Université de Lille

ÉCOLE DOCTORALE SPI
SCIENCES POUR L'INGÉNIEUR

THÈSE

en vue d'obtenir le titre de

Docteur en Informatique

de l'Université de Lille Présentée et défendue par

Matthieu ALLON

Ingénierie Dirigée par les Modèles Basée sur les Templates

Thèse supervisée par Jean-Marc GEIB

préparée au Laboratoire CRIStAL, Équipe CARAMEL

défendue le *04 octobre 2018*

Jury :

Rapporteurs :	Pr.Olivier BARAIS	Université de Rennes 1
	Dr.Christelle URTADO	IMT Mines Alès
Président :	Pr.Laurence DUCHIEN	Université de Lille
Examinateurs :	Dr.Sébastien GÉRARD	CEA-LIST
	Dr.Gilles VANWORMHOUDT	IMT Lille Douai
Invité :	Dr.Alexis MULLER	Axellience
Directeur :	Pr.Jean-Marc GEIB	Université de Lille

Il existe deux manières de concevoir un logiciel. La première est de le rendre si simple qu'il n'y a selon toute vraisemblance aucun défaut. Et la seconde est de le concevoir de façon si complexe qu'il n'y a pas de défaut apparent. La première méthode est de loin la plus difficile.

Tony (C.A.R.) Hoare

Remerciements

Je souhaite remercier toutes les personnes m'ayant assisté, directement ou non, dans l'achèvement de ce travail. Je suis reconnaissant envers Laurence Duchien pour m'avoir fait l'honneur d'accepter d'être dans mon jury. Je remercie également Olivier Barais et Christelle Urtado d'avoir consenti à rapporter cette thèse, ainsi que Sébastien Gérard et Alexis Muller qui ont bien voulu faire partie du jury. Je les remercie tous de la curiosité dont ils ont fait preuve à l'égard de mes travaux.
Je tiens aussi à remercier Jean-Marc Geib pour l'accueil dans son équipe ainsi que l'honneur qu'il m'a fait en étant mon directeur de thèse. Je suis particulièrement reconnaissant à l'égard de Gilles Vanwormhoudt, Bernard Carré et Olivier Caron pour leur patience, soutien et encouragements, ainsi que pour toutes leurs consciencieuses relectures. Je les remercie aussi pour tout ce qu'ils ont pu m'apprendre du métier de chercheur. J'espère avoir développé grâce à eux l'honnêteté et rigueur scientifique nécessaires. Sur un plan personnel, je tiens à les remercier pour le temps et la confiance qu'ils m'ont accordés.
Merci également aux personnes de l'IMT Lille Douai de m'avoir accueilli, ainsi qu'à celles avec qui j'ai partagé un bureau. Ce fut toujours dans une très bonne ambiance. Je tiens aussi à remercier toutes celles avec qui j'ai travaillé et appris, notamment les membres de la société Axellience.
Merci enfin à mes proches, famille et amis, pour leurs encouragements.

Table des matières

1 Introduction **1**
1.1 Problématique . . . 1
1.2 Structure du document . . . 3

2 État de l'art **5**
2.1 Parties de modèles . . . 5
2.1.1 Production de parties de modèles . . . 6
2.1.1.1 Extraction . . . 6
2.1.1.2 Décomposition . . . 8
2.1.1.3 Suppression . . . 10
2.1.2 Assemblage de parties de modèles . . . 13
2.1.2.1 Import . . . 14
2.1.2.2 Fusion . . . 14
2.1.2.3 Tissage . . . 15
2.1.2.4 Composition de modèles UML : cas de la relation "merge" . . . 17
2.2 Modèles paramétrés . . . 18
2.2.1 Modèles Variables . . . 19
2.2.2 Modèles de Rôles . . . 20
2.2.3 Templates . . . 21
2.2.3.1 Templates UML . . . 22
2.2.3.2 Approches à base de templates . . . 35
2.2.3.3 Synthèse . . . 45

3 Proposition **49**
3.1 Fondements . . . 49
3.1.1 Sous-modèles . . . 49
3.1.1.1 Structure d'un modèle et sous-modèle . . . 50
3.1.1.2 Extraction de sous-modèles . . . 50
3.1.2 Aspectual Templates . . . 51
3.1.2.1 Constituants : (Sous-)modèles paramètre et spécifique . . . 51
3.1.2.2 Application d'aspectual templates . . . 52
3.2 Ingénierie Dirigée par les Modèles Basée sur les Templates . . . 59
3.2.1 Le dépôt de modèles . . . 60
3.2.2 Espace de conception des templates . . . 61
3.2.3 Espace de conception des systèmes . . . 64
3.3 Instanciation de templates . . . 69
3.3.1 Instanciation et binding . . . 69
3.3.2 Instanciation et formes de templates . . . 70
3.3.3 Instanciation partielle . . . 71
3.3.4 Bilan . . . 73
3.4 Détection et suppression de templates dans un modèle . . . 74
3.4.1 Détection d'un aspectual template dans un modèle . . . 75
3.4.2 Suppression d'un aspectual template présent dans un modèle . . . 77

3.5 Applications de templates sur des hiérarchies de modèles . . . 79
3.5.1 Validité d'une application . . . 80
3.5.1.1 Validité de l'application d'un aspectual template sur un modèle . 80
3.5.1.2 Validité d'une application et surmodèles . . . 81
3.5.1.3 Validité d'une application et sous-modèles . . . 82
3.5.1.4 Synthèse : Validité d'une application sur une hiérarchie de modèles 83
3.5.1.5 Validité de l'application de plusieurs templates sur un modèle . . 84
3.5.2 Applications d'un aspectual template sur une hiérarchie : Inclusion entre résultats . . . 87
3.5.2.1 Applications identiques ($S = S'$) . . . 87
3.5.2.2 Applications différentes ($S \neq S'$) . . . 89
3.6 Conclusion . . . 91

4 Mise en œuvre 95
4.1 Opérateurs . . . 95
4.1.1 Notation et types . . . 95
4.1.1.1 Notation . . . 95
4.1.1.2 Types et opérations primitives . . . 96
4.1.2 Opérateurs de modèles . . . 97
4.1.2.1 Extract . . . 98
4.1.2.2 Complétion d'ensemble d'éléments de modèle : Closure et Dependents . . . 99
4.1.2.3 Included . . . 100
4.1.2.4 Merge . . . 101
4.1.2.5 Synthèse des opérateurs de modèles . . . 101
4.1.3 Opérateurs de *templates* . . . 102
4.1.3.1 Apply . . . 102
4.1.3.2 Instantiate . . . 102
4.1.3.3 Promote . . . 104
4.1.3.4 Restrict . . . 106
4.1.3.5 FindCompatibleHierarchy . . . 108
4.1.3.6 SubsInference . . . 110
4.1.3.7 GetBoundSubstitutions . . . 110
4.1.3.8 Unbind . . . 111
4.1.3.9 Synthèse . . . 115
4.2 Un atelier de modélisation à base d'*aspectual templates* . . . 116
4.2.1 Présentation générale . . . 116
4.2.2 Extension du métamodèle UML . . . 118
4.2.2.1 Les contraintes *OCL* du profil . . . 120
4.2.3 Architecture . . . 121
4.2.3.1 Moteur d'*aspectual templates* . . . 122

5 Application : Modélisation d'un serveur REST d'agrégation d'informations 125
5.1 Objectif . . . 125
5.2 Contexte du scénario . . . 125
5.2.1 Critères du serveur REST . . . 125
5.2.2 Équipe de conception . . . 126

5.3 Existant . . . 126
5.3.1 Bibliothèque de patrons de conception . . . 126
5.3.2 Hiérarchies de modèles . . . 126
5.4 Étapes de modélisation . . . 126
5.5 Première partie : Concepteurs de système . . . 127
5.5.1 Accès sécurisé au serveur de ressources . . . 127
5.5.2 Accès performant au serveur . . . 133
5.5.3 Extensibilité du serveur et accès aux informations . . . 138
5.6 Seconde partie : Concepteurs de Templates . . . 144
5.6.1 Accès sécurisé au serveur de ressources . . . 144
5.6.2 Extensibilité du serveur et accès aux informations . . . 148
5.7 Bilan . . . 150

6 Conclusion et perspectives 151
6.1 Bilan . . . 151
6.2 Perspectives . . . 152

Annexes 154

A Contraintes définies pour les applications totales 156

B Applicabilité d'un *template* sur un modèle 160

C Extraits de la hiérarchie de modèles issus du dépôt 161

D Étapes de modélisation des critères du serveur REST 164

Bibliographie 167

Chapitre 1

Introduction

1.1 Problématique

Face à la complexité grandissante des systèmes, les étapes de l'ingénierie logicielle doivent être facilitées, des phases d'analyse et de conception à celles d'implémentation et de maintenance. Cette complexité est due au nombre important de fonctionnalités que les systèmes doivent inclure mais aussi aux préoccupations techniques qu'ils doivent satisfaire. Afin de répondre à cette problématique, la réutilisation logicielle est étudiée depuis longtemps en génie logiciel (Parnas et al. [86], Brooks [20], Krueger [62]). Elle consiste à créer un système à partir de logiciels existants, plutôt que d'en concevoir à partir de rien. La réutilisation est considérée comme un facteur majeur de l'amélioration de la productivité et de la qualité (Boehm [18], Mili et al. [70]). Les efforts de réutilisation ont d'abord porté sur le code. Avec des initiatives comme le bus CORBA (Common Object Request Broker Architecture - OMG [81]) et les modèles de composants, cette réutilisation est devenue plus abstraite et peut, par le biais d'interfaces, se faire indépendamment des choix technologiques utilisés pour les implantations.
Afin d'aboutir à une réutilisation indépendante de toute plateforme comme CORBA, l'OMG a par la suite proposé l'architecture MDA (Model Driven Architecture - OMG [82]). Avec MDA, le développement des systèmes logiciels est centré sur des modèles indépendants des technologies (les modèles PIM - Platform Independent Model) qui sont ensuite raffinés en modèles contenant des détails techniques (les modèles PSM - Platform Specific Model). En exprimant avec des modèles les différents niveaux d'abstraction d'un système, l'architecture MDA a permis la réutilisation dans les phases les plus en amont du développement comme l'analyse et la conception. MDE (Model Driven Engineering - Kent [57], IDM - Ingénierie Dirigée par les Modèles) a ensuite généralisé l'approche au delà des problèmes d'architecture. Cette nouvelle approche représente toutes les dimensions de l'ingénierie, ses produits et ses processus par des modèles. En plus des modèles, cette ingénierie fait appel à une seconde forme d'artéfacts. Il s'agit des transformations de modèles. Celles-ci sont utilisées pour manipuler les modèles, par exemple les modifier, les représenter dans un autre formalisme, générer du code, etc... La réutilisation de transformations permet de répondre à des tâches complexes. Elle consiste à créer des transformations à partir d'existantes, en y ajoutant ou modifiant des règles de transformations et à combiner des transformations. Typiquement, la combinaison de transformations est effectuée par leur chaînage. De nombreuses approches, comme celles de Sendall & Kozaczynski [94], Etien et al. [42] et Chechik et al. [29], ont étudié la réutilisation de transformations de modèles.
Dans cette thèse, nous nous intéressons à la réutilisation de modèles qui vise à améliorer la conception et la qualité des systèmes par l'exploitation de solutions éprouvées (France et al. [48], Overstreet et al. [85]) ainsi qu'à aider à la modélisation en équipe (Altmanninger et al. [9], Ruhroth et al. [91]). Pour atteindre ces objectifs, la réutilisation nécessite des moyens de construction, d'assemblage (Krueger [62], Ambrosio & Ruggia [11]) ainsi que de capitalisation de modèles (par exemple avec la constitution de librairies - Herrmannsdörfer & Hummel [54] ou la mise en place de dépôts de modèles - Koegel & Helming [61]). Par la suite, nous allons étudier

les questions qui se posent lorsque l'on souhaite mettre en place une ingénierie ayant pour but d'améliorer la réutilisation de modèles.
La construction de nouveaux modèles est effectuée en s'appuyant sur des projets de modélisation passés. Ces projets ne répondent pas nécessairement aux besoins du projet en cours de conception. Il est alors nécessaire de prendre dans ces modèles les morceaux ou "parties de modèles" qui intéressent le projet courant (Melnik et al. [68], Sen et al. [93], Fondement et al. [46]). Ces parties de modèles ainsi créées sont à leur tour capitalisées pour une réutilisation future. L'étude des relations d'inclusion entre modèles et donc de leurs hiérarchies (Kelsen et al. [56], Carré et al. [24]) permet de caractériser et de mieux comprendre les espaces de modèles en jeu, offrant plusieurs avantages. On peut citer la recherche de parties de modèles qui est facilitée, lorsqu'il existe un nombre important de modèles. On peut aussi souligner le fait qu'elles autorisent une meilleure compréhension des différentes parties constituants de grands modèles. Lorsque les parties de modèles ont été construites ou bien sélectionnées dans l'existant, elles sont par la suite assemblées afin de concevoir le modèle de système visé (Clarke [30], Nejati et al. [77], OMG [79]).
Dans l'ensemble des modèles, des similitudes existent souvent et des études ont été effectuées pour les capturer dans des modèles "génériques" comme les patrons de conception (Gamma et al. [50], Fowler [47]) ou les frameworks (D'souza & Wills [40]). La paramétrisation est l'un des moyens qui permet de construire de tels modèles génériques. Les modèles paramétrés sont produits en exposant un ou plusieurs de leurs constituants comme paramètres. Les templates (OMG [78], de Lara & Guerra [37], Vanwormhoudt et al. [103]) à la base de notre approche, sont une forme de modèles paramétrés. Leur réutilisation dans un contexte applicatif visé est décrite comme l'"application" d'un template au modèle de ce contexte.
Afin d'appliquer un template, il est nécessaire de valuer ses paramètres avec des constituants du modèle applicatif. Deux principales formes d'application de templates existent : l'instanciation et le tissage. Avec l'instanciation (de Lara & Guerra [37], Allon et al. [8]), les templates sont considérés comme des composants logiciels génériques, tels les templates de *C++* (Ellis & Stroustrup [41]). Cette première forme d'application permet d'obtenir l'instance d'un template qui correspond à un nouveau modèle ayant la structure du template. La seconde forme d'application qu'est le tissage consiste en l'enrichissement d'un modèle par injection des constituants d'un template, ce qui correspond à une utilisation "aspectuelle" des templates (Clarke & Walker [32], Muller [73]).
L'OMG a standardisé en UML les templates et leur application au travers de la relation "bind" (OMG [78]). Cette relation lie un template à un modèle et permet de substituer les paramètres d'un template par les constituants du modèle. La relation bind autorise aussi l'application d'un template à un autre template ("composition de templates") pour en construire d'autres. L'application partielle d'un template à un autre est également possible, se traduisant par un phénomène de propagation de paramètres.
Dans le but de renforcer sémantiquement ces templates du standard UML, les travaux de Vanwormhoudt et al. [103], un des fondements de notre approche, proposent les "aspectual templates". Cette extension ajoute des contraintes au paramétrage garantissant sa cohérence en tant que modèle ainsi que les contraintes nécessaires pour garantir cela tout au long des processus d'application.
Pour tirer profit des capacités offertes par les templates et concevoir plus efficacement des modèles de systèmes, une ingénierie dirigée par les modèles centrée sur les templates est un enjeu. L'établissement d'une telle ingénierie pose plusieurs questions, notamment en ce qui concerne les relations entre les modèles. Ces derniers ayant des objectifs et niveaux d'abstraction différents, quelles sont leurs relations et comment celles-ci peuvent-elles être utilisées par les concepteurs

afin d'analyser l'existant ? Il est utile aussi de se questionner quant aux rôles que possèdent les concepteurs et comment ces derniers interagissent entre eux dans le processus de modélisation. En se basant sur cela et les relations entre les modèles impliqués, il apparaît nécessaire, afin d'assister les concepteurs, d'étudier quels opérateurs supportent l'ingénierie. Enfin, se pose la question de l'utilisation des hiérarchies de modèles qui surviennent dans tout processus de versionnement, conception incrémentale et plus généralement de travail en équipe. En effet, comment tirer profit de cette hiérarchisation pour les applications de templates ? C'est le sujet traité dans cette thèse.

1.2 Structure du document

Dans le chapitre suivant, nous effectuons un état de l'art, dans lequel nous étudions des approches de réutilisation de parties de modèles et de modèles paramétrés. En ce qui concerne les premiers, nous montrons les différents travaux permettant de produire une partie de modèle à partir d'un modèle d'application existant. Nous présentons aussi des approches autorisant leur assemblage afin d'obtenir le modèle de système souhaité. Quant aux modèles paramétrés, nous en étudions plusieurs formes : les modèles variables, de rôles et les templates. Concernant ces derniers, ils ont été retenus pour notre approche car ils permettent de représenter des spécifications réutilisables et adaptables à des contextes applicatifs. Nous leur consacrons ainsi une section à part entière dans laquelle nous présentons le standard UML et de nombreux travaux exploitant les templates de modèles. En chapitre 3, nous exposons en premier lieu (section 3.1) les notions de sous-modèles et d'aspectual templates à la base de notre approche. Puis, nous donnons une vue générale de nos travaux en présentant l'*Ingénierie Dirigée par les Modèles Basées sur les Templates* (IDMBT) (section 3.2). Cette ingénierie est structurée autour d'un dépôt de modèles et de deux espaces de conception : celui des (aspectual) templates et celui des modèles applicatifs. Dans le premier espace ("conception pour la réutilisation"), les concepteurs définissent des templates au travers de différentes activités de modélisation. Ces templates sont par la suite placés dans le dépôt de modèles et utilisés dans le second espace de conception ("conception par la réutilisation") afin de concevoir des modèles de systèmes.
À partir de l'étude du bind UML, nous présentons, en section 3.3, l'instanciation de templates qui est une des opérations permises par l'IDMBT. Cette opération permet de générer un modèle à partir d'un template en extrayant une partie du contexte applicatif. Nous montrons comment le bind se traduit au travers de l'instanciation et comment celle-ci augmente la réutilisabilité des templates.
L'instanciation est ensuite réutilisée par deux autres opérations, présentées en section 3.4. Celles-ci supportent l'analyse de modèles relativement à des templates puisqu'elles permettent de détecter et supprimer des instances de templates au sein de modèles. Nous exposons par la suite, en section 3.5, des règles concernant l'application des *templates* sur des *hiérarchies* de modèles ainsi que les relations entre les modèles résultant de telles applications.
Dans le chapitre 4, nous présentons une mise en œuvre de l'IDMBT et des opérateurs permettant de manipuler les templates (section 4.1). Puis, nous exposons en section 3.5 la technologie réutilisable qui s'en déduit et son application dans un prototype d'atelier de modélisation. Celui-ci assiste les concepteurs de modèles via des fonctionnalités faisant appel aux opérateurs précédents. En section 5, l'ensemble de l'approche et de ses résultats sont appliqués à la conception d'une architecture de serveur REST d'agrégation d'informations. Nous présentons le scénario et sa conception basée sur un dépôt de modèles hiérarchisés et de templates. Enfin, en chapitre 6, nous concluons notre proposition et exposons des perspectives à ces travaux.

Chapitre 2

État de l'art

La réutilisation est une problématique étudiée depuis longtemps en génie logiciel et qui consiste à créer un système à partir de logiciels existants, plutôt que d'en concevoir à partir de rien. Lors de la phase de conception des systèmes et plus particulièrement dans le contexte de l'Ingénierie Dirigée par les Modèles (IDM), les artéfacts réutilisés que sont les modèles vont représenter des savoir-faire métiers ou encore des connaissances logicielles plus générales, comme les patrons de conception (Gamma et al. [50], Fowler [47]). La réutilisation de modèles vise à accélérer la conception des systèmes, à faire de la vérification au plus tôt, à améliorer leur qualité (France et al. [49], Overstreet et al. [85]), ainsi qu'à favoriser la modélisation en équipe de systèmes (Altmanninger et al. [9], Koegel & Helming [61], Ruhroth et al. [91]). Réutiliser des modèles nécessite des moyens de capitalisation (par exemple avec la construction de librairies - Herrmannsdörfer & Hummel [54]) et d'assemblage.
Dans cet état de l'art, nous allons étudier deux formes de modèles pouvant être réutilisés. Les premiers sont des modèles construits lors de la phase de conception de projets applicatifs appartenant à un domaine et représentent des parties d'applications. On cherche à réutiliser tout ou partie de ces modèles dans d'autres projets. Par exemple, dans le domaine bancaire, la même modélisation des concepts de compte et de client est utile à différents projets applicatifs. De tels modèles (section 2.1) nécessitent des moyens pour les obtenir à partir de modèles existants et les intégrer à d'autres projets applicatifs, c'est-à-dire les assembler avec d'autres modèles. La seconde forme de modèles étudiée sont les modèles paramétrés. De tels modèles se veulent plus génériques que les premiers et représentent des connaissances moins spécifiques, tels que des patrons de conception, des frameworks, les templates de C++ (Ellis & Stroustrup [41]) ou les génériques de Java (Naftalin & Wadler [76]). Lors de leur réutilisation, les modèles paramétrés sont adaptés à un domaine applicatif par la valuation de leurs paramètres. Ces modèles paramétrés font l'objet de la section 2.2. Après avoir présenté plusieurs formes de modèles paramétrés, nous nous focalisons sur le concept de templates en présentant un état de l'art détaillé.

2.1 Parties de modèles

Dans cette section, nous nous concentrons sur la notion de parties de modèles. En premier lieu, cela nécessite de produire à partir d'un modèle existant, la partie incluant les fonctionnalités que l'on souhaite réutiliser. Plusieurs approches de l'Ingénierie Dirigée par les Modèles permettent ceci, telles que l'extraction, la décomposition, la différence ou encore la suppression d'ingrédients de modèles. Nous présentons ces travaux en section 2.1.1.
Ensuite, afin de pouvoir réutiliser ces parties de modèles dans des projets en cours de conception, il est nécessaire d'avoir des moyens de les assembler. Les divers travaux que nous avons étudiés proposent des moyens d'import, de fusion ou encore de tissage de modèles. Nous présentons ces approches en section 2.1.2.

2.1.1 Production de parties de modèles

La production de parties permet aux concepteurs d'isoler d'un modèle le ou les sous-ensembles des fonctionnalités qu'ils souhaitent réutiliser par la suite pour la modélisation de leur système. Les approches présentées ci-après permettent de produire ces parties par *extraction* (sous-section 2.1.1.1), décomposition (sous-section 2.1.1.2) ou encore par suppression d'ingrédients (sous-section 2.1.1.3) dans un modèle.

2.1.1.1 Extraction

Typiquement, l'extraction de modèles consiste à produire un modèle M' à partir de la sélection d'un ensemble d'éléments dans M. L'extraction permet ainsi d'isoler la partie de M répondant à l'objectif fonctionnel souhaité.
Carré et al. [24] présente une approche définissant la structure d'un modèle, la notion de sous-modèle avec différents niveaux de relations d'inclusion entre modèles, ainsi qu'un opérateur permettant d'extraire un sous-modèle. Cette approche est utilisée comme fondement de cette thèse et est présentée dans le chapitre suivant, en section 3.1.

