept>
  <genericConcept> <name>ConceptImportRelationship</name> ... </genericConcept>
  <genericConcept> <name>UseRelationship</name> ... </genericConcept>
  <atom> <name>AtomFeature</name> ... </atom>
  <atom> <name>AtomAttribute</name> ... </atom>
  <atom> <name>AtomMethod</name> ...  </atom>
  <genericAtom> <name>AtomClassifier</name> ... </genericAtom>
  <genericAtom> <name>AtomImportRelationship</name> ... </genericAtom>
  <conceptInstanciation> <name>ClassifierJavaClass</name> ... </conceptInstanciation>
  <conceptInstanciation> <name>ImportRelationshipExtendClass</name> ... </conceptInstanciation>
  <derivedAtom> <name>ClassifierJavaClass</name> ... </derivedAtom>
  <derivedAtom> <name>ImportRelationshipExtendClass</name> ... </derivedAtom>
  <parameterDefinition> <name>PolimorphismDirection</name> ... </parameterDefinition>
  <characteristicDefinition> <name>ValidSourceRelationships</name>...</characteristicDefinition>
  <characteristicDefinition> <name>ValidSourceClassifiers</name> ... </characteristicDefinition>
</BusinessModel>
```

FIG. 6.3.: Quelques éléments du modèle métier *OFL*.

à mesure du développement du modèle métier. Pour la réification des instances des entités, il s'agit des classifieurs (*AtomClassifier*), de leurs primitives (*AtomFeature*) et plus précisément des attributs (*AtomAttribute*) et des méthodes (*AtomMethod*), et enfin des relations d'importation, assimilées à l'héritage entre deux classifieurs (*AtomImportRelationship*). Pour la modélisation du savoir-faire métier, il s'agit de la définition d'un paramètre des relations d'importation qui définit le sens du polymorphisme (*Polymorphism*Direction) dans le modèle métier, de la caractéristique (*ValidSourceClassifiers*) qui mémorise l'ensemble des sortes de classifieurs (interface, classe, etc.) acceptés comme source d'une relation d'importation[18]. Pour la réification du niveau méta des entités, les concepts associés aux classifieurs (*ConceptClassifier*), ou aux relations entre classes (*ConceptRelationship*) et plus particulièrement les relations d'importation (*ConceptImportRelationship*), permettent de regrouper les paramètres et les caractéristiques propres à chaque concept.

Definition des entités de base du modèle La figure 6.4 propose une descripition UML des principaux éléments de SmartModels qui permettent de décrire les entités de base d'un modèle métier. En regardant cette figure, on constate que ce que nous proposons pour décrire le modèle métier repose sur des notions bien connues que l'on trouve aussi bien dans les langages de programmation que les méthodes de conception. À première vue, les entités proposées par SMARTMODELS sont très proches de MOF [232], mais, en fait, il y a des différences que nous mentionnons au fur et à mesure des explications.

Dans le métamodèle proposé, l'élément *Atom* (ou *GenericAtom*) est la structure qui supporte la description d'une entité ; c'est très proche de la notion de *class* dans MOF[19]. De plus les primitives fournies par MOF pour décrire le contenu d'une classe (comme par exemple les *attributs*, les *opérations*, les relations de *généralisation*) sont suffisantes pour définir la majeure partie de la réification d'une entité et le lecteur les retrouvera naturellement dans la figure

[18]Le modèle OFL [95, 93] propose, pour définir la notion de relations, une trentaine de paramètres et une dizaine de caractéristiques.

[19]Le concept de *class* fait selon nous trop référence aux langages de programmation alors qu'il faudrait des concepts plus abstraits pour les modèles métiers.

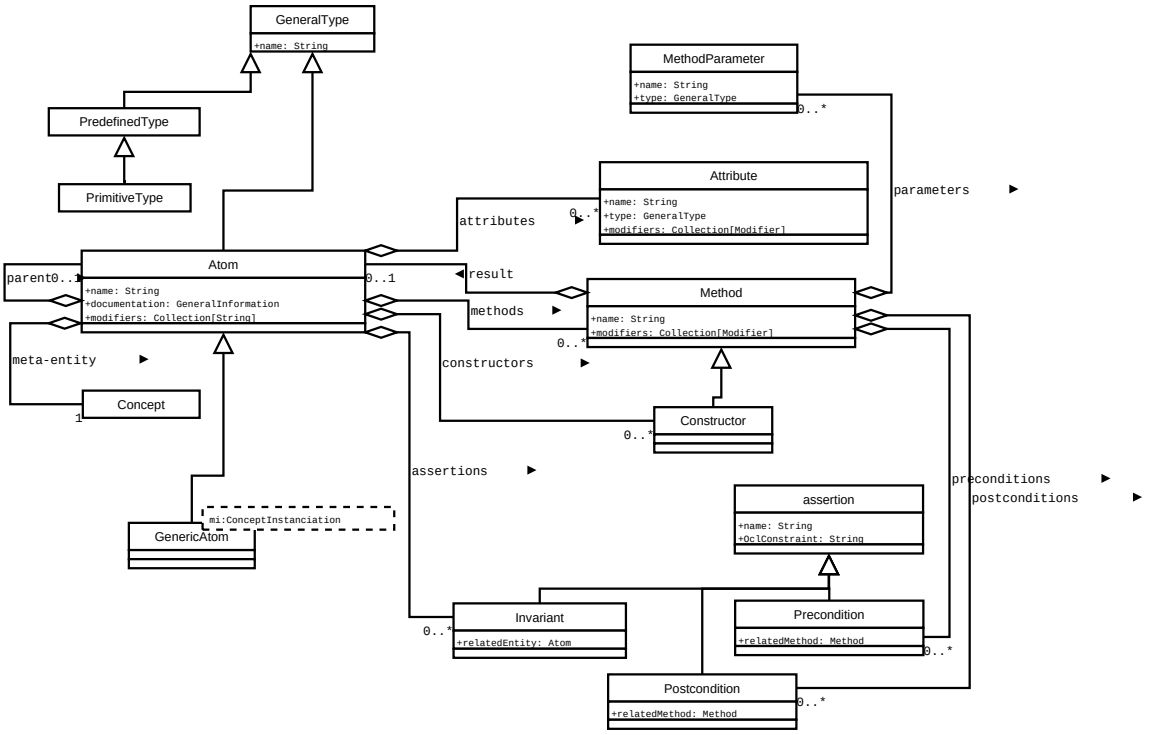

Fig. 6.4.: Description de la structure des entités dans SmartModels

6.4. Cette figure montre aussi que nous incluons la possibilité d'attacher des invariants, des préconditions et des postconditions à un modèle métier ; c'est une des facilités proposées par SmartModels pour contribuer à la description de sa sémantique. Les assertions s'appuient sur OCL [230, 235].

Le concepteur d'un modèle métier peut créer des atomes dans le but soit d'améliorer la structuration et la factorisation de l'information à l'intérieur du modèle (par exemple : *AtomFeature*), soit pour décrire des atomes dont des instances pourront être créées par les applications (par exemple : *AtomAttribute*, *AtomMethod*, etc.). Notre métamodèle permet de répondre à chacun de ces besoins ; MOF le fait à travers la propriété qui spécifie si une *classe est abstraite* ou non. Si cela signifie que la classe doit avoir au moins une primitive abstraite ou que toutes ses primitives doivent être abstraites, alors nous croyons que ce mécanisme est insuffisant. En particulier, les applications peuvent ne pas être intéressées par les mêmes atomes, ce qui est différent de ne pas pouvoir avoir d'instances (quelle que soit l'application), parce que l'atome n'est que partiellement défini.

Nous pensons qu'il faut *i)* fournir des informations supplémentaires pour mieux spécifier une entité, c'est la raison d'être de l'attribut *modifiers*[20] dans *Atom* et, *ii)* permettre au moment de la définition des applications sur un modèle métier, de ne considérer que certaines entités et de leur permettre ou pas de créer des instances.

La figure 6.5 propose une réification partielle d'une des entités de base que l'on peut trouver

[20]Il est défini pour simplifier comme une collection de chaînes de caractères.

```
<genericAtom> <name>classifier</name>
   <attribute  <name>name</name>
      <listOfModifiers> <identifier>private</identifier> ...
      <primitiveType>   <string></string>  </primitiveType> ... </attribute>
   <attribute> <name>isGeneric</name>
     <listOfModifiers> <identifier>private</identifier> ...
     <primitiveType>  <bool></bool>  </primitiveType> ... </attribute>
   <attribute> <name>attributes</name>
       <listOfModifiers> <identifier>private</identifier>...
       <collectionType>AtomAttribute</collectionType> ... </attribute>
    <attribute> <name>methods</name>
       <listOfModifiers> <identifier>private</identifier>...
       <collectionType>AtomMethod</collectionType> ... </attribute>
    <attribute> <name>qualifiers</name>
       <listOfModifiers> <identifier>private</identifier>...
       <collectionType>AtomQualifier</collectionType> ... </attribute>
    <attribute> <name>formalGenericParameters</name>
       <listOfModifiers> <identifier>private</identifier>...
       <collectionType>AtomType</collectionType> ... </attribute>
    <attribute> <name>sourceRelationships</name>
       <listOfModifiers> <identifier>private</identifier> ...
       <collectionType>AtomImportRelationship</collectionType> ... </attribute>
    <attribute> <name>targetRelationships</name>
       <listOfModifiers> <identifier>private</identifier>....
       <collectionType>AtomImportRelationship</collectionType> ... </attribute>
    <method> <name>changeName</name>
       <listOfModifiers> <identifier>private</identifier>...
       <returnType> <primitiveType>  <void></void>  </primitiveType> </returnType>
       <methodParameter> <name>newName</name>
          <primitiveType>  <string></string>  </primitiveType> </methodParameter>...</method>
    <invariant> <name>Invariant1</name>
      <ocl_Constraint>  assertion in OCL  </ocl_Constraint>
      <parentEntity></parentEntity> </invariant>
    <parentEntity></parentEntity>
    <metaEntity>ConceptClassifier</metaEntity>
    <listOfModifiers> <identifier>public</identifier>...
</genericAtom>
```

FIG. 6.5.: Quelques éléments de la structure des entités pour OFL

dans les langages à objets. Il s'agit de la notion de classifieur au sens d'UML (l'atome concerné est *AtomClassifier*). Il possède les propriétés suivantes : un nom (*name*), la propriété d'être générique ou pas (*isGeneric*), une liste d'attributs (*attributes*) et une autre de méthodes (*methods*), une collection de modifieurs (*qualifiers*) et une autre qui contient le cas échéant les paramètres formels (*formalGenericParameters*), et enfin la liste des relations d'importation qui arrivent ou qui partent du classifieur (*targetRelationships* et *sourceRelationships*). On peut noter que les informations présentes dans *AtomClassifier* évoque l'existence d'autres atomes qui ne sont pas décrits ici ; il s'agit entre autres de *Attribute*, *Method*, *Qualifier*, *Type*, *ImportRelationship*. *AtomClassifier* définit aussi *i)* une méthode, donnée là encore à titre d'exemple (*changeName*), qui permet de comprendre pourquoi la propriété *name* est marquée comme privée (*private*) et, *ii)* un invariant (*invariant1*) qui est censé participer à la définition de la sémantique d'un classifieur.

Les trois informations suivantes précédées des étiquettes XML *<parentEntity>*, *<metaEntity>* et *<listOfModifiers>* spécifient respectivement que *i) Classifier* n'est pas dérivé d'un autre atome (voir page 113 pour plus de détails à propos des atomes génériques), *ii)* le concept appelé *ConceptClassifier* représente la partie méta de toute instance de l'atome *AtomClassifier*[21] et, *iii)* une liste d'informations qui explique un peu mieux le statut de cet atome dans le modèle métier. La spécification d'un atome parent pourra être acompagnée d'une clause d'adaptation qui permet de spécifier qu'un attribut ou une opération est redéfini ou renommé (cette facilité n'est pas explicitée ici). Il faut cependant garder à l'esprit que le mécanisme doit rester simple pour des questions d'échelle. En effet, il ne semble pas que la modélisation de modèles métiers ait les mêmes besoins que le développement d'applications impliquant plusieurs centaines de classes et plusieurs bibliothèques généralistes développées par des tiers.

Définition des métainformations de SmartModels Un aspect important est de capturer la sémantique d'un modèle métier. Pour ce faire, nous avons déjà mentionné la définition d'assertion mais nous désirons aller plus loin. En particulier, nous avons mis en évidence que les modèles métiers comportent souvent des entités pour lesquelles il existe des variantes. UML propose des solutions mais elles ne nous semblent pas assez complètes. Nous proposons de considérer ces entités comme des entités génériques dont la généricité s'appuierait sur les notions de paramètres et de caractéristiques qui seront abordées ci-dessous. L'encapsulation de ces informations dans la notion d'atome générique (*GenericAtom*) sera quant à elle discutée plus loin, page 113. Un paramètre permet de mémoriser une propriété qui définit plusieurs comportements possibles selon la sémantique que l'on veut associer à un atome ; naturellement cela n'est utile que si l'on veut pouvoir proposer plusieurs variantes du même atome. Pour définir un paramètre, on dispose des types de base et des types prédéfinis permettant de définir des tuples, des collections et des types énumérés ; tous ces types peuvent être composés. Une caractéristique permet de mémoriser une information relative aux atomes et aux concepts définis dans le modèle métier lui-même ; il est possible d'utiliser les types prédéfinis mentionnés ci-dessus pour réaliser des combinaisons plus complexes.

Nous proposons dans la figure 6.7 la définition d'un paramètre et d'une caractéristique tous deux choisis parmi ceux du modèle OFL. Le paramètre indique de quelle manière est paramétré le polymorphisme d'une relation d'importation tandis que la caractéristique mémorise les types de classifieur qui sont acceptés comme source d'une relation entre classes. Il est à noter qu'à ce stade de la définition rien ne dit que le paramètre et la caractéristique décrits sont associés

[21] Il sera évoqué dans les sections 6.2.2 et 6.6.

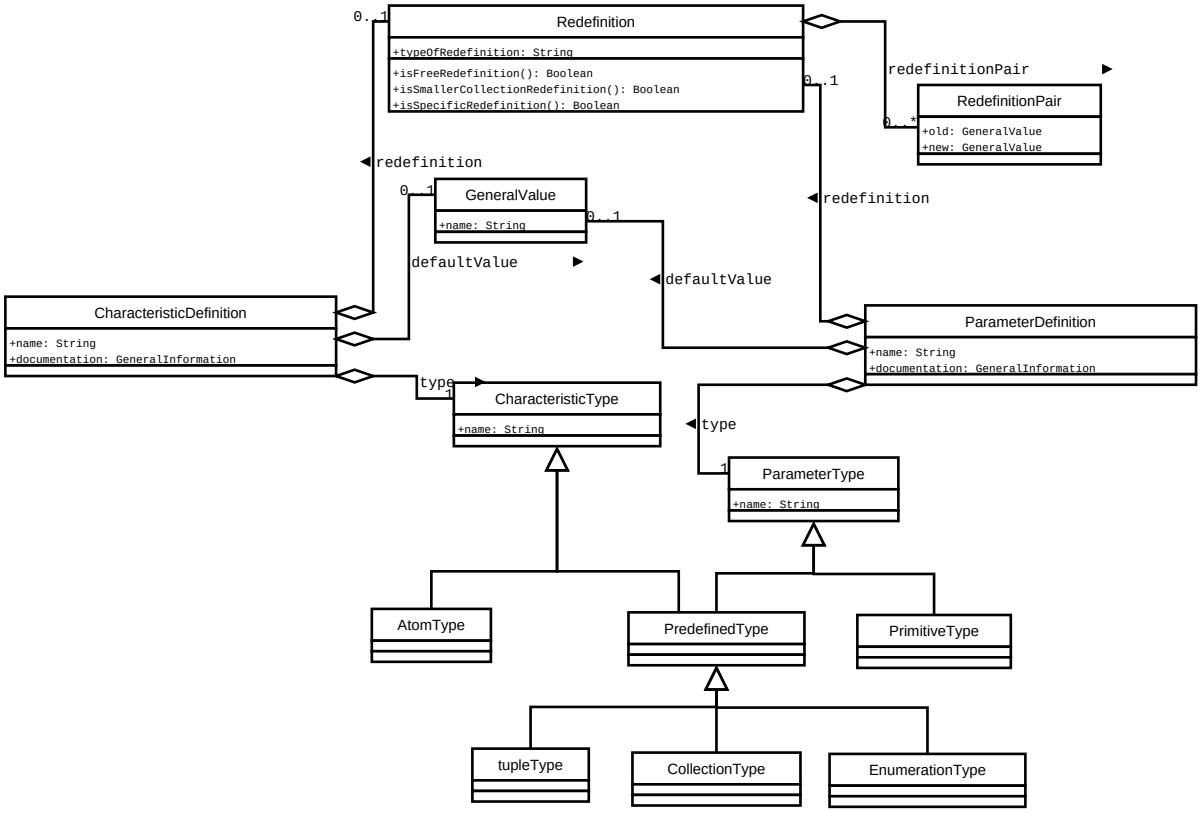

FIG. 6.6.: Définition des métainformations

```
<parameterDefinition> <name>Polymorphism_direction</name>
  <enumerationType> <identifier>none</identifier> <identifier>up</identifier>
                    <identifier>down</identifier> <identifier>both</identifier>
  </enumerationType>
  <defaultValue> <identifier>up</identifier> </defaultValue>
  <redefinitionPairs>
      <pairs> <oldParValue> <identifier>both</identifier> </oldParValue>
              <newParValue> <identifier>up</identifier> </newParValue>    </pairs>
      <pairs> <oldParValue> <identifier>both</identifier> </oldParValue>
              <newParValue> <identifier>down</identifier> </newParValue> </pairs>
      ...
      <pairs> <oldParValue> <identifier>up</identifier> </oldParValue>
              <newParValue> <identifier>none</identifier> </newParValue> </pairs>
  </redefinitionPairs>
</parameterDefinition>
<characteristicDefinition> <name>ValidSourceRelationships</name>
   <collectionType>ConceptImportRelationship</collectionType>
   <defaultValue> <identifier></identifier> </defaultValue>
   <redefinitionSmallerCollection></redefinitionSmallerCollection>
</characteristicDefinition>
```

Fig. 6.7.: Exemples de métaentités pour OFL

au niveau méta (concept) d'une relation entre classifieurs. Parmi les informations mémorisées, on note la possibilité de redéfinir un paramètre (les règles de redéfinition sont définies dans le paramètre lui-même). Le paramètre *Polymorphism_ direction* prévoit quatre comportements possibles en fonction de la relation d'importation et des redéfinitions (voir section 4.2)[22]. La caractéristique proposée dans l'exemple mémorise une collection d'instanciation du concept de relation d'importation (*ConceptImportRelationship*). On verra à partir de la page 113 que cette collection représente l'ensemble des types de relation d'importation qui sont acceptés à partir d'un type de classifieur donné. Dans cet exemple, la collection n'a pas de valeur par défaut et elle peut éventuellement être redéfinie par une collection plus petite.

