Construction de systèmes par application de modèles paramétrés

Alexis Muller

N° d'ordre: 3816

THÈSE

Présentée devant

devant l'Université de Lille 1

pour obtenir

le grade de : Docteur de l'Université de Lille 1
Mention Informatique

par

Alexis Muller

Équipe d'accueil : Goal
École Doctorale : Sciences pour l'Ingénieur
Composante universitaire : LIFL

Titre de la thèse :

Construction de systèmes
par application de modèles paramétrés

soutenue le 26 juin 2006 devant la commission d'examen

Président :	Pr. Sophie	Tison
Rapporteurs :	Pr. Marie-Pierre	Gervais
	M. Antoine	Beugnard
Examinateurs :	M. Philippe	Lahire
	M. Bernard	Carré
Directeur :	Pr. Jean-Marc	Geib

Remerciements

Je tiens à remercier tous ceux qui, directement ou indirectement, m'ont aidé dans l'aboutissement de ce travail. Je remercie Sophie Tison pour m'avoir fait l'honneur d'accepter de présider mon jury. Merci également à Marie-Pierre Gervais et à Antoine Beugnard d'avoir accepté de rapporter cette thèse et à Philippe Lahire d'avoir accepté de faire partie de mon jury. Je les remercie de l'intérêt qu'ils ont témoigné à l'égard de mon travail.

Merci à Jean-Marc Geib de m'avoir accueilli dans son équipe et de l'honneur qu'il me fait d'être mon directeur de thèse. Un immense merci à Olivier Caron, Bernard Carré et Gilles Vanwormhoudt pour tout ce qu'ils m'ont appris du métier de chercheur, pour leur soutien, leurs encouragements, ainsi que pour leurs nombreuses et attentives relectures. J'espère avoir acquis un peu de leur honnêteté et rigueur scientifique. Je tiens également à les remercier sur un plan plus personnel de la confiance qu'ils m'ont témoigné et du temps qu'ils m'ont accordé.

Merci également à ceux avec qui j'ai eu l'occasion de partager un bureau, Olivier Caron, Jean-François Roos, Emmanuel Renaux, Raphaël Marvie, ce fut toujours avec plaisir et dans une très bonne ambiance. Je tiens également à remercier toutes les personnes avec qui j'ai eu l'occasion de travailler et d'apprendre, notamment les membres du projet RNTL ACCORD.

Merci à Olivier Barais. Je peux à mon tour lui témoigner mon amitié et le remercier pour nos discussions et "collaborations", enrichissantes autant sur le plan personnel que professionnel.

Merci aussi à tous les membres de l'équipe[1] pour nos discussions, scientifiques ou non, qui ont fait des pauses café un moment toujours agréable. Mention spéciale à Nicolas Pessemier pour nos discussions cinéphiles ;)

Je tiens également à remercier mes collègues de l'IUT pour leur accueil et Philippe Mathieu pour m'avoir enseigné et fait apprécier, il y a maintenant quelques années, la programmation orientée objet.

Merci enfin à ma famille pour leurs encouragements et spécialement à ma femme Angélique pour sa patience et son soutien.

[1] Que je n'ose citer par peur d'oublier quelqu'un.

Table des matières

Première partie

Introduction

Chapitre 1

Problématique

1.1 Axes de complexité des systèmes

Malgré une évolution constante des techniques de génie logiciel, la conception de systèmes informatiques reste une tâche difficile. En effet, en parallèle de cette évolution, les besoins ont également évolué. Pour être efficace, le système d'information d'une organisation doit maintenant être unique et partagé entre tous les services de cette organisation. Les services de production, de gestion des stocks, des ventes,... jusqu'au site internet accessible aux clients doivent par exemple, partager des informations communes. Mais ces services ont également besoin d'informations et de traitements propres à leur fonction devant cohabiter dans le même système. Ces types de besoins sont désignés, suivant les approches, *aspect fonctionnel* ou *préoccupation métier*.

La nature même de ces systèmes d'information, partagés entre tous les services de l'organisation, leur impose d'autres types de besoin. Ils doivent être multi-utilisateurs, différents acteurs utilisent en même temps différentes fonctionnalités, et largement répartis, les parties du système s'exécutent simultanément sur des machines différentes. Ils doivent également assurer la persistance des informations ainsi que leur sécurité. Ces types de besoins sont désignés par *aspects non-fonctionnels* ou *préoccupations techniques*. Ceux-ci ne sont pas spécifiques à un seul service mais sont transverses à plusieurs, voire à tous les services de l'organisation.

Ceci conduit à l'accroissement de la complexité des systèmes qui se répercute naturellement dans toutes les étapes de leur cycle de vie : analyse des besoins, conception, développement, tests, exploitation, maintenance,... De plus, comme pour toute discipline d'ingénierie, les délais de conception sont de plus en plus courts, les coûts doivent être réduits, tout en améliorant la fiabilité [OMG01].

Enfin, les étapes du cycle de vie s'appuient sur de nombreuses technologies elles-mêmes non pérennes. Les plates-formes d'exécution par exemple, permettant de prendre en compte le caractère réparti des systèmes, sont nombreuses et de nouvelles apparaissent régulièrement [Sa00]. Il n'est pas rare qu'une entreprise doive adapter son système afin d'utiliser une nouvelle plate-forme d'exécution. Cependant la logique métier et les choix techniques sont souvent entrelacés dans le système, si bien que cette

adaptation à une nouvelle plate-forme prend souvent la forme d'une re-conception quasi complète du système.

La complexité des systèmes vient donc de deux facteurs : la grande quantité de fonctionnalités que ceux-ci doivent prendre en compte et le grand nombre de préoccupations techniques à satisfaire. On retrouve donc deux grandes familles de structuration. Les approches de structuration en *couches verticales* proposent de découper le système selon ses fonctionnalités. Inversement, les approches de structuration en *couches horizontales* proposent un découpage selon des niveaux d'abstractions. Cela pouvant être vue comme une structuration en niveaux de détails. La couche la plus élevée représentant le système sans aucun détail technique, la couche la plus basse étant l'implantation du système. Cette vision est illustrée par la figure 1.1 extraite de [DSo01]. On trouve dans la dimension horizontale, différentes couches correspondantes aux différentes fonctions du système, *marketing*, *engineering*, *sales*, *IT*, *services*. Chacune de ces couches étant elle-même structurée en niveaux d'abstractions dans la dimension verticale. Notons, comme l'illustre cette figure, qu'une troisième dimension, correspondant aux différentes versions du système dans le temps, peut également être considérée comme un axe de structuration.

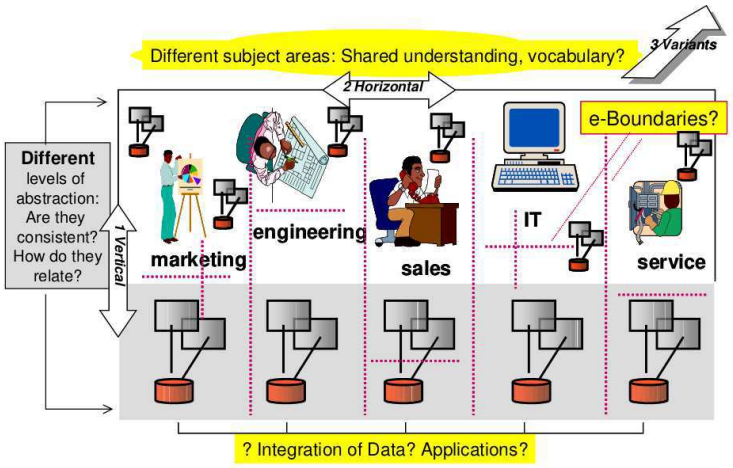

FIG. 1.1 – Dimensions de structuration d'un système [DSo01]

1.2 Structuration et réutilisation

Afin de maîtriser cette complexité, les techniques de génie logiciel ont depuis longtemps recours à des méthodes de structuration des systèmes. Celles-ci permettent de faciliter la conception et la compréhension en considérant le système comme un ensemble de parties clairement identifiées [Par72]. Ces parties pouvant être conçues et étudiées indépendamment les unes des autres.

Certaines dimensions de préoccupation pouvant être utilisées pour la structuration

d'un système [Bar98], ont été identifiées dans le domaine du développement logiciel orienté aspect (Aspect Oriented Software Development, AOSD) [FECA05]. La dimension d'un système relative à ses utilisateurs est appelée la dimension *sujet*. Plusieurs approches proposent un découpage du système par rapport à cette dimension, soit au niveau du code avec la programmation orientée-sujet (Subject-Oriented Programming, SOP) [HO93, OKK+96], soit au niveau design avec SOD (Subject-Oriented Design) [Sio01] ou Catalysis [DW99], ou encore nos travaux sur la structuration par vues en CROME [Van99, Deb98, CCD00a]. Une autre dimension identifiée est la dimension *aspect*. À l'origine celle-ci ne concernait que les préoccupations techniques du système (persistance, sécurité,...). Mais les approches comme l'Aspect-Oriented Programming (AOP) [Ka97], ont également été utilisées pour manipuler des préoccupations métiers [Ken99].

Pour faire face aux délais de conceptions et de tests toujours plus courts, la réutilisation de parties déjà conçues et validées parait indispensable [McI69]. Cet état de fait est présenté dans [CS99] qui propose un état de l'art sur la réutilisation dans l'ingénierie des systèmes d'information. Les éléments réutilisables sont appelés composants réutilisables et peuvent aussi bien être des programmes, des modèles conceptuels ou encore des documents.

Afin de rendre efficace l'ingénierie des systèmes d'information par réutilisation, de nombreux problèmes sont encore à résoudre [EG00]. En effet, pour rendre cette ingénierie exploitable, il est nécessaire de clairement identifier et valider des composants réutilisables, de pouvoir les stocker. Ces besoins se rapportent à la réalisation de bibliothèques de composants réutilisables. Au niveau de la conception de systèmes d'information à partir de ces bibliothèques, d'autres problèmes sont à résoudre. Il est d'abord nécessaire de permettre la recherche du ou des composants adéquats dans ces bibliothèques [Sha95]. Enfin il est indispensable de permettre l'adaptation et l'intégration de ces composants comme un tout cohérent afin de former le système dans son ensemble. Ce sont ces deux derniers points (adaptation et intégration) auxquels nous nous intéressons principalement dans cette thèse.

La conception et la programmation par objets offrent depuis plusieurs années une méthode de structuration efficace [Mey88a, Weg90]. Elles offrent également un premier niveau de réutilisation [Cox84, Mey88b] illustré par les bibliothèques de structures de données (listes, piles,...) et de composants d'interfaces graphiques disponibles pour de nombreux langages de programmation et largement (re-)utilisées.

Cependant, cette réutilisation à fine granularité n'est pas suffisante [PY93]. En effet l'assemblage de ce type de composant reste à la charge du développeur [CS99] et n'est exploitable que pour de petits éléments du système à un niveau programmation. Afin d'améliorer l'efficacité de la réutilisation, il est nécessaire de considérer des composants exprimant des structures complètes. Ce type de composant est appelé composant complexe [CS99] et doit permettre la réutilisation, non plus de simples éléments, mais de véritables structures ou *framework* [Gar90, SBF96].

Si l'utilisation de composants complexes doit permettre un meilleur rendement dans la conception de systèmes, leur intégration en est encore plus complexe. Pourtant, comme cela est énoncé dans [DW99], il est important de pouvoir composer les modèles

et les spécifications d'une façon claire et intuitive. Cette clarté rend la réutilisation et l'apprentissage plus faciles, puisqu'il est possible de comprendre le tout en comprenant les parties et en les recombinant d'une façon prévisible [Pol45].

L'adaptation de ce type de composants est également plus complexe. C'est cette capacité d'adaptation, aussi appelé principe de variabilité [Par76, Nei89, SG93], qui permet de rendre un composant générique et de faciliter sa réutilisation. Ce principe définit pour un composant une partie fixe et une partie variable. La partie variable peut être exprimée sous différentes formes, par des paramètres ou des points de choix par exemple.

1.3 Vers l'ingénierie des modèles

L'utilisation des techniques de modélisation en génie logiciel n'est pas nouvelle. Elles sont depuis longtemps utilisées pour concevoir, représenter, documenter... les systèmes informatiques [Boo94, Jac92, RBP+91]. Ce qui est plus récent et mis en avant par les approches comme Model Driven Architecture [MDA, Fra03] (MDA) de l'Object Management Group [OMG01] (OMG) ou l'ingénierie dirigée par les modèles (Model Driven Engineering MDE) [Ken02], est de placer les modèles au cœur des processus de génie logiciel.

Depuis quelques années maintenant, le langage UML (Unified Modeling Language) [BRJ98] promu par l'OMG, s'est imposé comme le standard pour l'expression de modèles de systèmes informatiques. Il repose sur une architecture à quatre niveaux [Bez01, FEBF06]. Le langage UML (niveau M2) permet de définir des modèles (niveau M1) qui eux-mêmes doivent être instanciés pour obtenir le système lui-même (niveau M0). Le langage UML est lui-même défini à l'aide d'un méta-modèle (niveau M3). Le langage standard de l'OMG pour la définition de méta-modèles est le MOF (Meta-Object Facility). Le MOF devant lui-même être défini, pour ne pas ajouter de méta-niveaux supplémentaires, il permet de se définir lui-même. Cette architecture est illustrée figure 1.2 [UML05a].

Pour faire face à la complexité croissante des systèmes informatiques et pour répondre aux besoins de réutilisation, l'OMG a d'abord proposé l'intergiciel[1] CORBA [OMG02] (Common Object Request Broker Architecture). Celui-ci, multi-langages, multi-plates-formes, devait être l'intergiciel ultime en permettant l'interopérabilité des applications et en proposant un ensemble d'objets standards pour des domaines d'activités spécifiques [MZ95, OMG97] (contrôle aérien, finance, biologie,...). Pour plusieurs raisons, cette initiative a échoué. De nouveaux intergiciels apparaissent et apparaîtront continuellement, l'utilisation des objets standards ne sont disponibles qu'au niveau implantation, à la charge du programmeur et ne pouvant pas être intégrés dès la phase de conception.

Face à ce constat, l'OMG a proposé l'architecture MDA. Son objectif est la séparation des spécifications fonctionnelles (ou métiers) d'un système de ses spécifications tech-

[1] Un intergiciel ou middleware permet de fournir un ensemble de services (répartition, persistance,...) aux applications qui s'y exécutent.

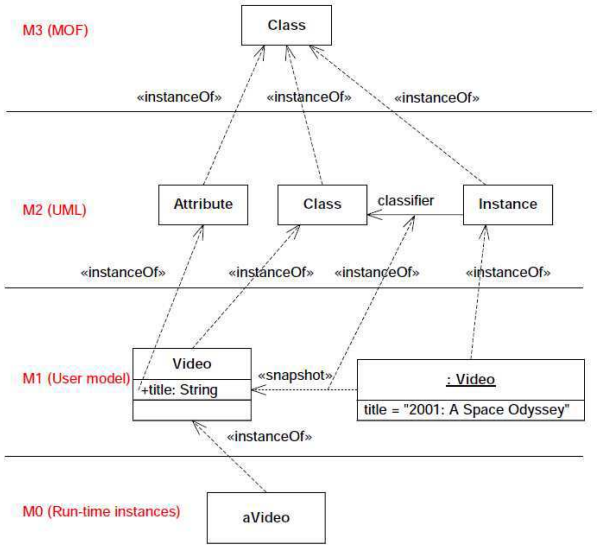

FIG. 1.2 – Hiérarchie de méta-modélisation à quatre niveaux [UML05a].

niques relatives à sa mise en œuvre sur une plate-forme particulière. Cette séparation a un avantage indéniable : elle permet de capitaliser les spécifications fonctionnelles des systèmes et ainsi de les réutiliser pour leur mise en œuvre sur différentes plates-formes. Cette idée correspond à une structuration du système selon des niveaux d'abstraction (axe vertical) évoquée figure 1.1.

Pour mettre en œuvre cette idée, l'OMG place les modèles au cœur de l'approche MDA, tout y est modèle, des schémas fonctionnels jusqu'aux codes source et binaire. Deux types de modèle y sont définis : les PIM pour Platform Independent Model et les PSM pour Platform Specific Model [MM01]. Un modèle indépendant d'une plate-forme (PIM) signifie que celui-ci ne contient aucun détail propre à une réalisation particulière. Il ne modélise que la partie métier du système sans considération technique. Un modèle spécifique à une plate-forme (PSM) au contraire contient des éléments nécessaires à sa réalisation sur une plate-forme, ou une famille de plates-formes, particulière.

L'OMG identifie quatre étapes dans la réalisation d'un système selon l'architecture MDA [Poo01, Mul02]. La première consiste en la réalisation d'un modèle abstrait du système : le modèle PIM. La seconde consiste à raffiner celui-ci en lui ajoutant des niveaux de détails, mais toujours sans aucune considération technique relative à une quelconque plate-forme d'exécution. Ce n'est que lors de la troisième étape que celle-ci est choisie et que le modèle est adapté à celle-ci. Le modèle résultant est dès lors considéré comme PSM. La dernière étape consiste à raffiner ce PSM jusqu'à obtenir le code exécutable du système sur cette plate-forme. Les étapes 2 et 4 pouvent être répétées autant de fois que nécessaire.

Dans la vision de l'OMG, chacun des modèles produits lors de ces différentes étapes peut être conservé et exploité à nouveau en cas de changement de plate-forme cible

(étape 3) de changement de choix technique (étape 4) ou de choix métier (étape 2). L'architecture MDA suggère également d'automatiser au maximum chacune de ces étapes. Pour y arriver, l'OMG propose d'utiliser ses technologies standards, le langage UML pour décrire les modèles, ou au moins un langage dont le méta-modèle est exprimé à l'aide du MOF. Le CWM (Common Warehouse Metamodel) pour permettre le stockage des modèles intermédiaires. Et enfin le langage XMI (XML Metadata Interchange) pour permettre l'expression et l'échange de modèles sous forme textuelle en XML. L'OMG travaille également sur un standard de manipulation (modifications, requêtes,...) de modèles nommé QVT [QVT05] (Query View Transformation).

Bien que séduisante, cette vision soulève de nombreuses questions. En pratique, cette séparation entre les choix spécifiques à une plate-forme et les choix relevant de considérations métiers n'est pas aussi évidente. De plus, l'architecture MDA ne permet d'exprimer que la dimension de structuration verticale des systèmes, selon ses niveaux d'abstractions. Les autres dimensions, ne sont pas prises en compte [Ken02]. Enfin, l'architecture MDA est contrainte par les technologies de l'OMG. Face à ces constats, une nouvelle idée de recherche a émergé pour compléter les faiblesses de l'architecture MDA, le Model Driven Engineering (MDE) ou l'ingénierie dirigée par les modèles [Ken02, GSCK04].

L'idée du MDE est plus générale que celle du MDA. Elle propose de structurer l'espace des modèles dans toutes ses dimensions, métier, technique, niveaux d'abstraction, etc. Chaque préoccupation du système pouvant être représentée par un modèle distinct. Se pose alors le problème de la mise en relation de ces différents modèles pouvant être exprimés dans des langages différents. Dans le cadre de l'ingénierie dirigée par les modèles, cette mise en relation ne peut rester implicite ou même définie par un ensemble de règles vagues [Ken02]. En effet, pour être efficace, le MDE vise à faire des modèles de véritables artefacts manipulables, si possible à l'aide d'outils.

Il est alors tentant de faire de ces modèles des artefacts réutilisables pour la construction de différents systèmes. Ils paraissent en effet de bons candidats à la réutilisation. Ils correspondent à de véritables structures complètes, disponibles dès la phase de conception du système. De plus, conformément à l'idée du MDE, leur assemblage devrait être clairement défini et vérifiable. Il est alors possible d'imaginer des ateliers de conception de systèmes par composition de modèles réutilisables.

Cependant, avant d'atteindre ce but, de nombreux problèmes sont encore à résoudre. Comment faire des modèles de véritables composants réutilisables ? Comment exprimer notamment cette partie variable permettant de les rendre génériques ? Comment exprimer et valider la composition de modèles ? Une fois ces problèmes résolus se pose encore le problème de leur mise en œuvre. La structuration introduite au niveau conception peut-elle être conservée à l'exploitation ? Comment assurer la traçabilité des préoccupations des modèles jusqu'au code ? Qu'en est-il de l'évolutivité des systèmes ? Est-il possible d'exprimer au niveau conception l'ajout dynamique d'une préoccupation au système ? Notre proposition vise à apporter une première réponse à ces questions.

Chapitre 2

Contribution

Cette partie nous a permis de mettre l'accent sur la complexité croissante des systèmes d'information actuels devant répondre à un grand nombre de préoccupations aussi bien métiers (gestion des ventes, des stocks, marketing,. . .) que techniques (sécurité, persistance, répartition,. . .).

Cette complexité affecte toutes les étapes du cycle de vie des systèmes alors que les besoins de productivité et de fiabilité sont de plus en plus importants. Dans ce contexte, les technologies et les plates-formes d'exécution sont elles-mêmes non pérennes et sujettes à remplacements et à modifications. Elles ne peuvent donc être utilisées comme invariants dans la conception des systèmes.

Il est possible de distinguer deux dimensions de structuration principales dans la conception de systèmes d'information. Les couches verticales correspondent aux différentes préoccupations métiers, les couches horizontales à des niveaux d'abstraction (de détails).

Il existe un grand nombre d'approches proposant de structurer les systèmes selon leurs dimensions métiers. Afin d'améliorer la productivité et la qualité, certaines d'entre elles proposent également la réutilisation de composants. Cependant, dans ces approches, les composants sont soit trop simples (une classe) limitant l'efficacité de la réutilisation, soit difficiles à intégrer. En effet, les composants arrivent souvent tard (implantation) dans le cycle de développement.

L'OMG introduit avec l'architecture MDA, l'idée d'exprimer clairement la structuration des systèmes selon des niveaux d'abstractions (couches verticales). Les deux types de niveaux mis en avant étant indépendants des plates-formes (PIM) et spécifiques à une plate-forme (PSM). Pour ce faire, l'OMG propose de placer les modèles au cœur des démarches de conception.

Pour faire de l'architecture MDA une réalité, l'OMG propose d'exploiter ses standards : comme le langage UML pour la modélisation, le langage MOF pour la métamodélisation, ou QVT pour la transformation de modèles.

Cette idée est séduisante mais l'architecture MDA ne prend en compte que cette seule dimension de structuration (niveaux d'abstraction). Afin de palier cette limitation, une nouvelle approche de conception a été proposée : l'ingénierie dirigée par les modèles

ou MDE. Celle-ci propose de structurer l'espace des modèles dans toutes ses dimensions et de faire des modèles de véritables artefacts manipulables.

Structure du document

Nous étudions dans la partie suivante différentes approches proposant des solutions aux besoins de structuration et de réutilisation de modèles. Nous y présentons notamment les approches de structuration par vues, Catalysis, Subject Oriented Design et Theme/UML. Après avoir présenté ces approches, nous les étudions au travers de différents critères permettant d'évaluer leurs apports au niveau de la réutilisation et de l'expression de systèmes complexes. Enfin, cet état de l'art présente également l'état actuel des standards de modélisation et de méta-modélisation que sont UML et MOF.

La troisième partie présente le cœur de notre proposition. Après avoir expliqué et expérimenté nos idées de structuration et de réutilisation au travers d'un modèle de vues génériques (chapitre 5), nous présentons au chapitre 6 notre proposition pour la construction de systèmes par composition de modèles génériques. Nous y proposons un moyen d'expression de fonctionnalités métiers au travers de modèles paramétrés que nous appelons composants de modèle. Nous y étudions ensuite un moyen de concevoir un système complet par compositions de ces modèles génériques. Nous avons pour cela défini un opérateur de composition ayant de bonnes propriétés pour la construction de systèmes complexes. Ces propriétés concernent l'ordre dans lequel les différents modèles génériques peuvent être composés. Sa formalisation et les preuves de ses propriétés sont présentées chapitre 7. Une fois le modèle du système construit par composition d'un ensemble de modèles génériques, le modèle résultant peut être exprimé selon différents paradigmes. Ces différentes expressions du modèle résultant sont décrites chapitre 8.

Toute cette partie est volontairement indépendante d'un langage de modélisation particulier. Mais pour faciliter son intégration dans les approches et outils existants, la quatrième partie se concentre sur les problèmes d'expression et de méta-modélisation des éléments de notre approche à l'aide des langages standards de modélisation et de méta-modélisation que sont UML et MOF. Le chapitre 9 décrit le méta-modèle, les contraintes qui lui sont associées et le profil UML permettant l'expression des composants vues expérimentés au chapitre 5. Au chapitre 10, nous étudions et formalisons la relation standard de manipulation d'éléments de modèle paramétrés. Nous y faisons le lien entre cette relation et notre opérateur de composition. Il est également nécessaire de permettre l'expression de nos composants de modèle et de leurs compositions au moyen d'éléments standards. C'est ce que nous proposons de faire par extension du méta-modèle UML2.

La cinquième partie est consacrée à la mise en œuvre de systèmes exprimés par composition de modèles. Le chapitre 11 présente différents patrons de conception permettant de préserver jusqu'à l'exploitation les qualités de traçabilité et de structuration de notre approche. Le chapitre 12 propose une technique de transformation permettant aux concepteurs de guider la mise en œuvre d'un système à partir de son modèle. Nous utilisons cette technique, dans le cadre de notre approche, pour privilégier telle ou telle

propriété lors de la mise en œuvre de systèmes construits par composition de modèles. Au chapitre 13, nous présentons différentes réalisations basées sur des technologies telles que EJB, Fractal ou CORBA et s'appuyant sur nos patrons de conception. Celles-ci permettent ainsi de préserver la structure et la trace des différentes fonctionnalités utilisées pour la construction du système, voire même suivant la technologie utilisée, de pouvoir ajouter et supprimer dynamiquement ces fonctionnalités lors de l'exécution du système. Nous y présentons également l'atelier de génie logiciel supportant les différents aspects de notre proposition que nous avons réalisé.

Enfin dans la dernière partie, nous concluons sur notre proposition et lui donnons un ensemble de perspectives.

Deuxième partie

Etat de l'art

Chapitre 3

Approches de structuration et de réutilisation de modèles

Comme nous l'avons évoqué en introduction, les idées de structuration et de réutilisation des systèmes visant à maîtriser la complexité et de réduire les coûts et les temps de développement ne sont pas nouvelles [WPD91, FI94, Pou95]. La programmation par objets en est certainement l'un des exemples les plus marquants. Celle-ci permet de structurer un système en un ensemble d'entités, chacune d'elles encapsulant les données et traitements qui lui sont relatifs. L'impressionnante bibliothèque Java illustre également la capacité de réutilisation des objets pour des structures de données riches (listes, piles,...) et traitements standards.

Cependant, face à la taille croissante des systèmes, l'approche objet n'est pas suffisante. En effet, celle-ci n'autorise que la seule dimension de structuration selon les entités, les différentes préoccupations que les systèmes doivent prendre en compte se retrouvent enfouies, entrelacées et éparpillées dans les entités [Van99, Sio01]. C'est de ce constat que sont nées les approches de programmation par aspects [Ka97] ou par sujets [HO93], ajoutant à la programmation objet une nouvelle dimension de structuration.

De plus, la réutilisation au niveau programmation de structures complexes n'est pas aisée. Pour être efficace, la réutilisation doit être considérée dès la phase de conception du système. C'est dans cette optique qu'ont été proposés les patrons de conception [GHJ+95], permettant de capturer une solution générique à un problème récurrent. Cependant, ces patrons correspondent plus à un ensemble d'idées devant être adaptées pour un problème particulier qu'à de véritables artefacts réutilisables.

Cette partie propose une étude des approches de structuration de modèles existantes sous l'angle de l'ingénierie dirigée par les modèles. Nous présentons dans un premier temps chacune des approches, puis nous les comparons par rapport à un ensemble de critères relatifs à la réutilisation dans l'ingénierie dirigée par les modèles.

La première section de ce chapitre présente des approches permettant de structurer les modèles de conception d'un système, mais ne proposant pas explicitement de mécanismes pour leur réutilisation. Les approches présentées dans la deuxième section proposent également des moyens de structuration des modèles, mais apportent en plus

une dimension générique permettant la réutilisation de certaines parties.

3.1 Approches pour la structuration de modèles

3.1.1 CROME

L'approche CROME [Van99, Deb98] propose d'enrichir l'approche objet afin de permettre une double structuration des systèmes suivant ses objets et ses fonctions. Elle utilise la notion de point de vue comme outil de cette structuration.

Dans cette approche, un schéma de base (ou plan de base) recense les éléments communs aux différentes fonctions du système. Les schémas vues (ou plans fonctionnels) quant à eux ajoutent au schéma de base un certain nombre d'éléments propres à leur fonction. Cette approche permet de tenir compte de l'orthogonalité objets/fonctions. En effet, pour un système donné, les objets peuvent intervenir dans plusieurs fonctions et celles-ci peuvent utiliser un certain nombre d'objets. Tous les objets n'interviennent pas dans toutes les fonctions, de même les fonctions n'utilisent pas systématiquement tous les objets. Si l'approche objet permet de découper le système en entités distinctes, elle ne permet pas de séparer les différentes fonctions à l'intérieur des objets. Le mécanisme des vues ajoute ce découpage et rend ainsi possible la conception des fonctions du système indépendamment les unes des autres tout en garantissant leur compatibilité.

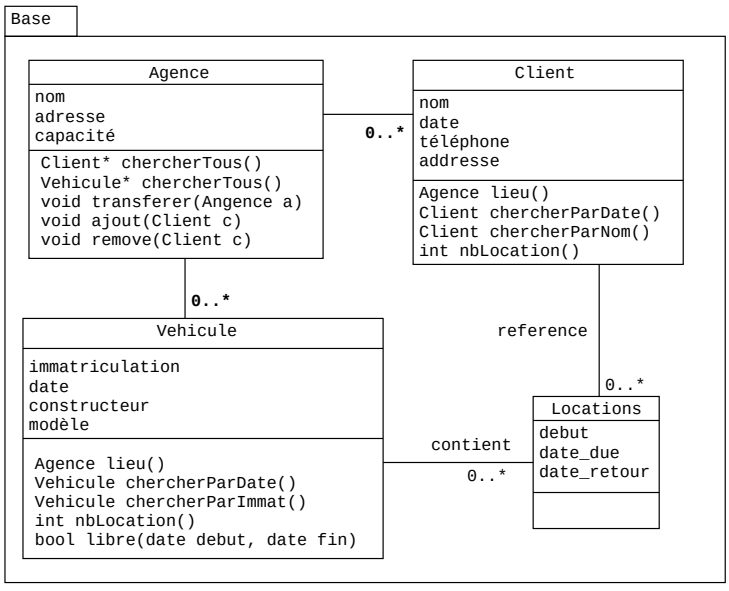

FIG. 3.1 – Location de voitures - Approche objet

Pour illustrer cette orthogonalité, considérons l'exemple d'un système de location de véhicules (cf figure 3.1). Grâce à l'approche objet, le découpage en entités distinctes est bien représenté. On y trouve les entités *Agence*, *Client*, *Vehicule* et *Locations*, mais

toutes les fonctions sont confondues. L'activité de gestion des voitures (transférer, ajouter, supprimer), par exemple, n'a pas d'intérêt à considérer la liste des clients. De même, l'activité consistant à rechercher à quelle agence est rattaché un client n'est pas concernée par la durée ou le nombre de locations de celui-ci. L'approche CROME permet de séparer les fonctions du système et permet de les réaliser indépendamment les unes des autres (cf figure 3.2). Les attributs communs à toutes les fonctions sont déclarés dans le schéma de base, les autres attributs et méthodes sont déclarés dans la vue où ils sont utilisés. Prenons en exemple la classe *Agence*, les attributs *nom* et *adresse*, qui sont utiles à toutes les fonctions du système, sont déclarés dans la base. L'attribut *capacité* et les méthodes *ajout*, *transférer* et *supprimer*, qui ne servent qu'à la gestion des ressources, sont déclarés dans la vue correspondante.

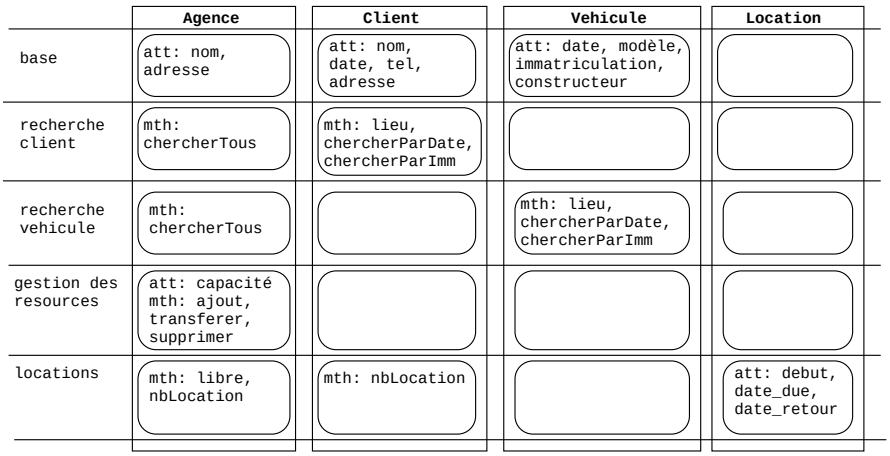

FIG. 3.2 – Location de voitures - Structuration par fonctions

D'abord proposée dans le contexte des bases de données à objets et des langages de programmation [VCD97, CCD00b], l'approche CROME peut également s'appliquer au niveau des modèles [CCD00a, MCCV03]. Elle étudie les problèmes et contraintes d'intégrité, de partage d'associations et d'interactions entre les différents plans.

3.1.2 Catalysis : Composition de modèles et de spécifications

Catalysis [DW99] est une méthodologie orientée objet proposant d'unifier les concepts d'objets, de frameworks et de composants. Elle propose un cadre d'utilisation des objets, des frameworks et de la notation UML pour concevoir, construire et réutiliser des systèmes à base de composants. Elle étudie notamment comment les patrons peuvent être exprimés à l'aide de frameworks de modèle.

Son but est de permettre la conception rapide de modèles, par adaptation et composition de frameworks génériques avec d'autres frameworks spécifiques à un domaine.

Elle s'inspire des travaux d'application de méthodes rigoureuses aux systèmes à objets [RBP+91, Wil91], et à l'utilisation des collaborations comme unité de conception

[RWL95].

Catalysis propose le découpage du système en couches horizontales et verticales. Les couches verticales correspondent à un découpage fonctionnel du système par rapport à des points de vue de différentes catégories d'utilisateurs. Les couches horizontales permettent un découpage suivant les préoccupations techniques du système. Dans cette approche, ce sont les paquetages qui sont utilisés pour représenter les différentes coupes du système. Afin de permettre l'expression de l'intégration de ces différentes coupes, l'approche introduit une nouvelle relation appelée *join*.

Cette relation *join* est basée sur la relation UML *import*, et précise que les éléments de même nom doivent être, par défaut, fusionnés. Les classes utilisées pour la définition d'une nouvelle classe par la relation de *jonction* sont appelées définitions partielles.

L'approche donne un ensemble de règles suivant le type des éléments à composer. La *jonction* de deux paquetages correspond à former un ensemble contenant tous les éléments définis dans ces deux paquetages et à réaliser la *jonction* de tous les éléments de même nom. Le résultat de la *jonction* de classes consiste en une classe contenant tous les attributs et toutes les méthodes de ses définitions partielles.

La figure 3.3 présente son utilisation. Les paquetages *Finance* et *Faults* sont tous les deux joints dans le paquetage *Telecoms Network Implementation*. Tous les éléments des paquetages joints sont copiés et les éléments de même nom sont fusionnés, illustré ici par l'élément *Call*.

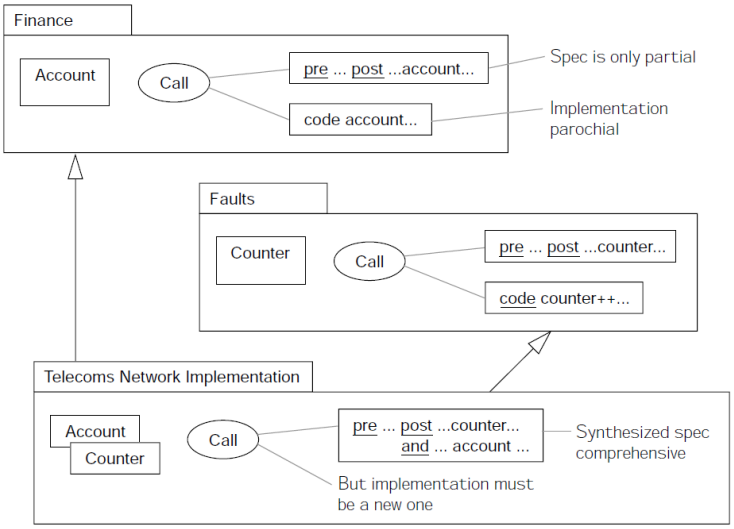

FIG. 3.3 – Jonction de paquetages Catalysis [DW99]

L'approche permet d'exprimer la composition d'un ensemble de paquetages mais ne permet pas de donner un ordre d'évaluation sur ces compositions. Les règles données en langage naturel imposent, par exemple, que les attributs joints lors de la *jonction* de classes doivent être de même type.

Bien qu'il soit possible de renommer les éléments lors de l'*import*, les artefacts de modélisation manipulés par la relation *join* ne sont pas paramétrables. Ils représentent une analyse particulière d'un système. Ils ne permettent donc pas l'expression d'éléments génériques réutilisables. Pour cela, l'approche Catalysis propose une construction particulière que nous présentons dans la section suivante (3.2.1) : les Frameworks de modèles.

3.1.3 Subject-Oriented Design (SOD)

L'approche Subject-Oriented Design [Sio01] essaie de répondre au problème de discordance entre la spécification des besoins d'un système logiciel et sa spécification orientée objets. Il en résulte qu'un même besoin se trouve éclaté dans les différentes unités de conception et qu'une même unité doit supporter de multiples besoins. Ce qui nuit à la compréhension, à la traçabilité et à la réutilisation des modèles de conception [Cla02].

Cette approche propose d'étendre le modèle orienté objet par de nouvelles capacités de décomposition. Ce qui doit permettre d'aligner directement les modèles de conception avec chaque besoin. Pour cela, un modèle doit être réalisé pour chaque préoccupation. Celui-ci contient également les concepts du domaine pouvant apparaître dans de multiples préoccupations.

Ces modèles sont appelés *Sujet* et prennent la forme de paquetages de modèles UML standard. Un nouveau type de relation (*CompositionRelationship*) est proposé pour permettre l'expression de la composition des éléments issus de différents sujets. Les éléments doivent être mis en relation, soit de façon explicite (en traçant un lien entre les éléments correspondants), soit implicitement, dans ce cas les éléments de même nom sont mis en correspondance. Il est également possible de définir des exceptions, i.e. des éléments de même nom ne devant pas être mis en correspondance.

L'approche SOD propose ensuite deux stratégies de composition. La première, *merge*, fusionne les éléments mis en correspondance. La seconde, *override*, permet de remplacer un élément d'un *sujet* par l'élément mis en relation d'un autre *sujet*. Dans cette seconde stratégie, la mise en relation est donc orientée.

La figure 3.4 présente la composition de deux *sujets*. L'un permet de gérer les ajouts de ressources (*AddResource*) l'autre les suppressions de ressources (*RemoveResource*). Les éléments de même nom de ces deux *sujets* sont fusionnés par la directive *match_name* entre les deux paquetages. Les deux classes *Resource* n'en formeront donc plus qu'une (contenant l'ensemble des attributs et méthodes) dans le modèle résultant, de même pour les classes *Copy*. Un lien est tracé entre les classes *ResourceManager* et *Manager* afin d'indiquer, bien que ces classes n'aient pas le même nom, que celles-ci doivent être fusionnées.

L'approche propose une modification du méta-modèle UML 1.4 ainsi qu'un ensemble de contraintes OCL afin de vérifier la cohérence des compositions. Celles-ci permettent notamment de vérifier que les structures des éléments composés sont compatibles. Par exemple, deux attributs ne peuvent être composés que si leur classe respective sont elles-mêmes composées. Des règles sont également définies afin de permettre une évaluation sans ambiguïté d'un ensemble de compositions.

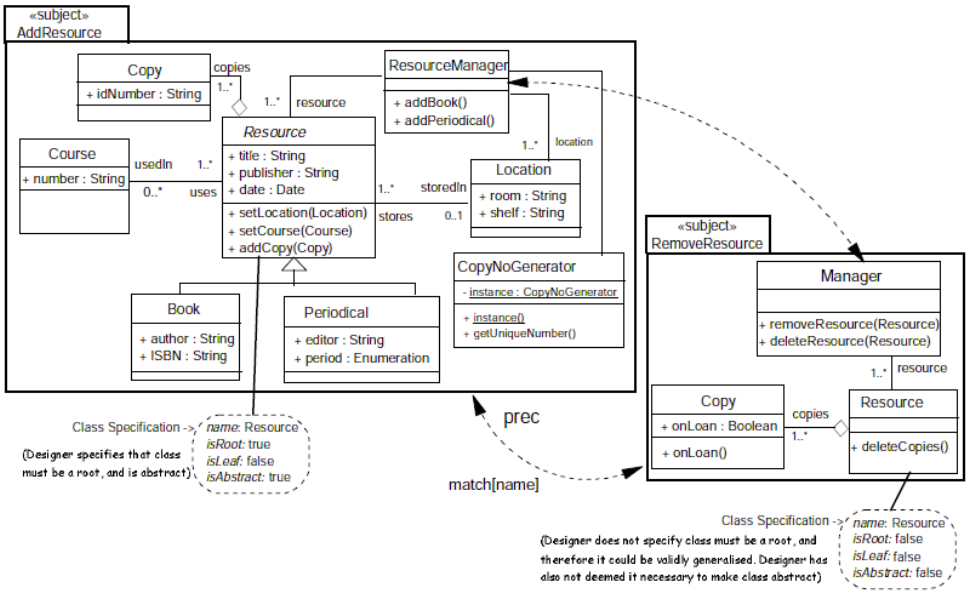

FIG. 3.4 – Composition de *sujets* [Sio01]

La simplicité d'utilisation de cette approche est basée sur le fait que les différents sujets sont conçus à partir de l'analyse d'un même système, et que par conséquent, une grande partie des éléments représentant les mêmes concepts ont le même nom. Les relations de composition s'en trouvent ainsi simplifiées ne devant être "détaillées" que lorsque cette propriété n'est pas respectée.

Dans l'approche SOD les *sujets* ne sont pas conçus pour être réutilisables. S. Clarke propose dans [Cla02] un rapprochement de l'approche SOD et des templates UML permettant de paramétrer les *sujets*. Cette idée est reprise dans l'approche Theme que nous présentons dans la section 3.2.2.

3.2 Approches pour la réutilisation de modèles

Les approches présentées dans la section précédente permettent de maîtriser la complexité des systèmes par une meilleure structuration. Cependant, elles ne sont pas conçues pour répondre au besoin de réutilisation. Les techniques introduites dans cette section présentent des extensions de ces approches. Nous présentons également les techniques de modélisation orientées aspect (AOM).

3.2.1 Catalysis : frameworks de modèle et paquetages templates

Afin de permettre la définition d'éléments de modélisation génériques, l'approche Catalysis propose l'utilisation de frameworks. Ceux-ci prennent la forme de paquetages

paramétrés. Ces paramètres, appelés *placeholders*, correspondent au nom d'un élément de modèle qui peut être substitué lorsque le framework est utilisé.

La figure 3.5 présente un framework de gestion d'allocation de ressources selon l'approche Catalysis. Les paramètres sont identifiés à l'aide de chevrons (< >). Ce sont ici les classes *JobCategory*, *Job*, *Facility* et *Resource*. Ce framework permet d'établir des liens entre ces concepts et de définir des règles sous forme d'invariants.

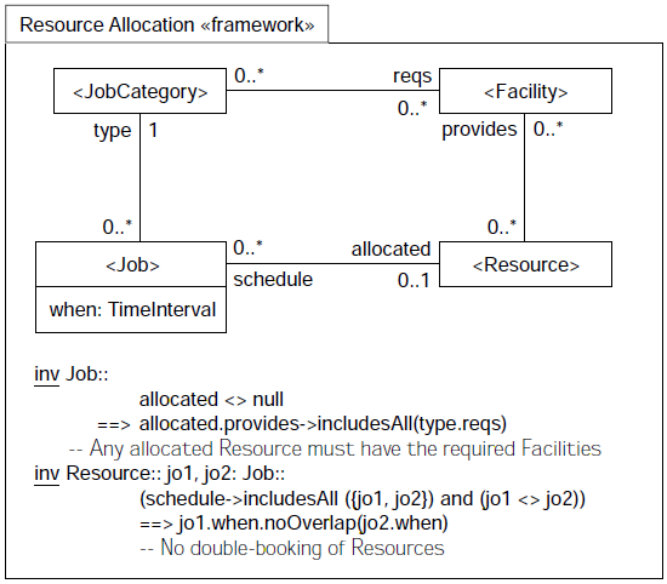

FIG. 3.5 – Framework d'allocation de ressources [DW99]

La figure 3.6 illustre l'utilisation de ce framework. Les classes paramètres sont ici respectivement associées aux classes *JobDescription*, *Job*, *Skill* et *Plumber* du système pour lequel on souhaite utiliser ce framework d'allocation des ressources. Le résultat de cette utilisation correspond à ajouter les liens et les invariants qui sont définis par le framework sur les éléments utilisés en paramètres. L'association définie dans le framework entre *JobCategory* et *Facility* est par exemple ajoutée entre *JobDescription* et *Skill* dans le système.

L'approche précise qu'il est possible de concevoir un framework à partir d'autres frameworks. Il n'y a pas de règle pour l'application de framework. Les *placeholders* permettent de suggérer quels éléments doivent être renommés afin d'utiliser le framework. Mais cela n'est pas imposé, ils peuvent être non substitués et correspondent alors à la définition de nouveaux éléments. De même, des éléments n'étant pas identifiés en tant que *placeholder* peuvent être renommés. En cela, les frameworks de l'approche Catalysis s'apparentent plus aux patrons de conception, devant être adaptés pour être utilisés, qu'à de véritables artefacts de modélisation manipulables et réutilisables.

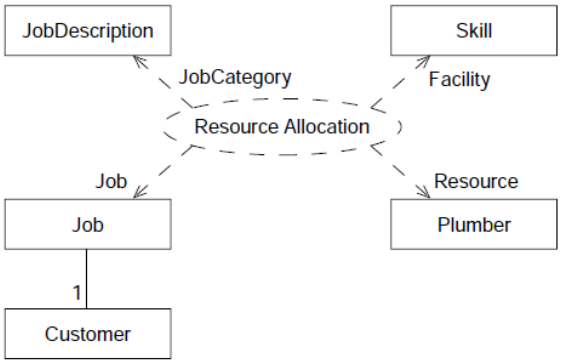

FIG. 3.6 – Utilisation du framework d'allocation de ressources [DW99]

3.2.2 Theme

Theme [CW05, BC04] est une approche pour l'analyse et la conception orientée aspects. Les aspects sont des préoccupations transverses et réparties dans un système. L'approche Theme se situe donc à deux niveaux : au niveau analyse *Theme/Doc* et au niveau design (modèle) *Theme/UML*. Theme/Doc aborde le problème de l'identification des aspects dans la phase d'analyse des besoins. Theme/Doc propose pour cela de représenter les relations entre les fonctionnalités du système à l'aide d'un graphe d'analyse.

Au niveau conception, *Theme/UML* permet de modéliser les fonctionnalités et les aspects d'un système, et de spécifier comment ceux-ci doivent être composés. À ce niveau, l'approche Theme reprend et étend les idées de SOD.

Un *Theme* correspond à un paquetage template, c'est-à-dire à un paquetage paramétré. L'ensemble des paramètres d'un tel paquetage sont listés dans un rectangle en pointillé dans son coin supérieur droit. Ce sont par exemple *Logged* et *_log* dans l'exemple figure 3.7. Une relation (nommée *bind*) permet d'exprimer la composition paramétrée de deux Themes. La vérification de l'assemblage est réalisée en ajoutant au graphe d'analyse (Theme/Doc) les informations relatives aux différentes compositions.

L'approche Theme est conçue pour permettre la définition de traitements génériques, comme des fonctionnalités de trace ou de synchronisation. Chaque paramètre correspond donc à une classe et à l'ensemble de ses méthodes sur lesquelles la fonctionnalité doit être "tissée". Cette modification du traitement est décrite à l'aide de diagrammes de séquences.

La figure 3.7 illustre cela pour la définition d'une fonctionnalité générique de trace. Les paramètres pour ce *theme* sont ici la classe *Logged* et la méthode *_log*. La figure 3.8 illustre l'utilisation de ce *theme*. La relation *bind* entre le *theme Logger* et le *theme CMS* porte le paramétrage permettant d'appliquer la fonctionnalité de trace aux classes *Person*, *Student* et *Professor* définies dans le paquetage *CMS* (Course Management System) à leurs méthodes respectives *register*, *unregister* et *giveMark*.

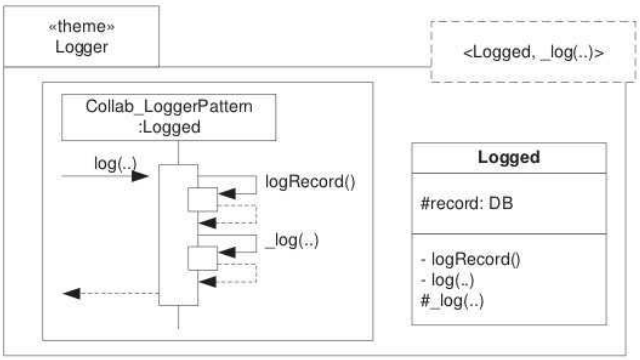

FIG. 3.7 – Theme d'une fonctionnalité de log [BC04]

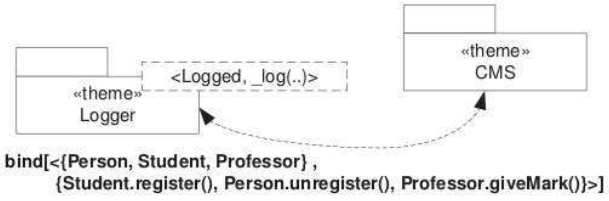

FIG. 3.8 – Utilisation du Theme de log [BC04]

3.2.3 Aspect Oriented Modeling

L'idée des techniques de modélisation par aspects découlent de celles de la programmation par aspects et proposent de considérer les aspects dans les modèles [CRS+05]. Deux grandes familles d'approche sont à distinguer. La première regroupe les approches visant à modéliser les programmes aspects [BS03, BGL04]. Elles proposent donc des constructions permettant de faire apparaître dans un modèle objet les éléments propres à la programmation par aspects comme les points de jonctions ou les coupes. Leur but n'est donc pas la structuration des modèles, mais la représentation de détails techniques pour la mise en œuvre par un langage d'aspects. Les approches de la seconde famille proposent quant à elles de structurer les modèles selon différents aspects. Elles restent au niveau des modèles de conception et ne font pas apparaître les constructions techniques propres aux langages de programmation par aspects.

C'est à cette seconde famille d'approches que les travaux de R. France et *al.* [SGS+04] appartiennent. Ils permettent la représentation de modèles d'aspects génériques dont les paramètres sont identifiés à l'aide de l'élément syntaxique "|" (pipe). Ces modèles d'aspects doivent ensuite être instanciés avant d'être composés à un modèle primaire. Ce modèle primaire correspond au modèle de base du système auquel les différents modèles d'aspects devront être ajoutés. L'instanciation des modèles d'aspects permet

d'adapter l'aspect au contexte particulier défini par le modèle primaire. Cela est réalisé en paramétrant les modèles d'aspects par des éléments de l'espace de nommage du modèle primaire.

Les modèles d'aspects contextualisés et le modèle primaire peuvent ensuite être composés. Les éléments de même nom du modèle d'aspect et du modèle primaire sont par défaut fusionnés. Cela peut être modifié par des directives de manipulation de modèles : création d'éléments, renommage, ajout et suppression d'éléments dans un espace de nommage, remplacement d'un élément par un autre.

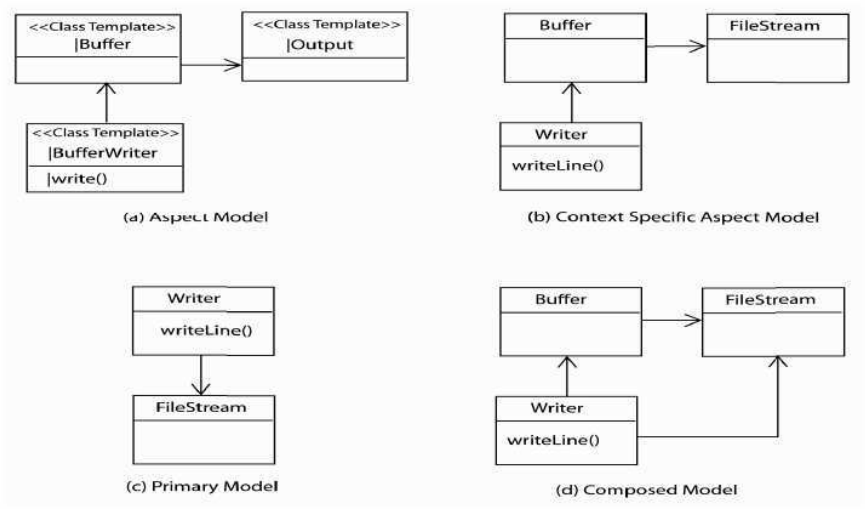

FIG. 3.9 – Exemple de composition [SGS+04]

La figure 3.9 illustre ce mécanisme pour un aspect d'écriture par l'intermédiaire d'un tampon. Le modèle (a) illustre le modèle d'aspect générique. Les trois classes *BufferWriter*, *Buffer*, *Output* ainsi que la méthode *write* sont définies, à l'aide de l'élément syntaxique "|" comme paramètre de l'aspect. Celui-ci est contextualisé (b) par rapport au modèle primaire (c). Les éléments *Output*, *BufferWriter* et *write* sont renommés par des noms appartenant à l'espace de nommage défini par le modèle primaire, soit respectivement, *FileStream*, *Writer* et *writeLine*. L'élément *Buffer* n'ayant pas de correspondance dans le modèle primaire, celui-ci n'est pas renommé dans le modèle d'aspect contextualisé. Le modèle (d) présente le modèle résultant de la composition du modèle primaire et du modèle d'aspect contextualisé. Les éléments de même nom sont fusionnés.

Les directives permettent de résoudre les conflits et incohérences pouvant apparaître dans le modèle composé. Elles permettent également d'imposer un ordre de composition des différents aspects. L'approche ne donne pas de contrainte sur le paramétrage ou sur la composition, l'identification et la résolution de ces incohérences restent à la charge du concepteur.