Extraction par sélection explicite d'éléments de modèles Plusieurs travaux proposent des opérateurs permettant d'extraire des modèles (Brunet et al. [21], Del Fabro & Bézivin [38], Bernstein [14], Melnik et al. [68], Carré et al. [24]) à partir de la sélection d'éléments. L'approche présentée par Melnik et al. [69] offre un ensemble d'opérateurs utilisables avec n'importe quel type de métamodèles ou applications de gestion de métamodèles. Un des opérateurs, nommé *Extract*, est dédié à l'extraction dans le contexte de schémas de base de données. Cet opérateur nécessite un schéma d'entrée et un ensemble d'éléments sélectionné. Il retourne une partie bien formée extraite du schéma d'entrée. Différentes contraintes sont définies pour cet opérateur : le schéma de sortie m' doit inclure tous les éléments sélectionnés dans le schéma d'entrée m, m' doit être bien formé et m doit pouvoir représenter tous les concepts de m'.
Un exemple est donné en figure 2.1. Dans cet exemple, le schéma m est formé de deux tables $PRODUCTS$ et $O\text{-}DETAILS$, avec divers attributs, de contraintes de clefs primaires $c1$ et $c3$ portant respectivement sur PID et DID, ainsi que d'une contrainte de clef étrangère $c2$ dans la table $O\text{-}DETAILS$ sur PID. L'extraction de la clef étrangère PID dans $O\text{-}DETAILS$ produit le schéma $m4$. Dû à la troisième condition citée plus haut, la table $PRODUCTS$, sa clef primaire PID et la relation entre les deux tables sont des ingrédients constituants $m4$. En effet, sans ces ingrédients, le PID dans $m4$ ne correspondrait pas à des produits existants, ce qui n'est pas possible avec le schéma d'entrée m et violerait donc la contrainte $c2$.

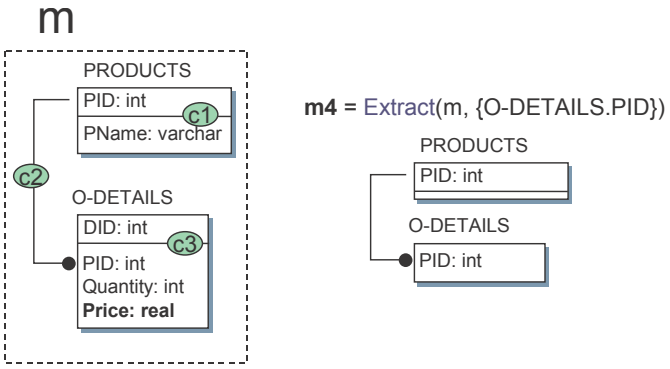
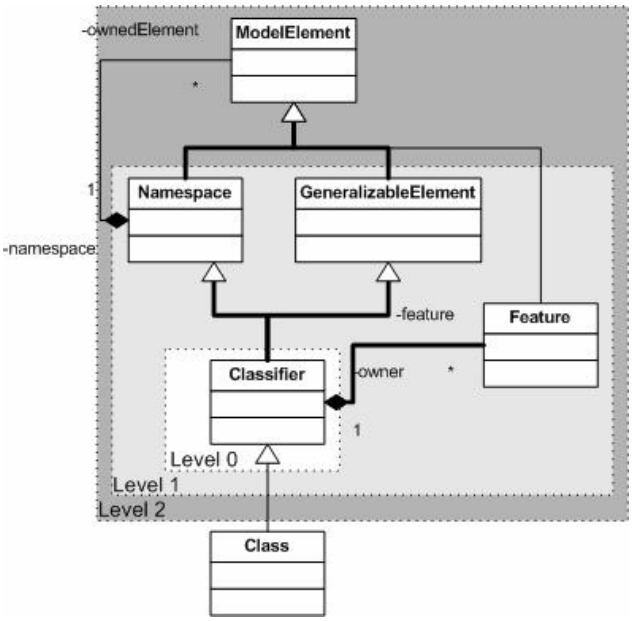


FIGURE 2.1 – Extractions à partir de m (Source : Melnik et al. [69])

Extraction par prédicats d'éléments de modèles Certaines approches ont étudié une telle extraction de parties en s'appuyant sur les contraintes structurelles des modèles, dans le contexte des modèles UML (Kagdi et al. [55], Lallchandani & Mall [64], Lano & Rahimi [65, 66]) ou encore des *feature models* (Acher et al. [1]).
Kagdi et al. [55] proposent l'extraction d'une partie à partir d'un modèle de classes UML via un critère d'extraction défini à l'aide de prédicats. Ils définissent un modèle M tel que $M = E, R, \Gamma$, avec E étant l'ensemble des éléments de M, R l'ensemble des relations et Γ, une fonction reliant des éléments de E avec une relation de R. Une partie M' de M est quant à elle spécifiée selon une fonction S_{cf}, avec en paramètre le modèle et un critère de sélection C_{cf} des éléments à extraire : $S_{cf}(M, C_{cf}) = M'$. Le critère de sélection est défini par un concepteur à l'aide de cinq prédicats. L'exemple de la figure 2.2 illustre l'utilisation de ces derniers (partie droite) afin d'extraire une partie d'un modèle de classes. Les différents *levels* (rectangles gris) représentent chaque itération de l'algorithme du parcours du modèle selon ces prédicats :

- Le prédicat $P_I(M)$ permet de définir l'ensemble initial (*Level 0*) des éléments sélectionnés. Ici, les éléments à partir desquels va s'effectuer le parcours du modèle doivent porter le nom de *Classifier*. En figure 2.2, un seul élément porte ce nom,



$P_I(M) := e \supseteq \{(name, \text{" Classifier"})\}$
$P_S(M) := T$
$P_D(M) := r \supseteq [R1 \vee R2]$
$R1 = \{(type, \text{"Generalization"})\}$
$R2 = \{(type, \text{"Association"})\}$
$T(M) := F$
$B(M) := (\forall r \supseteq R1) \vee$
$(\forall r \supseteq R2 \wedge (\forall e\ \exists e' \in E\ |path(e, e', r)| \leq 1))$

FIGURE 2.2 – Partie d'un modèle de classes UML
selon les prédicats définis (Source : Kagdi et al. [55])

- $P_S(M)$ représente une condition de sélection des éléments lors des itérations suivantes du parcours. Ici, puisqu'elle est définie à vrai, n'importe quel type d'éléments sera sélectionné,
- $P_D(M)$, les types de relations entre éléments utilisées pour parcourir le modèle. Dans l'exemple, il s'agit de *Generalization* ou d'*Association*,
- $T(M)$ permet d'indiquer une condition d'arrêt du parcours. En figure 2.2, comme ce prédicat est défini à faux, aucune condition d'arrêt n'est spécifiée,
- $B(M)$ définit la longueur de chemin maximum ou minimum entre chaque élément, pour un type de relations entre ces derniers. L'exemple ne définit aucune longueur particulière pour les relations de type *Generalization* (level 2) mais une longueur <= 1 pour celles de type *Association* (level 1).

Extraction guidée par les métamodèles Blouin et al. [17] propose une approche permettant d'extraire une partie de modèle par rapport au métamodèle d'un modèle donné. L'objectif est de réutiliser, dans un modèle, un sous-ensemble des fonctionnalités d'un modèle lié à un domaine particulier.
L'approche permet d'effectuer de telles extractions pour tout métamodèle compatible avec le méta-métamodèle MOF (OMG [84], plus précisement, son implémentation : Ecore (Steinberg et al. [96])). Le langage utilisé pour construire les extracteurs de modèles est nommé *Kompren* : tous ses concepts sont définis au sein d'un métamodèle. Il s'agit de $MSMM$, dans le haut de la figure 2.3. Ce métamodèle utilise le langage *Kermeta* (Chauvel & Fleurey [28]) pour définir le comportement des extracteurs. Via ce $MSMM$, un expert du domaine va donc définir un extracteur de modèle (MSM) référençant un sous-ensemble des ingrédients du métamodèle donné en entrée. Ainsi, toutes les instances de ces ingrédients dans un modèle d'entrée seront sélectionnés pour extraction. Une fonction d'extraction est générée à partir du MSM, permettant à l'utilisateur du domaine de réduire l'espace des instances sélectionnées pour extraction. Enfin la partie est produite à partir de cette fonction.

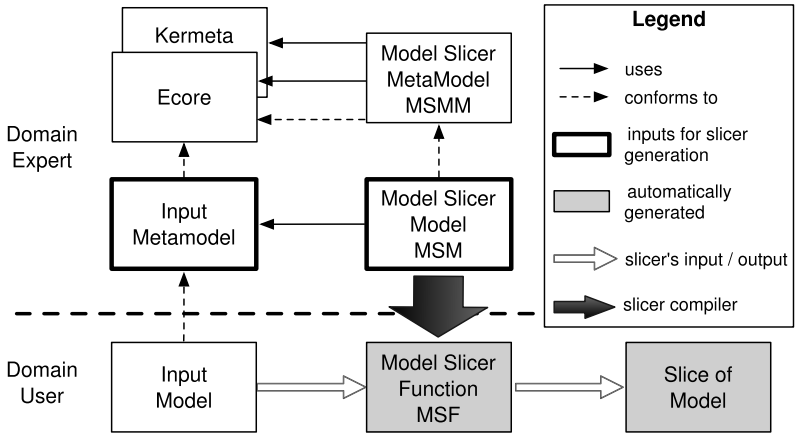


FIGURE 2.3 – Définition d'extracteurs de modèles avec *Kompren* (Source : Blouin et al. [17])

Proche de l'extraction de parties de modèles, certains travaux ont étudiés la possibilité d'extraire des parties de métamodèles, tels que l'approche décrite dans Bae et al. [12]. Les auteurs y étudient la décomposition du métamodèle UML en métamodèles plus petits selon les types de diagrammes UML. Il est à noter que les approches présentées plus haut pour l'extraction de parties de modèles sont aussi utilisables pour l'extraction de parties de métamodèles.

2.1.1.2 Décomposition

Plusieurs approches (Kelsen et al. [56], Fondement et al. [46]) proposent de produire des parties de modèles par décomposition. Celle-ci consiste à produire un ensemble de parties de modèles à partir d'un modèle M. À la différence de l'extraction étudiée précédemment, où la partie obtenue ne représente qu'une partie de M, cet ensemble de parties couvre la totalité de M. Ces parties, de par leur réutilisation, permettent ainsi la modification de la totalité d'un modèle de système. Dans l'approche de Kelsen et al. [56], les auteurs proposent une formalisation de telles parties, qu'ils nomment sous-modèles. Ces sous-modèles se conforment au même métamodèle que le modèle dont ils sont extraits. Cette conformité est assurée par l'inclusion des éléments entre

le modèle et ses sous-modèles, ainsi que plusieurs conditions liées au métamodèle, telles que la compatibilité de types et les multiplicités. Ce qui a motivé les auteurs pour cette notion de sous-modèles est la possibilité de voir et manipuler les sous-modèles avec les mêmes outils que ceux utilisés pour le modèle. À partir de cette notion de sous-modèles, les auteurs proposent un algorithme générant un treillis de sous-modèles à partir du modèle visé. Ce treillis est construit en s'appuyant sur l'identification des *relations détachables* du métamodèle, c'est-à-dire celles ayant une multiplicité débutant par 0 (*e.g. [0..*] ou [0..1]*).
Dans l'exemple de la figure 2.4, un modèle est présenté dans la partie gauche, sous forme de graphe. On souhaite extraire différentes parties du modèle, tout en respectant les contraintes du métamodèle. Tout d'abord, les liens fragmentables sont ignorés, *i.e.* ceux qui selon le métamodèle peuvent être ignorés (e.g. multiplicité de *[0..*]*). Il s'agit de ceux sur lesquels sont tracés deux lignes obliques. Puis, les nœuds faisant partie d'une *composante fortement connexe* (SCC[1]), c'est-à-dire ceux où chaque nœud est accessible par un autre nœud du sous-graphe, sont définis comme faisant partie d'un seul sous-modèle (nœuds *1 à 6* en *scc4*). Les sous-modèles obtenus sont ceux présentés dans la partie droite de la figure 2.4.

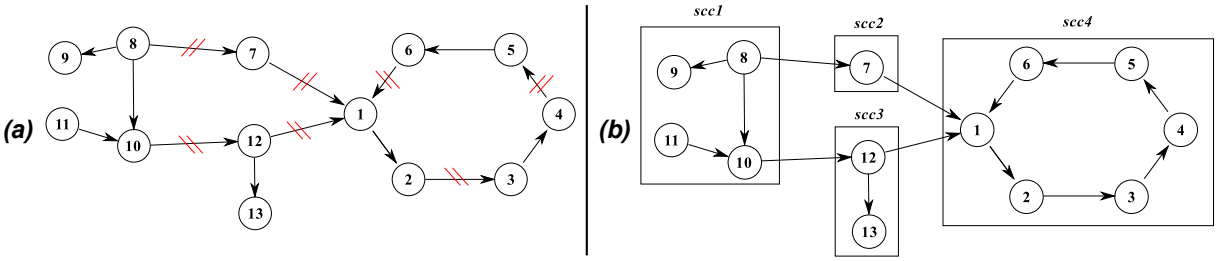


FIGURE 2.4 – Modèle sous forme de graphe et
découpage selon les *SCC* (Source : Kelsen et al. [56])

L'objectif d'un tel découpage est d'obtenir un graphe acyclique orienté dont tous les arcs sont des liens fragmentables selon les contraintes du métamodèle. La représentation d'un tel graphe est le treillis présenté en figure 2.5. Chaque sous-modèle possible est déterminé selon les liens de dépendances, représentés par les liens du graphe acyclique et lié à un modèle par la relation *"is a sub-model of"*. Par exemple, *scc*2 dépends de *scc*4 : il est donc possible de définir un modèle composé de *scc*2 et *scc*4, alors sous-modèle du modèle composé de *scc*2, *scc*3 et *scc*4.

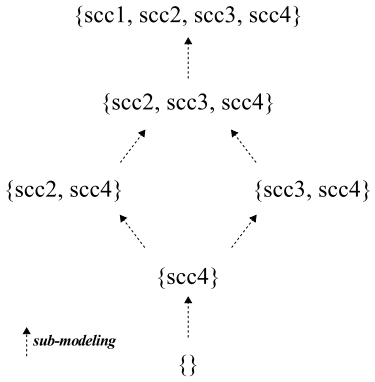


FIGURE 2.5 – Treillis de sous-modèles du modèle 2.4(a)
selon le graphe acyclique orienté de 2.4(b) (Source : Kelsen et al. [56])

1. SCC : *Strongly Connected Components*

Fondement et al. [46] présentent une nouvelle relation *unmerge* afin de diminuer la taille des grands (méta)modèles, pouvant devenir de plus en plus complexes au travers de techniques telles que les profils UML ou autres mécanismes d'extension comme le *merge* d'UML. Ceci conduit à des situations de *sur-spécification*, c'est-à-dire des situations où les métamodèles possèdent des constituants sans intérêt dans certains contextes. Afin de *décomposer* de tels (méta)modèles, la relation *unmerge* proposée fonctionne de la façon suivante : les éléments à supprimer sont copiés dans un autre modèle, avec une étape d'identification des éléments basée sur les noms, leurs méta-classes et leurs conteneurs. Le modèle contenant les éléments à supprimer est relié à l'autre avec la relation *unmerge*. De plus, les contraintes structurelles entre les éléments selon leur métamodèle, sont respectées, afin de générer des modèles bien formés.
Les figures 2.6 et 2.7 présentent quatre situations où cette relation est utilisée. En *(a)*, le paquetage P à supprimer est copié dans le modèle de droite. En exécutant l'algorithme de *décomposition* lié à la relation *unmerge*, la classe C est elle aussi supprimée du modèle de gauche, puisqu'elle est contenue dans le paquetage P.

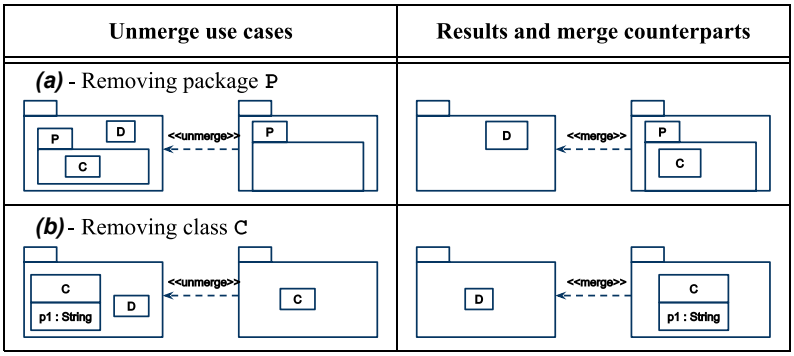


FIGURE 2.6 – Utilisation de la relation *unmerge* (Source : Fondement et al. [46])

Un autre exemple est celui de la situation *(d)*, où l'attribut $p1$ à supprimer est ajouté avec la classe C au modèle de droite afin d'éviter, après *décomposition*, que le modèle de droite soit mal-formé, c'est-à-dire avec l'attribut $p1$ sans aucune classe contenante.

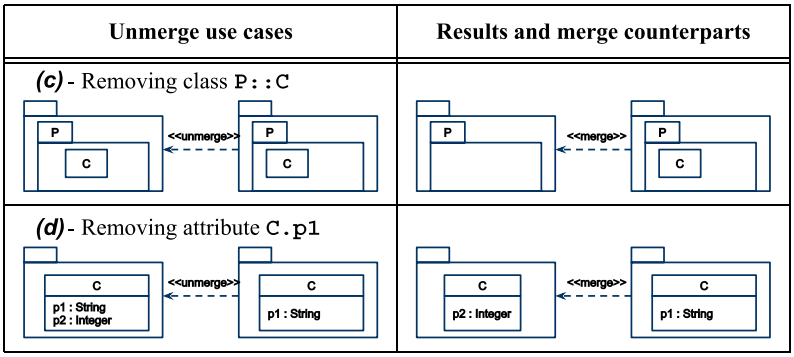


FIGURE 2.7 – Utilisation de la relation *unmerge* (Source : Fondement et al. [46])

2.1.1.3 Suppression

Contrairement à l'extraction, où la partie d'un modèle M dépend de la sélection d'éléments dans M, la suppression retire un modèle M' de M afin de produire la partie. La suppression de

modèles permet ainsi d'obtenir un modèle après avoir spécifié la partie incluant les fonctionnalités à ignorer (M'), là où l'extraction permet de préciser les fonctionnalités visées. Plusieurs travaux (Melnik et al. [68], Bernstein [14], Alanen & Porres [4], Brunet et al. [21], Del Fabro & Bézivin [38]) proposent des opérateurs supprimant des ingrédients de modèles.
Melnik et al. [68] présentent un opérateur de différence de modèles basé sur un *mapping* entre deux modèles m et m'. La partie de modèle produite inclut tous les ingrédients de m qui ne sont pas considérés, via ce mapping, comme étant identiques dans m'. La figure 2.8 illustre l'utilisation de cet opérateur. Les ingrédients $y2$ et $y3$ de m sont tous deux mappés (*map*) à $z2$ et $z3$ dans m' et ne peuvent être distingués l'un de l'autre. C'est pourquoi ils sont tous deux représentés dans m_d par respectivement $d1$ et $d2$. Quant à $y6$, puisqu'il n'est mappé à aucun des ingrédients de m', il est représenté par $d3$ dans m_d.

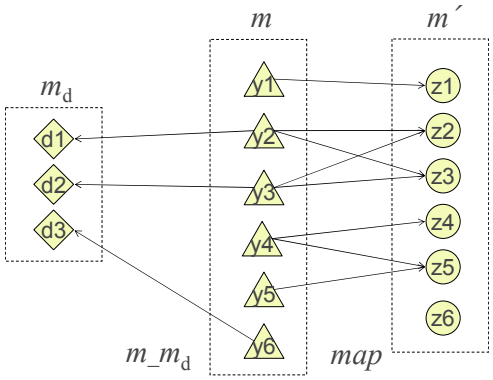


FIGURE 2.8 – Différence entre m et m' selon *map* (Source : Melnik et al. [68])

Alanen & Porres [4] présentent une approche dans le cadre d'un système de contrôle de versions de modèles conformes au métamodèle MOF ([84]). L'objectif de l'approche est de permettre de prendre en compte les situations de modélisation en équipe où des fonctionnalités sont non seulement ajoutées par un modeleur mais où certaines sont aussi supprimées par d'autres. Par exemple, dans la partie gauche de la figure 2.9, la suppression de la classe B par l'un des modeleurs dans $M1$ est prise en compte dans le résultat de l'union des deux modèles ($M3$). Les auteurs proposent d'utiliser la différence de modèles afin de déterminer ce qu'ils nomment des éléments *négatifs*, c'est-à-dire ceux à retirer lors de l'union de deux modèles, comme illustrée dans la partie droite de la figure 2.9, pour la suppression de la classe B. La différence entre deux modèles (Δ) correspond à une liste d'opérations (ajout, suppression, etc...) devant être effectuées sur un modèle afin d'obtenir le résultat. En figure 2.9, le Δ entre M et $M1$ est la suppression de B, c'est-à-dire : $\Delta_1 = M1 - M2 = del(B, Class)$. Ici, la classe est référencée par son nom mais les auteurs précisent que les éléments sont identifiés par un identifiant unique. Une fois les Δ déterminés entre modèles, l'union de deux modèles peut être effectuée en se basant sur ces différences. Comme présenté en figure 2.9, le modèle final $M3$ est obtenu en déterminant les différences entre $M1$ et $M2$, puis en effectuant une union de l'inverse de ces différences sur M (l'inverse des opérations de différences étant notées $\tilde{\Delta}_x$). L'inverse des opérations composant un Δ est défini par les auteurs pour effectuer une telle union.
Un *mapping* a d'ailleurs été défini entre opérations et opérations duales, comme exposé par la figure 2.10. Dans cet extrait, e représente l'identifiant de l'élément et t représente son type. Les auteurs considèrent aussi les situations plus complexes où de telles unions génèrent des conflits. Pour rappel (cf. section 2.1.1.1), les auteurs de Melnik et al. [69] ont défini plusieurs opérateurs

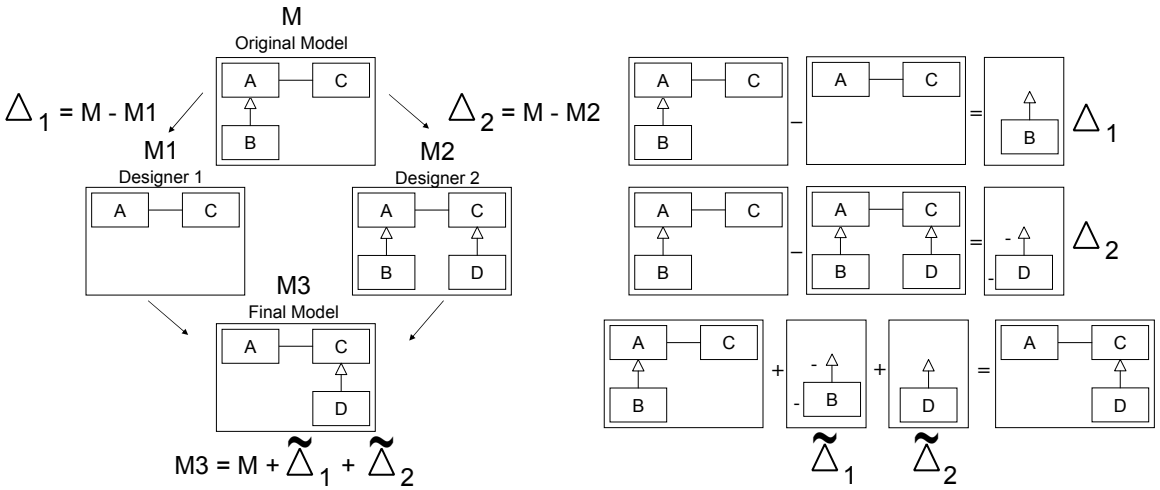


FIGURE 2.9 – Exemple d'union de deux versions de modèles (Source : Alanen & Porres [4])

Operation O	Dual operation $\tilde{O}$
new*(e , t)*	del*(e , t)*
del*(e , t)*	new*(e , t)*

FIGURE 2.10 – Extrait de l'ensemble des opérations et opérations duales (Source : Alanen & Porres [4])

génériques, *i.e.* utilisables avec n'importe quels métamodèles, dans le contexte de schémas de base de données. Parmi ces opérateurs, celui de suppression, *Delete*, prend en entrée un schéma et un ensemble d'éléments. Tout comme pour l'opérateur d'extraction (cf. section 2.1.1.1), *Delete* obéit aux contraintes suivantes : le schéma de sortie m' doit inclure tous les éléments sélectionnés dans le schéma d'entrée m, m' doit être bien formé et m doit pouvoir représenter tous les concepts de m'.