Définition des entités génériques Cette section a pour objectif de décrire l'architecture du niveau méta d'une entité modélisable par SmartModels et en particulier les entités dites génériques (voir à partir de la page 111).

Le support d'entités génériques (*generic atoms*) est une fonctionnalité importante pour la modélisation des modèles métiers. Prenons l'exemple du modèle OFL. Nous avons encapsulé la plupart de la sémantique à l'intérieur des classifieurs et des relations (d'héritage et d'agrégation), tandis que d'autres atomes tels *AtomAttribute*, *AtomMethod* ou *AtomMethodParameter* ont une sémantique minimale limitée pratiquement à leur réification. Le reste de leur sémantique dépend des classifieurs et des relations et elle est donc prise en compte par eux. À titre d'exemple, on peut citer les classifieurs (*classes java*, *classes internes*, *interfaces*, etc.) et les relations (*héritage entre interfaces, héritage entre classes, implémentation d'interface par une classe*, etc.), mises en œuvre dans Java [60]. Il semble tout à fait cohérent comme c'est le cas dans OFL, de les modéliser à l'aide d'un atome générique[23].
Définissons maintenant le terme *atome générique* (voir page 111). La généricité provient d'un ensemble de *paramètres hyper-génériques* et d'un ensemble de *caractéristiques* qui mémorisent les points communs et les différences entre les entités qui peuvent être dérivées à partir de

[22] Toutes les combinaisons ne sont pas décrites dans la figure 6.7.
[23] Une entité générique pour les classifieurs, une pour les relations d'héritage et une pour les relations d'agrégation.

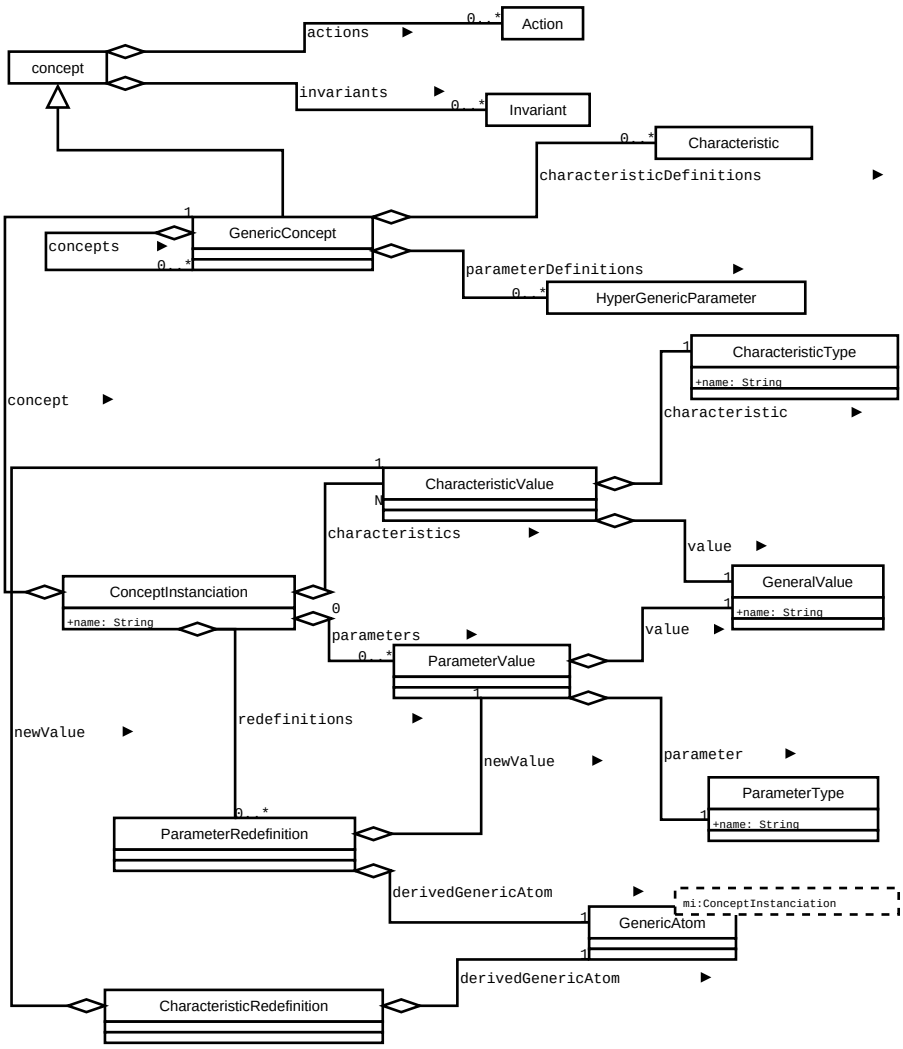

FIG. 6.8.: Définition des concepts

```
<genericConcept> <name>ConceptClassifier</name>
   <parameterDefinition>Genericity</parameterDefinition>
   <parameterDefinition>Generator</parameterDefinition>
   <parameterDefinition>ExtensionCreation</parameterDefinition> ...
   <characteristicDefinition>ValidSourceRelationships</characteristicDefinition> ...
   <invariant> <name>Invariant1</name>
      <ocl_Constraint> Assertion in OCL </ocl_Constraint> <parentEntity></parentEntity>
</invariant>
<parentEntity></parentEntity>
<concepts></concepts>
</genericConcept>
...
<genericConcept> <name>ConceptImportRelationship</name>
   <parameterDefinition>PolymorphismDirection</parameterDefinition> ...
   <characteristicDefinition>ValidSourceclassifiers</characteristicDefinition> ...
   <parentEntity>ConceptRelationship</parentEntity>
   <concepts>ConceptClassifier</concepts>
</genericConcept>
```

Fig. 6.9.: Exemples partiels de concepts pour OFL

ces paramètres[24]. Les atomes dérivés sont obtenus en choisissant une combinaison adaptée de valeurs pour les caractéristiques et les paramètres qui participent à la définition de l'*atome générique*.

Nous avons choisi d'utiliser la généricité plutôt qu'une relation d'héritage pour modéliser ces atomes pour plusieurs raisons : *i)* la définition du modèle de données (de la réification des instances) n'est pas mélangé avec la définition de la sémantique ; cela améliore la capacité de cette dernière à être transformée et réutilisée dans un modèle où la réification des instances est différente ; *ii)* une partie significative de la sémantique de toutes les entités dérivées d'une entité générique est décrite à un seul endroit, par la définition de ses paramètres et de ses caractéristiques ; cela favorise à la fois la réutilisation et la maintenance de ces entités, *iii)* la réutilisation du modèle métier est améliorée notamment lorsque le nouveau modèle étend celui d'origine. Par exemple, si on veut étendre le modèle métier qui décrit le langage Java en lui ajoutant une relation d'héritage inverse (voir la section 5.2.3), il suffit que le nouveau modèle propose une nouvelle instanciation de l'entité générique qui décrit les relations d'héritage.

On notera que pour chaque concept il est possible de spécifier une collection de concepts (*concepts*) qui peuvent redéfinir ses paramètres et ses caractéritisques en s'appuyant sur les règles mentionnées lors de la définition des paramètres (voir page 111). Ainsi, lors de l'instanciation d'un concept, il sera possible (*redefinitions*) de changer la valeur d'un paramètre ou d'une caractéristique pour un atome générique (instancié) dont le concept est présent dans cette collection.

La figure 6.9 propose deux concepts relatifs qui décrivent respectivement la sémantique d'un classifieur et d'une relation d'importation. On retrouve notamment ici certains paramètres (*Genericity*, *Generator*, *ExtensionCreation*) et une caractéristique (*ValidSourceRelationships*) du *ConceptClassifier*[25]. De même que pour un atome, un concept doit être placé dans le graphe d'héritage du modèle métier : ici *ConceptClassifier* ne spécialise aucun autre concept tandis que *ConceptImportRelationships* étend *ConceptRelationships*.

La figure 6.10 décrit une instanciation possible de *ConceptClassifier* et *ConceptImportRe-*

[24] « dérivé » est le terme qui est souvent employé dans la littérature pour évoquer les instances d'entités génériques.

[25] Pour plus de détails sur les paramètres et caractéristiques associés à cette sémantique voir le chapitre 4.

```
<ConceptInstanciation> <name>ClassifierJavaClass</name>
    <listOfModifiers> <identifier>public</identifier> </listOfModifiers>
    <parameterValue>  <primitiveType> <string>Genericity</string> </primitiveType>
                      <bool>FALSE</bool> </parameterValue>
    <parameterValue>  <primitiveType> <string>Generator</string> </primitiveType>
                      <bool>TRUE</bool> </parameterValue>  ...
    <relatedConcept>ConceptClassifier</relatedConcept>
    <redefinitions>ConceptClassifier</redefinitions>
</ConceptInstanciation>
<ConceptInstanciation> <name>RelationshipExtendClass</name>
     <listOfModifiers> <identifier>public</identifier> </listOfModifiers>
     <parameterValue>  <primitiveType> <string>PolymorphismDirection</string> </primitiveType>
                       <identifier>up</identifier> </parameterValue>
     <characteristicValue> <collectionType>ValidSourceClassifiers<collectionType>
                           <atom>ClassifierJavaClass</atom>
                           <atom>ClassifierAbstractJavaClass</atom> </characteristicValue> ...
     <relatedConcept>ConceptImportRelationship</relatedConcept>
    <redefinitions>ConceptClassifier</redefinitions>
</ConceptInstanciation>
<derivedGenericAtom> <name>ClassifierJavaClass</name>
    <relatedAtom>AtomClassifier</relatedAtom>
    <genericParameter>ClassifierJavaClass</genericParameter> </derivedGenericAtom>
<derivedGenericAtom> <name>ImportRelationshipExtendClass</name>
    <relatedAtom>AtomImportRelationship</relatedAtom>
    <genericParameter>RelationshipExtendClass</genericParameter>
</derivedGenericAtom>
```

FIG. 6.10.: Exemples partiels de l'instanciation d'atome générique pour OFL

lationships. Chacune de ces instanciations indique à quel concept elle s'applique et de quel atome elle prend la réification de ses instances. La première instanciation (*ClassifierJavaClass*) concerne les classes Java non abstraites et donc les paramètres présentés indiquent que le classifieur n'est pas générique et qu'il peut avoir des instances. La deuxième instanciation (*RelationshipExtendClass*) concerne l'héritage entre deux classes Java (mot-clé *extend*) ; la direction du polymorphisme est donc dirigée vers le « haut » et le type de classifieur qui peut être source de la relation est naturellement *ClassifierJavaClass.*

Définition du comportement Nous avons expliqué page 113, qu'une partie significative de la sémantique d'un modèle métier est encapsulée dans un petit nombre d'atomes génériques. Une partie de cette sémantique est capturée par les paramètres, les caractéristiques et les invariants[26]. C'est une première étape mais ce n'est pas suffisant pour prendre en compte toute la sémantique des atomes. Par exemple, la valeur des paramètres choisis lors de l'instanciation des atomes génériques affectera le comportement des atomes obtenus par dérivation. Il est nécessaire de pouvoir définir ce comportement.

Chaque atome, qu'il soit générique ou non, a un niveau méta (son concept) où il est possible de définir des actions ; quand l'atome est dérivé d'un atome générique, l'exécution de celle-ci est conduite par la valeur des paramètres. Pour d'autres atomes, il est possible de définir des actions, mais nous pensons que ce n'est pas vraiment utile dans la plupart des cas.

Une *action* a une signature, un « corps », des préconditions et des postconditions qui sont définis en s'appuyant sur la réification des entités et des paramètres hyper-génériques, lorsqu'il s'agit d'un atome générique[27]. Une action peut aussi se voir attacher l'exécution de préoccu-

[26]Comme dans MOF ou UML, il est possible dans SmartModels de définir des invariants. Cela contribue à la description de la sémantique des entités.

[27]On rappelle que les assertions sont décrites à l'aide d'OCL [230, 238] et enrichies par des informations

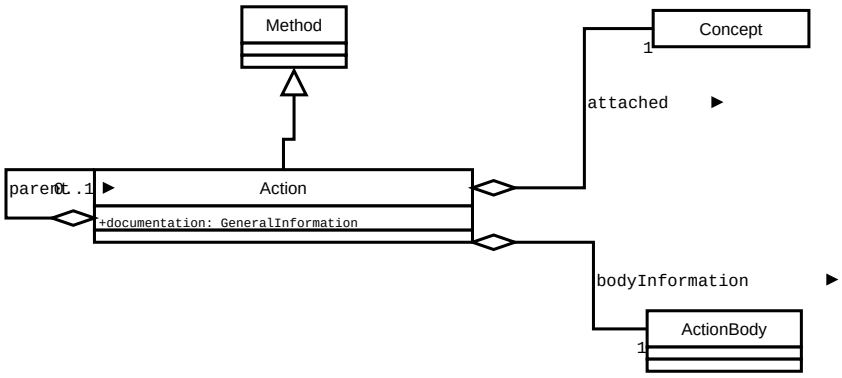

Fig. 6.11.: Description d'une action dans SmartModels

pations orthogonales (voir la section 6.2.3, page 122). Nous rappelons ici l'importance pour une action d'être complètement indépendante des applications définies sur le modèle métier dont elle participe à la description. Un scénario typique pour une application est de décrire des traitements qui s'appuient sur les actions, c'est-à-dire qui fassent appel à ces actions ou s'intéressent aux valeurs prises par les paramètres hyper-génériques.
Trois approches peuvent être envisagées pour décrire le « corps » d'une action dans le métamodèle : *i)* proposer une représentation complète du corps qui pourrait correspondre à la réification d'un pseudo-langage, *ii)* déléguer la description du corps de l'action au langage d'implémentation sous-jacent ; le métamodèle mémorise alors seulement que l'action a un corps et, *iii)* proposer une description partielle du corps de l'action[28]. Typiquement, pour la première et la troisième solution le corps de l'action doit être partiellement généré, tandis que la seconde solution laisse la description du corps complètement sous la responsabilité du développeur. Actuellement nous privilégions la troisième solution mais l'expressivité offerte pour la description du corps de l'action décrite ci-dessous nous semble *a priori* insuffisante (en particulier dans le contexte de l'informatique ubiquitaire) ; elle sera améliorée au fur et à mesure que des facilités utiles seront mises en évidence. Parmi les informations que nous proposons pour décrire un corps on trouve en particulier :

- le type d'assertion (vérification de propriétés à l'exécution, support pour la génération, etc.),
- la spécification des paramètres et des caractéristiques utilisés par une action,
- la définition d'un ensemble de triplets (une étiquette d'identification, un ensemble de couples composés d'un nom de paramètre hypergénérique et de sa valeur, une assertion OCL).

Pour définir une assertion, on pourra ainsi utiliser n'importe quelle propriété, méthode ou action du modèle, mais aussi une étiquette d'identification, dans le but par exemple d'introduire des règles de précédence ou des noms prédéfinis (par exemple : *result* qui permet d'accéder au résultat d'une fonction ou d'une action). L'objectif n'est pas seulement d'exécuter les assertions en fonction de la valeur de paramètre mais aussi de générer le code qui permet d'implanter le code correspondant.

complémentaires.

[28] Par exemple, mémoriser la liste des paramètres hypergénériques qui sont impliqués dans la sémantique de l'action.

```
<action> <name>Match</name>
    <signature> <returnType> <primitiveType> <string></string> </primitiveType> </returnType>
                <methodParameter> <name>feature</name>
                                  <atomType>Feature</atomType> </methodParameter>
                <methodParameter> <name>paramTypes</name>
                                  <collectionType></collectionType>  </methodParameter>
    </signature>
    <listOfActionAssertions>
           <preCondition> <name>precondition1</name>
              <ocl_Constraint> assertion in OCL </ocl_Constraint> <parentEntity></parentEntity>
           </preCondition>
           <postCondition> <name>Postcondition1</name>
              <ocl_Constraint> assertion in OCL </ocl_Constraint> <parentEntity></parentEntity>
           </postCondition>  </listOfActionAssertions>
    <body>
         <parameterName>Generator</parameterName>
         <parameterName>ExtensionCreation</parameterName>
         <characteristicName>ValidSourceRelationships</characteristicName> </body>
    <parentEntity></parentEntity>
</action>
```

Fig. 6.12.: Un exemple d'action pour OFL

Il est important de distinguer la capacité du modèle à plus ou moins décrire le corps des actions, du langage de description qui est fourni à l'utilisateur afin de saisir cette description. Cet aspect mériterait une discussion plus approfondie mais on peut tout de même mettre en avant deux approches intéressantes : *i)* utiliser les diagrammes d'activités et/ou la partie *Action Semantics* d'UML, *ii)* proposer un langage dédié à la description de ces actions et plus généralement de la sémantique d'un modèle. Nous n'avons pas pour l'instant évalué l'une ou l'autre de ces solutions (ce qui va être fait prochainement sur un exemple concret proposé par EDF [117, 118]). Il est dit qu'UML est le métamodèle pour la spécification des modèles métiers et il est important de réutiliser les standards. Cependant, il faut aussi se souvenir qu'UML n'a pas été conçu au début pour la modélisation de modèles métiers mais pour celle d'applications à objets. Une alternative à ces approches peut être d'augmenter l'expressivité de MOF en ce qui concerne la description de la sémantique.

Il est à noter que le fait de pouvoir, dans une assertion ou dans une action, hériter respectivement d'une autre assertion ou d'une autre action (avec une relation d'héritage spécifique à chacune de ces entités[29]) permet de réutiliser des entités déjà définies.

6.2.3. Conception des applications relatives à un modèle métier

Cette section concerne la description d'applications qui s'appliquent sur un modèle métier et qui ont besoin d'accéder ou de modifier les informations contenues dans les atomes qu'il contient. Comme cela a été mentionné précédemment, on peut distinguer deux grandes catégories d'application s : celles qui décrivent des transformations de modèles et celles qui interrogent le contenu du modèle de données (ses atomes) pour réaliser des calculs ou mettre à jour l'information qu'il contient. À ce point de l'explication, on comprend facilement que la spécification de ces applications va se démarquer assez sensiblement de celle des applications orientées objets classiques.

[29]Cela peut par exemple vouloir dire pour une assertion de redéfinir non pas la contrainte mais une propriété comme par exemple celle qui pourrait spécifier la catégorie de l'assertion.