3.3 Discussion

Toutes ces approches permettent la conception d'un système par composition d'éléments de modèle. Certaines d'entre elles proposent également des mécanismes permettant la réutilisation d'éléments de modélisation pour la conception de différents systèmes. Afin d'évaluer la possibilité de les utiliser pour l'expression de composants de modèle tels que nous les avons définis en introduction, nous analysons dans ce chapitre l'ensemble de ces approches par rapport aux critères suivants :

- **artefacts manipulables :** L'approche permet-elle la définition d'éléments de modèle manipulables au sens des méthodologies MDA/MDE ? Leur composition est-elle automatisable ? Ce critère permet de mesurer l'effort nécessaire pour exprimer sans ambiguïté les éléments de modélisation de l'approche et pour les intégrer dans un outil de modélisation.
- **généricité :** L'approche propose-t-elle des constructions permettant la définition d'éléments génériques, pouvant être utilisés pour la conception de différents systèmes ? Ce critère étudie si l'approche permet d'identifier clairement une partie variable dans les éléments de modélisation utilisés et de quelle façon celle-ci est exprimée.
- **compositions des éléments génériques entre eux :** L'approche permet-elle de composer les éléments les uns aux autres ? Peut-on définir des chaînes de composition ? L'approche propose-t-elle des règles d'évaluation ou des équivalences pour ces chaînes de composition ? Afin de concevoir de nouveaux éléments de modélisation génériques complexes à partir d'éléments génériques plus simples, il est nécessaire de pouvoir les composer entre eux. Ce critère permet d'étudier si l'approche prend en compte ce besoin.
- **ordres de compositions :** L'approche étudie-t-elle le problème de l'ordre des compositions ? Est-il possible de définir dans quel ordre les éléments doivent être composés ? L'ordre dans lequel les éléments sont composés influe-t-il sur le résultat ? Ce critère permet d'étudier si l'approche tient compte de l'ordre dans lequel les différents éléments peuvent être composés.
- **vérification de la composition :** L'approche propose-t-elle des règles permettant de vérifier la validité des éléments et de leurs compositions ? Ces règles sont-t-elles formalisées ? À quel moment ces règles peuvent-elles être vérifiées (à la conception des éléments, à l'expression de la composition, sur le modèle résultant,...) ? Ce critère permet de déterminer jusqu'à quel point l'approche définit formellement la composition et ainsi jusqu'à quel point les modèles obtenus peuvent être validés.
- **granularité des paramètres :** De quel type sont les paramètres, nom (chaîne de caractères), classe, modèle ? Un grand nombre d'approches utilise la notion de paramètre pour exprimer la partie variable des éléments génériques. Cependant, le type de paramètre utilisé n'est pas le même pour chacune d'elle. Ce critère permet d'étudier le niveau de paramétrage et la granularité de l'approche.

3.3.1 Artefacts manipulables

Même si certaines approches [SGJ00, FKGS04] visent à en faire des artefacts manipulables, les design patterns [GHJ+95] sont avant tout des idées de conception pour la résolution d'une famille de problèmes. Ils doivent être adaptés pour résoudre un problème précis, et se prêtent donc mal à la définition d'artefacts manipulables. Dans l'approche Catalysis, certains éléments peuvent être considérés comme des artefacts manipulables, mais ce n'est pas le cas des frameworks de modèles qui, dans le même esprit que les design patterns, représentent "une idée" de solution devant être adaptée "à la main". Les éléments paramètres ne sont que des suggestions pour le concepteur.

Les approches SOD, Theme et AOM, au contraire, sont basées sur des méta-modèles, souvent par extension de celui de UML, ce qui facilite l'utilisation d'artefacts manipulables. En effet, ces méta-modèles permettent de définir clairement quels sont les éléments constituant les modèles et comment ils peuvent être liés. Ce qui permet de contraindre et de vérifier les modèles exprimés dans ces approches, ceux-ci pouvant être manipulés de façon automatique à l'aide d'outils.

3.3.2 Généricité

Certaines de ces approches sont conçues pour permettre un meilleur découpage des modèles de systèmes, mais ne s'intéressent pas à la réutilisation possible de morceaux de ce découpage. C'est notamment le cas des approches SOD et CROME qui proposent de structurer un système selon différentes préoccupations, mais qui ne considèrent pas leur réutilisation pour la conception d'autres systèmes. De ce fait, les sujets ou les plans fonctionnels sont conçus pour structurer un système bien particulier et ne disposent pas d'éléments adaptables.

D'autres approches au contraire mettent la réutilisation et donc la conception d'éléments génériques en avant ; soit au niveau d'idées de conception avec les design patterns, soit à l'aide d'éléments paramétrables avec les approches Theme ou AOM. Les frameworks de modèles de Catalysis étant à mi-chemin entre les deux approches.

3.3.3 Composition des éléments génériques entre eux

L'intérêt principal de composer des éléments génériques entre eux est de permettre la construction de nouveaux éléments génériques plus complexes. Ces éléments génériques complexes peuvent à leur tour être composés entre eux ou utilisés pour la construction d'un système.

Ces possibilités sont très peu étudiées par les approches considérées. L'approche Catalysis supporte la conception d'un framework de modèle à partir d'un autre. Mais elle n'étudie pas plus en avant les questions soulevées par ce type de construction, comme la possibilité de transférer les paramètres non consommés du framework utilisé vers le modèle construit. La plupart des approches basées sur l'utilisation d'éléments paramétrés [CW05, SGS+04] étudient la composition de ses éléments à un modèle de base, mais non la composition de ces éléments paramétrés entre eux, ce qui ne permet pas la définition de chaînes de composition.

3.3.4 Ordre des compositions

La possibilité de concevoir un système par composition d'un ensemble de modèles, soit à l'aide de chaînes de composition, soit en les composant à un même modèle de base, soulève des problèmes sur l'ordre dans lequel ces compositions doivent être évaluées. L'approche SOD identifie un certain nombre de problèmes pouvant survenir dans le cas de compositions multiples et propose de les détecter par un ensemble de contraintes. L'approche Theme/UML n'étudie pas ce problème. Il est néanmoins possible de remonter les informations relatives aux différentes compositions au niveau du modèle d'analyse (Theme/Doc) et ainsi d'identifier des incohérences potentielles. L'approche [SGS+04] prend en compte explicitement le problème en proposant d'exprimer impérativement dans quel ordre les modèles doivent être composés. Aucune des approches étudiées ne propose de règles permettant de garantir des propriétés de cohérence suivant l'ordre de composition des modèles. Elles n'étudient pas non plus la possibilité d'identifier des ordres de composition équivalents entre eux.

3.3.5 Vérification de la composition

La composition de modèles valides ne donne pas forcément un modèle lui même valide. En effet, l'ordre dans lequel les éléments sont composés, la façon dont ils sont composés peut produire un modèle incohérent. Afin de traiter ce problème, deux techniques sont envisageables : la première consiste à vérifier le modèle résultant de la composition et à y détecter les incohérences. C'est celle utilisée par les frameworks de modèles Catalysis ou par l'approche [SGS+04].

La seconde technique consiste à donner des règles de composition et à vérifier si celles-ci sont bien respectées. C'est la technique utilisée par SOD qui fournit un ensemble de règles écrites à l'aide du langage OCL. L'avantage de cette technique est de pouvoir vérifier l'assemblage composition par composition et ainsi de localiser plus facilement les incohérences.

Toutes les approches ne donnent pas des règles de compositions aussi précises que l'approche SOD, permettant d'automatiser la vérification de l'assemblage. La composition de modèles de Catalysis par exemple donne en langage naturel quelques règles à respecter lors de la composition. Dans l'approche CROME, des règles d'interaction entre différentes vues sont formellement définies [Deb98]. Mais les vues étant conçues par rapport à une base, ces règles ne visent pas à vérifier leur composition.

3.3.6 Granularité des paramètres

Certaines des approches permettant la conception de modèles génériques font appel à des techniques de paramétrage. C'est le cas des frameworks de modèles de Catalysis et des approches Theme, [SGS+04] et [SGJ00]. Cependant la notion de paramètre n'a pas la même signification pour toutes ces approches. Dans Catalysis, les paramètres ne sont que des suggestions données par le concepteur du Framework. L'utilisateur est libre de les remplacer ou non. De même il peut utiliser d'autres éléments non désignés comme étant paramètres. Dans l'approche [SGS+04], les paramètres permettent d'instancier

les modèles d'aspect pour un contexte particulier. Cela est réalisé par renommage des éléments désignés comme paramètre. Dans cette approche, le paramétrage correspond donc à un ensemble de noms. La granularité des paramètres est plus importante dans l'approche Theme. Le but de l'approche est de modifier les comportements d'une classe en modifiant ses méthodes par composition de diagrammes de séquences. Un paramètre correspond donc à une classe et un ensemble de méthodes. Un Theme peut être paramétré par un ensemble de classes indépendantes.

Aucune des approches étudiées ne considère la possibilité d'utiliser un diagramme (ensemble de classes reliées par associations) comme paramètre, permettant ainsi l'expression de modèles eux-mêmes paramétrés par un modèle. Cette possibilité semble pourtant importante pour permettre l'expression de modèle requis autorisant la composition de fonctionnalités plus complexes et plus riches.

3.4 Bilan

Les tableaux suivants résument l'évaluation des différentes approches. Le tableau 3.1 regroupe les approches ne proposant pas la définition d'éléments paramétrés (composition de modèles de Catalysis, CROME et SOD). Le second tableau regroupe les approches permettant l'utilisation d'éléments paramétrés (les frameworks de Catalysis, Theme et les travaux de R. France).

Plus ancienne, l'approche CROME n'est pas conçue pour permettre la définition d'éléments de modélisation sous forme d'artefacts manipulables. La réutilisation des vues n'est pas non plus considérée. Les règles de composition des vues et le partage d'éléments entre différentes vues sont par contre formellement définies.

L'approche SOD est basée sur une extension du méta-modèle UML permettant la définition d'artefacts de modélisation manipulables. L'expression de la composition des *sujets* est également définie par cette extension du méta-modèle. Des règles de composition, exprimées à l'aide du langage OCL, permettent de vérifier la cohérence des modèles composés. Mais l'approche ne supporte pas la définition d'éléments génériques.

Nous avons étudié pour l'approche Catalysis, deux constructions différentes : la composition de modèles et les frameworks de modèles. L'approche Catalysis proposant l'utilisation du langage UML, les modèles utilisés pour la première construction peuvent être considérés comme des artefacts manipulables. Mais les règles de composition sont définies en langage naturel, ce qui ne permet pas une vérification stricte de l'assemblage. Il n'est pas non plus question d'éléments génériques. Cela est proposé par la seconde construction : les frameworks de modèles. Mais ce sont des idées de conception dont l'utilisation n'est pas contrainte par un ensemble de règles. Il est possible d'utiliser les deux constructions pour la conception d'un même système, mais pas simultanément. En effet, il est d'abord nécessaire d'utiliser le framework avant de pouvoir composer le modèle obtenu avec d'autres.

Les travaux de R. France et *al.* sont basés sur le langage UML. Ils proposent l'utilisation de modèles génériques pour la définition d'aspects dont l'adaptation à un système (contexte) particulier se fait par renommage de certains éléments. Le problème de

	Artefacts manipulables	générique	chaînes de composition	ordres de composition	vérification de la composition
CROME	non	non	non	oui	non
SOD	oui	non	non	oui	correspondance des structures exprimées en OCL
Catalysis Composition de modèles	oui	non	non	non	en langage naturel

TAB. 3.1 – Approche de structuration

	Artefacts manipulables	granularité des paramètres	chaînes de composition	ordres de composition	vérification de la composition
Catalysis Frameworks de modèles	non	nom des éléments	oui	non	non
France	oui	nom des éléments	non	oui	non
Theme	oui	classes	non étudié	non	oui

TAB. 3.2 – Approches génériques

l'ordonnancement des compositions est pris en compte par l'approche. Il est traité en définissant explicitement un ordre de composition des aspects sur le modèle primaire (la base) à l'aide d'un ensemble de directives. La vérification de la validité des compositions et du modèle obtenu sont à la charge du concepteur.

L'approche Theme/UML propose l'utilisation de paquetages templates UML pour exprimer les modèle d'aspects. La partie variable est ainsi décrite à l'aide des paramètres du template. Les éléments de modélisation génériques sont ainsi clairement définis, pouvant être facilement intégrés dans un outil de modélisation. L'approche propose sa propre relation *bind* pour exprimer la composition d'un Theme à un paquetage. De toutes les approches étudiées, c'est l'approche Theme qui propose la granularité la plus importante pour le paramétrage des éléments génériques. En effet, un paramètre n'est plus une simple chaîne de caractères, mais une classe avec un ensemble de méthodes. Cette granularité plus importante permet d'exprimer des éléments requis plus complexes et de vérifier l'adéquation entre ces éléments requis et ceux fournis lors de la composition. La notation Theme/UML n'est pas conçue pour permettre l'expression de chaînes de compositions alternatives. Enfin, l'approche Theme est principalement conçue pour exprimer la composition de comportements, à l'aide de diagrammes de séquences.

Cet état de l'art nous donne plusieurs pistes pour l'expression de composants de modèle. Les paquetages UML semblent être de bons candidats pour l'expression d'éléments de modélisation composables. En effet, ceux-ci présentent une granularité permettant l'expression de modèles complexes. De plus, le standard UML définit le mécanisme de *template* permettant le paramétrage d'éléments de modélisation comme les paquetages. Cependant bien que plusieurs approches identifient le problème, il n'existe pas de travaux permettant de garantir des propriétés sur l'ordre des compositions ou d'exprimer et d'identifier des équivalences sur des chaînes de composition.

Enfin, la granularité des paramètres utilisée dans les approches existantes est de l'ordre des noms ou même d'une classe. Bien que cela permettrait d'exprimer des éléments de modélisation plus complexes et plus riches, l'utilisation de modèles complets (ou partiels) en tant que paramètres n'est pas considéré.

En nous inspirant de ces approches, notre proposition vise à permettre l'expression de fonctionnalités génériques riches et de leurs compositions. Pour cela nous proposons d'exprimer, aussi bien la partie métier que la partie requise de ces fonctionnalités, à l'aide de modèles. La construction d'un système est alors réalisée par composition de ces différents modèles. Afin de garantir la cohérence des systèmes obtenus, nous souhaitons également pouvoir vérifier, par un ensemble de règles bien définies, l'adéquation entre les modèles requis et ceux fournis par les différentes fonctionnalités.

Il est également important que ces artefacts réutilisables puissent être exprimés dans un langage standard (ici UML) afin de permettre leur intégration dans les démarches et les outils existants. Enfin, pour être exploitable lors de la conception de systèmes réels nécessitant la composition d'un grand nombre de fonctionnalités, notre proposition doit garantir des propriétés de cohérence et d'équivalence entre les différentes évaluations possibles. En effet, dans une démarche de conception, les fonctionnalités peuvent être composées les unes à la suite des autres ou, au contraire, toutes composées sur le même

modèle de base. Dans le cadre de la conception d'un système réel, il est important d'identifier les différences, ou au contraire les équivalences, qui résulteront de l'utilisation d'une chaîne de composition plutôt qu'une autre.

Nous étudions dans le chapitre suivant l'état actuel des standards de modélisation et de méta-modélisation et dans quelle mesure ils permettent l'expression de compositions de modèles et l'utilisation de modèles génériques.

Chapitre 4

Etat des standards

En informatique, comme dans tous les domaines techniques, l'utilisation et le respect des standards sont primordiaux pour permettre l'interopérabilité des systèmes et l'échange de données. Les modèles n'échappent pas à cette règle. C'est pourquoi l'OMG a proposé un langage de modélisation standard, le langage UML. L'OMG a également défini un langage de méta-modélisation standard, le MOF (Meta Object Facility). Afin de permettre l'expression de contraintes ne pouvant être modélisées par le langage UML, l'OMG propose l'utilisation du langage OCL [WK03] (Object Constraint language). L'OMG a entamé une refonte complète de ces langages notamment afin de les harmoniser entre eux. Elle vise également à faciliter la manipulation des modèles et méta-modèles comme des artefacts concrets, notamment par des langages comme QVT [QVT05]. Cette refonte doit donner la version 2 de ces standards.

Afin de leur donner une définition stricte et proche des standards, nous définissons, dans la partie IV, les éléments de notre approche à l'aide de ces langages. C'est pourquoi nous présentons les standards MOF et UML, ainsi que leurs relations à la section 4.1.

Nous étudierons, à la section 4.2, parmi les relations existantes dans le MOF et UML, si celles-ci peuvent être utilisées dans le cadre de notre problématique de conception de système par composition de modèles génériques.

Enfin, la section 4.3 décrit la notion standard d'éléments de modèle paramétrables, appelés *Template*.

4.1 Articulations entre les méta-modèles MOF et UML

Bien que conçus à l'origine indépendamment l'un de l'autre, les méta-modèles UML et MOF définissent un ensemble de concepts communs. Avec les versions 2 de ces standards, les membres de l'OMG ont voulu en tenir compte en harmonisant leur méta-modèle.

Un cœur commun a donc été défini dans le document *UML2Infrastructure*[UML05a]. Ce cœur (*Core*) est inclus dans un paquetage nommé *InfrastructureLibrary* (cf. figure 4.1). Le but de ce cœur commun est de servir de pivot entre les différentes technologies de l'OMG.

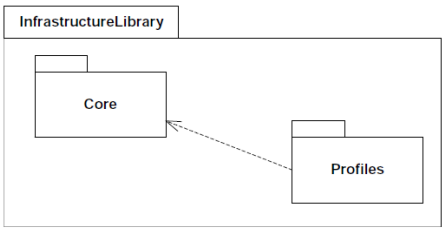

FIG. 4.1 – L'*InfrastructureLibrary* de l'OMG [UML05a]

Pour cela, le paquetage *Core* définit un méta-modèle complet que les autres méta-modèles peuvent importer ou spécialiser pour leurs besoins spécifiques. Cette vision est illustrée par la figure 4.2. L'OMG présente également ce paquetage *Core* comme le cœur architectural du MDA.

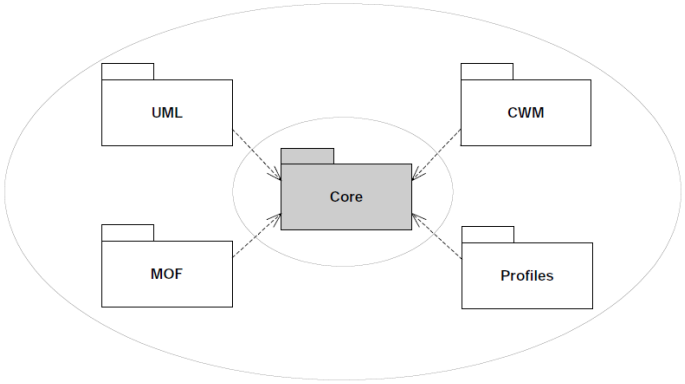

FIG. 4.2 – Le paquetage commun *Core* [UML05a]

Le paquetage *Core* est lui même structuré à l'aide de quatre paquetages (cf. figure 4.3), *PrimitiveTypes*, *Abstractions*, *Basic* et *Constructs*.

Le paquetage *PrimitiveTypes* définit les types de bases utilisés dans les modèles et méta-modèles : entier, booléen et chaîne de caractères.

Le paquetage *Abstractions* est lui même structuré en un grand nombre de paquetages. Chacun d'eux définit un concept abstrait le plus indépendamment possible des autres concepts. On trouve ici, la définition des concepts de *Classifier*, de relation (*Relationship*), de *Generalization* ou encore d'espace de nommage (*Namespace*). La raison invoquée par l'OMG pour cette structuration de fine granularité, est la possibilité de ne sélectionner que les concepts voulus pour la définition d'un nouveau méta-modèle.

Le paquetage *Basic* permet de définir les éléments de base des méta-modèles. On y trouve notamment la définition des concepts de classe, d'attribut (*Property*) et d'opération, ainsi que leurs relations. C'est ici qu'il est défini qu'une classe peut

contenir des attributs, des opérations, et qu'elle peut avoir des sur-classes.

Enfin, le paquetage *Constructs*, permet de mettre en relation l'ensemble des concepts définis. Comme, par exemple, la mise en relation du concept abstrait de *BehavioralFeature* avec celui d'opération. Cela est réalisé par leur redéfinition (à l'aide du mécanisme d'héritage). Des concepts plus complexes y sont également définis, notamment celui d'association ou d'*import*.

Ces quatre paquetages sont liés entre eux par ces relations d'*import*. Le paquetage *PrimitivesTypes* est importé par les paquetages *Abstractions* et *Basic* afin de pouvoir utiliser les types de base pour la définition de leurs concepts. Ce dernier importe également le paquetage *Abstraction* pour utiliser les concepts d'élément et de multiplicité. Enfin le paquetage *Constructs* importe les paquetages *Abstractions* et *Basic* afin de mettre en relation les concepts qui y sont définis. La relation d'*import* étant transitive, les types de base y sont également accessibles[1].

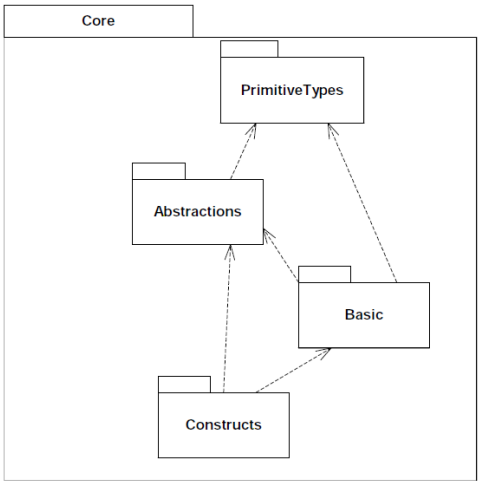

FIG. 4.3 – Le contenu du paquetage *Core* [UML05a]

Le langage UML2 est défini (dans le document UML2Superstructure[UML05b]) en trois parties : la partie structure, la partie comportement (*Behavior*) et la partie suppléments.

La partie structure permet de définir les concepts statiques qui peuvent être utilisés notamment dans les diagrammes de classes. La partie "comportement" définit les concepts nécessaires à la modélisation des actions, des activités, des interactions, des machines à états ou encore des cas d'utilisation. Enfin la partie "suppléments", définit des constructions auxiliaires comme les *Templates*, et la notion de profil qui permet de spécialiser un modèle et que nous utilisons au chapitre 9. La notion de *Model* y est également définie par extension du concept de paquetage.

Les parties comportement et suppléments sont basées sur la partie structure qui

[1] À noter que cela devrait également être le cas pour le paquetage *Basic*

est elle même définie à partir d'un paquetage noyau (*Kernel*). Ce dernier est défini en regroupant (par la relation *merge*) les constructions issues du paquetage *Core* de l'infrastructure (*Core : :Constructs*) et les concepts de généralisation, de littéral, d'expression de multiplicité et d'instance (cf. figure 4.4). La sémantique de cette relation *merge* est donnée section 4.2.

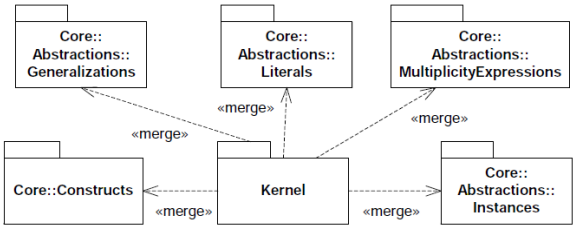

FIG. 4.4 – Le cœur du langage UML 2 [UML05b]

Le langage MOF[MOF03] est également défini à partir de concepts issus du paquetage *Core* de l'infrastructure (cf. figure 4.5). Le paquetage EMOF (Essential MOF) qui définit les concepts principaux du MOF est construit en fusionnant le paquetage *Basic* du *Core* et les concepts d'extension, de réflexion et d'identité définis pour le MOF. Cette fusion est exprimée par la relation *combine* dont nous donnons également la sémantique à la section 4.2.

Le paquetage CMOF (Complete MOF) est quant à lui défini en regroupant les éléments définis dans EMOF avec les constructions définies dans le *Core*. En plus des concepts élémentaires de EMOF (classe, attribut, opération,. . .), le CMOF contient les concepts d'association et d'instance.

Conceptuellement, le méta-modèle UML est instance du MOF. Il doit donc être écrit à l'aide du langage MOF. La figure 4.6 résume les liens existants entre l'infrastructure, UML2 (la superstructure) et le MOF2. Les méta-modèles UML2 et MOF2 dépendent, pour leur définition, d'éléments de l'infrastructure.

4.2 Relations entre les modèles

Les langages UML et MOF définissent un ensemble de relations entre modèles et éléments de modèle. Comme nous l'avons vu, certaines d'entre elles sont également utilisées pour la définition de ces langages eux-mêmes (*import*, *merge*, *combine*). Nous présentons dans cette section quelques unes de ces relations et étudions la possibilité de les utiliser pour exprimer la composition de modèles.

Nous proposons à la figure 4.7 la hiérarchie des relations définies par les standards UML2 et MOF2 à partir du concept abstrait de relation (`Relationship`). Certaines de ces relations étant spécifiques à un type de modèle particulier (comme *ProtocoleConformance* ou *InformationFlow*) nous ne les présenterons pas ici. Le concept d'association au contraire est très général. Il permet d'établir un lien entre deux classes dont

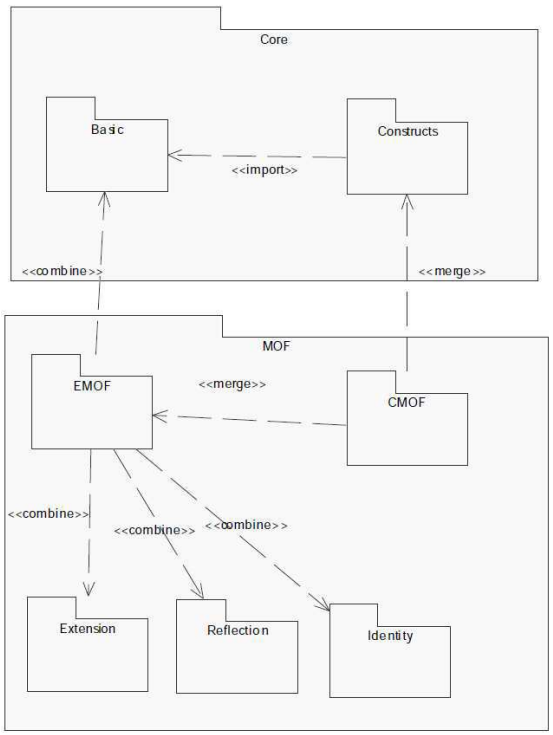

FIG. 4.5 – Le méta-modèle MOF [MOF03]

la sémantique doit être définie au niveau instance et non au niveau des éléments de modèle eux-mêmes. Il ne sera donc pas non plus étudié dans cette section.

À l'exception de ce concept d'association et de ceux qui en dérivent, l'ensemble des relations définies dans les méta-modèles standards, dérivent du concept abstrait de relation orientée (`DirectedRelationship`). Une telle relation désigne un ou plusieurs éléments sources et un ou plusieurs éléments cibles et n'a pas de sémantique à ce niveau.

La relation de `Generalization` correspond au concept d'héritage des langages objets. Elle permet d'indiquer qu'une classe est une spécialisation d'une (ou plusieurs) autre(s) classe(s). Elle peut être utilisée comme moyen de structuration, mais au niveau d'une classe.

La relation `ElementImport` permet de désigner qu'un élément (cible) défini dans un espace de nommage est importé dans un autre espace de nommage (source). Le concept abstrait d'espace de nommage est, par exemple, concrétisé par celui de paquetage. La sémantique de cette relation est que l'élément cible peut être désigné dans l'espace de nommage source, comme si il y était défini. Cependant, l'élément ainsi importé ne peut être modifié ou enrichi dans cet espace de nommage source. En particulier, elle ne permet pas de composer un ensemble d'éléments importés.

La relation `PackageImport` a pour cible et pour source un espace de nommage. Sa

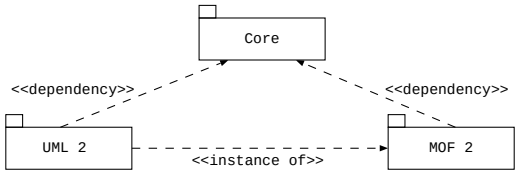

FIG. 4.6 – Liens entre infrastructure, UML2 et MOF2

sémantique est équivalent à avoir, pour chaque élément de l'espace de nommage cible, une relation *ElementImport* avec l'espace de nommage source. On peut distinguer deux types d'import, l'import publique, stéréotypé `<<import>>`, et l'import privé, stéréotypé `<<use>>`. Ces deux types ont la même sémantique, la différence est que l'import public est transitif, alors que l'import privé ne l'est pas. Cette relation ne permet donc pas non plus la composition des éléments issus de différents paquetages.

Le standard UML2 définit également un ensemble de relations de dépendance. Le concept de `Dependency` définit une relation entre un ou plusieurs *NamedElement* clients, et un ou plusieurs *NamedElement* fournisseurs. La présence d'une telle relation indique que la définition du (des) client(s) n'est pas sémantiquement ou structurellement complète sans son (ses) fournisseur(s). Elle n'a pas d'implication sémantique à l'exécution, elle indique une dépendance entre les éléments de modèles et non entre leurs instances.

La relation `Usage` est une relation de dépendance qui se répercute au niveau de l'implantation des éléments.

La relation `Abstraction` spécialise la dépendance. Elle permet d'indiquer que deux éléments ou deux ensembles d'éléments représentent le même concept à différents niveaux d'abstraction ou par différents points de vues. La norme UML2 inclut trois types de relations d'abstraction "prédéfinies" : `<<derive>>`, `<<refine>>` et `<<trace>>`. Le type `<<derive>>` indique que le ou les éléments clients peuvent être calculés à partir des éléments fournisseurs. Le type `<<refine>>` permet de lier des éléments entre différents niveaux sémantiques (analyse et design par exemple). Enfin, le type `<<trace>>` permet de lier des éléments issus de différents modèles, représentant le même concept. Il peut par exemple être utilisé pour tracer les évolutions d'un concept entre les modèles. Comme nous le verrons au chapitre 8, en liant des éléments issus de modèles différents, cette relation permet également de spécifier qu'un même concept résulte de la composition de ces éléments.

La relation `Realization` permet d'établir un lien entre des spécifications et leurs implantations. Cette dernière est spécialisée par la relation `Substitution` qui indique une substitution dynamique possible entre les éléments client et fournisseur. Cette relation n'implique pas une relation d'héritage entre le client et le fournisseur, mais indique qu'ils respectent le même contrat.

Comme son nom l'indique la relation `TemplateBinding` (notée `<<bind>>`) s'utilise conjointement avec le concept de *Template*. Cette relation est paramétrée par un ensemble de substitutions. Elle permet de décrire comment l'élément source a été construit

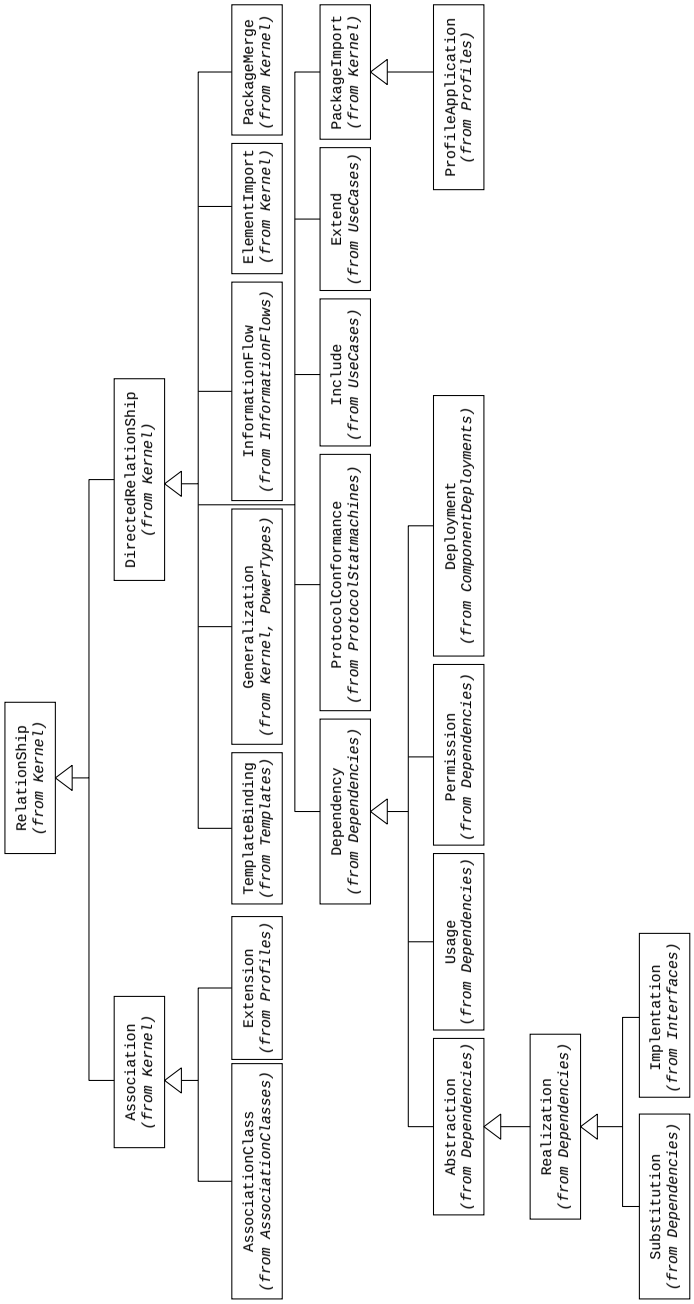

FIG. 4.7 – Hiérarchie des relations de UML2 et du MOF2

à partir d'un élément *template*. Nous reviendrons sur cette notion de *template* et cette relation de *TemplateBinding* dans la section suivante et au chapitre 10.2.

L'ensemble de ces relations permettent de donner de l'information sur les éléments de modélisation mais n'ont pas pour but de définir de nouveaux éléments. Pour cela, le

standard introduit une relation de fusion de paquetages, `PackageMerge`. L'infrastructure UML2 donne une sémantique de cette relation appelée *extend*, le méta-modèle MOF en définit une seconde nommée *define*.

Lorsque plusieurs relations de *PackageMerge* ont pour source un même paquetage, cela signifie que les éléments de même nom et de même type issus des paquetages cibles doivent être fusionnés. La façon de procéder à cette fusion diffère entre la version *extend*, qui utilise le mécanisme de généralisation et la version *define*, qui équivaut à une fusion réelle des éléments.

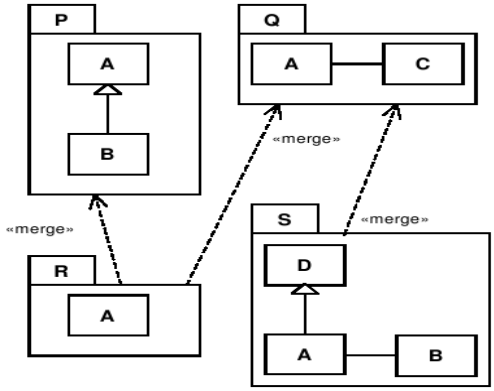

FIG. 4.8 – Exemple de paquetage merge [UML05b]

La relation *extend*, stéréotypée `<<merge>>`, procède à la fusion des éléments de même nom et de même type par héritage (ou par redéfinition). Cette relation est équivalente à importer tous les éléments des paquetages cibles dans le paquetage source et à créer, pour chacun d'eux, un nouvel élément qui en hérite. Dans le cas où plusieurs éléments importés sont de même type et de même nom, un seul élément, héritant de chacun d'eux, est créé.

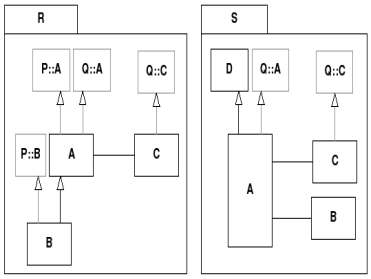

FIG. 4.9 – Résultat du paquetage *merge* [UML05b]

La figure 4.8 illustre l'utilisation de cette relation pour la définition des paquetages R et S. Les résultats de ces définitions sont illustrés par la figure 4.9. Les éléments A des

paquetages P et Q sont importés dans le paquetage R. L'élément A de ce paquetage est défini par héritage de ces deux éléments importés.

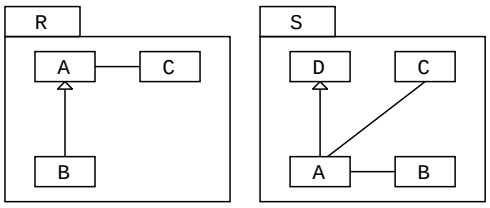

FIG. 4.10 – Résultat du paquetage *define*

La version *define* de cette relation de *PackageMerge*, stéréotypée `<<combine>>`, est équivalente à copier tous les éléments des paquetages cibles dans le paquetage source et à fusionner (réellement) les éléments de même type et de même nom (cf. figure 4.10). Aucun lien n'est créé entre les éléments des paquetages cibles et les éléments du paquetage source. Cela signifie qu'il y a indépendance entre le paquetage résultant et les paquetages à partir desquels il a été créé.

Comme nous l'avons vu, les définitions des méta-modèles UML2 et MOF2 sont réalisées à l'aide de ces relations `<<merge>>` et `<<combine>>` (cf. figures 4.4 et 4.5). Elles permettent donc la composition d'éléments définis dans différents paquetages. Cependant, ces éléments doivent porter le même nom, ce qui ne permet pas la composition de modèles conçus indépendamment.

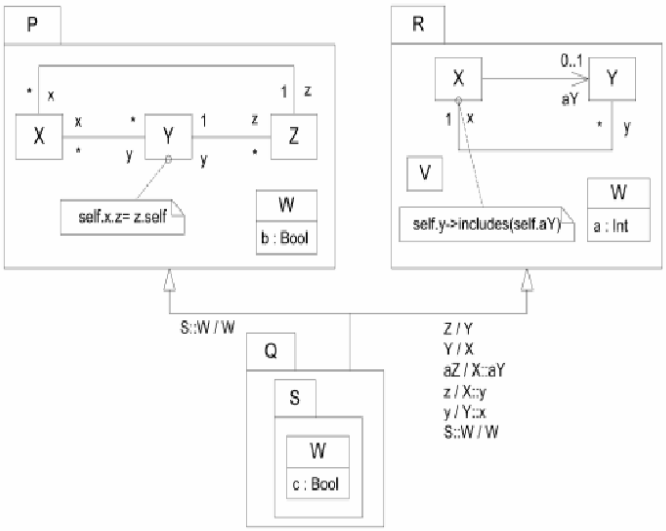

FIG. 4.11 – *Package extension* [CEK02]

Pour cela, une relation d'extension de paquetage a été proposée dans [CEK02]. Son

utilisation est comparable à celle de la relation de *PackageMerge*, mais il est possible de renommer les éléments en vue de leur fusion. Ce mécanisme est illustré à la figure 4.11. L'élément Y du paquetage R est, par exemple, renommé en Z. La figure 4.12 illustre le paquetage résultat de cette extension. L'élément Y du paquetage R est donc ici fusionné avec l'élément Z du paquetage P.

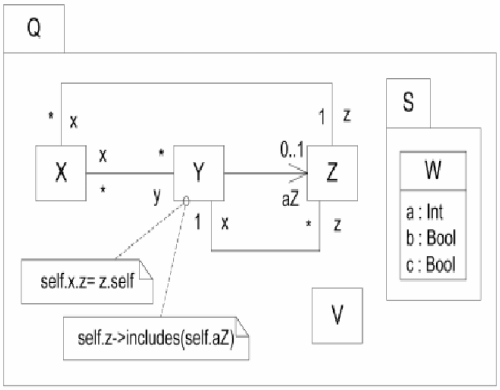

FIG. 4.12 – Résultat du *Package extension* [CEK02]

Avec ces relations (*PackageMerge* ou *Package extension*), tous les éléments des paquetages utilisés peuvent potentiellement être sélectionnés lors de la fusion. La notion de *template* permet quant à elle de préciser dès la conception du paquetage les éléments fixes des éléments paramétrables.

4.3 Templates UML2

Le standard UML définit la notion de template comme un élément de modèle paramétré par d'autres éléments de modèle. De tels éléments de modèle paramétrables peuvent être des classes ou des paquetages, et sont appelés respectivement *class templates* ou *package templates*. De nombreuses approches proposent l'utilisation de templates comparables [CW05, DW99, SR01, SGS+04]. Nous les utilisons également, au chapitre 10, pour donner à notre approche une représentation standard.

Pour spécifier son paramétrage, un élément de template possède une signature. Une signature de template correspond à une liste de paramètres formels où chaque paramètre désigne un élément faisant partie du template. La signature des éléments template est spécifiée dans un rectangle en pointillé dans leur coin supérieur droit.

Un template peut être utilisé pour spécifier d'autres éléments de modèle. Une relation *bind* lie un *bound* élément à la signature d'un template cible et spécifie un ensemble de substitutions de template paramétré, associant les éléments effectifs aux paramètres formels. L'attache d'un *bound* élément implique que son contenu est basé sur le contenu du template cible. Les éléments exposés comme paramètres formels sont alors substitués par les éléments actuels spécifiés par le paramétrage de la relation *bind*.

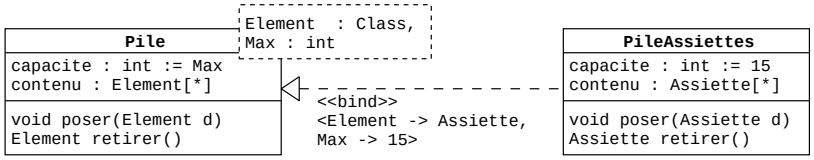

FIG. 4.13 – Classe template

La partie gauche de la figure 4.13 montre une classe template. Cette classe, *Pile*, est représentée graphiquement comme une classe standard UML munie d'un rectangle en pointillé contenant sa signature. Ici, la signature désigne deux éléments comme paramètres formels : *Element* de type classe et *Max* de type entier. La partie droite de cette figure montre la classe *PileAssiettes* reliée au template *Pile* au travers d'une relation *bind*. Cette classe est le résultat de la substitution des paramètres formels du template, *Element* et *Max* par les valeurs *Assiette* et *15* respectivement. Cette substitution est établie par la relation *bind*.

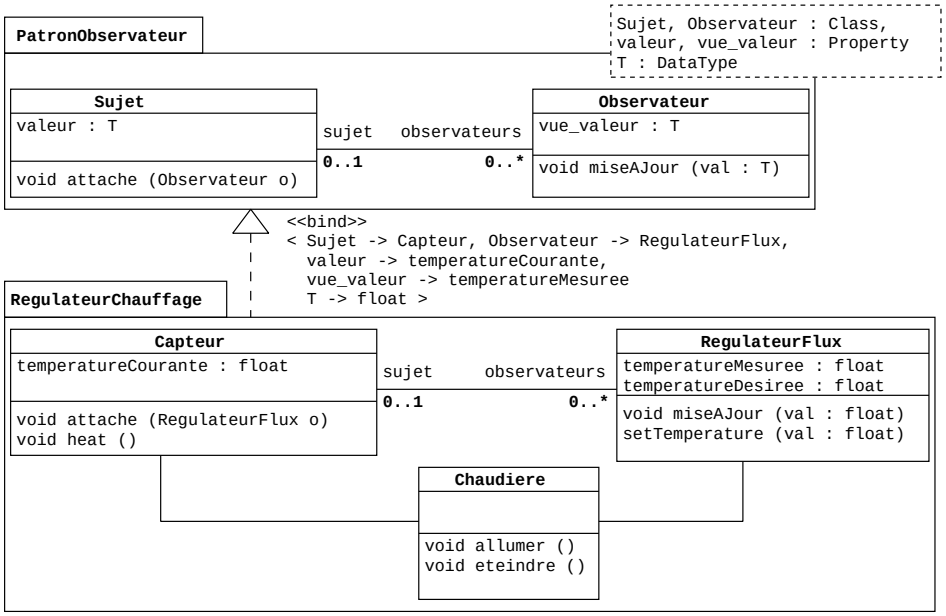

FIG. 4.14 – Paquetage template

La figure 4.14 présente un second exemple de template, qui illustre l'utilisation de ce mécanisme au niveau d'un paquetage. Ici, la notion de template est employée pour modéliser le patron observateur. Le contenu de ce template reflète la structure de ce patron et inclut les classes habituelles *Sujet* et *Observateur*. Comme spécifié par la signature attachée au template, ces classes et leurs attributs *valeur* et *vue_valeur* respectifs sont identifiés comme paramètres formels.

La figure 4.14 illustre également, l'utilisation de ce template en spécifiant un paquetage *RegulateurChauffage* spécifiant une partie d'un système de chauffage. Ce paquetage possède son propre contenu composé des classes *Capteur*, *RegulateurFlux* et *Chaudiere*. Dans cet exemple, la relation *bind* est utilisée pour définir la collaboration entre *Capteur* et *RegulateurFlux*. Ceci est spécifié en assignant *Capteur* à *Sujet*, *RegulateurFlux* à l'*Observateur* et leurs attributs *valeur* et *vue_valeur* à *temperatureCourante* et *temperatureMesuree* respectivement. Les classes *Capteur* et *RegulateurFlux* ont respectivement les mêmes éléments (opérations et associations) que *Sujet* et *Observateur*. À noter que les classes effectives peuvent avoir des contenus supplémentaires à ceux spécifiés par les paramètres formels.

La norme UML2 [UML03b] définit cette notion de template à l'aide de quatre méta-classes *TemplateSignature*, *TemplateableElement*, *TemplateParameter* et *ParameterableElement* (cf. figure 4.15). Le paramétrage est quant à lui défini à l'aide des concepts de *TemplateBinding* et de *TemplateParameterSubstitution* (cf. figure 4.16).

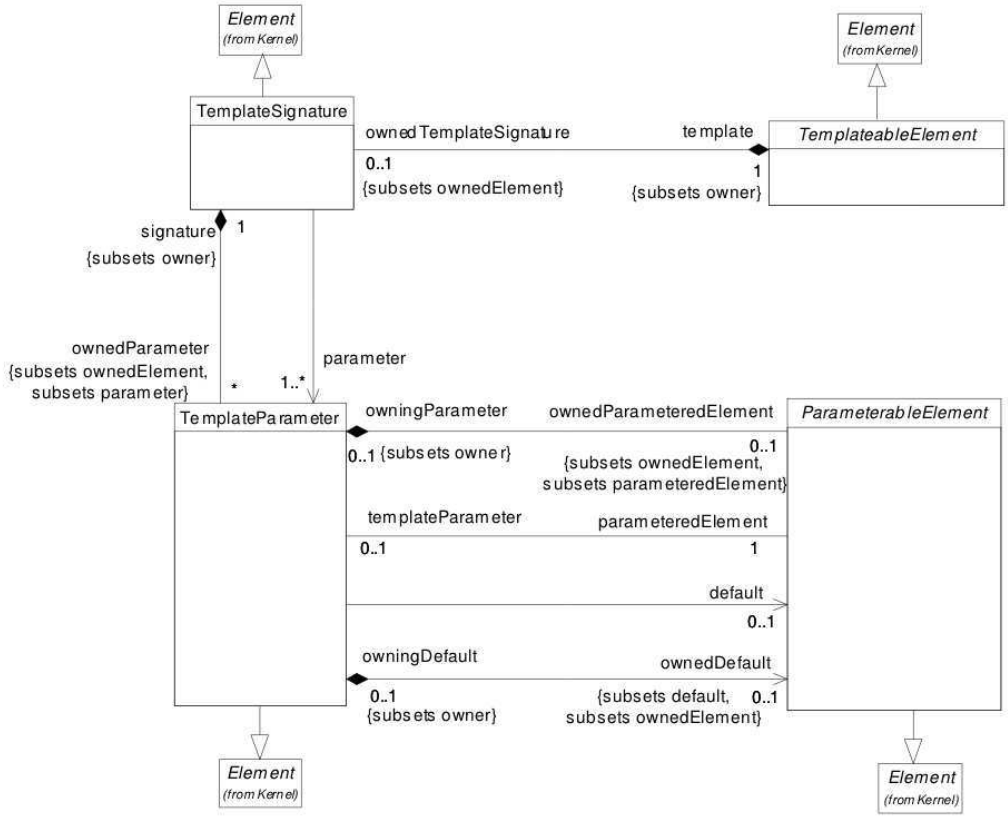

FIG. 4.15 – Le méta-modèle des templates

La figure 4.17 présente un extrait du diagramme d'instances correspondant à la figure 4.14 permettant d'illustrer chacun des concepts du méta-modèle de template.

La méta-classe abstraite *TemplateableElement* permet de définir le concept d'élément pouvant être paramétré, ces éléments sont *Classifier* (les classes et associations entrent dans cette catégorie), *Package* et *Operation*. Ce concept est illustré dans nos exemples par la classe *Pile* ou le paquetage *Observation*.

L'ensemble des paramètres (les *TemplateParameter*) d'un *TemplateableElement* sont regroupés dans une *TemplateSignature*. Ce concept correspond aux rectangles en pointillés. Les *TemplateParameters* (illustrés par *Subjet* figure 4.17) représentent les paramètres formels d'un *TemplateableElement* (les éléments contenus dans la signature), chacun d'eux correspond (par le rôle *parameteredElement*) à un élément du *TemplateableElement* de même nom (*Subjet* dans notre exemple). Enfin, les éléments de type *ParameterableElement* sont les éléments pouvant être "remplacés" par un paramètre (*Subjet*) ou pouvant jouer le rôle de paramètre effectif (*Capteur*). Ces éléments peuvent être des *Classifiers*, des *PackageableElements*, des *Operations* ou des *Properties*.

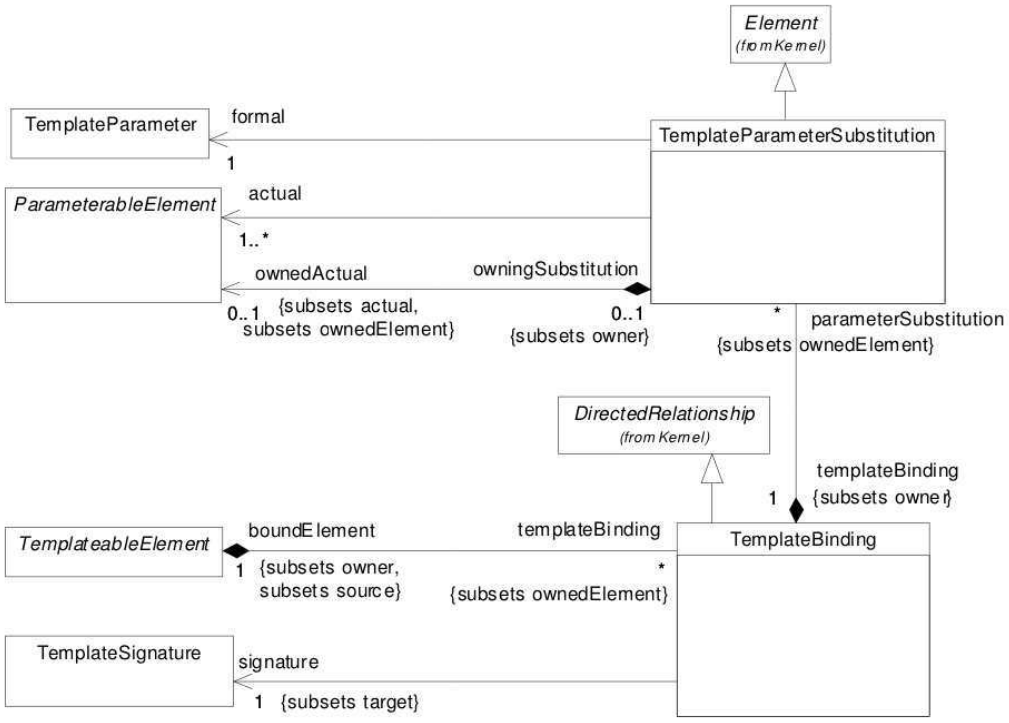

FIG. 4.16 – Le méta-modèle du passage de paramètres

Le concept de *TemplateBinding* permet d'exprimer l'utilisation du template pour un système particulier, représenté par une relation stéréotypée `<<bind>>`. Il comprend un ensemble de *TemplateParameterSubstitution* correspondant au remplacement de chaque paramètre formel par un paramètre effectif, chacun de ces paramètres effectifs appar-

tenant au *boundElement* de la relation de binding.

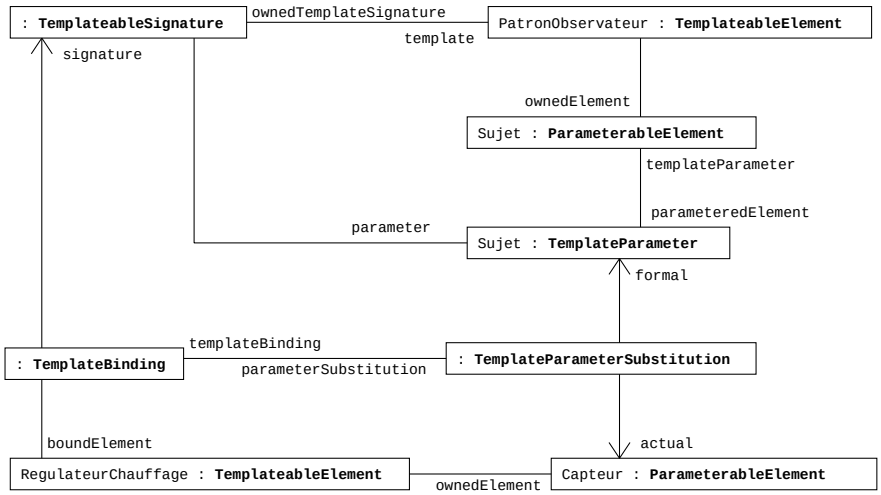

FIG. 4.17 – Diagramme d'instance du paquetage Observation

La norme UML2 définit également la notion de binding partiel pour lequel certains paramètres ne sont pas fournis. Dans ce cas, le *boundElement* est également un élément template contenant ces paramètres dans sa signature.

Les contraintes définies sur ce méta-modèle dans la norme UML permettent de garantir que le type de chaque paramètre effectif est bien compatible avec le type du paramètre formel correspondant. C'est-à-dire pour notre exemple figure 4.13, que *Assiette* est bien une classe et *15* est bien un *int*. D'autres contraintes sont nécessaires pour vérifier la validité du *binding*. La formalisation de ces contraintes est présentée section 10.2.2 [CCMV04].

4.4 Résumé

Les standards de modélisation et de méta-modélisation sont en constante évolution. Mais leur utilisation est cependant indispensable pour permettre l'inter-opérabilité des approches et l'utilisation d'outils de modélisation. C'est pourquoi nous définissons, dans la partie IV de ce document, nos éléments de modélisation relativement à ces standards.

De plus, l'utilisation des techniques de méta-modélisation, couplée à l'utilisation d'un langage de contrainte comme OCL, permet de donner une définition stricte des éléments de modélisation et de leur utilisation.

Nous avons également montré que les moyens d'expression standard pour la composition de modèles était limités : *import* d'éléments d'un modèle dans un autre, ou fusion des éléments issus de paquetages différents.

Enfin, nous avons étudié la notion de template proposée par le standard UML2, qui permet l'expression de modèles génériques. Cependant, leur partie variable est exprimée par un ensemble d'éléments paramètres indépendants, ce qui n'est pas adapté à l'expression de fonctionnalités génériques complexes nécessitant elles-mêmes un modèle complexe pour être utilisées. Il est donc nécessaire d'étendre cette notion de template afin de permettre l'expression de véritables composants de modèle. C'est ce que nous proposons d'étudier au chapitre 10.

Troisième partie

Proposition

La partie précédente nous a permis de faire un état de l'art et des standards autour de la structuration et de la composition de modèles. Cette partie présente le cœur de notre proposition ainsi que sa formalisation. Sa méta-modélisation et sa mise en œuvre seront présentées dans les parties suivantes.