La figure 2.11 illustre ceci : le schéma m est formé de deux tables avec divers attributs, de contraintes de clefs primaires $c1$ et $c3$ portant respectivement sur PID et DID, ainsi que d'une contrainte de clef étrangère $c2$ dans la table *O-DETAILS* sur PID. La sélection de $PRODUCTS.PID$ et des contraintes $c1$ et $c2$ ne suffit pas pour la suppression de l'attribut $PRODUCTS.PID$. En effet, l'attribut *O-DETAILS.PID* qui est une clef étrangère sur $PRODUCTS.PID$ n'est pas sélectionnée. Ainsi, la suppression seule de $PRODUCTS.PID$ étendrait l'ensemble des valeurs possibles de *O-DETAILS.PID*. Ceci violerait la troisième contrainte citée précédemment, puisque m serait alors plus contraint que m'. C'est pourquoi le schéma résultant de cette suppression, $m6$, inclut $PRODUCTS.PID$, $c1$ et $c2$ même s'ils font partie des éléments sélectionnés pour suppression.

Sen et al. [92] présente une approche générique permettant de produire un métamodèle qui est un sous-ensemble d'un métamodèle donné, tel que celui d'UML. Dans cette approche, un nouveau métamodèle est produit par la suppression d'éléments dans un métamodèle donné, tout en préservant un ensemble de types et propriétés sélectionnés et leurs dépendances. Les auteurs mettent en œuvre ce *prunning* de métamodèles (Sen et al. [93]) dans une approche ayant pour objectif de rendre réutilisable des transformations de modèles, en adaptant les métamodèles utilisés en entrée des transformations. Pour cela, leur approche est constituée de deux étapes. Tout d'abord, le découpage du métamodèle d'entrée d'une transformation afin d'obtenir le métamodèle

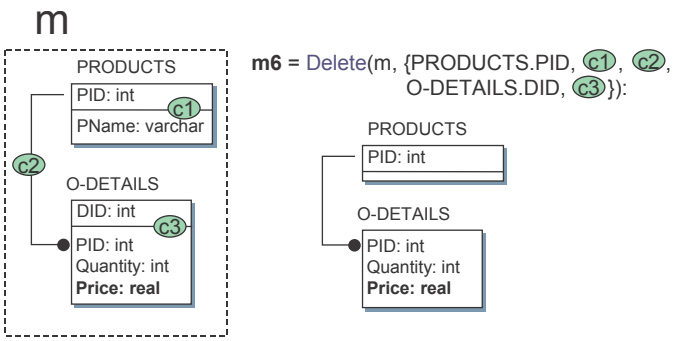


FIGURE 2.11 – Suppression d'éléments dans un schéma relationnel m (Source : Melnik et al. [68])

incluant uniquement les concepts utilisés par cette transformation. Par exemple, en figure 2.12 (points 1 et 2), le métamodèle effectif de la transformation est obtenu en découpant le métamodèle UML. Ensuite, pour pouvoir utiliser la même transformation avec d'autres métamodèles d'entrée similaires, par exemple le métamodèle de Java, les concepts utilisés par la transformation et faisant partie du métamodèle effectif sont tissés avec ce métamodèle similaire.

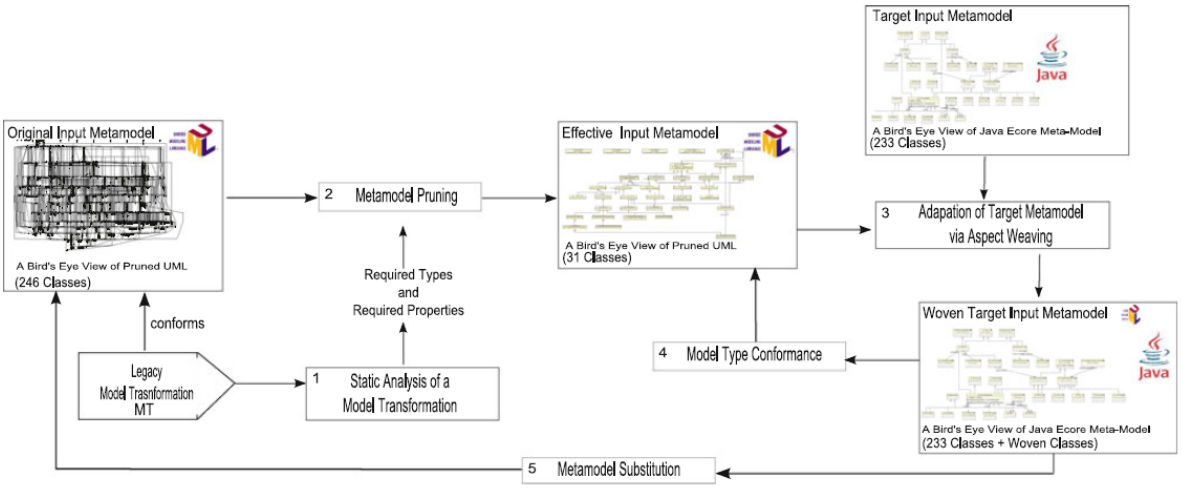


FIGURE 2.12 – Vue d'ensemble de l'approche avec les métamodèles *UML et Java* (Source : Sen et al. [93])

2.1.2 Assemblage de parties de modèles

L'assemblage de parties de modèles consiste à réutiliser des parties de modèles et à les assembler afin d'en obtenir de plus riches, en termes de fonctionnalités. Il est ainsi possible de construire de façon incrémentale des modèles de systèmes, comme par exemple avec les relations *merge* et *import* du standard UML [79]. Les approches étudiées correspondent respectivement à l'import d'éléments (sous-section 2.1.2.1), l'assemblage par fusion (sous-section 2.1.2.2) et par tissage de parties de modèles (sous-section 2.1.2.3). Le cas particulier de la relation *merge* du standard UML est présenté en sous-section 2.1.2.4.

2.1.2.1 Import

L'import consiste à référencer dans un modèle les ingrédients d'un autre modèle afin d'y avoir accès. En UML (OMG [79]), les relations *import* et *access* sont définies entre un espace de noms (*importing namespace*) et un élément de package (*packageable element*). Ces relations permettent, dans l'espace de noms, d'identifier et de référencer un élément de package par son nom. Le nom de cet élément référencé est alors ajouté à l'espace de noms. La relation *access* définit la visibilité de l'élément à *privé*. Avec cette relation, il n'est donc pas possible de propager la référence de l'élément à d'autres espaces de noms, contrairement à la relation *import*. Un exemple est donné en figure 2.13. En figure 2.13, *Program* importe *Time*. Cet élément est alors ajouté à l'espace de noms *Program* et d'autres packages peuvent le référencer à partir de *Program*, contrairement à *String* qui est référencé et ajouté à *Program* avec la relation *access*.

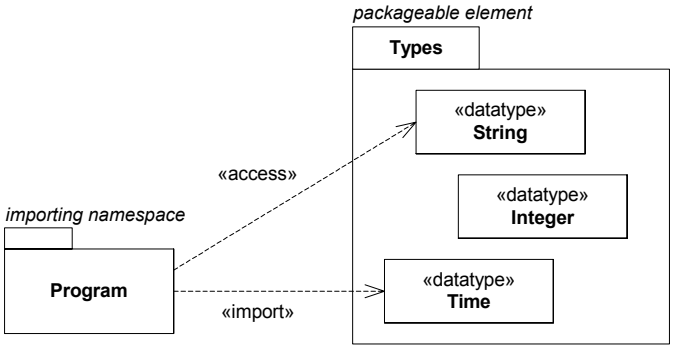


FIGURE 2.13 – Référence des éléments *Time* et *String* de *Types* par *Program* via les relations *acces* et *import* (Source : OMG [79])

Le standard permet aussi de définir les relations import et access entre un espace de noms et un package. Cela revient à référencer chacun des éléments du package avec des relations import ou access. La figure 2.14 fournit un exemple d'utilisation de ces relations entre packages. Suivant la même règle de visibilité que présentée plus haut, le package *WebShop* qui importe *ShoppingCart*, peut faire référence à des éléments de *Types* mais pas à des éléments de *Auxiliary*. Dans le standard, il est précisé que les éléments référencés ne peuvent être modifiés dans les modèles les référençant mais dans ceux à partir desquels ils ont été importés.

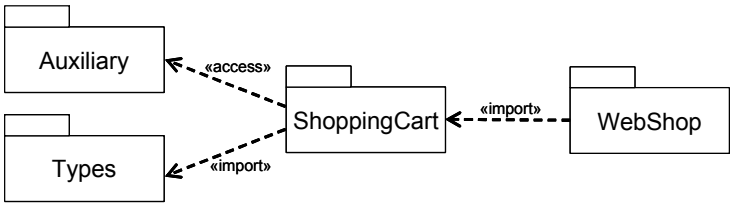


FIGURE 2.14 – Référence des éléments de *Types* par *WebShop* et *ShoppingCart* (*import*) et référence des éléments de *Auxiliary* par *ShoppingCart* (*access*)(Source : OMG [79])

2.1.2.2 Fusion

La fusion de modèles se compose de deux étapes : la définition de similarités entre les modèles à fusionner, puis la production d'un modèle possédant les éléments non communs aux modèles d'entrées et les éléments définis comme identiques lors de l'étape précédente.

Plusieurs approches (D'souza & Wills [40], Clarke [30]) présentent un opérateur de fusion de modèles avec une étape de correspondance par défaut, basé notamment sur l'identification par le nom des éléments. La méthodologie *Catalysis*, définie par D'souza & Wills [40], propose un cadre d'utilisation des objets, des *frameworks* et de la notation UML pour concevoir et réutiliser des systèmes à base de composants. Dans cette approche, les paquetages sont utilisés pour représenter les différentes coupes du système. Pour intégrer ces divers paquetages, les auteurs proposent la relation *join*. Celle-ci, basée sur la relation *import* d'UML, fusionne les éléments des paquetages en les identifiant par leur nom, par défaut.
Melnik et al. [68] et Bernstein [14] ont étudié la fusion de méta-données sous la forme de schémas relationnels et *XML*. La plateforme implémentée, *Rondo*, fournit un ensemble d'opérateurs permettant aux modeleurs de manipuler des modèles. Parmi ces opérateurs, l'opérateur *match* permet de définir des correspondances sémantiques entre des éléments de schéma similaires. Un autre opérateur, *merge*, permet quant à lui de combiner différents éléments de modèles.
Brunet et al. [21] présentent un *framework* de gestion de modèles qui définit un ensemble d'opérateurs participant à la fusion de modèles, tel les opérateurs *merge* et *match*. La fusion de modèles est décrite comme un processus prenant en entrée deux modèles et un ensemble de relations spécifiant quels éléments des deux modèles doivent être fusionnés. Le modèle de sortie contient les deux modèles d'entrée, modulo les relations définies. Suivant ceci, les auteurs définissent un opérateur *merge*.

$$merge : modèle\ X\ modèle\ X\ relations \rightarrow modèle$$

L'idée est que cet opérateur n'est pas une opération fixe, *i.e.* les relations spécifient comment les modèles doivent être fusionnés. La création de ces relations est possible via l'opérateur *match*.

$$match : modèle\ X\ modèle \rightarrow relations$$

Cet opérateur interprète donc les modèles d'entrées, recherche des similarités entre leurs éléments et définit des relations entre ces derniers. Les auteurs indiquent que cet opérateur peut très bien être utilisé dans la définition d'autres opérateurs, par exemple pour produire des relations utilisées dans le cadre de transformations de modèles.
L'approche de Nejati et al. [77] présente deux opérateurs pour les diagrammes d'états-transitions d'UML. Le premier est l'opérateur *match*. Celui-ci permet d'identifier des éléments similaires dans deux modèles distincts en se basant sur l'identification statique (*e.g.* la similarité des noms d'états) et l'identification comportementale, basée sur la sémantique du comportement des états (conditions sur les transitions entre états). L'opérateur *merge* permet quant à lui de construire un modèle incluant les éléments communs aux modèles fusionnés et ceux non-communs aux deux modèles. Ces éléments non-communs ont alors le nom de leur modèle d'origine signalé par des gardes sur les transitions.

2.1.2.3 Tissage

Contrairement à la fusion de modèles précédente, le tissage n'est pas symétrique puisqu'il enrichit un modèle de base avec les fonctionnalités d'un autre. Certaines approches (Clarke [30], Reiter et al. [90]) présentent des opérateurs permettant de tisser des modèles.
L'approche de Clarke [30] présente un mécanisme de composition de diagrammes de classes UML. Chacun des modèles composés représente différents concepts, ces derniers pouvant être présents dans plusieurs modèles (*overlapping*). Deux stratégies de composition sont présentées, au travers des deux relations *merge* et *override*. La première permet d'effectuer une fusion de modèles, tout comme les approches présentées dans la sous-section précédente. La seconde relation, nommée

override, permet quant à elle de substituer l'élément d'un modèle M par celui d'un autre modèle M' et d'enrichir M' avec les autres éléments de M.
En figure 2.15, la relation *override* est utilisée entre deux modèles. Cette relation de substitutions *override* permet de spécifier, quel élément prévaut sur l'autre. Ici, la relation part de l'opération *check*() vers $checkMax()$: cette dernière sera donc substituée par *check* dans le modèle résultant de la composition.

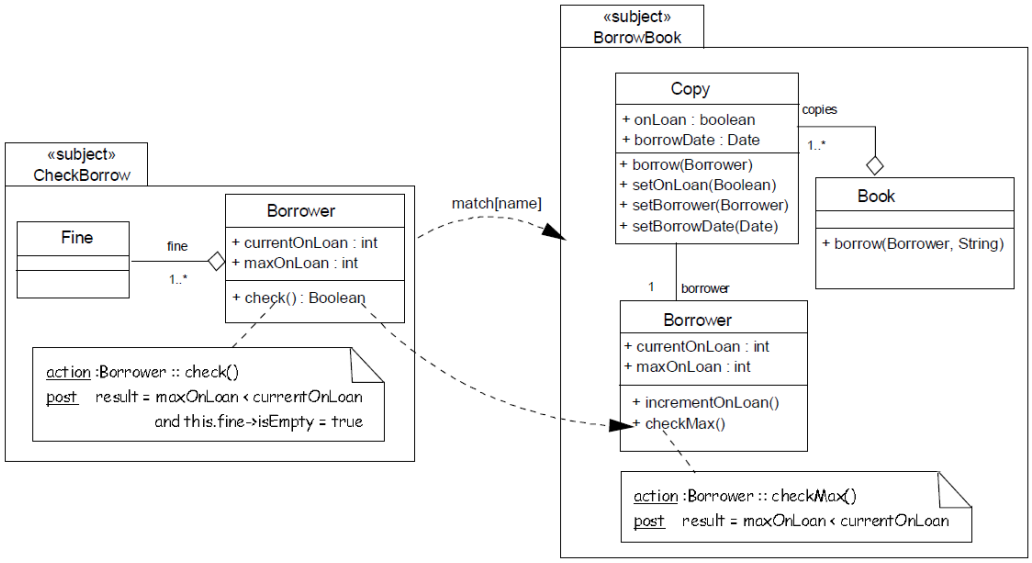


FIGURE 2.15 – Exemple d'utilisation de la relation *override* (Source : Clarke [30])

Les auteurs Reiter et al. [90] proposent quant à eux des méta-opérations permettant de spécifier l'intégration de modèles, couvrant diverses préoccupations au niveau méta. Ces méta-opérations de *weaving* et de *sewing* sont définies en utilisant un ensemble d'opérateurs primitifs. Les opérations de *sewing* permettent aussi d'intégrer plusieurs modèles mais leur permet de continuer à exister indépendamment, sans affecter leur structure. Les opérations de cette seconde catégorie permettent aussi de garder les modèles synchronisés entre eux. Les opérations de *weaving* permettent de tisser les modèles entre eux, telle l'opération *override* présentée en figure 2.16. Tout comme la relation du même nom présentée plus haut, elle permet de substituer un élément de modèle par un autre et d'enrichir un modèle avec les éléments d'un autre. Cependant, sa définition est effectuée ici au niveau métamodèle et l'identification des éléments est définie par l'utilisation de ce que les auteurs nomment des *contraintes d'intégration de modèles (MIC : Model Integration Constraints)*. Dans la figure 2.16, *overrides* est définie entre les méta-classes *Place*. Les *MIC* portant sur les attributs *id* de ces méta-classes, les instances de *Place* dans le modèle $mPetri$ et ayant des attributs *id* identiques à celles de $mMark$ sont substituées par celles-ci.
Del Fabro & Bézivin [38] présentent une mise en œuvre du *framework* de Brunet et al. [21] (cf. section 2.1.2.2 précédente) au travers de leur plateforme d'IDM et des deux notions attenantes que sont les modèles de transformations et les modèles de tissage. Leur objectif est de montrer la capacité de leur plateforme à définir des opérateurs génériques de gestion de modèles.
Les modèles de tissage sont constitués d'éléments typés permettant de mettre en relation des éléments de modèles distincts. Les modèles de transformation représentent quant à eux les transformations d'un modèle à un autre. Les opérateurs *merge* et *match* sont redéfinis de la façon suivante :

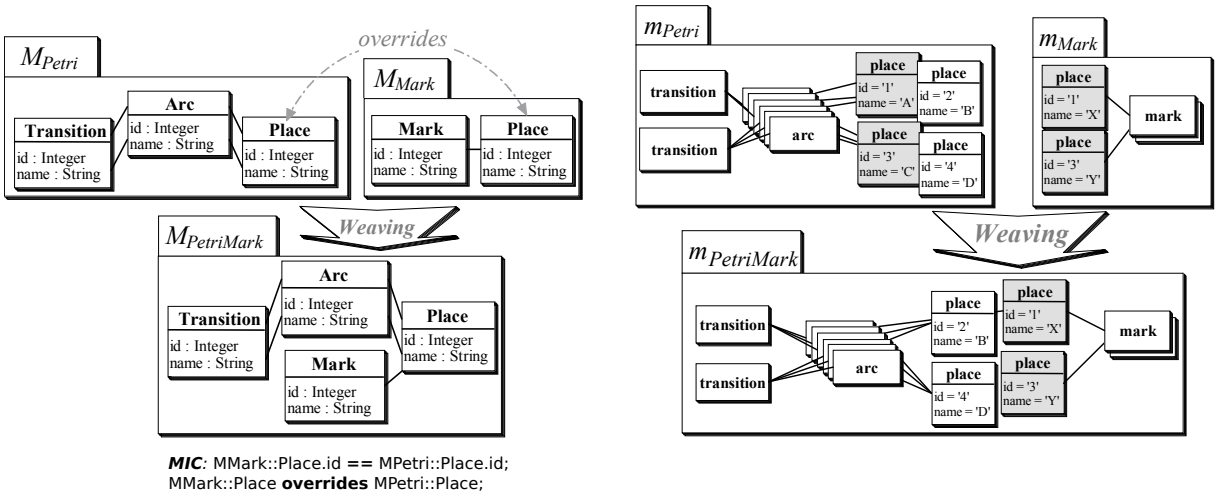


FIGURE 2.16 – Exemple d'utilisation de la relation *override* et de MIC sur l'attribut *id* (Source : Reiter et al. [90])

merge : modèle X modèle X modèle de tissage → modèle
match : modèle X modèle → modèle de tissage

2.1.2.4 Composition de modèles UML : cas de la relation "merge"

Dans le standard UML, la relation *merge* permet de fusionner deux modèles mais est assez proche de la définition du tissage donnée dans la section précédente puisqu'elle est aussi asymétrique. En effet, un des deux modèles fusionnés (le *receiving model*) est à la fois opérande et résultat du *merge* et est enrichi du second modèle (le *merged model*).
La figure 2.17 illustre ceci avec des paquetages : le *merged package* et le *receiving package* (ou plus généralement, *merged model* et *receiving model*). Le modèle résultat est quant à lui nommé *resulting package* et correspond au *receiving package* enrichis des éléments du *merged package*. Les éléments de même nom et signature sont fusionnés.

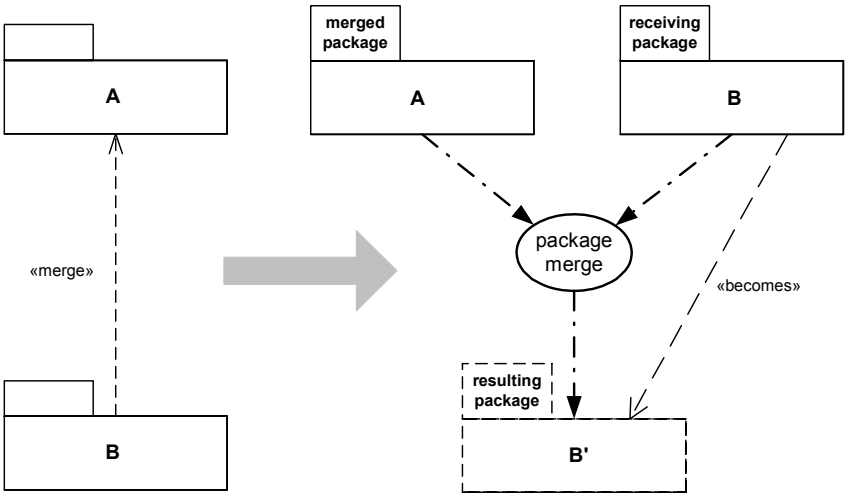


FIGURE 2.17 – Sémantique de la relation *merge* sur des paquetages (Source : OMG [79])

La figure 2.18 donne un exemple de fusion, avec les paquetages *CarAgency* et *AgencyClient*.

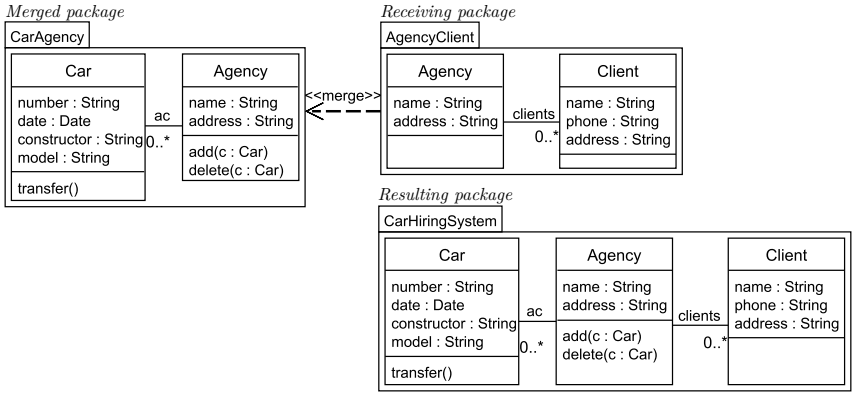


FIGURE 2.18 – Fusion des paquetages *CarAgency* et *AgencyClient*

L'étude effectuée par Dingel et al. [39] porte sur cette opération du standard. Les auteurs listent les ambiguïtés et les règles qui leur semblent manquer à l'opérateur *merge* afin que celui-ci respectent le principe général le définissant dans le standard, à savoir :

> "A resulting element will not be any less capable than it was prior to the merge. This means, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge."