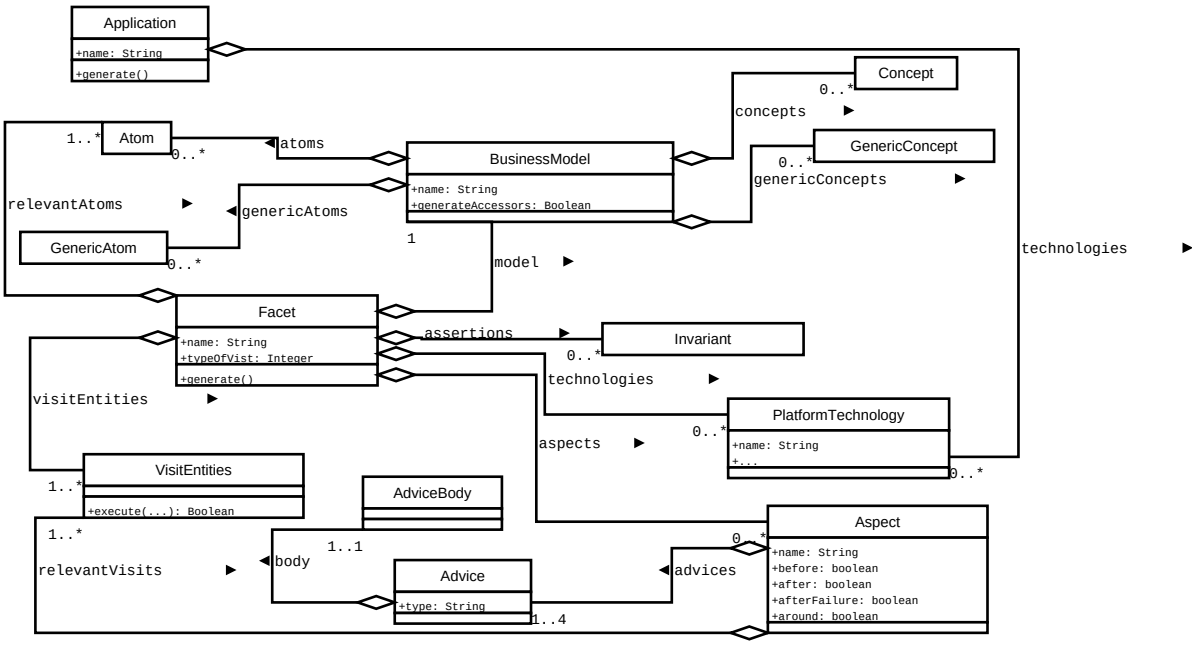

Fig. 6.13.: Description d'une application

Intuitivement, dans la programmation orientée modèle, une application correspond à une traversée du graphe des atomes représentés par le modèle métier. Pendant cette traversée (ce parcours), le comportement qui est contenu dans les facettes de l'application est exécutée séquentiellement (voir page 120). Pendant que les traitements associés à ces *facettes* sont exécutés il est possible de déclencher l'exécution d'*aspects* qui permettent d'intégrer des services orthogonaux (voir page 122). La figure 6.15 donne un aperçu de la vision que nous avons d'une application.

La figure 6.13 montre cinq entités principales qui sont : *i)* le modèle métier qui regroupe un accès à toutes les entités définies précédemment, *ii)* l'application qui encapsule toutes les informations relatives à une application, *iii*) la facette qui définit une préoccupation plutôt verticale (*sujet* au sens de SOP) d'une application, *iv)* l'entité de visite qui spécifie pour chaque atome le bout de code d'une facette donnée qui doit être exécuté et, *v)* l'aspect qui représente un service orthogonal pouvant être ajouté à des entités de visite d'une facette.

La définition des facettes de l'application et des aspects qui leurs sont associés pourraient être réalisée par l'intermédiaire de trois modèles métiers différents[30] en utilisant SmartModels lui-même. Nous décidons de suivre cette approche parce qu'elle permet de garder dans le

[30]La spécification d'une application est réalisée en plusieurs phases. Généralement, il y a en premier la spécification des facettes, puis la définition de l'application et finalement la description des aspects à appliquer.

```
<BusinessModel> <name>SmartModelsFacets</name>
  <generalInformation> ... <version>0.1</version> ... </generalInformation>
  <genericConcept> <name>ConceptFacet</name>
     <parameterDefinition>TypeOfVisit</parameterDefinition>
     <characteristicDefinition>handledModifiers</characteristicDefinition>
     <invariant>...</invariant> <parentEntity></parentEntity> <concepts></concepts>
  </genericConcept>
  <atom> <name>AtomVisitEntities</name> ... </atom>
  <atom> <name>AtomTechnology</name> ... </atom>
  <atom> <name>AtomBusinessModel</name> ... </atom>
  <atom> <name>AtomInvariant</name> ... </atom>
  <atom> <name>AtomAtom</name> ... </atom>
  <genericAtom> <name>AtomFacet</name>
     <attribute> <name>name</name>
                 <listOfModifiers> <identifier>private</identifier>...
                 <primitiveType> <string></string>  </primitiveType> </attribute>
     <attribute> <name>relevantAtoms</name>
                 <listOfModifiers> <identifier>private</identifier>...
                 <collectionType>AtomAtom</collectionType> </attribute>  ...
     <parentEntity></parentEntity> <metaEntity>ConceptFacet</metaEntity>
     <listOfModifiers> <identifier>public</identifier> ... </listOfModifiers>
  </genericAtom>
  <conceptInstanciation> <name>FacetForTransformation</name>... </conceptInstanciation>
  <derivedAtom> <name>FacetForTransformation</name>...
  <parameterDefinition> <name>typeOfVisit</name>
     <enumerationType> <identifier>abstractVisit</identifier>
                       <identifier>TransformationVisit</identifier>
                       <identifier>FullCustomizabeVisit</identifier> </enumerationType>
     <defaultValue> <identifier></identifier> </defaultValue>
     <redefinitionPairs> ... </redefinitionPairs>
  </parameterDefinition>
  <characteristicDefinition> <name>handledModifiers</name> ... </characteristicDefinition>
  <action>  <name>generate</name> ... </action>
</BusinessModel>
```

FIG. 6.14.: Extrait du modèle métier relatif aux facettes de SmartModels.

métamodèle seulement les entités de base et de définir toute nouvelle fonctionnalité par la spécification d'un nouveau modèle métier. Naturellement, cela amènera à délocaliser dans ces modèles métiers toutes les informations utiles à sa description. La figure 6.14 montre quelques-uns des principaux éléments du modèle métier dédié à la spécification des facettes.

Les facettes d'une application Une *facette* regroupe un ensemble d'entités appelées *entités de visite*, et représente une préoccupation de l'application par rapport à l'utilisation du modèle métier (voir la figure 6.15).

Une entité de visite est décrite en particulier par une méthode *execute* qui contient non seulement le comportement de la visite mais aussi des facilités pour décrire et vérifier des préconditions et des postconditions sur cette visite. À l'image de langages de programmation comme Eiffel [210, 212], les préconditions et les postconditions sont évaluées respectivement au début et à la fin de la visite ; elles déterminent si celle-ci a réussi ou échoué. Une entité de visite contient aussi des accès directs[31] à l'atome lui-même ou à certaines de ses propriétés quand elles doivent être manipulées par la méthode *execute*. L'avantage est double : minimiser le code à écrire et cacher la complexité de la représentation du modèle métier.

Une facette représente un découpage vertical de l'application tandis que la relation d'hé-

[31] Ils sont générés automatiquement en fonction des informations mentionnées dans la description d'une facette et d'un modèle métier.

ritage offre un découpage horizontal qui introduit plusieurs niveaux d'abstraction dans le modèle métier et l'application[32]. L'organisation par facettes d'une application s'inspire de la programmation orientée sujets (SOP). Chaque facette correspond à une partie des traitements qui sont exécutés sur une entité. Selon le type d'application, une facette d'une application donnée adresse soit le même ensemble d'atomes que les autres facettes de l'application soit, pour certains traitements plus spécifiques, des ensembles différents (une facette peut choisir l'ensemble des atomes auquel elle s'intéresse). La description d'une facette s'appuie sur le patron de conception *Visiteur* [252, 221]. Selon les besoins des applications, le comportement relatif à la visite d'un atome peut être plus ou moins dispersé. Dans une première approximation, un atome d'un modèle métier est associé à une seule entité de visite (voir page 121 pour une description plus approfondie).

Comme le montre la figure 6.13, une facette spécifie *i)* le modèle métier sur lequel elle s'appuie, *ii)* comment le modèle métier est visité, *iii)* les entités (atomes) qui sont significatifs par rapport aux objectifs et éventuellement *iv)* des technologies supplémentaires.

Les technologies sont définies indépendamment de SmartModels (par exemple, par une API ou une bibliothèque de classes). Elles contiennent des fonctionnalités qui permettent de définir plus facilement l'application. Par exemple, l'API *DOM* [314, 315] permet de manipuler des représentations XML des modèles métiers. D'autres informations peuvent être ajoutées afin de générer des entités de visite qui correspondent exactement aux attentes du programmeur. La génération du code source est importante pour notre approche parce qu'elle permet à ce dernier de se concentrer seulement sur les entités de visite qui concernent la facette mais aussi de l'assister lors de la description de leur comportement.

Comme cela a été dit précédemment, nous proposons d'encapsuler le savoir-faire relatif à la description des facettes mais aussi des applications dans un modèle métier « indépendant ». Une première étape consiste à ce qu'une application soit composée d'une ou plusieurs facettes qui seront exécutées séquentiellement, mais un mécanisme de composition plus sophistiqué devra être proposé. De plus, les facettes peuvent aussi devenir des composants et une application peut être réalisée à partir de composants. Le fait de s'appuyer sur un modèle métier favorise la réutilisation et l'évolution du comportement d'une application.

Paramétrage des visites sur le modèle métier Nous nous intéressons ici au paramètre *typeOfVisit* qui est défini dans une facette (*Facet*) et plus précisément dans *ConceptFacet* (voir figures 6.13 et 6.14). Le comportement relatif à la visite d'un atome peut être découpé différemment selon les besoins des applications. En particulier, les applications, qui sont le résultat de transformations ou qui sont associées à un modèle qui subit des transformations (par exemple dans le cadre du passage d'un PIM vers un PSM), nécessitent un découpage plus fin des entités de visite.

Le modèle métier est une structure passive sur lesquelles les facettes et les aspects potentiels sont exécutés. Le parcours de la structure est automatiquement généré ; il part d'un atome donné qui peut être spécifié par l'application ou la facette ; par défaut il s'agit de l'atome racine du modèle métier. Ce parcours correspond à la navigation à l'intérieur du graphe qui correspond à l'ensemble des propriétés des atomes[33]. Les atomes considérés doivent à la fois

[32] Le modèle supporte des hiérachies d'atomes, de concepts, d'entités de visite, de facettes et plus généralement des hiérarchies de toute entité de première classe. Ces relations d'héritage n'ont d'ailleurs pas toutes la même sémantique.

[33] Selon la communauté concernée, ce graphe peut s'appeler graphe de *clientèle*, graphe de *composition* ou *d'agrégation* ou encore graphe de *délégation*.

être « visibles » (*public* en Java) dans le modèle métier et faire partie des atomes intéressants (*relevantAtoms*) des facettes ; ils peuvent éventuellement représenter uniquement un niveau d'abstraction sans pouvoir avoir d'instances (*abstract* en Java).

À l'heure actuelle, nous proposons trois types d'entité de visite et un parcours en profondeur d'abord[34]. L'ordre utilisé pour visiter les propriétés d'un atome n'est pas significatif sauf s'il est précisé lors de la description d'une facette ou d'un atome. Voici les trois types de visite ou parcours :

- *Une visite adaptable.* C'est le type de visite qui offre la meilleure flexibilité puisque le programmeur peut décider par exemple de ne pas s'intéresser à certains sous-graphes, ou de ne les parcourir que sous certaines conditions. La contre-partie de cette flexibilité est la perte de lisibilité puisque le traitement associé à la visite contient aussi les instructions liées au parcours du graphe. L'utilisation de ce type d'entité de visite n'est pas recommandée quand la sémantique des applications doit être réutilisée sur des atomes dont la forme (la réification) évolue (par exemple en gagnant ou perdant certaines propriétés).
- *Une visite transparente.* Elle convient aux applications dont la sémantique est indépendante du parcours du graphe qu'il est donc inutile de montrer au concepteur de l'application. Une telle entité de visite ne contient que le traitement à associer à la visite ce qui permet de garantir une bonne lisibilité et de pouvoir réutiliser ces traitements dans d'autres applications. Néanmoins, force est de constater qu'elle reste très dépendante de la structure des atomes (la description du comportement est fortement couplé avec les propriétés qui décrivent l'atome).
- *Une visite morcelée.* Elle convient aux applications dédiées à la transformation de modèles et dont l'objectif est de transformer à la fois la structure et la sémantique du modèle métier, et les traitements mis en œuvre par ses applications. Pour cela, il faut que le comportement associé à la visite soit le plus indépendant possible de la structure des atomes du modèle métier et donc que la granularité de la visite soit augmentée. La granularité qui est proposée dans SmartTools à cet effet, et que nous reprenons à notre compte, implique qu'il y a autant d'entités de visite qu'il y a de propriétés dans les atomes. Intuitivement si, dans un modèle obtenu par transformation, une propriété est déplacée dans un autre atome, il devrait être relativement simple[35] de transformer le comportement de la visite de façon à ce qu'elle s'applique au nouveau modèle métier.

Gestion des aspects Comme cela a déjà été mentionné, la gestion des aspects devrait être assurée par un modèle métier indépendant. La programmation par facettes permet d'insérer des comportements supplémentaires *a posteriori* à l'intérieur d'une nouvelle facette qui sera ajoutée à la liste des facettes[36]. Par contre, ce mécanisme n'est pas suffisant pour ajouter facilement des bouts de comportement qui sont orthogonaux à l'ensemble des facettes (découpage horizontal). Ce comportement sera exécuté à certains points de l'exécution d'une ou plusieurs facettes afin par exemple, de vérifier la validité d'une contrainte, le chargement de données, la vérification de droits d'accès ou le traçage d'appels de méthode. Afin de satisfaire à ces besoins, nous introduisons une nouvelle facilité qui est dérivée de la programmation par aspects (voir figure 6.15).

[34] D'autres sortes de visite et de parcours de graphe devraient être proposés dans le futur, en fonction des besoins qui apparaissent.

[35] Dans la plupart des cas, cela devrait pouvoir être réalisé de manière automatique.

[36] Une facette se veut plutôt indépendante des autres facettes, mais il est évident que l'ordre d'exécution peut influencer le résultat.

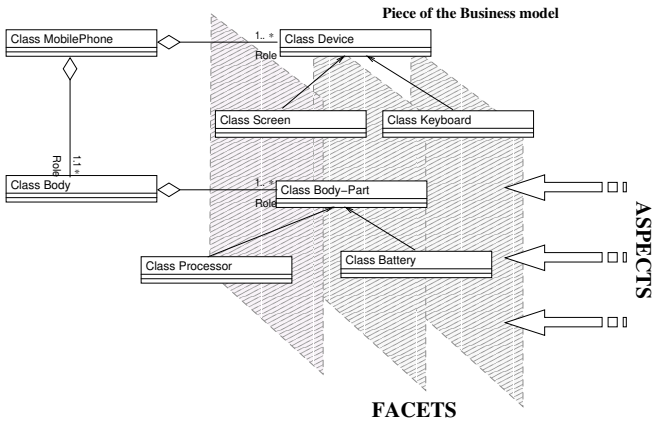

Fig. 6.15.: Description de la sémantique des applications par facettes et aspects

Chaque entité de visite accepte une liste d'aspects ; cela permet en particulier d'attacher un même et unique aspect à l'ensemble des entités de visite d'une facette particulière ou de toutes les facettes d'une application donnée. Naturellement ce ne sont là que deux cas particuliers ; l'ensemble des entités de visite auquel un aspect est associé est complètement libre. L'expressivité relative à la spécification de cet ensemble dépend seulement du langage dédié à la définition des points de jointure à l'intérieur des facettes[37]. Ce langage n'a pas besoin de l'expressivité proposée par l'AOP et il ne concerne pas des structures de programme mais des atomes. Cependant, il doit être possible d'intégrer un aspect avant l'invocation de la visite ou lorsque celle-ci débute ou se termine et de distinguer deux sortes de terminaison : la terminaison de l'exécution après un échec (une assertion n'est pas satisfaite) ou un succès (toutes les assertions sont vérifiées).

La section 6.2.2 a montré que dans SmartModels la description de la sémantique du modèle s'appuie sur *i)* des actions qui sont décrites dans un *concept* (au niveau méta) et, *ii)* des invariants de concept ou d'atome et des pré/postconditions pour des actions ; elles sont très similaires à celles qui sont dédiées aux entités de visite. Pourquoi est-il utile d'équiper la sémantique d'un modèle métier avec des aspects ? Une première réponse est que la sémantique d'un modèle métier peut être assez complexe pour que l'on sente le besoin de séparer ses différentes parties ; cela favorise ainsi la lisibilité, la maintenance et la réutilisation.

À première vue, il semble plus intéressant d'ajouter dynamiquement des aspects aux applications, alors que les aspects relatifs aux modèles métiers sont plus statiques. Cependant quand un modèle est obtenu par transformation d'un autre, il peut être intéressant d'avoir la possibilité d'adapter la sémantique encapsulée dans une action (proposée dans le modèle d'origine), afin de la rendre compatible avec le nouveau modèle. C'est typiquement pour une telle application que nous fournissons des aspects pour les assertions. Par exemple, un aspect qui met en œuvre un *around*[38] peut décider qu'en fonction du contexte d'exécution, l'assertion (dans le nouveau modèle) garde toute sa signification ou au contraire n'a plus de raison d'être.

[37] Dans AspectJ [171], un point de jointure est un endroit du programme où un aspect peut être intégré.

[38] Un aspect *around* est exécuté avant l'invocation d'une *i)* action, *ii)* entité de visite ou *iii)* assertion. Cela suggère le fait que l'on peut ou pas déclencher son exécution.