Nous présentons dans le chapitre suivant un premier travail permettant de faire le lien entre les notions d'aspects fonctionnels et de composants métiers. En effet, les aspects fonctionnels forment de bons candidats pour la conception de composants métiers génériques, moyennant leur déconnexion de tout contexte particulier. Nous proposons dans ce chapitre un modèle de vues génériques que nous appelons *composants vue*. Chaque composant vue permet de décrire une fonctionnalité générique indépendamment d'un système particulier. Un mécanisme de connexion de ces vues génériques avec un modèle de base est alors nécessaire pour ajouter cette fonctionnalité à ce modèle. Le bénéfice attendu est double, préserver la double structuration entités/fonctionnalités introduite par le mécanisme de vues et permettre la réutilisation de ces fonctionnalités pour la conception de différents systèmes.

Le chapitre 6 généralise cette idée en proposant l'expression de modèles métiers génériques sous forme de modèles eux-mêmes paramétrés par un modèle requis. Nous appelons ces modèles métiers génériques des *composants de modèle*. Le but est de permettre la construction de systèmes par assemblage d'un ensemble de composants de modèle décrivant chacun une fonctionnalité du système. Nous proposons d'exprimer ces assemblages par application de composants de modèle, mettant en relation leur modèle requi avec le modèle du système. L'utilisation de ce mécanisme est également présenté dans le chapitre 6.

Sans règles appropriées, l'assemblage d'un ensemble de composants de modèle ayant été conçus indépendamment les uns des autres peut produire un modèle invalide. C'est pourquoi, afin de garantir des propriétés de cohérence sur les systèmes ainsi obtenus, nous proposons dans le chapitre 7, un opérateur d'application de modèles génériques. Nous proposons une formalisation des modèles et modèles génériques sous forme d'ensembles d'éléments de modèle. Après avoir défini l'opération d'application de modèles génériques à partir de cette formalisation, nous présentons et démontrons un ensemble de propriétés d'ordre sur cet opérateur.

Enfin, le chapitre 8 présente différents modes d'expressions des modèles résultants de la composition de modèles génériques. Il est en effet possible d'exprimer ces modèles résultants sous différentes formes, par un modèle objet par exemple, mais également à l'aide de modèles à vues, génériques ou non. Chacune de ses expressions ayant des propriétés différentes.

Chapitre 5

Réutilisation d'aspects fonctionnels

Comme nous l'avons déjà évoqué, un système doit répondre à un grand nombre de préoccupations. Pour des systèmes mettant en jeu des ressources, on retrouve par exemple, des préoccupations de gestion des stocks, de gestion des allocations de ressources, de recherches d'éléments (exemples inspirés de [Cla02] et [Wil96]). Nous parlons ici de préoccupations métiers, par opposition aux préoccupations techniques auxquelles les systèmes doivent également répondre (sécurité, persistance, répartition,...).

Chacune de ces préoccupations est prise en compte par un aspect du système. Celui-ci regroupe tous les éléments nécessaires pour répondre à la préoccupation à laquelle il correspond. À un niveau modèle, ces éléments peuvent être des classes, des attributs, des opérations ou des associations. Les aspects répondant aux préoccupations métiers sont appelés des aspects fonctionnels[1].

Les mêmes préoccupations se retrouvant dans de nombreux systèmes, il parait donc intéressant de permettre l'expression d'aspects génériques pouvant être utilisés pour différents systèmes. Pour cela, il est nécessaire de permettre l'expression d'aspects fonctionnels génériques utilisables pour la réalisation de modèles de systèmes. C'est ce que nous proposons de faire dans la première section sous la forme de *composants vue*. La section suivante montrera leur (ré-)utilisation par leur capacité de connexion à d'autres bases, ainsi qu'à différentes parties d'une même base.

5.1 Aspects fonctionnels

Nous avons vu dans la partie précédente, notamment avec l'approche CROME (section 3.1.1), que les aspects fonctionnels offrent une dimension de structuration complémentaire à la dimension de structuration par entités classique de l'approche objet. Il devient ainsi possible d'identifier pour chaque aspect fonctionnel les entités du système qui y participent.

[1] On parle sinon d'aspects non-fonctionnels ou techniques.

Ces aspects fonctionnels sont cependant spécifiques au système pour lequel ils ont été conçus. Le travail nécessaire pour les adapter à un autre système devant répondre aux mêmes préoccupations est relativement important. Pour réaliser ce travail d'adaptation, il est en effet nécessaire de comprendre l'ensemble du système afin de pouvoir identifier les éléments propres à l'aspect fonctionnel. Il est ensuite nécessaire de comprendre comment ces éléments doivent se composer avec les éléments du nouveau système. De plus, cette tâche doit être répétée pour chaque système auquel on souhaite appliquer l'aspect fonctionnel.

Notre proposition [MCCV03] vise à permettre l'expression d'aspects fonctionnels génériques afin qu'ils soient facilement utilisables pour la conception de nouveaux systèmes. En effet, il est possible d'identifier deux parties dans les aspects fonctionnels, les éléments propres à la préoccupation et les éléments spécifiques à un système. La spécification des seconds est nécessaire, il n'est en effet pas possible de définir les premiers sans décrire comment ils se composent avec les éléments du système.

L'idée de notre approche est d'identifier et d'exprimer de façon générique ces éléments ne faisant pas réellement partie de la fonctionnalité modélisée par l'aspect, mais nécessaires pour décrire sa composition avec un système. Nous nous appuyons pour cela sur les approches par vues présentées dans la partie précédente.

Pour illustrer notre approche considérons un système de gestion de bibliothèque universitaire présenté figure 5.1. Les approches par vues permettent la structuration de ce système selon différentes préoccupations, gestion des ressources (en haut), recherche de document (à gauche) et gestion des locations (en bas). La partie au centre présente quant à elle le modèle de base du système. Celui-ci permet d'exprimer les concepts propres au domaine du système modélisé.

Chaque vue définit les éléments propres à une préoccupation donnée. La vue de gestion de ressources définit, par exemple, l'attribut *capacité* pour l'entité *Rayon*, ainsi que les opérations d'ajout, de suppression et de transfert de documents. Bien que les mêmes préoccupations puissent se retrouver dans de nombreux systèmes, les vues étant spécifiques au système pour lequel elles ont été conçues, elles ne peuvent être utilisées pour d'autre systèmes.

Afin d'en faire des aspects fonctionnels génériques réutilisables, nous proposons d'exprimer les éléments requis indépendamment d'un système particulier. C'est ce que la figure 5.2 illustre.

À chaque vue correspond un aspect fonctionnel générique, que nous appelons composant vue. Ils prennent la forme de paquetages stéréotypés `<<Component>>` et sont ici illustrés par *Recherche*, *GestionDesRessources* et *Location*. Les éléments requis par un composant vue sont exprimés à l'aide de ce que nous appelons des éléments vues. Ceux-ci peuvent être des classes (identifiées par `<<ViewClass>>`), des attributs (identifiés par `<<ViewAttribute>>`) ou des associations (identifiées par `<<ViewAssociation>>`). Ces éléments décrivent la partie générique, leurs noms ne doivent donc pas être spécifiques à un système, mais doivent correspondre au concept que l'élément représente.

L'ensemble de ces éléments vue forment le modèle requis pour appliquer le composant. Pour appliquer, par exemple, le composant de recherche, il est nécessaire de disposer dans le modèle de base du système, d'une entité qui servira de *Localisation*,

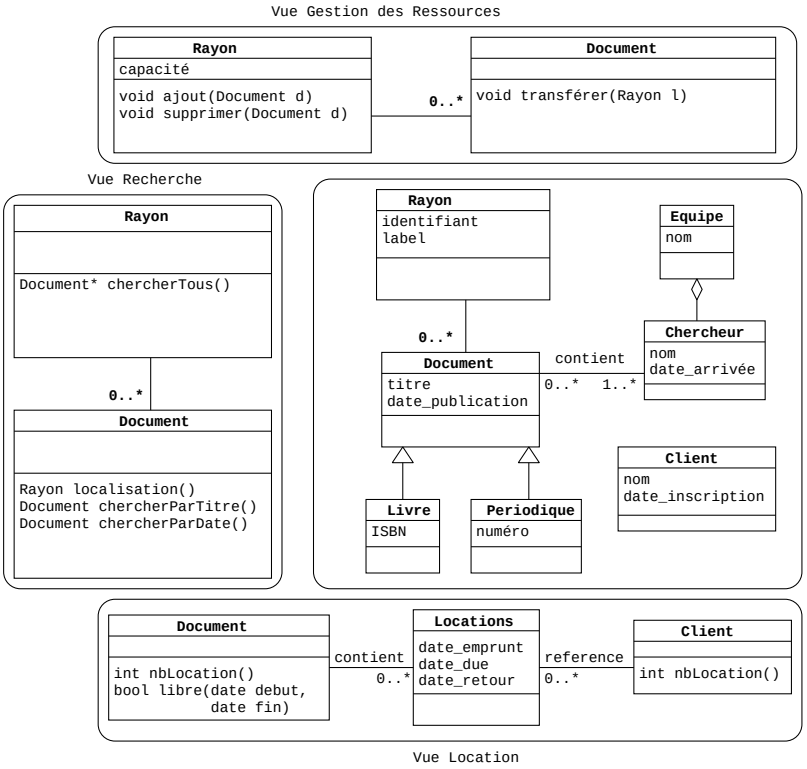

FIG. 5.1 – Approche par vues

d'une autre qui servira de *Ressource* et que ces deux entités devront être reliées par une association. Les `<<ViewAttribute>>` indiquent également que l'entité qui servira de *Localisation* devra disposer de deux attributs, l'un servant de nom, l'autre d'adresse. De même, l'entité qui servira de *Ressource*, devra disposer d'un attribut servant d'identifiant et d'un autre pour date.

L'aspect fonctionnel lui-même est défini par l'ensemble des éléments ne faisant pas partie du modèle requis (de l'ensemble des éléments vues) et qui sont apportés par la préoccupation. Ces éléments peuvent être des classes, des attributs, des opérations ou des associations. Cela correspond, par exemple, pour le composant de gestion des locations, aux opérations *nbLocation* et *libre*, à la classe *Locations* avec ses attributs et aux associations *contient* et *reference*.

L'expression d'aspects fonctionnels génériques est ainsi rendu possible grâce à une identification précise de la partie requise par rapport à la partie spécifique de la fonctionnalité modélisée.

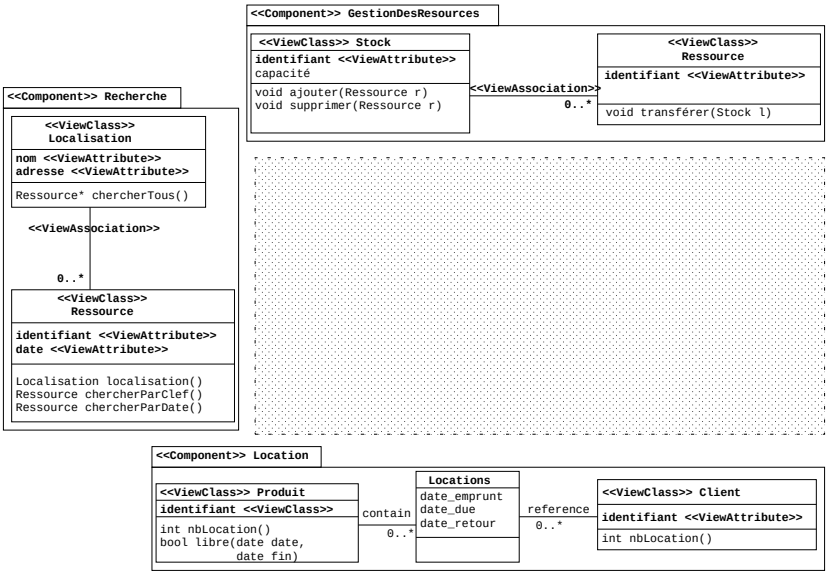

FIG. 5.2 – Composants vue.

5.2 Réutilisation

Cette section présente l'utilisation de ces composants vue pour enrichir un système de nouvelles fonctionnalités. Nous proposons pour cela un mécanisme de connexion entre un composant vue et un modèle de base. Ce mécanisme ne nécessite aucune modification, ni du composant vue, ni du modèle de base.

Pour ajouter une fonctionnalité exprimée par un composant vue, le modèle de base doit fournir le schéma requis par ce composant vue. Comme nous l'avons présenté dans la section précédente, ce schéma requis est exprimé par un ensemble d'éléments vue (`<<View...>>`). Cependant, un modèle de base peut contenir plusieurs schémas conformes à celui attendu par le composant. Il est donc nécessaire d'exprimer, lors de la connexion d'un composant vue, à quelle partie du système l'on souhaite ajouter la fonctionnalité. Un même aspect fonctionnel peut d'ailleurs être utilisé pour différentes parties du même système (recherche de documents et recherche de clients par exemple).

Afin de réaliser cette mise en correspondance du schéma requis par un composant vue avec le modèle de base, notre mécanisme de connexion doit mettre en relation chaque élément vue avec un élément du modèle de base. Pour que cette connexion soit valide, le schéma formé par l'ensemble des éléments issus du modèle de base doit donc être conforme au schéma requis par le composant vue. Il est possible d'effectuer cette vérification à l'aide d'un ensemble de contraintes que nous présentons au chapitre 9.

Nos composants vue étant génériques, ces mises en relation des éléments vue avec les éléments du modèle de base ne peuvent être retrouvées grâce au nommage des éléments, comme cela était le cas avec les approches de structuration par vues. Il est donc nécessaire de lier explicitement chaque élément vue à un élément du modèle de

base.

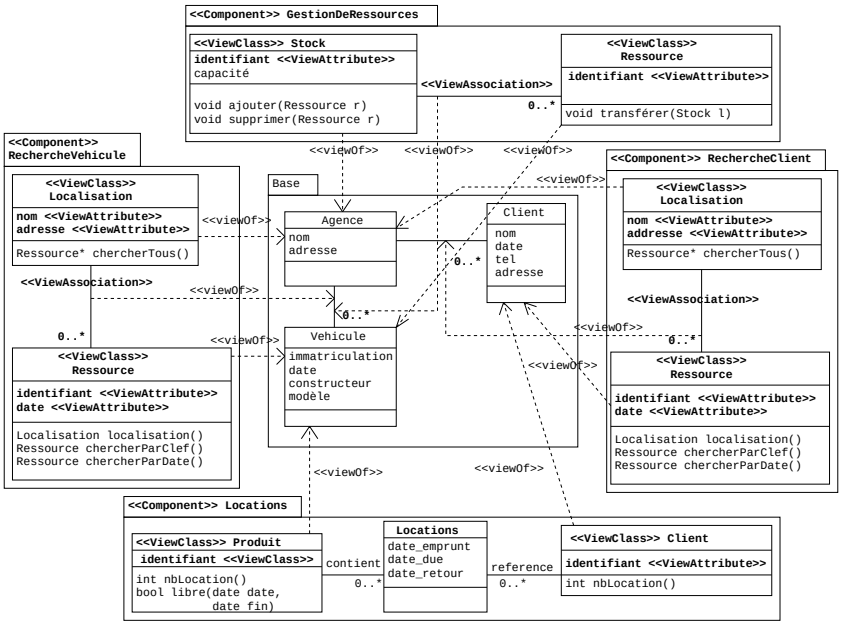

FIG. 5.3 – Reconnexion à une autre base.

La figure 5.3 présente ce mécanisme de connexion pour l'ajout de nos fonctionnalités de gestion des ressources, gestion de locations et de recherche à un système de location de véhicules. Les composants sont connectés au modèle de base et chaque `<<ViewClass>>` est connectée à la classe correspondante, cette connexion est représentée ici par les liens de dépendance `<<viewOf>>`. Les `ViewAttribute` et les `ViewAssociation` sont également connectés aux éléments de la base auxquels ils correspondent. On lie par ce mécanisme les éléments vues aux éléments de la base.

L'application du composant de gestion des ressources, se fait par exemple, en liant la `<<ViewClass>>` *Stock* à la classe du modèle de base *Agence*, la `<<ViewClass>>` *Ressource* à la classe *Vehicule* et la `<<ViewAssociation>>` entre *Stock* et *Ressource* à l'association entre *Agence* et *Vehicule*. Les attributs *identifiant* de *Stock* et de *Ressource* seront respectivement matérialisés par les attributs *nom* de *Agence* et *immatriculation* de *Vehicule*[2].

Cet exemple illustre également que grâce à la généricité de nos composants vue et à notre mécanisme de connexion, il est possible d'appliquer la même fonctionnalité sur des éléments différents du même système. La fonctionnalité de recherche est ici utilisée une première fois pour permettre de rechercher des véhicules par rapport aux agences, et une seconde fois pour rechercher les clients par rapport aux agences.

Cette expérimentation nous a permis de faire le lien entre les notions d'aspects

[2]Les liens entre les `<<ViewAttribute>>` et les attributs ne sont pas ici représentés par souci de lisibilité.

fonctionnels et de composants métiers. Les composants vue ont été outillé à travers un profil UML, sous l'atelier Objecteering, présenté au chapitre 9. Nous avons montré que les aspects fonctionnels forment de bons candidats pour la conception de composants métier génériques, moyennant leur déconnexion de tout contexte particulier.

Nos composants vue souffrent cependant de certaines limites. Même s'il est possible de le retrouver par l'ensemble des éléments vue, le modèle requis est entremêlé avec des éléments propres au composant vue. Il n'y a pas d'explicitation de ce modèle requis. De plus, le mécanisme de connexion entre modèle requis et modèle de base est fastidieux. Il est en effet nécessaire d'établir un lien de dépendance entre chaque élément vue et l'élément du modèle de base correspondant. La vérification de la conformité du modèle fourni par rapport au modèle requis doit donc être réalisée de manière ad-hoc à ce mode de connexion [Mul04].

Enfin, nos composants vue ne peuvent être composés entre eux. Ce qui interdit la conception d'aspects fonctionnels complexes par compositions d'aspects fonctionnels plus simples, ou la réalisation incrémentale d'un système par applications successives de ces aspects fonctionnels. Ce sont ces limitations que nous proposons de dépasser dans le chapitre suivant.

Chapitre 6

Construction de systèmes par application de modèles paramétrés

Le but de notre proposition est de permettre une conception plus facile et plus rapide des systèmes. Les composants vue présentés dans le chapitre précédent apportent une première réponse à ces besoins grâce à la réutilisation de fonctionnalités génériques que nous généralisons et systématisons.

Cependant, afin de rendre ces fonctionnalités génériques facilement utilisables par un concepteur ne les ayant pas lui même définies, il est important de spécifier clairement leurs interfaces. Les modèles paramétrés présentés dans ce chapitre, que nous appelons des composants de modèle, répondent à ce besoin de généricité par une explicitation de leur modèle requis et de leur modèle fourni. Cette explicitation doit également permettre un assemblage plus instinctif des fonctionnalités par mises en correspondance d'un modèle requis avec un modèle fourni.

Si les systèmes sont complexes, les fonctionnalités qui le composent peuvent l'être également. Il est donc important de pouvoir bénéficier des mêmes possibilités que pour la conception du système lui même. Ce qui doit permettre la réalisation de fonctionnalités génériques complexes par composition de fonctionnalités plus simples. De plus, certaines fonctionnalités peuvent se retrouver souvent utilisées conjointement d'un système à un autre. Là encore, il est intéressant de pouvoir composer ces fonctionnalités de manière générique afin de pouvoir réutiliser cette composition d'un système à l'autre.

L'idée que nous présentons dans ce chapitre vise donc à permettre la construction de systèmes par assemblage de fonctionnalités, chacune décrite à l'aide d'un composant de modèle [MCCV05]. Ces derniers peuvent à leur tour être conçus par assemblage de fonctionnalités selon la même technique. Il devient ainsi possible de concevoir des bibliothèques de composants de modèle, correspondant chacun à une fonctionnalité générique, dans lesquelles le concepteur pourra sélectionner ceux qu'il veut utiliser pour son système.

La conception d'un système par un ensemble de modèles génériques peut se faire

selon différents processus : spécification de l'assemblage des composants de modèle constituant le système ou construction incrémentale par sélection et application de composants de modèle progressivement et interactivement. Nos composants de modèles et l'expression de leur assemblage doivent pouvoir être utilisés quelque soit le processus de conception du système. Cela implique notamment que le résultat d'un assemblage de composants de modèle ne doit pas être influencé par le processus utilisé pour le définir.

La première section présente nos composants de modèle. La section 6.2 décrit l'utilisation de ces composants de modèle pour la construction de systèmes complexes.

6.1 Modèle paramètre

Afin de permettre l'assemblage de composants, il est nécessaire d'exprimer leurs interfaces fournies et requises. Dans notre approche ces interfaces sont exprimées à l'aide de modèles. On dépasse ainsi la notion de contrat d'assemblage souvent réduite à une interface de services unitaires. Le modèle requis pour utiliser un composant de modèle est ainsi explicite. Celui-ci est utilisé comme paramètre de son modèle fourni.

Le modèle fourni par un composant est défini par enrichissement du modèle requis par les éléments propres à la fonctionnalité offerte, ce qui permet d'établir le lien entre ces deux modèles. Nos composants peuvent donc être considérés comme des composants boites blanches : la relation entre le modèle requis et le modèle fourni est visible. Dans notre cas, ces modèles sont formés de classes, d'attributs, d'opérations et d'associations.

Afin d'uniformiser la construction des systèmes, les modèles de base sont également considérés comme des composants de modèle. Ils n'ont pas de modèle requis mais expriment un modèle fourni définissant les concepts de base d'un système particulier.

Les figures 6.1, 6.2, 6.3 et 6.4 illustrent un ensemble de composants de modèle utilisables pour la conception de systèmes d'information semblables à ceux présentés dans le chapitre précédent. Chaque composant, exprimé sous la forme d'un paquetage, correspond à une préoccupation récurrente pour ce type de système. Le modèle requis de chacun d'eux est explicité dans le coin supérieur droit. Le schéma figuré à l'intérieur du paquetage de chaque composant correspond à son modèle fourni. Celui-ci est paramétré par le modèle requis qui y est ici représenté à l'aide des éléments en pointillés et italique.

La figure 6.1 illustre un composant pour une fonctionnalité de gestion des stocks. Celui-ci est défini à partir des concepts de stock et de ressource exprimés dans son modèle requis. Le concept de stock doit disposer d'un attribut matérialisant son identifiant et celui de ressource d'un attribut matérialisant sa référence (*ref*). Ces deux concepts doivent également être reliés par une association (*dans*). La fonctionnalité de gestion des stocks est donc exprimée à partir de ce modèle et définit des opérations d'ajout, de suppression et de transfert de ressources. Un attribut permettant de définir la capacité dans stock est également ajouté.

Le composant suivant (cf. figure 6.2) définit une fonctionnalité de recherche de ressources par rapport à des localisations. Ces ressources doivent être datées et disposer d'une clef. Les localisations doivent quant à elles être nommées et associées aux ressources. Ces définitions sont données par le modèle requis. Les opérations ajoutées par

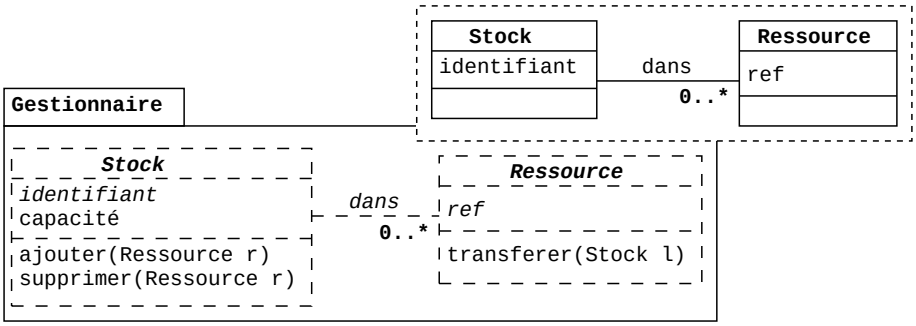

FIG. 6.1 – Composant de gestion des ressources

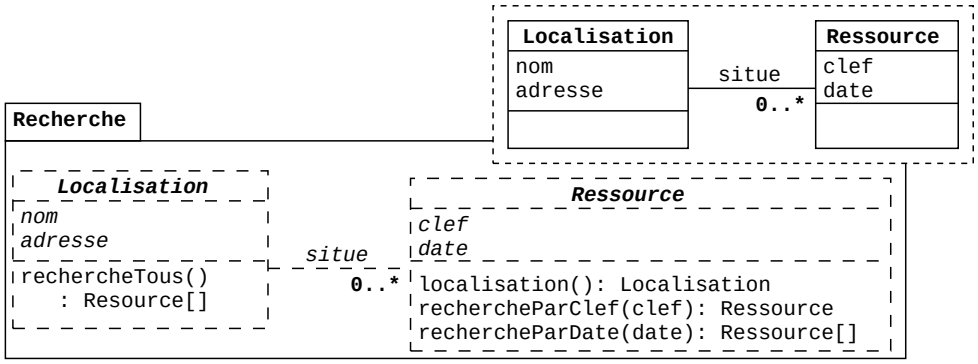

FIG. 6.2 – Composant de recherche

la fonctionnalité permettent la recherche de ressources par rapport aux localisations, par clef ou encore par date.

Le composant d'allocation de ressources (cf. figure 6.3) illustre la possibilité de définir un modèle requis non connexe. Pour appliquer ce composant, il est nécessaire de disposer de produits et de clients, les classes correspondantes à ces concepts pouvant être totalement dissociées. Ce composant a pour but d'allouer des produits à des clients et installe pour cela une relation entre ces entités. Ce composant illustre donc la possibilité de fournir de nouvelles classes, et de nouvelles associations. La classe *Allocation* est ajoutée entre les produits et clients à l'aide de deux associations. Cette classe dispose d'attributs pour la date d'allocation, la date de retour prévue et la date de retour effective, et d'une opération permettant le calcul du coût de l'allocation. Des opérations de gestion des allocations sont également ajoutées aux classes *Produit* et *Client*.

Enfin, notre dernier exemple de composant de modèle (cf. figure 6.4) définit une fonctionnalité mathématique simple. Il permet d'ajouter à une classe qui joue le rôle de compteur, une opération de comptage (total) par rapport à des éléments qui lui sont associés, ces éléments fournissant leur valeur de comptage.

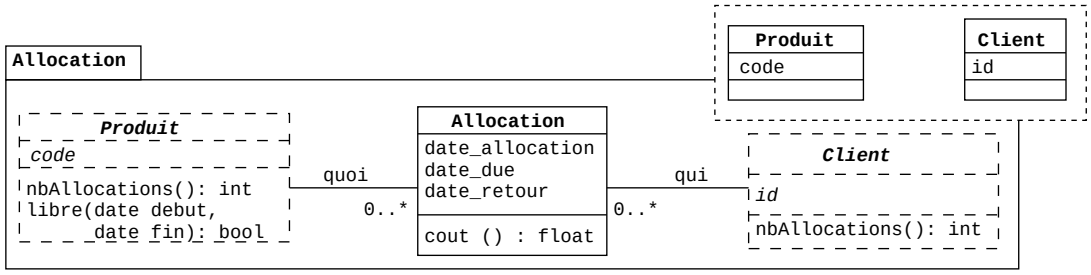

FIG. 6.3 – Composant d'allocation de ressources

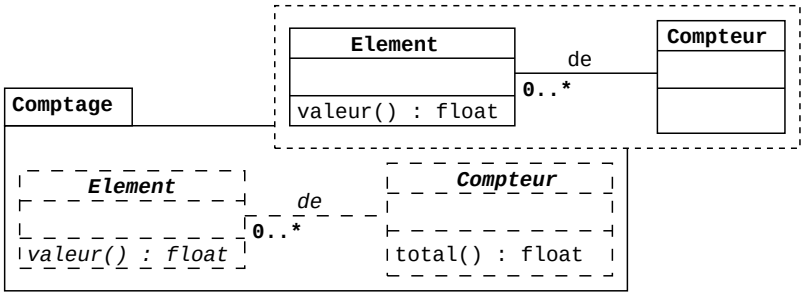

FIG. 6.4 – Composant de calcul

6.2 Application de modèles paramétrés

Lorsque l'on dispose d'un ensemble suffisant de fonctionnalités génériques exprimées à l'aide de nos composants de modèle, la construction d'un nouveau système peut être réalisée grâce à leur assemblage. Cet assemblage doit se baser sur un modèle initial. Celui-ci permet de définir les concepts et les associations propres au domaine du système à construire. C'est ce que nous appelons le modèle de base.

Un exemple de modèle de base pour un système de locations de véhicules est illustré par la figure 6.5. Il définit les concepts d'agence, de véhicule et de client, ainsi que leurs relations. Un certain nombre de propriétés pour ces concepts sont également définies sous forme d'attributs. La figure 6.6 illustre un autre modèle de base, mais pour un système de gestion de bibliothèque tel que vu au chapitre précédent, qui nous permettra d'illustrer la généricité de nos composants de modèle.

Une *composition* permet d'associer un composant de modèle à un autre. Un *assemblage* correspond à un ensemble de compositions. Les systèmes étant complexes, les compositions nécessaires pour construire leurs modèles le sont également. La seule possibilité de composer les fonctionnalités au système de base (comme dans le cas des composants vue) n'est donc pas suffisante. Il est en effet également nécessaire de pouvoir composer les fonctionnalités entre elles. Nous étudions donc dans cette section différents types de composition, que nous appelons des *applications*. Celles-ci sont orientées d'un

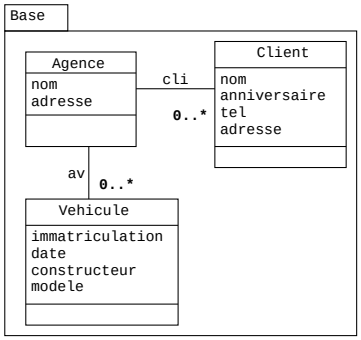

FIG. 6.5 – Système de base de locations de véhicules

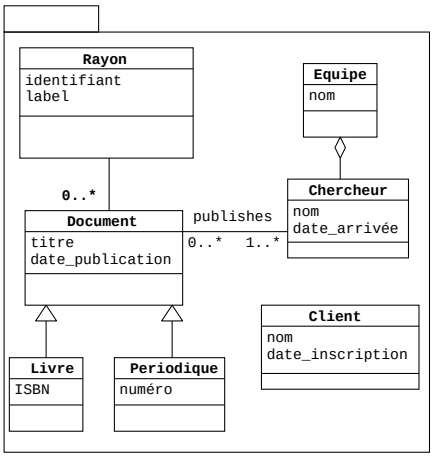

FIG. 6.6 – Système de base d'une bibliothèque

composant de modèle source vers un composant de modèle cible.

Dans tous les cas, la composition est exprimée par une mise en correspondance du modèle requis par un composant avec le modèle fourni par un autre (ou par la base). La fonctionnalité exprimée par le composant est ainsi paramétrée par ce modèle requis. Le modèle fourni doit au moins contenir un sous-modèle conforme au modèle requis, il peut cependant contenir d'autres éléments.

C'est cette notion de conformité entre modèle fourni et modèle requis qui permet de garantir la validité d'une composition. Celle-ci est formalisée à l'aide d'un méta-modèle et de contraintes OCL dans le chapitre 10. Un modèle est conforme à un autre si les éléments qu'il fournit respectent la même structure que les éléments du modèle requis. Nous verrons qu'une application peut-être complète, tous les éléments du modèle requis ont une correspondance dans le modèle fourni, ou partielle, certains éléments n'ont pas de correspondance.

6.2.1 Application d'un composant de modèle à une base

Le premier type de composition est l'application d'un composant de modèle à une base. Cette composition permet de décrire l'ajout de la fonctionnalité exprimée par le composant au modèle du système.

Cette application est réalisée en identifiant dans le modèle de base la partie à laquelle on souhaite ajouter la fonctionnalité exprimée par le composant de modèle. Cette composition est illustrée par la figure 6.7. La fonctionnalité de gestion de stocks est ici appliquée sur les classes *Agence* et *Vehicule*. Cette application est décrite par la mise en correspondance du modèle requis par le composant et la partie du modèle de base mise ici en évidence par le cadre en pointillés.

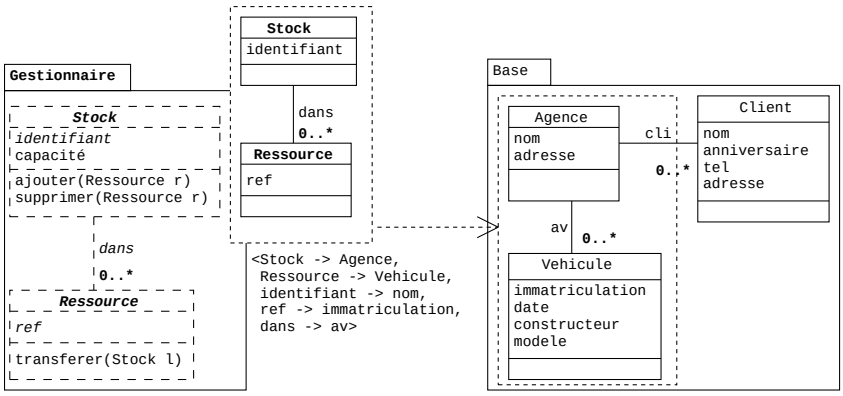

FIG. 6.7 – Application de gestion des stocks au système de location de véhicules

Les éléments du modèle fourni pouvant avoir plusieurs éléments conformes au modèle requis, le concepteur doit préciser pour chacun d'eux celui qu'il souhaite utiliser. C'est ce qu'illustrent les mises en correspondance entre chevrons. Ce mécanisme est décrit en détail au chapitre 10.

Une formulation possible du résultat de l'application du composant *Gestionnaire* à notre base est présentée par la figure 6.8. Nous verrons au chapitre 8 que d'autres formulations du résultat sont possibles, comme la fusion du composant avec la base ou l'utilisation de vues. Dans cette formulation, des éléments paramètres de la fonctionnalité sont mis en relation avec les éléments du modèle de base correspondant à l'aide de liens stéréotypés `<<trace>>` (permettant de lier des éléments dénotant le même concept). La classe *Stock* est ainsi reliée à la classe *Agence* et la classe *Ressource* à la classe *Vehicule*.

Grâce à leur généricité, un même composant peut être appliqué à différentes bases. C'est ce qu'illustre la figure 6.9. Le composant de gestion des stocks est ici utilisé pour ajouter cette fonctionnalité à notre modèle de gestion de bibliothèque. Le système obtenu doit ainsi permettre la gestion des *Document* relativement aux *Rayon*.

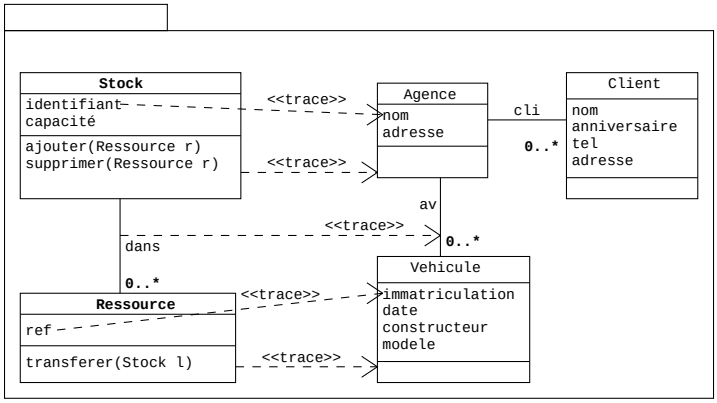

FIG. 6.8 – Système de base avec les fonctions de gestion des véhicules

6.2.2 Application d'un composant de modèle à un autre

Comme nous l'avons évoqué, il est important de permettre l'application d'un composant de modèle à un autre pour la construction de nouveaux composants. Cette possibilité doit permettre la construction de composants répondant à des fonctionnalités complexes à partir de fonctionnalités plus simples. Elle doit également permettre de regrouper des fonctionnalités complémentaires dans un même composant. Les composants ainsi construits peuvent alors être ajoutés à la bibliothèque de composants pour l'enrichir de nouvelles fonctionnalités plus complexes.

L'application d'un composant de modèle à un autre est comparable à l'application d'un composant à un modèle de base. Les éléments du modèle fourni par le composant cible peuvent indifféremment être des éléments paramètres ou de ce dernier. Une différence est qu'une partie du modèle requis peut ne pas être fourni, on parle alors d'application partielle. Dans ce cas cette partie sera requise par le composant résultant de cette application. Elle devra donc être fournie par le modèle de base (ou par un autre composant) lors de son utilisation.

Ce type de composition est illustré par la figure 6.10. Son résultat est ici représenté sous forme fusionnée (cf. section 8). Le composant de recherche est appliqué (a) au composant de gestion de stocks afin de construire un nouveau composant (b) permettant à la fois la recherche et la gestion de ressources dans les stocks. Dans cet exemple, la plus grande partie du modèle requis par le composant de recherche est fournie par le composant de gestion des stocks. Seuls les attributs *date* de la classe *Ressource* et *adresse* de la classe *Localisation* ne sont pas substitués. Ceux-ci se retrouvent donc dans le modèle requis du composant résultant (b). Ce modèle requis est ainsi construit par fusion du modèle requis du composant cible avec la partie non substituée du modèle requis du composant source. La formalisation de cette construction est présentée dans le chapitre suivant.

La figure 6.11 permet d'illustrer la possibilité d'appliquer un composant à des éléments indifféremment paramètres ou non. L'élément compteur de *Comptage* est ap-

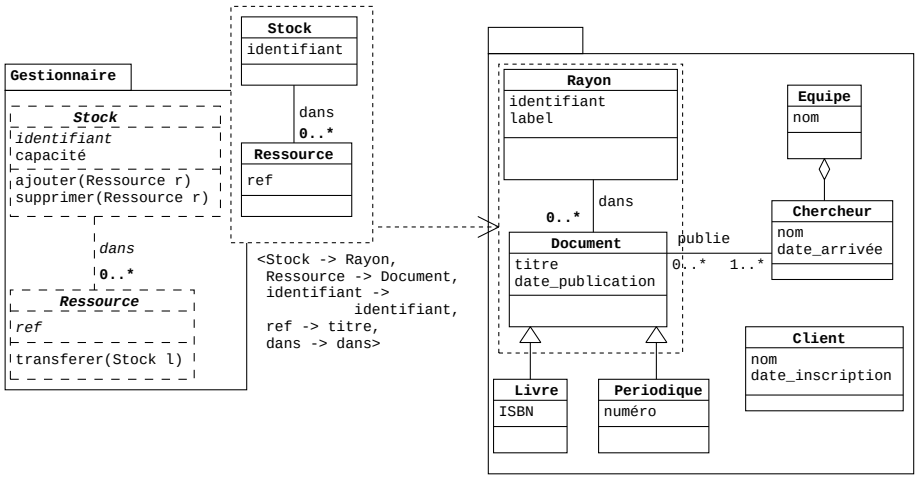

FIG. 6.9 – Application de gestion des stocks au système de bibliothèque

pliqué à un élément paramètre de *Allocation* (*Client*), alors que *Element* est appliqué à un élément non paramètre (*Allocation*).

6.2.3 Chaînes d'applications

Enfin, pour supporter l'évolution incrémentale des systèmes par ajout ou suppression de fonctionnalités, il est nécessaire de pouvoir exprimer ce que nous appelons des chaînes d'applications. Celles-ci sont constituées d'une suite d'applications. La figure 6.11 illustre une telle chaîne d'applications. Le composant *Comptage* est appliqué au composant *Allocation* qui lui même est appliqué à la base.

Pour qu'une suite d'applications puisse être considérée comme une chaîne, les applications qui la composent doivent être complètes. C'est-à-dire que le modèle requis d'un composant doit être entièrement fourni par le composant auquel il est directement appliqué. Cela a pour conséquence qu'un composant n'est directement lié qu'au composant suivant. Cette propriété permet de garantir qu'en cas de suppression d'une application, les autres applications de la chaîne restent valides.

L'application de *Recherche* à *Gestionnaire* illustré par la figure 6.10 n'étant pas complète, elle ne peut faire partie d'une chaîne. En effet, le composant de modèle résultant de cette application devra à son tour être appliqué pour être utilisé (cf. figure 6.12). Cette dernière application ne peut être valide si la première est supprimée. *Recherche* a besoin d'éléments venant à la fois de *Gestionnaire* (*Stock* par exemple) et de *Base* (*addresse* et *date*).

Il est donc possible de couper une chaîne de composants sans remise en cause des autres applications. Dans notre exemple, si on supprime l'application de *Comptage*, l'application de *Allocation* à la base reste valide. De même, si on supprime l'application de *Allocation* à la base, l'application de *Comptage* à *Allocation* reste elle aussi valide.

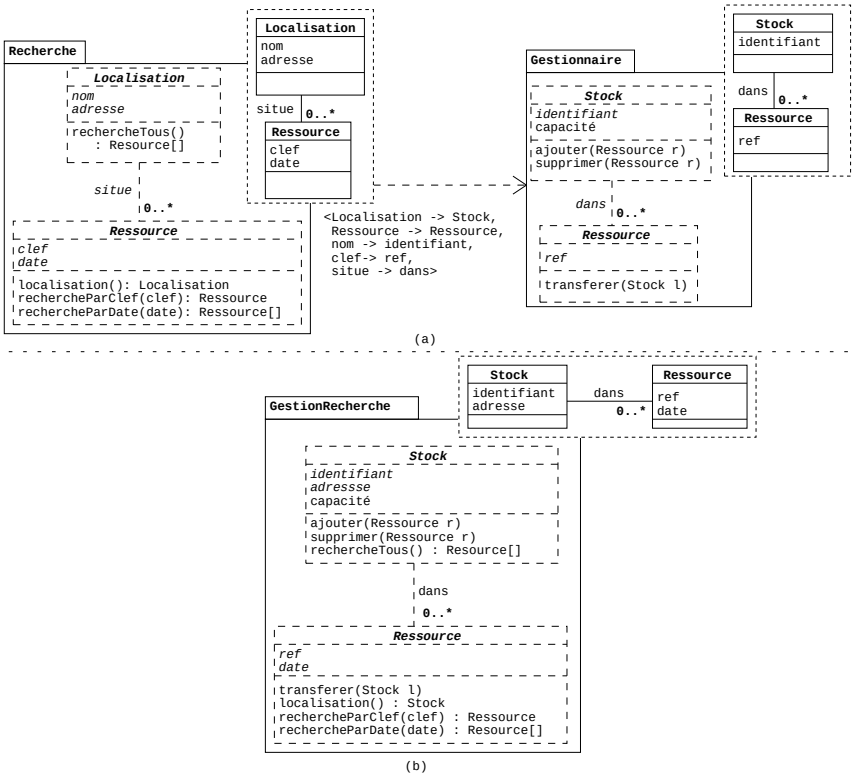

FIG. 6.10 – Application d'un composant de modèle à un autre

6.2.4 Construction de systèmes

La conception d'un système à partir de son modèle initial requiert l'ajout d'un grand nombre de fonctionnalités. Il est donc nécessaire de pouvoir utiliser l'ensemble des types de composition présentés ici : ajout d'une fonctionnalité directement au modèle initial, compositions de fonctionnalités et utilisation de chaînes de compositions.

La figure 6.12 illustre le modèle d'assemblage d'un système de locations de véhicules à partir du modèle de base (cf. figure 6.5) et de notre bibliothèque de composants de modèle. Tous les types de compositions sont ici utilisés.

Le système ainsi construit dispose des fonctionnalités de gestion et de recherche des véhicules par rapport aux agences, de recherche de clients dans les agences, ainsi que de gestion des locations de véhicules et de leur coût. Nous utilisons le composant construit à la section 6.2.2 pour ajouter à la fois les fonctionnalités de gestion des stocks de véhicules et de recherche dans ces stocks. Le composant de recherche est également utilisé pour ajouter la fonctionnalité de recherche des clients par rapport aux agences. Cette application illustre la possibilité d'utiliser le même composant pour différentes parties du même système. Enfin, la chaîne d'application présentée dans la section précédente est également utilisée ici pour ajouter les fonctionnalités de gestion

des locations de véhicules et de calcul de leur coût.

Nous avons présenté et illustré dans ce chapitre les capacités de (ré-)utilisation de nos composants de modèle : utilisation d'un même composant pour la conception de différents systèmes (cf. figure 6.7 et 6.9), applications d'un ensemble de composants à une même base (cf. figure 6.12), construction de composants complexes à partir de composants plus simples (cf. figure 6.10) et possibilité d'utiliser des chaînes d'applications (cf. figure 6.11).

Les principaux avantages attendus d'une telle approche sont la réduction des temps de conception et l'amélioration de la qualité des systèmes ainsi construits. Ces objectifs sont atteints grâce à la réutilisation de composants de modèle éprouvés. De plus la préservation du découpage selon les différentes préoccupations permet une meilleure maîtrise de la complexité. Enfin, cette approche permet également l'évolution des systèmes par ajout de nouvelles fonctionnalités sans remise en cause du modèle déjà construit.

Ce mécanisme de composition permet donc l'expression de systèmes complexes à l'aide d'un assemblage de composants de modèle. Différents processus peuvent être utilisés pour définir cet assemblage. Il est, par exemple, possible d'ajouter une à une les fonctionnalités au système ou au contraire de spécifier son assemblage complet comme illustré à la figure 6.12.

Quelque-soit le processus utilisé, le modèle résultat doit être le même. Notamment, l'ordre d'évaluation des applications ne doit pas influencer ce résultat. Par exemple (figure 6.12), *Recherche* doit pouvoir être appliqué à la base, avant ou après *Allocation* et produire le même résultat. Il est en effet important, pour des raisons de cohérence et également pratiques, que le système obtenu soit indépendant du processus utilisé pour sa construction. C'est ce que nous proposons de démontrer dans le chapitre suivant à l'aide d'une formalisation des composants de modèle et des applications.

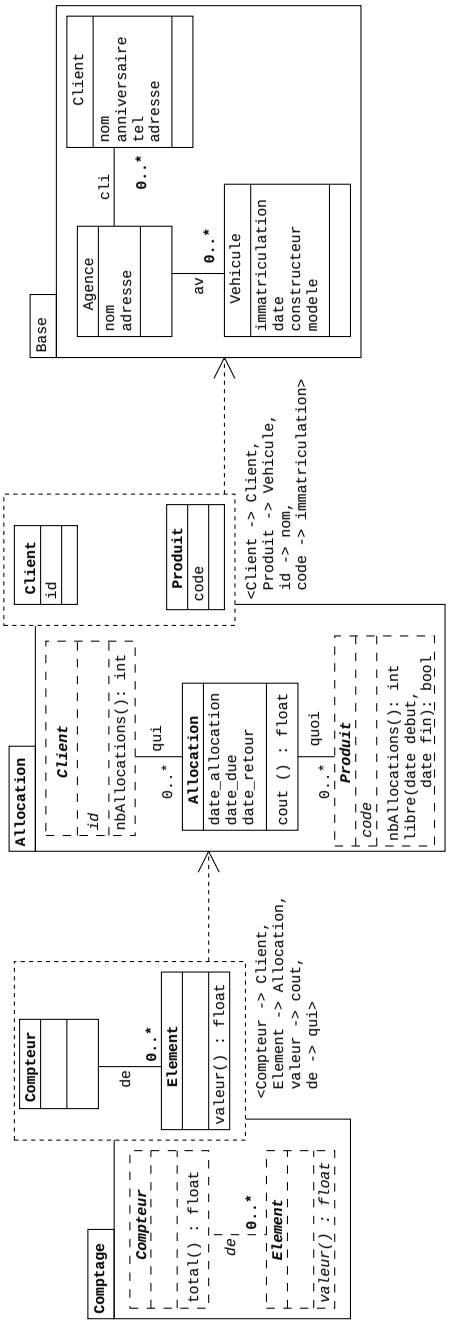

FIG. 6.11 – Chaîne d'applications de *Comptage* à *Allocation* à *Base*

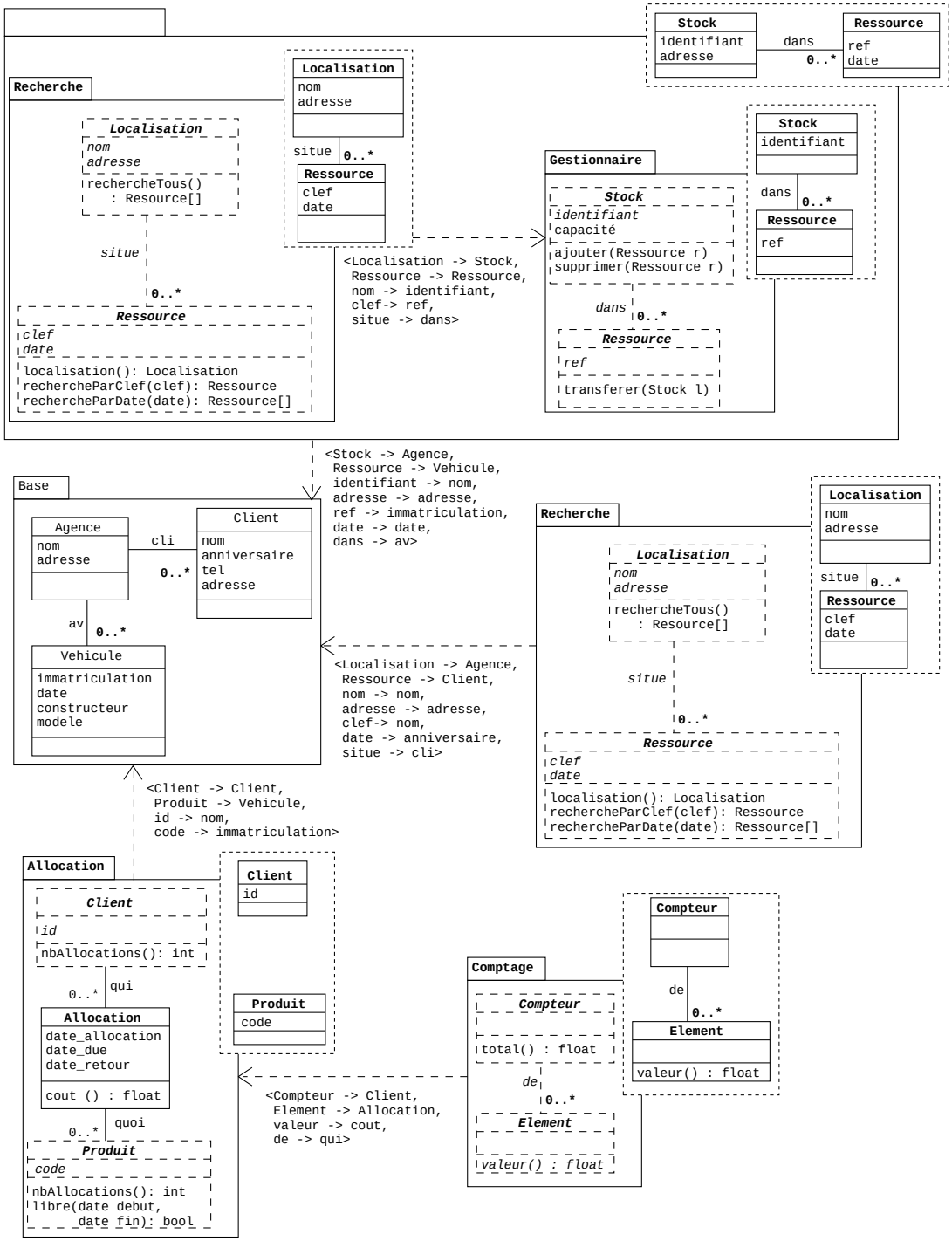

FIG. 6.12 – Modèle d'assemblage du système de location de véhicules

Chapitre 7

Un opérateur d'application de modèles paramétrés

Afin de pouvoir garantir des propriétés sur les modèles construits à l'aide de notre approche, il est nécessaire de donner une définition formelle de nos composants de modèle et du mécanisme d'application. C'est ce que nous proposons de faire dans ce chapitre. Nous exprimons pour cela nos composants de modèles sous forme d'ensemble d'éléments de modélisation et nous définissons un opérateur d'application que nous nommons *apply*.

La mise en relation d'un modèle requis avec un modèle fourni est ici exprimée par un ensemble de correspondances mettant en relation un élément du modèle requis avec un élément du modèle fourni correspondant. Les éléments issus du modèle requis sont appelés les paramètres formels. Les éléments issus du modèle fourni sont appelés les paramètres effectifs.

Nous donnons dans la première section les définitions formelles des composants de modèle et de l'opérateur d'application. En nous basant sur ces définitions nous donnons dans la section 7.2 des propriétés sur l'ordre d'évaluation d'un ensemble d'applications, ainsi que leurs démonstrations [MCCV05].

7.1 Définitions

Nous considérons dans cette section les modèles sous forme d'ensembles d'éléments pouvant être aussi bien des classes, des attributs, des opérations ou des associations. On note $\mathscr{E}$ l'ensemble de tous les éléments de modèles. Dans le cas d'un modèle paramétré, le modèle requis est exprimé par un ensemble d'éléments paramètres. L'opérateur *apply* construit des modèles contenant un ensemble de relations de correspondance exprimées par des paires d'éléments (correspondants aux liens de `<<trace>>` illustrés figure 6.8). La définition suivante généralise tous les types de modèles.

Définition 1 *Un modèle A est défini par un triplet (E_A, P_A, V_A). E_A est un ensemble d'éléments de modèle ($\subset \mathscr{E}$). $P_A \subset E_A$ est un ensemble de paramètres. V_A est un*

ensemble de relations de correspondance définies dans $(E_A \times E_A)$.

Notons que, dans le cas de modèles non paramétrés, P_A est vide et que dans le cas d'un modèle simple (qui n'est pas le résultat d'une application), V_A est vide.

Définition 2 *Deux modèles sont égaux si et seulement si, ils contiennent le même ensemble d'éléments, ils ont le même ensemble de paramètres et ils ont le même ensemble de relations de correspondance.*

Soit deux modèles Z *et* R, $Z = R \Leftrightarrow \begin{cases} V_Z = V_R \\ E_Z = E_R \\ P_Z = P_R \end{cases}$

En nous basant sur cette définition, nous spécifions l'opérateur *apply* comme suit :

Définition 3 *On note* $R = B \underset{s}{\rightarrow} A$ *l'application d'un modèle paramétré* B *à un modèle* A *conformément à l'ensemble de correspondances* s.

On note FP_s *l'ensemble des paramètres formels et* EP_s *l'ensemble des paramètres effectifs de* s.

Le modèle résultat R *est construit conformément aux règles suivantes :*

$$R = B \underset{s}{\rightarrow} A \Rightarrow R = \begin{cases} V_R = V_B \cup s \cup V_A \\ E_R = E_B \cup E_A \\ P_R = (P_B \setminus FP_s) \cup P_A \end{cases}$$

Les modèles source et cible d'une application paramétrée ne peuvent partager des éléments : $E_A \cap E_B = \varnothing$.
Les paramètres formels sont des éléments du modèle source : $FP_s \subseteq P_B$.
Les paramètres effectifs sont des éléments du modèle cible : $EP_s \subseteq E_A$.

Notons que, conformément à ces définitions, les modèles paramétrés peuvent être appliqués à n'importe quel type de modèles, paramétrés (cf. figure 6.10) ou non (cf. figure 6.7). Dans le cas d'un modèle cible paramétré, le modèle résultant est lui même paramétré. Rappelons que les paramètres du modèle résultant sont ceux du modèle cible complétés par ceux non substitués du modèle source. La règle de calcul des paramètres résultant (P_R) formalise cela.