L'une des imprécisions présentées par les auteurs concerne la référence de *type*, telle qu'avec les propriétés. Ceci est illustré en figure 2.19. Lorsque *Communications* et *SimpleTime* sont fusionnés, le standard ne permet pas de spécifier quel type référence *TimeEvent*. En observant le métamodèle UML et son utilisation du *merge* comme moyen d'extension, les auteurs soulignent qu'il apparaît que le type le plus spécifique doit être référencé dans le modèle résultat, alors qu'aucune règle ne le spécifie dans le standard.

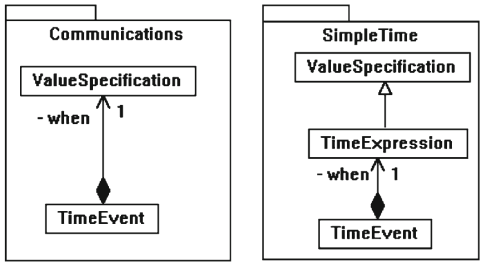


FIGURE 2.19 – Exemple de fusion de types (Source : Dingel et al. [39])

2.2 Modèles paramétrés

Dans cette section, nous nous focalisons sur les modèles paramétrés. Ces derniers permettent d'exposer un ou plusieurs de leurs éléments comme paramètres. Nous en présentons les différentes formes existantes et débutons tout d'abord par les modèles variables (section 2.2.1). Ceux-ci sont utilisés dans les lignes de produits pour dériver différentes variantes de modèles. Nous nous intéressons par la suite (section 2.2.2) aux modèles de rôles qui permettent l'enrichissement de

modèles par attribution de rôles aux éléments. Enfin, en section 2.2.3, nous nous concentrons plus particulièrement sur les templates qui sont à l'origine de ce travail. Dans cette partie, nous passons en revue les travaux autour du standard UML, ainsi que des approches proposant des notions de templates spécifiques.

2.2.1 Modèles Variables

Les modèles variables sont une forme de modèle paramétré utilisés dans les lignes de produits. Un modèle variable est constitué d'éléments dont certains sont déclarés comme *optionnels*, *point de variation* ou *variants*. Un ensemble de variants est lié à un *point de variation*. En sélectionnant des variants correspondant à divers points de variation, il est ainsi possible de créer un nouveau modèle (une variation) à partir du modèle variable.
Différents travaux ont étudié les modèles variables en UML. On peut citer l'approche de Ziadi et al. [105], étendant les diagrammes de classes et de séquences d'UML afin de représenter les variants et points de variation dans les lignes de produits ; les travaux de Tessier et al. [99] qui représentent notamment les relations entre points de variation à l'aide de contraintes telles *And, Alternative, Optional* ; ou encore les *Generic UML models* de Clauss [33], présentés en figure 2.20.
Dans les *Generic UML models*, les éléments optionnels, variants et points de variation sont définis sous forme de stéréotypes UML (notamment sur des classes et *packages*), respectivement nommés *optional*, *variant* et *variation point*.

Des variantes peuvent être reliées aux points de variation via des relations de dépendances (associations, généralisation, etc...). Pour un point de variation donné, il est possible de spécifier le nombre de variants sélectionnables, les instants du cycle de vie du système (attribut *BindingTime*) où ils doivent être choisis. Les *Conditions* représentent quant à elles quelles *features* doivent être sélectionnées, dans un modèle de *features* de plus haut niveau, afin de sélectionner les éléments UML correspondants. Les relations entre points de variation sont définies via la relation *requires* : un variant sélectionné impliquera la sélection d'un autre. L'exemple de la figure 2.20 décrit le passage d'une commande (*Order*) de plusieurs articles (*OrderItem*). Les données du paiement d'une commande sont représentées par le point de variation *PaymentInfo* : à l'exécution (*BindingTime*=runtime), un seul des variants peut être sélectionné (*Multiplicity*=1), *i.e.* les données de facturation, de carte de crédit ou de livraison. La relation *requires* entre *Contract* et *contractBased* indique qu'aucun des deux ne peut être présent sans l'autre.

Dans la figure 2.21, le modèle représente une des possibles variations du modèle variable, suite à la sélection du variant *CreditCard_Data*.

On note que l'énumération *PriceMethod* ne possède pas l'attribut optionnel *contractBased*. En effet, pour cela, il aurait été nécessaire de sélectionner, dans le modèle variable de la figure 2.20, la classe *Contract*. Ceci est dû à la relation de dépendance *requires* entre *Contract* et l'attribut *contractBased* de *PriceMethod*.

Avec ces modèles variables, le paramétrage est limité par les variants, points de variation et éléments optionnels. Par exemple, avec le modèle variable de la figure 2.20, il n'est pas possible d'obtenir un modèle avec des données de paiement concernant l'utilisation d'une crypto-monnaie, puisque seuls des variants concernant les données de facturation, de carte de crédit ou de livraison sont définis.

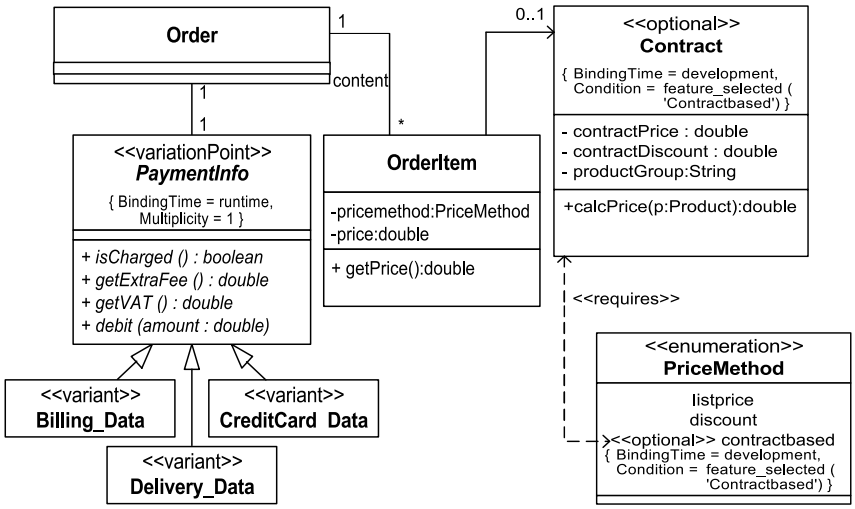


FIGURE 2.20 – Modèle variable - *Generic UML models* (Source : Clauss [33])

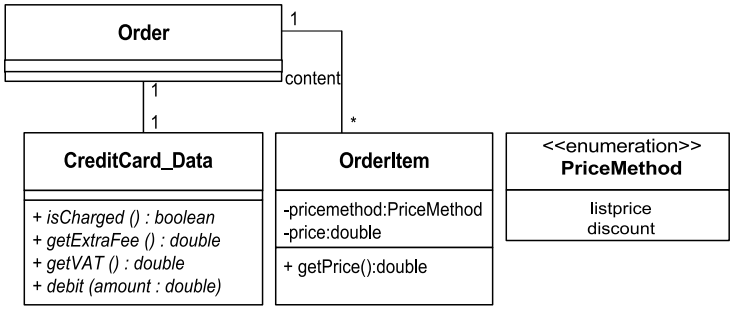


FIGURE 2.21 – Modèle (variation) après sélection de *CreditCard_Data*

2.2.2 Modèles de Rôles

Plusieurs travaux se sont intéressés à la notion de rôles afin d'exprimer des fonctionnalités génériques applicables à des modèles pour les enrichir. On peut citer les travaux de Albin-Amiot et al. [5], Bottoni et al. [19] et Tombelle et al. [100] pour appliquer des patrons à des modèles. Dans ces approches, un modèle de rôles est une structure constituée de rôles et de relations entre ces rôles. Un rôle est la représentation d'une préoccupation de la fonctionnalité générique et peut être vu comme un paramètre. Un modèle de rôles est appliqué à un modèle en affectant des rôles à ses éléments, ce qui a pour conséquence de les enrichir avec de nouvelles caractéristiques. En général, il est possible d'affecter plusieurs rôles à un même élément pour l'enrichir de différentes manières. Les approches proposent en général des règles ou l'expression de contraintes afin de garantir la cohérence de l'affectation des rôles. Un modèle de rôles, tiré de l'approche de Tombelle et al. [100] et représentant le pattern de conception *Proxy*, est présenté en figure 2.22.

Ce pattern est composé des rôles *Subject, Proxy* et *RealSubject* s'appliquant à des classes (<*Class*>). Les rôles sont représentés par des rectangles aux coins arrondis. Les relations entre ces rôles (classes, opérations et attributs) sont représentées selon la légende de la partie droite de la figure. Ces relations contraignent la structure des enrichissements. En figure 2.23, ce modèle de rôles est lié (*binding*) au modèle *Bank* : les rôles *RealSubject* et *request* sont respectivement attribués à la classe *Bank* et à son opération *débit*.

Le modèle résultant de ce binding est présenté en figure 2.24. Les rôles non substitués du proxy

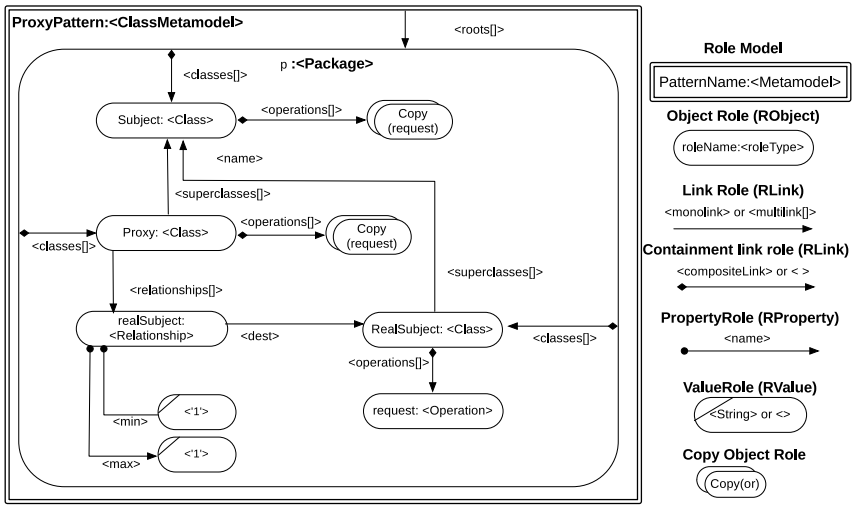


FIGURE 2.22 – Modèle de rôles, représentant le pattern *Proxy* (Source : Tombelle et al. [100])

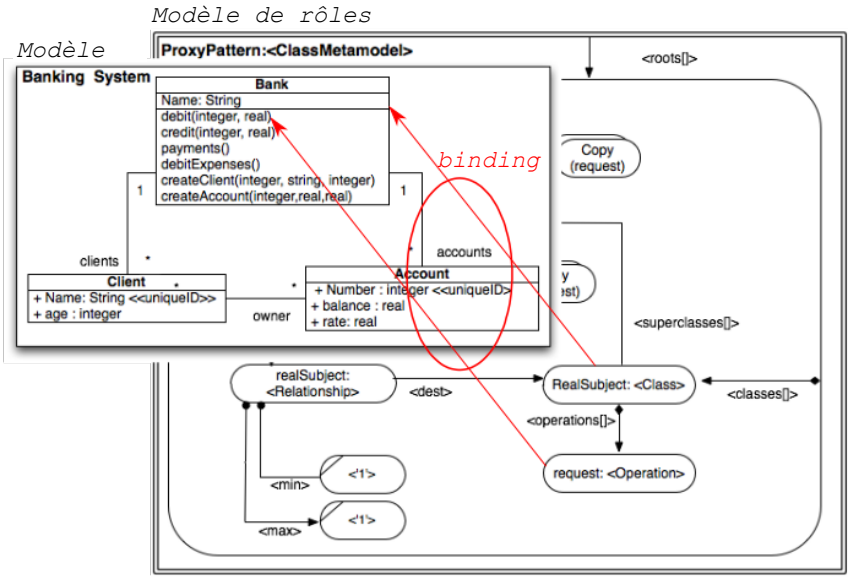


FIGURE 2.23 – Modèle de rôles appliqué à un modèle (Source : Tombelle et al. [100])

enrichissent le modèle. On note que la présence des opérations *débit* dans ces classes est due aux rôles *copy (request)* et aux relations avec *Subject* et *Proxy* du modèle de rôles. *copy (request)* copie l'opération à laquelle est attribué *request* (ici, *débit*) dans les classes *Subject* et *Proxy*.

2.2.3 Templates

Dans cette section, nous présentons différentes approches définissant et utilisant les templates. On trouve généralement deux principaux usages dans les travaux existants.
Le premier permet la représentation, au niveau modèle, de composants logiciels génériques, tels les templates de *C++* ou les génériques de *Java*. Avec les approches de ce type, les templates sont appliqués de manière générative, ce qui permet d'obtenir des instances de templates qui sont des modèles spécifiques à des contextes.
Le second usage concerne l'application de ces templates à un système en construction, afin

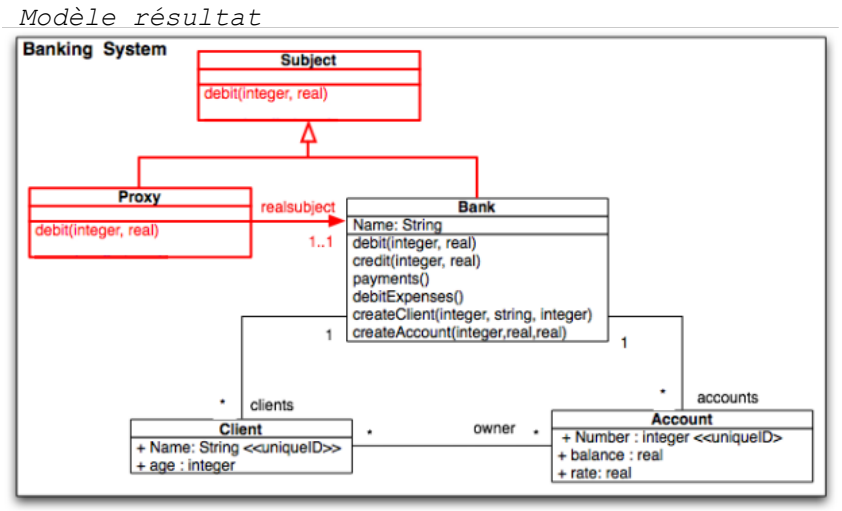


FIGURE 2.24 – Résultat de l'attribution des rôles (Source : Tombelle et al. [100])

d'enrichir celui-ci. Cette pratique correspond à une utilisation *aspectuelle* des templates.
Ci-après, nous présentons de façon détaillée les templates du standard UML, et les extensions proposées par divers travaux, afin de mieux appréhender la suite de la thèse. Enfin, nous étudions différents travaux hors du standard UML.

2.2.3.1 Templates UML

Les templates de modèles UML (ou plus brièvement *templates UML*) permettent de capturer et de réutiliser des constructions génériques en exposant certains de leurs constituants sous la forme de paramètres. De telles constructions peuvent être (*i.e.* pas uniquement) des classes ou des *packages*.
Par exemple, en figure 2.25, le patron de conception *ObserverPattern* est représenté par un template de *package*.

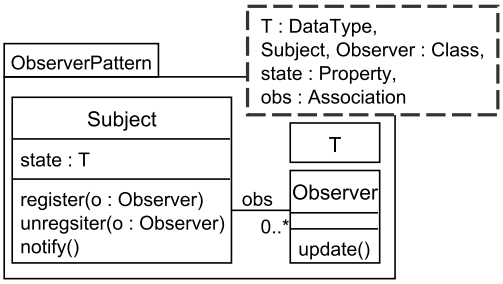


FIGURE 2.25 – Un template UML représentant le pattern Observer

Afin de spécifier cette paramétrisation, un template possède une signature, représentée sous la forme d'une liste de paramètres typés (le rectangle en pointillé en figure 2.25) où chacun d'entre eux désigne un constituant du modèle de template.
L'objectif des templates est d'être réutilisés pour enrichir des modèles existants ou produire de nouveaux modèles. L'application d'un template correspond à la substitution de ces paramètres par des éléments d'un modèle. Une telle substitution est effectuée au travers de la relation *bind*

qui est spécifiée dans le standard de la façon suivante :

> "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 a formal template parameter by the corresponding elements specified as actual parameters in this binding" (OMG [78], page 650)

Cette relation lie donc un *bound model* à un template (à partir duquel le *bound model* a été obtenu) via un ensemble de substitutions qui associe les *paramètres formels du template* aux *éléments actuels* du *bound model*. Seuls les éléments en paramètres sont substitués, assurant ainsi un contrôle plus important de leur application (*i.e.* en fonction de ce paramétrage, il est possible de déterminer si un template peut être appliqué ou non sur un modèle). De plus, les contraintes du standard imposent que le type de chaque élément actuel doit être un sous-type ou du même type que le paramètre formel correspondant.
La figure 2.26 présente le *binding* d'une classe avec un template de classe. Cet exemple correspond à l'usage génératif d'un template (à la manière des templates en C++). Le *template Stack* est paramétré par *Element*, de type *Class* et par *Max*, de type *int*. Ces paramètres sont respectivement substitués par *Plate* et *15* dans le modèle *PlateStack*. Dans cet exemple, on observe que, comme *CarHiringSystem* et *ObserverPattern* en figure 2.27, *PlateStack* inclut la structure du *template Stack*. Puisque cette classe inclut ici *uniquement* la structure du template, il s'agit bien d'une instance de *Stack*. De plus, le contexte applicatif d'où proviennent les éléments *Plate* et *15* est indéterminé.

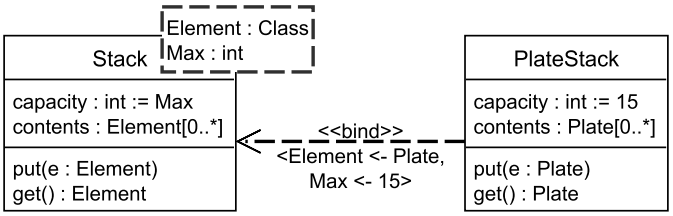


FIGURE 2.26 – Instanciation du template de classe *Stack*

La relation *bind* permet également d'enrichir un modèle avec les éléments d'un template. La figure 2.27 présente un *template UML* utilisé pour étendre un modèle de système via la relation *bind*.
Le template, sous la forme d'un *package*, est utilisé pour modéliser le *patrons de conception Observer* (Gamma et al. [50]). Il est paramétré avec les classes *Subject* et *Observer*, l'attribut *value*, l'association *obs* et le type *T*. Le modèle sur lequel doit être appliqué le *pattern* représente une agence de location de voitures avec ses clients. Le template est utilisé pour injecter les fonctionnalités du *pattern Observer* entre *Agency* et *Clients* et permettre aux clients d'observer l'état d'une agence. Ce choix de conception est spécifié par un ensemble de substitutions et la relation *bind* entre le modèle *CarHiringSystem* et le *template ObserverPattern*. Les éléments en gras dans chacun d'eux représentent respectivement les éléments actuels et les éléments formels. En conséquence de cette application, *CarHiringSystem* inclut la structure du *template ObserverPattern*, *i.e.* les éléments en italique dans *CarHiringSystem* (*register, unregister, notify et update*).

En plus de l'application, le standard UML présente deux autres façons d'utiliser les templates :

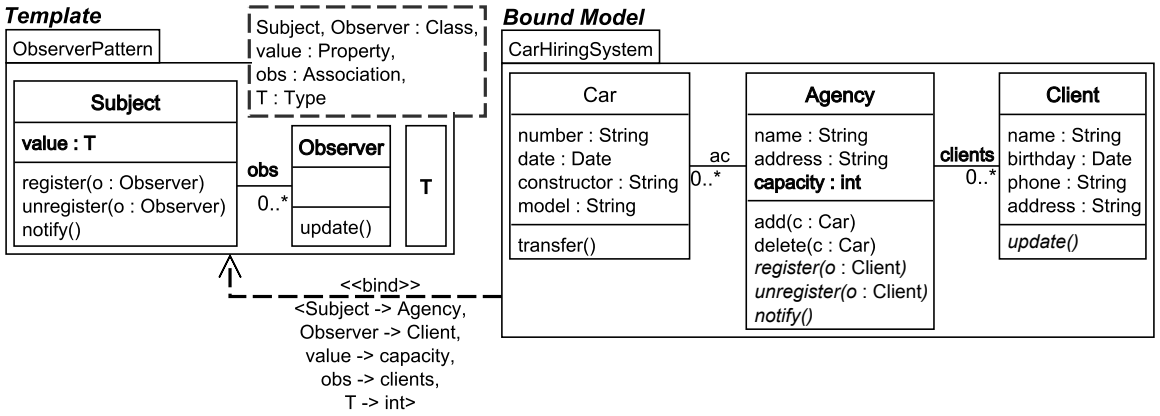


FIGURE 2.27 – Application d'un template de package UML

- l'application partielle (OMG [79], page 634), c'est-à-dire lorsque seule une partie des paramètres est substitué. La documentation d'UML précise que les paramètres non substitués sont alors propagés au bound model, faisant de celui-ci un template,
- la composition de templates, *i.e.* l'application d'un template sur un autre.

Ces deux points ne sont pas davantage développés par le standard et sont étudiés dans le chapitre 3 de cette thèse.

En ce qui concerne la représentation structurelle des templates, le métamodèle UML [78] utilise quatre classes principales : *TemplateSignature, TemplateableElement, TemplateParameter et ParameterableElement* (cf. figure 2.28).

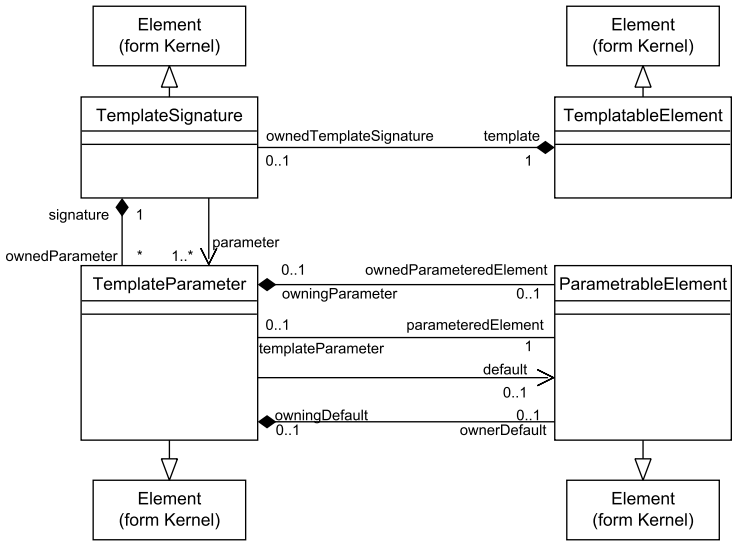


FIGURE 2.28 – Métamodèle des *templates UML*

Les méta-classes *TemplateBinding* et *TemplateParameterSubstitution* sont toutes deux utilisées afin de lier les templates à des modèles (cf. figure 2.29).
Les éléments UML qui sont des sous-classes de *TemplateableElement* peuvent être paramétrés. Les *classifiers*, en particulier les classes et *packages*, sont des éléments *"templatables"*. Les paramètres (*TemplateParameter*) d'un template (*TemplateableElement*) sont inclus dans une signature (*TemplateSignature*).