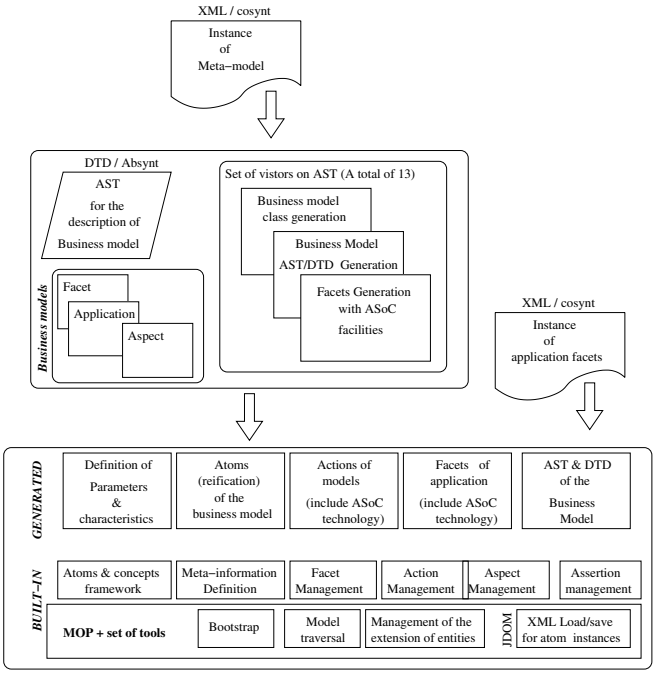

FIG. 6.16.: Implémentation de l'approche avec SmartTools

6.3. SmartFactory : une plate-forme pour des composants métiers

Pour mieux comprendre l'intérêt de SMARTMODELS, il est important de donner un aperçu de l'implémentation de SMARTFACTORY [189]. Nous allons nous intéresser aux principaux aspects, de la description du métamodèle à la réalisation d'un système opérationnel. La figure 6.16 décrit une implantation de ce prototype qui est basé sur la technologie mise en œuvre dans SMARTTOOLS[39] et qui s'en inspire bien évidemment. Grâce à ce prototype, nous pouvons montrer la faisabilité et l'intérêt de l'approche mais aussi obtenir des retours d'expérience qui nous permettront de l'améliorer.

Le métamodèle est décrit avec le langage *Absynt* qui est proche d'une description sous forme BNF. À partir d'une instance de ce métamodèle (qui est par défaut au format XML), SMARTTOOLS génère automatiquement :

- une réification en Java de l'arbre de syntaxe abstraite correspondant (AST),
- une DTD qui correspond à cet AST,
- une classe qui contient l'ensemble des méthodes de visite qui correspondent au traitement à appliquer sur chaque entité du modèle (implémentation du patron de conception *Visiteur* [252]).

Chaque méthode de visite contient les traitements qui doivent être exécutés sur les nœuds de l'AST. L'AST correspond au modèle métier et l'instance de l'AST généré correspond à une instance de ce modèle ; ils sont décrits en XML. Afin de faciliter la tâche du concepteur de modèle, nous avons utilisé le langage *cosynt* pour définir une syntaxe concrète de l'AST.

[39] Le fait qu'un premier prototype opérationnel ait été achevé en trois mois montre la flexibilité et la rapidité de développement avec SMARTTOOLS.

Un visiteur est créé pour chaque grande tâche à réaliser ; ils sont au nombre de treize à l'heure actuelle. Chacun d'entre eux pourra donc être remplacé ou supprimé en fonction de l'évolution du prototype. Ces visiteurs permettent la génération : *i)* des classes Java relatives au modèle métier, *ii)* des classes relatives aux applications et *iii)* d'autres modèles de représentation (AST et DTD). Par exemple, sept visiteurs traitent de la réification Java d'un modèle métier : définition des paramètres hypergénériques et des caractéristiques, spécification des actions, description des assertions, réification des atomes (génériques ou pas), description des concepts.
Les classes produites par ces générateurs s'appuient sur une hiérarchie de classes *built-in* et sur un protocole métaobjets (MOP) qui inclut un certain nombre de fonctionnalités pour *i)* la gestion de l'ensemble des instances d'un atome (son extension), qui prend en compte le polymorphisme, *ii)* le chargement et la sérialisation des instances du modèle dans ou à partir de fichiers XML et, *iii)* le parcours des instances des modèles métiers. Certaines classes de la hiérarchie *built-in* encapsulent les mécanismes dédiés à la mise en œuvre des assertions, des actions, des facettes et des aspects. D'autres modélisent les entités du modèle, leur niveau méta, leurs instances et à un plus bas niveau, l'intégration de ces objets dans le MOP.
Nous sommes en train de développer un petit module qui sera intégré dans le MOP et qui permet le *bootstrap* des principaux mécanismes (facette, application, etc.). En parallèle, nous avons débuté la description (en utilisant donc SmartModels lui même), des modèles métiers dédiés à la description des aspects, des facettes et des applications. Ces derniers points sont importants car ils montrent l'auto-exensibilité de notre approche.

Page vide

7. Perspectives

Nous désirons concentrer nos efforts sur les applications fortement évolutives et nous intéresser d'une part à leur modélisation et d'autre part à leur adaptation et à leur réutilisation.

Une des deux grandes perspectives concerne la réalisation de modèles métiers dédiés à l'évolution et à la réutilisation de classifieurs en s'appuyant sur les relations entre classifieurs et sur la séparation des préoccupations (voir section 7.1).

L'autre grande perspective que nous désirons considérer concerne la modélisation de modèles métiers. Cet axe de recherche qui a été décrit dans le chapitre 6 est jeune et a donc de nombreuses perspectives de développement que nous avons résumées dans la section 7.2.

La programmation par composants est particulièrement adaptée au développement d'applications fortement évolutive, il sera intéressant d'étudier l'impact de ce paradigme sur les deux perspectives citées ci-dessus.

7.1. Extension de langages ou langages métiers

Un domaine de recherche qui nous intéresse beaucoup est celui des langages de programmation ; il représente d'ailleurs la plus grande partie de notre activité de recherche passée. Nous désirons utiliser les techniques décrites dans les sections précédentes (définition de modèles métiers, description de langages métiers), pour proposer des approches qui permettent d'étendre la capacité des langages à réutiliser et à adapter les hiérarchies de classes qui composent les applications. Nous réfléchissons actuellement dans trois directions :

- améliorer et étendre le travail de Laurent Quintian (section 7.1.1),
- coupler la relation d'héritage inverse avec une version légère d'adapteur (section 7.1.2),
- définir une approche basée sur une relation de réutilisation de code et sur les traits [271] (section 7.1.3).

7.1.1. Réutilisation des préoccupations dans les langages à objets

Les langages à objets à la fin des années 1980 ont contribué à une meilleure réutilisation du logiciel mais les besoins toujours plus importants des applications ont mis en évidence leurs limites. C'est pourquoi nous avons proposé dans le chapitre 3 un modèle et une implémentation qui visent à augmenter leur capacité de réutilisation tout en préservant la robustesse.

Notre approche repose sur l'idée que chaque fois que nous voulons rendre une préoccupation réutilisable (celle-ci pouvant prendre la forme d'une bibliothèque de classes), il faut l'équiper de son *manuel d'utilisation*. Une des originalités de notre approche est que celui-ci n'est pas seulement une documentation textuelle et statique mais une entité à part entière (*l'adaptateur*) qui est exploité au début du processus de compilation pour construire l'application à partir d'un ensemble de préoccupations. Il permet de spécifier à la fois les entités qui vont subir des adaptations (les *cibles d'adaptation*) et les *adaptations* elles-mêmes.

Nous avons vu qu'un adaptateur offre *i)* une unité d'encapsulation qui supporte des déclarations incomplètes, *ii)* un typage des adaptations et des cibles d'adaptation, *iii)* une relation

d'héritage, notamment un héritage de concrétisation, qui met en œuvre une liaison dynamique entre les adaptations et leurs cibles d'adaptation, *iv)* un mécanisme d'assertions pour spécifier les dépendances entre les cibles d'adaptation et, bien sûr, *v)* une documentation textuelle.

L'indépendance de cette approche par rapport au langage de programmation choisi est aussi un aspect important même si dans la description des adaptateurs elle n'est pas aussi forte que nous l'aurions souhaité[1]. Mais des pistes ont été évoquées dans [191] pour pallier cela. Les capacités d'évolution du modèle sont réelles. Elles s'appuient sur une réification objet qui bénéficie du langage sous-jacent (modularité, héritage, etc.). L'ensemble des concepts objets considérés et la hiérarchie des adaptations sont notamment extensibles. L'indépendance de l'approche et l'évolutivité du modèle sont favorisées au niveau de l'implémentation par l'utilisation des technologies XML et par les capacités génératives fournies par la plate-forme Eclipse [121].

Parmi les perspectives de notre travail, certaines sont destinées à améliorer et étendre le modèle, d'autres à améliorer son implémentation. Concernant l'implémentation il serait intéressant de considérer directement le *byte-code* Java pour pouvoir par exemple, utiliser notre approche pour réutiliser les bibliothèques livrées sans le code source. Il serait aussi intéressant de valider notre modèle sur d'autres langages tels Eiffel ou C++.

Parmi les améliorations que nous désirons apporter au modèle qui a été décrit, il est possible de citer en particulier : *i)* introduire de nouveaux types d'adaptation comme par exemple la possibilité de modifier la clause *import* d'une classe et les chemins d'accès aux classes quand ils sont explicitement spécifiés dans le code source, *ii)* proposer une approche pour composer les adaptateurs associés à une même préoccupation, et *iii)* offrir plus de flexibilité au protocole de composition en permettant de décrire des adaptations alternatives. Il est en effet important de pouvoir réutiliser la préoccupation associée dans un encore plus grand nombre de contextes sans être obligé de réduire le spectre du protocole de composition et donc le guidage du programmeur et les contrôles opérés. Une perspective à plus long terme consiste à faire évoluer l'approche proposée par notre modèle pour qu'elle s'applique à la programmation par composants ou à la programmation par modèles.

7.1.2. Héritage inverse et séparation des préoccupations

Dans la section 5.2.3, nous avons présenté les grandes lignes d'une relation d'héritage inverse. Il nous semble intéressant d'étendre ce travail en couplant la relation d'héritage inverse avec des facilités supplémentaires qui s'inspirent à la fois du travail décrit dans le chapitre 3 et de ses perspectives (voir section 7.1.1) et d'autres approches comme gbeta [122], les mixins [50] ou d'autres travaux autour de l'héritage [123, 124]. Nous pensons en particulier que l'expressivité qui a été obtenue grâce à l'héritage inverse peut être encore améliorée en couplant celui-ci avec le concept d'*adaptateur*. Ainsi, nous proposons que lorsque la composition de deux hiérarchies de classes à l'aide de l'héritage inverse n'est pas immédiate, celle-ci soit réalisée avec une notion légère[2] d'adaptateur. Nous avons observé en particulier que lorsque deux hiérarchies sont assemblées, elles doivent en général subir des adaptations. Les principales que nous avons relevé sont les suivantes[3] :

[1] En effet, les traitements à insérer utilisent ce langage (Java dans notre prototype).

[2] Par rapport à celle définie dans [261].

[3] Les adaptations possibles sont peu nombreuses car nous désirons que le mécanisme soit dédié à des cas simples d'utilisation. Pour des adaptations plus lourdes nous continuons à investiguer l'approche développée dans le chapitre 3.

```
interface Subject {
  public void addObserver (Observer o) ;
  public void removeObserver (Observer o) ;
  public void notifyObserver () ;
 }
interface Observer {
  public void update (Subject o) ;
 }
import java.util.*.* ;
class ImplSubject {
  protected List observers = new ArrayList() ;
  public void addObserver (Observer o){observers.add (o) ;}
  public void removeObserver (Observer o){observers.remove (o) ;}
  public void notifyObserver () {
        for (Iterator iter = observers.iterator() ; iter.hasNext() ;)
           ((Observer) iter.next()).update (this) ; }
 }
```

FIG. 7.1.: Implantation possible du patron de conception *Observer*

- pouvoir substituer/redéfinir une méthode (ou un constructeur) d'une classe qui est la cible d'une relation d'héritage inverse. Le but est de pouvoir ajouter des traitements supplémentaires au comportement initial. Ils pourraient s'ajouter avant, après ou autour la méthode concernée [170, 171] ;
- pouvoir désigner une primitive déclarée dans un descendant (la cible de l'héritage inverse) dans une méthode ou un constructeur défini dans la classe qui déclare l'utilisation de l'héritage inverse, c'est à dire la source de la relation.

Nous pensons que ces facilités ne doivent être autorisées qu'à l'intérieur d'un adaptateur car cela suppose de faire référence explicitement à un descendant, ce qui est contraire aux principes d'une bonne réutilisation mais utile pour réaliser une adaptation spécifique. Il pourra être intéressant d'intégrer dans un adaptateur un mécanisme simple d'expression régulière qui simplifie la déclaration *i)* des primitives à factoriser ou à redéfinir quand par exemple, le même traitement doit être exécuté avant le code décrit dans les méthodes concernées ou bien *ii)* de la cible des relations d'héritage inverse lorsque nous voulons que la même classe soit l'ancêtre d'un ensemble de classes dont le contenu n'est pas connu à l'avance.
Même si ce n'est qu'une première réflexion sur le sujet, l'exemple proposé dans les figures 7.2 et 7.3 qui s'appuie sur le mécanisme proposé dans la section 5.2.3 permet de se faire une première idée de nos objectifs. Il correspond à la description partielle de l'adaptation qu'il faut réaliser pour introduire le patron de conception *Observer* (voir figure 7.1) dans une hiérarchie d'objets graphiques [133].

La figure 7.2 décrit la partie du protocole de composition qui est relativement indépendante de l'utilisation tandis que la figure 7.3 décrit la partie qui est spécifique à l'intégration dans la hiérarchie d'objets graphiques.

L'adaptateur défini par la classe *DP_ Use* représente une tentative pour définir le protocole de composition, c'est-à-dire toutes les adaptations qui sont indépendantes des contextes futurs d'utilisation. Dans une perspective identique à celle proposée dans [191], cela permet de réutiliser une partie de la description de l'adaptation au lieu de la dupliquer et, de guider ou contrôler le programmeur lorsqu'il désire adapter le patron de conception à un contexte spécifique (voir figure 7.3). Ainsi, l'utilisation de *redefine before* à l'intérieur de *DP_ Use* force le

```
adapter abstract class DP_Use {
   class ObserverL implements Observer exherits Label {
       // The method update is abstract and must be implemented
     }

   class I_SubjectB extends I_Subject exherits Button {
      redefine before click (); // make the method abstract
     }
 }
```

FIG. 7.2.: Exemple d'un adaptateur abstrait

```
adapter class DP_Use1 extends DP_Use {
   class ObserverL implements Observer exherits Label {
      update (Subject o){
         inferior (Label).setDefaultColor();
        }
     }
   class I_SubjectB extends I_Subject exherits Button {
      click () {
         notifyObservers;
         inferior (Button).click();
        }
     }
 }
```

FIG. 7.3.: Exemple d'un adaptateur concret

programmeur à ajouter un traitement avant le code original de la méthode *click*[4]. L'insertion de l'interface *Observer* oblige le programmeur à concrétiser la méthode *update*.

7.1.3. Réutilisation de code

Dans sa thèse [93], Pierre Crescenzo a proposé les principaux éléments d'une relation de réutilisation de code (voir section 5.1.1). Par ailleurs, pendant le workshop MASPEGHI'O4 [184], nous avons pu discuter avec les auteurs de [262] à propos de leurs travaux sur les *traits*. Nous avons décidé conjointement d'utiliser les idées mises en avant dans la relation de réutilisation de code pour étendre la sémantique des traits. Nous nous intéresserons notamment aux points suivants : *i)* garantir que le mécanisme proposé soit suffisant pour prendre en compte toutes les facettes de la réutilisation de code, *ii)* inclure la réutilisation de variables d'instances, *iii)* ne pas décider au niveau du trait si la primitive requise (*required*) qu'il utilise est une méthode ou un attribut et, *iv)* permettre d'affiner le comportement d'une méthode déjà définie, ce qui est l'objectif à la fois des mots-clés *super* [16] et *inner* [38, 122], et du mécanisme d'alias défini dans les traits [270, 271]. En parallèle nous voudrions étudier les différents statuts (module, type, etc.) que peut prendre une classe par rapport à la problématique de réutilisation. Pour illustrer cette idée, revisitons l'exemple de la section 5.1.1 (figure 7.4).

Dans la partie *A*, la classe *Person* est complètement implémentée et est utilisée pour implémenter la classe *Car*. La clause *renamed as* permet d'adapter la primitive *yearOfBirth* de la classe *Person* au contexte de la classe *Car*.

Dans *B*, la classe *Aged* ne peut pas avoir d'instance car elle contient une primitive incomplète

[4]On remarque ici que l'utilisation d'une expression régulière après le mot-clé *exherits* permettrait de rendre l'adaptateur abstrait beaucoup plus indépendant du contexte. De même il faudrait pouvoir renommer le cas échéant la méthode *click* mentionnée dans l'adaptateur abstrait dans l'adaptateur concret.

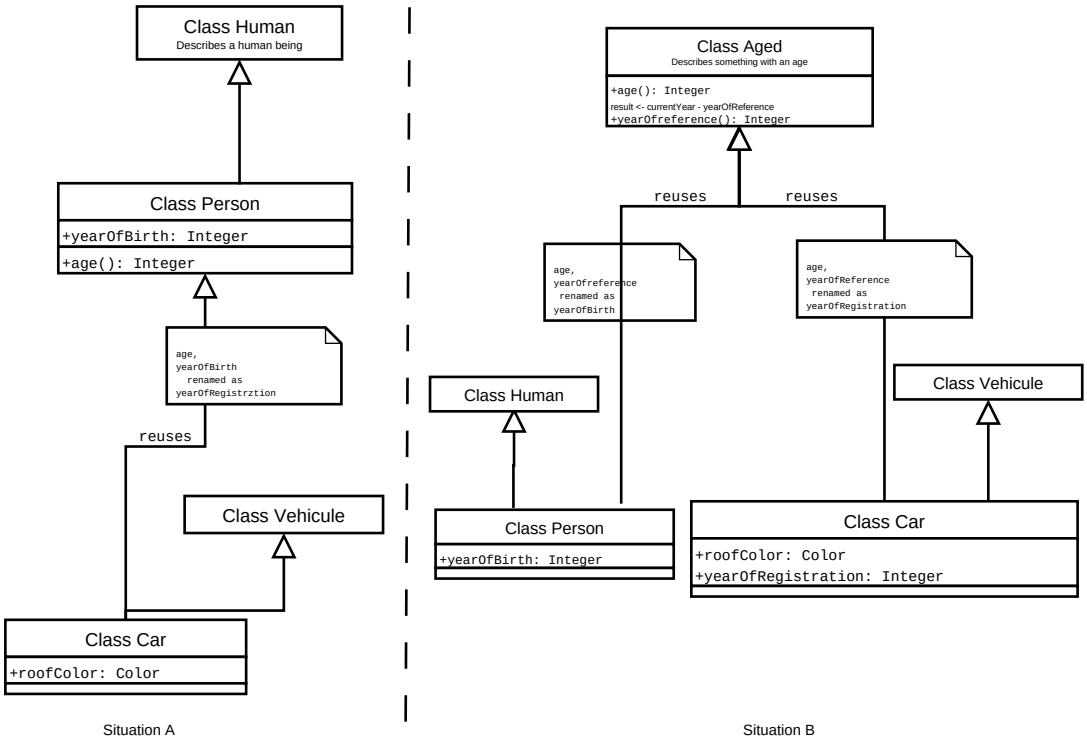

FIG. 7.4.: Exemples de réutilisation de code

yearOfReference qui est concrétisée à l'intérieur des classes *Car* et *Person* sous deux noms différents (grâce à la clause *renamed as*).