7.2 Propriétés

À partir de ces définitions, il est possible de démontrer un ensemble de propriétés qui garantissent la validité d'un ensemble d'applications et de leurs différentes alternatives pour un même assemblage.

Propriété 1 *L'ordre d'application de deux modèles à un troisième n'influence pas le résultat.*
Soit deux ensembles de correspondances s, s' *tel que* $EP_s \subseteq E_A$ *et* $EP_{s'} \subseteq E_A$. *Alors* $B \underset{s}{\rightarrow} (C \underset{s'}{\rightarrow} A) = C \underset{s'}{\rightarrow} (B \underset{s}{\rightarrow} A)$

Grâce à cette propriété, il n'est pas nécessaire d'exprimer dans quel ordre les modèles paramétrés doivent être appliqués. Ceci est illustré figure 6.12. Quelque soit l'ordre d'application de *Recherche* et d'*Allocation* à la base, le modèle résultat est le même.

Démonstration *Soit* $Z = B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A)$,

$$Z = B \underset{s}{\rightarrow} Z' \text{ avec } Z' = (C \underset{s'}{\longrightarrow} A) \Rightarrow Z' = \begin{cases} V_{Z'} = V_C \cup s' \cup V_A \\ E_{Z'} = E_C \cup E_A \\ P_{Z'} = (P_C \setminus FP_{s'}) \cup P_A \end{cases}$$

$$\Rightarrow Z = \begin{cases} V_Z = V_B \cup s \cup V_{Z'} \\ E_Z = E_B \cup E_{Z'} \\ P_Z = (P_B \setminus FP_s) \cup P_{Z'} \end{cases}$$

$$\Rightarrow Z = \begin{cases} V_Z = V_B \cup s \cup V_C \cup s' \cup V_A \\ E_Z = E_B \cup E_C \cup E_A \\ P_Z = (P_B \setminus FP_s) \cup (P_C \setminus FP_{s'}) \cup P_A \end{cases}$$

Soit $Y = C \underset{s'}{\longrightarrow} (B \underset{s}{\rightarrow} A)$,

$$Y = C \underset{s'}{\longrightarrow} Y' \text{ avec } Y' = (B \underset{s}{\rightarrow} A) \Rightarrow Y' = \begin{cases} V_{Y'} = V_B \cup s \cup V_A \\ E_{Y'} = E_B \cup E_A \\ P_{Y'} = (P_B \setminus FP_s) \cup P_A \end{cases}$$

$$\Rightarrow Y = \begin{cases} V_Y = V_C \cup s' \cup V_{Y'} \\ E_Y = E_C \cup E_{Y'} \\ P_Y = (P_C \setminus FP_{s'}) \cup P_{Y'} \end{cases}$$

$$\Rightarrow Y = \begin{cases} V_Y = V_C \cup s' \cup V_B \cup s \cup V_A \\ E_Y = E_C \cup E_B \cup E_A \\ P_Y = (P_C \setminus FP_{s'}) \cup (P_B \setminus FP_s) \cup P_A \end{cases}$$

Ainsi nous avons $Z = Y \Rightarrow B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A) = C \underset{s'}{\longrightarrow} (B \underset{s}{\rightarrow} A)$

Propriété 2 *Pour toute séquence* $B \underset{s_1}{\longrightarrow} (C \underset{s'_1}{\longrightarrow} A)$, *il existe une séquence* $(B \underset{s_2}{\longrightarrow} C) \underset{s'_2}{\longrightarrow} A$, *qui produit le même résultat, tel que* $s_2 = s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1)$ *et* $s'_2 = s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1)$[1]

La figure 7.1 montre l'application du modèle générique *Gestionnaire* à la base puis l'application de *Recherche* au résultat. C'est une chaîne d'applications alternative à l'application de *Recherche* à *Gestionnaire* puis à la base. Cette dernière construction est équivalente à l'application du modèle générique *GestionRecherche* à la base. Nous pouvons vérifier que leurs ensembles de paramètres sont conformes à cette propriété.

Les éléments paramètres de *Recherche* appliqués à des éléments de la base dans la figure 7.1 (*adresse* et *date*) ne sont pas utilisés pour l'application directe (figure 6.10) de *Recherche* à *Gestionnaire*. Il sont ainsi transférés à l'application de *GestionRecherche* à la base dans la figure 6.12.

[1] A noter que, pour être valide s_2 ne peut contenir d'élément dans $(E_A \times \mathscr{E})$. $s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1)$ est donc la seule forme que l'on peut déduire pour s_2 à partir de s_1.

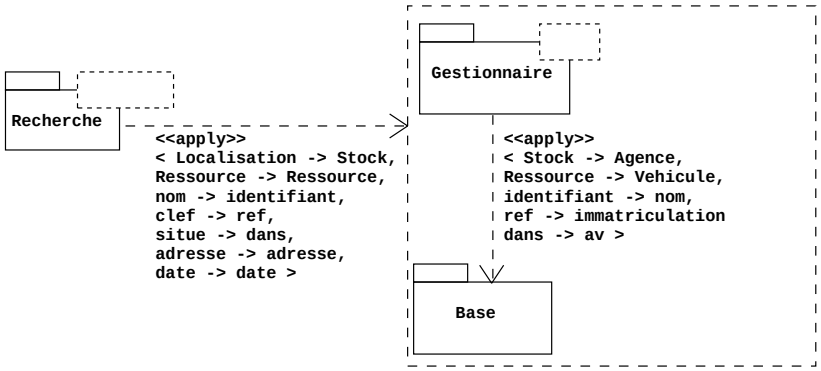

FIG. 7.1 – Une alternative à l'application de *GestionRecherche*

Afin de démontrer cette propriété, nous avons besoin de la propriété intermédiaire suivante sur la substitution des paramètres.

Propriété 3 *Un paramètre ne peut être substitué qu'une seule fois. Une fois substitué, un paramètre n'est plus paramètre dans le modèle résultant.*
$(B \underset{s_1}{\longrightarrow} C) \underset{s_2}{\longrightarrow} A \Rightarrow FP_{s_1} \cap FP_{s_2} = \varnothing$.

Démonstration *Soit* $(B \underset{s_1}{\longrightarrow} C) \underset{s_2}{\longrightarrow} A = R \underset{s_2}{\longrightarrow} A$ *avec* $R = B \underset{s_1}{\longrightarrow} C$.
Par définition, $P_R = (P_B \setminus FP_{s_1}) \cup P_C$ *mais* $FP_{s_1} \subseteq P_B$ *et* $P_B \cap P_C = \varnothing$ *(*$E_B \cap E_C = \varnothing$, $P_B \subset E_B$ *et* $P_C \subset E_C$*) donc* $FP_{s_1} \cap P_C = \varnothing$.

$$\begin{aligned} \Rightarrow P_R &= (P_B \setminus FP_{s_1}) \cup (P_C \setminus FP_{s_1}) \\ &= (P_B \cup P_C) \setminus FP_{s_1} \end{aligned}$$

$\Rightarrow P_R \cap FP_{s_1} = \varnothing$ *et par définition* $FP_{s_2} \subseteq P_R$ *donc* $FP_{s_1} \cap FP_{s_2} = \varnothing$.

Nous pouvons maintenant démontrer la propriété 2.

Démonstration *Soit* $R = B \underset{s_1}{\longrightarrow} (C \underset{s'_1}{\longrightarrow} A)$,

$$R = \begin{cases} V_R = V_B \cup s_1 \cup V_C \cup s'_1 \cup V_A \\ E_R = E_B \cup E_C \cup E_A \\ P_R = (P_B \setminus FP_{s_1}) \cup (P_C \setminus FP_{s'_1}) \cup P_A \end{cases}$$

Par définition $E_B \cap (E_C \cup E_A) = \varnothing$ *donc* $E_B \cap E_C = \varnothing$, *puisque* $P_B \subseteq E_B$ *et* $P_C \subseteq E_C$, *on a :*
$P_B \cap P_C = \varnothing$ *et* $FP_{s_1} \cap FP_{s'_1} = \varnothing \Rightarrow P_R = (P_B \cup P_C \cup P_A) \setminus (FP_{s_1} \cup FP_{s'_1})$

$$\textit{Soit } R' = (B \underset{s_2}{\longrightarrow} C) \underset{s'_2}{\longrightarrow} A,\ R' = \begin{cases} V_{R'} = V_B \cup s_2 \cup V_C \cup s'_2 \cup V_A \\ E_{R'} = E_B \cup E_C \cup E_A \\ P_{R'} = ((P_B \setminus FP_{s_2}) \cup P_C) \setminus FP_{s'_2} \cup P_A \end{cases}$$

Conformément à la propriété 3 :
$P_B \cap P_C = \varnothing$ *and* $FP_{s_2} \cap FP_{s'_2} = \varnothing \Rightarrow P'_R = (P_B \cup P_C \cup P_A) \setminus (FP_{s_2} \cup FP_{s'_2})$. *Ainsi nous obtenons* $R = R'$ *si* $s_1 \cup s'_1 = s_2 \cup s'_2$
Puisque $s_2 = s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1)$ *et* $s'_2 = s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1)$,

$$\begin{aligned} s_2 \cup s'_2 &= s_1 \setminus ((E_A \times \mathscr{E}) \cap s_1) \cup s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1) \\ &= s_1 \cup s'_1 \cup ((E_A \times \mathscr{E}) \cap s_1) \setminus ((E_A \times \mathscr{E}) \cap s_1) \\ &= s_1 \cup s'_1 \end{aligned}$$

Un cas particulier de cette propriété 2 se présente lorsque, pour chaque application, tous les paramètres du modèle source sont substitués avec des éléments du modèle cible. On se trouve donc dans le cas d'une chaîne d'applications. Dans ce cas, chaque modèle paramétré peut être directement appliqué au modèle suivant dans la chaîne d'application. Pour ces chaînes d'applications, l'ordre d'évaluation n'influence pas le résultat. Ceci est formalisé par la propriété suivante :

Propriété 4 *Soit* $B \underset{s}{\rightarrow} C \underset{s'}{\longrightarrow} A$ *une chaîne d'application telle que* $EP_s \subseteq E_C$, *elle peut être évaluée aussi bien comme* $B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A)$, *que comme* $(B \underset{s}{\rightarrow} C) \underset{s'}{\longrightarrow} A$.

Démonstration *Conformément à la propriété 2 :*
$$\textit{soit } B \underset{s}{\rightarrow} (C \underset{s'}{\longrightarrow} A) = (B \underset{s_2}{\longrightarrow} C) \underset{s'_2}{\longrightarrow} A \textit{ avec } \begin{cases} s_2 = s \setminus ((E_A \times \mathscr{E}) \cap s) \\ s'_2 = s' \cup ((E_A \times \mathscr{E}) \cap s) \end{cases}$$

Puisque $EP_s \subseteq E_C$, $EP_s \cap E_A = \varnothing$ *(par définition* $E_C \cap E_A = \varnothing$*). Nous pouvons en déduire* $(E_A \times \mathscr{E}) \cap s = \varnothing$ *(il ne peut exister* x *tel que* $x \subseteq EP_s$ *et* $x \subseteq E_A$*)*
$$\Rightarrow \begin{cases} s_2 = s \\ s'_2 = s' \end{cases}$$
Ce qui démontre la propriété.

Cette dernière propriété permet, dans l'exemple figure 6.12, d'appliquer *Comptage* à *Allocation* à *Base*, sans préciser un quelconque ordre d'évaluation.

Cet ensemble de propriétés nous permet de garantir que quelque soit l'ordre d'évaluation des différentes applications d'un assemblage, le résultat est toujours le même. L'expression de ce résultat, et notamment de l'ensemble de relations V, peut cependant prendre plusieurs formes. C'est ce que nous présentons dans le chapitre suivant. Les règles permettant de garantir qu'une application est valide sont quant à elles présentées au chapitre 10.

Chapitre 8

Expressions du modèle résultant

L'opérateur de composition présenté dans le chapitre précédent permet donc de concevoir un système par assemblage d'un modèle de base et d'un ensemble de composants de modèle, cela indépendamment du processus de conception lui même. Le modèle du système résultant de cet assemblage doit lui aussi pouvoir être exprimé indépendamment de ce processus.

Nous présentons dans ce chapitre deux modes d'obtention d'un modèle de système, et la façon de les obtenir à partir d'un modèle d'assemblage. Le premier est basé sur la fusion des différentes fonctionnalités avec le modèle initial et permet d'obtenir un modèle classes/associations classique. Le second mode permet de préserver la structuration. Il est possible d'utiliser pour cela différents mécanismes, des relations de traces ou un mécanisme de vues par exemple.

Afin d'illustrer ces modes d'obtention, nous utiliserons le système construit par l'assemblage de la figure 8.1. Le système décrit ici correspond au modèle initial de location de véhicules auquel on ajoute les fonctionnalités de gestion de stock et de recherche. La fonction de gestion de stock doit permettre la gestion des véhicules par rapport aux agences. Les éléments *Stock* et *Ressource* du modèle requis du composant sont donc fournis par les éléments *Agence* et *Vehicule* du modèle initial. Le composant de recherche est ici utilisé pour disposer des fonctionnalités de recherche de véhicules dans les agences. Il est donc également appliqué aux éléments *Agence* et *Vehicule*.

8.1 Fusion

La représentation la plus directe pour obtenir le système issu d'un assemblage, consiste à fusionner les éléments définis par les différents composants dans un modèle unique. Tous les éléments propres à la fonctionnalité, c'est-à-dire ne faisant pas partie du modèle paramètre, sont ajoutés au modèle auquel le composant est appliqué. Les attributs et opérations introduits par les composants sont ajoutés aux classes correspondantes. De même, les classes et associations introduites par les composants sont ajoutées au modèle.

Dans un composant, une opération peut être définie à l'aide de paramètres ou une

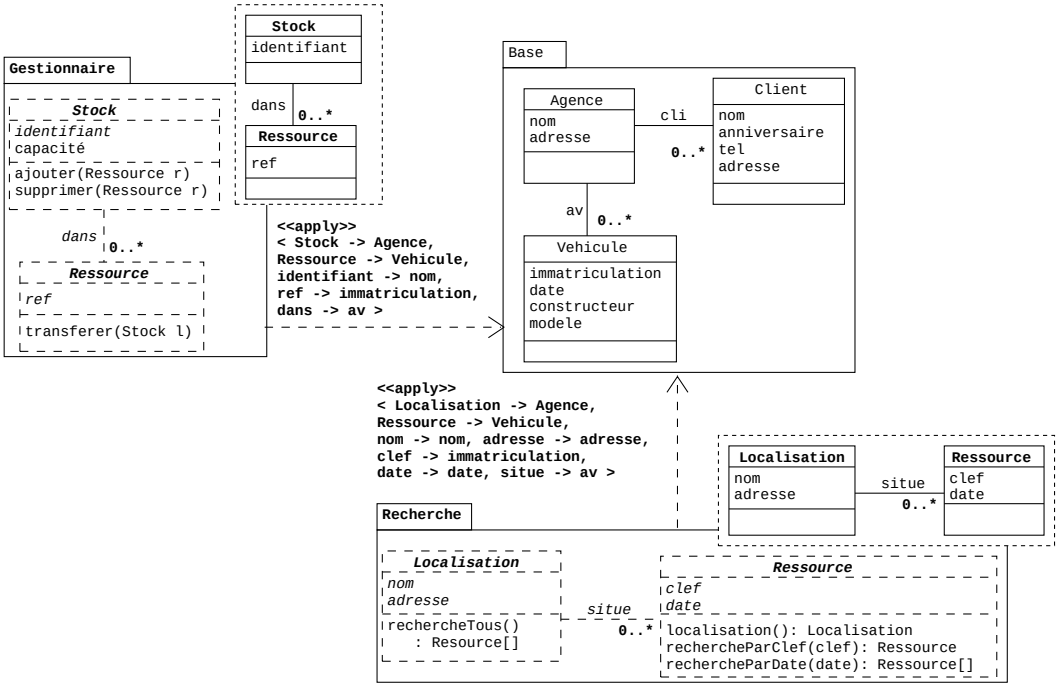

FIG. 8.1 – Gestion de stock et recherche de véhicules

valeur de retour typés par des éléments appartenant au modèle requis. Dans ce cas, ceux-ci doivent être typés dans le modèle fusionné, par les éléments du modèle initial correspondant.

Le modèle ainsi obtenu est un modèle classes/associations classique qui pourra être directement mis en œuvre avec tout langage ou plate-forme à objets.

Ce mode de représentation par fusion est illustré par la figure 8.2 pour le système de location de véhicules conçu à partir de notre assemblage. L'attribut *capacite* et les opérations *ajouter* et *supprimer* définis pour un stock sont ajoutés à l'entité *Agence*. Celle-ci est également enrichie par l'opération *rechercherTous* issue de la fonctionnalité de recherche. De même, l'entité *Client* est enrichie par les opérations relatives à cette fonctionnalité de recherche.

Cet exemple permet également d'illustrer le typage cohérent des paramètres et valeurs de retour des opérations ajoutées aux entités. L'opération permettant l'ajout d'une ressource dans un stock (*ajouter (r : Ressource)*) est bien définie dans le modèle résultat comme une opération permettant l'ajout d'un véhicule à une agence (*ajouter (r : Vehicule)*).

Des éléments issus de différents composants étant fusionnés dans un modèle unique, des conflits apparaissent si des éléments de même nom doivent être ajoutés au même espace de nommage. En effet, dans un modèle tel que présenté figure 8.2, l'identification des éléments se fait par leur nom et l'élément qui les contient. Il peut alors être nécessaire de mettre en place une stratégie de renommage des éléments devant être fusionnés.

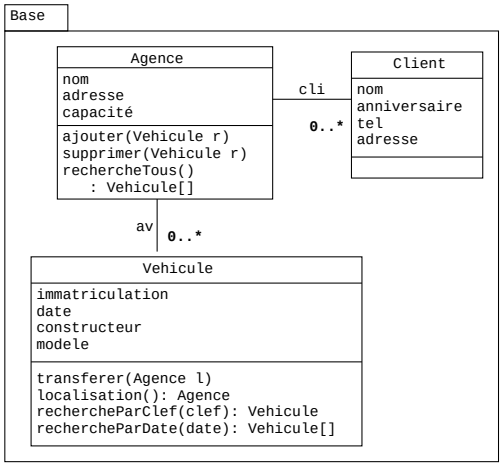

FIG. 8.2 – Fusion des différentes fonctionnalités

Cette représentation permet d'exprimer le système selon un modèle objet conventionnel, mais elle ne permet pas de préserver la structuration de ce système selon ses différentes fonctionnalités. C'est cette possibilité que nous proposons d'étudier dans la section suivante.

8.2 Préservation de la structuration

La préservation de la structuration du système introduite lors de sa conception a pour but de répondre aux besoins, déjà évoqués, de traçabilité, de maîtrise de la complexité ou encore d'évolution. Nous présentons dans cette section des moyens d'obtention du modèle résultat d'un assemblage répondant à ces besoins. Les premiers utilisent des relations de trace ou nos composants vues, les seconds exploitent le principe de représentation des systèmes à l'aide de vues.

8.2.1 Traçabilité des composants de modèle

Il est donc possible de représenter le système résultat à l'aide de composants vues. On déduit pour chaque composant de modèle un composant vue en identifiant chaque élément paramètre comme étant un élément vue (*ViewClass*, *ViewAssociation*,...). Les liens de vue (*ViewOf*) peuvent quant à eux se déduire du paramétrage utilisé pour l'application du composant de modèle correspondant.

La figure 8.3 illustre notre système à l'aide de cette représentation. Par construction, le modèle à base de composants vues obtenu à partir d'un assemblage donne un ensemble de connexions valides.

Cette représentation à l'aide de composants vue permet la réutilisation des outils développés pour leur manipulation et offre, comme nous l'illustrons dans la quatrième partie, une modélisation adaptée à la réalisation de composants binaires génériques.

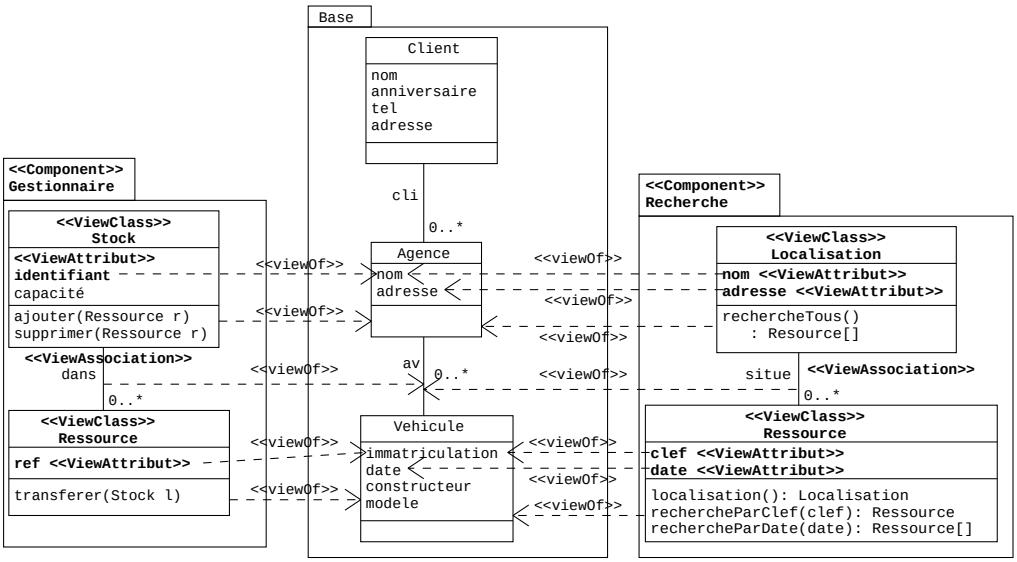

FIG. 8.3 – Représentation éclatée à l'aide de composants vue

Cependant, une fois connectés, l'explicitation des éléments requis n'est plus nécessaire. C'est pourquoi, nous proposons une autre représentation du modèle résultat structuré à l'aide de la relation standard `<<trace>>`. Celle-ci permet de lier entre eux des éléments dénotant le même concept (cf. section 4.2).

Cette représentation consiste à inclure dans le modèle résultat tous les éléments issus des composants de modèle, aussi bien ceux propres à la fonctionnalité que ceux appartenant au modèle requis. Ces éléments, issus du modèle requis, sont liés dans le modèle résultat aux éléments auxquels ils ont été appliqués. On utilise pour cela les relations de `<<trace>>`.

La figure 8.4 illustre une représentation à l'aide de traces pour notre exemple. Les paquetages correspondants aux composants de modèle sont ici conservés (*Gestionnaire* et *Recherche*), ce qui permet de faire figurer la structuration utilisée pour l'assemblage dans le modèle résultat. Les espaces de nommage étant ainsi conservés, des éléments issus de différents composants peuvent porter le même nom sans entrer en conflit. Dans le cas d'applications multiples d'un même composant, recherche client et recherche véhicule par exemple, les paquetages correspondants peuvent être renommés pour les différencier.

On retrouve pour chaque relation de correspondance définie par l'assemblage (figure 8.1) une relation de trace. La mise en correspondance de la classe *Stock*, par exemple, avec la classe *Agence* dans l'assemblage est ici matérialisée par un lien de trace.

Il est possible de retrouver, à partir du modèle à base de traces ou du modèle à base de composants vue, la représentation classes/associations de la section précédente, en fusionnant les éléments liés par des relations de traces (ou de *ViewOf*).

Ces représentations permettent ainsi de garder dans le modèle de système résultat la

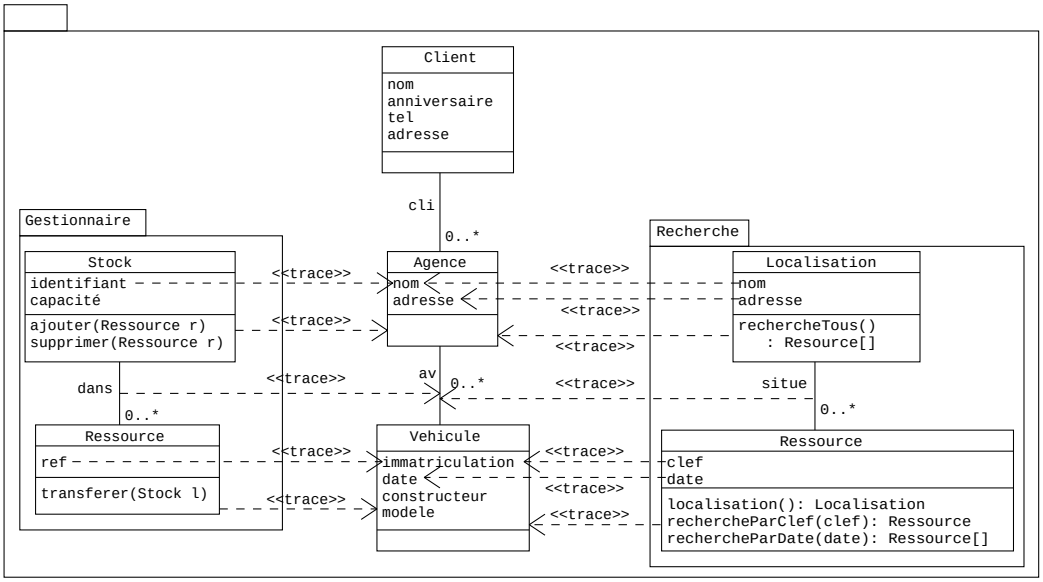

FIG. 8.4 – Représentation éclatée à l'aide de traces

structuration des différentes fonctionnalités, mais aussi une traçabilité des composants utilisés pour sa conception. Il est en effet possible de retrouver à partir de ces modèles les éléments paramètres ainsi que les relations de mise en correspondance utilisées.

8.2.2 Vues

Comme cela a été présenté dans la section 3.1.1, les approches à base de vues permettent une représentation adaptée aux systèmes complexes par une expression claire de la double structuration entités/fonctions. Nous présentons dans cette section la déduction d'une représentation en CROME d'un système conçu par assemblage de composants de modèle.

Dans une représentation à base de vues, les entités des différentes vues sont mis en correspondance avec les entités du modèle de base par leur nom. Pour déduire la vue correspondant à un composant de modèle, il est donc nécessaire de renommer les entités appartenant au modèle requis avec le nom de l'entité du modèle initial avec laquelle elles sont mises en correspondance dans l'assemblage.

Tous les éléments nécessaires à la vue qui n'y sont pas définis le sont dans le modèle de base. Les attributs des entités du modèle requis d'un composant n'apparaissent donc pas dans la vue correspondante.

La figure 8.5 illustre la représentation du système résultat à l'aide d'un modèle à vues. On y retrouve deux vues, l'une correspondant au composant *Gestionnaire*, l'autre au composant *Recherche*.

Les entités *Stock* et *Ressource* du composant *Gestionnaire* par exemple, sont respectivement renommés *Agence* et *Vehicule* dans la vue correspondante. Leurs attributs

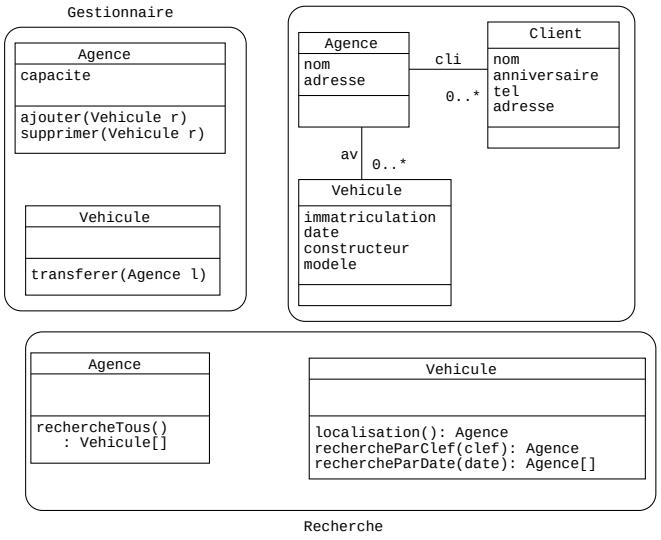

FIG. 8.5 – Représentation à l'aide de vues

requis *identifiant* et *ref*, hérités de la base, n'apparaissent pas dans la vue. Au contraire, l'attribut *capacite* étant un attribut ajouté par le composant, il est défini dans la vue correspondante. Il en est de même pour les opérations définies par les composants.

Il est ainsi possible de déduire une représentation du système par vues à partir d'un modèle d'assemblage. Le système bénéficie alors des propriétés des approches à base de vues.

Chacune des représentations présentées ici permet donc d'exprimer le modèle du système issu d'un assemblage de composants de modèles et offre des propriétés différentes, simplicité pour la représentation fusionnée, traçabilité et gestion de la complexité pour les représentations structurées. De plus, le choix d'un mode de représentation du système peut influencer les propriétés de sa mise en œuvre. Ces modes de représentation ne sont pas limitatifs et d'autres peuvent être utilisés. Notamment, les représentations présentées ici sont extrêmes, entièrement fusionnée ou entièrement structurée. Il peut être intéressant de pouvoir, selon leur granularité par exemple, fusionner certains composants de modèles et déduire les vues correspondantes pour d'autres. Nous étudions ces possibilités au chapitre 12.

Comme nous l'avons vu dans le chapitre précédent, notre approche de construction de systèmes à partir de composants de modèle est indépendante de tout processus. Et, comme nous venons de le voir, elle est aussi indépendante d'une représentation particulière du système résultat. Enfin, les artefacts de modélisation proposés dans cette partie (composants vues et composants de modèle) ne sont pas spécifiques à un langage de modélisation particulier.

Nous proposons dans la partie suivante une méta-modélisation de ces artefacts afin de leur donner une définition stricte et conforme au standard actuel UML2.

Quatrième partie

Méta-modèles

La partie précédente nous a permis de présenter un modèle de vues génériques que nous appelons composants vue (chapitre 5). Nous y avons également décrit notre proposition pour la construction de systèmes par composition de modèles génériques que nous appelons composants de modèles (chapitre 6). Un opérateur de composition a été défini afin de permettre d'évaluer et de garantir la cohérence d'un ensemble de compositions quelle que soit leur ordre d'évaluation (chapitre 7). Enfin nous avons présenté (chapitre 8) différentes expressions du modèle résultant d'un ensemble de compositions, toutes compatibles avec notre opérateur.

Nous nous intéressons dans cette partie à la représentation et à la méta-modélisation des artefacts présentés dans la partie précédente. La réalisation de méta-modèles pour nos artefacts permet de leur donner une définition stricte, cela grâce notamment à l'utilisation du langage OCL. En effet, l'utilisation de contraintes permet d'exprimer des propriétés à respecter qui ne peuvent l'être par une simple définition structurelle du méta-modèle. Cette définition stricte permet également la réalisation d'outils pour la manipulation d'éléments conformes au méta-modèle, tels que celui présentés à la section 13.4.

Les langages de modélisation et de méta-modélisation standards (UML et MOF) sont également définis à l'aide de méta-modèles. Réaliser nos méta-modèles par extension de ces méta-modèles standards permet une utilisation et une diffusion plus faciles, notamment par la disponibilité d'outils supportant ces standards. C'est pourquoi les modèles et méta-modèles présentés dans cette partie sont définis par extension du méta-modèle actuel d'UML 2.

Le premier chapitre (chapitre 9) décrit le méta-modèle des composants vue présentés dans le chapitre 5. Les contraintes proposées permettent de garantir la cohérence des composants vue et de leur assemblage. Le profil utilisé pour leur représentation en UML y est également décrit.

Le chapitre suivant (chapitre 10) décrit l'utilisation des paquetages templates UML2 pour la représentation des composants de modèle. La sémantique de nos composants de modèle étant différente de celle des paquetages templates, il est nécessaire de définir un ensemble de contraintes sur cette représentation. L'idée est de permettre le paramétrage des paquetages templates UML2, non plus par une simple liste d'éléments, mais par un modèle. Nous présentons ensuite l'utilisation de nos composants de modèles au travers de notre relation d'application de composants de modèles *apply*. Nous décrivons le rapport qui existe entre notre relation et la relation standard *bind*. La définition de notre relation *apply* est présentée, toujours par extension du méta-modèle UML2 et à l'aide d'un ensemble de contraintes permettant de garantir la validité des applications. Enfin nous proposons une formalisation de la relation *bind* à l'aide d'un ensemble de contraintes OCL conformes à sa définition standard.

Chapitre 9

Méta-modélisation des composants vue

Les composants vue présentés dans le chapitre 5 permettent la définition de vues fonctionnelles génériques. Ils utilisent des concepts inexistants dans le standard UML, notamment les éléments vue. Pour les représenter à l'aide du langage UML, il est donc nécessaire d'étendre celui-ci. C'est ce que nous présentons dans la première section. La deuxième section présente les contraintes, exprimées en OCL, qui sont associées à ces nouveaux concepts. Enfin, la troisième section présente le profil issu du méta-modèle permettant l'utilisation des composants vue en UML.

9.1 Méta-modèle

Nous proposons dans cette section la définition de nos composants vue par rapport à la version actuelle du méta-modèle UML 2 [UML05b]. Une première version de cette extension, basée sur le méta-modèle UML 1.4 [UML03a], a déjà été réalisée et présentée dans [MCCV03].

La figure 9.1 présente ce méta-modèle. Les éléments sur fond grisé correspondent aux éléments issus du méta-modèle UML 2, les autres correspondent aux concepts que nous avons ajoutés afin de permettre l'expression d'éléments vue. Il est à noter que certains rôles comme `package` ou `featuringClassifier` ne sont pas directement définis sur les concepts de `Class`, d'`Association` ou de `StruturalFeature`, mais le sont par héritage.

Extrait du méta-modèle UML 2

Nous définissons les concepts propres aux composants vue par rapport à cinq concepts issus du méta-modèle UML2 : `Package`, `Class`, `Property`, `Association` et `StructuralFeature`.

Le concept de **Package** permet de regrouper un ensemble de `PackageableElement` dont les concepts de `Class` et d'`Association` font partie. L'ensemble des éléments

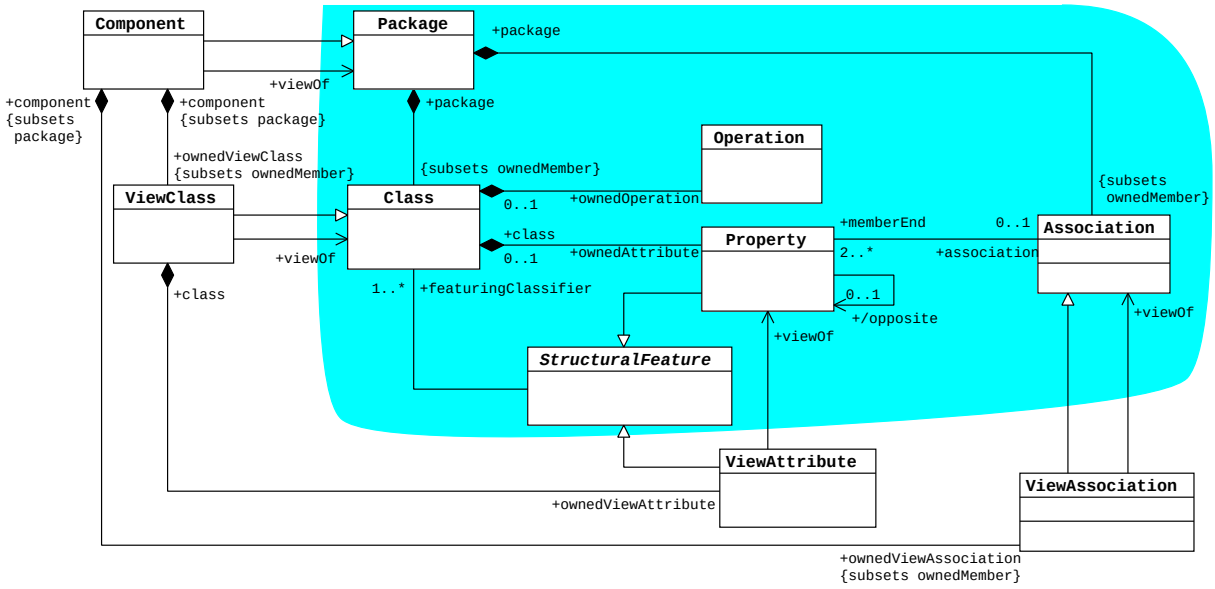

FIG. 9.1 – Le méta-modèle de composants vue par extension du méta-modèle UML2

contenus dans un paquetage est désigné par le rôle `ownedMember`.

Les **Class** peuvent contenir des attributs (sous forme de `Property`) et des opérations. Ces deux ensembles sont respectivement désignés par les rôles `ownedAttribute` et `ownedOperation`.

StructuralFeature est un concept abstrait correspondant aux éléments structurels, comme les attributs, appartenant à un classifier. On les appelle élément structurel par opposition aux éléments comportementaux comme les opérations.

Le concept de **Property** correspond donc à celui d'attribut, mais aussi à celui d'extrémité d'association (en remplacement des `AssociationEnd` de UML 1.4). Dans ce cas, il est possible de retrouver l'association à laquelle il participe par le rôle `association` et la `Property` correspondant à l'autre extrémité de l'association (dans le cas d'une association binaire) par le rôle `opposite`.

Une **Association** permet donc de relier plusieurs `Class` par l'intermédiaire de `Property`. L'`Association` désigne l'ensemble (deux ou plus) des `Property` correspondants à ses extrémités par le rôle `memberEnd`.

Concepts relatifs aux vues

Un **Component** est une spécialisation de la méta-classe UML `Package`. Son rôle est de regrouper tous les éléments nécessaires à la modélisation d'un composant vue. Ceux-ci pouvant être des `ViewClass`, des `ViewAssociation`, ainsi que tout autre élément UML pouvant être contenu dans un paquetage "standard". Il définit les rôles `ownedViewClass` pour désigner ses `ViewClass` et `ownedViewAssociation` pour désigner ses `ViewAssociation`, ceux-ci correspondent à un sous-ensemble des `ownedMember` défini par le concept `Package`.

Le rôle `viewOf` permet d'indiquer à quel paquetage est connecté le composant.

Une **ViewClass** est une spécialisation de la méta-classe UML `Class`. Une instance de celle-ci peut contenir les mêmes éléments qu'une autre `Class` ainsi que des `ViewAttribute`. Le rôle `viewOf` indique à quelle `Class` de base la `ViewClass` correspond. Le concept de `ViewClass` définit également un nouveau rôle, `ownedViewAttribute`, permettant de désigner ses attributs vue.

Un **ViewAttribute** est une nouvelle `StructuralFeature` ayant un lien racine (`viewOf`) vers une `Property`. Un `ViewAttribute` ne peut appartenir qu'à une `ViewClass`.

Une **ViewAssociation** est une spécialisation de la méta-classe UML `Association`. Elle ne peut appartenir qu'à un `Component` et désigne, grâce à son rôle `viewOf`, l'association de base à laquelle elle correspond.

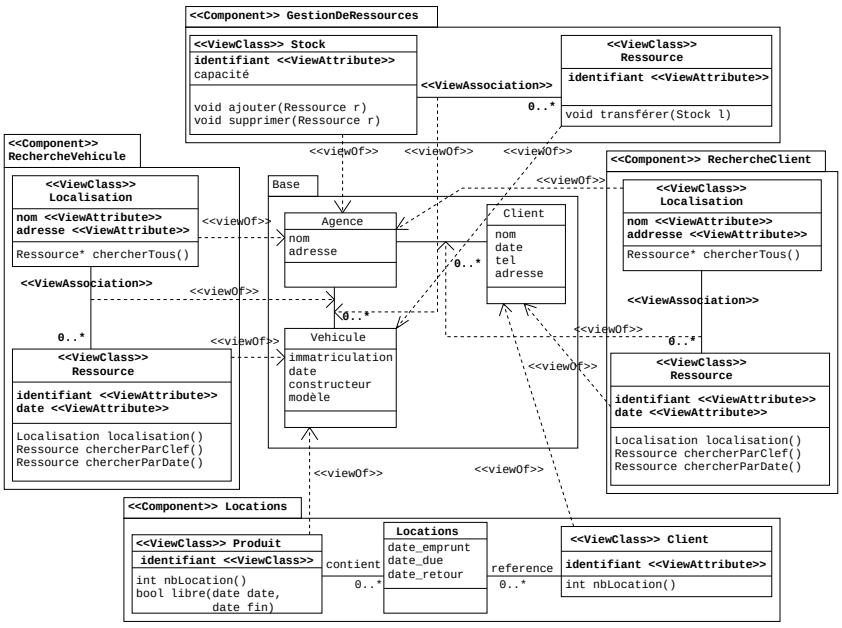

FIG. 9.2 – Utilisation de composants vue

Chacun de ces concepts est illustré figure 9.2. Les paquetages *RechercheVehicule*, *RechercheClient*, *Gestionnaire* et *Locations* correspondent au concept de `Component`. Les classes *Localisation* et *Ressource* du paquetage *RechercheVehicule* illustrent le concept de `ViewClass`. De même, les attributs *nom* et *adresse* de la classe *Localisation*, illustrent le concept de `ViewAttribute`. Enfin, le concept de `ViewAssociation` est illustré par l'association présente entre les `ViewClass` *Localisation* et *Ressource*. Les classes et associations vue sont reliées à l'élément de base auquel elles correspondent à l'aide d'une relation de dépendance stéréotypée par *viewOf*.

9.2 Contraintes

Afin de garantir la cohérence des vues, il est nécessaire de définir un ensemble de contraintes à respecter. Ces contraintes sont de deux types : les contraintes de conception et les contraintes de connexion. Les premières permettent de prévenir les erreurs pouvant survenir lors de la conception d'un modèle de base ou d'un composant vue. Les contraintes de connexion, permettent de vérifier la cohérence d'un assemblage, c'est-à-dire la vérification de la conformité du modèle fourni par le modèle de base avec le modèle requis par le composant vue.

Contraintes de conception

Les trois contraintes de conception suivantes permettent de garantir qu'un élément vue ne peut se trouver que dans un élément vue. En effet, il serait incohérent qu'un `ViewAttribute` appartienne à une classe de base. Cette propriété est vérifiée par la contrainte [1]. De même une `ViewClass` ou une `ViewAssociation` ne peuvent appartenir qu'à un `Component`. Cela est garanti respectivement par les contraintes [2] et [3]. Enfin, une `ViewAssociation` étant une vue d'une association existante, elle ne peut faire participer de nouvelles classes à cette association. Une `ViewAssociation` ne peut donc être en relation qu'avec des `ViewClass` (ceci est vérifié par la contrainte [4]). Cette dernière contrainte ne s'applique pas à une nouvelle association introduite par le composant vue, qui peut faire participer aussi bien des *ViewClass* que des *Class*.

```
[1]Le propriétaire d'un ViewAttribute doit être une ViewClass.
context ViewAttribute inv :
  self.featuringClassifier.oclIsKindOf(ViewClass);

[2]Une ViewClass ne peut apparaître que dans un Component.
context ViewClass inv : self.package.oclIsKindOf(Component);

[3]Une ViewAssociation ne peut apparaître que dans un Component.
context ViewAssociation inv : self.package.oclIsKindOf(Component);

[4]Seules des ViewClass peuvent participer à une ViewAssociation
context ViewAssociation inv : self.memberEnd->forAll
   ( p : Property | p.class.oclIsKindOf (ViewClass) );
```

Contraintes de connexion

Les contraintes de connexion permettent de vérifier que la structure formée par les éléments connectés par les liens `viewOf` est la même que celle formée par les éléments vue. Ce sont ces contraintes qui permettent de garantir que le modèle fourni correspond bien au modèle requis exprimé par les éléments vue et donc que la connexion est valide.

La première contrainte de connexion permet de vérifier que tous les `ViewAttribute` d'une `ViewClass` sont bien connectés à un attribut de la classe de base correspon-

dante. Violer cette contrainte reviendrait dans notre exemple (figure 9.2), à connecter la `ViewClass` *Localisation* à la classe *Agence* et son `ViewAttribute` *nom* à l'attribut *immatriculation* de la classe *Vehicule*, ce qui n'a aucun sens.

```
[5] context ViewClass inv :
self.ownedViewAttribute
          ->forAll ( f : ViewAttribute | f.viewOf.class = self.viewOf );
```

La contrainte suivante permet de garantir que l'attribut utilisé pour connecter un `ViewAttribute` est bien du même type.

```
[6] context ViewAttribute inv : self.viewOf.type = self.type;
```

La contrainte suivante permet de vérifier que la structure formée par une `View-Association` et les `ViewClass` qui lui sont associées, est bien la même que la structure formée par l'association et les classes auxquelles elles sont connectées. Cette vérification est réalisée en testant pour chaque `ViewClass` du composant vue, que si elle participe à la `ViewAssociation`, sa `Class` participe à l'association correspondante.

```
[7] context ViewAssociation inv :
self.component.ownedViewClass
     ->forAll( v | self.viewOf.memberEnd.class
                ->includes(v.viewOf)
        implies self.memberEnd.class
                ->includes(v));
```

La huitième contrainte vérifie que toutes les `ViewClass` sont bien connectées à une classe du système de base auquel leur `Component` est connecté.

```
[8] context Component inv :
self.ownedViewClass
     ->forAll ( v | v.viewOf.package = self.viewOf );
```

La neuvième contrainte garantit que les cardinalités entre une `ViewAssociation` et son `Association` sont respectées.

```
[9] context ViewAssociation inv :
self.memberEnd->forAll (p : Property |
   let  p2 : self.viewOf.memberEnd
       ->one (p2 : Property | p.class.viewOf = p2.class);
   p.upper = p2.upper and p.lower = p2.lower)
```

Enfin la dernière contrainte garantit que pour un composant vue donné, comme cela a été expliqué au chapitre 5, il ne peut y avoir plus d'une `ViewClass` pour une classe du système de base. Mais il peut bien entendu y avoir plusieurs `ViewClass` pour une même classe de base dans des composants vue différents.

```
[10] context Component inv :
self.ownedViewClass
     ->forAll ( v1, v2 | not v1.viewOf = v2.viewOf );
```

9.3 Profil UML

La définition de nos composants vue nécessitant une extension du méta-modèle UML, il n'est pas possible de les utiliser directement dans un modèle UML. La norme UML prévoit un mécanisme particulier, permettant de spécialiser les modèles pour un domaine ou un besoin particulier. Ce mécanisme est celui des profils [Pel02]. Un profil définit un ensemble de stéréotypes et de contraintes. Les stéréotypes sont des annotations à apposer sur des éléments de modélisation pour les spécialiser. Ils prennent la forme de chaînes de caractères entre chevrons (`<< >>`). Ces stéréotypes peuvent posséder des attributs[1] permettant d'ajouter de l'information aux éléments stéréotypés. Enfin, un profil ne serait pas complet sans l'ensemble de règles permettant de contraindre les constructions possibles afin que les modèles construits en utilisant ce profil respectent les lois du domaine.

Nous présentons ici le profil correspondant à notre extension du méta-modèle, permettant la modélisation de composants vue dans tout outil UML standard. Le tableau 9.1 présente la liste des stéréotypes et attributs du profil. La première colonne donne les stéréotypes et attributs, la seconde colonne indique à quelle méta-classe ils peuvent s'appliquer et la dernière colonne en donne une définition.

Notre profil définit cinq stéréotypes. `<<Component>>` qui s'applique à un `Package` pour représenter un composant vue. `<<ViewClass>>`, `<<ViewAttribute>>` et `<<ViewAssociation>>` qui s'appliquent respectivement aux `Class`, `Property` et `Association` pour représenter des `ViewClass`, `ViewAttribute` et `ViewAssociation`. Et `<<viewOf>>` qui s'applique à une dépendance entre une `Class` et une `ViewClass` ou à une dépendance entre un `Package` et un `Component` pour matérialiser un lien de vue. La figure 9.2 illustre l'utilisation de ce profil.

Nous proposons dans notre profil de matérialiser les liens de type `viewOf` pour les `ViewAttribute` et les `ViewAssociation` par un attribut de stéréotype ayant pour valeur le nom de l'élément de base correspondant. Dans notre exemple (figure 9.2) la valeur de l'attribut du `ViewAttribute` *identifiant* de *Produit* est donc *immatriculation*. Ce choix est motivé par le fait que, bien que cela soit défini par le méta-modèle, peu d'ateliers UML permettent de placer un lien de dépendance entre des attributs ou associations.

Les contraintes imposées par notre profil correspondent à celles définies par notre méta-modèle. Les contraintes qui s'appliquent, par exemple, à la méta-classe `ViewClass` dans notre méta-modèle, s'appliquent aux classes portant le stéréotype `<<ViewClass>>` dans notre profil.

Afin de vérifier la cohérence de nos composants vue et de notre profil, ce dernier a été outillé (à partir du méta-modèle UML 1.4) dans l'atelier de modélisation Objecteering[2]. Celui-ci permet de vérifier si un schéma UML donné est conforme à notre modèle abstrait, par la vérification des contraintes de conception et de connexion (traduites en J, le langage propriétaire d'Objecteering). Il permet également la génération automatique des spécifications IDL3 pour CCM des composants vue. Une autre projection, vers les EJB a également été réalisée pour vérifier sa faisabilité jusqu'à l'obtention d'un

[1] Ces attributs de stéréotype correspondent aux valeurs marquées de UML 1.4

[2] Le module est librement téléchargeable à l'adresse http ://www.lifl.fr/~mullera.

Stéréotype	**s'applique à**	**Définition**
`<<ViewClass>>`	`Class`	Indique que la `Class` est une vue.
`<<viewOf>>`	`Dependency`	Permet d'indiquer une dépendance entre une classe et sa vue ou entre un paquetage et un composant.
`<<ViewAttribute>>`	`Property`	Indique que l'attribut est une vue.
`<<ViewAssociation>>`	`Association`	Indique que l'association est une vue.
`<<Component>>`	`Package`	Indique que le `Package` est un composant vue.
Attributs de stéréotype	**s'applique à**	**Définition**
`viewOf`	`<<View Attribute>>`	Nom d'un `Attribut` Désigne la racine de l'élément.
`viewOf`	`<<View Association>>`	Nom d'une `Association` Désigne la racine de la `ViewAssociation`.

TAB. 9.1 – Les éléments du profil

système exécutable. Le profil et les deux projections sont détaillées dans [Mul].

Chapitre 10

Méta-modélisation des composants de modèles

Comme nous l'avons présenté dans notre état de l'art, plusieurs approches proposent l'utilisation des paquetages templates UML pour la représentation de modèles génériques. Nous avons également présenté dans la première partie les extraits du méta-modèle UML2 permettant la définition de ces paquetages templates.

Ce sont également de bons candidats pour l'expression de nos composants de modèle. Cependant, comme nous l'avons vu, ils ne permettent d'exprimer la partie variable des paquetages que par une liste d'éléments paramètres. Dans notre approche cette liste d'éléments paramètres doit impérativement former un modèle cohérent, correspondant au modèle requis de nos composants de modèle.

Afin de garantir cette propriété, nous étendons, dans la section 10.1, le méta-modèle UML 2 par une nouvelle méta-classe et lui associons un ensemble de contraintes sur la forme de ses paramètres. Nous reprenons dans la section suivante, un des exemples de composants de modèle utilisés chapitre 6 et présentons sa représentation à l'aide de paquetages templates UML présenté section 4.3.

La section 10.2 présente l'utilisation de nos composants de modèles exprimés à l'aide de paquetages templates, à l'aide de notre relation nommées *apply*. Celle-ci est définie dans cette section par extension du méta-modèle UML2. Nous proposons également dans cette section une formalisation de la définition de la relation standard *bind* à l'aide de contraintes OCL.

10.1 Formulation à l'aide de templates

La figure 10.1 rappelle notre composant de modèle pour la fonctionnalité de gestion des ressources utilisé section 6.1. Nous proposons l'utilisation des paquetages templates pour formuler nos composants de modèle en UML, tel que figure 10.2.

Tous les éléments, aussi bien ceux issus du modèle fourni que ceux issus du modèle requis, sont représentés dans le paquetage template. Les éléments issus du modèle requis sont listés dans le coin supérieur droit du paquetage. Cette liste correspond aux

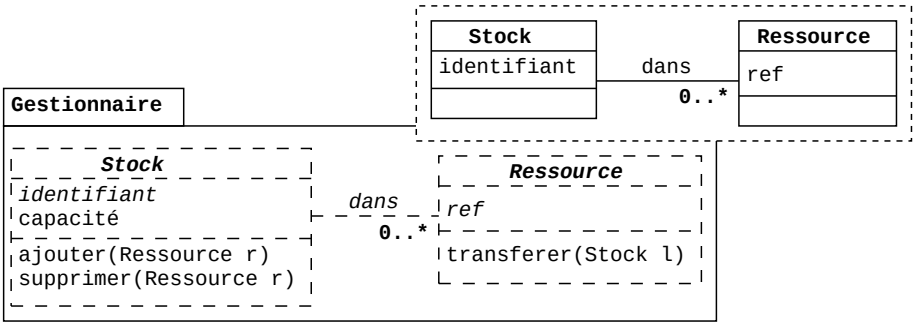

FIG. 10.1 – Composant de gestion des ressources

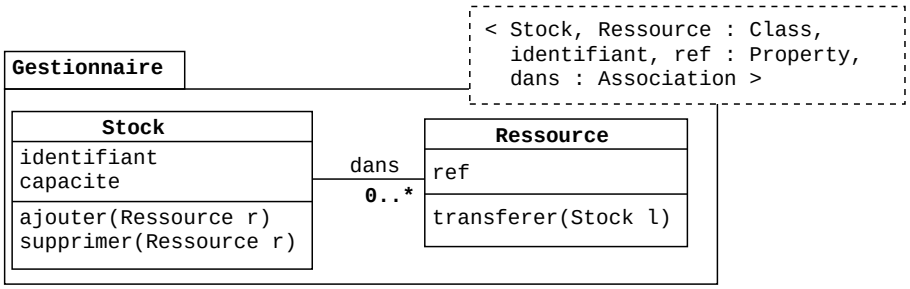

FIG. 10.2 – Template de gestion des ressources

paramètres du template.

Le paquetage template obtenu est en tous points conforme au standard. Aucune construction nouvelle n'est nécessaire. Cependant, bien que tous nos composants de modèle puissent être représentés à l'aide d'un paquetage template, à tout paquetage template ne correspond pas forcément un composant de modèle. En effet, comme défini section 6.1, la partie requise d'un composant de modèle doit être un modèle.

Un exemple de paquetage template ne répondant pas à la définition de composant de modèle est donné figure 10.3. Celui-ci est pratiquement identique à celui de la figure 10.2 mais la classe *Ressource* n'y est pas paramètre. L'élément formé par l'ensemble des paramètres n'est donc pas un modèle valide. Pour s'en persuader, considérons le composant de modèle qui devrait correspondre à ce paquetage template. Celui-ci est représenté figure 10.4.

On observe ainsi que l'élément requis ne correspond pas à un modèle consistant[1]. Afin de pouvoir garantir qu'un paquetage template correspond bien à un composant de modèle, nous présentons dans la section suivante les contraintes à respecter sur les éléments paramètres.

[1]au sens d'un diagramme de classes.