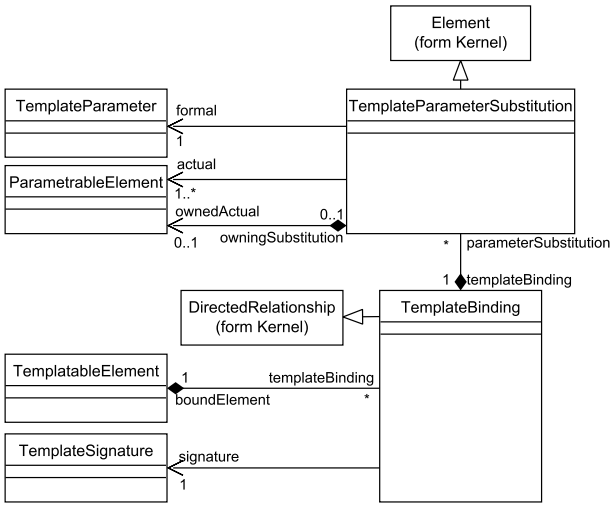


FIGURE 2.29 – Métamodèle du *binding de templates UML*

Un *TemplateParameter* est un paramètre formel de template et expose un ingrédient du template grâce au rôle *parameteredElement*. Seuls les éléments paramétrables (*ParameterableElement*) peuvent être exposés en tant que paramètres formels d'un template ou définis comme des arguments actuels dans un *binding* de template. Les éléments de types *Classifier, PackageableElement, Operation et Property* sont paramétrables.
Un *binding* de template (cf. figure 2.29) est une relation unidirectionnelle, labellisée par le stéréotype *bind*, partant de l'élément lié (*boundElement*) au template (signature). Un *binding* possède un ensemble de substitutions de paramètres de template (*TemplateParameterSubstitution*). Une substitution associe un paramètre formel, provenant de la signature d'un template, à un élément actuel paramétrable de l'élément lié.
La figure 2.30 présente un extrait de l'instanciation de ce métamodèle pour l'exemple décrit en figure 2.27. Il décrit la substitution entre le paramètre formel de *template "Subject"* et l'argument actuel *"Agency"* du modèle lié *CarHiringSystem*. Enfin, la spécification UML introduit aussi des

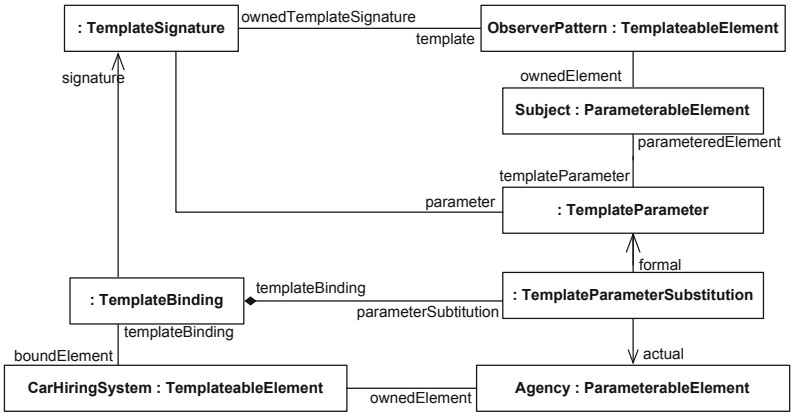


FIGURE 2.30 – Extrait du diagramme d'objets pour le paquetage *CarHiringSystem*

contraintes vérifiant la définition des templates et leur *binding*. Ces contraintes vérifient que :

- Les éléments exposés en tant que paramètres dans la signature du template appartiennent

au template,
- Dans une substitution, le paramètre formel et l'argument actuel ont des métatypes compatibles,
- Chacun des paramètres d'un ensemble de substitutions correspond à un paramètre formel de la signature du template cible.

Les spécifications actuelles d'UML fournissent peu d'informations quant aux éléments pouvant être des templates. Le standard inclut uniquement des exemples et explications pour les templates de classes, collaborations et *packages*. Les autres templates autorisés ne sont pas décrits, rendant difficile leur identification. Une telle identification est utile, afin de pouvoir utiliser la notion de templates hors des diagrammes de classes et de collaborations mais aussi des extensions à ces templates, telles que celle présentée dans le chapitre suivant, en section 3.1.
Dans l'approche de Vanwormhoudt et al. [103], une introspection du standard a été effectuée via le *plugin UML* de l'*IDE Eclipse*, implémentant la spécification elle-même. Cette introspection a été réalisée en collectant toutes les sous-classes directes et indirectes des éléments *TemplateableElement* et *ParameterableElement*, puis en visitant tous les constituants des éléments "templatables".

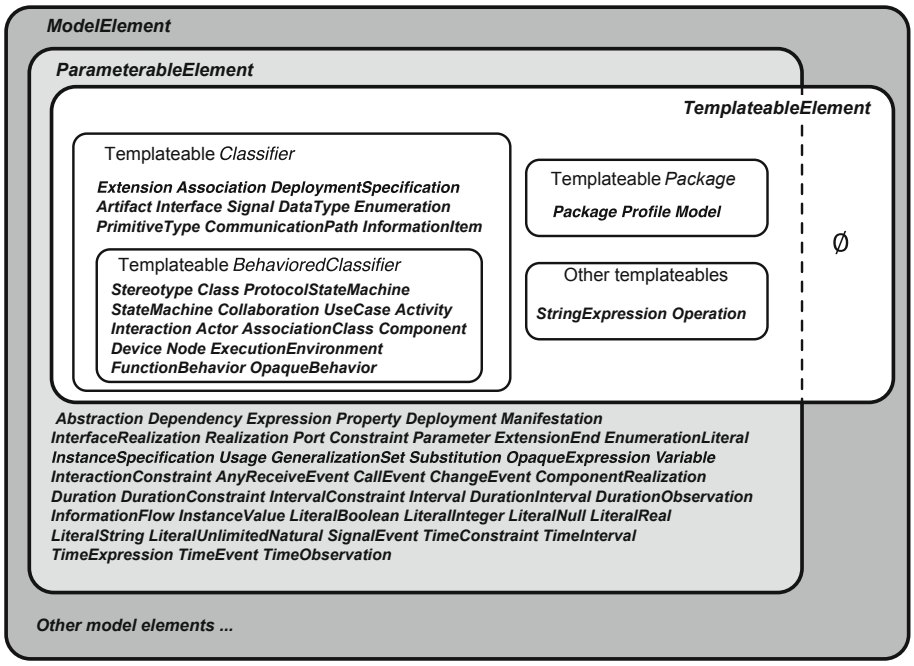


FIGURE 2.31 – Éléments paramétrables et templatables (Source :Vanwormhoudt et al. [103])

La figure 2.31 présente l'ensemble complet des éléments templatables et paramétrables du standard. Comme prévu, seul un sous-ensemble des éléments UML sont paramétrables. Concernant les éléments templatables, ils sont un peu moins nombreux que ceux paramétrables mais en les comparant, on remarque la chose suivante. Bien que *TemplateableElement* ne soit pas une sous-classe de *ParameterableElement* dans le métamodèle, tous les éléments templatables sont aussi des éléments paramétrables. Concrètement, cela signifie que ces éléments peuvent avoir l'un des deux rôles selon la situation de modélisation (par exemple, *Class* ou *Component* peuvent être aussi bien template que paramètre). En figure 2.31, ceci est montré par l'inclusion de tous les éléments templatables dans l'intersection avec l'ensemble des éléments paramétrables.
L'exemple de la figure 2.32 présente l'aspectualisation d'un modèle de composant. Dans cet

exemple, l'aspectual template installe un service d'enregistrement entre deux composants. Le modèle paramètre de ce template spécifie que ces deux composants doivent être connectés. Ce template est lié à un modèle représentant un système de gestion de chambres d'hôtels.

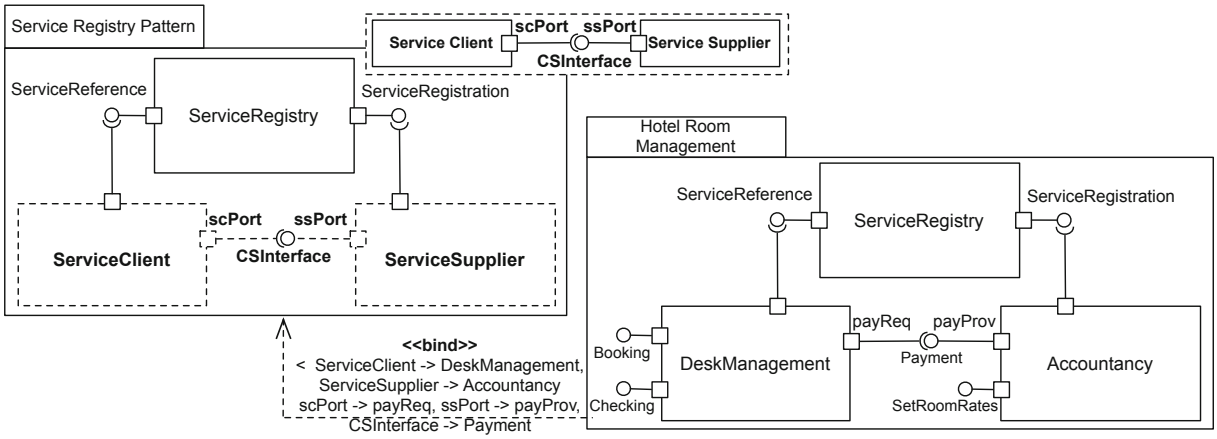


FIGURE 2.32 – Aspectual templates de composants (Source :Vanwormhoudt et al. [103])

L'étude effectuée par Vanwormhoudt et al. [103] a permit de définir quels éléments étaient templatables pour chacun des diagrammes. La table 2.33 présente les principaux types de modèles avec leur concept templatable racine et les constituants paramétrables correspondants. Selon cette table, on observe que, par exemple, un modèle de cas d'utilisation peut avoir ses constituants définis en tant que paramètres. En plus des éléments paramétrables, la table 2.33 donne l'ensemble des éléments qui sont essentiels pour chacun des types de modèles mais qui ne sont pas paramétrables dans le template correspondant. On peut noter que certains éléments templatables permettent une paramétrisation limitée. C'est le cas des modèles d'interaction, d'activités et de machine à états.

Extensions des *templates UML*

Les travaux présentés ci-après étendent les *templates UML* avec des contraintes concernant le binding et le paramétrage des templates. L'objectif de telles contraintes est de renforcer la cohérence des applications. L'approche de Vanwormhoudt et al. [103], utilisée comme fondement de cette thèse, fait partie de cette catégorie de travaux. Elle renforce la sémantique du standard UML et des travaux de Muller [73] à l'aide de contraintes OCL. Cette approche est présentée dans le chapitre suivant, en section 3.1.

Cohérence du modèle résultat

Contraintes sur le binding Les travaux de Caron et al. [23] portent sur la définition de contraintes OCL(OMG [83]) pour vérifier qu'un modèle désigné par la relation *bind* comme *bound model* d'un template UML est valide, c'est-à-dire est effectivement enrichi avec le template selon la substitution définie avec la relation *bind*.
Les auteurs rappellent que le standard UML définit que le type d'un élément substitué doit être du même type ou sous-type de celui du paramètre. Afin de répondre à la définition donnée par UML concernant le binding (cf. définition 2.2.3.1), ils définissent les contraintes OCL supplémentaires suivantes :

- Tous les éléments actuels utilisés dans la substitution :
  . doivent appartenir au bound model,

Model kind	Templateable root concept	Parameterable constituents	Nonparameterable constituents
Class model	Package	Package, Class, Association, Property, Operation, Parameter	Generalization
Instance model	Package	InstanceSpecif cation, InstanceValue, Type	Slot
Use-case model	Package	UseCase, Actor, Association	Include, Extend
Collaboration model	Collaboration	Property (Role), Type	Connector, CollaborationUse
Component model	Package	Component, Port, Property, Dependency, Type	Connector
Interaction model	Interaction	Operation, Port	Lifeline, Message, MessageOccurrenceSpecif cation, BehaviourExecutionSpecif - cation
Activity model	Activity	Operation, Port	ActivityPartition, ActivityEdge, ActivityNode, Action
State machine model	StateMachine	Operation, Port	Region, State, Transition
Deployment model	Node	Node, Device, Execution Environment, DeploymentSpecif cation, Artifact, Communication Path, Deployment, Manifestation, Property, Operation	

FIGURE 2.33 – Classification des types de modèles UML templatables (Source :Vanwormhoudt et al. [103])

. correspondent à un élément paramètre du template

- Tous les éléments du template correspondent à un élément du bound model selon les règles suivantes :
  . Tout élément e_t de template contenu dans un paramètre p doit être contenu dans l'élément e_{bm} du bound model substitué avec e_t,
  . Une opération d'un bound element correspond à une opération d'un template si leur signature est identique par copie ou selon une substitution du binding. La figure 2.34 donne un exemple de ceci. Seule la classe X du template est substituée à la classe Y du bound model. L'opération *foo*, non substituée, devrait donc posséder la même signature dans le template et le bound model, excepté le paramètre de type X, substitué par Y. Hors, le type de son paramètre est Z dans le bound model. Celui-ci est donc invalide.

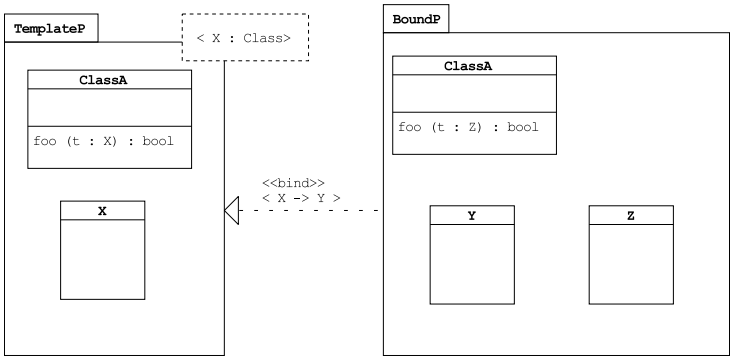


FIGURE 2.34 – Erreur concernant la signature d'une opération (Source :Caron et al. [23])

. Toute association dans un bound model :
doit être liée à des propriétés (les propriétés permettent de lier l'association à une classe, interface ou type de donnée) conformément (par copie ou substitution) aux propriétés de l'association correspondante dans le template,
doit posséder la même arité que l'association à laquelle elle correspond dans le template.

La figure 2.35 illustre les dernières contraintes concernant les associations. Dans cet exemple, l'arité de *asso*, substituée avec *link*, est identique dans le bound model. La contrainte concernant les proriétés d'*asso* et donc les classes A et B auxquelles elle est reliée, n'est pas respectée dans le bound model avec *link*. En effet, cette dernière est reliée à Y et C, alors que A et B sont respectivement substituées par X et Y.

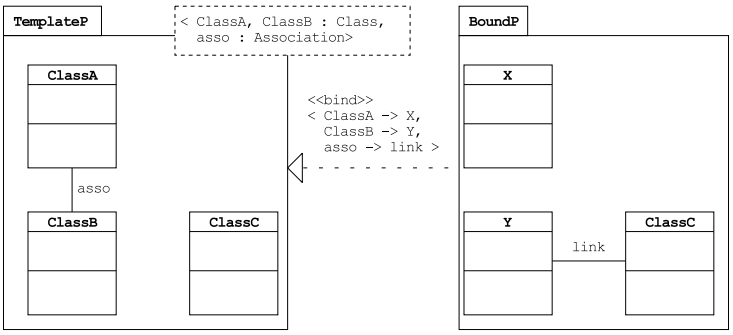


FIGURE 2.35 – Erreur concernant une des propriétés d'une association (Source :Caron et al. [23])

Contraintes sur le binding et "conformité fonctionnelle"

L'approche de Farinha & Ramos [43] propose un ensemble de contraintes lié au concept défini par les auteurs comme la *conformité fonctionnelle*. Cette conformité est imposée entre chaque paramètre d'un template et les éléments du domaine de substitution, afin d'éviter l'obtention de modèles résultats mal-formés. L'approche définit ainsi pour cette conformité fonctionnelle des conformités sur le type, la multiplicité, la relation de contenance (par exemple, une propriété incluse dans une classe) et la "staticité" et un besoin concernant la visibilité des éléments.

Pour rappel, UML définit la conformité de type comme suit : le type d'un élément substitué doit être du même type ou sous-type de celui du paramètre. Selon les auteurs, cette conformité est incomplète car :

- UML applique ceci seulement sur les propriétés et les valeurs spécifiées,
- si le type t du paramètre substitué est lui aussi un paramètre substitué, alors la conformité ne doit pas considérer t mais le type t' qui substitut t.

La figure 2.36 illustre ceci. *Att* est substitué avec *name*. La conformité de type ne doit donc pas considérer le type de *Att* directement avec celui de *name* (soit T avec *String*) mais le type substituant celui de *Att* avec *name*, soit *String* et *String*. Le type de *Att* est ici conforme à celui de *name*.

Un contre-exemple est donné en figure 2.37, où le type substitué avec T est *Integer*. Le type de *Att* est donc ici non conforme à celui de *name* (*Integer* et *String*).

En ce qui concerne la conformité de multiplicité, les auteurs proposent une règle vérifiant si des éléments substitués ont les mêmes multiplicités et, lorsque celles-ci sont multivaluées (valeur haute à 1), sont ordonnées à l'identique. Ils expliquent que cela pose problème pour du code uti-

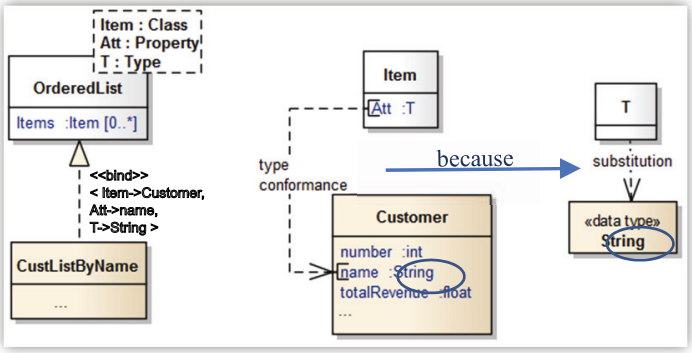


FIGURE 2.36 – Substitution du type d'un paramètre (Source :Farinha & Ramos [43])

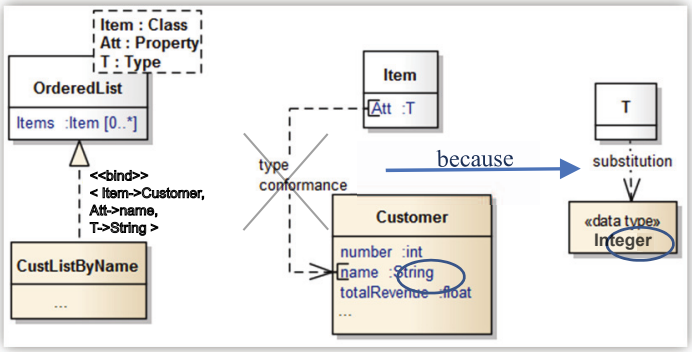


FIGURE 2.37 – Substitution du type d'un paramètre - (Contre-exemple)

lisant des éléments non-multivalués et substitués par d'autres multivalués, tout comme certaines opérations sur des éléments ordonnés ne sont pas disponibles sur des éléments non-ordonnés.
La troisième conformité est en rapport avec la relation de contenance. Cette conformité assure qu'un espace de noms $NS1$ à un contenu conforme à un espace de noms $NS2$ si chaque élément de $NS2$ référencé par le template est substitué par un élément de $NS1$.
La figure 2.38 présente un exemple de non-conformité sur une telle relation. En effet, la classe *Item* inclut la propriété *Name* et cette dernière n'est pas substituée. Seuls les éléments inclus dans le template de classe *AlphabeticList* sont ajoutés à *Biography* avec la relation *bind*. Un exemple de solution est de paramétrer *Item* avec *Name* et substituer cet élément avec *title* de la classe *Document*.
Vient ensuite la conformité de "staticité" et le besoin de visibilité d'un élément. Pour la première, les auteurs rappellent qu'une fonctionnalité statique est exécutée par son classifier et que celles non-statiques par des instances de celui-ci. Selon eux, il est donc nécessaire, que deux fonctionnalités substituées soient toutes deux statiques ou non, afin de ne pas modifier le comportement défini dans le template. Ensuite, en ce qui concerne la visibilité, les auteurs précisent qu'un élément substitué par un paramètre doit rester visible afin de permettre aux éléments du *bound model* qui le référencent de pouvoir toujours le référencer après substitution.

Cohérence du modèle résultat et granularité du paramétrage
Contrats sur le paramétrage Cuccuru et al. [35] proposent l'utilisation des *templates UML* afin d'avoir une sémantique de variabilité au niveau méta et modèle.

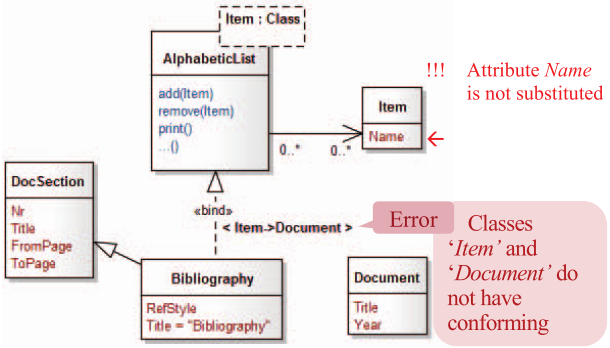


FIGURE 2.38 – Exemples de non-conformité à une relation de contenance (Source : Farinha & Ramos [43])

Pour les contraintes d'applications des templates au niveau modèle, les auteurs utilisent la notion de *contrat*. Ce mot-clef, utilisé dans le paramétrage du template (cf. figure 2.39), désigne la classe spécifiant la structure et le comportement requis par la classe qui instanciera le template.
Par exemple, en figure 2.39, le *template Iterator* spécifie au travers de son paramétrage que l'élément l'instanciant devra être une classe, respectant le *contrat* défini par la classe *IterableForward*.

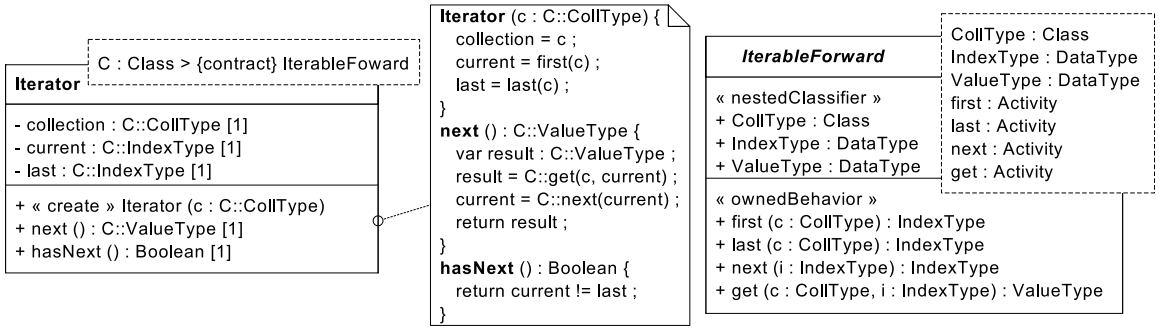


FIGURE 2.39 – Spécification de *Iterator* et de la classe définissant le contrat *IterableForward* (Source : Cuccuru et al. [35])

Afin de spécifier la substitution, la relation *bind* du standard est utilisée, ainsi que la relation de réalisation. L'élément respectant le contrat défini par le template est lié au classifier définissant le contrat par ces deux relations. L'objectif des auteurs est ici de modifier le comportement de la relation *bind* du standard avec l'utilisation de la relation de réalisation ("realize"). Cette dernière relation spécifie donc qu'un élément réalise le classifier définissant le contrat selon la substitution définie avec la relation *bind*. Contrairement au standard, aucun élément n'est injecté dans l'élément réalisant le classifier.
En figure 2.40, la classe *Vector* réalise la classe définissant le contrat de *Iterator*. Ensuite, *Iterator* est appliqué sur la classe *Vector* via la relation *bind*, dont le résultat (*Iterator*<*Vector* <*Integer* > >) est visible dans la partie droite de la figure 2.40.
Via les contrats définis pour le paramétrage d'un template, l'approche présentée permet de spécifier de façon cohérente les paramètres d'un template. L'utilisation conjointe des relations *realize* et *bind* permet de mettre en œuvre l'instanciation de templates.