Dans cet exemple, la classe *Aged* (respectivement *Person*) pourrait intervenir comme un module uniquement[5] (grâce à la relation ***reuses***) ou comme un type si la classe *Aged* (respectivement *Person*) était héritée en utilisant la relation d'héritage normale fournie par le langage)[6].

Par contre, quand il s'agit de classes qui définissent seulement des constantes (attribut *final* en Java), ou des méthodes (par exemple, une bibliothèque de routines pour les mathématiques), elles devraient être considérées uniquement comme des modules. Pour améliorer la lisibilité, il pourrait être intéressant de suivre la même approche qu'en Eiffel [212] pour les classes expansées (*expanded* class) afin de pouvoir exprimer qu'une classe n'intervient que comme un module (*module* class).

7.2. Développement dirigé par les modèles

7.2.1. Sémantique d'un modèle et de ses applications

Lorsque nous avons commencé à réfléchir à la conception de SMARTMODELS, nous avons intégré le savoir-faire de SMARTTOOLS [255, 88, 89], c'est pourquoi celui qui connaît ce dernier trouvera de grandes similitudes entre lui et l'approche SMARTMODELS. Notre effort s'est porté en particulier sur la séparation des sémantiques du modèle métier et de ses applications mais

[5] C'est-à-dire que les règles du polymorphisme ne sont pas appliquées.

[6] Par les classes *Vehicle* et *Human*, si le langage ne supporte que l'héritage simple.

aussi sur une expressivité suffisante et adaptée à ce contexte qui conduise à la réalisation de modèles exécutables. Un modèle est exécutable si on peut spécifier le comportement des entités qui le décrivent et si ces entités peuvent être interrogées et modifiées par des applications tout en garantissant que les contraintes qui leurs sont associées seront contrôlées.

Sémantique d'un modèle Comme cela a déjà été évoqué dans le chapitre 6, l'objectif n'est pas de concurrencer UML qui peut très bien servir d'interface de définition d'un modèle métier (en fixant cependant certaines règles d'utilisation) mais de proposer à la fois un ensemble d'entités essentielles, une architecture et des propriétés qui permettent d'aboutir aussi aisément que possible à l'élaboration d'un modèle métier exécutable pouvant éventuellement être transformé (données et sémantique) pour s'adapter au contexte d'une application.

Séparer la sémantique entre le modèle et ses applications est selon nous une idée intéressante mais il n'est pas sûr qu'il faille la mettre en œuvre quel que soit le contexte. Cette séparation est une des différences qui existe entre les approches SmartTools et SmartModels, cependant il est clair que dans certain cas, la sémantique du modèle n'est pas identifiable par rapport à celle de ses applications car la partie indépendante du contexte est trop restreinte ou difficile à mettre en évidence. Nous avons basé la description de la sémantique sur la spécification *i)* de contraintes entre les entités, *ii)* d'une notion de généricité qui permet de définir des familles d'entités et, *iii)* de méthodes dont l'exécution s'appuie sur les paramètres caractérisant la généricité quand elle existe ou qui contiennent un comportement *ad hoc*.

À l'heure actuelle, nous identifions plusieurs travaux intéressants à développer. Le premier concerne l'élaboration d'un modèle minimal de type PIM qui permette de définir le contenu des méthodes décrivant la sémantique du modèle. Il devra pouvoir interagir avec le modèle associé au langage OCL [238] et être projeté, comme lui vers une plate-forme logicielle. Un second travail pourrait consister à étendre OCL afin de compléter la spécification d'une contrainte par la définition de métainformations qui pourront être utiles par exemple pour déterminer le moment de l'exécution ou différencier plusieurs types de contraintes.

Pour atteindre ces objectifs, nous avons notamment étudié les fonctionnalités offertes par l'outil de modélisation *OpenTool* [302] utilisé par EDF pour définir ses modèles métiers, dont nous avons étudié le contenu [117, 118].

Sémantique des applications La définition de la sémantique des applications dans SmartModels repose sur deux approches de séparation des préoccupations : la programmation par sujets et la programmation par aspects. Quand on considère des développements dirigés par les modèles, le parcours du modèle devient un aspect fondamental car c'est sur lui que va reposer la capacité à réaliser tel ou tel type d'application (voir page 120). Chaque parcours est associé à un sujet (nous l'appelons *facette*). Les traitements sont appliqués au modèle pendant un parcours de ce dernier. Ils sont inclus soit dans la définition de la facette, soit dans celle d'un aspect. La seconde approche est plus flexible puisqu'un traitement peut être associé ou non à un parcours en fonction du contexte et des besoins. Les facettes décrivant une application tout comme les différents aspects potentiels doivent parfois être appliquées dans un certain ordre, et même sous certaines conditions. Il faut donc pouvoir disposer d'un modèle de composition des facettes et des aspects qui permette d'adapter l'exécution d'un aspect ou d'une facette aux contraintes imposées par le contexte. Une piste pour atteindre cet objectif est de considérer un « parcours » comme une entité à part entière qui a ses propres opérateurs et de l'utiliser lorsque l'on décrit une facette. Ce supplément d'informations et d'opérateurs devrait faciliter

leur composition.

7.2.2. Composants et modèles métiers

Dans le chapitre 6, nous avons donné notre point de vue sur les qualités que devaient avoir un métamodèle dédié à la mise en œuvre de modèles exécutables. Si l'on considère par ailleurs que définir des applications centrées sur un modèle métier est une approche intéressante qui permet de séparer le savoir-faire d'une application spécifique, il est particulièrement important d'étudier l'impact de cette approche sur les modèles de composant. SmartTools propose un modèle de composants [87, 89] qui entend répondre à un certain nombre de besoins des modèles métiers. Dans un premier temps, il nous a semblé nécessaire de le positionner par rapport à l'état de l'art [206] et de s'appuyer sur les propositions de SmartModels pour mettre en évidence les qualités que doit posséder un modèle de composants dans le contexte d'applications centrées sur les modèles. À titre d'exemple, nous pouvons citer la faculté de rendre visible une représentation abstraite du modèle contenu dans un composant, à l'extérieur de ce dernier ou bien, la possibilité de décrire un composant générique qui accepte tout modèle en paramètre[7]. Il sera particulièrement intéressant de proposer une taxinomie des besoins liés à la présence d'un modèle dans un composant et de définir le modèle de composants à l'aide de SmartModels afin d'une part de pouvoir valider et appliquer notre approche sur un exemple concret et de proposer un modèle de composant abstrait qui contienne toutes les fonctionnalités nécessaires aux développements orientés modèles.

7.2.3. Adaptabilité de la représentation d'un modèle

Les travaux mentionnés dans le chapitre 6 et les perspectives de développement proposées dans la section 7.2.1 reposent sur la présence d'une plate-forme d'exécution qui est ouverte, adaptable et évolutive (ceux sont des préoccupations récurrentes dans ce document). L'implémentation d'applications demande l'utilisation de services génériques dans le sens où ils sont indépendants des modèles qu'ils manipulent ; la figure 7.5 en montre quelques exemples. Nous voulons utiliser l'approche sur la séparation et la composition de préoccupations qui a été développée dans le chapitre 3 dans le but de l'adapter aux besoins spécifiques d'une plate-forme d'exécution et de proposer une approche homogène pour définir et composer les PIMs ou les PSMs de nouveaux services. Ces services seront insérés dans les graphes d'héritage et d'instanciation de la plate-forme à laquelle est associé un protocole métaobjet qui permet d'accéder à chacune des entités du modèle (*concepts*, *atomes*, *attributs*, *méthodes*, *aspects*, *assertions*, etc.). Tout service générique sera défini par des opérations spécifiées sur ces entités et pourra ainsi être intégré dans la plate-forme. Au contraire, les services qui dépendent d'un modèle peuvent être vus comme une partie de leur sémantique ou une application.

7.2.4. L'aspect dynamique dans le contexte du DDD

Dans les sections 7.2.1, 7.2.2 et 7.2.3, nous avons évoqué un certain nombre de fonctionnalités qui pour certaines peuvent être mises en œuvre par transformation ou composition de modèles ; on peut citer par exemple l'intégration de nouveaux traitements (facettes ou aspects) ou bien l'adaptation des services offerts au développeur d'applications. Il sera particulièrement

[7] Cela signifie que la description des traitements est indépendante du modèle. Cette possibilité est offerte par le modèle de composants de SmartTools.

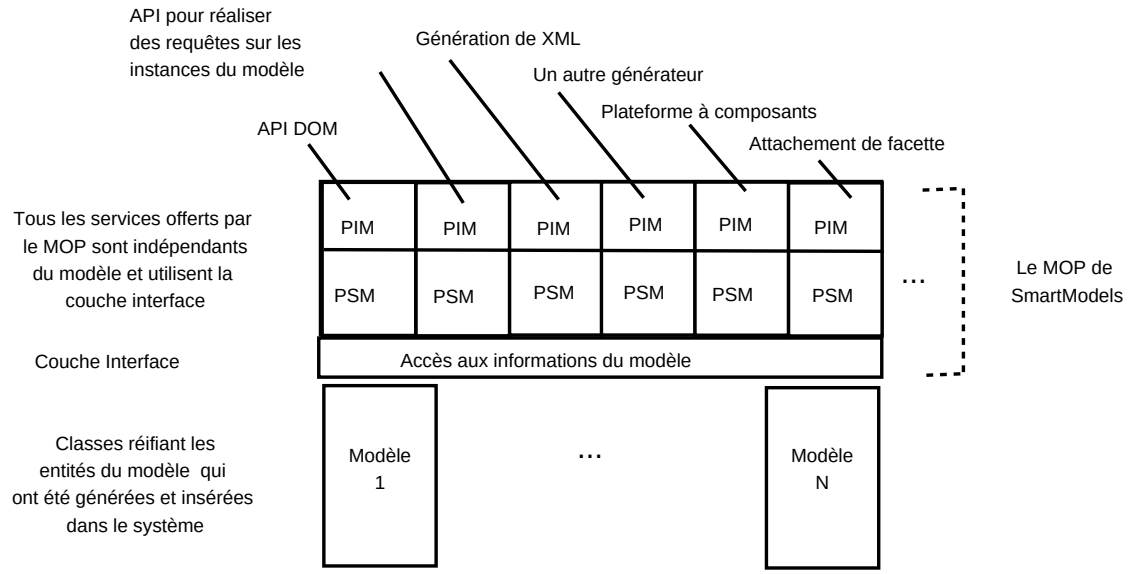

FIG. 7.5.: Plate-forme d'exécution pour SMARTMODELS

intéressant d'étudier comment ces différentes fonctionnalités pourraient être ajoutées dynamiquement, notamment dans le cadre des plates-formes à composants. La liaison dynamique des traitements pourrait intervenir au moment de la reconfiguration partielle d'un composant (échange de modèle, modification de l'interface, etc.), de son remplacement ou encore de l'échange de son implémentation.

Prenons un exemple concret. Transformer un modèle implique souvent une adaptation des entités de ce modèle de manière plus ou moins profonde. Le fait d'avoir au préalable associé de la sémantique à ces entités suggère qu'il sera nécessaire de modifier ce code de façon à ce qu'il s'adapte au modèle ainsi transformé. Le faire de manière statique peut être réalisé par des facettes qui vont interroger les informations contenues dans le modèle et mettre en œuvre une génération des nouvelles entités (avec leur sémantique) qui constitueront le modèle ainsi transformé. Réaliser ceci de telle manière à ce que les traitements puissent être chargés dynamiquement implique des choix de conception différents qui devront tenir compte des mécanismes mis en œuvre dans les principales plates-formes. Nous explorerons en particulier les approches basées sur un héritage dynamique comme par exemple celle de Self [3, 2].

7.2.5. Applications de SmartModels

À partir d'un modèle métier spécifié en utilisant SMARTMODELS qui correspond à OFL et à partir des modèles métiers relatifs aux applications (aspects et facettes), nous projetons de réaliser une application de SMARTMODELS qui génère des profils UML dans le but de pouvoir définir des applications qui soient conformes à un modèle métier donné. Pour cela, nous nous appuierons sur les thèses de Pierre Crescenzo [93] et Dan Pescaru [260].

Par ailleurs, il serait intéressant de revisiter le modèle défini dans la thèse de Laurent Quintian [261] sur l'adaptation et la réutilisation de hiérarchies de classes afin d'en proposer une modélisation avec SMARTMODELS. Cela permettra en particulier de vérifier que les possibilités de transformations de modèles proposées sont suffisantes.

8. Conclusion

Ce mémoire a tenté d'atteindre un triple objectif : donner un aperçu de notre contribution, montrer le cheminement qui a été suivi pour réaliser ces contributions et enfin, donner un certain nombre de perspectives possibles. Nous pouvons constater que le champ de recherche a évolué plusieurs fois, ce qui a nécessité à chaque fois un important travail de fond. Il y a quatre grandes étapes dans notre travail de recherche.

Il y a eu tout d'abord l'ajout d'un service de persistance à Eiffel à travers lequel nous avons voulu réunir deux domaines, celui des bases de données et celui des langages à objets, qu'il a donc fallu approfondir [183].

Une seconde étape a ensuite consisté à vouloir ouvrir les langages afin de leur permettre de mieux intégrer un large éventail de préoccupations ; nous nous sommes ainsi intéressés à la métamodélisation et au recensement des différents concepts objets [93].

Une troisième étape a permis d'approfondir les approches par séparation des préoccupations [58, 261] ; l'étude de ces approches a permis d'une part de continuer à contribuer à l'amélioration des langages à objets [162] et d'autre part, de préparer la quatrième étape relative aux développements dirigés par les modèles que nous avons entamée il y a moins d'un an.

Depuis la fin des années 1980, période qui correspond au commencement de mon implication dans la recherche nous avons constaté un certain nombre de faits marquants et d'évolution dans le domaine du développement des logiciels :

- La percée des langages à objets (fin des années 1980) avec des langages comme C++ [290], Smalltalk [139] ou Eiffel [210].
- L'introduction du paradigme objet dans les systèmes de gestion de bases de données au début des années 1990, avec notamment O_2 [109, 110] qui est issu d'un projet de l'INRIA. Puis l'échec et la disparition des bases de données objets à fin des années 1990.
- L'ouverture des langages : réflexivité, métaprogrammation (fin des années 1980, début des années 1990). En fait il y a eu deux grandes approches : celles basées sur le « tout objet » [54, 79, 169] et celles construites sur des langages existants comme C++ [70] ou Java [298].
- Les standards (OMG, W3C) occupent une place prépondérante depuis la fin des années 1990 ou le début des années 2000.
- Le constat de demi-échec des langages à objets (fin des années 1990) malgré le succès de Java [130] et l'arrivée de C# [154].
- L'apparition des approches par séparation des préoccupations (plutôt la fin des années 1990).
- Les débuts de la programmation par composants (fin des années 1990).
- L'avènement de l'approche MDA et la programmation par modèles (début des années 2000).

Cette énumération n'a bien sûr pas la prétention d'être exhaustive, mais elle montre le dynamisme de l'informatique face aux besoins toujours plus importants des applications et surtout de leurs utilisateurs. En particulier, la réflexion menée lors du colloque organisé en l'honneur

de Jean-François Perrot « *20 ans après : où en sont les objets ?* » et notamment [40, 84] ou bien la table ronde d'OOPSLA 2002 [111], montrent que c'est le succès mitigé des objets dans des domaines comme la réutilisation qui ont permis l'émergence de nouveaux styles de programmation (composant, aspect, modèles).
Notre contribution se place exactement dans la perspective de cette évolution de l'informatique et se rapproche d'une certaine manière des travaux relatifs aux projets OBASCO [224] et ATLAS [27]. Dans la suite nous positionnons nos travaux par rapport à cette évolution et par rapport à la réflexion qui est menée actuellement sur les objets.

FLOO. C'est une extension du langage eiffel pour intégrer un service de persistance transparent et efficace. Eiffel n'offrant que des facilités limitées d'introspection, la modification de la sémantique de l'accès aux attributs ou de l'appel de méthode (protocole d'intercession) a dû être faite directement dans l'exécutif du langage. Ce choix a permis d'offrir un service de persistance transparent (sans modification du code des classes) qui n'altère pas la réutilisabilité des classes. La description d'une transformation de modèle d'Eiffel vers O_2 a permis l'exécution de requêtes en mémoire persistante (à travers le serveur d'objets O_2) et donc de garantir l'efficacité de la solution. Il faut admettre que si la solution obtenue est parfaitement intégrée dans le langage Eiffel, cette solution a demandé un investissement conséquent qui est acceptable uniquement si le nombre de services à intégrer est limité et peu évolutif. L'évolution des besoins des applications va malheureusement à l'encontre de cette hypothèse.

OFL. C'est un modèle qui décrit les concepts objets en s'appuyant sur des paramètres dont les valeurs décrivent une partie de la sémantique opérationnelle que chaque langage associe aux principaux concepts objets (relations entre classifieurs, sortes de classifieur) et sur des actions dont l'exécution est dirigée par ces paramètres. Une de nos principales motivations a été d'ouvrir la sémantique opérationnelle des langages pour pouvoir leur associer de nouveaux services (persistance, gestion de version, distribution) à un moindre coût. L'étude approfondie des différents comportements que peuvent avoir les mécanismes d'héritage ou bien la mise en évidence d'une grande variété d'usages [94] a permis d'envisager d'une part l'extension de langages à objets avec une relation d'héritage inverse ou de réutilisation de code et d'autre part l'utilisation de l'approche paramètre/action développée pour la description du modèle objet pour décrire la sémantique d'autres modèles métier.

JAdaptor. Il reprend la problématique de l'ajout de services à la demande mais veut aller plus loin afin d'augmenter les capacités de réutilisation des langages à objets. Il s'appuie fortement sur les approches par séparation des préoccupations et tient compte des expériences passées. Par exemple, il est indépendant du langage de programmation : JADAPTOR doit être vu comme un modèle métier dédié à la réutilisation des préoccupations. Si la syntaxe associée est très influencée par le langage, il sera vu comme une extension de ce dernier [261] tandis que si la syntaxe est elle aussi dédiée il deviendra un langage métier (*Domain-Specific Language - DSL*) [191]. Par ailleurs, le modèle proposé met l'accent sur le guidage et le contrôle du programmeur d'application à travers une documentation embarquée dans la préoccupation qui permet donc d'allier la robustesse à la réutilisation.