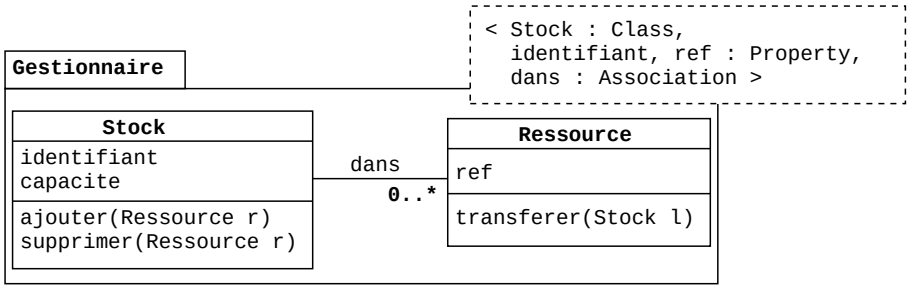

FIG. 10.3 – Template ne correspondant pas à un composant

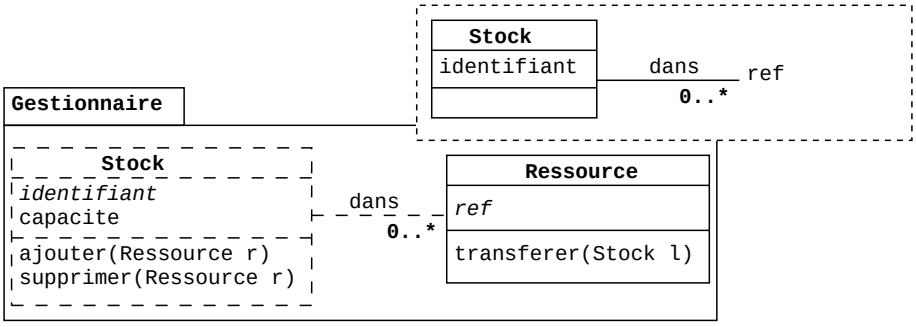

FIG. 10.4 – Composant de modèle invalide

10.1.1 Méta-modélisation

Pour limiter l'application de nos contraintes aux paquetages devant représenter des composants de modèle et non à tout paquetage, nous ajoutons au méta-modèle UML une spécialisation (`ModelComponent`) de la méta-classe `Package` (figure 10.5). Cette méta-classe n'est utilisée que pour y attacher les contraintes, elle n'ajoute aucun élément (ni attribut, ni association). Aucun élément particulier n'est donc nécessaire pour représenter ces composants de modèle[2].

Comme dans notre formalisation (cf. chapitre 7), un modèle de base est lui aussi considéré comme un `ModelComponent` mais n'ayant pas de signature.

Contraintes

Le respect du méta-modèle UML permet de garantir la validité du modèle exprimé par un paquetage (et donc par un `ModelComponent`). Les paramètres d'un template correspondent à un sous-ensemble de ce modèle. Pour vérifier que les paramètres d'un `ModelComponent` forment bien un modèle il suffit de vérifier que le sous-ensemble qu'ils

[2]Un stéréotype pourra toutefois être appliqué aux paquetages templates devant représenter des composants de modèle afin de les différencier des autres paquetages templates.

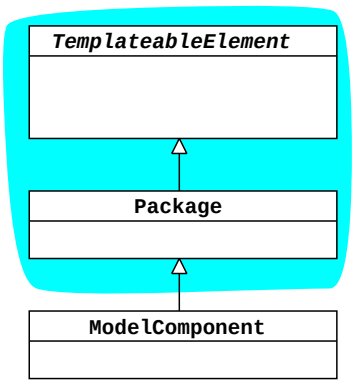

FIG. 10.5 – Méta-modèle des composants de modèle

forment est consistant, c'est-à-dire qu'il ne manque pas d'élément qui amènerait à violer les règles du méta-modèle UML. C'est ce que les deux contraintes suivantes permettent de faire en vérifiant qu'il ne peut y avoir d'attribut sans classe ou d'association sans classe à l'une de ses extrémités. À noter que le modèle peut-être valide sans être connexe. Le respect du méta-modèle UML et de ces deux contraintes permet donc de garantir que les paramètres d'un paquetage template forment bien un modèle.

La première contrainte permet de vérifier que seuls des éléments structurés peuvent être paramètres d'un composant de modèle. Il serait en effet incohérent qu'un attribut soit paramètre sans que sa classe soit elle-même paramètre.

```
[1] Si une feature est utilisée comme paramètre d'un ModelComponent, alors le
      classifier auquel elle appartient doit également être un paramètre.

context ModelComponent :
self.ownedTemplateSignature.ownedParameter.parametredElement->forAll (e |
  e.isKindOf (Feature) implies
   self.ownedTemplateSignature.ownedParameter.parametredElement->includes
    (e.featuringClassifier))
```

La seconde contrainte vérifie que lorsqu'une association est paramètre, toutes les classes qui y participent le sont également. C'est cette deuxième contrainte qui permet de détecter que le paquetage template de la figure 10.3 n'est pas un composant de modèle.

```
[2] Si une association est paramètre d'un ModelComponent, alors toutes les
     classes participant à cette association doivent être des paramètres.

context ModelComponent :
self.ownedTemplateSignature.ownedParameter.parametredElement->forAll (e |
  e.isKindOf (Association) implies
     self.ownedTemplateSignature.ownedParameter.parametredElement
       ->includesAll (e.memberEnd.class))
```

Il est ainsi possible de formuler nos composants de modèle à l'aide de paquetages templates standards UML2 et de vérifier que ces paquetages templates expriment bien un composants de modèle valide.

10.2 Utilisation de modèles paramétrés

La section précédente nous a permis de présenter l'utilisation de paquetages templates pour l'expression de nos composants de modèles. Nous nous intéressons dans ce chapitre à leur utilisation pour la construction de systèmes.

Afin de permettre l'expression de l'application d'un composant de modèle à un autre, nous proposons à la section 10.2.1 une représentation de l'opérateur *apply* sour forme d'une relation. Cette relation porte un paramétrage permettant la mise en correspondance du modèle requis par le composant de modèle à appliquer et le modèle auquel on l'applique. Cette relation est formalisée par extension du méta-modèle UML2 et par un ensemble de contraintes.

La norme UML2 propose une relation standard *bind* permettant d'exprimer le lien de trace existant entre un modèle concret et un template utilisé pour sa conception[3]. Elle permet notamment d'exprimer quels éléments dans le modèle résultant ont été utilisés pour substituer les paramètres du template. Cette relation peut être utilisée dans notre approche afin d'exprimer le lien qui existe entre un système résultant de l'application d'un composant de modèle et ce composant de modèle.

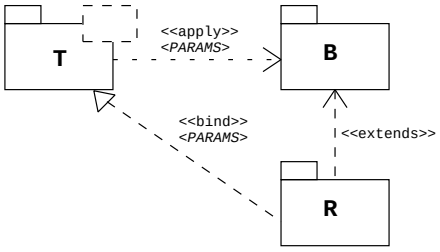

FIG. 10.6 – Rapport entre les relations *bind* et *apply*

La figure 10.6 illustre le rapport qui existe entre ces deux relations. Le paquetage T représente un paquetage template correspondant à un composant de modèle. Le paquetage B représente le modèle de base auquel on applique le composant de modèle. Le paquetage R correspond au résultat de l'application de T à B, soit le modèle de base (B) enrichi par les éléments de la fonctionnalité exprimée par le composant de modèle T.

S'il existe une relation *apply* entre le template T et le paquetage B, alors il se déduit une relation *bind* entre le template T et le paquetage résultant R.

[3] *"A binding is a relationship between a template (as supplier) and a model element generated from the template (as client)."* [UML03a]

Il existe également un rapport entre les éléments paramètres de la relation *apply* et les paramètres de la relation *bind*. Chaque paramètre de la relation *apply* met en correspondance un élément paramètre du template avec un élément du modèle de base. Chaque paramètre de la relation *bind* met en correspondance un élément paramètre du template avec un élément du modèle résultat R. Les éléments de R étant définis par extension des éléments de B, il est possible d'établir le lien entre les paramètres du *apply* et ceux du *bind*. En effet, s'il existe pour la relation *apply* une substitution entre un élément x du template et un élément y de la cible, alors il existe une substitution pour la relation *bind* entre l'élément x du template et l'élément y du résultat.

La relation *bind* correspond à un lien de trace entre le composant de modèle exprimé sous forme d'un paquetage template et le modèle résultat construit à l'aide de ce template.

10.2.1 La relation *Apply*

Afin de permettre l'utilisation du langage UML pour l'expression de systèmes par applications de composants de modèle exprimés sous forme de templates, nous proposons dans cette section une extension du méta-modèle UML 2 permettant de formuler, sous-forme d'une relation nommée *apply*, l'application d'un composant de modèle à un autre.

La première section illustre cette formulation en UML. La section suivante présente l'extension du méta-modèle et les contraintes permettant de garantir la cohérence d'une application.

10.2.1.1 Formulation des applications

La figure 10.7 illustre l'application du composant de modèle *Gestionnaire* (formulé à l'aide d'un paquetage template) au système *Base*.

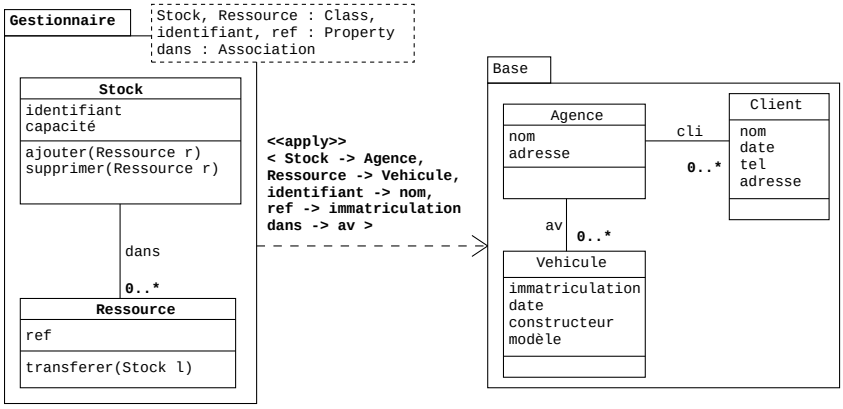

FIG. 10.7 – Application de Gestionnaire au système de location de véhicules

Celle-ci est formulée par une relation stéréotypée `<<apply>>` orientée du composant

de modèle vers le modèle auquel on souhaite l'appliquer. Elle porte un ensemble de relations de correspondances comparable à celui porté par une relation de *binding*. Cet ensemble exprime la mise en correspondance du modèle requis avec le modèle cible.

Ces relations de mise en correspondance permettent d'indiquer pour chaque élément du modèle requis, l'élément du modèle cible auquel il correspond. Pour que l'application exprimée soit cohérente, il est nécessaire que le modèle formé par l'ensemble des éléments désignés issus du modèle cible corresponde au modèle requis. Cela peut être vérifié par l'ensemble de contraintes présenté dans la section suivante.

Dans notre exemple, ces relations de mise en correspondance sont définis par les couples *Stock*/*Agence*, *Ressource*/*Vehicule*, *identifiant*/*nom*, *ref*/*immatriculation* et *dans*/*av*.

Comme notre approche et sa formalisation (cf. chapitre 7), cette représentation supporte les applications partielles. Dans ce cas, seul un sous-modèle du modèle requis est mis en correspondance avec le modèle cible. Les contraintes présentées dans la section suivante tiennent également compte de cette possibilité.

10.2.1.2 Méta-modèle

La méta-classe `TemplateApplying` permet de définir notre relation *apply*. Sa structure est comparable à celle de la méta-classe `TemplateBinding` présentée à la section 4.3. Cependant, sa sémantique étant différente, les contraintes qui s'appliquent au `TemplateBinding` (décrites dans la section suivante) ne s'appliquent pas au `TemplateApplying`. C'est pourquoi cette dernière n'est pas définie comme une spécialisation de `TemplateBinding`, mais comme une spécialisation de `DirectedRelationship`[4].

Il aurait été possible de définir une méta-classe abstraite commune aux méta-classes `TemplateBinding` et `TemplateApplying` afin de regrouper les contraintes s'appliquant à ces deux concepts, mais cette solution aurait nécessité de modifier le méta-modèle UML. Il aurait en effet été nécessaire de supprimer le lien d'héritage direct existant entre les concepts de `DirectedRelationship` et de `TemplateBinding`. Notre solution au contraire ne nécessite aucune modification du méta-modèle UML, elle ne fait que définir de nouveaux concepts.

La relation *apply* lie la signature d'un `TemplateableElement` et un `ModelComponent` par le rôle *applyTarget*. Elle est composée d'un ensemble de `TemplateParameterSubstitution` permettant de mettre en relation un élément issu du modèle source (un `TemplateParameter`) avec un élément du modèle cible (un `ParameterableElement`). Seules les méta-classes `ModelComponent` et `TemplateApplying` sont spécifiques à notre proposition, les autres éléments sont issus du méta-modèle standard UML2. Rappelons qu'un modèle de base est également considéré comme un `ModelComponent`, mais n'ayant pas de signature.

[4]Cela ne concerne pas les contraintes portant sur les `TemplateableElement` ou les `Paquetage`, qui continuent à s'appliquer sur les `ModelComponent`.

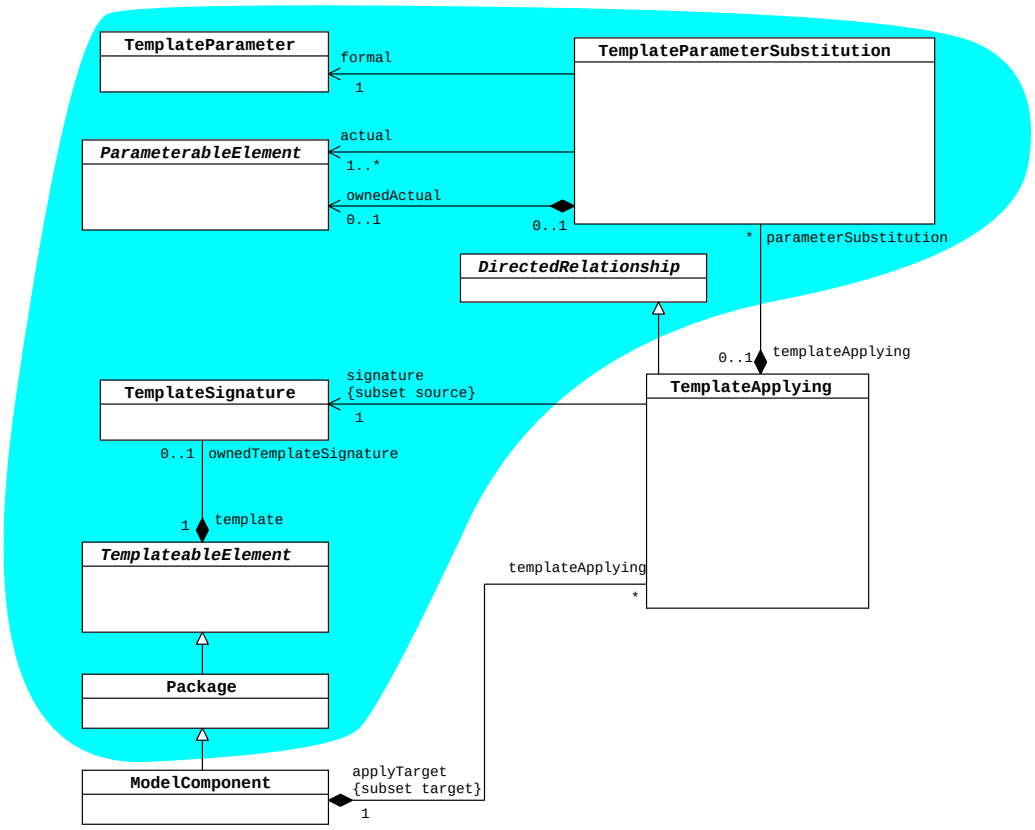

FIG. 10.8 – Méta-modèle des composants de modèles et de la relation *apply*

Contraintes

Afin de pouvoir vérifier qu'une application donnée est bien cohérente, nous avons défini un ensemble de contraintes. La première vérifie que l'origine d'un *apply* est bien un `ModelComponent`. Cette vérification est réalisée en testant si la `TemplateSignature` appartient bien à un `ModelComponent`.

```
[1] La signature d'un TemplateApplying doit appartenir à un ModelComponent.

context TemplateApplying inv :
self.signature.template.isKindOf (ModelComponent)
```

La seconde contrainte vérifie que les éléments utilisés en tant que paramètre pour appliquer un `ModelComponent` appartiennent bien au modèle cible (`applyTarget`).

```
[2] Les éléments utilisés comme paramètres effectifs d'une substitution doivent
    appartenir au modèle cible.

context TemplateBinding inv :
```

```
    self.parameterSubstitution->actual.forAll (p |
      self.applyTarget.allOwnedElement()->includes (p))
```

La troisième contrainte vérifie la compatibilité entre les méta-types des paramètres formels et effectifs. Elle utilise pour cela l'opération *isCompatibleWith* définie par le standard qui permet de réaliser cette vérification entre deux éléments de modèle.

```
[3] Tous les paramètres effectifs d'un apply doivent être compatibles avec leur
      paramètre formel.

context TemplateApplying inv :
self.parameterSubstitution->forAll (t |
   t.actual.isCompatibleWith(t.formal.parameteredElement))
```

Les contraintes [4] et [5] permettent de vérifier que les structures formées par les éléments attendus et ceux fournis lors de l'assemblage sont identiques. La contrainte [4] vérifie cette propriété au niveau de la relation contenu/contenant (*owned/owner*) des éléments. La contrainte [5] vérifie cette propriété au niveau des schémas formés par les associations.

```
[4] Si le paramètre formel d'une substitution est contenu dans un
     autre paramètre formel (param2) alors son paramètre
     actuel est contenu dans le paramètre actuel de param2.

context TemplateApplying inv :
self.parameterSubstitution->forAll (t1, t2 |
  t1.formal.ownedElement->includes (t2.formal) implies
     t1.actual.ownedElement->includes (t2.actual))

[5] Les paramètres formels correspondant aux extrémités d'une
     association formelle doivent être substitués par les
     extrémités de l'association actuelle correspondante.

context TemplateApplying inv :
self.parameterSubstitution->forAll(t1, t2 |
  let asso : t1.formal.parametredElement;
  let cla    : t2.formal.parametredElement;
  (asso.isKindOf (Association) and cla.isKindOf (Class)
     and cla.ownedAttribute.association->includes (asso))
  implies
     t2.actual.ownedAttribute.association->includes (t1.actual))
```

Enfin la dernière contrainte permet de vérifier que les cardinalités, entre une association représentant un paramètre formel et l'association donnée en tant que paramètre actuel, sont les mêmes. Cette vérification est réalisée pour chaque association paramètre, en retrouvant parmi l'ensemble des substitutions les extrémités d'associations correspondantes dans le modèle source et le modèle cible. On teste ensuite si ces extrémités d'association (sous forme de `Property`) ont les mêmes cardinalités.

```
[6] context TemplateApplying inv :
self.parameterSubstitution->select
  (asso | asso.formal.isKindOf (Association)).memberEnd->forAll
    (p : Property |
      let  p2 : asso.actual.memberEnd
       ->one (p2 : Property | self.parameterSubstitution
           ->exist (s | s.formal = p.class and s.actual = p2.class);
    p.upper = p2.upper and p.lower = p2.lower)
```

Grâce à ces extensions et ces contraintes, il est ainsi possible d'exprimer la construction de systèmes par application de composants de modèle à l'aide du langage UML et de vérifier la cohérence de ces assemblages. Cette vérification pourra, par exemple, être effectuée à l'aide d'outils au fur et à mesure des applications ou avant la réalisation du système.

Nos extensions restent compatibles avec le standard et permettent notamment l'utilisation de la relation *bind* afin de garder un lien entre le modèle d'un système et les composants de modèles utilisés pour sa construction. C'est cette relation que nous présentons dans la section suivante.

10.2.2 La relation *Bind*

La relation *bind* est la seule relation standard proposée pour l'utilisation des templates. Celle-ci n'étant pas formellement définie, et afin d'en comprendre la sémantique, nous avons proposé dans [CCMV04] de la formaliser à l'aide de contraintes OCL. C'est cette formalisation que nous présentons dans cette section.

La norme UML2 définit la relation de *binding* comme une **copie** de tous les éléments du template dans le *boundElement* modulo substitution des paramètres.

"The presence of a TemplateBinding relationship implies the same semantics as if the contents of the template owning the target template signature were copied into the bound element, substituting any elements exposed as formal template parameters by the corresponding elements specified as actual parameters in this binding." [UML2].

Nous proposons ici de formaliser cette définition à l'aide d'un ensemble de contraintes exprimées à l'aide du langage OCL. Les figures 10.9 et 10.10 rappellent les extraits du méta-modèle UML2 présentés à la section 4.3 sur lesquels s'appliquent ces contraintes.

Les contraintes définies dans le méta-modèle permettent de vérifier la validité du nombre et des types des éléments utilisés comme paramètres du template. Ces paramètres représentent les éléments nécessaires pour la construction du *boundElement*, ils doivent donc être des éléments de ce *boundElement*. La contrainte suivante permet de vérifier cette propriété.

```
[1] Les éléments utilisés comme paramètres effectifs d'une substitution doivent
     appartenir au boundElement.

context TemplateBinding inv :
    self.parameterSubstitution.actual->forAll (p |
       self.bounedElement.allOwnedElement()->includes (p))
```

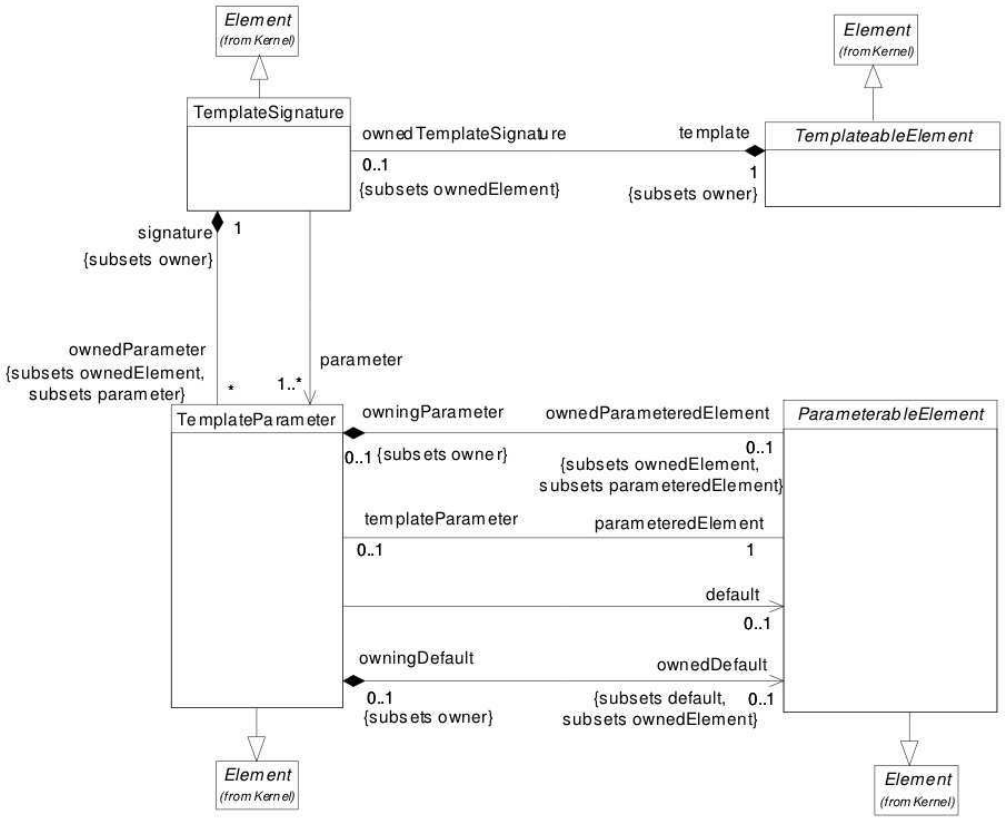

FIG. 10.9 – Le méta-modèle des templates [UML05b].

La deuxième contrainte permet de vérifier qu'il existe bien pour chaque élément du template, paramètre ou non, un élément correspondant dans le *boundElement*. Cette correspondance est établie à l'aide de la méta-opération *isBound* définie sur la méta-classe racine Element de UML 2.

```
[2] Pour tout élément du template, il doit exister un élément correspondant
     dans le boundElement.

context TemplateBinding inv :
self.signature.template.ownedElement->forAll (template_element |
        self.boundElement.ownedElement->exists (b |
                        b.isBound (template_element, self)))
```

La méta-opération *isBound* permet de vérifier qu'un élément *e* correspond bien à l'élément *te* donné en paramètre par rapport à un binding particulier (également donné en paramètre). L'élément *e* doit appartenir au *boundElement* de la relation *binding* et *te* au template.

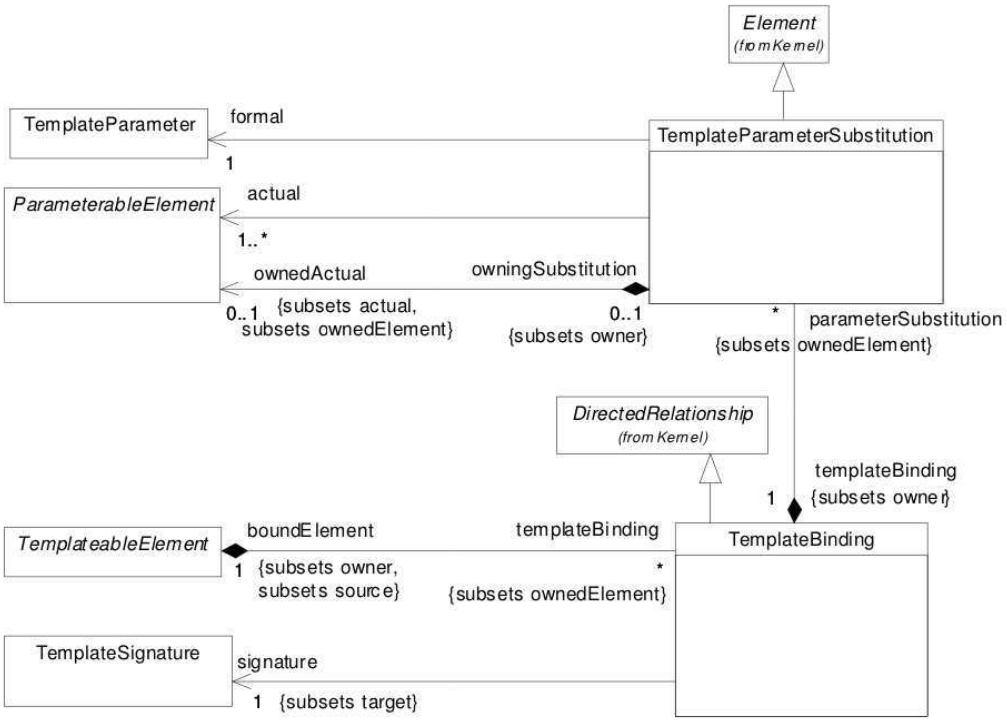

FIG. 10.10 – Le méta-modèle du passage de paramètres [UML05b].

Cette correspondance peut être une substitution du binding ou une correspondance par copie. L'opération *isBound* s'applique, grâce à l'opérateur *or*, dans les deux cas. Dans le cas d'un binding partiel, les paramètres du template non substitués sont pris en compte dans les éléments copiés.

Dans le cas où le *boundElement* aurait été construit par application d'un composant de modèle, cette contrainte permet de garantir que tous les éléments définis par le composant de modèle ont bien été ajoutés au système.

```
[3] Un élément est lié à un autre élément par une relation de binding s'il a
     le même nom et le même type ou s'il existe une substitution entre ces
     deux éléments dans la relation de binding, et s'il vérifie la relation
     bindOwnedElement.

context Element::isBound (te : Element, binding : TemplateBinding) : Boolean
body : (binding.parameterSubstitution->exists (p | p.formal = te
                            and p.actual->includes (self)) or
       (self.oclIsKindOf (NamedElement) implies
          self.oclAsType (NamedElement).name =
              te.oclAsType (NamedElement).name
          and self.oclIsTypeOf (te.oclType))
       and self.bindOwnedElement (te, binding))
```

Dans les deux cas (copie ou substitution), il est aussi nécessaire de vérifier qu'il y a bien correspondance entre les éléments contenus dans *te* et ceux contenus dans *e*. Cette correspondance est vérifiée à l'aide de la méta-opération *bindOwnedElement*. Cette méta-opération vérifie qu'il existe, pour chaque élément de *te* un élément correspondant dans *e*, au sens de l'opération *isBound*.

```
[4] Un élément e relie (bind) les éléments contenus dans un élément d'un
     template s'il existe pour chacun d'eux un élément de e qui lui est lié.

context Element::bindOwnedElement (te : Element,
            binding : TemplateBinding) : Boolean
body : te.ownedElement->forAll (te_owned |
                self->ownedElement->exist (self_owned |
                          self_owned.isBound (te_owned, binding)))
```

La contrainte [2] et les opérations [3] et [4] permettent de vérifier que la structure formée par les éléments du template est bien conservée dans le *boundElement*. Le type d'erreur de la figure 10.11 où *att1* est attribut de la classe *ClassA* dans le template et qu'il n'existe pas d'attribut correspondant dans la classe *X* du *boundElement*, est ainsi impossible.

Elle permettent de garantir dans le cas d'un *boundElement* construit par application d'un composant de modèle que la structure des éléments ajoutés a bien été conservée.

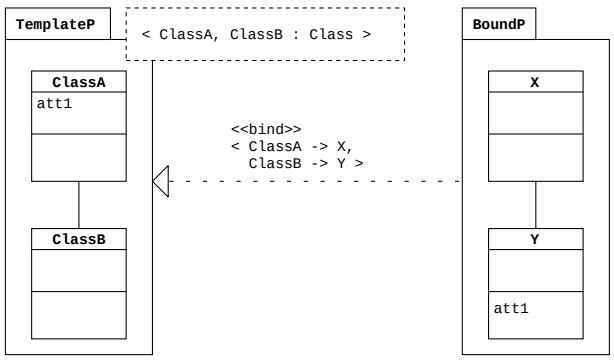

FIG. 10.11 – Erreur de "déplacement" d'un attribut

Ces opérations permettent, grâce à un appel récursif et polymorphe de vérifier la correspondance d'éléments complexes. L'opération *isBound* doit néanmoins être enrichie pour vérifier un certain nombre de propriétés spécifiques aux opérations et associations. Il est en effet nécessaire de vérifier que, pour chaque opération du *boundElement* liée à une opération du template, leurs signatures sont identiques, au paramétrage près. La contrainte [5] vérifie cette propriété et interdit, par exemple, les erreurs comme celle illustrée figure 10.12, où le paramètre *t* de l'opération *foo* de *ClassA* du *boundElement* devrait être typé *Y*. En effet, le type de *t* dans le template (X) est remplacé dans le *boundElement* par *Y*.

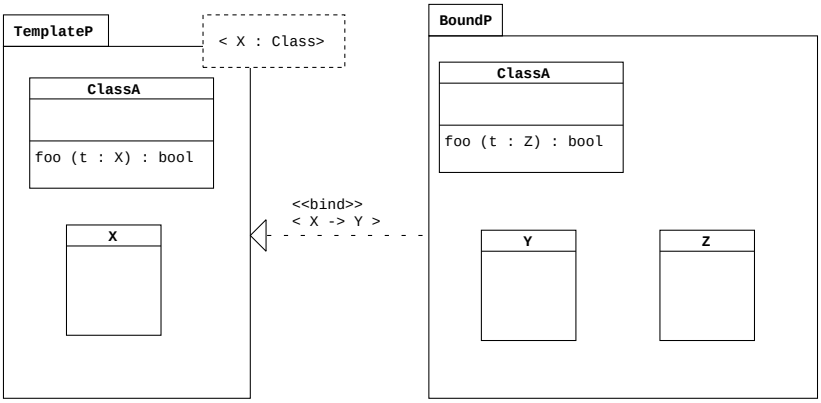

FIG. 10.12 – Erreur sur la signature d'une opération

```
[5] La signature d'une opération du boundElement doit être
     la même que celle de l'opération du template à laquelle elle est liée,
     modulo substitutions.

context Operation::isBound (te : Element, binding : TemplateBinding) : Boolean
body : self.oclAsType (Element).isBound (te, binding) and
    self.formalParameter->size() =
       te.oclAsType (Operation).formalParameter->size() and
          Sequence {1..self.formalParameter->size ()}
            ->forAll (i : Integer | self.formalParameter->at (i).type.isBound
                        (te.oclAsType (Operation).formalParameter->at (i).type,
                           binding) and
          self.returnResult->size() =
             te.oclAsType (Operation).returnResult->size()
      and Sequence {1..self.returnResult->size ()}
            ->forAll (i : Integer | self.returnResult->at (i).type.isBound
                      (te.oclAsType (Operation).returnResult->at (i), binding))
```

Pour prendre en compte le cas particulier des associations et prévenir les erreurs du type de celle présentée figure 10.13, où *asso* est "déplacée" d'une association entre *ClassA* et *ClassB* en une association entre *ClassB* et *ClassC*, la contrainte suivante est définie. Elle vérifie que, si une *Property* est liée à une association dans le template, la *Property* associée dans le *boundElement* est liée à l'association correspondante.

```
[6] Si une property du template est extrémité d'une association A alors
     la property liée doit être extrémité de l'association liée à A.

context Property::isBound (te : Element, binding : TemplateBinding) : Boolean
body : self.oclAsType (Element).isBound (te, binding) and
            te.oclAsType (Property).association->notEmpty () implies
              self.association.isBound (te.oclAsType (Property).association,
                binding)
```

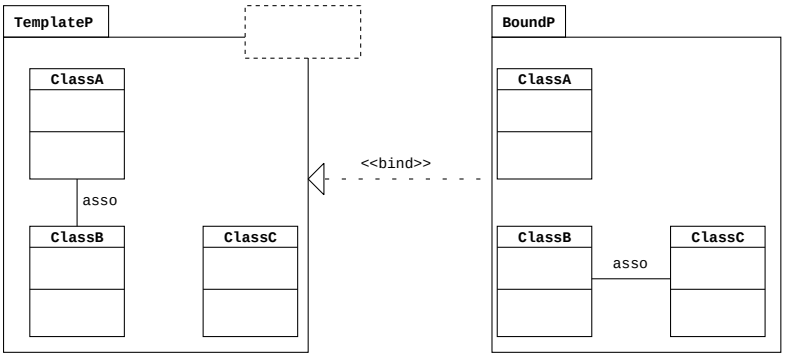

FIG. 10.13 – Erreur de "déplacement" d'une association

Pour la comprendre, il est nécessaire d'étudier l'extrait du méta-modèle UML2 donné à la figure 10.14. Comme nous l'avons déjà évoqué, dans la version 2 d'UML, il n'existe plus de méta-classe *AssociationEnd*, celle-ci est remplacée par *Property* jouant le rôle de *ownedAttribute* pour une classe et reliée à une association. C'est pourquoi la contrainte [6] est définie sur la méta-classe *Property* pour vérifier le respect des connections entre les classes et associations.

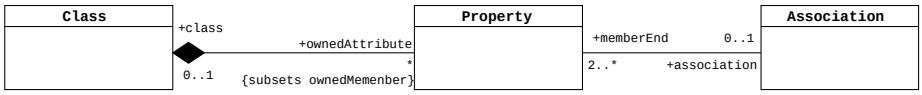

FIG. 10.14 – Méta-modèle Classe-Propriété-Association

Grâce à l'appel récursif et polymorphe de l'opération *isBound* définie sur la méta-classe *Element* et utilisée dans l'opération *isBound* de la méta-classe *Property*, les mêmes contraintes s'appliquent que les éléments soient ou non paramètres. L'erreur illustrée figure 10.15 est ainsi également détectée.

Enfin, pour qu'une association du *boundElement* corresponde à une association du template, elles doivent avoir la même arité. Ce qui interdit par exemple les constructions comme celle de la figure 10.16. Cette propriété est vérifiée à l'aide de la contrainte [7].

```
[7] L'arité d'une association liée doit être la même que celle de
     l'association correspondante du template.

context Association::isBound (te : Element, binding : TemplateBinding)
                    : Boolean
body : self.oclAsType (Element).isBound (te, binding) and
            self.memberEnd->size ()
                      = te.oclAsType (Association).memberEnd->size ()
```

Cet ensemble de contraintes sur la relation de *binding* permet de garantir l'existence dans le *boundElement* de tous les éléments du template avec la même structure,

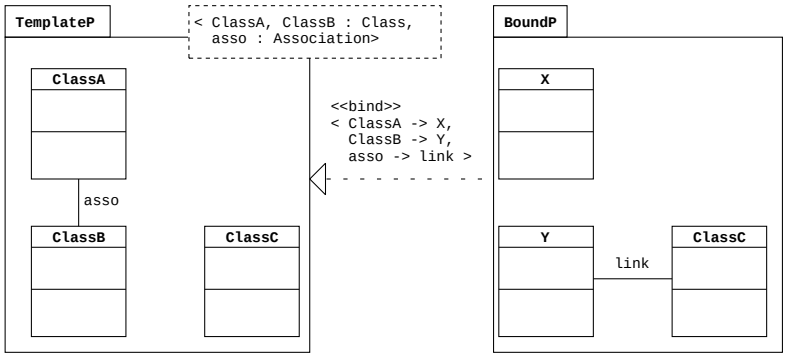

FIG. 10.15 – Erreur de "déplacement" d'une association paramètre

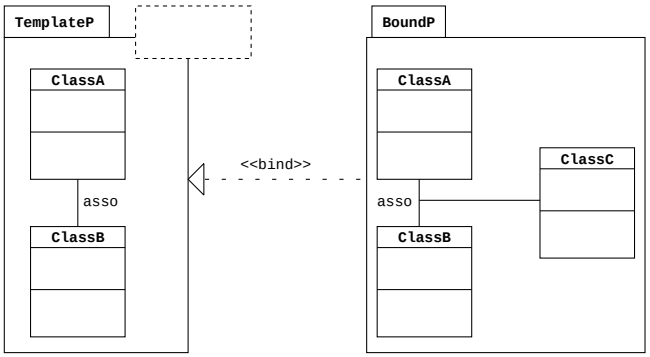

FIG. 10.16 – Erreur de non respect de l'arité

conformément à sa définition. Cette formalisation rend ainsi possible l'automatisation de la vérification de modèles utilisant la relation *bind*.

La sémantique de notre application (définie au chapitre 7), impose également de retrouver, avec la même structure, les éléments du composant de modèle dans le modèle résultat. Ce modèle résultat est donc bien en relation de *binding* avec le composant de modèle utilisé pour sa construction (cf. figure 10.6).

Les relations *apply* et *bind* sont compatibles et complémentaires. La première permet de concevoir un système par assemblage d'un ensemble de composants de modèle, la seconde permet d'exprimer la trace entre le modèle du système et les composants de modèle utilisés pour sa construction.

L'ensemble des modes d'expression du modèle résultat présentés au chapitre 8 vérifient cette propriété. Que les éléments issus des composants de modèles soient fusionnés ou représentés explicitement, ils se retrouvent tous dans le modèle résultat conformément à la sémantique de la relation *apply*. On peut donc tracer une relation *bind* entre ces modèles résultats et chaque composant de modèle utilisé pour sa construction. Le paramétrage de ces relations de *binding* étant déduit du paramétrage

de la relation *apply* correspondante.

La relation *bind* permet donc, non seulement de garder la trace des composants de modèle utilisés pour la construction d'un système, mais également de retrouver le paramétrage utilisé.

Nous nous intéressons dans la partie suivante à la mise en œuvre de systèmes construits par assemblage de composants de modèle, à partir de ces modèles résultats. Il existe, en effet, plusieurs stratégies donnant au système des propriétés différentes.

Cinquième partie

Mises en œuvre

Le travail réalisé dans les parties précédentes propose une approche pour l'expression de composants de modèle génériques et de leurs assemblages. Il est ainsi possible de construire et de garantir la cohérence de modèles de systèmes complexes par composition de ces artefacts de modèle réutilisables. Ces artefacts peuvent être stockés dans des bibliothèques de composants afin de faciliter leur réutilisation. Comme nous l'avons vu au chapitre 8, notre approche permet également d'exprimer le modèle du système résultant selon différentes représentations.

Nous proposons d'étudier dans cette partie les mises en œuvre possibles de systèmes construits par assemblage d'un ensemble de composants de modèle, et les propriétés qui en découlent. Nous avons identifié plusieurs stratégies de mise en œuvre à partir des modes d'expression du modèle résultat d'un assemblage.

La figure 10.17, illustre différentes chaînes de production d'un système à partir d'un assemblage de composants de modèle jusqu'à une plate-forme d'exécution. À partir du modèle d'assemblage (niveau 1), il est possible d'obtenir un modèle du système (niveau 2) selon les modes fusion, traces ou vues.

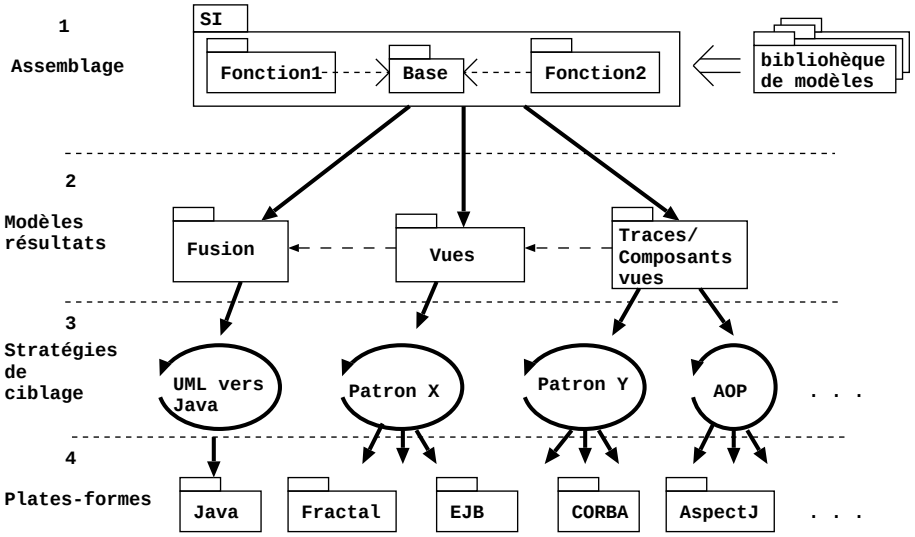

FIG. 10.17 – Chaînes de production

Comme nous l'avons vu au chapitre 8, il est possible de passer d'un modèle de traces à un modèle de vues, et d'un modèle de vues à un modèle fusionné (illustré sur la figure par les flèches en pointillés). Nous verrons par la suite (au chapitre 12) comment il est possible d'assouplir le choix entre le tout fusionné et le tout éclaté.

En effet, de nombreux critères peuvent influencer le choix de fusionner ou non la fonctionnalité définie par un composant de modèle. La granularité du composant, pouvant aller de l'expression d'une simple fonction de calcul à une fonctionnalité métier complexe, en est un exemple. Les préoccupations techniques sont également des critères pouvant influencer ce choix. Il peut, en effet, être intéressant de permettre la distribution des composants sur différents sites d'exécution selon, par exemple, des préoccupations

de trafic réseau ou de sécurité. La préservation de la structuration rend également possible la gestion de droits d'accès en associant, par exemple, à chaque vue la liste des utilisateurs pouvant y accéder directement. La réutilisation d'éléments binaires, ou l'évolution d'un système en fonctionnement est également un critère déterminant. Il est, en effet, difficilement envisageable de fusionner une nouvelle fonctionnalité avec l'existant dans ces conditions.

Toutes ces formulations, fusionnées ou non, restent dans l'espace des modèles. Nous présentons dans cette partie un ensemble de stratégies de mise en œuvre de ces modèles (niveau 3) permettant de préserver leurs propriétés jusqu'à l'exploitation du système. Ces stratégies s'inscrivent dans une démarche MDA, permettant d'obtenir une implantation sur une plate-forme particulière (PSM) à partir d'un modèle métier (PIM). Elles correspondent, dans la vision OMG, à des transformations PIM vers PSM.

La stratégie de mise en œuvre la plus directe consiste à générer du code dans un langage objet traditionnel à partir du modèle fusionné. En effet, la représentation du modèle résultat sous forme fusionnée correspond à un modèle classes/associations classique pour lequel il existe de nombreuses traductions et outils permettant la génération du code correspondant vers différents langages (illustré sur notre figure par la stratégie *UML vers Java*). Cette solution se caractérise par sa simplicité et l'efficacité du système obtenu. Il n'y a pas de sur-coût à l'exploitation du système dû à sa construction par assemblage. En contrepartie, sa structuration est perdue, ce qui ne facilite pas sa maintenance et son évolution.

La représentation du système à l'aide de vues permet de préserver sa structuration dans le modèle résultat. Cependant, les langages ou plates-formes actuelles ne supportent pas une mise en œuvre directe des vues. Nous proposons donc au chapitre 11, un ensemble de patrons de conception permettant le support des vues sur une technologie objet traditionnelle, que nous appelons mise en œuvre éclatée. Nous proposons également dans ce chapitre l'intégration du patron adaptateur pour préserver le caractère générique des composants vues au niveau de la plate-forme.

Une autre stratégie possible pour la mise en œuvre du système construit par assemblage est basée sur l'utilisation d'une plate-forme de programmation par aspects. S. Clarke décrit dans [CW02] une mise en œuvre de Theme/UML sur la plate-forme AspectJ [Asp] qui peut être adaptée à nos composants vues (illustrée par la stratégie *AOP*).

Ces différentes stratégies possèdent des avantages différents : simplicité et efficacité pour l'approche fusionnée, structuration, traçabilité et dynamicité pour les mises en œuvre éclatées. Il est donc nécessaire de faire un choix lors de la mise en œuvre du système. Cependant, il peut être intéressant de ne pas faire ce choix pour le système dans sa globalité, mais au niveau de chaque fonctionnalité, voire au niveau de chaque entité, pour, par exemple privilégier l'efficacité sur une partie critique du système et la dynamicité pour des parties étant amenées à évoluer régulièrement. Nous avons étudié cette possibilité, à l'aide d'une technique de transformation paramétrée par annotations suggérée par l'OMG dans le cadre du MDA [MM03], que nous présentons au chapitre 12.

Le chapitre 13 présente un ensemble de réalisations utilisant nos patrons de mise en

œuvre éclatée. Chacune de ces réalisations utilise une plate-forme différente, EJB[Mic02], Fractal[BCS04] et CORBA[COR]. Enfin nous y présentons un prototype d'atelier de modélisation supportant l'ensemble de notre proposition. Celui-ci permet la réalisation de composants de modèle et de leur assemblage pour la conception de systèmes, la production de modèles résultat et la génération de code vers différentes plates-formes.

Chapitre 11

Patrons de conception

La préservation des différentes vues d'un système jusqu'à sa mise en œuvre garantit une structuration et une traçabilité facilitant ainsi son exploitation, sa maintenance et son évolution. Nous proposons dans ce chapitre un ensemble de patrons pour répondre à ces besoins. Nous présenterons, au chapitre 13 l'application de ces patrons à différentes plates-formes technologiques.

La première section propose un patron de conception permettant la réalisation éclatée d'une entité par un ensemble de fragments. Chacun de ces fragments peut être issu de la structuration utilisée pour concevoir le système et ces entités. Il permet ainsi la mise en œuvre de systèmes tout en conservant leur structuration en vues.

Les fragments (et donc les vues) réalisés à l'aide de ce patron sont spécifiques au système pour lequel ils sont conçus. On perd ainsi la dimension générique de nos composants de modèle, ce qui ne permet pas la réutilisation facile de ces fragments pour la réalisation d'autres systèmes. C'est pourquoi, nous proposons section 11.2 de combiner ce patron de conception éclatée avec le patron adaptateur [GHJ$^+$95] afin de permettre la réalisation d'éléments binaires réutilisables. On obtient ainsi de véritables composants binaires, dont l'assemblage peut être conçu à un niveau modèle.

L'exploitation de systèmes conçus en représentation éclatée nécessite des opérations de gestion des fragments et entités spécifiques à cette représentation (initialisation, suppression,...). Ces opérations, si elles doivent être effectuées au niveau de chaque fragment, peuvent être fastidieuses et sources d'erreurs. Nous proposons section 11.3 une autre extension du patron de représentation éclatée, permettant la gestion des fragments au niveau de chaque vue.

Afin d'uniformiser au niveau mise en œuvre l'application des vues, et notamment de permettre l'application d'une vue à une autre, nous proposons dans la section 11.4 une troisième extension de notre patron.

11.1 Patron de représentation éclatée

Nous décrivons ici une adaptation de notre patron présenté dans [CCMV03] permettant la conception de vues sous la forme de représentation éclatée[1].

Dans les systèmes à objets traditionnels, une entité est représentée par une instance, cette instance étant entièrement décrite par sa classe. Il n'est donc pas possible d'enrichir une entité, par une vue par exemple, de façon dynamique. De plus, même s'il est possible de structurer les entités au niveau de leur classe (à l'aide du mécanisme d'héritage par exemple) il n'est pas aisé de préserver cette structuration au niveau des instances.

Le but de notre patron est de répondre au problème de représentation éclatée d'entités adaptée à la structuration fonctionnelle de systèmes [BD96]. Il permet de préserver la structuration des entités au niveau instance et autorise ainsi l'ajout et le retrait dynamique de propriétés et de fonctionnalités à ces instances. Pour cela, le patron permet la réalisation d'une seule et même entité par un ensemble de fragments tout en gérant leur coréférence indépendamment d'une technologie de mise en œuvre particulière.

Celui-ci peut se décomposer en deux parties, la première permet la gestion des fragments d'une entité, la seconde permet la gestion des associations partagées (*View-Association*).

11.1.1 Représentation éclatée des entités

Chaque entité est réalisée par un fragment de base et un ensemble de fragments-vue. Le fragment de base implante les opérations et contient les attributs communs aux différentes fonctions du système. Chaque fragment de vue implante les opérations et contient les attributs spécifiques à une fonction particulière du système.

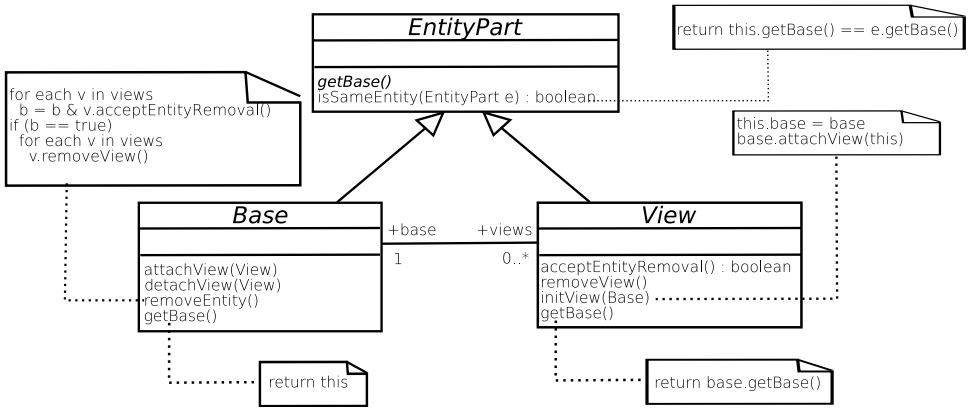

FIG. 11.1 – Patron de représentation éclatée

[1]La modification par rapport à la version originale concerne la gestion de l'identité conceptuelle décrite plus loin.

La figure 11.1 illustre la structure de ce patron. La classe *EntityPart* définit les éléments communs aux fragments de base et fragments de vue pour la gestion de l'identité conceptuelle d'une entité. L'opération *isSameEntity* permet de déterminer si deux fragments appartiennent à la même entité.

La classe *Base* définit les éléments communs aux fragments de base. Un fragment de base est lié à l'ensemble de ses fragments de vue (par le rôle *views*). Il fournit les opérations pour lui ajouter un fragment de vue (*attachView*), pour lui détacher un fragment de vue (*detachView*) et pour supprimer l'entité par la suppression de l'ensemble des fragments la constituant (*removeEntity*). Enfin, la classe *View* définit les éléments communs à l'ensemble des fragments de vue. Chaque fragment de vue est lié à son fragment de base (par le rôle *base*). Il fournit une opération pour initialiser le fragment de vue (*initView*), c'est-à-dire l'attacher à son fragment de base. Il fournit également une opération pour supprimer le fragment de vue de l'entité (*removeView*), et une opération utile pour le protocole de suppression d'entité (*acceptEntityRemoval*).

En effet, dans une représentation éclatée, la suppression d'une entité nécessite la suppression de tous ses fragments. La suppression d'un fragment-vue doit être faite après avoir vérifié qu'elle n'introduit pas d'incohérence pour les vues associées. Pour prévenir ce type d'erreur, le patron propose un protocole de suppression entre les fragments-vue et le fragment de base. Celui-ci consiste à vérifier, avant la suppression effective d'une entité, que tous ses fragments de vue peuvent être supprimés. Cela est vérifié en interrogeant chaque fragment vue à l'aide de l'opération *acceptEntityRemoval*. L'entité n'est réellement supprimée (le fragment de base et l'ensemble de ses fragments de vue) que si tous ses fragments de vue acceptent la suppression. Tout ce mécanisme est réalisé par l'opération *removeEntity* du fragment de base.

11.1.2 Vues et associations partagées

Le patron propose également un mécanisme pour le support des associations partagées par plusieurs vues. En effet, comme indiqué précédemment (cf. chapitre 5), une vue association ne correspond pas à une nouvelle association, mais à une vue sur une association du schéma de base. Pour permettre leur mise en œuvre, les associations sont matérialisées par des méthodes d'accès [CCD00a], illustrées dans la figure 11.2 par les méthodes *getEntityB* et *setEntityB* de la classe de base *EntityABase*. Les associations-vue sont également traduites en méthodes d'accès délégant leurs appels vers les méthodes d'accès de l'association à laquelle elles sont liées. Cette délégation est illustrée figure 11.2 par les méthodes *getEntityB* et *setEntityB* de la classe de vue *EntityAView*. Les opérations de manipulation d'associations de base retournant des fragments de base, il est nécessaire de retrouver les fragments de vue correspondants dans les opérations de manipulation d'association vue. Le patron propose de réaliser cette opération à l'aide de la référence sur le fragment de base partagée par tous les fragments d'une même entité (obtenue grâce à l'opération *getBase*). Ceci est également illustré figure 11.2 par l'appel à l'opération *findEntity*.

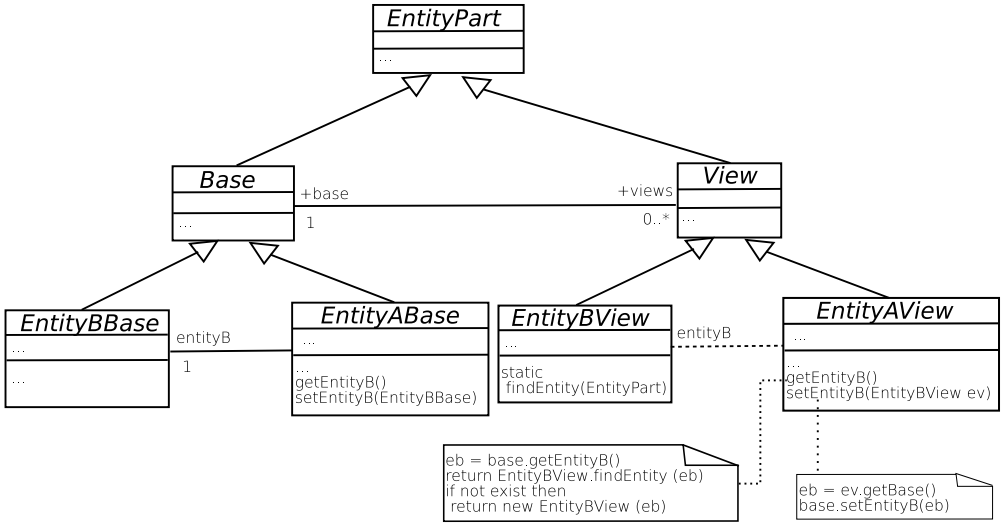

FIG. 11.2 – Gestion des vues associations

Application du patron au modèle de composant vue

La figure 11.4 présente l'utilisation de ce patron de vue pour la mise en œuvre du composant *Recherche* pour les vues *RechercheVehicule* et *RechercheClient* de la figure 11.3.