Classes et méthodes paramètres et composition de templates L'approche *Theme* (Clarke & Walker [31], Baniassad & Clarke [13], Clarke & Walker [32], Carton et al. [26]) permet l'analyse et

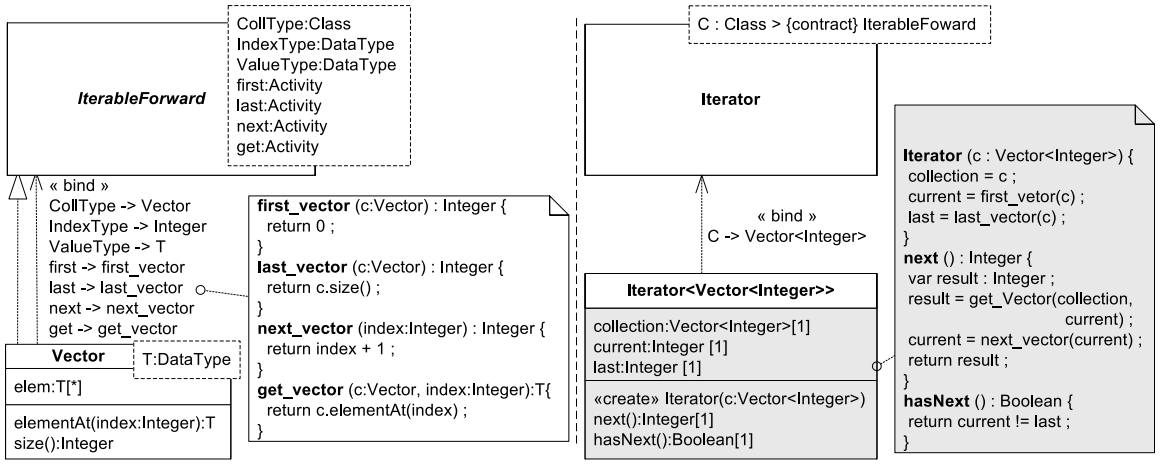


FIGURE 2.40 – Réalisation du contrat de *IterableForward* et application de *Iterator* (Source : Cuccuru et al. [35])

la modélisation orientée aspect avec, respectivement, *Theme/Doc* et *Theme/UML*. *Theme/UML* apporte la prise en compte des diagrammes de séquences, une granularité plus élevée du paramétrage et la possibilité de composer des templates (*Themes*).
Theme/Doc permet d'identifier les aspects lors de l'analyse des besoins. Pour cela, les relations entre les fonctionnalités du système sont représentées via un graphe d'analyse.
Theme/UML permet quant à lui de modéliser les aspects et fonctionnalités d'un système, et spécifie aussi comment ils doivent être composés. Un *Theme* correspond à un template de paquetage et permet de définir des traitements génériques, comme des fonctionnalités de trace ou de synchronisation. Les paramètres d'un tel paquetage sont représentés sous la forme d'une liste, dans un rectangle en pointillé dans le coin supérieur droit du paquetage. Ces paramètres correspondent à une ou plusieurs classes et les méthodes de celles-ci.
Un exemple est donné en figure 2.41. Les paramètres du theme *Logger* sont la classe *Logged* et la méthode *_log*.
Le diagramme de séquences d'un *theme* représente les modifications apportées aux fonctionnalités d'un autre *theme* lorsque les *theme* sont composés. Une relation *bind* permet de représenter de telles compositions.

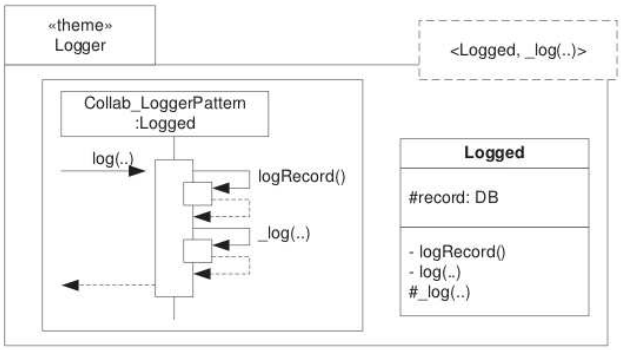


FIGURE 2.41 – theme d'une fonctionnalité de *log* (Source : Baniassad & Clarke [13])

La figure 2.42 présente la composition du theme *Logger* avec le theme *CMS*, via la relation *bind*. Chacune des classes du premier ensemble, c'est-à-dire *Person, Student* et *Professor*, provenant de *CMS*, sont substituées avec la classe *Logged*. La fonctionnalité de trace de *Logger* est appliquée aux méthodes de ces trois classes, citées dans le second ensemble, soit *register, unregister* et *giveMark*.

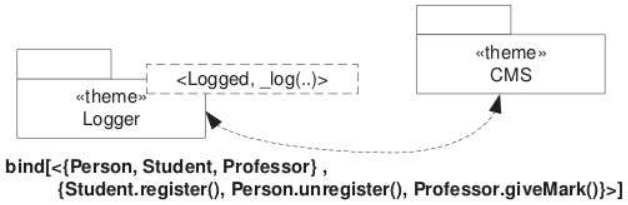


FIGURE 2.42 – Utilisation du Theme de *log*
(Source : Baniassad & Clarke [13])

Cette approche *Theme/UML* propose une granularité pour le paramétrage définie par des classes et un ensemble de méthodes appartenant à ces classes.

"Modèle paramètre" et propriétés d'ordre sur les compositions de templates Les auteurs présentent une approche (Muller [72], Muller et al. [75], Muller [73]) permettant de construire un système par application de *composants de modèles* représentant divers *aspects*. Ces composants sont exprimés sous la forme de templates. Cette approche propose de propager les paramètres non substitués lors de la composition de templates, tout en définissant des propriétés d'ordre sur ces séquences de composition.
Pour assembler les composants décrivant des aspects fonctionnels, l'auteur spécifie les interfaces fournies via un modèle, afin d'expliciter le modèle requis pour utiliser un composant, comme présenté en figure 2.43.

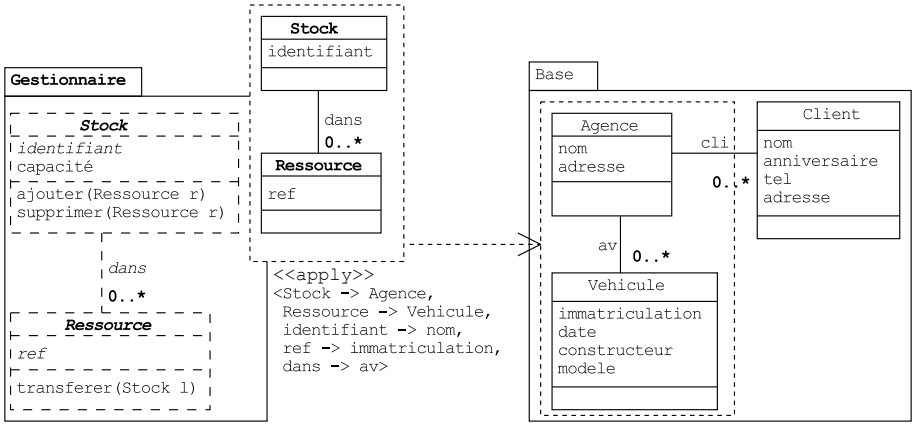


FIGURE 2.43 – Application d'un composant de modèle
à un modèle de base (Source : Muller [73])

Différentes contraintes concernant le respect de cette structure lors des applications de composants ont été définies. Les travaux utilisés comme fondements de cette thèse sont une extension de l'approche de Muller [73] : les contraintes concernant ce modèle requis (ou modèle paramètre) et les applications de template sont présentées en section 4.2.

L'application d'un composant sur un modèle de *base* est définie par l'opérateur *apply* qui utilise un ensemble de substitutions afin de fournir des éléments du modèle de base au modèle requis du composant paramétré.
La figure 2.43 donne l'exemple de l'application du composant *Gestionnaire*, possédant un modèle requis formé de deux classes et d'une association, sur le composant *Base*.
Il est de plus possible d'appliquer un composant paramétré à un autre, afin d'enrichir les composants avant leur application au modèle de base.
Ceci est illustré par la figure 2.44, avec l'application du composant *Recherche* sur *Gestionnaire*.

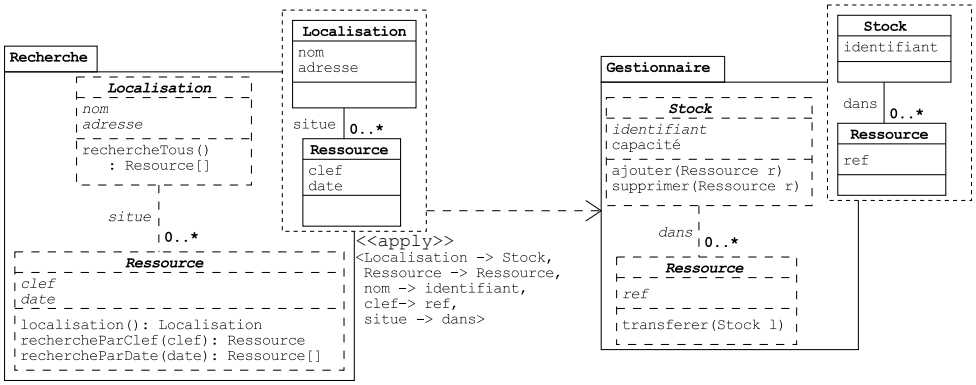


FIGURE 2.44 – Application d'un composant de modèle à un autre (Source : Muller [73])

Une chaîne d'application est définie par l'auteur comme une séquence d'applications de composants résultant sur un modèle de base, augmenté avec les fonctionnalités fournies par ces composants. Il est possible de couper une chaîne de composants sans remise en cause des autres applications. La figure 2.45 présente une telle chaîne. Dans cet exemple, si l'application de *Comptage* est supprimée, l'application de *Allocation* à la base reste alors valide. De même, si on supprime l'application de *Allocation* à la *Base*, celle de *Comptage* à *Allocation* reste elle aussi valide.

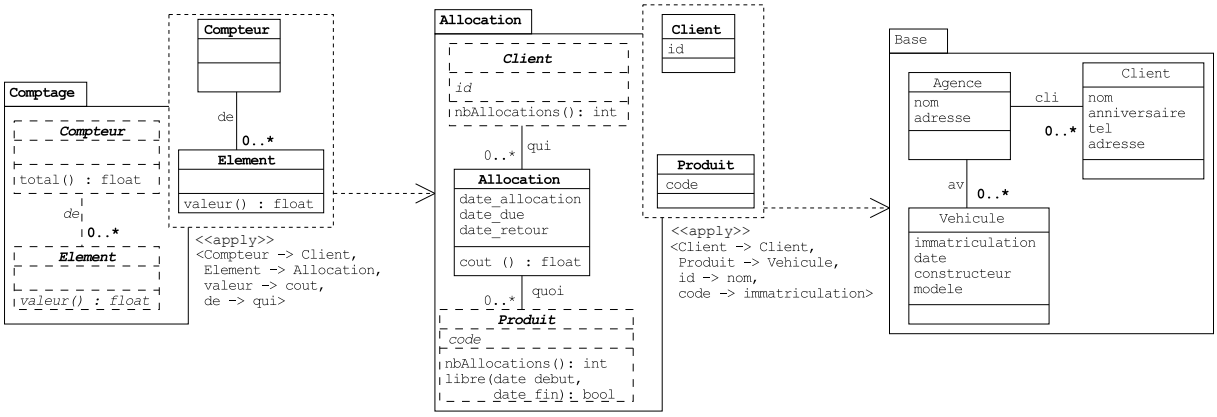


FIGURE 2.45 – Séquences d'applications de *Comptage* à *Allocation* à *Base* (Source : Muller [73])

L'approche définit aussi des propriétés d'ordre concernant de telles séquences d'applications et leur modèle résultat.

2.2.3.2 Approches à base de templates

Catalysis Afin de définir des éléments génériques, l'approche *Catalysis* de D'souza & Wills [40] propose des *Frameworks* ou templates de paquetages. Ces paquetages sont paramétrés avec des *placeholders* qui sont des éléments de modèle. Ces *placeholders* sont identifiés par leurs noms et peuvent être substitués lors de l'utilisation du *framework* avec un composant.
L'exemple de la figure 2.46 illustre ceci avec un *framework* de gestion d'allocation de ressources. Les *placeholders* sont désignés via des chevrons (<>), soit ici les classes *JobCategory, Job, Facility* et *Resource*. Le framework définit des règles entre ces concepts sous forme d'invariants.

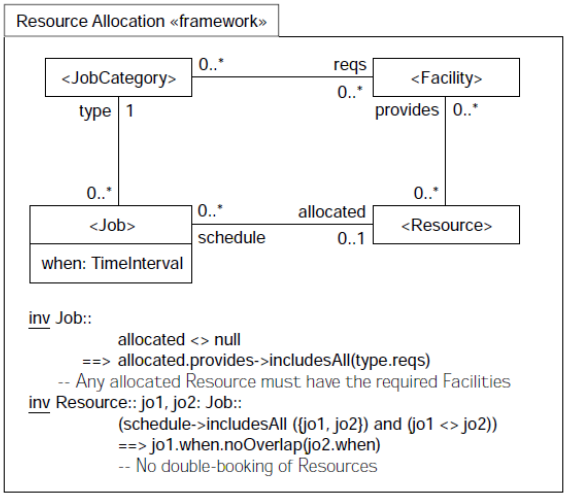


FIGURE 2.46 – Framework d'allocation de ressources (Source : D'souza & Wills [40])

Un exemple d'utilisation du framework *ResourceAllocation* est présenté en figure 2.47. Les classes paramètres sont associées aux classes *JobDescription, Job, Skill, Plumber* du modèle sur lequel on souhaite appliquer *ResourceAllocation*. Cette application consiste à ajouter les liens et les invariants définis par le framework sur les éléments utilisés en paramètre. En figure 2.47, l'association entre *JobCategory* et *Facility* du Framework est ajoutée entre *JobDescription* et *Skill*.

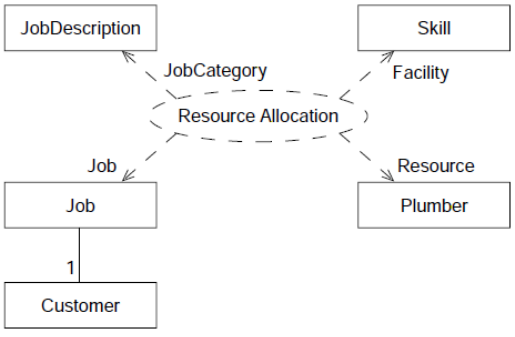


FIGURE 2.47 – Utilisation du framework d'allocation de ressources (Source : D'souza & Wills [40])

Avec cette approche, il est possible de composer des *frameworks* entre eux afin d'en concevoir un autre, mais aucune règle n'est spécifiée pour cela. En ce qui concerne l'adaptation du paramétrage, les *placeholders* peuvent être non-substitués. Ils correspondent alors à la définition de

nouveaux éléments, *i.e.* ils n'ont pas le statut de *placeholders* après utilisation du *framework*.
L'approche permet aussi de spécifier des structures génériques et réutilisables que sont les templates de packages. À l'instar des templates en C++ et contrairement aux frameworks et placeholders précédents, la substitution de leurs paramètres avec des éléments de modèles n'enrichissent pas ces modèles mais définissent des structures utilisables dans un contexte spécifique.
Un template de package possède un ou plusieurs paramètres, pouvant être des attributs ou des types rendus génériques. Ces paramètres sont exprimés au sein d'un rectangle en pointillé, sur l'élément incluant ces éléments génériques. Dans la figure 2.48, la classe *Item* et l'opérateur < sont les paramètres de la classe *SortedList*.

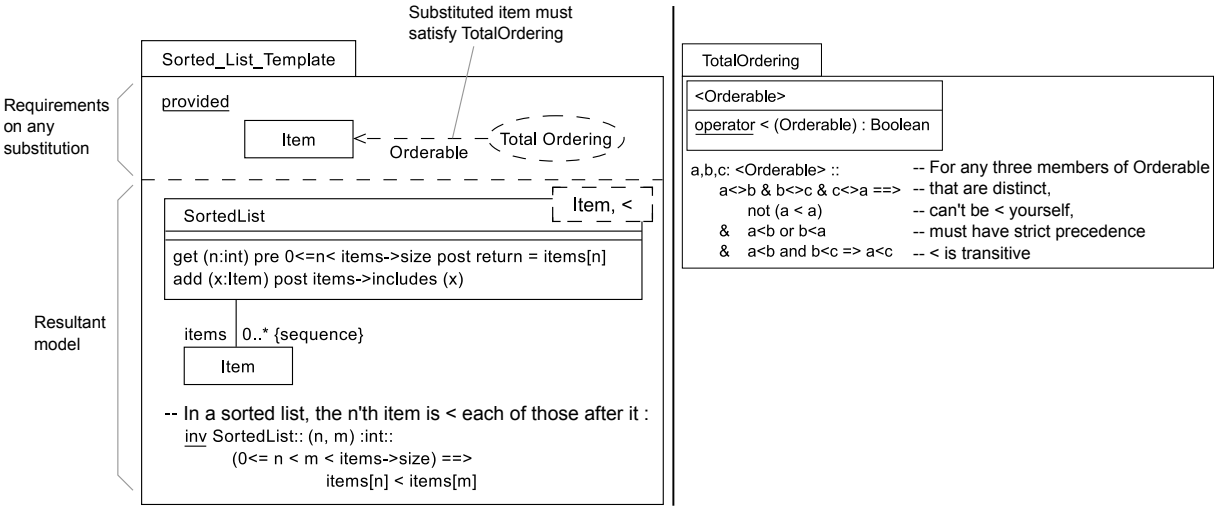


FIGURE 2.48 – Défintion du template Sorted_List_Template (Source : D'souza & Wills [40])

Ces paramètres peuvent être contraints, c'est-à-dire que les éléments devant substituer ces paramètres doivent répondre à certains critères. L'ensemble de ces contraintes est spécifié dans le template par des *provisions*, représenté en partie haute d'un template, dans la zone nommée *provided*. En figure 2.48, l'élément substituant *Item* doit répondre à la structure du framework *TotalOrdering* qui est la définition de l'opérateur <. La figure 2.49 présente quant à elle l'application de ce template avec des éléments du modèle *FruityDefinitions*, à savoir *Banana* et *shorterThan* substituant respectivement *Item* et <. Le modèle résultant de cette application est *Sorted List of Banana_ shorterThan*.

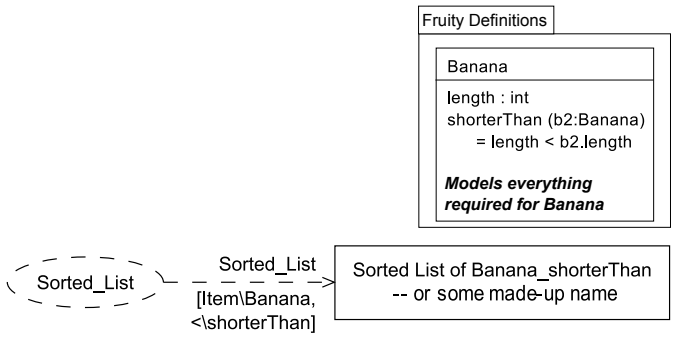


FIGURE 2.49 – Application du template Sorted_List_Template (Source : D'souza & Wills [40])

Les génériques du framework *EMF* Tout ce qui peut être effectué avec les génériques de Java est réalisable avec les génériques du framework EMF (Steinberg et al. [96]), c'est-à-dire la paramétérisation de collections, classes et interfaces. Ce framework utilise le métamodèle *Ecore* qui est une implémentation du métamodèle *MOF* de l'OMG (OMG [84]).
La figure 2.50 présente les concepts ajoutés à Ecore afin de modéliser des génériques. Ce métamodèle permet de spécifier des paramètres formels sur EClassifiers et EOperations et d'utiliser des types génériques dans ETypedElements, les supertypes EClass et les EOperations. Les ETypeParameter et EGenericType permettent de représenter des types génériques.

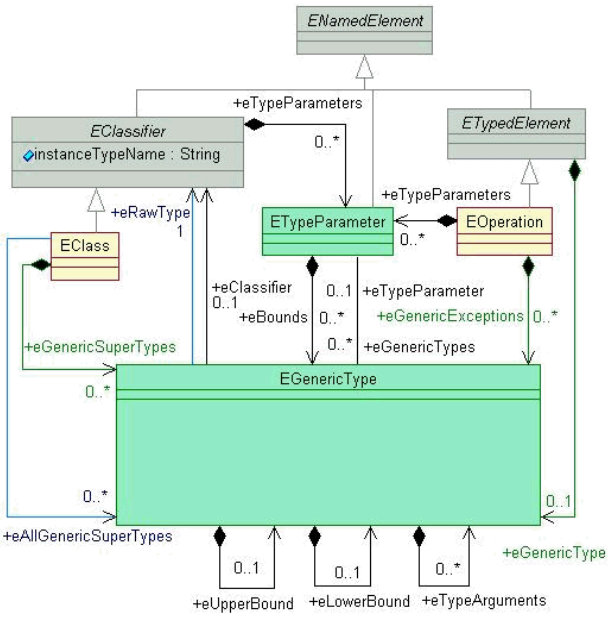


FIGURE 2.50 – Extrait du métamodèle Ecore

L'exemple suivant de la figure 2.51 montre comment un template est défini en Ecore et en Java, à partir de la définition d'un template UML.
Dans la partie gauche de cet exemple, une liste générique est définie sous la forme d'un template UML. On retrouve la *TemplateSignature*, permettant de spécifier le paramétrage d'un template en UML. Le type E correspond au paramètre du template. Ce paramètre porte le stéréotype *ETypeParameter* du métamodèle *Ecore*. Ainsi, dans la partie droite de la figure, la *eClass* générique *List* possède le *ETypeParameter* E. En Java, comme visible dans le coin droit de la figure, un template est défini comme une interface générique.
La substitution de paramètre d'un template s'effectue avec un *TemplateBinding* entre un contexte qui est ici la classe $Iterator\_E$ et un template (ici, *Iterator*).
Le stéréotype *EGenericType* permet de définir un type générique en Ecore. Dans le modèle UML, ce stéréotype est attribué à la classe $Iterator\_E$. Puisque l'opération *iterator* est typée par cette classe générique et que cette classe générique est substituée (*TemplateBinding*) par la classe paramétrée $Iterator < E >$ alors *iterator* est typée par $Iterator < E >$ dans le modèle Ecore.