Autour de l'héritage. Ces travaux résultent de la réflexion que nous avons menée sur les relations entre classes dans le cadre du projet OFL et ils ont pour objectif d'augmenter l'ouverture

et la réutilisation des applications. Un premier travail a étudié la faisabilité d'introduire un héritage générique au même titre qu'il y a des types génériques [162] ; il est pour l'instant en sommeil. Un second concerne l'introduction de l'héritage inverse afin de pouvoir agir de manière non intrusive sur des classes existantes ; nous avons commencé à réfléchir sur un couplage de cette relation avec une notion d'adaptateur simplifié (voir section 7.1.2). Un troisième projet qui est seulement en train de débuter mais auquel nous pensons depuis longtemps est la définition et la mise en œuvre d'un mécanisme de réutilisation de code (voir section 7.1.3).

SmartModels. Ce métamodèle représente une première incursion dans les développements dirigés par les modèles. La motivation de ce travail repose à la fois sur l'intérêt de captitaliser le savoir-faire des entreprises en le rendant indépendant des applications qui le manipulent et sur l'hypothèse qu'il pouvait être intéressant, comme cela a été fait dans le cadre d'OFL, de décrire la sémantique en s'appuyant sur un jeu de paramètres et d'actions mais aussi d'aspects [188]. SmartModels propose au concepteur du modèle à travers un langage métier, de décrire les paramètres qui vont permettre ensuite soit de proposer des variantes ou des évolutions de sémantique, soit d'adapter le modèle aux besoins d'une application.

Bilan final. Les approches à objets et les langages qui les mettent en œuvre représentent peut être simplement une étape dans le développement de l'informatique ; l'arrivée de nouveaux paradigmes comme les composants, les aspects, les sujets ou les modèles rendent peut-être le paradigme de l'objet moins attrayant. Pourtant, l'intérieur d'un composant repose souvent sur un programme objet, la plupart des langages relatifs à la programmation par sujets ou par aspects (pour ne citer que ces approches par séparation des préoccupations) sont des extensions de langages à objets et, c'est UML qui est l'outil de description de modèles métiers.

Comme d'autres, nous pensons que la classe n'est pas toujours la bonne entité d'encapsulation et l'héritage n'est pas non plus toujours le meilleur mécanisme pour la réutilisation logicielle. Par ailleurs, les techniques de réflexion et de métaprogrammation offrent une expressivité complémentaire intéressante, tandis que l'ajout d'un niveau d'encapsulation de première classe supplémentaire (tels la préoccupation, le composant ou le modèle), peut contribuer à l'amélioration de la réutilisation.

Cependant, même si ce constat n'est pas totalement positif, nous défendons l'idée que l'approche à objets n'est pas morte mais qu'au contraire elle doit continuer à évoluer en s'appuyant sur les nouveaux paradigmes et en créant des passerelles avec eux. Nous désirons donc exploiter l'expérience que nous avons acquise lors de la réalisation des différents modèles ou extensions pour proposer des évolutions de l'approche à objets qui permettent d'assurer une meilleure réutilisation du logiciel.

Bibliographie

[1] M. Abadi. Protection in Programming Language Translation. In *Proceedings of ICALP'98*. Springer-Verlag, July 1998.

[2] O. Agesen, L. Bak, C. Chambers, , B.-W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The Self 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc., Mountain View, CA, 1995.

[3] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference of SELF : Analysis of Objects with Dynamic and Multiple Inheritance. In O. Nierstrasz, editor, *Procedings of ECOOP'93*, LNCS(707), pages 247–267, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[4] R. Agrawal and N. Gehani. ODE (Object Database and Environment) : the Language and the Data Model. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 36–45, Portland, Oregon, May-June 1989. ACM Press.

[5] M. Aksit, editor. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 2003. ACM Press.

[6] M. Aksit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model : The Composition-Filters Approach. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, LNCS(615), pages 372–395, Utrecht, The Netherlands, June - July 1992. Springer-Verlag.

[7] A. Albano, L. Cardelli, and R. Orsini. Galileo : A Strongly-Typed Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2) :230–260, 1985.

[8] A. Albano, G. Ghelli, and R. Orsini. The Implementation of Galileo's Persistent Values. In Atkinson et al. [26], pages 253–263.

[9] A. Albano, G. Ghelli, and R. Orsini. Types for Databases : the Galileo Experience. In Hull et al. [157], pages 196–206.

[10] D. Ancona, G. Lagorio, and E. Zucca. Jam, Theory and Practice of a Java Extension with Mixins. Technical report, University of Genova, 1999.

[11] T. Andrews. ONTOS : a Persistent Database for C++. In Zdonik and Maier [319], pages 387–407.

[12] T. Andrews. Programming with VBASE. In Zdonik and Maier [319], pages 130–177.

[13] G. Ardourel. *Modélisation des mécanismes de protection dans les langages à objets*. Thèse de doctorat, Université de Montpellier II, décembre 2002.

[14] G. Ardourel, P. Crescenzo, and P. Lahire. LAMP : vers un langage de définition de mécanismes de protection pour les langages de programmation à objets. In J.-P. Briot and

J. Malenfant, editors, *Actes de LMO'2003, conférence nationale sur les Langages et Modèles à Objets. Publiés dans la revue L'Objet : Logiciels, Bases de données, Réseaux*, volume 9(1-2/2003), pages 151–163, Vannes, France, janvier 2003. Editions Hermès Science Publications.

[15] G. Arévalo, H. Astudillo, A. P. Black, E. Ernst, M. Huchard, P. Lahire, M. Sakkinen, and P. Valtchev, editors. *Proceedings of MASPEGHI'04, 3rd International Workshop on MechAnisms for SPEcialization, Generalization and inHerItance, at ECOOP'04*, Oslo, Norway, June 2004. I3S Laboratory, University of Nice - Sophia Antipolis, France, research report I3S/RR-2004-15-FR.

[16] K. Arnold and J. Gosling. The Java Programming Language. In *Java Series ... from the Source.* Sun Microsystems Inc., 2000.

[17] AspectJ team. Aspect J. http ://www.aspectj.org, 2003.

[18] H. Astudillo, M. Huchard, and P. Valtchev, editors. *Proceedings of the Workshop on Managing Specialization/Generalization Hierarchies (MASPEGHI) at ASE'03*, Montréal, Québec, Canada, October 2003.

[19] C. Atkinson and T. Kühne. Strict Profiles : Why and How. In *Proceedings of UML'02*, LNCS(1939), pages 309–322. University of York, UK, Springer-verlag, October 2000.

[20] C. Atkinson and T. Kühne. The Role of Meta-modeling in MDA. In J. Bézivin and R. France, editors, *Proceedings of the Workshop in Software Model Engineering*, 2002.

[21] M. Atkinson. Persistent Architectures. In J. Rosenberg and D. Koch, editors, *Proceedings of the Third International Workshop on Persistent Object Systems*, pages 73–97, Newcastle, Australia, January 1989.

[22] M. Atkinson. Questioning Persistent Types. In Hull et al. [157], pages 2–24.

[23] M. Atkinson, P. Bailey, K. Chisholm, W. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4) :360–365, November 1983.

[24] M. Atkinson, F. Bancilhon, D. DeWitt, K. dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database Manifesto. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object Databases (DOOD'89)*, pages 223–240, Kyoto, Japan, December 1989. North-Holland/Elsevier Science Publishers.

[25] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM surveys*, 19(2) :106–190, june 1992.

[26] M. Atkinson, P. Buneman, and R. Morrison, editors. *Proceedings of the First Workshop on Persistent Objects*, Appin, Scotland, August 1985.

[27] Atlas. Activity report of the ATLAS project : Complex Data Management in Distributed Systems. Technical report, INRIA Rennes - University of Nantes, 2003.

[28] I. Attali, C. Courbis, P. Degenne, A. Fau, J. Fillon, D. Parigot, C. Pasquier, and C. Sacerdoti Coen. SmartTools : a Development Environment Generator based on XML Technologies. In *Proceedings of the Workshop on XML Technologies and Software Engineering at ICSE'01*, Toronto, Canada, May 2001.

[29] C. Austin. J2SE 1.5 in a Nutshell. developers.sun.com : The Source for Developers, May 2004. http ://java.sun.com/developer/technicalArticles/releases/j2se15/.

[30] A. ban Deursen, P. Klint, and J. Visser. Domain-Specific Languages : An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, June 2000.

[31] F. Bancilhon, S. Cluet, and D. C. A Query Language for the O_2 Object-Oriented Databases. *Communications of the ACM*, 34(10) :122–138, October 1991.

[32] F. Bancilhon and C. Delobel. Object-Oriented Database Systems & Recent Advances in OO-DBMS. In *Tutorials of the TOOLS'91 International Conference*, pages 106–156, Paris, France, March 1991.

[33] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 1–38, Los Angeles, CA, February 1988.

[34] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS : Tools for Implementing Domain-Specific Languages. In P. Devanbu and J. Poulin, editors, *Proceedings of the Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, June 1998.

[35] L. Bergmans. *The Composition-Filters Object Model*. Ph.D Thesis, University of Twente, Netherlands, 1994.

[36] L. Bergmans and M. Aksit. Analyzing Multi-Dimensional Programming in AOP and Composition Filters. In Ossher et al. [247].

[37] L. Bergmans and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, 44(10) :51–57, October 2001.

[38] Beta team. FAQ BETA. http ://www.faqs.org/faqs/beta-language-faq/, 2003.

[39] J. Bézivin. From Object Composition to Model Transformation with MDA. In Q. Li, editor, *Proceedings of TOOLS'01 USA*, pages 350–354, Santa-Barbara, CA, USA, August 2001. IEEE Computer Society.

[40] J. Bézivin. Le changement de paradigme des objets aux modèles : rupture ou continuité ? In *Numéro spécial de la revue "L'objet"* [182]. (à paraître).

[41] A. Bjornerstedt and S. Britts. Avance, an Object Management System. In Meyorwitz [214], pages 206–221.

[42] A. Bjornerstedt and C. Hulten. Version Control in an Object-Oriented Architecture. In Kim and Lochovsky [178], pages 451–486.

[43] A. Black, E. Ernst, P. Grogono, and M. Sakkinen, editors. *Proceedings of the Workshop on Inheritance at ECOOP'02*, Malaga, Spain, June 2002.

[44] G. Bobrow and al. The Common Lisp Object System Specification : Chapter 1 and 2. Technical report, Technical report 88-002R, X3J13 standards committee document, 1988.

[45] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. The Object Technology Series. Addison-Wesley Publishing Co., October 1998.

[46] M. N. Bouraqadi-Saâdani. Concern Oriented Programming Using Reflection. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'00*, 2000.

[47] N. Bouraqadi-Saâdani and T. Ledoux. Le point sur la programmation par aspects. *Technique et Sciences Informatiques*, 20(4) :505–528, 2001.

[48] R. Bourret. XML and Databases. http ://www.rpbourret.com/xml/XMLAndDatabases.htm, Updated in July 2004.

[49] R. Bourret. XML Database Products. http ://www.rpbourret.com/xml/XMLDatabaseProds.htm, updated june 29^{th} 2004.

[50] G. Bracha and W. Cook. Mixin-Based Inheritance. In N. K. Meyrowitz, editor, *Proceedings of OOPSLA/ECOOP'90*, volume 25(10) of *ACM SIGPLAN Notices*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[51] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. Williams, and M. Williams. The Gemstone Data Management System. In Kim and Lochovsky [178], pages 283–308.

[52] J. Brichau, M. Glandrup, S. Clarke, and L. Bergmans, editors. *Proceedings of the Workshop on Advanced Separation of Concern at ECOOP'01*, Budapest, Hungaria, June 2001.

[53] J.-P. Briot and P. Cointe. The OBJVLISP Model : Definition of a Uniform, Reflexive and Extensible Object-Oriented Language. In J. B. H. du Boulay, D. Hogg, and L. Steels, editors, *Proceedings of ECAI'86*, pages 225–232, Brighton, United Kingdom, July 1986. North-Holland.

[54] J.-P. Briot and P. Cointe. Programming with ObjVlisp Metaclasses in Smalltalk-80. In Meyrowitz [217], pages 419–431.

[55] P. Brown. *Object-Relational Database Development : A Plumber's Guide*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000. 828 pages.

[56] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Gestion des objets persistants grâce aux liens entre classes. In U. de Nantes, S.-M. G. Sodifrance, and É. des Mines de Nantes, editors, *Actes de OCM 2000, conférence nationale sur les Objets, Composants, Modèles « Passé, Présent, Futur»*, pages 145–154, Nantes, France, mai 2000.

[57] A. Capouillez, R. Chignoli, P. Crescenzo, and P. Lahire. Hyper-généricité pour les langages à objets : le modèle OFL. In R. Godin and I. Borne, editors, *Actes de LMO'01, conférence nationale sur les Langages et Modèles à Objets. Publiés dans la revue L'objet : Logiciels, Bases de données, Réseaux*, volume 7(1-2/2001), pages 63–78, Le Croisic, France, janvier 2001. éditions Hermès Science Publications.

[58] A. Capouillez, P. Crescenzo, and P. Lahire. Separation of Concerns in OFL. In Á. Frohner, editor, *Synthesis of the Workshop "Advanced Separation of Concerns" at ECOOP'01, in "Object-Oriented Technology : ECOOP 2001 Workshop Reader"*, LNCS(2072), Budapest, Hungaria, June 2001. Springer Verlag.

[59] A. Capouillez, P. Crescenzo, and P. Lahire. Le modèle OFL au service du métaprogrammeur - Application à Java. In M. Dao and M. Huchard, editors, *Actes de LMO'2002, conférence nationale sur les Langages et Modèles à Objets. Publiés dans la revue L'objet : Logiciels, Bases de données, Réseaux*, volume 8(1-2/2002), pages 11–24, Montpellier, France, janvier 2002. Editions Hermès Science Publications. ISSN : 1262-1137 ; ISBN : 2-7462-0403-7.

[60] A. Capouillez, P. Crescenzo, and P. Lahire. OFL : Hyper-Genericity for Meta-Programming - An Application to Java. Research Report I3S/RR–2002-16–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, April 2002.

[61] L. Cardelli and D. MacQueen. Persistence and Type Abstraction. In Atkinson et al. [26], pages 31–42.

[62] L. Cardelli, E. J. Neuhold, and M. Paul. Typefull Programming. In *IFIP Advanced Seminar on Formal Methods in Programming Langage Semantics*, State of the Art Reports Series. Springer Verlag, 1989.

[63] M. Carey and al. The EXODUS Extensible DBMS Project : An Overview. In Zdonik and Maier [319], pages 474–499.

[64] M. Carey, D. Dewitt, and S. Vandenberg. A Data Model and Query Language for EXODUS. In Haran Boral [151], pages 413–423.

[65] P. Caro. *Adding Systemic Crosscuting and Super-Imposition to Composition Filters.* Ph.D Thesis, University of Twente, Netherlands, 2001.

[66] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. In G. C. Fox, editor, *Concurrency Practice and Experience*, pages 1043–1061. Wiley & Sons, Ltd., September - October 1998.

[67] L. Carver and W. Griswold. Sorting out Concerns. In Ossher et al. [247].

[68] R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *Object Database Standard : ODMG 2.0.* Morgan Kaufmann Publishers, Inc., June 1997.

[69] W. Cazzola, R. J. Stroud, and F. Tisato, editors. *Proceedings of the First Workshop on Reflection and Software Engineering (OORaSE) at OOPSLA'99*, LNCS(1826), Denver, Colorado, USA, November 2000. Springer Verlag.

[70] S. Chiba. A Metaobject Protocol for C++. In Wirfs-Brock [311], pages 285–299.

[71] S. Chiba. OpenC++ Programmer's Guide for Version 2. Technical Report SPL-96-024, Xerox PARC, 1996.

[72] S. Chiba. Javassist - A Reflection-based Programming Wizard for Java. In *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, Vancouver, British Columbia, Canada, October 1998.

[73] S. Chiba and M. Tatsubori. Yet Another java.lang.Class. In *Proceedings of the Workshop on Reflective Object-Oriented Programming and Systems at ECOOP'98*, Brussels, Belgium, July 1998.

[74] C.-B. Chirila, P. Crescenzo, and P. Lahire. Reverse Inheritance : an Approach for Modeling Adaptation et Evolution of Applications. Research Report I3S/RR–2003-XX–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, October 2003.

[75] C.-B. Chirila, P. Crescenzo, and P. Lahire. Towards Reingineering : an Approach Based on Reverse Inheritance - Application to Java. Research Report I3S/RR–2003-XX–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, November 2003.

[76] C.-B. Chirila, P. Crescenzo, and P. Lahire. A Reverse Inheritance Relationship for Improving Reusability and Evolution : the Point of View of Feature Factorization. In Arévalo et al. [15], pages 9–14.

[77] H. Chou and W. Kim. Versions and Change Notification in an Object-Oriented Database System. In *Proceedings of the 25th ACM/EEE Design Automation Conference (DAC'88)*, pages 275–281, Anaheim, CA, USA, June 1988. IEEE Computer Society Press / ACM.

[78] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice-Hall, 1991.

[79] P. Cointe. Metaclasses are First Class : The ObjVlisp Model. In Meyrowitz [216], pages 156–165.

[80] P. Cointe. The ObjVlisp Kernel : a Reflexive Lisp Architecture to Define a Uniform Object-Oriented System. In P. Maes and D. Nardi, editors, *MetaLevel Architectures and Reflection*, pages 155–176. NorthHolland, Amsterdam, 1987.

[81] P. Cointe. The ClassTalk System : a Laboratory to Study Reflection in Smalltalk. In *Proceedings of the First Workshop on Reflection and Meta-Level Architectures in Object-Oriented Programmming at OOPSLA/ECOOP'90*, Ottawa, Canada, October 1990.

[82] P. Cointe. Reflective Languages and Meta-Level Architectures. *ACM Computing Surveys*, 28(4es), 1996. Art. number 151.

[83] P. Cointe and J.-P. Briot. ClassTalk : une transposition des métaclasses d'ObjVlisp à Smalltalk-80. In *Actes de RFIA'89*, volume I, pages 127–146, Paris, France, novembre - décembre 1989.

[84] P. Cointe, J. Noyé, R. Douence, T. Ledoux, J.-M. Menaud, G. Muller, and M. Südholt. Programmation post-objets : des langages d'aspects aux langages de composants. In *Numéro spécial de la revue "L'objet"* [182]. (à paraître).

[85] C. Constantinides, T. Skotiniotis, and T. Elrad. Providing Dynamic Adaptability in an Aspect-Oriented Framework. In Brichau et al. [52].

[86] S. Cook and S. Kent. The Tool Factory. In *Proceedings of the Workshop on Generative Techniques in the context of MDA at OOPSLA'03*, Anaheim - USA, October 2003.

[87] C. Courbis, P. Degenne, A. Fau, and D. Parigot. Un modèle de composants pour l'atelier de développement SmartTools. In *Actes des Journées systèmes à composants adaptables et extensibles*, Grenoble, France, octobre 2002.