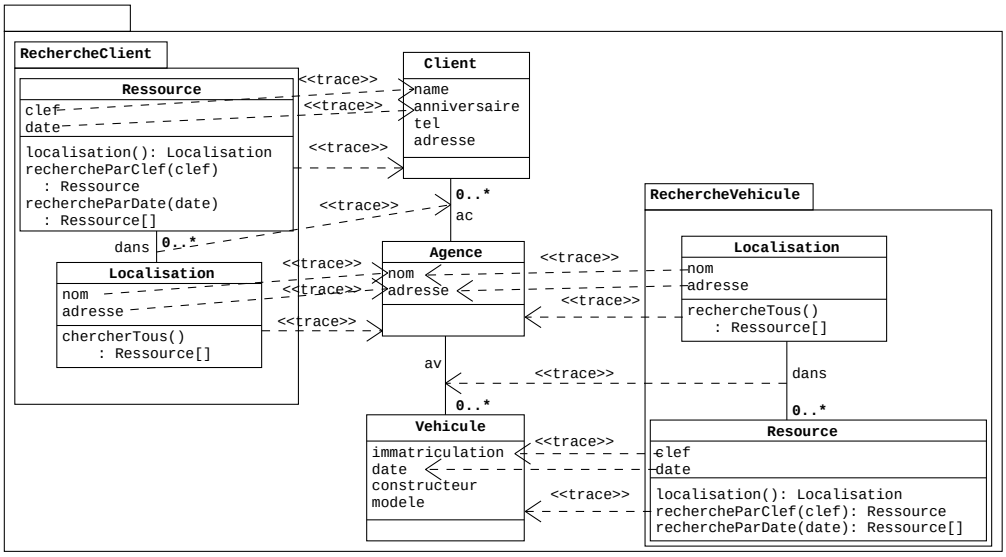

FIG. 11.3 – Fonctions de recherche des clients et des véhicules

Dans cette mise en œuvre, chaque fragment de base (*Client*, *Agence*, *Vehicule*)

hérite de la classe *Base* du patron, et chaque fragment de vue (*VehiculeViewRessource*, *ClientViewRessource*, et *AgenceViewLocalisation*) hérite de la classe *View*. Dans cet exemple, *VehiculeViewRessource* est définie comme une vue de l'entité de base *Vehicule*, et les fragments de vues *AgenceViewLocalisation* comme vues de l'entité *Agence*.

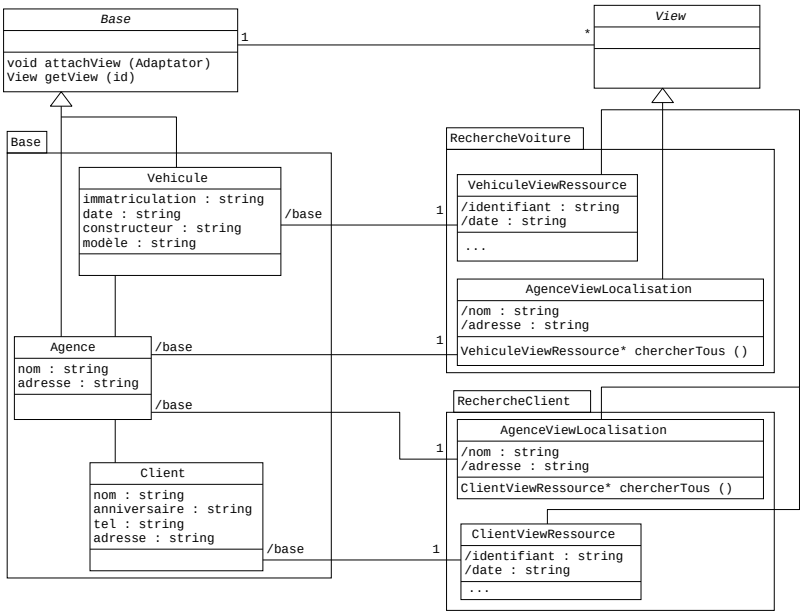

FIG. 11.4 – Utilisation du patron de représentation éclatée pour la mise en œuvre de deux applications du composant *Recherche* (RechercheVehicule et RechercheClient)

11.2 Introduction du patron adaptateur

Le patron de représentation éclatée permet de préserver la traçabilité des vues qui peuvent être ajoutées et retirées dynamiquement du système. Mais celles-ci sont spécifiques au schéma de base et aux entités auxquelles elles sont appliquées. Il est en effet nécessaire de réaliser la vue pour chaque application du composant. Ceci est illustré par la figure 11.4. Le même composant vue *Recherche* est réalisé par deux paquetages différents, l'un pour l'application *RechercheVehicule* l'autre pour l'application *RechercheClient*.

Afin de préserver la généricité des composants au niveau implantation et ainsi de faire de nos composants de modèle des composants binaires, nous proposons d'introduire l'utilisation du patron adaptateur. Nous nous basons sur le patron adaptateur d'objets de [GHJ+95] pour permettre la réalisation de classes devant coopérer sans connaître leur interface a priori. Cette extension à été décrite dans [CCMV05].

Ce patron, illustré figure 11.5, permet la réalisation de la classe *Client* par rapport à l'interface *Target*. La coopération de cette classe avec la classe *Adaptatee* est ensuite

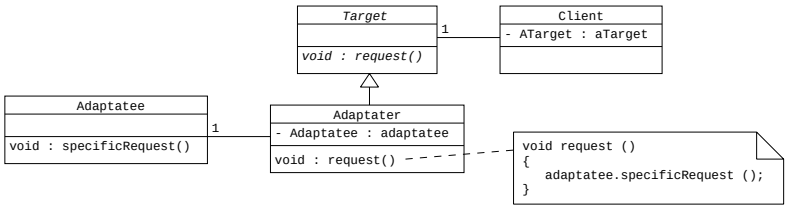

FIG. 11.5 – Le patron adaptateur [GHJ+95]

rendue possible grâce à une implantation (*Adaptater*) de cette interface qui traduit les appels vers les opérations spécifiques de l'*Adaptatee*.

La figure 11.6 présente l'introduction dans notre exemple d'adaptateurs entre les fragments de base et les fragments de vue. Les éléments du patron se retrouvent par exemple avec la classe *Ressource* qui joue le rôle de *Client*, *RessourceAdaptator* qui joue le rôle de l'interface *Target*, *RessourceVehiculeAdaptator* jouant le rôle de l'adaptateur et enfin la classe *Vehicule* en tant qu'*Adaptatee*.

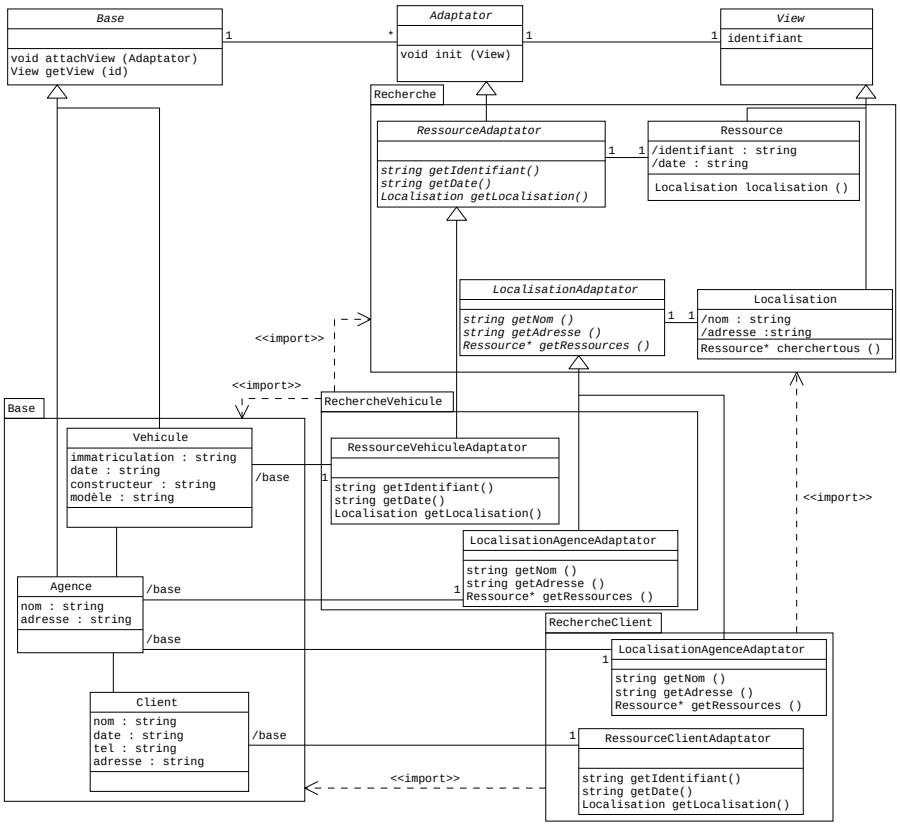

FIG. 11.6 – Introduction du patron Adaptateur

L'adaptateur permet ainsi de concevoir les fragments de vue indépendamment d'un fragment de base particulier. Un adaptateur abstrait est défini pour chacun des fragments vues, il représente l'interface requise de ce fragment. Ainsi, dans l'exemple, le fragment de vue *Ressource* est réalisé par rapport à l'adaptateur abstrait *RessourceAdaptator* définissant toutes les opérations requises du fragment. Il n'y a ainsi aucune convention de nommage à respecter entre les fragments de base et les fragments de vue.

La dimension générique introduit le besoin d'identifiant de vue (qui peut être le nom de la vue, comme *RechercheVehicule* et *RechercheClient*) partagé par tous les fragments appartenant à une même vue. Cet identifiant de vue permet notamment l'utilisation d'un même composant (comme dans notre exemple *Recherche* pour la réalisation des deux vues précédentes) plusieurs fois dans le même système.

Lors de l'assemblage, un adaptateur concret doit être réalisé pour chaque connexion entre un fragment de base et un fragment de vue. Celui-ci doit "traduire" les appels du fragment de vue en fonction du paramétrage de la connexion. Dans notre exemple, c'est *RessourceVehiculeAdaptator* qui joue ce rôle d'adaptateur concret en implantant, par des appels aux opérations du fragment de base *Vehicule*, les opérations définies par l'adaptateur abstrait *RessourceAdaptator*. Ce mécanisme est illustré par la figure 11.7. Le fragment de vue *view* réalise ses appels sur l'adaptateur (*adpt*) celui-ci effectuant les appels correspondants sur le fragment de base auquel il est lié (*base*).

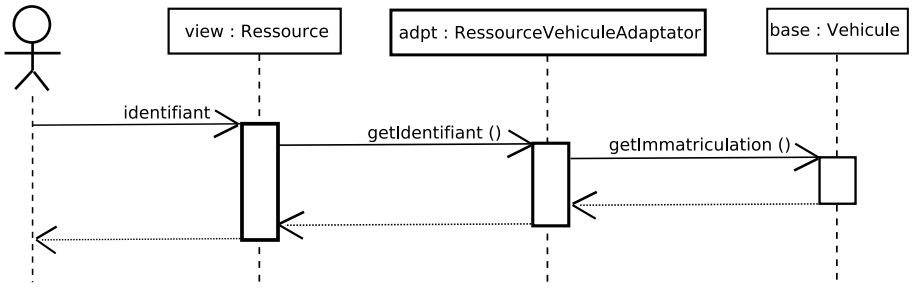

FIG. 11.7 – Délégation du fragment de vue.

La mise en œuvre des `ViewAssociation` suit le même protocole que celui présenté section 11.1 : les appels aux opérations de manipulation d'une `ViewAssociation` sont délégués aux opérations de l'association de base correspondante. Mais ce schéma d'adaptation fait intervenir deux nouveaux intermédiaires dans ce protocole, les adaptateurs des fragments vues aux extrémités de la `ViewAssociation`. De plus, les fragments de vue n'ayant pas connaissance des fragments de base, l'opération permettant de retrouver les fragments de vue correspondants après le parcours d'une association (celle-ci retournant comme pour la version précédente des fragments de base) doit être réalisée par les adaptateurs.

Nous présentons dans le paragraphe suivant la mise en œuvre, illustrée figure 11.8, du parcours de la `ViewAssociation` entre les fragments de vue *Localisation* et *Ressource* dans la vue *RechercheVehicule*.

Lors de l'appel de l'opération *chercherTous* sur un fragment de vue *Localisation*,

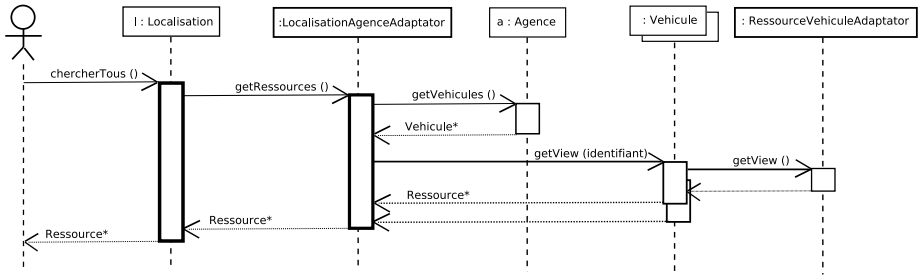

FIG. 11.8 – Parcours d'une vue association par adaptateur.

celui-ci effectue un appel à l'opération *getRessources* de son adaptateur (*LocalisationAgenceAdaptator*). Cette opération, définie dans l'interface requise de *Localisation* (*LocalisationAdaptator*) correspond au parcours de la `ViewAssociation`. L'adaptateur (*LocalisationAgenceAdaptator*) déclenche donc l'opération *getVehicules* sur le fragment de base *Agence* auquel il est lié. Cette opération correspond au parcours de l'association de base correspondante.

L'adaptateur obtient alors (suivant la cardinalité) un fragment ou une collection de fragments de base, ici une collection de fragments *Vehicule*. Il reste maintenant à obtenir le ou les fragments de vue correspondants. Ceci est toujours réalisé par l'adaptateur en s'adressant à chacun des fragments de base correspondants par l'opération *getView(identifiant)*, où *identifiant* correspond à l'identifiant commun à tous les fragments d'une même vue (ici "RechercheVehicule"). Chaque fragment de base retourne ensuite le fragment de vue ayant cet identifiant en s'adressant à l'adaptateur correspondant (ici *RessourceVehiculeAdaptator*).

L'adaptateur obtient ainsi les fragments de vue correspondants au parcours de la `ViewAssociation` (ici une collection de *Ressource*) qu'il peut retourner au fragment de vue à l'origine du parcours (ici le fragment de *Localisation*). Ce fragment de vue peut à son tour effectuer des traitements propres à la vue.

Comme l'illustre la figure 11.6, l'approche par adaptateur permet de ne réaliser qu'une seule implantation générique d'un composant (ici le composant *Recherche*) et d'utiliser un ensemble d'adaptateurs spécifiques pour la réalisation d'une vue (ici *RechercheVehicule* et *RechercheClient*).

Leurs traitements étant systématiques, les adaptateurs peuvent facilement être générés à partir de la connexion entre un composant de modèle et un schéma base. En effet, les traitements contenus dans les adaptateurs ne sont que des opérations de conversion, directe entre son fragment de base et son fragment de vue, ou par le mécanisme des `ViewAssociation` décrit plus haut. Les traitements spécifiques aux vues, sont eux réalisés dans les fragments de vues eux-mêmes.

Dans le cas de l'utilisation de plusieurs vues sur un même fragment de base, chaque fragment de vue dispose de son propre adaptateur, il n'y a donc aucune interaction directe entre deux fragments de vue d'une même entité. Dans ce modèle, les vues s'appliquent uniquement aux bases. Il n'est pas possible d'appliquer une vue à une

autre vue.

La combinaison des patrons de vue et adaptateur permet donc la mise en œuvre de nos composants vues, sous forme d'un ensemble de fragments génériques pouvant être appliqués par adaptation à un ensemble de fragments de base répondant à une structure requise. Mais l'instanciation et la connexion de chaque fragment de vue pour chaque instance des fragments de base restent à la charge de l'utilisateur de la vue. Nous présentons dans la section suivante un mécanisme basé sur une notion de gestionnaire permettant l'automatisation de cette tâche.

11.3 Gestion des fragments de vues

Afin de permettre l'application globale d'une vue à une base, c'est-à-dire afin d'attacher à chaque fragment de base le fragment de vue correspondant, il est nécessaire de pouvoir considérer l'ensemble des fragments d'une même vue comme un tout. Pour ajouter cette possibilité, nous proposons d'introduire pour chacune d'elles (et pour la base) une entité représentant la vue dans sa globalité. Au même titre que les fragments de vue considérés jusqu'ici sont des fragments d'entité, les entités de gestion de vue introduites ici sont des fragments du système d'information. Nous utilisons donc le même patron de représentation éclatée pour définir ces entités de gestion de vue. Cette extension est également décrite dans [CCMV05].

Afin de définir les relations entre une entité de gestion de vue et les fragments de vue qu'elle doit gérer, nous nous appuyons sur le patron composite [GHJ+95] présenté figure 11.9.

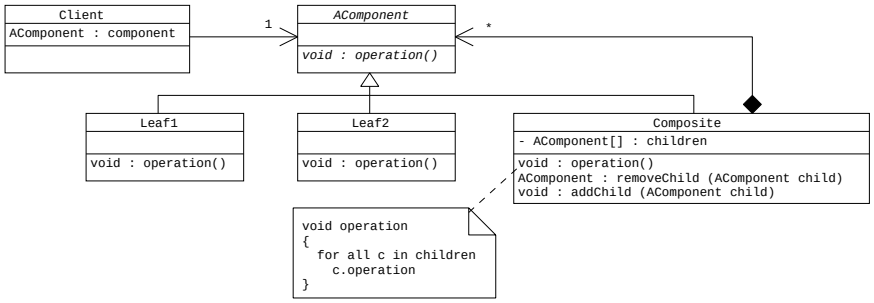

FIG. 11.9 – Le patron composite [GHJ+95].

Ce patron permet à un client de s'adresser, au travers d'une interface *AComponent*, aussi bien à un élément simple comme *Leaf1* ou *Leaf2* qu'à un élément composite comme *Composite*. L'appel de *operation* sur un composite se concrétise par l'appel de *operation* sur tous les éléments de celui-ci.

La figure 11.10 présente l'utilisation de ce patron pour la définition d'un gestionnaire pour notre composant vue *Recherche* (b) ainsi que pour notre base (a). Ces gestionnaires représentent respectivement la vue *Recherche* et la base dans leur ensemble.

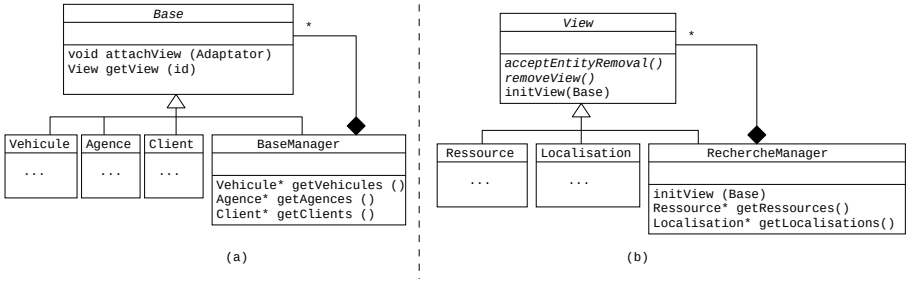

FIG. 11.10 – Application du patron composite pour la gestion d'une vue.

Nous avons proposé dans la section précédente l'utilisation du patron adaptateur afin de rendre génériques les fragments de vue. Nous proposons ici d'utiliser le même schéma d'adaptation afin de rendre les entités de gestion de vue indépendantes de toute base. La figure 11.11 présente l'adaptation de l'entité de gestion de vue *RechercheManager* à l'entité de gestion de base *BaseManager*.

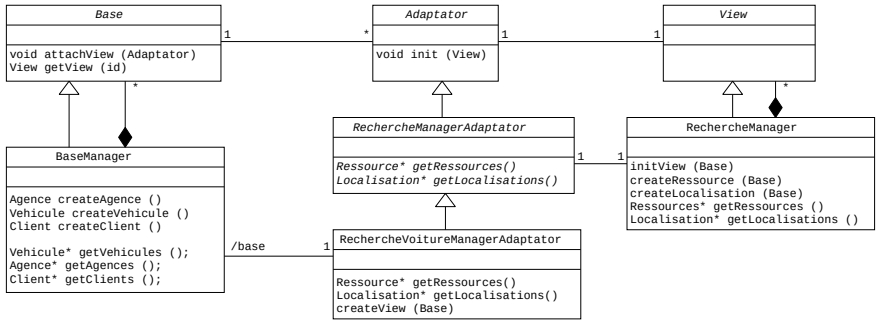

FIG. 11.11 – Adaptation de l'entité de gestion de vue RechercheManager à la base.

L'utilisateur peut ensuite appliquer la vue à la base grâce à l'opération *initView* d'une entité de gestion de vue *RechercheManager* avec en paramètre une entité de type *BaseManager*. Le rôle de cette opération est la création et l'initialisation pour chaque instance de fragment de base, de l'instance du fragment de vue correspondant si il existe. La figure 11.12 illustre ce mécanisme. L'utilisateur déclenche l'initialisation de la vue par l'appel de l'opération *initView* sur l'entité de gestion de vue, en lui donnant en paramètre l'entité de gestion de base à laquelle la vue doit être liée. Celle-ci demande à son adaptateur la création des fragments de vue pour chacun des types de fragments de base qu'elle doit gérer (ici *Localisation* pour *Agence* et *Ressource* pour *Vehicule*).

L'adaptateur obtient alors l'ensemble des instances de ce type auprès du gestionnaire de base (*BaseManager*). Il va ensuite créer pour chacun d'eux l'adaptateur et le fragment de vue adéquat. Chaque instance de fragment vue va, dès sa création, s'enregistrer (*init(this)*) auprès de son adaptateur. Chaque adaptateur est ensuite attaché (*attachView(adpt)*) à l'instance de fragment de base correspondante.

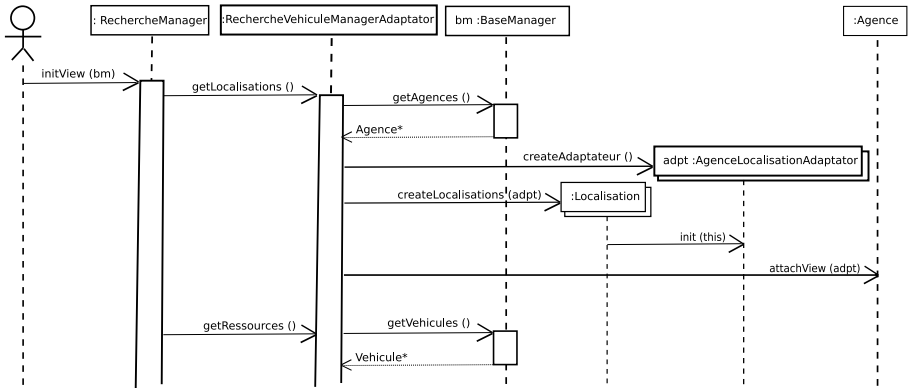

FIG. 11.12 – Attachement d'une vue.

Ce mécanisme permet donc la création automatique de tous les fragments de vue nécessaires lors de l'initialisation de la vue. Afin de permettre également la création automatique des fragments de vue lors de la création d'un fragment de base, nous proposons l'utilisation du patron observateur entre l'entité gestion de base et les entités gestion de vue. Ainsi, lors de la création d'un fragment de base, réalisé par une opération adressée à l'entité gestion de base, toutes les entités gestion de vue liées en sont notifiées et déclenchent si nécessaire la création du fragment de vue correspondant.

11.4 Extension du patron pour le support d'applications uniformes de vues

Le patron de représentation éclatée, tel qu'il est défini ne permet l'application d'un fragment de vue qu'à un fragment de base et ne permet donc que l'application de vues directement sur une base. Cependant, afin de supporter l'ajout de fonctionnalités à un système existant ou si l'on souhaite préserver la structuration introduite lors de la conception du système, il est nécessaire de pouvoir appliquer une vue à une autre (cf. section 6.2.3).

Nous proposons donc dans cette section une adaptation du patron afin de permettre une uniformisation de la composition de vues. L'idée est de permettre l'attachement des fragments de vue à tout autre fragment, que ce dernier soit de base ou de vue. L'architecture de cette adaptation est présentée figure 11.13.

Grâce à cette architecture, tout fragment de vue peut également être considéré comme un fragment cible. Il est ainsi possible d'associer un fragment de vue à un autre. Cependant, grâce à la cardinalité sur le rôle *target*, ce patron garantit qu'un fragment de vue doit obligatoirement être rattaché à un (et un seul) autre fragment. Elle garantit donc également, par transitivité, qu'il y a un et un seul fragment de base pour un fragment vue (et donc pour une entité).

Les opérations *getBase* restent inchangées et fonctionnent toujours dans cette archi-

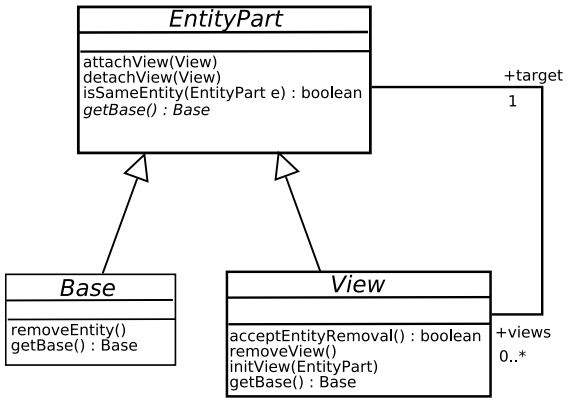

FIG. 11.13 – Adaptation du patron pour une application uniforme des vues.

tecture. En effet, l'opération *getBase* d'un fragment de vue retourne le résultat de cette même opération sur le fragment qui lui est affecté comme cible (*target*). Si ce fragment est lui même un fragment de vue, l'appel est récursif. Seule l'opération *getBase* sur un fragment de base retourne une référence sur lui même. Dans tous les cas, l'opération *getBase* retourne donc une référence sur le fragment de base (forcément unique) de l'entité. L'opération *isSameEntity* (qui est définie à l'aide de l'opération *getBase*) est également inchangée.

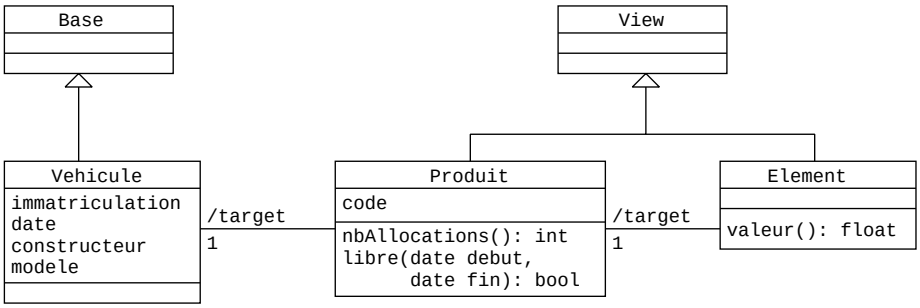

FIG. 11.14 – Connexion d'un fragment de vue à une autre.

L'utilisation de cette extension est illustrée figure 11.14 pour la chaîne d'applications *Comptage-Allocation-Base* (cf. section 6.2.3). Le fragment de vue *Produit* (*Allocation*) est lié au fragment de base *Vehicule* afin de lui ajouter les opérations nécessaires à la fonctionnalité de gestion des locations. Le fragment de vue *Element* (*Comptage*) est ensuite lié à ce fragment de vue *Produit* afin de lui ajouter une opération de calcul de valeur en fonction de ses locations (*nbAllocations*).

Cette extension permet ainsi un ajout uniforme des fonctionnalités sur la base ou sur d'autres vues.

La combinaison de cet ensemble de patrons permet la réalisation de composants

binaires génériques pour la mise en œuvre de vues d'un système. Les tâches de gestion de cette mise en œuvre particulière étant prises en charge par les composants binaires eux-mêmes.

Le chapitre 13 présente un ensemble de réalisations de systèmes sur différentes plates-formes utilisant ces patrons de conception.

Il est donc possible de mettre en œuvre un système conçu par assemblage, sous une forme entièrement fusionnée ou, grâce à nos patrons de conception, de préserver la structuration introduite par chaque composant de modèle. Mais, comme nous l'avons évoqué, il peut être intéressant d'assouplir la stratégie de projection en permettant de fusionner certains composants, tout en conservant la structuration introduite par d'autres. C'est ce que nous proposons de faire dans le chapitre suivant.

Chapitre 12

Transformations paramétrées

Grâce aux patrons présentés dans le chapitre précédent, il est possible de choisir pour la mise en œuvre d'un système entre une approche préservant les différentes vues et une approche fusionnée.

Une approche préservant les vues permet de conserver jusqu'à l'exécution, la structuration du système introduite lors de sa conception. Cependant, contrairement à l'approche fusionnée, cette approche introduit un sur-coût à l'exécution dû aux mécanismes de manipulation des différents fragments qui composent une entité. De plus, la structuration introduite par l'utilisation de composants de modèle lors de la phase de conception, n'est pas toujours significative à l'exploitation. Certains composants peuvent en effet être utilisés pour ajouter une fonctionnalité au système sans pour autant nécessiter d'en faire une vue à l'exécution. Du fait, notamment, que notre approche permet l'expression de composants de modèle de granularité très différente.

D'autres critères peuvent également intervenir dans le choix de la fusion ou non d'un composant : distribution, sécurité, droits d'accès, réutilisation d'existant,... Il parait donc intéressant de laisser au concepteur le choix de l'approche pour la mise en œuvre de son système, non pas à un niveau global, mais au niveau de chaque composant, voire de chaque entité.

Nous avons proposé dans [BCGM04] un mécanisme de transformations paramétrées suggéré par l'OMG dans le cadre du MDA [MM03]. Nous présentons dans ce chapitre ce mécanisme et étudions comment il peut être appliqué dans notre approche pour donner au concepteur le moyen d'adapter et d'hybrider les modes de mise en œuvre, fusionnée ou éclatée, pour des parties du système à différents niveaux de granularité.

12.1 Approche par annotations

Contrairement à l'approche "classique" [Bez01, Lem98] consistant à spécifier la transformation entre le méta-modèle source et le méta-modèle cible, l'approche que nous avons étudiée consiste à ajouter un ensemble d'annotations (ou marques) au modèle source et à spécifier la transformation par rapport à ce marquage. Ces deux approches sont résumées par la figure 12.1.

L'OMG (dans [MM03]), ne donne pas plus de précisions de ce que peut être une transformation à l'aide d'annotations. Nous l'avons donc expérimenté (dans le cadre du projet RNTL ACCORD [GLAM$^+$04]) et avons défini un processus de production de modèles PSM à partir de modèles PIM basé sur ce type de transformation.

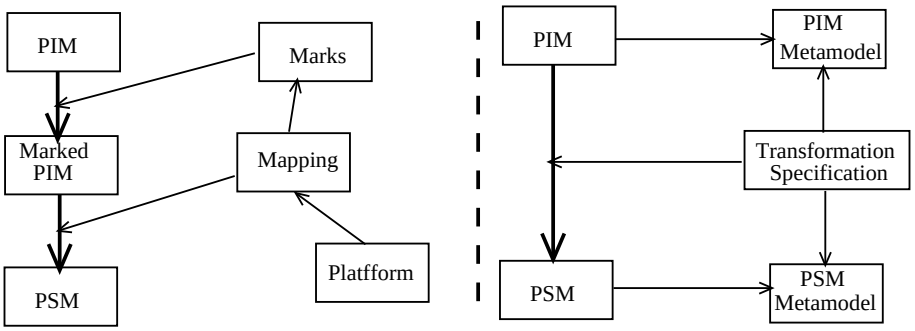

FIG. 12.1 – Les approches MDA de transformation de modèles [MM03].

Une différence fondamentale entre une approche utilisant les annotations par rapport à une approche utilisant les méta-modèles, est la possibilité, pour l'utilisateur, de guider la transformation. En effet, dans l'approche basée sur les méta-modèles, les règles sont écrites indépendamment du modèle auquel elles seront appliquées. Il n'est donc pas possible de guider la transformation pour un modèle particulier. Notre approche par annotations, au contraire, est conçue pour permettre à l'utilisateur de contrôler le processus de transformation en fonction de ses besoins.

L'approche par annotations utilise un modèle intermédiaire correspondant à une copie du modèle de départ auquel a été ajouté un ensemble d'annotations. Dans notre processus, l'utilisateur peut intervenir sur ce modèle en modifiant s'il le souhaite ces annotations. Enfin, le dernier niveau consiste à transformer ce modèle en fonction des annotations. C'est donc par l'intermédiaire des annotations que l'utilisateur sélectionne les règles de transformations à appliquer. Concrètement, lorsque plusieurs stratégies de transformation sont possibles pour un élément donné, une annotation différente est associée à chacune d'elle.

La figure 12.2 illustre notre architecture pour la transformation de modèles métiers (indépendant d'une plate-forme particulière) vers un modèle de mise en œuvre spécifique à une plate-forme particulière. Cette architecture repose sur une décomposition en quatre niveaux de modèles et trois transformations.

Le *Niveau 1* est le niveau PIM indépendant de toute cible PSM. Il doit être possible à ce niveau de vérifier la cohérence de ce modèle avant de poursuivre le processus.

La première transformation consiste a copier le modèle PIM et à l'enrichir par des concepts nécessaires pour un PSM particulier ou une famille de PSM nécessitant les mêmes concepts.

Au *Niveau 2*, le concepteur peut modifier le modèle PIM pour spécifier tous les concepts PIM nécessaires à la transformation PSM. À ce niveau également, il doit être

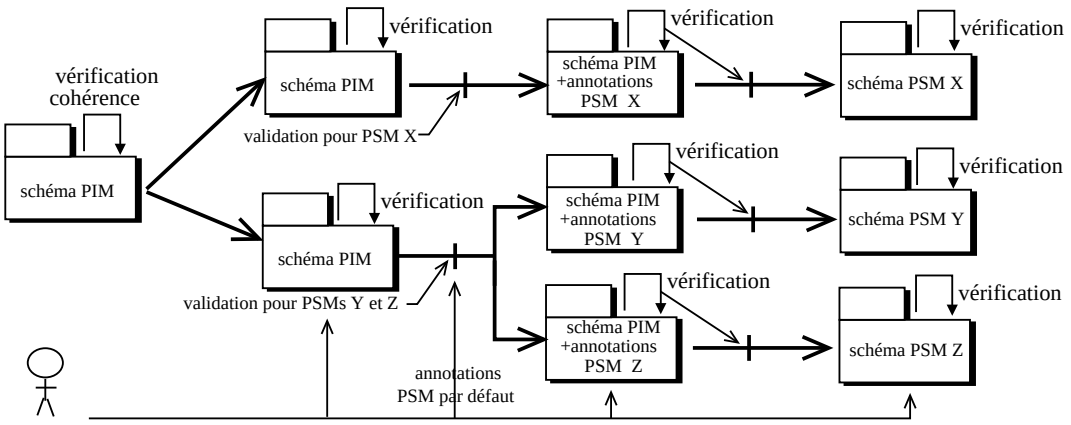

FIG. 12.2 – Architecture unifiée de modèles pour la transformation PIM-PSM

possible de vérifier la validité du modèle obtenu.

La seconde transformation consiste au passage vers un modèle PIM doté d'annotations spécifiques à un PSM. La transformation est paramétrée par des annotations par défaut. Ce mécanisme d'annotation par défaut permet de spécifier une stratégie de transformation par défaut pour l'ensemble du système. Ainsi le concepteur ne doit modifier les annotations que pour les éléments où cette stratégie par défaut ne lui convient pas. Ce mécanisme facilite l'utilisation de notre processus, en évitant de devoir annoter à la main tous les éléments du modèle où il est possible de faire un choix de projection.

Le *Niveau 3* correspond au modèle intermédiaire de choix. À ce niveau, la tâche du concepteur se résume donc uniquement à modifier les annotations là où il souhaite appliquer une autre stratégie de projection que celle par défaut. Seules les annotations sont modifiables à ce niveau.

La troisième transformation est la génération du PSM qui ne peut s'effectuer qu'après vérification du modèle. Celui-ci est valide lorsque les annotations introduites sont cohérentes entre elles et lorsque ces annotations permettent de sélectionner une et une seule règle de projection parmi les différentes alternatives.

Enfin, le *Niveau 4* est le modèle conforme au modèle PSM, le concepteur peut modifier, adapter son modèle résultat.

12.2 Flexibilité de mise en œuvre d'un assemblage

Cette architecture peut être utilisée pour la mise en œuvre d'un système conçu par assemblage afin d'apporter ce besoin de flexibilité sur le choix de mise en œuvre de chaque composant.

Dans ce cas, le premier niveau correspond au modèle d'assemblage du système pour lequel il est possible de vérifier la cohérence. La première transformation correspond quant à elle, à la génération d'un modèle résultat (cf. chapitre 8) à partir de ce modèle

d'assemblage[1].

Le niveau 2 correspond donc à un modèle résultat, à l'aide de traces ou de vues. Les modifications possibles à ce niveau sont, par exemple, le renommage de certains éléments en prévision de leur fusion.

La transformation suivante permet d'annoter ce modèle résultat (niveau 3). Nous utilisons ici ce mécanisme pour permettre au concepteur de choisir pour chaque composant s'il doit être fusionné ou non. Cette possibilité est illustrée figure 12.3. Dans cet exemple, les points de choix (matérialisés ici par les stéréotypes `<<merge>>` et `<<view>>`) indiquent que le concepteur a choisi de fusionner *Vue1* avec la base et de réaliser *Vue2* comme une vue.

La figure 12.4 illustre le modèle (niveau 4) résultant de ces choix. Le composant *Vue1* a disparu et tous les éléments qu'il définissait sont fusionnés dans les éléments du paquetage de base. Au contraire, le composant *Vue2* prend la forme d'une vue. On obtient ainsi un modèle résultat d'un assemblage, au même titre que les modèles résultats présentés au chapitre 8, mais pour lequel des choix de représentation différents ont été faits pour chaque composant. Cette étape correspond à la dernière transformation de notre architecture.

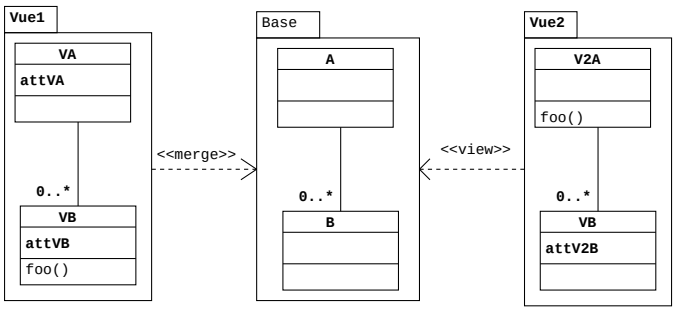

FIG. 12.3 – Modèle intermédiaire de choix sur les vues

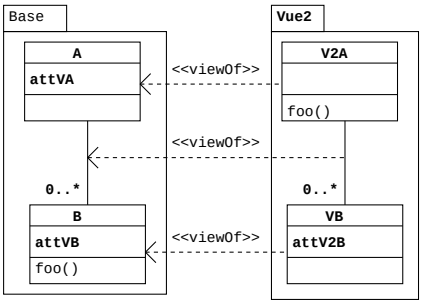

FIG. 12.4 – Modèle résultant

[1] À noter que la génération à ce niveau d'un modèle résultat fusionné supprime toute possibilité de flexibilité.

On dispose ainsi d'un moyen d'expression du modèle résultat, et de mise en œuvre du système plus flexible. Ce qui permet de privilégier des propriétés différentes pour chaque vue.

De nombreuses contraintes techniques peuvent justifier ce besoin de flexibilité, compromis entre performance et structuration par exemple, mais aussi des contraintes dues à la réutilisation de l'existant. Des besoins comme la répartition ou la sécurité peuvent également influencer sur la mise en œuvre du système.

Ces contraintes et besoins ne s'appliquent pas à tous les composants d'où ce besoin de flexibilité au niveau de leur mise en œuvre. Par extension, ces contraintes et besoins ne s'appliquent pas non plus à tous les éléments issus d'un même composant.

Le même mécanisme de transformation paramétrée peut être étendu au niveau des entités, pour permettre au concepteur de choisir pour chacune d'elles, si elle doit être fusionnée ou non. Cette possibilité est illustrée par la figure 12.5. Dans cet exemple, le concepteur a choisi de fusionner, grâce aux annotations sous forme de stéréotypes `<<merge>>`, les fragments *Ressource* et *Localisation*. Il a par contre choisi de mettre en œuvre le fragment *Stock* sous forme d'une vue, à l'aide du stéréotype *view*.

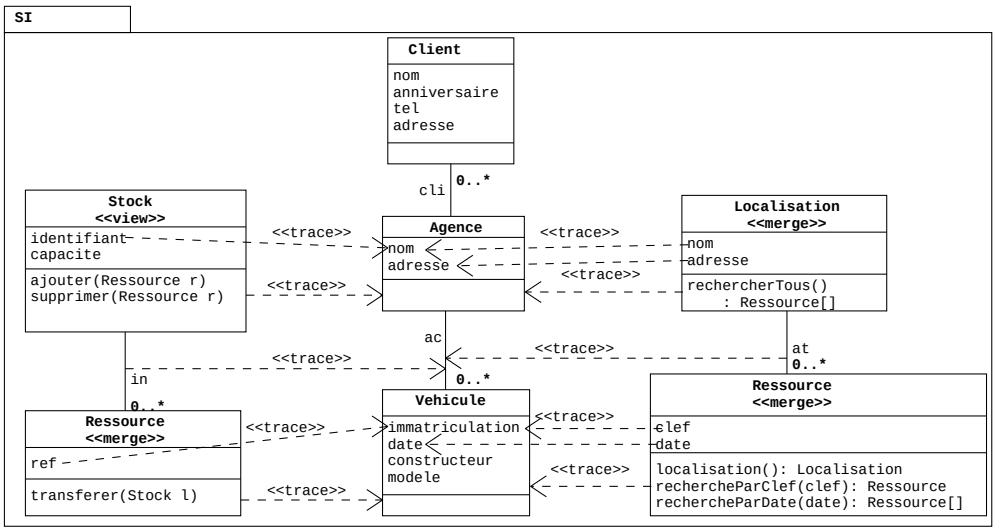

FIG. 12.5 – Modèle intermédiaire de choix sur les entités

Afin de préserver la simplicité de l'approche sur des modèles de taille plus importante, et de ne pas imposer au concepteur le travail fastidieux d'annotation de tous les fragments, le système d'annotation par défaut peut être ici aussi utilisé. Ce système permet d'annoter automatiquement tous les fragments avec une valeur par défaut (`<<merge>>` par exemple) le concepteur n'ayant plus qu'à modifier ces valeurs où il le souhaite.

La figure 12.6 présente le modèle résultant des choix de notre exemple. Tous les fragments annotés par `<<merge>>` dans notre modèle intermédiaire sont fusionnés dans le modèle résultat avec le fragment de base correspondant. Les fragments annotés

`<<view>>` sont quant à eux toujours représentés sous forme éclatée.

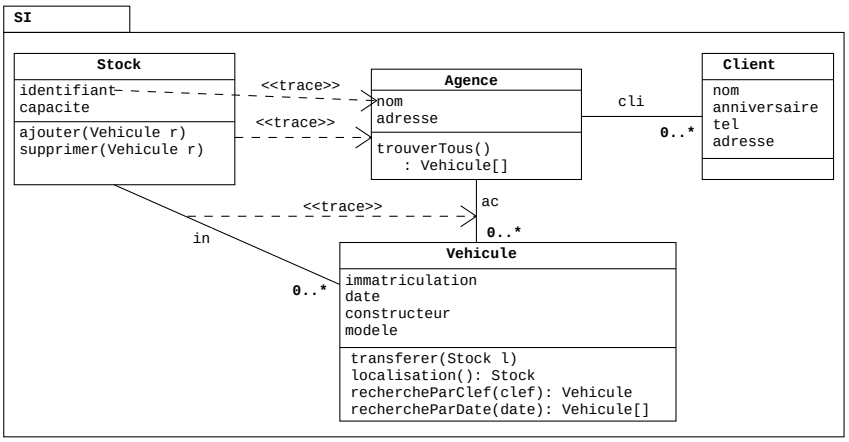

FIG. 12.6 – Modèle résultant de nos choix

Ce mécanisme permet ainsi d'étendre la flexibilité de mise en œuvre au niveau de chaque fragment. Des fragments peuvent ainsi être fusionnés avec le fragment de vue pour des raisons de performance ou de sécurité. D'autres peuvent être réalisés sous forme éclatée afin de permettre la répartition d'une entité entre différents sites d'exécution ou la réutilisation d'existant.

Il est ainsi possible d'obtenir un modèle résultat dans lequel certains composants ou certaines entités sont fusionnés et d'autres conservés. Il faut alors choisir une stratégie de projection adaptée pour la mise œuvre des composants conservés, à l'aide par exemple, de nos patrons de conception.

Nous présentons dans le chapitre suivant un ensemble de réalisations sur différentes plates-formes utilisant ces patrons pour préserver la structuration introduite lors de la conception du système.

Chapitre 13

Réalisations

Nous proposons d'étudier dans ce chapitre l'intégration de nos patrons de représentation éclatée dans différentes plates-formes (EJB, Fractal et CORBA). Une fois enrichies, ces plates-formes peuvent être utilisées pour la mise en œuvre d'un système, en conservant la structuration définie lors de sa conception par assemblage. Nous illustrons cette utilisation sur chacune des plates-formes étudiées par la mise en œuvre du système de location de véhicules.

Les sections 13.1 et 13.2 présentent l'intégration des patrons de représentation éclatée et de gestion des associations partagées respectivement aux plates-formes EJB et Fractal. La section 13.3 présente l'intégration de ces deux patrons ainsi que l'ensemble des extensions présentées dans le chapitre précédent (adaptation, gestion, fragments et uniformisation) à la plate-forme CORBA.

13.1 EJB

La plate-forme EJB (Entreprise Java Bean) [Mic02] vise à faciliter le lien entre entités métiers et bases de données. Ce lien est assuré grâce aux notions d'*EntityBean* et de *Container*. La notion d'*EntityBean* permet la réalisation d'entités métiers et de leurs relations, dont l'état est systématiquement sauvé dans une base de données. La notion de *Container* maintient le lien avec le contenu de la base de données et assure sa cohérence. Ainsi, un schéma de base de données relationnelle ou orientée objet peut facilement être traduit en un schéma EJB. De plus, des outils sont disponibles afin de distribuer et déployer de tels schémas dynamiquement dans les conteneurs.

À partir de notre patron de représentation éclatée, nous avons développé un framework pour la plate-forme EJB. Ce framework est construit par extension du framework EJB standard [CCMV03]. Nous présentons d'abord ce framework puis nous illustrons son utilisation pour mettre en œuvre l'application de gestion de véhicules (cf. chapitre 8) de façon éclatée.

13.1.1 Architecture et implémentation du framework

Le standard EJB dans sa version 2.x fournit un framework d'interfaces Java pour la définition de composants. Dans ce framework, un composant bean entité est défini (dans sa version accessible à distance) par : une interface *remote* qui hérite de *EJBObject*, d'une interface *home* qui hérite de *EJBHome* et d'une classe qui implémente l'interface *EntityBean*. Sur une plate-forme EJB, le *home* joue le rôle de fabrique et de gestionnaire du cycle de vie des composants (*Bean*).

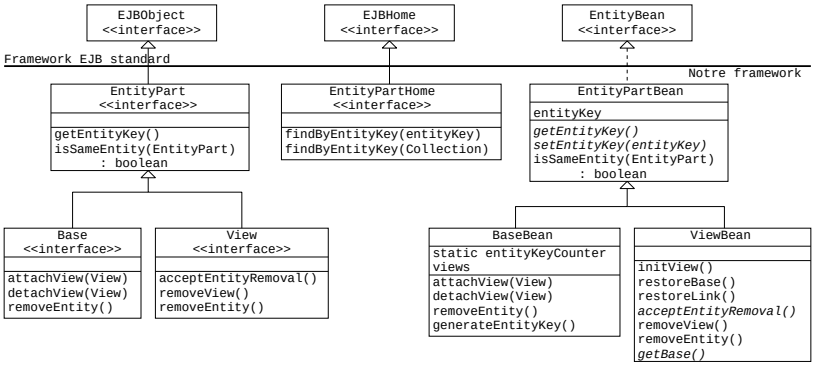

FIG. 13.1 – Extension du framework EJB

Notre framework est présenté figure 13.1. Il est composé d'interfaces *remote* (*EntityPart*, *Base* et *View*), d'une interface *home* (*EntityPartHome*), et de classes d'implémentation (*EntityPartBean*, *BaseBean* et *ViewBean*) qui correspondent aux éléments abstraits de notre patron, i.e. *EntityPart*, *Base* et *View*. Ces interfaces et classes sont liées par des liens d'héritages conformément au patron. L'interface *EntityPartHome* définit les opérations *findByEntityKey* qui correspondent à l'opération *findEntity* du patron de gestion des associations vue (cf. section 11.1.2). Cette opération est ici définie en prenant en paramètre une clef ou une collection de clefs d'entité, et retourne respectivement une *EntityPart* ou une collection d'*EntityPart*. Ce qui permet de gérer les associations de cardinalité simple ou multiple conformément au patron.

La figure 13.1 illustre également les liens d'héritage de nos éléments avec les éléments du framework EJB. Ces liens d'héritage permettent d'assurer la compatibilité avec le standard EJB. Ainsi, les composants correspondant à une entité ont tous les fonctionnalités des composants bean entités traditionnels.

Lors de la définition des classes d'implémentation il est nécessaire de faire des choix techniques pour implémenter certaines fonctionnalités du patron comme la gestion de l'identité conceptuelle ou la gestion des relations entre les fragments vues et bases.

L'identité conceptuelle est assurée par l'attribut *entityKey* (*EntityPartBean*) et gérée par l'opération *generateEntityKey* et le compteur *entityKeyCounter* (*BaseBean*).

La solution retenue pour l'implémentation des liens entre fragments utilise des références vers des composants à distance. Cette solution permet de supporter la dis-

tribution sur plusieurs sites[1] [MMJD01]. Cependant, avec cette solution, les liens sont perdus lorsque les composants sont sauvegardés. Nous résolvons ce problème en ajoutant une opération (*restoreLink*) qui restaure les liens à partir de l'identité conceptuelle. Cette opération doit être invoquée lorsqu'un composant est rechargé en mémoire.

13.1.2 Exemple

À l'aide de notre framework, le développeur peut définir des composants correspondants aux entités du domaine. Pour cela, ces composants doivent hériter des interfaces et classes du framework et doivent respecter quelques règles de programmation comme appeler les opérations héritées afin d'initialiser et gérer correctement les fragments d'entités[2].

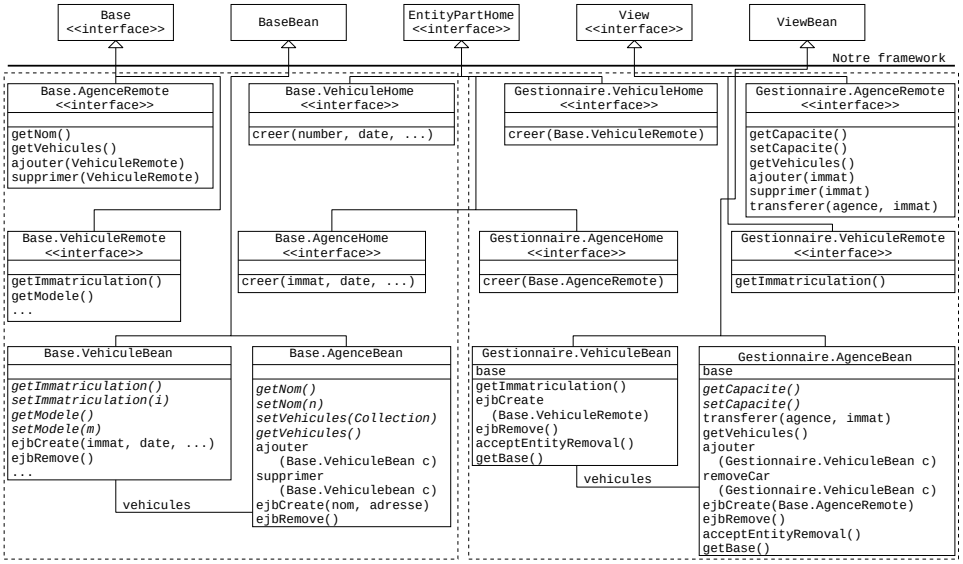

FIG. 13.2 – Utilisation du framework EJB étendu.

La figure 13.2 illustre la définition de composants. Ces composants correspondent aux fragments de deux entités : *Vehicule* et *Agence* utilisées dans notre exemple. Pour des raisons de taille, un seul schéma vue est représenté. Il est intéressant de noter que ces composants sont définis avec le même nom dans des paquetages Java différents, cela simplifie leur traçabilité.

Le mécanisme de clef d'identité conceptuelle permet de garantir que tous les fragments d'une même entité sont identifiés par la valeur identique de leur attribut *entityKey*. Dans notre framework, chaque type de fragment est géré par son propre *home*

[1] Cela n'aurait pas été le cas si nous avions implémenté ces liens par des associations gérées par le container. En effet les plates-formes EJB actuelles ne supportent pas les associations entre des composants distants.

[2] Ces tâches peuvent être automatisées par la mise en œuvre de notre patron de gestion de vue.

(*Base.VehiculeHome*, *Gestionnaire.VehiculeHome*, *Base.AgenceHome*, *Gestionnaire.-AgenceHome*).

Chaque partie du système (issu de nos composants de modèle) peut être empaquetée dans une unité de déploiement autonome. Les paquetages ainsi obtenus peuvent être déployés dans différents conteneurs à l'aide d'outils conformes au standard EJB. Comme l'illustre la figure 13.3, chaque partie du système peut alors être déployée dans des conteneurs localisés sur différents sites.