Reusable Aspectual Models (RAM) L'approche présentée dans Klein & Kienzle [59], Kienzle et al. [58] et Bhalotia [16] permet de spécifier et tisser des modèles d'aspects définissant la structure et le comportement logiciel. Celle-ci propose des moyens permettant d'effectuer

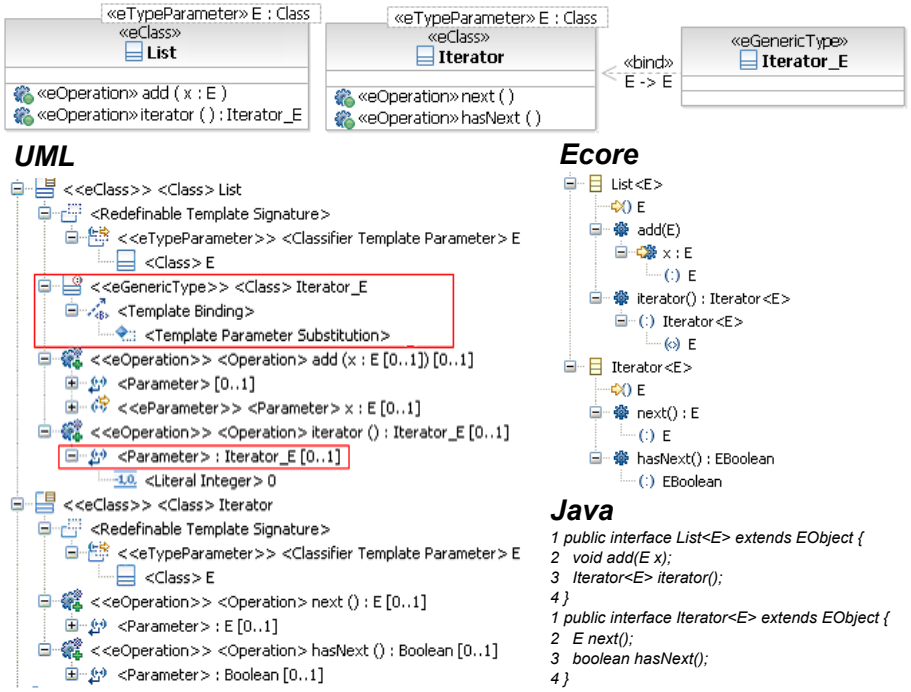


FIGURE 2.51 – Exemple de définition et de binding de template UML vers Java

des séquences de composition de templates.
Les aspects sont représentés par des templates. La structure l'est au travers de diagrammes de classes et le comportement par des diagrammes de séquences. Les paramètres sont symbolisés en utilisant une barre verticale juste avant le nom du paramètre et un résumé de tous les paramètres du template est effectué en les plaçant dans un rectangle en pointillé, situé en haut et à droite du template. L'exemple de la figure 2.52 présente un tel template.

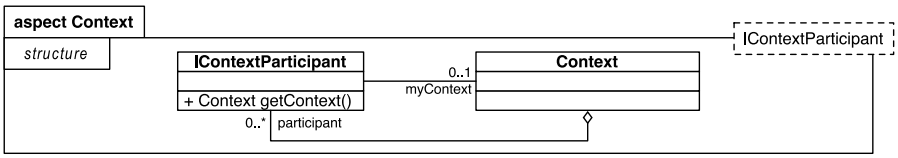


FIGURE 2.52 – Aspect sous forme de templates (Source : Klein & Kienzle [59])

La partie comportementale d'un aspect est représentée par un diagramme de séquences divisé en deux parties : le *pointcut* et l'*advice*. Tout comme le diagramme de classes, celui-ci peut aussi posséder des paramètres. En figure 2.53, le template *Copyable* possède ainsi un diagramme de séquences *clone*, avec quatre paramètres.
Afin de tisser les modèles d'aspect à un autre modèle ou *modèle d'application*, ces aspects sont instanciés en utilisant les paramètres des *templates UML* et des directives d'instanciation. Un *binding* est effectué entre les paramètres du modèle et ceux de l'aspect.
La figure 2.53 présente l'instanciation d'un aspect (*Checkpointable*) avec un autre (*Copyable*). Les instanciations à effectuer sont déclarées dans la partie basse gauche de *Checkpointable*. Par exemple, le paramètre */Copyable* de *Copyable* instancie le paramètre */Checkpointable* de

l'aspect\template *Checkpointable*.

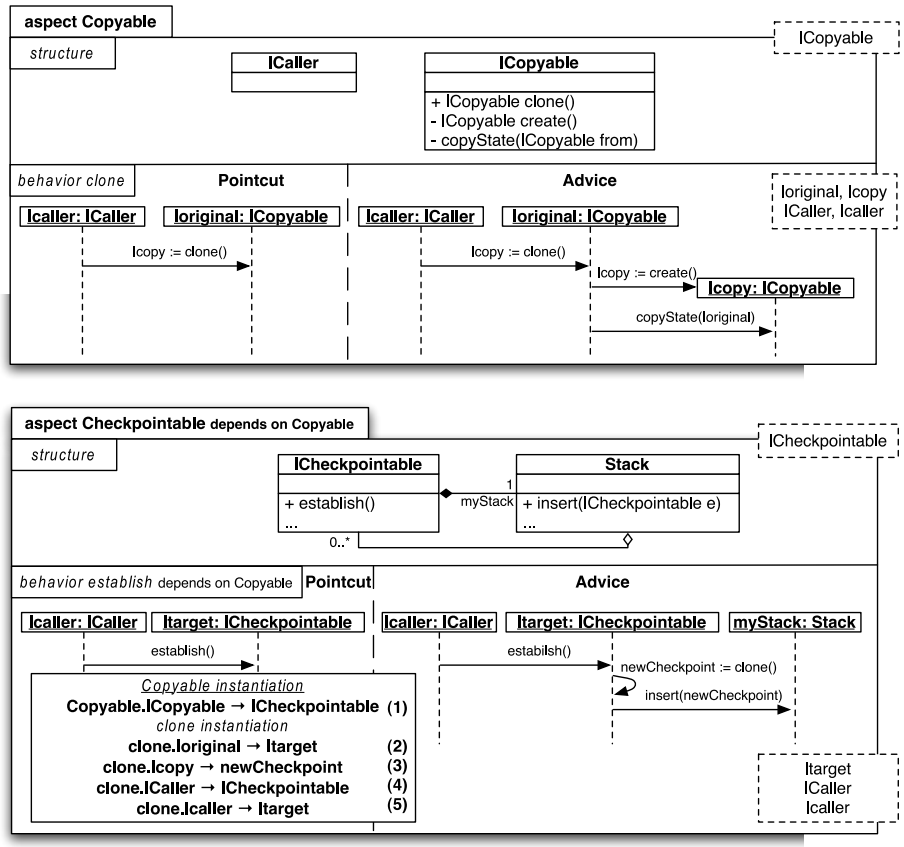


FIGURE 2.53 – Instanciation de templates (Source : Klein & Kienzle [59])

Lorsqu'il s'agit d'instanciations d'aspects par d'autres, comme en figure 2.53, un aspect dépend alors d'un autre. Pour cela, l'expression *depends* est utilisé. C'est le cas de *Checkpointable* dépendant de *Copyable*.
Pour tisser un aspect avec un modèle d'application (*contexte*), le même principe d'instanciation est utilisé, permettant ainsi d'obtenir un modèle spécifique au contexte. Une fois ce modèle obtenu, celui-ci est composé avec le modèle d'application.
Les auteurs expliquent que les dépendances entre les aspects doivent être résolues seulement lors de la phase de création du modèle final. Ainsi, aucun modèle intermédiaire indépendant n'est généré.
Bhalotia [16] étend cette approche en permettant de valuer plusieurs fois un paramètre, c'est-à-dire en autorisant le tissage multiple de différentes parties d'un template. Ceci permet de ne pas à avoir instancier plusieurs fois un même template, comme illustré en figure 2.54.
Pour spécifier le nombre d'instances minimales et maximales d'un élément du modèle d'aspect, les auteurs proposent d'utiliser la même syntaxe que celle utilisée pour les multiplicités d'associations UML. La figure 2.54 donne un exemple de ceci. Il est ici possible de créer plusieurs *Observer* pour un *Subject*, puisqu'*Observer* possède une multiplicité de *{1..*}* et *Subject*, une multiplicité de *{1}*. Les directives d'instanciations utilisées sont présentées dans la partie basse *(a)* de cette figure : on note l'utilisation des cardinalités, définis pour *Observer*, avec *PlayerStatsDisplay et BattleFieldDisplay* (chaque instanciation étant numérotées). On voit donc ici que ces instanciations permettent de réutiliser des parties de l'aspect *NavalBattle*, à savoir *PlayerStatsDisplay et*

BattleFieldDisplay : lors de la phase de tissage des deux aspects, seul *Observer* sera instancié deux fois.

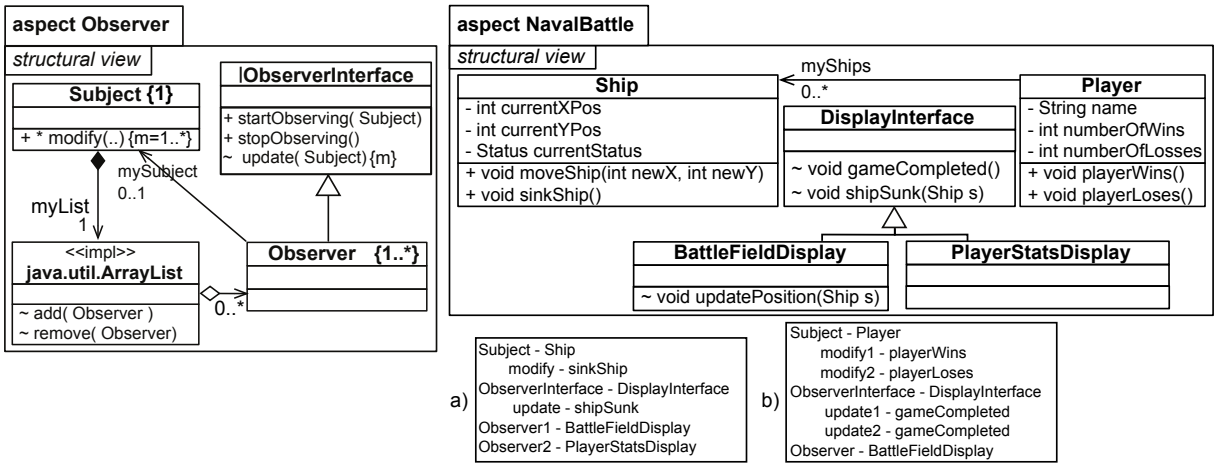


FIGURE 2.54 – Aspect réutilisable avec des cardinalités d'instanciations (Source : Bhalotia [16])

Lorsque le nombre d'instanciations d'un élément doit être égal à celui d'un autre, l'approche utilise la notion de variables avec cardinalités. On observe ceci dans la figure 2.54, avec la définition de la variable *m* pour la méthode *modify de Subject* : *{m=1..*}* et la réutilisation de cette variable au niveau de la méthode *update de ObserverInterface*. Ainsi, pour chaque instanciation de la méthode *modify de Subject*, la méthode *update de ObserverInterface* est instanciée. Le second exemple de directives d'instanciations (*(b)*) en figure 2.54 montre l'instanciation de *modify* avec *playerWins et playerLoses*. *update* est donc instanciée deux fois, ici avec *gameCompleted*. Les méthodes *playerWins et playerLoses* du sujet sont ainsi toutes deux liées à *gameCompleted*.

Model-snippets Ramos et al. [88] proposent un *framework* afin :

- d'assister à la construction de *patterns*,
- de détecter des instances des modèles paramétrés (*Model-snippets*) dans des modèles.

Les *patterns* présents dans un modèle sont exprimés sous la forme de *model-snippets*. Un *model-snippet* est une partie de modèle dont la construction est basée sur le métamodèle du modèle incluant le *pattern*. Ceci évite d'avoir à appréhender de nouveaux langages et facilite ainsi la spécification des *patterns*.

Un exemple est donné en figure 2.55, avec un *template UML*. Les constituants d'un *pattern* sont *(1)* le *model-snippet* et *(2)* une séquence d'éléments variables (comme *traceOp()* en figure 2.55) parmi les éléments de ce *model-snippet*. Quand les éléments variables sont substitués, on obtient alors un *modèle attendu*.

Les auteurs expliquent que les *model-snippets* et leur métamodèle (MM') sont plus abstraits que le *modèle attendu* et son métamodèle (MM), comme exposé en figure 2.56. En effet, l'objectif des *model-snippets* est de permettre, de par leur généricité, la définition de différents *modèles attendus*. De plus, toujours dans ce but :

- aucun invariant ni pré-condition n'est défini dans le métamodèle d'un *model-snippet*,
- tous les éléments sont optionnels,
- aucun éléments abstraits n'existe.

La recherche de *pattern* est effectuée pour :

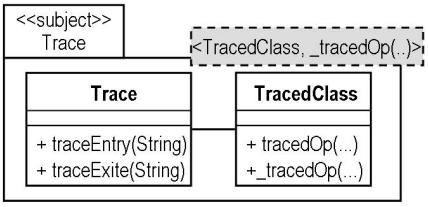


FIGURE 2.55 – *Template UML* d'une classe
(Source : Ramos et al. [88])

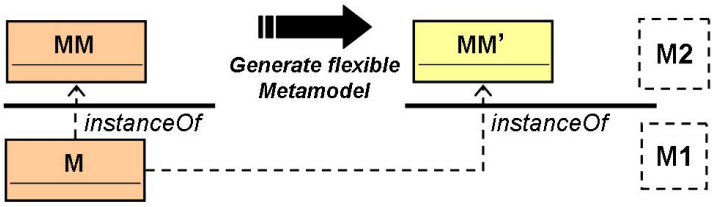


FIGURE 2.56 – Processus pour dériver un métamodèle de *snippets* de patterns (Source : Ramos et al. [88])

- détecter si un *pattern*, représenté sous la forme d'un *model-snippet*, existe dans un modèle. Les éléments variables ne sont pas considérés durant cette phase de recherche, *i.e.* la totalité du modèle est considérée. Ceci permet de trouver toutes les applications possibles du *pattern*,
- trouver toutes les variantes (*modèles attendus*) du *pattern* qui existent dans un modèle. Les éléments variables sont considérés durant cette phase de recherche. Chaque *modèle attendu* est défini selon une des applications possibles du *pattern* (model-snippet).

Directives de composition Certains auteurs (France et al. [49], Reddy et al. [89]) présentent une approche permettant de gérer la séparation des préoccupations (*concerns*) et plus précisément la séparation multidimensionnelle des préoccupations de conceptions ou *MDSoC* (*Multi-Dimensional Separation of Design Concerns*).
Les préoccupations sont représentées par un ou plusieurs aspects. Les auteurs définissent deux niveaux d'aspects : les *aspect models* et les *context-specific models*. Un *Aspect Model* représente une préoccupation (par exemple, un patrons de conception) et possède des paramètres, préfixés par le symbole "|". Un *context-specific model* résulte de l'adaptation (via les paramètres) d'un *aspect model* à un contexte particulier, nommé *primary Model*. Le *context-specific aspect* est ensuite tissé avec le *primary model*, via une fusion des éléments (identifiés par leurs noms). Un *composed model* résulte de ce tissage.
L'exemple de la figure 2.57, présente un contexte ((b) - *primary model*) et un aspect ((a) - *aspect model*), représentant un tampon d'écriture utilisé pour écrire en sortie. Le *binding* (cf. au-dessus du *context-specific model* - (c)) est effectué entre les paramètres de l'aspect model (|) et les éléments du primary model (On remarque que le binding permet de renommer un paramètre, lorsqu'aucun élément du primary model ne correspond. Dans l'exemple, il s'agit de |*Buffer* avec *WriterBuffer*). Le context-specific model, résultat de cette adaptation, est ensuite tissé avec le contexte, résultant en un *composed model* ((d)) intégrant la fonctionnalité d'écriture via un tampon.
Lors de la phase de fusion d'un context-specific model avec un primary model, le composed model

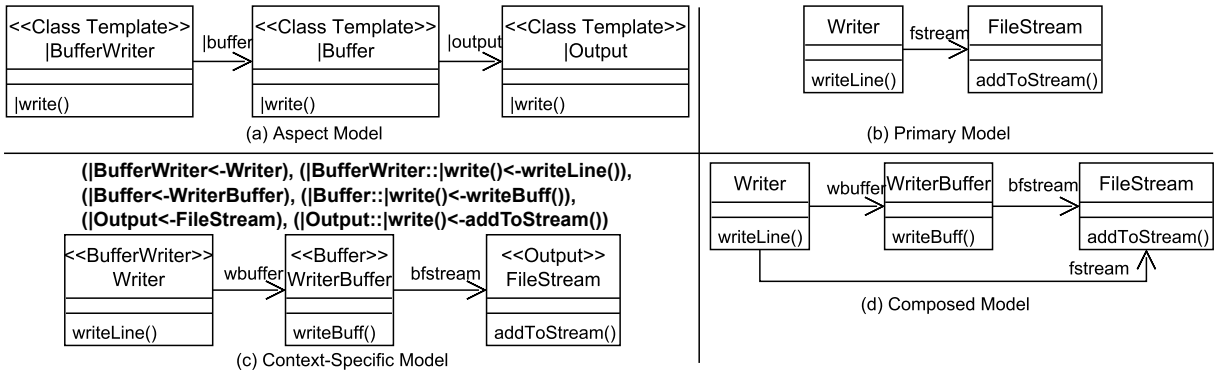


FIGURE 2.57 – Source : France et al. [49]

résultant de cette fusion peut être mal-formé ou ne pas correspondre entièrement aux besoins. Afin de résoudre de telles incohérences, les auteurs proposent des directives de composition. Celles-ci sont divisées en deux catégories :

- les directives sur les modèles. Elles déterminent l'ordre de composition des modèles. On peut citer les directives *"precedes"* ou *"follows"*, permettant de spécifier qu'un aspect model est composé avec un primary model respectivement avant ou après un autre,
- celles sur leurs éléments de modèles. Elles définissent la composition des éléments d'un context-specific model avec ceux d'un primary model. Il est par exemple possible d'ajouter ou de retirer des éléments à un primary model avant son tissage avec un context-specific model, avec respectivement les directives *"add"* ou *"remove"*. Pour cette dernière, nous présentons par la suite un exemple de son utilisation.

En figure 2.57, la fusion du primary model et du context-specific model résulte sur un modèle où l'opération *writeLine* de la classe *Writer* fait appel à l'opération *writeBuff* de *WriteBuffer* et l'opération *addToStream* de *FileStream*. Ceci n'est pas le résultat souhaité, puisque l'objectif est de découpler entièrement la classe *Writer* de *FileStream*, en passant par *WriteBuffer*. Pour cela, une directive est utilisée sur le primary model, avant la fusion. Il s'agit de la suppression ("remove") de l'association entre *Writer* et *FileStream*. Le résultat de cette suppression est visible dans la partie haute de la figure 2.58, permettant, lors du tissage avec le context-specific model, d'obtenir le composed model visible en partie basse.

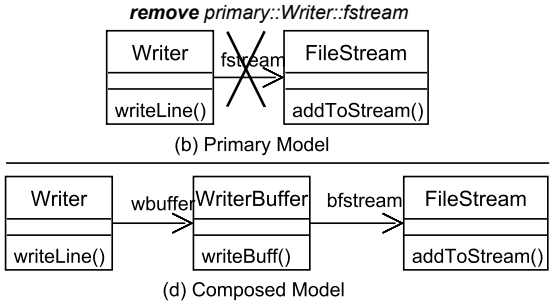


FIGURE 2.58 – *Primary model* et *Composed model* après utilisation de la directive *"remove"* (Source : Reddy et al. [89])

Templates aspectuels avec variantes L'approche présentée par Lahire et al. [63], Morin et al. [71] permet de définir des variantes de templates aspectuels par la définition de paramètres

et contraintes de composition optionnelles. Leurs travaux comportent aussi la vérification de la cohérence du paramétrage et du binding avant application du template. Un modèle d'aspect est composé de trois parties : un *graft model*, un modèle d'interface et un protocole de composition. La figure 2.59 donne un exemple de modèle d'aspect. Le graft model représente ce qui doit être tissé.

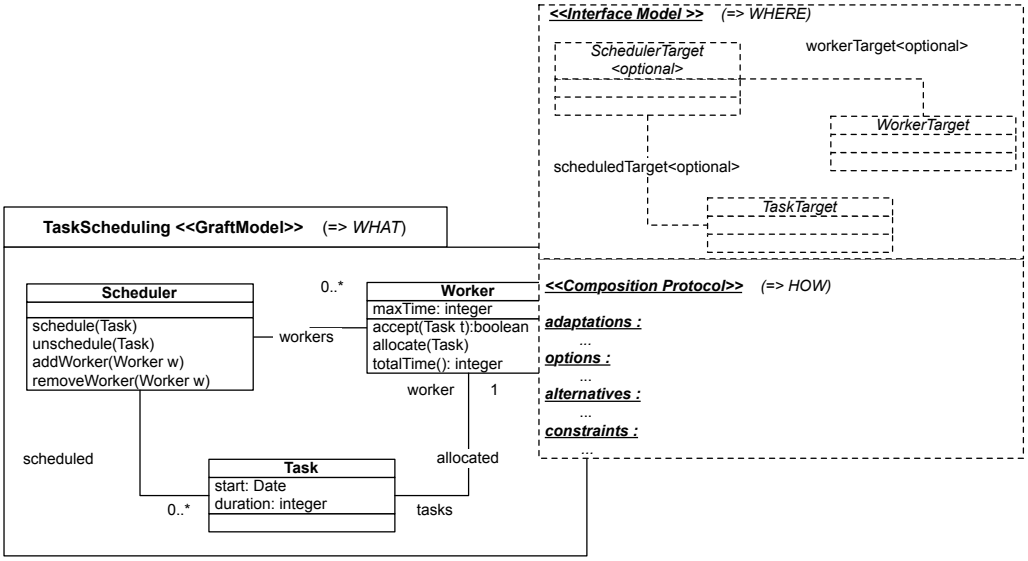


FIGURE 2.59 – Le template aspectuel *TaskScheduling* (Source : Morin et al. [71])

Le modèle d'interface représente le paramétrage, c'est-à-dire là où l'aspect doit être tissé. Des éléments peuvent être définis comme optionnels, avec le stéréotype *optional*, ce qui signifie qu'ils peuvent être substitués avec des éléments d'un modèle de base ou non. En figure 2.59, ce modèle d'interface est montré dans la partie en haut à droite. Elle déclare trois classes, dont une optionnelle, liées par des associations. Quant au protocole, il permet de définir la façon dont l'aspect est tissé. Cette composition est effectuée via des primitives de transformation de modèles, manipulant des éléments du graft model et du modèle d'interface. Dans ce protocole de composition, une transformation alternative indique qu'il existe différentes manières de composer le graft model dans le modèle d'interface, alors qu'une transformation avec option peut être effectuée ou non. Ce protocole est défini en dessous du modèle d'interface.