[88] C. Courbis, P. Degenne, A. Fau, and D. Parigot. L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools. *RSTI numéro spécial XML et les Objets*, 9(11-2003) :1–26, 2003.

[89] C. Courbis, P. Degenne, A. Fau, and D. Parigot. Un modèle abstrait de composants. *RSTI-TSI numéro spécial sur composants et adaptabilité*, 23(2) :231–252, 2004.

[90] S. Crawley, S. Davis, J. Indulska, S. McBride, and K. Raymond. Meta-Meta is Better-Better ! In H. König, K. Geihs, and T. Preub, editors, *Proceedings of DAIS'97 (Distributed Applications and Interoperable Systems)*. Chapman & Hall, September-October 1997.

[91] P. Crescenzo. Un système de vues pour Eiffel. Mémoire de DEA, Laboratoire I3S (UNSA/CNRS), Sophia antipolis, France, juin 1995.

[92] P. Crescenzo. OFL : les relations et descriptions d'Eiffel et de Java. Rapport de recherche I3S/RR–2001-06–FR, Laboratoire I3S (UNSA/CNRS), avril 2001.

[93] P. Crescenzo. *OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes.* Thèse de doctorat, Université de Nice-Sophia Antipolis, France, décembre 2001.

[94] P. Crescenzo, C. Jalady, and P. Lahire. Annotations of Classes and Inheritance Relationships : an Unified Mechanism in Order to Improve Skills of Library of Classes. In Astudillo et al. [18], pages 1–10.

[95] P. Crescenzo and P. Lahire. Customisation of Inheritance. In A. Black, E. Ernst, P. Grogono, and M. Sakkinen, editors, *Proceedings of the Workshop on Inheritance at ECOOP'02*, page 7, Malaga, Espagne, June 2002. University of Jyväskylä, Finlande. pages 6.

[96] P. Crescenzo, P. Lahire, and D. Pescaru. An Extension of OFL Model through Modifiers. Research Report I3S/RR–2003-31–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, November 2003. pages 36.

[97] P. Crescenzo, P. Lahire, and D. Pescaru. Automatic Profile Generation for OFL-Languages. Research Report I3S/RR–2003-32–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, November 2003. pages 67.

[98] Y. Crespo, J.-M. Marques, and J.-J. Rodriguez. On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In Black et al. [43], pages 30–37.

[99] K. Czarnecki and U. Eisenecker. Synthesizing Objects. In Guerraoui [147], pages 18–42.

[100] K. Czarnecki and W. Eisenecker. *Generative Programming : Methods, Techniques, and Applications*. Addison-Wesley, June 2000.

[101] K. Czarnecki and J. Vlissides. Domain-Driven Development. Special Track at OOPSLA'03 URL : http ://oopsla.acm.org/oopsla2003/files/ddd.html, 2003.

[102] C. Damon and G. Landis. Abstract State and Representation in VBASE. In Gupta and Horowitz [148], pages 178–198.

[103] M. Dao, M. Huchard, T. Libourel, and A. Pons. Extending the Notation for Specialization/Generalization. In Astudillo et al. [18], pages 21–26.

[104] K. De Volder and T. D'Hondt. Aspect-Oriented Logic Meta Programming. In *Proceedings of the Workshop on Aspect-Oriented Programming at ECOOP'98*, Brussels, Belgium, July 1998.

[105] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier 88 - A Database Programming Language ? In Hull et al. [157], pages 179–195.

[106] P. Desfray. *Object Engineering, the Fourth Dimension*. Addison-Wesley Publishing Co., 1994.

[107] O. Deux. *The O_2-Engine Programmer's Manual*. O2-Technology, Versailles, France, July 1991. version 3.1.

[108] O. Deux. *The O_2 Programmer's Manual*. O2-Technology, Versailles, France, July 1991. version 3.0.

[109] O. Deux and al. The Story of O_2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1) :91–108, March 1990.

[110] O. Deux and al. The O_2 System. *Communications of the ACM*, 34(10) :35–48, October 1991.

[111] M. Devos, R. Gabriel, B. Foote, G. Steele, and J. Noble. Resolved : Objects Have Failed (Part 1 of 2) - Panel of OOPSLA'02. http ://oopsla.acm.org/oopsla2002/fp/-files/pan-1.html, November 2002. Detailed comments on http ://www.dreamsongs.com/-Essays.html.

[112] K. Dittrich, editor. *Proceedings of the 2^{nd} Workshop on Advances in Object-Oriented Database Systems*, LNCS(334), Pacific Grove, CA, September 1988. Springer-Verlag.

[113] J. Dowling, T. Schäfer, V. Cahill, P. Haraszti, and B. Redmond. Using Reflection to Support Dynamic Adaptation of System Software : A Case Study Driven Evaluation. In Cazzola et al. [69].

[114] D. D'Souza, A. Sane, and A. Birchenough. First Class Extensibility for UML - Packaging of Profiles, Stereotypes an Patterns. In *Proceedings of UML'99*, LNCS(1723), pages 265–277, Fort Collins, CO, USA, October 1999. Springer-verlag.

[115] S. Ducasse. *Intégration réflexive des dépendances dans un modèle à classes.* Thèse de doctorat, Université de Nice-Sophia Antipolis, janvier 1997.

[116] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna-Dery. A Reflective Model for First Class Dependencies. In Wirfs-Brock [311], pages 265–280.

[117] A. Duclos. Le métamodèle OpenCat V1 : Modèle de sructuration du modèle Unité des données utilisateur (MUDU). Technical report, EDF Pôle Industrie, 2000.

[118] A. Duclos. Principes de modélisation des catalogues MUDU intégrés dans le métamodèle OpenCat V1. Technical report, EDF Pôle Industrie, 2001.

[119] R. Ducournau. "Real World" as an Argument for Covariant Specialization in Programming and Modeling. In Springer-Verlag, editor, *Proceedings of the workshop on advances in Object-Oriented Information Systems at OOIS'02*, pages 3–12, September 2002.

[120] B. Eckel. *Thinking in Java.* Prentice Hall PTR, 3^{rd} edition, December 2002.

[121] Eclipse fundation. Eclipse Environment. http ://www.eclipse.org, 2004.

[122] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance.* Ph.D Thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[123] E. Ernst. Family Polymorphism. In Knudsen [180], pages 303–326.

[124] E. Ernst. Simple, eh ? In *Proceedings of Software Engineering Properties of Languages for Aspect Technologies, SPLAT 2004, in assoc. with AOSD 2004*, 2004.

[125] M. Evered. Unconstraining Genericity. Technical report, University of Ulm, 1997.

[126] M. Fayad, D. Schmidt, and R. Johnson, editors. *Building Application Frameworks.* John Wiley & Sons, 1999.

[127] C. Fernström, K.-H. Närfelt, and L. Ohlsson. Software Factory Principles, Architecture, and Experiments. *IEEE Software*, 9(2) :36–44, March 1992.

[128] D. Fishman and al. Iris : An Object-Oriented DBMS. *ACM Transaction on Office Information Systems*, 5(1) :216–226, January 1987.

[129] D. Fishman and al. Overview of the Iris DBMS. In Kim and Lochovsky [178], pages 219–250.

[130] D. Flanagan. *Java in a Nutshell : a Desktop Quick Reference.* O'Reilly, 3^{rd} edition, December 1999.

[131] K. Flower and K. Scott. *UML Distilled : A Brief Guide to the Standard Object Modeling Language.* Addison-Wesley Professional, 2^{nd} edition, August 1999. pages 185.

[132] S. Ford and al. ZEITGEIST : Database Support for Object-Oriented Programming. In Dittrich [112], pages 23–42.

[133] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Catalogue de modèles de conception réutilisables.* Addison-Wesley Publishing Co., 1999.

[134] J. Garza and W. Kim. Transaction in an Object-Oriented Database System. In Haran Boral [151], pages 37–45.

[135] X. Girod. *Conception par objets - MECANO : une Méthode et un Environnement de Construction d'Applications par Objets.* Thèse de doctorat, université Joseph Fourier, Grenoble I, France, juin 1991.

[136] S. Gjessing and K. Nygaard, editors. *Proceedings of ECOOP'88*, LNCS(322), Oslo, Norway, August 1988. Springer-Verlag.

[137] M. Glandrup. Extending C++ Using the Concepts of Composition Filters. Master Thesis, University of Twente, Netherlands, 1995.

[138] A. Goldberg. *Smalltalk-80 : The Interactive Programming Environment.* series in Computer Science. Addison-Wesley Publishing Co., 1984.

[139] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and its Implementation.* series in Computer Science. Addison-Wesley Publishing Co., 1983.

[140] A. Goldberg and D. Robson. *Smalltalk-80 : the Language.* series in Computer Science. Addyson-Wesley Publishing Co., 1989.

[141] M. Golm and J. Kleinöder. MetaXa and the Future of Reflection. In *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, Vancouver, British Columbia, Canada, October 1998.

[142] M. Gordon, A. Milner, and C. Wadsworth. Edinburgh LCF : A Mechanized Logic of Computation. In *LNCS(78)*, New-york, 1979. Springer-verlag.

[143] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* The Sun Microsystems Press Java Series. Sun Microsystems Inc., June 2000.

[144] B. Gowing and V. Cahill. Meta-object Protocols for C++ : The Iguana Approach. In *Proceedings of Reflection'96*, pages 137–152, San Francisco, CA, April 1996.

[145] N. Graube. Reflexive Architecture : from ObjVLisp to CLOS. In Gjessing and Nygaard [136], pages 110–127.

[146] J. Greenfield and S. Short. Software Factories : Assembling Applications with Patterns, Models, Frameworks and Tools. In R. Crocker and G. L. Steele, editors, *proceedings of OOPSLA'03*, pages 16–27, Anaheim, CA, USA, October 2003. ACM Press.

[147] R. Guerraoui, editor. *Proceedings of ECOOP'99*, LNCS(1628), Lisbon, Portugal, June 1999. Springer-Verlag.

[148] R. Gupta and E. Horowitz, editors. *Object-Oriented Databases with Applications to Case, Networks, and VLSI CAD.* series in Data and Knowledge Base Systems. Prentice-hall Inc., Englewood Cliffs, NJ, 1991.

[149] D. Halbert and P. O'Brien. Using Types and Inheritance in Object-Oriented Languages. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97*, LNCS(1241), pages 20–31, Jyväskylä, Finland, june 1987. Springer-Verlag.

[150] S. Hanenberg and R. Unland. Using and Reusing Aspects in AspectJ. In *Proceedings of the Workshop ASoC at OOSPLA'01*, 2001.

[151] P.-A. L. Haran Boral, editor. *Proceedings ACM SIGMOD International Conference on Management of Data*, volume 17(3), Chicago, Illinois, June 1988. ACM Press.

[152] S. P. Harbison. *Modula-3.* Prentice Hall, 1992.

[153] W. Harrison and H. Ossher. Subject-Oriented Programming - A Critique of Pure Objects. In *proceedings of OOPSLA'93*, pages 411–428, Washington, D.C., United States, September 1993. ACM Press.

[154] A. Hejlsberg. *The C# Programming Language*. Addison-Wesley Pub Co, 1^{st} edition, October 2003.

[155] U. Hlözle. Integrating Independently-Developed Components in Object-Oriented Languages. In O. Nierstrasz, editor, *proceedings of ECOOP'93*, LNCS(707), pages 36–56, Kaiserslautern, Germany, 1993. Springer-Verlag.

[156] E. Horowitz and Q. Wan. An Overview of Existing Object-Oriented Database Systems. In Gupta and Horowitz [148], pages 101–116.

[157] R. Hull, R. Morrison, and D. Stemple, editors. *Proceedings of the 2nd workshop on database programming languages*, Oregon, June 1989. Morgan Kaufmann publishers Inc.

[158] J. Hunter and B. McLaughlin. Java DOM. http ://www.jdom.org, 2004.

[159] Hyper/J team. Hyper/J. http ://www.research.ibm.com/hyperspace/HyperJ/-HyperJ.htm, 2003.

[160] J. Ichbiah and al. Rationale of the Design of the Programming Language ADA. *ACM Sigplan notices,*, 4(6), 1979.

[161] I. Jacobson, G. Booch, and J. Rumbaugh. *Unified Software Development Process*. The Object Technology Series. Addison-Wesley Publishing Co., January 1999.

[162] C. Jalady. Vers une relation d'héritage générique. Mémoire de DEA, Laboratoire I3S (UNSA/CNRS), Sophia antipolis, France, juin 2003. pages 20.

[163] J.-M. Jézéquel, W.-M. Ho, A. Le Guennec, and F. Pennaneac'h. UMLAUT : an Extendible UML Transformation Framework. In R. Hall and E. Tyugu, editors, *Proceedings of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.

[164] J.-M. Jugant and P. Lahire. Lessons Learned with Eiffel 3 : the K2 Project. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17, USA'95, 17^{th} International Conference on "Technology of Object-Oriented Languages and Systems"*, pages 207–215. Prentice Hall Inc., July 1995.

[165] T. Kaehler and G. Krasner. Loom - large object-oriented memory for smalltalk- 80 systems,. In Zdonik and Maier [319], pages 298–307.

[166] S. Keene. *Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS*. Addison-Wesley Publishing Co., 1989.

[167] E. Kendall. Role model designs and implementations with aspect-oriented programming. In *proceedings of OOPSLA'99*, pages 353–369, Denver, Colorado, United States, November 1999. ACM Press.

[168] E. Kendall. Reengineering for Separation of Concerns. In *Proceedings of the Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE'00*, Limerick, Ireland, June 2000.

[169] G. Kiczales, J. Des Rivières, and D. Bobrow. *The Art of the MetaObject Protocol*. MIT-Press, 1991.

[170] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In Knudsen [180].

[171] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with ASPECTJ. *Communications of the ACM*, 44(10) :59–65, October 2001.

[172] G. Kiczales, J. Lamping, A. Mendhekar, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, LNCS(1241), pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[173] W. Kim. Architecture of the Orion Next-Generation Database System. *IEEE Transaction on Knowledge and Data Engineering*, 2(1) :109–124, March 1990.

[174] W. Kim. *Introduction to Object-Oriented Databases*. MIT press, Cambridge, Massachusets, 1991.

[175] W. Kim and al. Composite Object Support in an Object-Oriented Database System. In Meyrowitz [216], pages 118–125.

[176] W. Kim, N. Ballou, H. Chou, and J. Garza. Integrating an Object-Oriented Programming System with a Database System. In Meyorwitz [214], pages 142–152.

[177] W. Kim and H. Chou. Versions of Schema for Object-Oriented Databases. In *Proceedings of the 14th VLDB International Conference*, pages 148–159, Los Angeles, CA, September 1988. Morgan Kaufmann.

[178] W. Kim and F. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. Frontier series. ACM Press, September 1989.

[179] J. Kleinöder and M. Golm. MetaJava : An Efficient Run-Time Meta Architecture for Java. In *International Workshop on Object Orientation in Operating Systems - IWOOOS'1996*, Seattle, Washington, October 1996.

[180] J. L. Knudsen, editor. *Proceedings of ECOOP'01*, LNCS(2072), Budapest, Hungaria, June 2001. Springer Verlag.

[181] P. Koopmans. Sina User's Guide and Reference Manual. Master thesis, Dept. of Computer Science, University of Twente, Netherlands, 1995.

[182] Laboratoire LIP6 - Université Paris 6. *Actes du colloque scientifique en l'honneur de Jean-Fran cois Perrot - 20 ans aprés : où en sont les objets ?*, Paris, France, octobre 2003. Hermés. (à paraître).

[183] P. Lahire. *Conception et réalisation d'un modèle de persistance pour le langage Eiffel*. Thèse de doctorat, Université de Nice Sophia Antipolis, France, mai 1992. pages 343.

[184] P. Lahire, G. Arévalo, H. Astudillo, A. Black, E. Ernst, M. Huchard, T. Oplustil, M. Sakkinen, and P. Valtchev. Report from the ECOOP 2004 Workshop on Mechanisms for Specialization, Generalization and Inheritance (MASPEGHI 2004). In J. Malenfant and B. M. Østvold, editors, *Object-Oriented Technology : ECOOP 2004 Workshop Reader*, LNCS(3344), pages 101–117, Oslo, Norway, June 2004. Springer-Verlag. pages 15, to appear at the end of 2004.

[185] P. Lahire and J.-M. Jugant. *K2 developper's Manual - Global Architecture and Design*. I3S laboratory (UNSA/CNRS), Sophia Antipolis, France, 1^{st} edition, December 1995. Project FAO/ONU - Food Agriculture Organization of the United Nations).

[186] P. Lahire and J.-M. Jugant. *K2 Internal Documentation : listing of source code*. I3S laboratory (UNSA/CNRS), Sophia antipolis, France, 1^{st} edition, December 1995. Project FAO/ONU - Food Agriculture Organization of the United Nations.

[187] P. Lahire and J.-M. Jugant. *K2 system : Modules Demand and SUA*. I3S laboratory (UNSA/CNRS), Sophia Antipolis, France, December 1995. Software delivered to FAO. It contains more than 100 000 lines of code, more than 800 classes and 2 500 pages of internal documentation.

[188] P. Lahire, D. Parigot, C. Courbis, P. Crescenzo, and E. Tundrea. Toward a New Approach for the Development of Software : the Model-Oriented Programming. Research Report I3S/RR–2003-33–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, décembre 2003. pages 25.

[189] P. Lahire, D. Parigot, and E. Tundrea. SMARTFACTORY - an Implementation of the Domain Driven Development Approach. In *SACI'2004, 1st Romanian - Hungarian Joint Symposium on Applied Computational Intelligence*, Timisoara, Roumania, May 2004. pages 6.

[190] P. Lahire and D. Pescaru. Modifiers in ofl - an approach for access control customization. In *Proceedings of the workshop on Encapsulation and Access Rights (WEAR'2003) at OOIS 2003*, Geneva, Switzerland, September 2003. pages 10.

[191] P. Lahire and L. Quintian. New perspective to improve reusability in object-oriented languages. Research Report I3S/RR–2004-XX–FR, I3S laboratory (UNSA/CNRS), Sophia-Antipolis, France, September 2004. pages 20.

[192] M. Lai. *UML : la notation de modélisation objet - Applications en Java*. InterEditions (Masson), 1997.

[193] M. Lanza and S. Ducasse. Beyond Language Independent Object-Oriented Metrics : Domain Independent Metrics. In *QAOOSE 2002 : 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 77–84, 2002.

[194] T. Lawson, C. Hollinshead, and M. Qutaishat. The Potential Reverse Type Inheritance in Eiffel. In B. Magnusson, B. Meyer, and J.-M. Nerson, editors, *Proceedings of TOOLS Europe'94*, Versailles, France, 1994. Prentice-Hall.