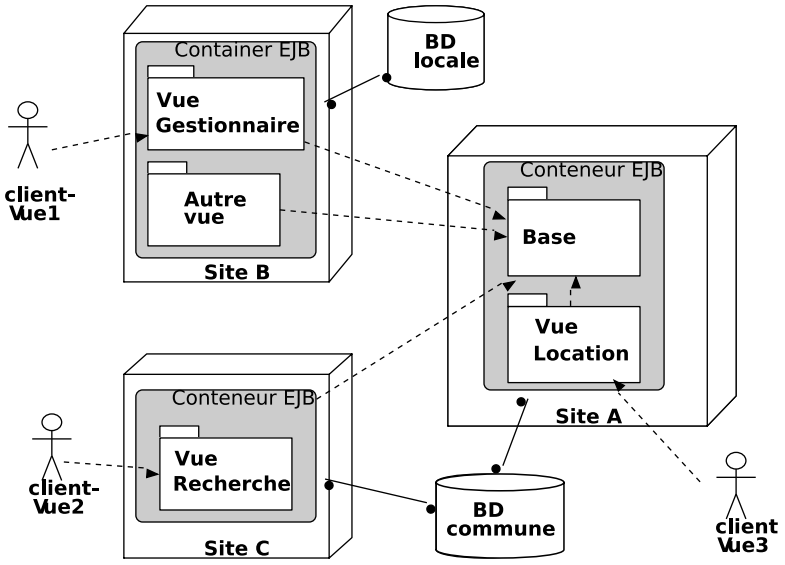

FIG. 13.3 – Distribution dans différents conteneurs.

Dans cet exemple, la base et la vue de gestion des locations sont localisées dans le même conteneur sur le même site (*Site A*), alors que les vues *Gestionnaire* et *Recherche* sont réparties sur des sites différents. Cet exemple illustre également la possibilité pour les vues réparties sur des sites différents d'utiliser la même base de données pour stocker les *EntityBean* ou au contraire, d'utiliser leur propre base.

Cette extension du framework EJB permet ainsi de conserver à l'exploitation la structuration introduite lors de la conception du système à l'aide de composants de modèle. Elle permet également de répondre aux besoins évoqués de répartition et de gestion des droits d'accès à l'aide des différentes vues qui se déduisent de cette structuration. Nous étudions dans la section suivante une extension comparable sur une autre plate-forme à composants, la plate-forme Fractal.

13.2 Fractal

Fractal [BCS04] est un modèle de composants basé sur les concepts de membrane, contenu, d'interface et de liaison. La *membrane* représente la frontière d'un composant. Celui-ci possède un *contenu*, le modèle Fractal étant hiérarchique, ce contenu

peut être une boite noire (primitif) ou un ensemble de composants. Dans ce cas, on parle de composant composite. La membrane d'un composant est munie d'*interfaces* de contrôle et d'*interfaces* fonctionnelles. Les interfaces de contrôle permettent de gérer l'aspect technique des composants : connexions, cycle de vie,... Les interfaces fonctionnelles correspondent aux fonctionnalités métiers auxquelles le composant peut répondre (interfaces serveurs) ou dont le composant a besoin (interfaces clientes). Ces interfaces définissent un ensemble de messages (correspondant à des méthodes dans les implantations). Ces composants peuvent être *liés* par mise en relation d'une interface serveur et d'une interface cliente compatibles. Cette compatibilité est vérifiée si l'interface serveur répond au moins aux messages définis par l'interface cliente.

Il existe plusieurs implantations du modèle Fractal (en Java, C ou Smalltalk) et différents outils autour de ce modèle, dont celui permettant de décrire un assemblage de composants, appelé une architecture (Fractal-ADL [BCS04]).

Nous proposons dans cette section de présenter notre extension [BMP05] pour le support de notre mécanisme de représentation éclatée dans le modèle Fractal. La figure 13.4 illustre cette idée.

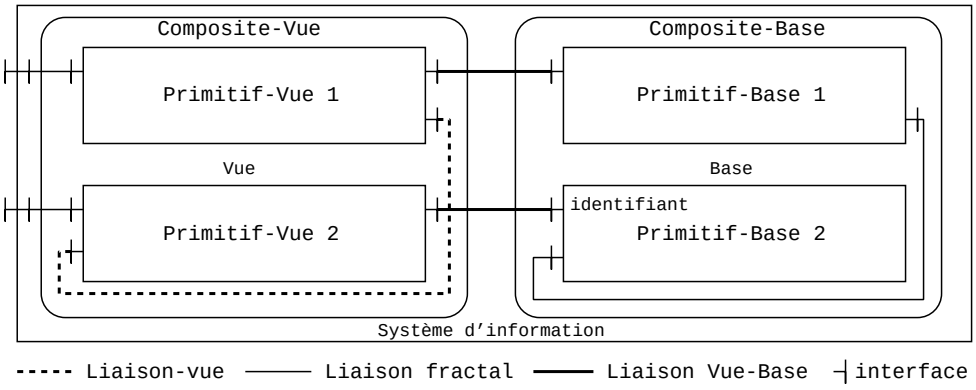

FIG. 13.4 – Représentation éclatée sur la plate-forme Fractal.

Chaque fragment est ici un composant (*Base* ou *Vue*). Cette extension supporte également la notion d'association vue, mise en œuvre sous forme de *Liaison-vue*.

13.2.1 Mise en œuvre au sein de la plate-forme Fractal

Dans un composant Fractal, le support des propriétés non fonctionnelles est réalisé par les interfaces de contrôle. L'extension de la plate-forme consiste donc à créer deux nouveaux contrôleurs permettant la définition de quatre nouveaux types de composant : les primitifs-vue, les primitifs-base, les composites-vue et les composites-base. Comme illustré figure 13.4, un *composite-base* contient les *primitifs-base* correspondant aux fragments de base. De façon semblable, à chaque vue est associé un *composite-vue* contenant les *primitifs-vue* correspondant aux fragments de vue.

Ces types de composant étendent la notion de composant primitif ou composite du

modèle Fractal. On ajoute à chacun une interface de contrôle : *base-controller* pour les composants de la base et *view-controller* pour les composants de la vue.

Les interfaces de contrôle *base-controller* et *view-controller* sont directement issues du patron de représentation éclatée (cf. figure 13.5).

Une interface de contrôle *base-controller*

L'interface de contrôle *base-controller* prend en charge la connexion et la déconnexion des composants-vue au sein de l'architecture de base, mais il permet aussi son intégration comme un nouvel élément de la membrane des composants. Ainsi, il implante à la fois l'interface « *Controller* »définie dans la spécification Fractal et l'interface « *Base* » issue du patron.

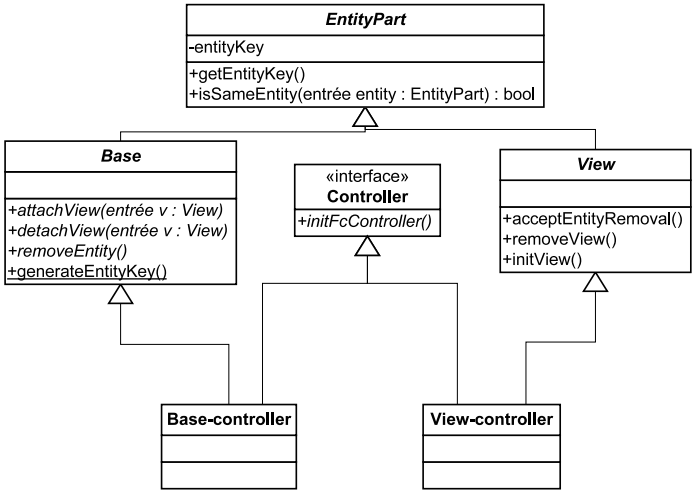

FIG. 13.5 – Interfaces de contrôle pour la gestion des vues.

L'interface de contrôle *base-controller* maintient une liste des vues qui lui sont liées. Cette liste lui permet de notifier l'ensemble des vues auxquelles il est accroché lors de l'arrêt du composant ou de sa destruction.

Une interface de contrôle *view-controller*

L'interface de contrôle *view-controller* prend en charge la demande de connexion et de déconnexion des composants-vue au sein de l'architecture de base et les fonctionnalités nécessaires à son intégration comme un nouvel élément de la membrane des composants. Ainsi, il implante à la fois l'interface « *Controller* » et l'interface « *View* » (cf. figure 13.5).

L'interface de contrôle *view-controller* gère, au moment de l'attachement du composant-vue sur le composant-base, la création des liaisons au sens Fractal entre les interfaces clientes du composant-vue et les interfaces serveur du composant-base. De même, il

gère la suppression de ses liaisons de façon dynamique en cas d'appel à la méthode *removeView*.

Une interface de contrôle de liaisons-vue *viewbind-controller*

La notion de liaison-vue introduit le principe d'une liaison entre deux composants-vue validée uniquement si une liaison entre deux composants-base est présente à l'exécution. Comme Fractal est un modèle de composant permettant une reconfiguration dynamique de l'architecture, il est impossible de savoir a priori s'il existe toujours une liaison entre deux composants à l'exécution. Nous introduisons une interface de contrôle de liaison-vue au niveau des composants-vue. Celle-ci prend en charge les liaisons-vue. Elle permet une invocation du service entre les vues uniquement si la liaison de référence entre les bases est toujours valide.

Exemple d'architecture logicielle étendue à l'aide de vues

En reprenant une partie de notre exemple d'agence de location de véhicules, l'architecture du système d'information est composée d'une base possédant trois types de composants et de plusieurs vues venant étendre les fonctionnalités de la base. Les trois types de composants sont un composant *Agence*, un ensemble de composants *Vehicule* que l'agence peut louer et un ensemble de *Clients* de l'agence.

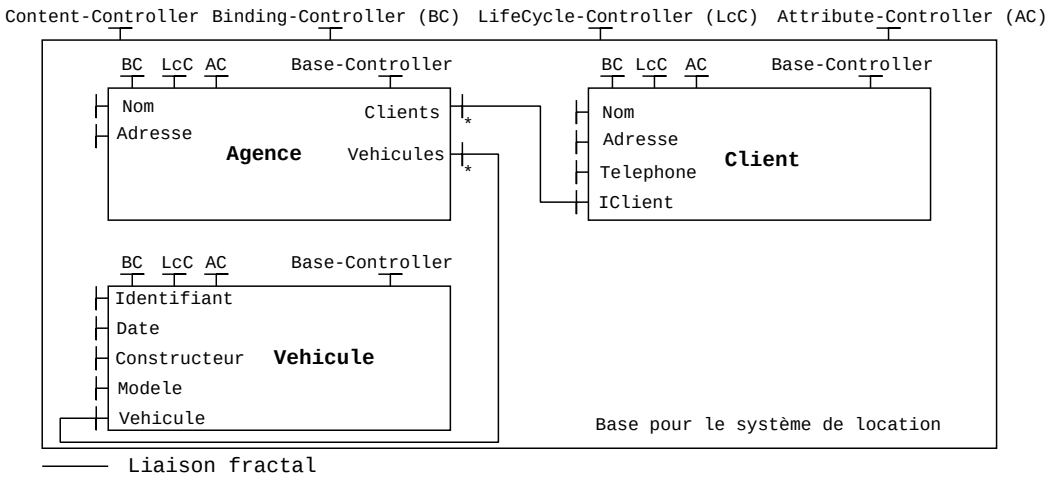

FIG. 13.6 – Exemple d'une architecture de base

Une *Agence* fournit trois interfaces serveur permettant d'accéder à son nom, son adresse la liste des *Vehicule* dont elle a la gestion. Elle requiert une interface permettant de gérer la liste des véhicules dont elle a la charge (*Vehicules*). Cette dernière est une interface de type multiple, elle permet la connexion de plusieurs composants *Vehicule* sur cette interface. Le composant *Vehicule* possède quant à lui cinq services fournis

permettant d'identifier le véhicule (*Identifiant*), de connaître sa marque (*Constructeur*), son modèle (*Modele*), sa date de mise en circulation (*Date*). Le dernier service fourni, *IVehicule*, est une interface permettant d'établir un lien en entre un véhicule et une agence. Enfin, le composant Client fournit quatre interfaces fonctionnelles. Elles permettent respectivement de connaître, le nom, l'adresse et le téléphone du client. La dernière interface *IClient* sert de référence pour les composants *Client* au sein de l'agence (cf. figure 13.6).

Une première vue de recherche permet d'accéder à un service de recherche pour le système de location de véhicule (cf. figure 8.1 chapitre 8). Cette nouvelle vue s'articule autour de deux composants-vue augmentant les fonctionnalités de base du système d'information de l'agence. Le premier ajoute la possibilité de faire une recherche de l'ensemble des véhicules contenus dans l'agence. Le deuxième fournit un service de recherche par critères, ici la date et l'identifiant du véhicule (cf. figure 13.7).

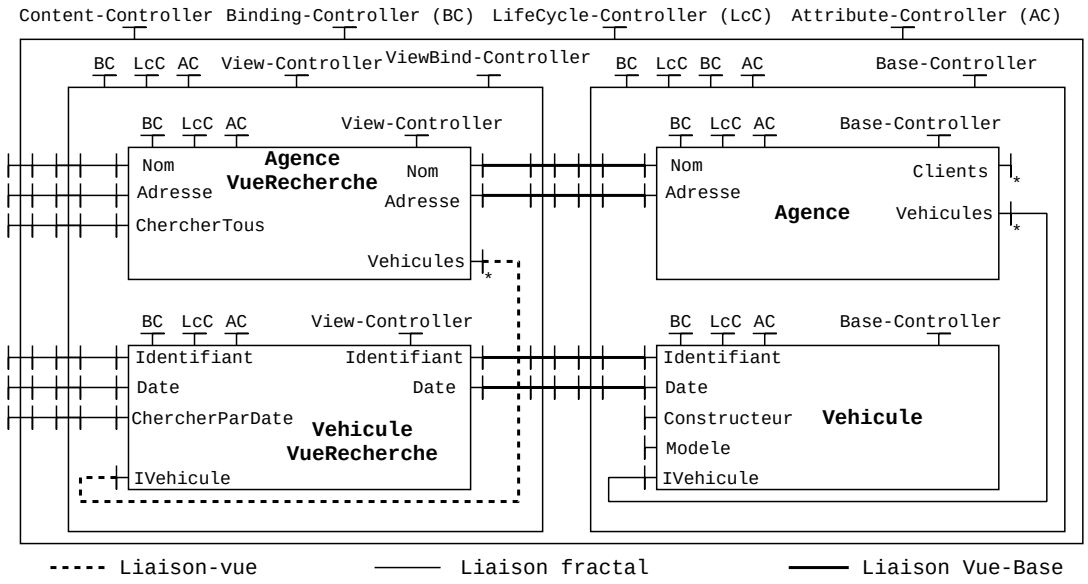

FIG. 13.7 – Exemple d'attachement d'une vue *Recherche* sur un système d'agence de location

Une deuxième vue ajoute la gestion des locations au système d'information de l'agence. Cette vue introduit deux composants-vue et un composant primitif classique représentant chaque location. Les composants-vue viennent s'accrocher respectivement sur les composants *Client* et *Vehicule*. Le composant *Location* est un composant primitif classique créé pour chaque nouvelle location d'un véhicule par un client. Il propose une interface fonctionnelle pour la gestion des locations et une interface associant le client et le véhicule (cf. figure 13.8). Il est ainsi possible d'enrichir un même composant de base par un ensemble de composants vues, ici illustré par l'enrichissement du composant *Vehicule* par les composants *VehiculeVueRecherche* et *VehiculeVueLocation*.

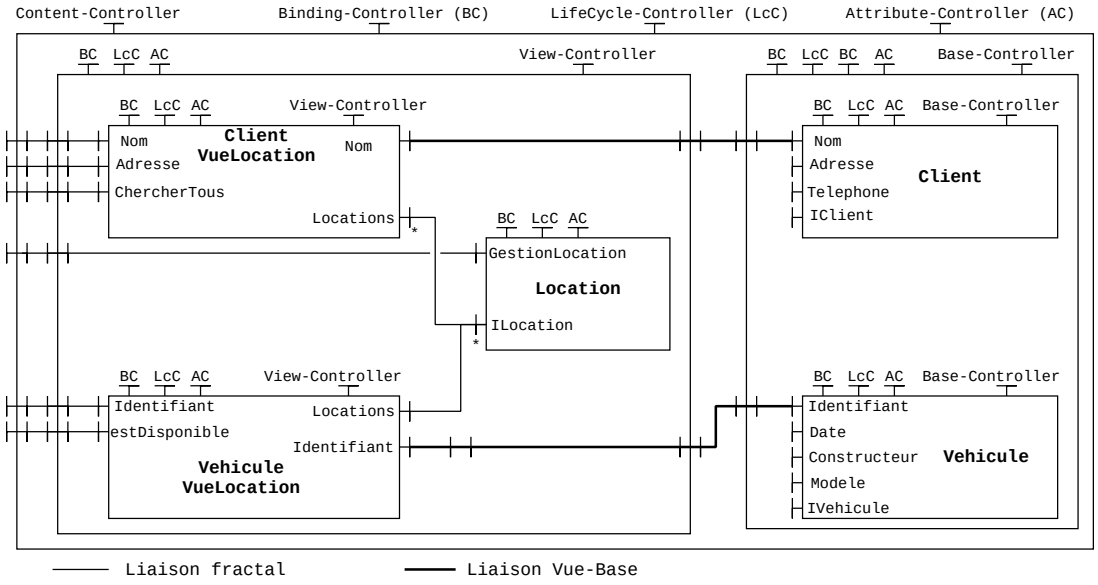

FIG. 13.8 – Exemple d'attachement d'une vue *location* sur un système d'agence de location

13.2.2 Extension de l'ADL

L'ADL (Architecture Description Language) Fractal repose sur un ensemble de descripteurs XML permettant la définition d'une architecture logicielle. Ce langage est prévu dans sa version 2.0 pour être extensible. Dans ce sens, il permet l'utilisation de nouveaux types de composants comme les *basePrimitive* ou les *viewPrimitive* dans la mesure où un descriptif de ces types est fourni au niveau de la configuration de la plate-forme. Aucune balise n'est ajoutée pour la définition des composants-vue ou des composants-base. Nous utilisons la balise *controller desc* pour préciser les éléments des membranes des composants. Par exemple, la balise *controller desc = basePrimitive* crée un composant muni d'une interface de contrôle *Base-Controller*.

Le modèle introduit, par contre, deux nouvelles balises au sein de l'ADL. La première caractérise la relation entre un composant-vue et un composant de base. La deuxième définit une liaison-vue entre deux composants-vue. Cette dernière balise identifie les deux interfaces qu'elle connecte au niveau de la vue et les deux interfaces devant être connectées au niveau de la base.

La première balise nommée *attachVB* permet de lier un composant-vue à un composant-base. Cette balise possède deux attributs désignant le composant-vue et le composant-base à attacher.

La deuxième balise nommée *viewBind* permet de connecter deux composants-vue. Cette balise possède quatre attributs déterminant les deux interfaces dans la vue à connecter et les deux interfaces de référence permettant de déterminer la liaison de référence au niveau de la base. En effet, une liaison en Fractal n'est pas nommée, son

identification se fait donc par l'intermédiaire des interfaces qu'elle connecte.

La figure 13.9 illustre une partie du descripteur XML permettant la définition d'un système d'information contenant deux composants-base et deux composants-vue liés (le fichier complet est donné en annexe B).

```
01 <definition name="System">
02    <component name="Agence">
04     <interface name="Vehicules" role="client" signature="Vehicules"/>
05     ...
06     <interface name="Nom" role="server" signature="Name"/>
07     < controller desc = "basePrimitive"/>
08   </component>
09   <component name="Vehicule">
10     <interface name="IVehicule" role="server" signature="Vehicules"/>
11     ...
12     <controller desc = "basePrimitive"/>
13   </component>
14   <component name="AgenceVue">
15     <interface name="Vehicules" role="client" signature="Vehicules"/>
16     ...
17     < controller desc = "viewPrimitive"/>
18   </component>
19   <component name="VehiculeVue">
20     <interface name="IVehicule" role="server" signature="Vehicules"/>
21     ...
22     <controller desc = "viewPrimitive"/>
23   </component>
24   <binding client="Agence.Vehicules" server="Vehicule.IVehicule"/>
25    ...
26   <attachVB base="Agence" view="AgenceVue"/>
27   <attachVB base="Vehicule" view="VehiculeVue"/>
28   <viewBind client="AgenceVue.Vehicules" server="VehiculeVue.IVehicule"
          refClient="Agence.Vehicules" server="Vehicule.IVehicule" />
29 </definition>
```

Fig. 13.9 – Descripteur XML partiel de l'application de la figure 13.7

Cette première intégration de notre patron de représentation éclatée dans le modèle Fractal permet de préserver la structuration du système jusqu'à son exécution. L'extension de l'ADL permet quant à lui de décrire l'assemblage des différents fragments d'une entité. Les mécanismes de membrane et de composite permettent de faire figurer dans l'architecture la structuration issue des différentes vues. Fratcal propose également un mécanisme de partage d'un même composant entre différentes membranes. Ce mécanisme semble une piste intéressante pour tenir compte de la double structuration du système, par vue et par entité (cf. partie VI).

Ces extensions des plates-formes EJB et Fractal nous ont permis d'expérimenter l'utilisation des patrons de représentation éclatée et de gestion des associations vue.

Nous proposons dans la section suivante d'expérimenter l'ensemble des extensions proposées au chapitre 11 sur la plate-forme CORBA, en particulier l'adaptateur et le gestionnaire de vue, afin de permettre la réalisation de composants binaires réutilisables.

13.3 CORBA

CORBA (Common Object Request Broker Architecture) [COR] est une spécification de l'OMG pour une plate-forme à objets répartis indépendante d'un langage, d'un système ou d'un vendeur particulier. Elle supporte de nombreuses préoccupations techniques, distribution, inter-opérabilité, sécurité,...

Une application CORBA est basée sur l'utilisation de contrats IDL (Interface Definition Language). Ces fichiers IDL permettent de définir les interfaces des objets indépendamment de tout langage.

Nous avons utilisé ces contrats IDL dans notre expérimentation pour définir les interfaces issues de notre patron de représentation éclatée et de l'ensemble de ses extensions. Nous avons ensuite réalisé un ensemble de classes abstraites afin d'implanter les mécanismes de gestion des fragments vues[3].

La réalisation d'un système sur notre plate-forme revient alors à spécialiser ces interfaces IDL et classes abstraites. La figure 13.10 illustre l'ensemble de ces contrats IDL. Le module *Entities* regroupe les interfaces abstraites pour la définition des éléments issues de nos patrons. Les autres modules correspondent aux interfaces pour la mise en œuvre de notre exemple (cf. figure 8.1 chapitre 8).

Le concept d'association n'existe pas en IDL, elles sont ici représentées pour faciliter la compréhension et matérialisées au travers des attributs.

13.3.1 Interfaces abstraites issues des patrons

L'interface *EntityPart* définit les opérations pour attacher une vue, détacher une vue, tester si une autre *EntityPart* appartient à la même entité. Ces opérations sont directement issues des patrons de représentation éclatée et de gestion des vues associations. Elles sont définies au niveau de *EntityPart* conformément à l'extension d'application uniforme des vues.

Elle définit également une opération pour obtenir un fragment de vue appartenant à un *Manager* particulier. Une *EntityPart* est, en effet, toujours associée à un *Manager* particulier au travers de son attribut *manager*. Les managers peuvent être des managers de base (*BaseManager*) ou de vue (*ViewManager*). Ils mettent en œuvre l'extension de gestion des fragments de vue de notre patron.

L'interface *Base* spécialise l'interface *EntityPart* et ajoute simplement l'opération permettant de supprimer l'entité dans sa globalité (cf. section 11.4).

L'interface *View* définit les opérations spécifiques à un fragment de vue. L'interface *Adaptator* permet de faire le lien entre un fragment de vue et son *EntityPart* cible conformément à l'extension adaptateur de notre patron.

[3]Il existe un grand nombre d'implantations de la spécification CORBA. Nous avons utilisé pour notre expérimentation, la plate-forme CORBA du JDK de SUN.

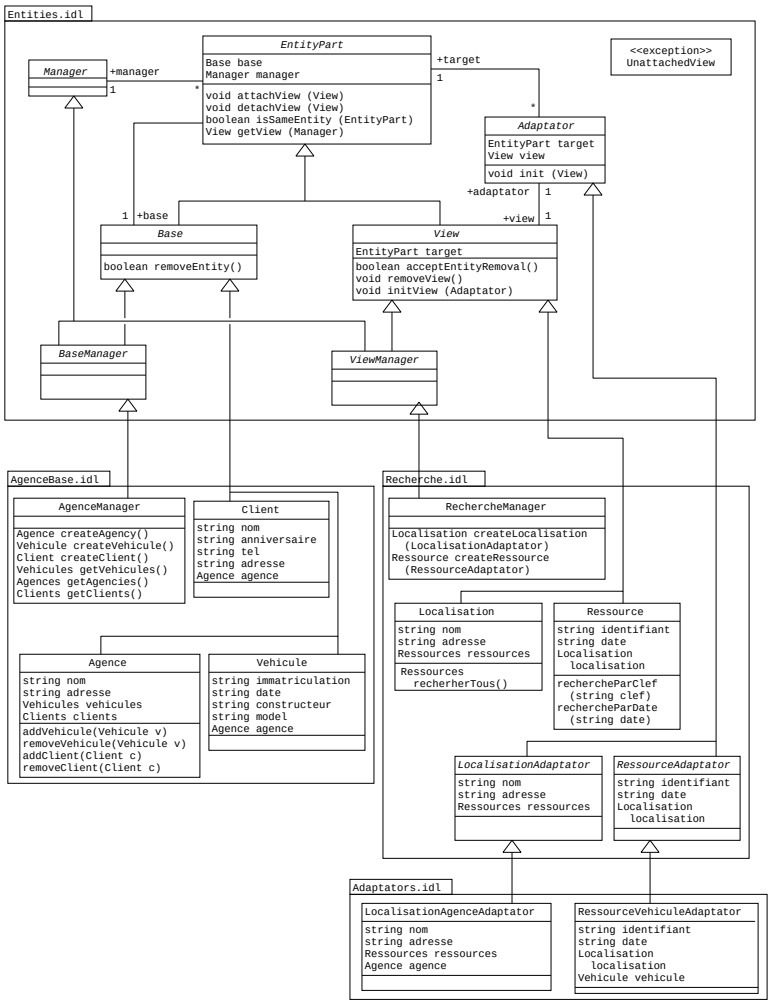

FIG. 13.10 – Définition et utilisation du framework de représentation éclatée sur la plate-forme CORBA.

Une fois initialisé, un fragment de vue est toujours associé à un et un seul adaptateur. Il est donc toujours possible d'obtenir son adaptateur, et inversement d'obtenir le fragment de vue d'un adaptateur. C'est pourquoi, même si un *EntityPart* est lié à ses fragment de vue par l'intermédiaire d'adaptateurs, ce sont des éléments de type *View* qui sont paramètres des opérations *attachView* et *detachView*.

Enfin le module *Entities* définit également une exception *UnattachedView* qui sera levée lors de l'accès à un fragment de vue non initialisé.

Ces méthodes peuvent être implantées une fois pour toutes dans un ensemble de classes abstraites déchargeant ainsi le développeur de cette tâche. C'est ce que nous avons réalisé dans notre framework pour le plate-forme CORBA. Ce framework implante directement les protocoles de gestion des fragments issus de nos patrons.

13.3.2 Application à notre exemple

La figure 13.10 illustre également la définition des interfaces IDL pour notre exemple. Celles-ci sont regroupées dans trois modules distincts. *AgenceBase* regroupe les interfaces pour la définition des éléments de base du système (*Agence*, *Client* et *Vehicule*) ainsi que de leur manager (*AgenceManager*). *Recherche* regroupe les définitions des éléments de la vue de recherche et *Adaptators* la définition des adaptateurs de la vue de recherche appliquée aux véhicules.

Nous définissons pour chaque interface un type *sequence* afin de pouvoir en représenter une collection. Les associations entre entités sont ici traduites en attributs (simples ou de type Collection suivant la cardinalité). Cependant, en CORBA, ces attributs seront traduits à l'implantation en méthodes d'accès, ce qui est conforme à notre patron de gestion des associations vue.

L'implantation des éléments de base ne contient aucun code spécifique à la représentation éclatée, les mécanismes issus des patrons étant pris en charge par notre framework.

Les éléments vues (ici *Localisation* et *Ressource*) peuvent quant à eux être réalisés par rapport aux interfaces abstraites des adaptateurs (*LocalisationAdaptator* et *Ressource-Adaptator*). Ces implantations étant indépendantes d'un système particulier, elles peuvent être réutilisées sans modifications. Cela autorise la réalisation de composants binaires génériques dont les interfaces sont définies à l'aide des contrats IDL.

L'utilisation d'un composant (ici *Recherche*) pour un système se fait alors par l'implantation des adaptateurs spécifiques à ce système (*LocalisationAgenceAdaptator* et *RessourceVehiculeAdaptator*) du module *Adaptators*. Il est alors possible, dans notre exemple, d'associer les fragments de vue *Localisation* et *Ressource* aux fragments de base correspondant *Agence* et *Vehicule*. Le code d'un adaptateur étant facilement "calculable", il peut être généré automatiquement à partir d'un assemblage. Pour utiliser un composant binaire sur une autre partie du système ou une autre base, il suffit alors de générer les adaptateurs spécifiques au nouvel assemblage défini au niveau des composants de modèle. Un exemple complet (modèle d'assemblage, modèle de traces et spécifications IDL) montrant une double application du composant recherche et une chaîne d'applications est donné en annexe C.

Cette réalisation nous a permis d'expérimenter dans un même framework l'ensemble des patrons proposés au chapitre 11. Les propriétés de la plate-forme CORBA permettent de répartir les différentes vues d'un même système et offrent la possibilité de les réaliser dans des langages de programmation différents. La plate-forme CORBA offre, comme sur la plate-forme EJB[4], des propriétés de dynamicité autorisant l'ajout et le retrait dynamique d'une vue au système. En outre, notre framework permet la réalisation de composants binaires réutilisables par adaptation.

Ces réalisations sur les plates-formes EJB, Fractal ou CORBA constituent le dernier niveau dans nos chaînes de production (cf. figure 10.17). Nous présentons dans la section suivante notre atelier de modélisation supportant l'ensemble des niveaux, de la réalisation des composants de modèle et de leurs assemblages, jusqu'à la génération de code sur une plate-forme d'exécution.

[4]Cette propriété n'est pas systématique en Fractal et dépend de la plate-forme choisie.

13.4 Un atelier de composition de modèles

Comme présenté dans la section 9.3, une première expérimentation sous la forme d'un profil UML 1.X[5] dans l'atelier Objecteering a été réalisée. Elle a permis d'outiller l'approche par vues de la conception de modèles jusqu'aux projections EJB et CCM [Mul, MCCV03]. Nous présentons ici un prototype d'atelier basé sur le standard UML2 et reprenant l'ensemble de l'approche. Cet outil permet de démontrer comment nos composants de modèle peuvent être intégrés dans des ateliers de modélisation et leur pertinence pour la conception de modèles de systèmes. Ce prototype est librement disponible[6].

13.4.1 Fonctionnalités

Notre outil a été conçu pour permettre la conception de composants de modèle sous forme de paquetages templates et la réalisation de modèles d'assemblage. La figure 13.11 illustre la fenêtre principale de cet outil. Pour définir l'application d'un composant, l'utilisateur établit une relation *apply* entre ce composant et celui auquel il souhaite l'appliquer. Il doit ensuite renseigner les paramètres de cette application. Pour cela, notre outil propose une aide à la saisie qui ne propose à l'utilisateur que les éléments de type compatible. Cette aide pourra être étendue en ne proposant que les éléments conformes au modèle requis.

À partir d'une application, notre outil permet de fusionner les éléments définis par le composant dans le modèle cible en tenant compte des éventuelles substitutions de type dues au paramétrage conformément au mécanisme exposé à la section 8.1. Le résultat de la fusion est illustrée par la figure 13.12. À partir de ce modèle fusionné, l'outil permet de générer une implantation Java.

Enfin, notre outil permet également, à partir d'un composant de modèle, de générer le fichier IDL et l'implantation Java pour une mise en œuvre sur la plate-forme CORBA telle que nous l'avons présentée dans le chapitre précédent. Les adaptateurs (fichiers IDL et implantations Java) sont quant à eux générés à partir du paramétrage des relations d'application.

Nous étendons actuellement notre outil afin de lui ajouter des fonctionnalités pour la gestion de bibliothèques de composants. L'intégration du vérificateur de contraintes OCL [Van05] est également envisagée afin, notamment, d'intégrer les contraintes telles que nous les avons définies pour nos composants de modèle et notre relation *apply*. En effet, ces contraintes sont actuellement écrites en Java.

13.4.2 Réalisation

Pour réaliser notre outil, nous nous sommes basés sur le framework EMF (Eclipse Modeling Framework) [FDE+04] et l'implantation UML2 réalisée à partir de ce framework. Cette implantation permet de représenter en mémoire un modèle UML2.

[5] Le module est librement téléchargeable à l'adresse http ://www.lifl.fr/∼mullera.
[6] http ://www.lifl.fr/∼mullera/cocoamodeler/

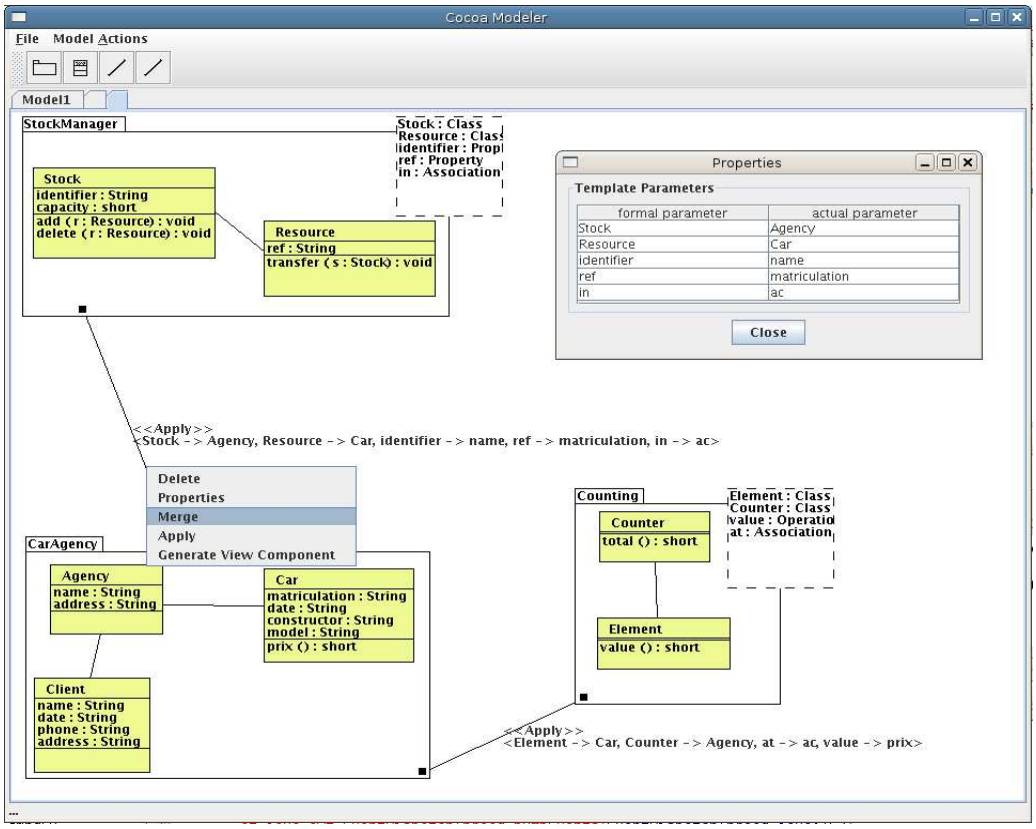

FIG. 13.11 – Atelier de conception de composants de modèles

Concrètement, à chaque méta-classe UML2, correspond une classe Java.

Cette réalisation est structurée en trois paquetages, le premier contient les classes de représentation graphique des éléments de modélisation (classe, paquetage, association,...), le second les fenêtres d'édition des propriétés de ces éléments. Le dernier contient les mécanismes non graphiques propres à notre outil :

- Vérification du paramétrage (modèle bien formé).
- Vérification de la conformité de l'application.
- Fusion de composants de modèles.
- Génération Java.
- Génération CORBA, IDL et implantations Java.

Ces mécanismes manipulent directement les éléments instances du méta-modèle UML2, comme l'illustre la figure 13.13.

L'éditeur proposé avec EMF permet de manipuler les modèles sous une forme arborescente mal adaptée à l'édition de modèles complexes. Nous avons donc réalisé une hiérarchie d'éléments de modèle graphiques. Cette hiérarchie de classes est basée sur la classe JComponent de la bibliothèque Java afin de disposer sur nos éléments des mécanismes de gestion des événements. L'affichage proprement dit de nos éléments

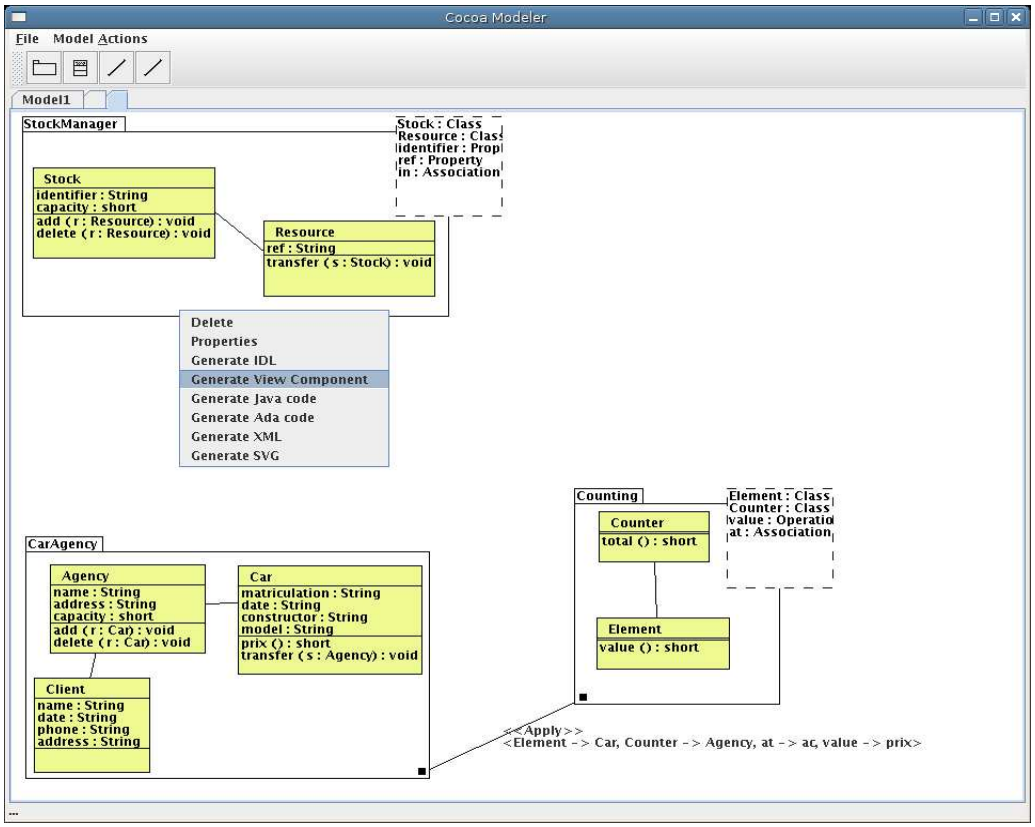

FIG. 13.12 – Menu contextuel d'un composant de modèle

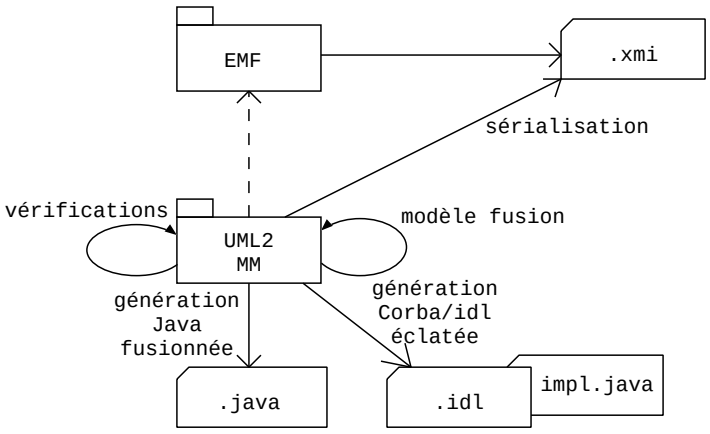

FIG. 13.13 – Architecture de notre outil

graphiques est programmée à l'aide de primitives Java2D.

Notre outil permet l'édition simultanée de plusieurs modèles afin, par exemple, de pouvoir concevoir les composants de modèle séparément avant de les assembler dans un même modèle. Ces modèles peuvent être enregistrés et ouverts dans le format standard XMI (XML Metadata Interchange) pour la sauvegarde de modèles. Nous avons utilisé le moteur Velocity pour la génération de code (aussi bien Java que IDL) et la génération de fichiers SVG (format standard de dessin vectoriel). Celle-ci permet de définir des fichiers templates de génération facilitant la mise au point de tels traitements.

Le vérificateur de contraintes OCL de Dresden [HDF02] a déjà été couplé au framework EMF [CCMV04] et devrait ainsi être facilement utilisable dans notre outil. Un travail d'intégration du langage de scripts de l'équipe à EMF [EMS] pourrait également être ajouté à notre outil pour laisser par exemple à l'utilisateur la possibilité de définir ses propres transformations.

Notre outil n'est qu'à l'état de prototype nécessitant encore de nombreuses améliorations pour être réellement utilisable. Mais celui-ci nous a permis d'expérimenter l'intégration de nos composants de modèle dans un atelier de génie logiciel.

Sixième partie

Conclusion et perspectives

Bilan

La première partie nous a permis d'introduire les différents axes de complexité des systèmes devant répondre à un grand nombre de préoccupations, aussi bien métier que technique et évoluer dans le temps. Nous avons vu que la maîtrise de cette complexité passe par la structuration des systèmes et que la réutilisation permet de faire face aux délais de conception toujours plus courts. Nous avons également pu voir que les modèles, grâce aux niveaux d'abstraction qu'ils ajoutent, apportent une meilleure maîtrise de la complexité. Ils permettent, en effet, d'exprimer les systèmes indépendamment de leurs mises en œuvre. De plus, cette indépendance permet de capitaliser les solutions exprimées par les modèles afin de les réutiliser sur différentes plates-formes. C'est ce que propose l'OMG avec la démarche MDA. L'idée du MDE va plus loin en proposant de structurer l'espace des modèles dans toutes ses dimensions et de permettre leur réutilisation pour la construction de nouveaux systèmes. Mais de nombreuses questions se posent quant à l'expression de modèles génériques, de leur composition et de leurs mises en œuvre.

La deuxième partie nous a permis d'étudier différentes approches pour la structuration de modèles. Nous avons vu que certaines d'entre elles proposent l'expression de modèles génériques réutilisables pour la conception de différents systèmes. Certaines permettent la définition d'artefacts manipulables grâce à une définition stricte des éléments de modélisation utilisés. Cependant, la partie générique de ces modèles est exprimée par un ensemble de noms d'éléments ou au mieux par un ensemble de classes à un faible niveau de granularité [Mul04]. Aucune des approches étudiées ne propose l'expression de modèles eux-mêmes paramétrés par un modèle afin de permettre la composition de fonctionnalités plus complexes et plus riches.

Nous avons également étudié dans cette seconde partie, l'état des standards MOF et UML. Nous avons pu constater qu'ils sont en constante évolution et qu'ils ne proposent pas d'autres mécanismes de composition de modèles que la fusion basée sur le nom des éléments. Ils proposent cependant un mécanisme d'expression de modèles paramétrés par un ensemble d'éléments, nommé template et utilisé par différentes approches.

Dans la troisième partie, nous avons illustré (au chapitre 5) la possibilité d'exprimer, à un niveau modèle, des fonctionnalités génériques et de les utiliser pour la conception de systèmes [MCCV03]. Nous avons ensuite présenté notre proposition de construction de systèmes par application de modèles paramétrés. Dans notre approche, ces modèles sont eux-mêmes paramétrés par un modèle afin de permettre l'expression d'une partie requise complexe. Il est ainsi possible de définir des composants de modèle dont les

interfaces requises et fournies sont exprimées par un modèle. La fonctionnalité exprimée par un composant peut alors être ajoutée à un modèle par un mécanisme d'application. Ce mécanisme consiste à mettre en relation le modèle requis par le composant avec un modèle conforme auquel la fonctionnalité doit être ajoutée. Ce mécanisme fonctionne aussi bien pour l'application à un modèle de base qu'à un autre composant de modèle, permettant ainsi la construction de fonctionnalités complexes à partir de fonctionnalités plus simples. La conception d'un système peut alors être réalisée par l'assemblage d'un ensemble de composants de modèle.

Le chapitre 7 nous a permis de formaliser ce mécanisme d'application sous forme d'un opérateur [MCCV05]. Nous avons ainsi démontré des propriétés d'ordre permettant de garantir la cohérence des alternatives de composition. Elles permettent notamment de définir des chaînes d'applications et la possibilité d'ajouter une fonctionnalité à un système sans remise en cause des applications précédentes.

Nous avons ensuite illustré la possibilité d'exprimer le modèle du système résultat selon différents modes, notamment un mode fusionné permettant l'expression du résultat sous forme d'un modèle classes-associations classique. Nous avons également proposé des modes d'expression permettant de préserver dans le modèle résultat la structuration introduite lors de sa conception.

Nous avons donné, dans la quatrième partie, une définition stricte de nos artefacts de modélisation, les rendant en particulier manipulables. Nous avons utilisé pour cela une technique de méta-modélisation et un ensemble de contraintes exprimées à l'aide du langage OCL. Afin de permettre l'expression de nos composants de modèle et de leur assemblage à l'aide du langage standard UML, nous avons réalisé nos méta-modèles par extension du méta-modèle UML. Nos composants de modèle sont ainsi définis à partir du concept de paquetage template auquel nous avons ajouté un ensemble de contraintes afin de garantir que son paramétrage forme bien un modèle. Notre relation pour exprimer l'application d'un composant de modèle (*apply*) est également définie par extension du méta-modèle standard UML. Les contraintes que nous avons définies (en OCL) permettent de garantir la conformité entre modèle requis et modèle fourni. Nous avons également vu, après l'avoir formalisé, que la relation *bind* permet d'exprimer la trace entre un modèle de système et les composants de modèles utilisés pour sa conception [CCMV04].

Enfin, dans la cinquième partie, nous avons étudié des mises en œuvre de systèmes conçus par assemblage de composants de modèle. En effet, à partir des modèles résultats présentés au chapitre 8, il est possible d'utiliser différentes stratégies de projection. En partant d'un modèle fusionné, des stratégies et outils de génération de code à partir d'un modèle UML peuvent, par exemple, être utilisés. Afin de pouvoir préserver la structuration du système jusqu'à son exploitation, nous avons proposé un ensemble de patrons de conception, utilisable sur toutes plates-formes à objets [CCMV03, CCMV05]. Ceux-ci permettent également de préserver, jusqu'à l'exploitation, le caractère générique des fonctionnalités exprimées par les composants de modèle. Ce qui autorise la réalisation de véritables composants binaires.

Au chapitre 12, nous avons proposé une technique de transformation paramétrée par des annotations [BCGM04], afin de laisser au concepteur plus de flexibilité, pour la mise

en œuvre de son système, entre le tout fusionné et le tout éclaté. Cette technique lui permet ainsi de sélectionner les fonctionnalités ou les entités qui devront être fusionnées et celles qui seront mises en œuvre sous forme éclatée.

Nous avons également étudié dans cette partie l'application de nos patrons de conception à différentes plates-formes, EJB, Fractal et CORBA [CCMV03, BMP05]. À partir de chaque plate-forme étendue, nous avons pu illustrer la mise en œuvre d'un système tout en conservant sa structuration. En outre, nous avons vu que ce type de mise en œuvre permet une évolution dynamique du système par ajout et/ou retrait de fonctionnalités.

Enfin, nous avons présenté notre outil de conception. Celui-ci supporte l'ensemble de notre démarche, de la conception des composants de modèles jusqu'à la génération du code de mise en œuvre d'un système, en passant par l'expression de son assemblage.

Nous avons ainsi montré qu'il est possible de concevoir un système par l'assemblage de fonctionnalités génériques exprimées sous forme de modèles. Nous avons démontré qu'il est possible d'exprimer et de valider la composition de modèles. Nous avons également exploité la structuration introduite lors de la conception du système pour la conserver jusqu'à son exploitation. Ceci facilite la traçabilité et il est ainsi possible de faire évoluer un système en exprimant au niveau modèle l'ajout dynamique d'une préoccupation au système.

Perspectives

Ces premiers résultats laissent entrevoir de nombreuses perspectives. Des perspectives à court terme se situent au niveau des plates-formes. Il serait en effet intéressant d'étudier des mécanismes propres à certaines plates-formes pour faciliter la mise en œuvre structurée des systèmes. Une expérimentation en ce sens est actuellement étudiée sur la plate-forme Jboss/aop [JBA] afin d'y intégrer notre patron de représentation éclatée sous forme d'un service d'aspects. De même, une perspective intéressante pour la mise en œuvre sur la plate-forme Fractal s'appuie sur le concept de composant partagé. En effet, celui-ci est caractéristique du modèle Fractal et peu présent dans les autres modèles de composants existants. Il permet à un composant d'appartenir à plusieurs composites et donc plusieurs assemblages, chacun d'eux étant délimité par une membrane. En considérant les membranes comme un moyen de structurer le système en différentes dimensions, un composant peut ainsi intervenir dans plusieurs dimensions. L'objectif de nos patrons est de permettre la mise en œuvre des systèmes en conservant une double structuration entités/fonctions. Il est intéressant de faire le parallèle avec ce concept de composant partagé. En effet, dans notre modèle, un même fragment appartient à la fois à une entité et à une vue. L'utilisation de membranes Fractal pour réaliser cette double structuration semble offrir de bonnes propriétés. Notamment la possibilité de pouvoir s'adresser globalement à l'ensemble des composants constituant une même entité, ou à l'ensemble des composants intervenant dans une même vue.

D'autres perspectives se trouvent au niveau des ateliers de conception. Notamment l'étude de l'intégration de notre opérateur dans les processus de conception, par son opérationalisation à l'aide de langages d'actions comme ASL [WKC+01], QVT [QVT05] ou KerMeta [KER]. L'intégration d'un moteur OCL permettrait, comme nous l'avons évoqué à la section 13.4, d'utiliser les contraintes telles qu'elles sont définies en OCL pour vérifier l'intégrité des modèles. Une fois les composants de modèle réalisés, il est possible de les regrouper dans des bibliothèques afin de les rendre disponibles pour la phase de conception du système. Les mécanismes de gestion de ces bibliothèques de composants de modèle doivent pouvoir être intégrés aux ateliers, notamment les opérations permettant de sélectionner les composants pertinents dans ces bibliothèques.

L'étude de la dynamique d'ajout et de retrait de fonctionnalités à l'exécution semble également une piste intéressante. Il est en effet possible d'étudier ce mécanisme comme un moyen d'activation de services en fonction, par exemple, du contexte dans le cadre des applications à forte dynamicité de services. Notre approche permet, en effet, grâce à la structuration, de garantir la validité du système avant et après, l'ajout ou le

retrait de fonctionnalités. Cette propriété semble une base intéressante pour ce cadre d'utilisation. Il deviendrait ainsi possible de définir le modèle de base du système et les enrichissements possibles en fonction d'éléments de contexte : présence d'un réseau, d'une imprimante,... ou en fonction de l'état des instances. De plus, les caractéristiques de notre approche font que les services pourraient être définis sous forme de modèle générique pouvant s'appliquer à tout système présentant le modèle requis et non par rapport à un système particulier.

Plusieurs voies de recherches sont également à étudier au niveau de l'approche de modélisation elle-même. Dans notre approche, un modèle fourni est conforme à un modèle requis s'il présente la même structure. Mais le modèle fourni peut contenir d'autres éléments et être lui-même issu d'un modèle plus complexe. L'utilisation conjointe des mécanismes d'héritage et d'enrichissement fonctionnel, par exemple, soulève de nombreuses questions. L'enrichissement d'une classe provoque t'il l'enrichissement de ses sous-classes ? Cela ne risque t'il pas d'entraîner des conflits ? Est-il possible d'utiliser un modèle avec héritage en tant que paramètre d'un composant de modèle ? Concernant la structure des modèles, peut-on dire que deux modèles avec même structure mais avec des cardinalités différentes sur les associations, sont de même type ou que l'un est sous-type de l'autre ? Ou encore, un modèle ayant la même structure qu'un autre, mais dont les types de ses éléments (classes, attributs,...) sont sous-types des éléments de ce dernier, en est-il lui-même un sous-type ? Cette notion de conformité peut être considérée comme un premier niveau de typage de modèles. Existe t'il une (ou des) notion(s) de type et de sous-type de modèle ?

L'étude du typage de modèle est une voie de recherche intéressante plus générale encore [SJ05]. Elle semble, par exemple, nécessaire pour permettre l'inter-opérabilité d'approches et d'outils de manipulation de modèles en tant qu'artefacts de premier niveau. Afin de pouvoir considérer les modèles comme des artefacts manipulables, il est important de pouvoir définir leurs utilisations non pas par rapport à un modèle particulier, mais par rapport à un type de modèle. En effet, d'autres opérateurs que des opérateurs de composition, tels que des opérateurs de traduction, simplification,... pourraient bénéficier d'une notion de typage de modèle. Cette notion de typage pourrait, par exemple, servir comme critère de recherche dans des bibliothèques de modèles des ateliers tel que nous l'avons évoqué plus haut.

Enfin, notre étude porte sur la composition de modèles de structure statique. Il semble intéressant de l'étendre à la composition de modèles d'éléments dynamiques, tels que des diagrammes d'états ou de séquences [BC04, FKGS04, Egi97]. Ce travail nécessiterait une étude comparable à la nôtre sur les ordres de compositions. En effet, il semble indispensable de pouvoir garantir des propriétés sur la composition d'un ensemble de comportements dynamiques. Cette possible extension soulève de nombreuses questions. Les propriétés d'ordre démontrées pour la composition de modèles statiques s'appliquent-elles à la composition de modèles dynamiques ? Sont-elles suffisantes ? Plus généralement, cette prise en compte à un niveau modèle ne permettrait-elle pas de valider le comportement du système obtenu par composition d'un ensemble de fonctionnalités ?

En quelques années, les modèles sont passés dans l'ingénierie informatique, de

simples éléments de documentation à de réels éléments de conception et de réutilisation. Poussée par les approches tant académiques qu'industrielles (MDA puis MDE), cette tendance ne devrait que s'accentuer avec les techniques de *modélisation générique* telle que celle utilisée dans cette thèse. Les bénéfices apportés par les modèles pour la conception de systèmes ne sont plus à démontrer : maîtrise de la complexité, analyse et validation plus faciles, abstraction et indépendance technologique, réutilisation,... Mais de nombreuses voies autour de l'ingénierie des modèles sont encore à explorer pour que les systèmes puissent être entièrement conçus à partir de modèles et ainsi exploiter au mieux ces bénéfices.

Annexe A

Extraits du méta-modèle UML2

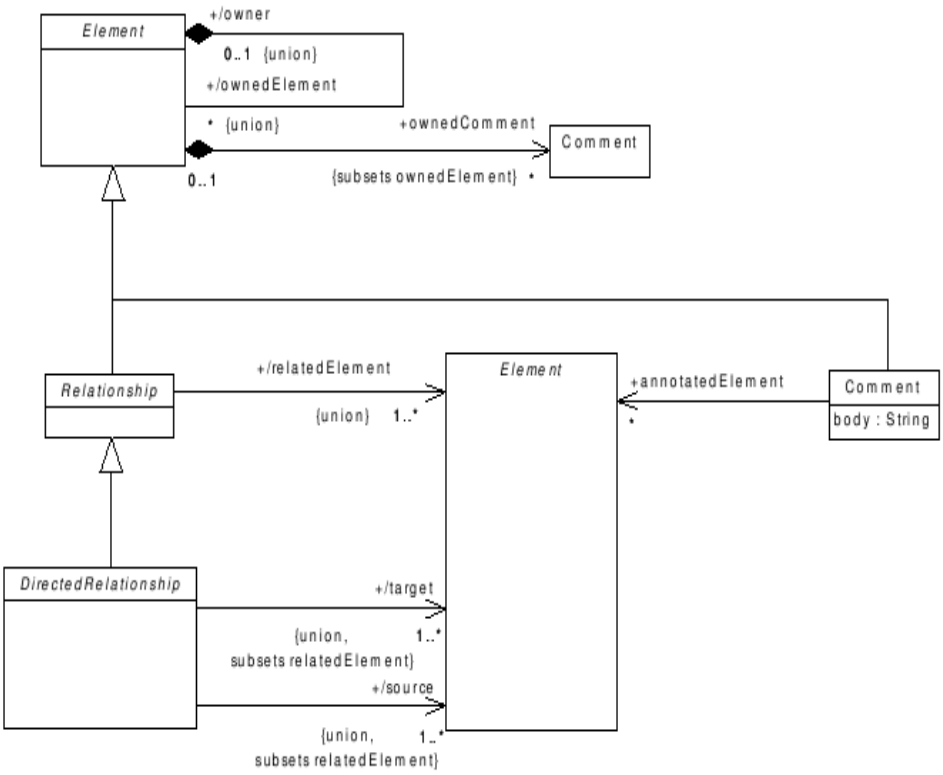

FIG. A.1 – The Root diagram of the Kernel package

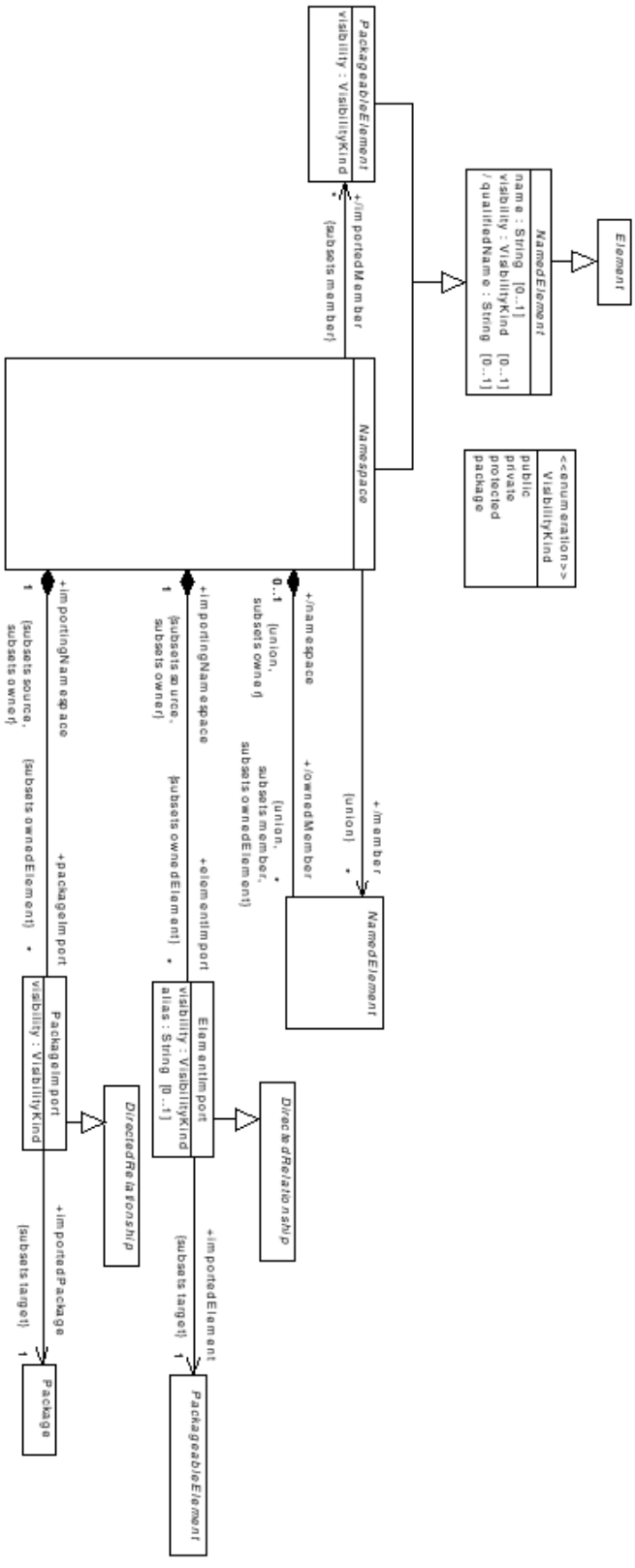

FIG. A.2 – The Namespaces diagram of the Kernel package

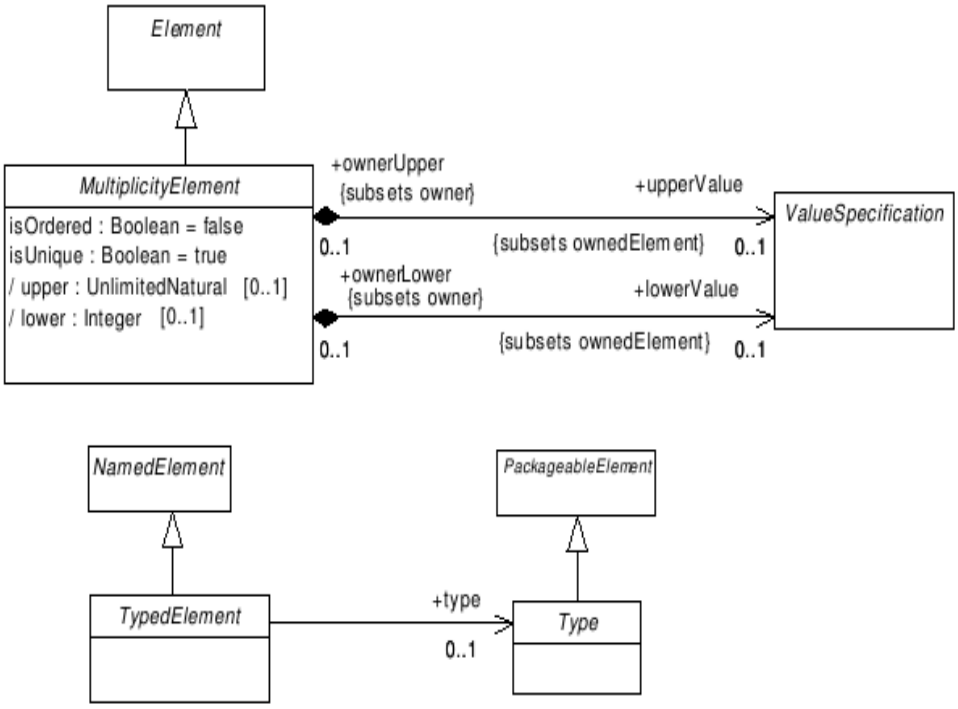

FIG. A.3 – The Multiplicities diagram of the Kernel package

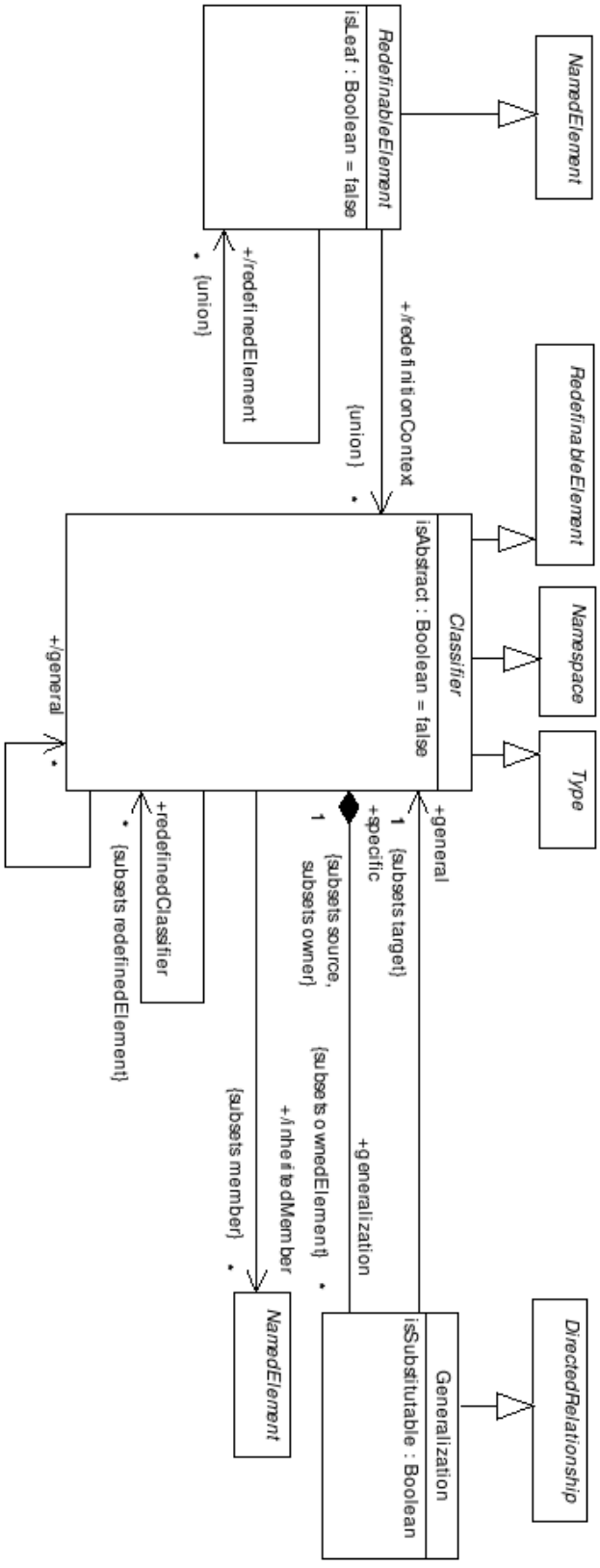

FIG. A.4 – The Classifiers diagram of the Kernel package

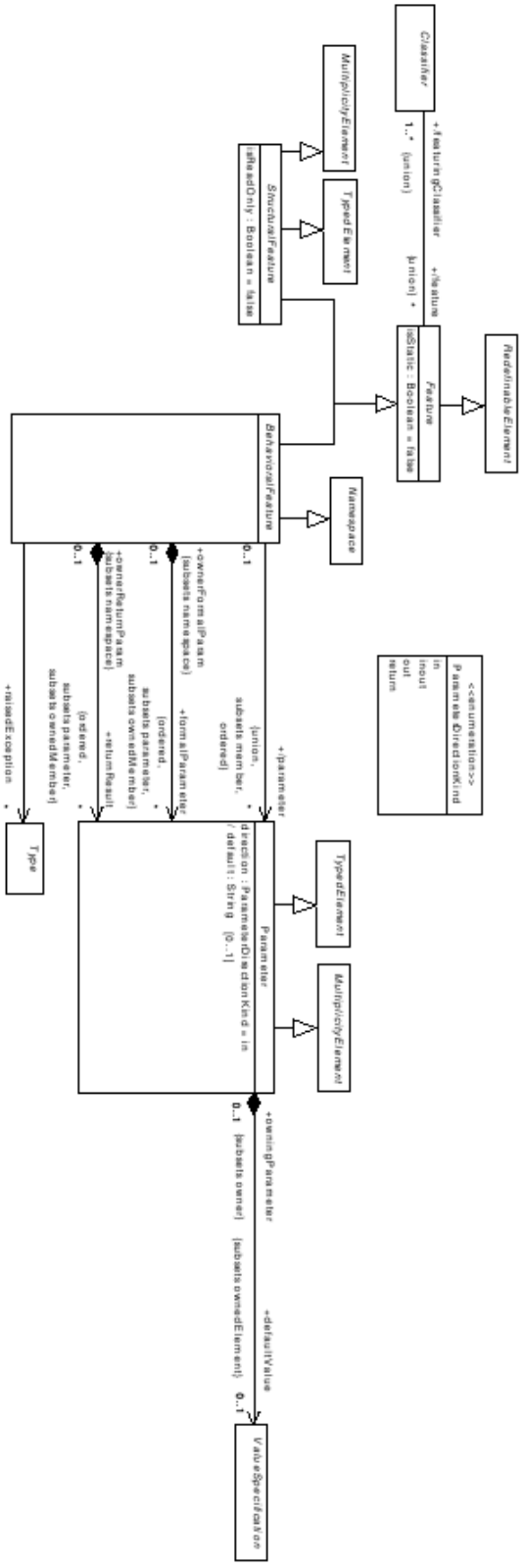

Fig. A.5 – The Features diagram of the Kernel package

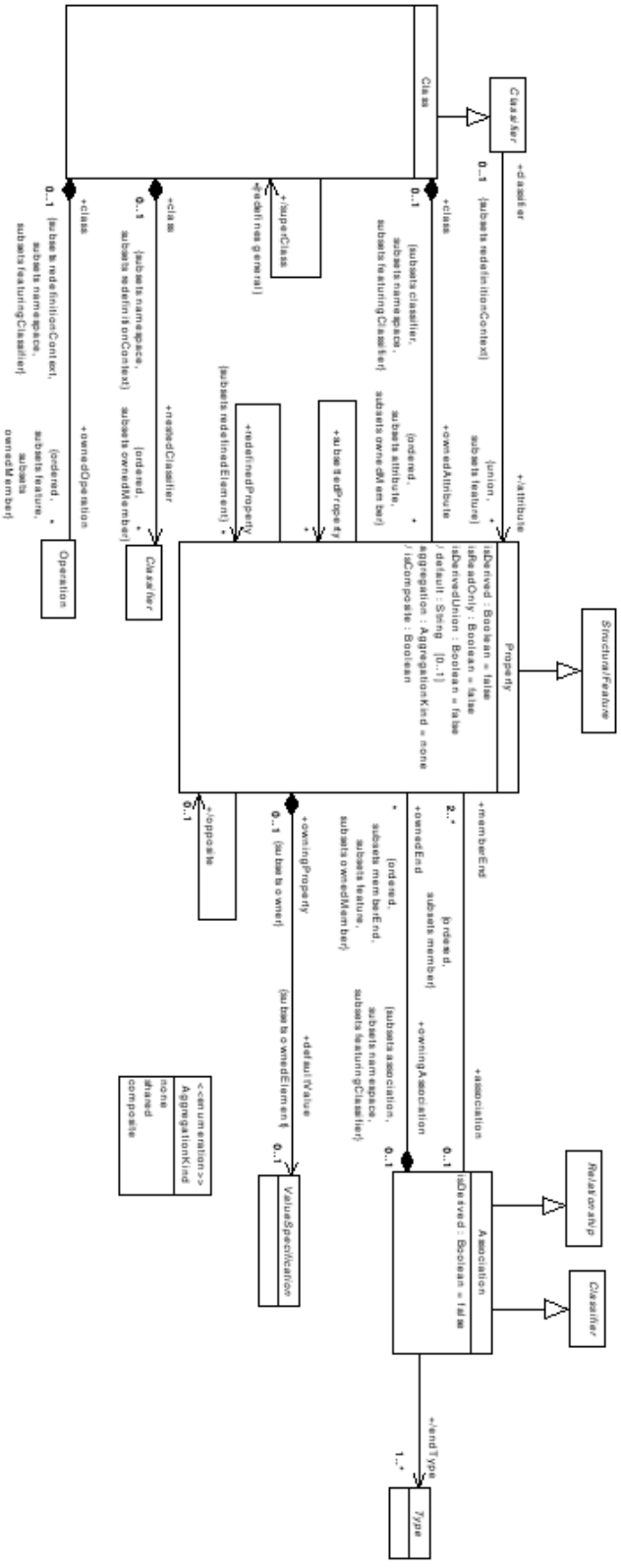

FIG. A.6 – The Classes diagram of the Kernel package

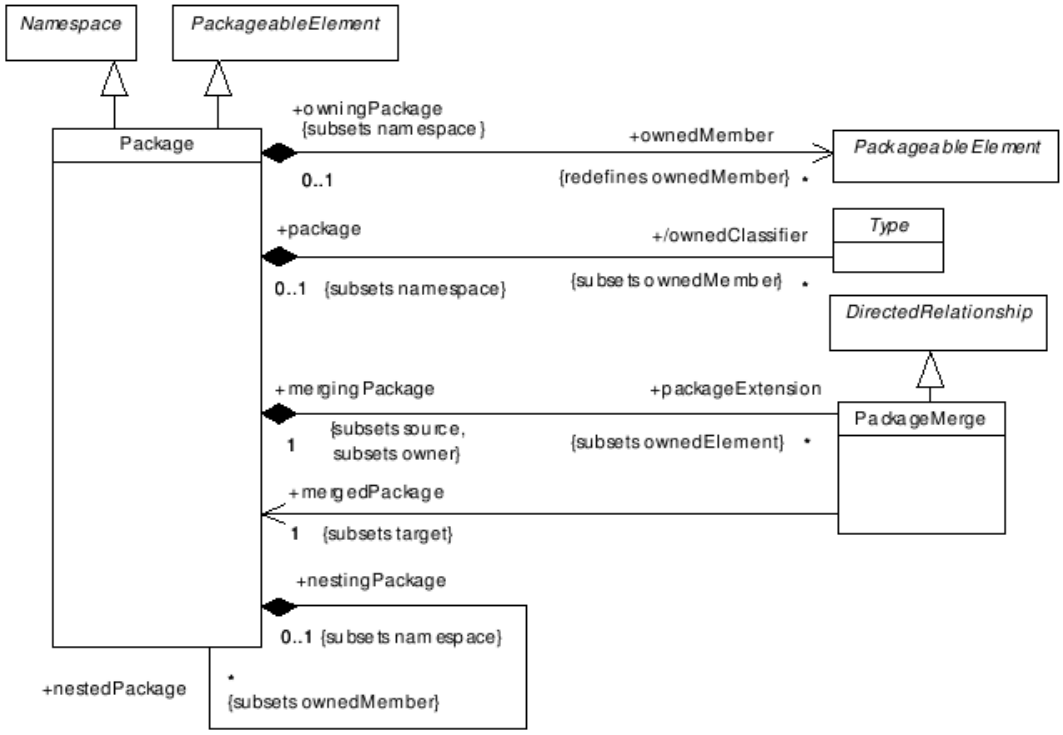

FIG. A.7 – The Packages diagram of the Kernel package

Annexe B

Descripteur XML Fractal-Vues

```
<definition name="System">
    <component name="Agence">
    <interface name="Vehicules" role="client" signature="Vehicules"/>
    <interface name="Clients" role="client" signature="Clients"/>
    <interface name="Adresse" role="server" signature="Adresse"/>
    <interface name="Nom" role="server" signature="Name"/>
    <controller desc = "basePrimitive"/>
  </component>
  <component name="Vehicule">
    <interface name="IVehicule" role="server" signature="Vehicules"/>
    <interface name="Identifiant" role="server" signature="Indentifiant"/>
    <interface name="Date" role="server" signature="Date"/>
    <interface name="Constructeur" role="server" signature="Constructeur"/>
    <interface name="Modele" role="server" signature="Modele"/>
  <controller desc = "basePrimitive"/>
  </component>
  <component name="AgenceVueRecherche">
    <interface name="Vehicules" role="client" signature="Vehicules"/>
    <interface name="Adresse" role="client" signature="Adresse"/>
    <interface name="Nom" role="client" signature="Name"/>
    <interface name="Adresse" role="server" signature="Adresse"/>
    <interface name="Nom" role="server" signature="Name"/>
    <interface name="ChercherTous" role="server" signature="ChercherTous"/>
    <controller desc = "viewPrimitive"/>
  </component>
```

```
  <component name="VehiculeVueRecherche">
    <interface name="IVehicule" role="server" signature="Vehicules"/>
    <interface name="Identifiant" role="client" signature="Indentifiant"/>
    <interface name="Date" role="client" signature="Date"/>
    <interface name="Identifiant" role="server" signature="Indentifiant"/>
    <interface name="Date" role="server" signature="Date"/>
    <interface name="ChercherParDate" role="server"
               signature="ChercherParDate"/>
    <controller desc = "viewPrimitive"/>
  </component>
  <binding client="Agence.Vehicules" server="Vehicule.IVehicule"/>

  <attachVB base="Agence" view="AgenceVue"/>
  <binding client="AgenceVueRecheche.Nom" server="Agence.Nom"/>
  <binding client="AgenceVueRecheche.Adresse" server="Agence.Adresse"/>

  <attachVB base="Vehicule" view="VehiculeVue"/>
  <binding client="VehiculeVueRecheche.Identifiant"
            server="Vehicule.Identifiant"/>
  <binding client="VehiculeVueRecheche.Date" server="Vehicule.Date"/>

  <viewBind client="AgenceVueRecherche.Vehicules"
        server="VehiculeVueRecherche.IVehicule"
        refClient="Agence.Vehicules" server="Vehicule.IVehicule" />
</definition>
```

FIG. B.1 – AgenceDescripteur.xml

Annexe C

Exemple complet IDL

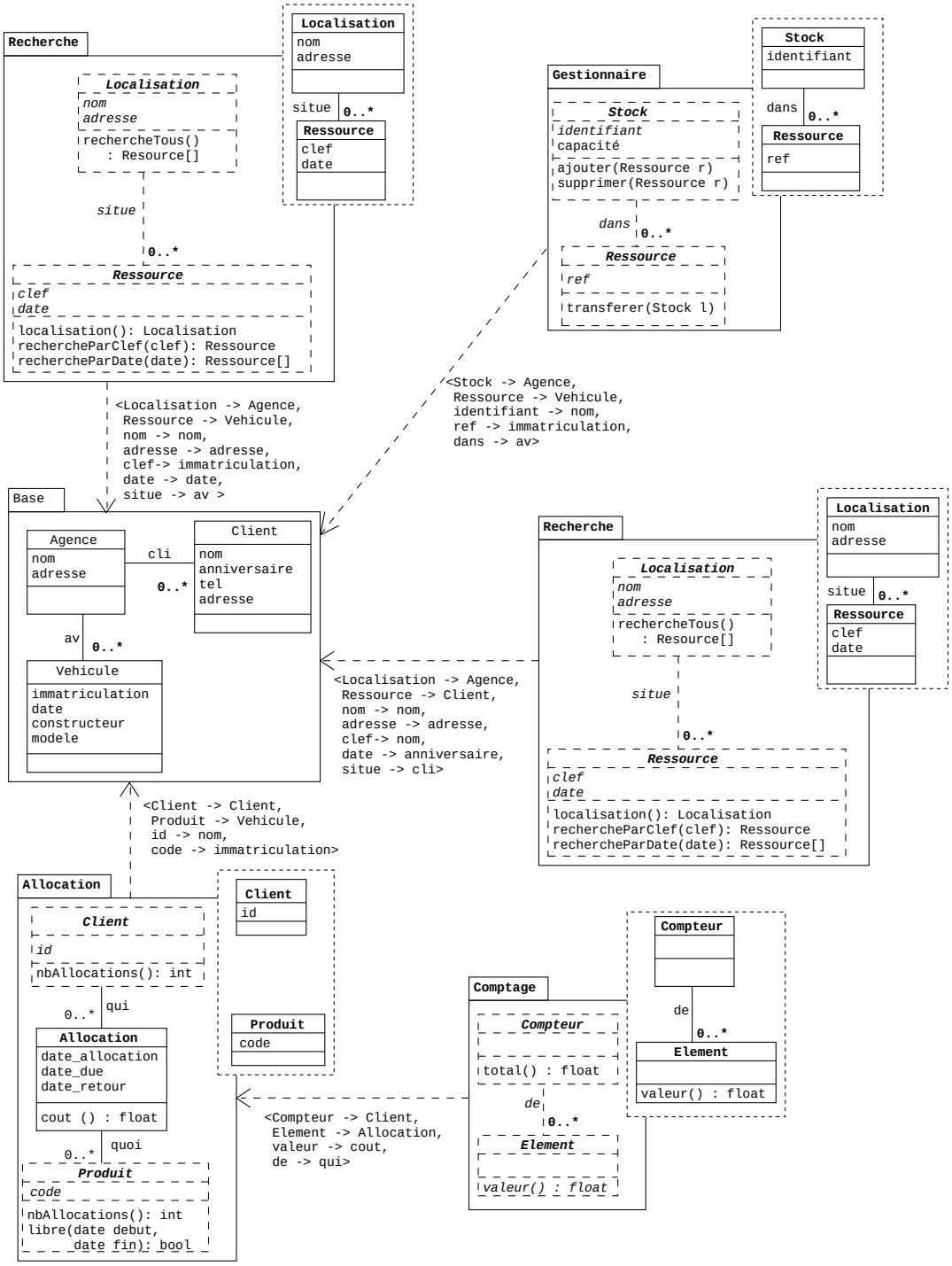

FIG. C.1 – Modèle d'assemblage d'un système de location de véhicules

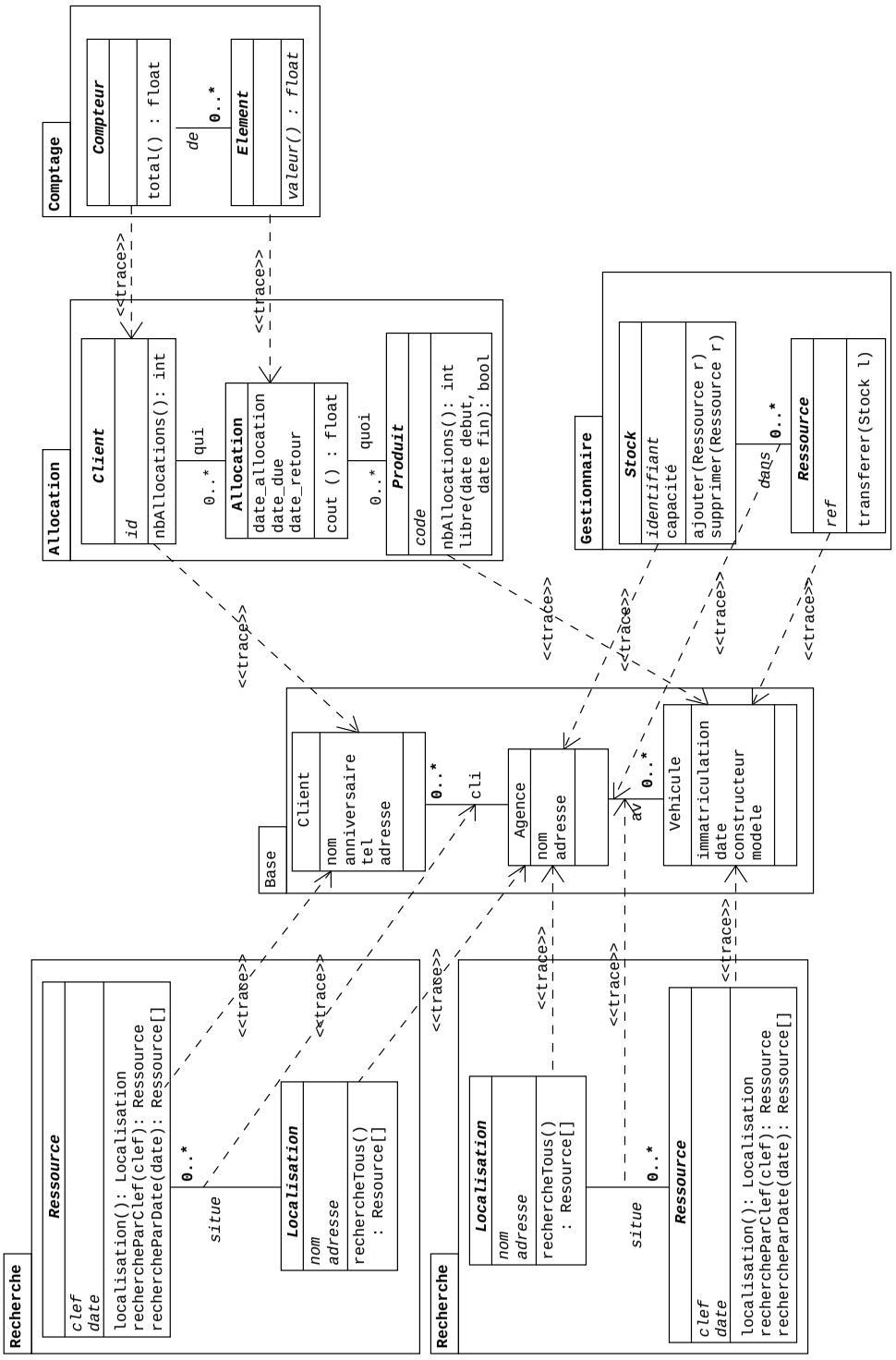

FIG. C.2 – Modèle résultat du système de location de véhicules

```
#ifndef __ENTITIES_IDL__
#define __ENTITIES_IDL__

module Entities
{
  abstract interface Base;
  abstract interface View;
  abstract interface Adaptator;
  abstract interface Manager;

  exception UnattachView {};

  abstract interface EntityPart
  {
    readonly attribute Base base;
    readonly attribute Manager manager;

    void attachView (in View view);
    void detachView (in View view) raises (UnattachView);
    boolean isSameEntity (in EntityPart part);
    View getView (in Manager viewManager) raises (UnattachView);
  };

  abstract interface Base : EntityPart
  {
    boolean removeEntity();
  };

  abstract interface View : EntityPart
  {
   readonly attribute Adaptator adaptator;
   readonly attribute EntityPart target;

   void initView (in Adaptator adpt);
   boolean acceptEntityRemoval();
   void removeView();
  };

  abstract interface Adaptator
  {
    readonly attribute EntityPart target;
    readonly attribute View view;

    void init (in View view);
  };
```

```
  abstract interface Manager
  {
  };

  abstract interface BaseManager : Base, Manager
  {
  };

  abstract interface ViewManager : View, Manager
  {
  };
};

#endif
```

FIG. C.3 – Entities.idl

```
#ifndef __AGENCE_BASE_IDL__
#define __AGENCE_BASE_IDL__

#include "Entities.idl"

module AgenceBase
{
  interface Vehicule;
  interface Agence;
  interface Client;

  typedef sequence<Vehicule> Vehicules;
  typedef sequence<Agence> Agences;
  typedef sequence<Client> Clients;

  interface AgenceBaseManager : Entities::BaseManager
  {
    Agence createAgence ();
    Vehicule createVehicule ();
    Client createClient ();

    Vehicules getVehicules ();
    Agences getAgences ();
    Clients getClients ();
  };
```

```
  interface Agence : Entities::Base
  {
    attribute string nom;
    attribute string adresse;

    readonly attribute Vehicules owns;
    void addOwnedVehicule (in Vehicule c);
    void removeOwnedVehicule (in Vehicule c);

    readonly attribute Clients rentsTo;
    void addRentsToClient (in Client c);
    void removeRentsToClient (in Client c);
  };

  interface Vehicule : Entities::Base
  {
    attribute string immatricumation;
    attribute string date;
    attribute string constructeur;
    attribute string modele;

    attribute Agence owner;
  };

  interface Client : Entities::Base
  {
    attribute string nm;
    attribute string date;
    attribute string tel;
    attribute string adresse;

    attribute Agence agence;
  };
};

#endif
```

FIG. C.4 – AgenceBase.idl

```
#ifndef __RECHERCHE_IDL__
#define __RECHERCHE_IDL__

#include "Entities.idl"

module Recherche
{
  interface Localisation;
  interface Ressource;
  abstract interface LocalisationAdaptator;
  abstract interface RessourceAdaptator;

  typedef sequence<Localisation> Localisations;
  typedef sequence<Ressource> Ressources;

  interface RechercheManager : Entities::ViewManager
  {
      Localisation createLocalisation (in LocalisationAdaptator adpt);
      Ressource createRessource (in RessourceAdaptator adpt);
  };

  interface Localisation : Entities::View
  {
    readonly attribute string nom;
    readonly attribute string adresse;

    Ressources rechercherTous ();
  };

  interface Ressource : Entities::View
  {
    readonly attribute string identifiant;
    readonly attribute string date;

    Localisation localisation();
    Ressource rechercheParClef (in string clef);
    Ressources rechercheParDate (in string date);
  };

  abstract interface RechercheManagerAdaptator : Entities::Adaptator
  {
    Localisations getLocalisations();
    Ressources getRessources();
  };

  abstract interface LocalisationAdaptator : Entities::Adaptator
  {
    readonly attribute string nom;
    readonly attribute string adresse;
    readonly attribute Ressources asso;
  };
```

```
  abstract interface RessourceAdaptator : Entities::Adaptator
  {
    readonly attribute string identifiant;
    readonly attribute string date;

    readonly attribute Localisation asso;
  };
};
```

FIG. C.5 – Recherche.idl

```
#ifndef __GESTIONNAIRE_IDL__
#define __GESTIONNAIRE_IDL__

#include "Entities.idl"

module Gestionnaire
{
  interface Stock;
  interface Ressource;
  abstract interface StockAdaptator;
  abstract interface RessourceAdaptator;

  typedef sequence<Stock> Stocks;
  typedef sequence<Ressource> Ressources;

  interface GestionnaireManager : Entities::ViewManager
  {
      Stock createLocalisation (in StockAdaptator adpt);
      Ressource createRessource (in RessourceAdaptator adpt);
  };

  interface Stock : Entities::View
  {
    readonly attribute string identifiant;
    readonly attribute string capacite;
    readonly attribute Ressources dans;

    void ajouter (in Ressource r);
    void supprimer (in Ressource r);
  };
```

```
  interface Ressource : Entities::View
  {
    readonly attribute string ref;
    readonly attribute Stock dans;

    void transferer (in Stock l);
  };

  abstract interface GestionnaireManagerAdaptator : Entities::Adaptator
  {
    Stocks getStocks();
    Ressources getRessources();
  };

  abstract interface StockAdaptator : Entities::Adaptator
  {

    readonly attribute string identifiant;
    readonly attribute string capacite;
    readonly attribute Ressources dans;
  };

  abstract interface RessourceAdaptator : Entities::Adaptator
  {
    readonly attribute string ref;
    readonly attribute Stock dans;
  };
};
```

FIG. C.6 – Gestionnaire.idl

```
#ifndef __ALLOCATION_IDL__
#define __ALLOCATION_IDL__

#include "Entities.idl"

module Allocation
{
  interface Client;
  interface Produit;
  interface Allocation;
  abstract interface ClientAdaptator;
  abstract interface ProduitAdaptator;

  typedef sequence<Client> Clients;
  typedef sequence<Produit> Produits;
  typedef sequence<Allocation> Allocations;
```

```
  interface AllocationManager : Entities::ViewManager
  {
      Client createClient (in ClientAdaptator adpt);
      Produit createProduit (in ProduitAdaptator adpt);
      Allocation createAllocation (in Client cli, in Produit p,
           in string date_allocation, in string date_due);
  };

  interface Client : Entities::View
  {
    readonly attribute string id;
    readonly attribute Allocations alloc;

    long nbAllocations ();
  };

  interface Produit : Entities::View
  {
    readonly attribute string code;
    readonly attribute Allocations alloc;

    long nbAllocations ();
    boolean libre (in string debut, in string fin);
  };

  interface Allocation : Entities::Base
  {
    readonly attribute string date_allocation;
    readonly attribute string date_due;
    attribute string date_retour;
    attribute Clients cli;
    attribute Produits pro;

    double cout ();
  };

  abstract interface AllocationManagerAdaptator : Entities::Adaptator
  {
    Clients getClients();
    Produits getProduits();
    Allocations getAllocations();
  };

  abstract interface ClientAdaptator : Entities::Adaptator
  {
    readonly attribute string id;
    readonly attribute Allocations alloc;
  };

  abstract interface ProduitAdaptator : Entities::Adaptator
  {
    readonly attribute string code;
    readonly attribute Allocations alloc;
  };
};
```

Fig. C.7 – Allocation.idl

```
#ifndef __COMPTAGE_IDL__
#define __COMPTAGE_IDL__

#include "Entities.idl"

module Comptage
{
  interface Compteur;
  interface Element;
  abstract interface CompteurAdaptator;
  abstract interface ElementAdaptator;

  typedef sequence<Compteur> Compteurs;
  typedef sequence<Element> Elements;

  interface ComptageManager : Entities::ViewManager
  {
      Compteur createCompteur (in CompteurAdaptator adpt);
      Element createElement (in ElementAdaptator adpt);
  };

  interface Compteur : Entities::View
  {
    readonly attribute Elements de;

    double total();
  };

  interface Element : Entities::View
  {
    readonly attribute Compteur cpt;

    double valeur();
  };

  abstract interface ComptageManagerAdaptator : Entities::Adaptator
  {
    Compteurs getCopteurs();
    Elements getElements();
  };

  abstract interface CompteurAdaptator : Entities::Adaptator
  {
  };

  abstract interface ElementAdaptator : Entities::Adaptator
  {
  };
};
```

Fig. C.8 – Comptage.idl

```
#ifndef __ADAPTATORS_IDL__
#define __ADAPTATORS_IDL__

#include "AgenceBase.idl"
#include "Recherche.idl"
#include "Gestionnaire.idl"
#include "Allocation.idl"
#include "Comptage.idl"

module Adaptators
{

  interface RechercheManagerAdaptator : Recherche::RechercheManagerAdaptator
  {
   readonly attribute AgenceBase::AgenceBaseManager base;
  };

  interface LocalisationAgenceAdaptator : Recherche::LocalisationAdaptator
  {
   readonly attribute AgenceBase::Agence base;
  };

  interface RessourceClientAdaptator : Recherche::RessourceAdaptator
  {
    readonly attribute AgenceBase::Client base;
  };

  interface RessourceVehiculeAdaptator : Recherche::RessourceAdaptator
  {
    readonly attribute AgenceBase::Vehicule base;
  };


  interface GestionnaireAgenceManagerAdaptator : Gestionnaire::GestionnaireManagerAdaptator
  {
   readonly attribute AgenceBase::AgenceBaseManager base;
  };

  interface StockAdaptator : Gestionnaire::StockAdaptator
  {
   readonly attribute AgenceBase::Agence base;
  };

  interface RessourceAdaptator : Gestionnaire::RessourceAdaptator
  {
    readonly attribute AgenceBase::Vehicule base;
  };
```

```
  interface AllocationAgenceManagerAdaptator : Allocation::AllocationManagerAdaptator
  {
   readonly attribute AgenceBase::AgenceBaseManager base;
  };

  interface ClientClientAdaptator : Allocation::ClientAdaptator
  {
    readonly attribute AgenceBase::Client base;
  };

  interface ProduitVehiculeAdaptator : Allocation::ProduitAdaptator
  {
    readonly attribute AgenceBase::Vehicule base;
  };

  interface ComptageAllocationManagerAdaptator : Comptage::ComptageManagerAdaptator
  {
   readonly attribute Allocation::AllocationManager base;
  };

  interface CompteurAdaptator : Comptage::CompteurAdaptator
  {
   readonly attribute Allocation::Client base;
  };

  interface ElementAdaptator : Comptage::ElementAdaptator
  {
   readonly attribute Allocation::Allocation base;
  };
};

#endif
```

FIG. C.9 – Adaptators.idl

Bibliographie

[Asp] *Aspectj Project*
http ://www.eclipse.org/aspectj/index.php.

[Bar98] Daniel Bardou. Roles, Subjects and Aspects : How do they relate ? In *ECOOP'98 Workshop Reader*, LNCS 1543, pages 418–419. Springer-Verlag, December 1998.

[BC04] E. Baniassad and S. Clarke. Theme : An approach for aspect-oriented analysis and design. In *ICSE '04 : Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society, 2004.

[BCGM04] X Blanc, O Caron, A Georgin, and A Muller. Transformation de modèles : d'un modèle abstrait aux modèles CCM et EJB. In *Proceedings of Langages, Modèles, Objets (LMO'04)*. Hermès Sciences, Mars 2004.

[BCS04] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal Component Model, version 2.0-3. http ://fractal.objectweb.org/specification/, February 2004.

[BD96] Daniel Bardou and Christophe Dony. Split Objects : a Disciplined Use of Delegation within Objects. In *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 122–137, San Jose, California, USA, October 1996. ACM Press.

[Bez01] J. Bezivin. From Object-Composition to Model-Transformation with the MDA. In *Proceedings of TOOLS-USA'2001*, Santa Barbara, August 2001.

[BGL04] Eduardo Barra, Gonzalo Génova, and Juan Llorens. An approach to Aspect Modelling with UML 2.0. In *Aspect-Oriented Modeling Workshop, AOM 2004*, Lisbon, Portugal, October 2004.

[BMP05] O. Barais, A. Muller, and N. Pessemier. Vers une séparation entités/fonctions au sein d'une architecture logicielle à base de composants. *Numéro spécial de la revue L'OBJET : Ingénierie des composants et systèmes d'information*, 11(4), 2005.

[Boo94] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[BRJ98] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[BS03] M. Basch and A. Sanchez. Incorporating Aspects into the UML. *Third International Workshop on Aspect-Oriented Modeling (AOM'03)*, March 2003.

[CCD00a] O. Caron, B. Carré, and L. Debrauwer. Contextualization of OODB Schemas in CROME. In *DEXA 2000, 11th International Conference*, volume 1873 of *Lecture Notes in Computer Sciences*, pages 135–149. Springer Verlag, September 2000.

[CCD00b] O. Caron, B. Carré, and L. Debrauwer. CromeJava : une implémentation du modèle CROME de conception par contextes pour les bases de données à objets en Java. In *Proceedings of Langages et Modèles à Objets (LMO'00)*. Hermès Sciences, January 2000.

[CCMV03] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. A Framework for Supporting Views in Component Oriented Information Systems. In *Proceedings of International Conference on Object Oriented Information Systems (OOIS'03)*, volume 2817 of *LNCS*, pages 164–178. Springer, September 2003.

[CCMV04] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. An OCL Formulation of UML 2 Template Binding. In *Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2004)*, volume 3273 of *LNCS*, pages 27–40. Springer, October 2004.

[CCMV05] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. Mise en oeuvre d'aspects fonctionnels réutilisables par adaptation. *L'objet, Programmation par Aspects*, 11(3) :105–118, January 2005.

[CEK02] T Clark, A Evans, and S Kent. A Metamodel for Package Extension with Renaming. In *The Unified Modeling Language 5th International Conference*, Proceedings LNCS 2460, pages 305–320, Dresden, Germany, September 2002.

[Cla02] S. Clarke. Extending standard UML with Model Composition Semantics. In *Science of Computer Programming, Elsevier Science*, volume 44, pages 71–100, 2002.

[COR] CORBA/IIOP Specification, http ://www.omg.org/technology/documents/formal/corba_iiop.htm.

[Cox84] Brad J. Cox. Message/object programming : An evolutionary change in programming technology. *IEEE Software*, 1(1) :50–61, 1984.

[CRS+05] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design,

http ://www.aosd-europe.net/deliverables/d11.pdf. Technical Report Deliverable D11, AOSD-Europe, may 2005.

[CS99] C. Cauvet and F. Semmak. *Génie Objet*, chapter La réutilisation dans l'ingénirie des systèmes d'information, pages 25–54. Hermès, 1999.

[CW02] S. Clarke and Robert J. Walker. Towards a standard design language for AOSD. In *AOSD '02 : Proceedings of the 1st international conference on Aspect-oriented software development*, pages 113–119, New York, NY, USA, 2002. ACM Press.

[CW05] Siobhán Clarke and Robert J. Walker. Generic Aspect-Oriented Design with Theme/UML. In Filman et al. [FECA05], pages 425–458.

[Deb98] L. Debrauwer. *Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, décembre 1998.

[DSo01] D. DSouza. Model-Driven Architecture and Integration : Opportunities and Challenges, feb 2001. www.kinetiuym.com.

[DW99] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML : The Catalysis Approach*. Addison-Wesley, 1999.

[EG00] Gregor Engels and Luuk Groenewegen. Object-oriented modeling : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 103–116. ACM Press, 2000.

[Egi97] Egil P. Andersen. *Conceptual Modeling of Objects. A Role Modeling Approach*. PhD thesis, University of Oslo, November 1997.

[EMS] *http ://www.enic.fr/people/Vanwormhoudt/modelscripting/*.

[FDE+04] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2004.

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, editors. *L'ingénirie dirigée par les modèles au-delà du MDA*. Hermès Sciences, 2006.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[FI94] William B. Frakes and Sadahiro Isoda. Success factors of systematic reuse. *IEEE Softw.*, 11(5) :14–19, 1994.

[FKGS04] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Trans. Softw. Eng.*, 30(3) :193–206, 2004.

[Fra03] David S. Frankel. *Model Driven Architecture : Applying MDA to Enterprise Computing.* Wiley, 2003.

[Gar90] David Garlan. The role of formal reusable frameworks. In *Conference proceedings on Formal methods in software development*, pages 42–44, New York, NY, USA, 1990. ACM Press.

[GHJ+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Westley Professional Computing, USA, 1995.

[GLAM+04] A. Georgin, F. Legond-Aubry, S. Matougui, N. Moteau, A. Muller, A. Tauveron, J. Thibaut, and B. Traverson. Description des assemblages et des contrats pour la conception par composants. In *Journées Composants (JC'04)*, Lille, France, March 2004.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[HDF02] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting ocl. In *Proceedings of UML*. Elsevier North-Holland, Inc., June 2002.

[HO93] William H. Harrison and Harold Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, pages 411–428. ACM Press, 1993.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1992.

[JBA] *JBoss Aspect Oriented Programming (AOP)*
http ://labs.jboss.com/portal/jbossaop/index.html.

[Ka97] G. Kiczales and al. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume LNCS 1241. Springer-Verlag, June 1997.

[Ken99] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *OOPSLA '99 : Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 353–369, New York, NY, USA, 1999. ACM Press.

[Ken02] S Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, 2002.

[KER] IRISA. *http ://www.kermeta.org/*.

[Lem98] R. Lemesle. Transformation rules based on meta-modeling. In *Proceedings Enterprise Distributed Object Computing (EDOC'98)*, San Diego, November 1998.

[MCCV03] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. In *Langages et Modèles à Objets (LMO'03)*, pages 241–255. Hermès Sciences, January 2003.

[MCCV05] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. On Some Properties of Parameterized Model Application. In *First European Conference*

on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), volume 3748 of *LNCS*, pages 130–144. Springer, November 2005.

[McI69] M. D. McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[MDA] OMG Model-Driven Architecture Home Page, http ://www.omg.org/mda.

[Mey88a] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey88b] B. Meyer. *Software reuse : emerging technology*, chapter Reusability : the Case for Object-Oriented Design, pages 201–215. IEEE Computer Society Press, 1988.

[Mic02] L. Michiel. *Enterprise Java Beans Specification v2.1*. Sun Microsystems, August 2002.

[MM01] Joaquin Miller and Jishnu Mukerji. Model Driven Architecture (MDA), July 2001. http ://www.omg.org/cgi-bin/apps/doc ?ormsc/01-07-01.pdf.

[MM03] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0.1, 2003. http ://www.omg.org/cgi-bin/apps/doc ?omg/03-06-01.pdf.

[MMJD01] H. Mili, H. Mcheick, J.Dargham, and S. Delloul. Distribution d'objets avec vues. In *Proceedings of LMO'01*, Nantes, France, January 2001.

[MOF03] MOF 2.0 Core Final Submission, 2003. http ://www.omg.org/cgi-bin/apps/doc ?ad/03-04-07.pdf.

[Mul] A. Muller. Assemblage par vues de composants logiciels. Mémoire de DEA, Laboratoire d'Informatique Fondamentale de Lille I, 2002.

[Mul02] A. Muller. La démarche MDA. Technical report, Projet RNTL Accord http ://www.infres.enst.fr/projets/accord/, 2002.

[Mul04] Alexis Muller. Reusing Functional Aspects : From Composition to Parameterization. In *Aspect-Oriented Modeling Workshop, AOM 2004*, Lisbon, Portugal, October 2004.

[MZ95] Thomas J. Mowbray and Ron Zahavi. *The essential CORBA : Systems Integration Using Distributed Objects*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Nei89] James M. Neighbors. Draco : A method for engineering reusable software systems. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability – Concepts and Models*, volume I, pages 295–319. ACM Press, 1989.

[OKK+96] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3), 1996.

[OMG97] OMG. *CORBAServices : Common Object Services Specification*. Object Management Group, Novembre 1997. OMG TC Document formal/98-07-05.

[OMG01] Object Management Group Home Page, 2001. http ://www.omg.org.

[OMG02] OMG. *CORBA 3 Specification*. Object Management Group, July 2002. http ://www.omg.org/cgi-bin/doc ?formal/02-06-33.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12) :1053–1058, 1972.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1) :1–9, 1976.

[Pel02] Mikaël Peltier. Transformation entre un profil UML et un métamodèle MOF. application du langage MTrans. In *Proceedings of Langages, Modèles, Objets (LMO'02)*. Hermès Sciences, 2002.

[Pol45] George Polya. *How to Solve It : a New Aspect of Mathematical Method*. Princeton University Press, 1945. Re-published by Penguin, 1990.

[Poo01] J. D. Poole. Model-driven architecture : Vision, standards and emerging technologies. *ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, April 2001.

[Pou95] Jeffrey S. Poulin. Populating software repositories : Incentives and domain-specific software. *Journal of Systems and Software*, 30(3), September 1995.

[PY93] J. S. Poulin and K. P. Yglesias. Experiences with a faceted classification scheme in a large reusable software library (RSL). In *17th Int'l Computer Software & Applications Conf(COMPSAC)*, pages 90–99, 1993.

[QVT05] MOF QVT final adopted specification, 2005. http ://www.omg.org/cgi-bin/apps/doc ?ptc/05-11-01.pdf.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[RWL95] T. Reenskaug, P. Wold, and O.A. Lehne. *Working with Objects : The OORam Software Engineering Method*. Prentice-Hall, Inc., 1995.

[Sa00] Richard Soley and al. Model Driven Architecture, 2000. ftp ://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

[SBF96] Steve Sparks, Kevin Benner, and Chris Faris. Managing Object-Oriented Framework Reuse. *Computer*, 29(9) :52–61, 1996.

[SG93] Murali Sitaraman and Jeff Gray. Software reuse : a context for introducing software engineering principles in a traditional computer science second course. In *TRI-Ada '93 : Proceedings of the conference on TRI-Ada '93*, pages 137–146, New York, NY, USA, 1993. ACM Press.

[SGJ00] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.

[SGS+04] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and James M. Bieman. Model composition directives. In *Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2004)*, volume 3273 of *LNCS*, pages 84–97. Springer, 2004.

[Sha95] M. Shaw. Patterns for Software Architectures. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*, pages 453–462. Addison-Wesley, 1995.

[Sio01] Siobhán Clarke. *Composition of Object-Oriented Software Design Models*. PhD thesis, Dublin City University, January 2001.

[SJ05] Jim Steel and Jean-Marc Jézéquel. Model typing for improving reuse in model-driven engineering. In *Proceedings of 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, volume 3713 of *LNCS*, pages 84–96. Springer, 2005.

[SR01] Siobhán Clarke and Robert J. Walker. Composition Patterns : An Approach to Designing Reusable Aspects. In *Proceedings of 23rd International Conference on Software Engineering (ICSE)*, May 2001.

[UML03a] OMG Unified Modeling Language Specification, 2003. http ://www.omg.org/cgi-bin/doc ?formal/03-03-01.

[UML03b] Auxiliary Constructs Templates, http ://www.omg.org/cgi-bin/apps/doc ?formal/05-07-04.pdf, pages 600-632. UML 2.0 Superstructure Specification, 2003.

[UML05a] UML 2.0 Infrastructure Specification, 2005. http ://www.omg.org/cgi-bin/apps/doc ?formal/05-07-05.pdf.

[UML05b] UML 2.0 Superstructure Specification, 2005. http ://www.omg.org/cgi-bin/apps/doc ?formal/05-07-04.pdf.

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

[Van05] Gilles Vanwormhoudt. Précision et validation de métamodèles avec emf et ocl. In *Objets, Composants et Modèles (OCM 2005)*, Berne, Suisse, mars 2005.

[VCD97] G. Vanwormhoudt, B. Carré, and L. Debrauwer. Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk. In *Proceedings of Langages, Modèles, Objets (LMO'97)*. Hermès Sciences, 1997.

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1) :7–87, 1990.

[Wil91] Alan Cameron Wills. *Formal Methods applied to Object-Oriented Programming*. PhD thesis, University of Manchester, Manchester, 1991.

[Wil96] Alan Wills. Frameworks and component-based development. In *Proceedings of International Conference on Object Oriented Information Systems (OOIS'96)*, pages 413–431, 1996.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language – Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

[WKC+01] I. Wilkie, A. King, M. Clarke, C. Weaver, , and C. Rastrick. *UML ASL Reference Guide*. Kennedy Carter, 2001.

[WPD91] S. Wartik and R. Prieto-Dfaz. Criteria for comparing domain analysis approaches. *Fourth Annual Workshop on Software Reuse*, November 1991.

Table des figures

Résumé

L'ingénierie logicielle vise à se rationaliser toujours plus et commence à atteindre des niveaux de productivité proches d'autres domaines, mécanique ou électronique par exemple.

Notre approche vise la spécification de composants métiers réutilisables et composables dans des contextes (domaines) applicatifs différents. Nous proposons d'en faire des composants de modèles génériques paramétrés eux-mêmes par des "modèles requis" et fournissant un modèle enrichi. On dépasse ainsi la notion de contrat d'assemblage de composants souvent réduite à une interface de services unitaires. La conception d'un système revient alors à assembler de tels composants par les modèles. Nous proposons pour cela un opérateur d'application de modèles paramétrés. Celui-ci permet de spécifier des assemblages à partir d'un ensemble de composants de modèles. Nous étudions des propriétés d'ordre permettant de garantir la cohérence des alternatives de composition. Ceci conduit à des règles et contraintes au niveau des modèles, afin d'assurer la cohérence de systèmes ainsi construits. Nous formulons une méta-modèlisation de l'approche par extension du méta-modèle UML2 et un ensemble de contraintes. Nous proposons également différentes stratégies de mise en œuvre, sous la forme de patron de conception, permettant de préserver, jusqu'à l'exploitation, les qualités de structuration et de généricité obtenues au niveau modèle. Des projections ont été expérimentées sur différentes plates-formes à composants.

Abstract

Software engineering aims at being rationalized always more and tends to reach levels of productivity and reuse that come near to other fields such as mechanics or electronics.

Our approach aims to specifying business components reusable in different contexts. Our idea is to introduce the notion of 'model components' parametrized by a 'required model' and that produce an 'augmented model'. Then the modeling phase can be seen as the assembly of such components by connecting provided models to required ones. Note that component ports (specified by a model) can be more sophisticated than simple interfaces of objects or software components. To support such processes, we introduce an operator (apply) to express the application of parametrized models. This operator allows to specify how to obtain a model from an existing one by the application and composition of generic ones. Alternative composition sequences of parametrized models can be elaborated to build the same system model. We formalize our approach by extending the UML2 meta-model. We propose some design patterns to preserve structuration and genericity down to exploitation phase. These patterns have been experimented in different technological platforms.