[195] C. Lecluse and P. Richard. Modeling Complex Structures in Object-Oriented. In *Proceedings of the 8th ACM PODS*, pages 360–368, March 1989.

[196] T. Ledoux and P. Cointe. Les Métaclasses Explicites comme Outil pour Améliorer la Conception des Bibliothèques de Classes. GDR'95, 1995.

[197] K. Lieberherr. *Adaptive Object-Oriented Software : The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, Massachusetts, 1996.

[198] K. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java. In *Proceedings of ICSE'97*, pages 604–605, Boston, Massachusetts, May 1997. ACM Press.

[199] B. Liskov, L. Shrira, and A. Adya. Providing Persistent Objects in Distributed Systems. In Guerraoui [147], pages 230–257.

[200] C. Lopes and W. Hursch. Separation of Concerns. Technical Report TR NU-CCS-95-03, North-eastern University, Boston, February 1995.

[201] O. Madsen and B. Moller-Pedersen. Virtual Classes : A Powerful Mechanism in Object-Oriented Programming. In Meyrowitz [217], pages 397–406.

[202] D. Maier. Why Database Languages Are a Bad Idea. In *Proceedings of the International Workshop on Database Programming Languages*, pages 277– 287, Roscoff, France, September 1987.

[203] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In Meyrowitz [215], pages 472–482.

[204] D. Maier and S. Zdonik. Fundamentals of Object-Oriented Databases. In Zdonik and Maier [319], pages 1–32.

[205] J. Malenfant and P. Cointe. Aspect-Oriented Programming versus Reflection : a First Draft. In *Position Statement for the OOPSLA'96 AOP meeting*, San Jose, CA, October 1996.

[206] F. Martel. Méta-Modèle de composants. Mémoire de DEA, Laboratoire I3S - Université de Nice-Sophia Antipolis, Sophia antipolis, France, juin 2004. pages 48.

[207] S. McDirmid and W. Hsieh. Aspect-Oriented Programming with Jiazzi. In Aksit [5].

[208] B. Meyer. The Eiffel Language : Libraries. Technical report, Interactive Software Engineering, California, USA, 1987.

[209] B. Meyer. *Object-Oriented Software Construction*. series in Computer Sciences. Prentice-Hall, 1^{st} edition, 1988.

[210] B. Meyer. *Eiffel : The Language*. Object-Oriented Series. Prentice Hall, 1992.

[211] B. Meyer. Systematic Concurrent Object-Oriented Programming. *Communications of the ACM*, 36(9) :56–80, September 1993.

[212] B. Meyer. *Eiffel, le langage*. InterEditions, 1994.

[213] B. Meyer. *Object-Oriented Software Construction*. Professional Technical Reference. Prentice-Hall, 2^{nd} edition, 1997.

[214] N. Meyorwitz, editor. *Proceedings of OOPSLA'88*, volume 23(11), San Diego, California, November 1988. ACM Press.

[215] N. K. Meyrowitz, editor. *Proceedings of OOPSLA'86*, ACM SIGPLAN Notices, Portland, Oregon, October 1986. ACM Press.

[216] N. K. Meyrowitz, editor. *Proceedings of OOPSLA'87*, volume 22(12), Orlando, Florida, October 1987. ACM Press.

[217] N. K. Meyrowitz, editor. *Proceedings of OOPSLA'89*, volume 24(10), New Orleans, Louisiana, October 1989. ACM Press.

[218] M. Mezini and K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proceedings of OOPSLA'02*, volume 37(11) of *ACM SIGPLAN Notices*, pages 52–67, Seattle, Washington, USA, November 2002. ACM Press.

[219] M. Mezini and K. Ostermann. Conquering Aspects with Casear. In Aksit [5], pages 90–99.

[220] M. Mezini, L. Seiter, and K. Lieberherr. Component Integration with Pluggable Composite Adapters. In *Software Architectures and Component Technology : The State of the Art in Research and Practice*. Kluwer Publishing, 2000.

[221] T. Millstein and C. Chambers. Modular Statically Typed Multimethods. In Guerraoui [147], pages 279–303.

[222] J. Mylopoulos, P. Berstein, and H. Wong. A Language Facility for Designing Database Intensive Applications. *ACM Transaction Database Systems*, 5(2) :185–207, June 1980.

[223] S. Nakajima. Separation Of Concerns in Early Stage of Framework Development. In Ossher et al. [247].

[224] Obasco. Activity Report of the OBASCO Project : Objects, Aspects and Components. Technical report, INRIA Rennes - École des Mines de Nantes, France, 2003.

[225] P. O'Brien. Common Object-Oriented Repository System. In Dittrich [112], pages 329–334.

[226] P. O'Brien, B. Bullis, and C. Schaffert. Persistent and Shared Objects in Trellis/Owl. In K. Dittrich and U. Dayal, editors, *Proceedings of the International Workshop on Database Systems and Languages*, pages 113–123, Pacific Grove, California, September 1986. IEEE Computer Society.

[227] M. Odersky and P. Wadler. Pizza into Java : Translating Theory into Practice. In *Proceedings of the 24^{th} ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159, Paris, France, 1997. ACM Press.

[228] A. Oliva. *The Guaranä API.* Institute of Computing of the State University of Campinas, 2001.

[229] OMG. MDA - Model-Driven Architecture. http ://www.omg.org/mda, 2000.

[230] OMG. Object Constraint Language Specification. Version 1.3. Technical report, Object Management Group, March 2000. http ://www.omg.org.

[231] OMG. *XML MetaData Interchange (XMI).* Object Management Group, November 2000. Version 1.1.

[232] OMG. *Meta Object Facility Specification (MOF).* Object Management Group, November 2001. Version 1.3.1.

[233] OMG. *Unified Modeling Language Specification (UML).* Object Management Group, February 2001. Version 1.4.

[234] OMG. *XML MetaData Interchange (XMI) - XML DTDs.* Object Management Group, January 2002. Version 1.2.

[235] OMG. *Unified Modeling Language Specification (UML) - Version 1.5.* Object Management Group (OMG), March 2003. Version 1.5.

[236] OMG. *Unified Modeling Language (UML) Diagram Interchange - Final Adopted Specification.* Object Management Group, September 2003. Version 2.0 - undergoing finalization.

[237] OMG. *Unified Modeling Language (UML) Infrastructure - Final Adopted Specifcation.* Object Management Group, September 2003. Version 2.0 - en cours de finalisation.

[238] OMG. *Unified Modeling Language (UML) OCL - Final Adopted Specification.* Object Management Group, October 2003. Version 2.0 - undergoing finalization.

[239] OMG. *Unified Modeling Language (UML) Superstructure - Final Adopted Specification.* Object Management Group, August 2003. Version 2.0 - undergoing finalization.

[240] OMG. *Meta-Object Facility (MOF) Core - Final Adopted Specification.* Object Management Group, March 2004. Version 2.0.

[241] OMG. *XML MetaData Interchange (XMI) - XML Schema.* Object Management Group, May 2004. Version 2.0.

[242] D. Orleans and K. Lieberherr. DJ : Dynamic Adaptive Programming in Java. In Yonezawa and Matsuoka [317], pages 73–80.

[243] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In A. Paepcke, editor, *Proceedings of OOPSLA'92*, volume 27(10) of *ACM SIGPLAN Notices*, pages 25–40, Vancouver, British Columbia, Canada, October 1992. ACM Press.

[244] H. Ossher, W. Harrison, F. Budinsky, and I. Simmonds. Subject-Oriented Programming : Supporting Decentralized Development of Objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, July 1994.

[245] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-Oriented Composition Rules. In Wirfs-Brock [311].

[246] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and The Hyper-space Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology : The State of the Art in Software Development*. Kluwer, 2000.

[247] H. Ossher, P. Tarr, and G. Murphy, editors. *Proceedings of the Workshop on Multidimensional Separation of Concerns at OOPSLA'99*, Denver, Colorado, USA, November 1999.

[248] K. Ostermann. Dynamically Composable Collaborations with Delagation Layers. In *Proceedings of ECOOP'02*, LNCS(2374), pages 99–110, Malaga, Spain, June 2002. Springer-Verlag.

[249] K. Ostermann and G. Kniesel. Independent Extensibility - an Open Challenge for AspectJ and Hyper/J. In *Proceedings of the Workshop on Aspect and Dimension of Concern at ECOOP'00*, Sophia antipolis et Cannes, France, June 2000.

[250] A. Paepcke. PCLOS : A Flexible Implementation of CLOS Persistence. In Gjessing and Nygaard [136], pages 374–389.

[251] A. Paepcke. PCLOS : A Critical Review. In Meyrowitz [217], pages 221–237.

[252] J. Palsberg and C. Jay. The Essence of the Visitor Pattern. In *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998. IEEE Computer Society.

[253] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A New Approach to Compiling Adaptive Programs. *Science of Computer Programming*, 29(3) :303–326, 1997.

[254] Parc Place Systems. Boss user's guide, 1990.

[255] D. Parigot, C. Courbis, P. Degenne, A. Fau, C. Pasquier, J. Fillon, C. Help, and I. Attali. Aspect and XML-oriented Semantic Framework Generator : SmartTools. In *Proceedings of the LDTA workshop at ETAPS'02*, Grenoble, France, April 2002. Electronic Notes in Theoretical Computer Science (ENTCS).

[256] R. Pawlak. *La Programmation par Aspects Int'eractionnelle pour la Construction d'Application à Préoccupations Multiples*. Thése de doctorat CNAM-CEDRIC, CNAM Paris, décembre 2002.

[257] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic Wrappers : Handling he Composition Issue with JAC. In *proceedings of TOOLS'01*, pages 56–65, 2001.

[258] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A Flexible Efficient Solution for Aspect-Oriented Programming in Java. In Yonezawa and Matsuoka [317], pages 1–24.

[259] J. Penney and J. Stein. Class Modification in the Gemstone Object-Oriented DBMS. In Meyrowitz [216], pages 111–117.

[260] D. Pescaru. *Bridging the Gap between Object Oriented Modeling and Implementation Languages Using a Meta-Language Approach*. Ph.D Thesis, "Politehnica" University of Timisoara, Timisoara, Roumanie, December 2003.

[261] L. Quintian. *JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet*. Thèse de doctorat, Université de Nice-Sophia Antipolis, France, juillet 2004.

[262] P. J. Quitslund and A. P. Black. Java with Traits – Improving Opportunities for Reuse. In Arévalo et al. [15], pages 45–50.

[263] S. Ramakanth. Object-Relational Database Systems - The Road Ahead. *Crossroads*, 7(3) :15–18, April 2001.

[264] J. Richardson and M. Carey. Programming Constructs for Database System Implementation in EXODUS. *SIGMOD record*, 16(3), December 1987.

[265] J. Richardson and M. Carey. Implementing Persistence in E. In J. Rosenberg and D. Koch, editors, *Proceedings of the Third International Workshop on Persistent Object Systems*, pages 175–199. Springer-Verlag, January 1989.

[266] J. Richardson and M. Carey. Persistence in the E Language : Issues and Implementation. *Software Practice and Experience*, 19(12) :1115–1150, December 1989.

[267] F. Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. Thèse de doctorat, Université de Nantes, France, 1997.

[268] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. The Object Technology Series. Addison-Wesley Publishing Co., December 1998.

[269] M. Sakkinen. Exheritance-Class Generalisation Revived. In Black et al. [43].

[270] N. Schaerli, S. Ducasse, and O. Niestrasz. Classes = Traits + States + Glue. Beyond Mixins and Multiple Inheritance. In Black et al. [43].

[271] N. Schaerli, s. Ducasse, O. Niestrasz, and A. Black. Traits : Composable Units of Behaviour. In *Proceedings of ECOOP'03*, LNCS(2743), page 25, Darmstat, June 2003. Springer-Verlag.

[272] N. Schirmer. Analysing the Java Package/Access Concepts in Isabelle/HOL. In *Proceedings of the Workshop on Formal Techniques for Java-like Programs (fJP'2002) at ECOOP'02*, Malaga, Spain, June 2002.

[273] G. Schlageter, R. Unland, and al. OOPS - an Object-Oriented Programming System with Integrated Data Management Facility. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 118–125, Los Angeles, California, February 1988. IEEE Computer Society.

[274] S. Schmitz. OFL-Meta - un éditeur graphique pour le modèle OFL. Stage de deuxième année de l'École Supérieure en Sciences Informatiques de l'Université de Nice-Sophia Antipolis, septembre 2002.

[275] C. Shaffert and al. An introduction to Trellis/Owl. In Meyrowitz [215], pages 9–16.

[276] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1) :140–173, March 1981.

[277] A. Silberschatz, M. Stonebraker, and J. Ullman. Database Systems : Achievements and Opportunities. *Communications of the ACM*, 34(10) :110–120, October 1991.

[278] J. Skaller. Article 20339. dans comp.lang.C++, 1993.

[279] A. Skarra and S. Zdonik. The Management of Changing Types in an Object-Oriented Database. In Meyrowitz [215], pages 483–495.

[280] A. Skarra and S. Zdonik. Type Evolution in an Object-Oriented Database. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–415. MIT Press, 1987.

[281] A. Skarra and S. Zdonik. Concurrency Control and Object-Oriented Databases. In Kim and Lochovsky [178], pages 395–422.

[282] Y. Smaragdakis and D. Batory. DiSTiL : A Transformation Library for Data Structures. In *proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[283] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In E. Jul, editor, *Proccedings of ECOOP'98*, LNCS(1445), pages 550–570, Brussels, Belgium, July 1998. Springer-Verlag.

[284] J. Smith, S. Fox, and T. Landers. *ADAPLEX : Rationale and Reference Manual*. Computer Corporation of America, Cambridge, Massachussets, 2^{nd} edition.

[285] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In Meyrowitz [215], pages 38–45.

[286] SoftTeam. UML Profiles and the J Language : Totally Control your Application Development. http ://www.softeam.fr/pdf/us/uml_profiles.pdf, 1999.

[287] Y. V. Srinivas and R. Jullig. Specware(TM) : Formal Support for Composing Software. Technical Report KES.U.94.5, Kestrel Institute, 1994.

[288] M. Stiegler. The E Language in a Walnut. http ://www.skyhunter.com/marcs/ewalnut.html, 2000.

[289] M. Stonebraker, P. Brown, and D. Moore. *Object-Relational DBMSs : Tracking the Next Great Wave*. Morgan-Kauffman Publishers, San Francisco, 2^{nd} edition, 1999.

[290] B. Stroustrup. *Le langage C++*. Addison-Wesley, 2*me* edition, 1992.

[291] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Co., 2^{nd} edition, January 1994.

[292] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., 3^{rd} edition, 1997.

[293] Sun microsystems Inc. Java Architecture for XML Binding JAXB. http ://java.sun.com/xml/jaxb/index.jsp, 2004.

[294] G. Sunyé, F. Pennaneac'h, W.-M. Ho, A. Le Guennec, and J.-M. Jézéquel. Using UML Action Semantics for Executable Modeling and Beyond. In *proceedings of CAISE'01*, LNCS(2068), pages 433–447. Springer-Verlag, june 2001.

[295] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[296] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3) :438–479, September 1996.

[297] P. Tarr, J. Wileden, and A. Wolf. A Different Track to Providing Persistence in a Language. In Hull et al. [157], pages 41–60.

[298] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava : A Class-based Macro System for Java. In Cazzola et al. [69], pages 119–135.

[299] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Ditribued Execution of "Legacy" Java Software. In Knudsen [180], pages 236–255.

[300] TELESYSTEMES and all. Business Class Project Number 5311. http ://www.newcastle.research.ec.org/esp-syn/text/5311.html#top, February 1991.

[301] S. Thibault. *Langage Dédiés : Conception, Implémentation et Application*. Thèse de doctorat, Université de Rennes 1, France, octobre 1998.

[302] TNI Europe. Opentool : an Extensible Tool Set for UML Analysis, Design and Code Generation. http ://www.tni-world.com/opentool.asp, 2002.

[303] W. Vanderperren and B. Wydaeghe. Separating Concerns in a High-Level Component-Based Context. In *Proceedings of the Workshop EasyComp at ETAPS'02*, 2002.

[304] B. Vanhaute, B. De Win, and B. De Decker. Building Frameworks in AspectJ. In Brichau et al. [52].

[305] W3C. XML Schema. http ://www.w3.org/XML/Schema, 2004.

[306] S. Weiser and F. Lochovsky. OZ+ : an Object-Oriented Database System. In Kim and Lochovsky [178], pages 309–337.

[307] I. Welch and R. Stroud. Dalang - A Reflective Extension for Java. Technical Report CS-TR-672, Computing Science Department, University of Newcastle upon Tyn, September 1999.

[308] I. Welch and R. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In Cazzola et al. [69], pages 155–167.

[309] J. Whishman. ComposeJ The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Master's thesis, University of Twente, December 1999.

[310] J. Wileden, A. Wolf, C. Fisher, and P. Tarr. PGRAPHITE : an Experiment in Persistent Typed Object Management. In *SIGSOFT'88 : Third Symposium on Software Development Environments*, pages 130–142, Boston, Massachusetts, November 1988. ACM Press.

[311] R. J. Wirfs-Brock, editor. *Proceedings of OOPSLA'95*, volume 30(10), Austin, Texas, USA, October 1995. ACM Press.

[312] World Wide Web Consortium. *XML Schema Part 1 : Structures - W3C Recommendation*, 1^{st} edition, May 2001. Version - May 2001.

[313] World Wide Web Consortium. *XML Schema Part 2 : Datatypes - W3C Recommendation*, 1^{st} edition, May 2001. Version - May 2001.

[314] World Wide Web Consortium. *Document Object Model (DOM) Level 3 Core Specification - W3C Recommendation*, 1^{st} edition, April 2004. Version 1.0.

[315] World Wide Web Consortium. *Document Object Model (DOM) Level 3 Load and Save Specification - W3C Recommendation*, 1^{st} edition, April 2004. Version 1.0.

[316] World Wide Web Consortium. *Extensible Markup Language (XML) - W3C Recommendation*, 3^{rd} edition, February 2004. Version 1.0.

[317] A. Yonezawa and S. Matsuoka, editors. *Proceedings of the third International Conference, Reflexion'01*, LNCS(2192), Kyoto, Japon, January 2001. Springer-Verlag.

[318] S. Zdonik. Maintening Consistency in a Database with Changing Types. In *Object-Oriented Programming Workshop*, volume 21(10) of *ACM SIGPLAN Notice*, Yorkton Heights, NY, USA, June 1986.

[319] S. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufmann, January 1990.

[320] S. Zdonik and P. Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, pages 155–171, Hawaii, January 1986.