L'exemple de la figure 2.60 montre la composition par héritage du modèle d'aspect de la figure 2.59 avec le modèle *HealthWatcherSystem*. Cette composition par héritage est effectuée par la sélection des alternatives *VInheritanceTW*, *VIntroductionSched* et *VIntroductionTWInheritance* du protocole de composition de *TaskScheduling* et avec la définition d'un binding des éléments du modèle d'interface de *TaskScheduling* avec les éléments du modèle d'aspect : *Worker* avec *Employee*, *Task* avec *Complaint*. Ainsi, *Employee* et *Complaint* héritent respectivement de *Worker* et *Task*.

Pour éviter des modèles incohérents, les auteurs proposent d'ajouter des contraintes au protocole (en bas à droite, en figure 2.59). En figure 2.60, deux contraintes sont définies dans le cas d'une composition par fusion : *VMappingTWMerge* et *VMergingTW*. Par exemple, la seconde contrainte précise que si les classes *Worker* et *Task* sont fusionnées avec *Employee* et *Complaint*, il n'est pas possible de définir une association entre *Scheduler* et *Worker*, ainsi qu'une seconde association entre *Scheduler* et *Task*.

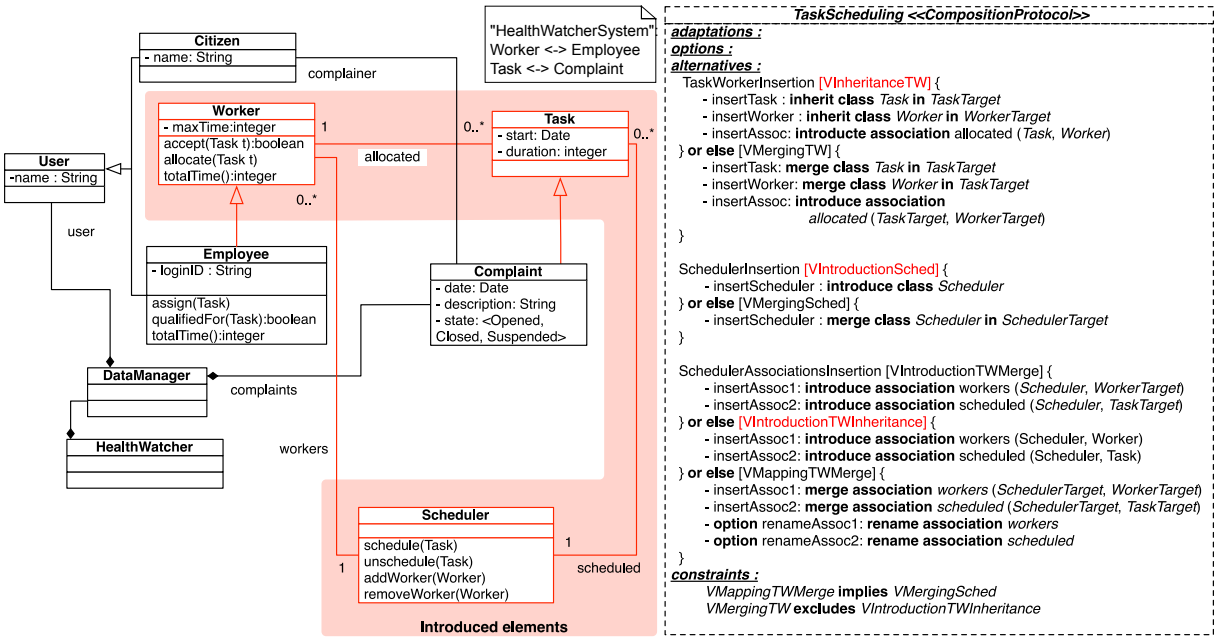


FIGURE 2.60 – Composition par héritage (Source : Morin et al. [71])

Semantic mixin layers Certains auteurs (de Lara & Guerra [36], Cuadrado et al. [34], de Lara & Guerra [37]) proposent de réutiliser, au niveau modèle, la notion de *concept* du language C++ (Sutton & Stroustrup [98]) [2], ainsi que la notion de *semantic mixin layers*, correspondant aux templates de métamodèles.

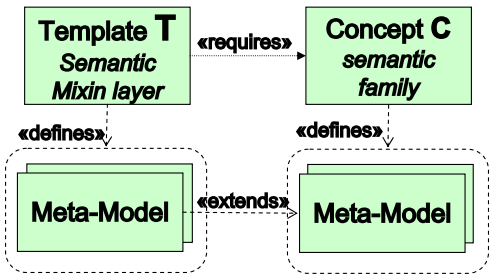


FIGURE 2.61 – *Semantic mixin layers* (template) et *concept* (Source : de Lara & Guerra [36])

Un template de métamodèle vise à définir un comportement générique pouvant être ajouté à un ensemble de métamodèles. Concernant le paramétrage, les *concepts* sont utilisés afin de définir ce que requiert un template afin d'être instancié (contraintes sur le paramétrage). Ainsi, les *concepts* permettent de déterminer une *famille* de métamodèles utilisables avec un template (cf. figure 2.61).
La figure 2.62 montre comment spécifier et instancier un template, utilisant un concept, dans le contexte des réseaux de Petri. Cette spécification est effectuée via *MetaDepth*, le framework de modélisation au sein duquel est implémentée l'approche des auteurs.
Le template *Buff2* (lignes 7 à 13) définit un buffer générique avec une transition en entrée et une autre en sortie. Ces deux transitions (*&Tri, &Tro*) et leurs processus respectifs (*&PNi, &PNo*)

2. Cette notion a été introduite afin de spécifier les contraintes structurelles et comportementales sur les paramètres des templates C++.

```
1  concept SimpleTrans(&M, &T) {
2   PetriNet &M {
3     Transition &T {}
4   }
5  } where $&T.inPl.size()=1 and
5            &T.outPl.size()=1$
6  // ---------------------------
7  template<&PNi,&Tri,&PNo,&Tro>
8   requires SimpleTrans(&PNi,&Tri),
8            SimpleTrans(&PNo,&Tro)
9  PetriNet Buff2 imports &PNi,&PNo{
10   Place Buffer {}
11   ArcPT (Buffer, &Tro);
12   ArcTP (&Tri, Buffer);
13 }
```

```
14 // ---------------------------
15 PetriNet TwoStateProc {
16   Place p1 {}
17   Place p2 {}
18   Transition t12 {}
19   Transition t21 {}
20   ...
21 }
22 // ---------------------------
23 TwoStateProc Producer;
24 TwoStateProc Consumer;
25 Buff2<Producer,Producer::t12,
26       Consumer,Consumer::t12>
26       ProducerConsumer;
```

```
1  PetriNet ProducerConsumer {
2    Place WP { name="waitProduce"; }
3    Place RP { name="ReadyProduce"; }
5    Transition ReadyP  { name="readyP"; }
4    Transition Produce { name="in"; }
6    ArcPT (RP, Produce);
7    ...
8    Place Buffer { name="Buffer"; }
9    ...
10   Place C  { name="Consume"; }
11   Place WC { name="waitConsume"; }
12   Transition Consume { name="out"; }
13   Transition ReadyC  { name="waitC"; }
14   ...
15 }
```

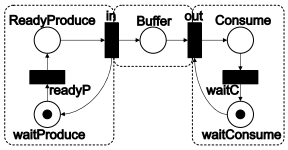


FIGURE 2.62 – Définition et utilisation d'un template de modèle (Source : de Lara & Guerra [36])

sont définis en tant que paramètres du template. Ensuite, le template importe les deux modèles (ligne 9), déclare une place (ligne 10) et la connecte aux transitions (lignes 11 et 12). De plus, le template requiert en ligne 8 que les paramètres satisfassent les contraintes exprimées par le concept *SimpleTrans*.
Le concept, défini dans les lignes 1 à 5, nécessite que la transition soit liée à une seule place en entrée et en sortie, via un ensemble de contraintes définies ligne 5. La classe TwoStateProc (lignes 15 à 20) définit un processus avec deux places. Les lignes 23 à 26 instancient le template *Buff2*. Le modèle *ProducerConsumer*, visible dans la partie droite de la figure, est un exemple de modèle résultant de l'instance du template *Buff2*.

2.2.3.3 Synthèse

Dans cette thèse, les templates sont au cœur de nos contributions. C'est pourquoi nous effectuons une synthèse des travaux sur les templates. À partir de l'étude effectuée précédemment, nous avons relevé plusieurs caractéristiques des approches à base de templates. Celles-ci sont présentées dans le tableau 2.1.
En ce qui concerne l'application aspectuelle, il s'agit de l'application d'un template sur un modèle afin d'enrichir ce dernier de fonctionnalités provenant du template. On note que des approches permettent à la fois d'effectuer une application aspectuelle et une application générative. Les travaux qui proposent des applications partielles autorisent la substitution d'une partie des paramètres.
Lors d'applications de templates, quelles soient partielles ou non, les modèles qui résultent de celles-ci peuvent être mal formés. Afin de prévenir ceci, plusieurs approches ont défini différentes règles de vérification des applications. Toujours dans un but de cohérence des applications, d'autres travaux ont spécifié des contraintes définissant le paramétrage sous la forme d'un modèle.
Les templates sont applicables sur des modèles mais sont aussi composables avec d'autres templates, créant ainsi des templates enrichis. Cette possibilité de composer des templates est notamment présentée par Catalysis et les travaux de Muller [73]. Cette dernière approche définit

aussi des propriétés d'ordres sur les compositions de templates. Ces propriétés garantissent la validité des séquences de compostions et leurs alternatives, sous certaines conditions.
La compatibilité avec UML concerne des approches comme celles de Clarke & Walker [32] et Cuccuru et al. [35]. Celles-ci sont compatibles dans le sens où elles étendent la notion de templates du langage. Ces travaux bénéficient du fait que les templates UML sont suffisamment généraux pour être utilisés de façon aspectuelle ou générique.
Enfin, le niveau de modélisation se rapporte à l'utilisation des templates au niveau modèle et\ou au niveau métamodèle. Au niveau modèle, les templates permettent de définir des patterns et des composants de librairies dans un langage de modélisation particulier et autorise, au niveau métamodèle, l'extension de ces langages.
L'analyse des approches précédentes nous permet d'orienter le travail de cette thèse. Tout d'abord, ces approches présentent des constructions de modèles à l'aide d'applications aspectuelles ou génératives de templates. Les travaux de Reddy et al. [89], Klein & Kienzle [59] et Cuccuru et al. [35] utilisent ces deux formes d'applications conjointement et les travaux de Ramos et al. [88], Steinberg et al. [96] et de Lara & Guerra [37] illustrent l'utilisation de l'application générative seule. Cependant, il serait utile de définir la relation entre l'application générative et l'application aspectuelle de façon plus précise. Nous étudierons cette relation dans le chapitre qui suit.
Ensuite, les approches étudient des situations où un template est appliqué sur un modèle et ne considèrent pas des applications de templates sur une hiérarchie de modèles [3] qui émerge lors de modélisations en équipe. De telles situations ont notamment lieu lors de modélisations en équipe et de l'utilisation de dépôts de modèles. Il serait utile de définir des règles d'ingénierie pour ces applications, notamment afin de garantir la validité d'une application sur un ensemble de modèles.
À partir de cet état de l'art, nous présentons dans le chapitre suivant notre proposition. Celle-ci présente différentes activités de modélisation, règles d'ingénierie et opérateurs pour les deux dimensions que sont la définition et la réutilisation des templates.

3. Ensemble de modèles liés entre eux, par exemple via une relation d'inclusion comme présenté en section 3.1.

TABLE 2.1 – Caractéristiques des applications de templates par approches

	Application aspectuelle	Application générative	Application partielle	Vérification des applications	Modèle paramètre	Composition de templates	Propriétés d'ordre	Compatible avec UML	Niveau de modélisation
Templates UML [78]	✓		✓	✓				✓	Modèle
Caron et al. [23]	✓			✓				✓	Modèle
Farinha & Ramos [43]	✓			✓				✓	Modèle
Cuccuru et al. [35]	✓	✓		✓				✓	Modèle
Themes [26]	✓			✓				✓	Modèle
Muller [73]	✓		✓	✓	✓	✓	✓	✓	Modèle
Catalysis [40]	✓		✓	✓		✓			Modèle
R.A.M. [59]	✓	✓	✓	✓		✓			Modèle
Ramos et al. [88]		✓		✓					Modèle
Reddy et al. [89]	✓	✓							Modèle
Smart Adapters [71]	✓		✓	✓					Modèle
EMF Generics [96]		✓							Modèle
de Lara & Guerra [37]		✓		✓					Métamodèle et Modèle
Aspectual Templates [103]	✓		✓	✓	✓	✓		✓	Modèle

Chapitre 3

Proposition

Les templates UML, présentés dans le chapitre précédent, ont été retenus car ils permettent de représenter des spécifications réutilisables et adaptables à des contextes applicatifs. Nous avons vu dans l'état de l'art que les templates peuvent avoir des usages aspectuels ou génératifs. Pour le premier usage, l'équipe a développé une extension des templates, les *aspectual templates*, visant à améliorer le contrôle de leur application. Cette extension repose sur l'utilisation d'un modèle pour le paramétrage. Elle est présentée en section 3.1.
En prenant comme point de départ ces aspectual templates, nous posons la question d'une ingénierie basée sur de tels templates. Cette ingénierie, nommée *Ingénierie Dirigée par les Modèles Basées sur les Templates* (*IDMBT*), est exposée en section 3.2. Ceci nous amènera à présenter les différentes activités de modélisation possibles et à les structurer, notamment vis-à-vis des utilisateurs (concepteurs de templates et concepteurs de systèmes).
Après la présentation de cette IDMBT, nous approfondissons trois des opérations permises par cette ingénierie. Premièrement, l'*instanciation*, étudiée en section 3.3. Elle permet la génération d'*instances* de templates à partir d'un contexte applicatif. La notion d'instance qui en résulte est également utilisée comme primitive pour d'autres opérations de l'ingénierie.
Les deux opérations suivantes, que nous examinons en section 3.4, concernent respectivement la détection et le retrait de templates. Ces opérations sont utiles pour la compréhension et l'évolution des modèles de systèmes à l'aide des templates.
Enfin, en dernière partie (section 3.5) de ce chapitre, nous étudions le lien entre les templates et les hiérarchies d'inclusion de modèles. Il s'agit de déterminer quelles sont les modalités concernant l'application d'un template sur une hiérarchie et les relations entre modèles résultats. Ces questions se posent en particulier dans les domaines de la modélisation incrémentale, en équipe et du versioning.

3.1 Fondements

Dans cette section, nous présentons les notions de sous-modèles [24] (section 3.1.1) et d'aspectual templates [103] (section 3.1.2). La première notion nous permettra d'expliquer la structure d'un aspectual template qui se décompose en deux sous-modèles : le *modèle paramètre* et le *modèle spécifique*.

3.1.1 Sous-modèles

En sous-section 3.1.1.1, nous exposons la structure d'un modèle, formulée dans Carré et al. [24], permettant de manipuler des modèles ou des bouts de modèles de façon homogène. Cette structure est représentée par un ensemble d'éléments et un ensemble de contraintes de dépendances entre ces éléments.
Elle permet de caractériser les sous-modèles et leurs relations d'inclusion. Ces dernières dépendent de la variation des contraintes structurelles d'un modèle par rapport à un autre. Le résultat majeur de Carré et al. [24] est la transitivité d'inclusion entre modèles. Pratiquement, le besoin de

transitivité est essentiel en IDM car elle permet de garantir des qualités de modularité dans les espaces de modèles et donc de localité et de composabilité de leurs traitements, pour un meilleur contrôle et plus d'efficacité.
Puis, nous rappellerons en section 3.1.1.2 l'opération d'*extraction de sous-modèles* à partir de la sélection d'un sous-ensemble d'ingrédients d'un modèle.

3.1.1.1 Structure d'un modèle et sous-modèle

Structure d'un modèle
Dans Carré et al. [24] un modèle est défini comme un ensemble d'ingrédients de modélisation muni d'un ensemble de contraintes de dépendances structurelles entre ses ingrédients, formant ainsi un graphe de dépendances. En figure 3.1 un exemple de modèle (*HeatingSystem* à gauche) est montré avec sa représentation (à droite) sous la forme de graphe de dépendances.

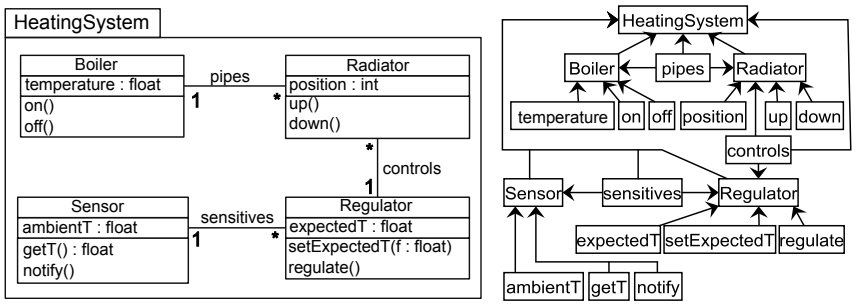


FIGURE 3.1 – Modèle *Heating System* et son graphe de dépendances

Ce formalisme autorise aussi bien la représentation de modèles que de bouts de modèles. Au sens d'UML, ces modèles peuvent être bien formés ou mal formés. Ceci est illustré en figure 3.2 où l'attribut *temperature* n'appartient à aucune classe.

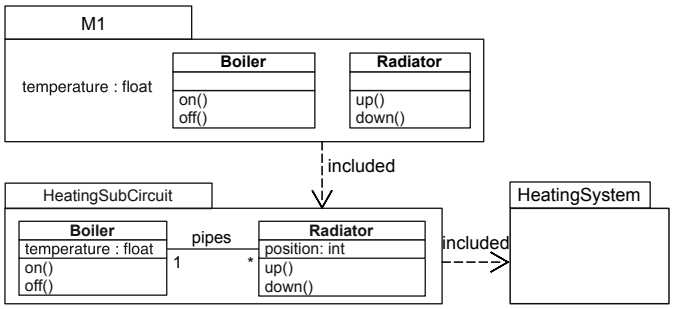


FIGURE 3.2 – Sous-modèles

Sous-modèles
Un modèle M est *inclus* dans un modèle M' (M sous-modèle de M') quand ce dernier contient les ensembles d'éléments et de contraintes structurelles de M. Dans la figure 3.2, $M1$ est un sous-modèle mal formé de $HeatingSubcircuit$ et ce dernier est le sous-modèle bien formé du modèle $HeatingSystem$ de la figure 3.1.

3.1.1.2 Extraction de sous-modèles

L'opération d'extraction présentée dans [24] consiste à produire le sous-modèle d'un modèle M à partir de la sélection d'un ensemble d'éléments de M.

En figure 3.3, en utilisant le modèle *HeatingSystem* de la figure 3.1 et le sous-ensemble d'éléments suivant : *Sensor, ambienT, getT, notify, sensitives, Regulator, expectedT, setExpectedT et regulate*, le sous-modèle *RegulationSubCircuit* est obtenu.

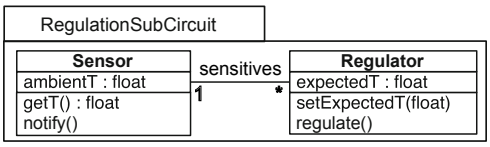


FIGURE 3.3 – Sous-modèle *RegulationSubCircuit* extrait de *HeatingSystem*

Cette opération est particulièrement utile dans l'IDMBT pour :

- extraire les sous-modèles constitutifs d'un aspectual template intervenant dans leur processus d'application : le sous-modèle *paramètre* et son complément qui sera désigné *sous-modèle spécifique* (cf. section 3.1.2.1),
- circonscrire, dans une application d'un aspectual template à un système particulier, le sous-modèle de ce système, objet de l'application, nommé *modèle actuel* (cf. section 3.1.2.2).

3.1.2 Aspectual Templates

Comme vu au chapitre 2, diverses approches et techniques se sont inspirées du monde de la programmation afin d'améliorer la réutilisabilité des modèles, telles que la paramétrisation. La technologie *UML* contribue beaucoup à ceci, notamment au travers de son concept de template de modèle. Celui-ci est suffisamment général et permissif pour représenter la plupart des besoins. En ce qui concerne l'utilisation des templates, nous allons montrer que le paramétrage spécifié par le standard peut être contraint pour que les paramètres constituent un modèle bien formé.
Ceci conduit à la définition des aspectual templates, possédant des paramètres qui sont des modèles. Le standard étant suffisamment général, il a été possible de le renforcer sémantiquement, permettant ainsi de garantir que les aspectual templates sont des templates UML de plein droit (permettant ainsi tout autre traitement favorisé par le standard).
Les templates UML permettent le *binding* partiel, *i.e.* lorsque seule une partie des paramètres est substituée et que ceux non substitués restent paramètres dans le modèle résultat, faisant de lui un template. Dans le cas des aspectual templates, nous verrons comment le *binding* partiel est pris en compte.

3.1.2.1 Constituants : (Sous-)modèles paramètre et spécifique

Les templates UML ont des paramètres qui représentent les éléments requis pour leur application sur des modèles. Le standard considère ces paramètres de manière isolée mais ceci n'est pas suffisant lorsqu'on souhaite représenter la structure d'un modèle requis pour l'application d'un template à un système.
Ceci est illustré dans la partie gauche (a) de la figure 3.4, avec un template de *package* représentant le pattern *Observer*.
Les éléments paramètres du template ne forment pas un modèle. En effet, la propriété *value* est exposée sans la classe qui la contient (*i.e. Subject*), alors que cette dernière est requise pour permettre la substitution de *value* avec une propriété contenue dans une classe appartenant à un

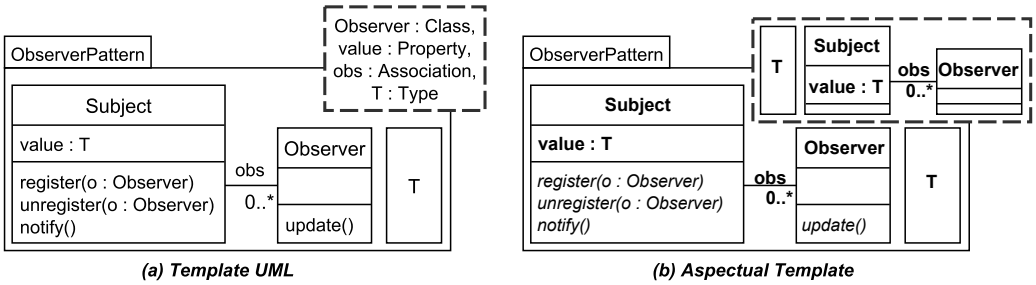


FIGURE 3.4 – Template UML et Aspectual Template

bound model. De la même manière, l'association *obs*, exposée en tant que paramètre, est sous-spécifiée puisque l'un de ses membres (la classe *Subject*) n'est pas déclaré en tant que paramètre. Dans Vanwormhoudt et al. [103], nous avons proposé une amélioration compatible avec les templates UML : les aspectual templates. Avec ceux-ci, les paramètres sont représentés sous la forme d'un *modèle bien formé*, remplaçant la liste de paramètres du standard afin d'éviter les incohérences de paramétrage telles que celle présentée plus haut.
La partie droite (*b*) de la figure 3.4 montre ceci. Les paramètres du template doivent être complétés pour former le modèle requis pour l'application, *i.e.* le *modèle paramètre*. Celui-ci définit la structure attendue du *bound model* (deux classes connectées par une association et un attribut de type T), afin d'injecter correctement les fonctionnalités du template.
Dans un aspectual template, on distingue :

- le *core*, correspondant au modèle complet du template,
- le sous-modèle *paramètre*,
- le sous-modèle *spécifique*, qui est le complément du sous-modèle paramètre par rapport au core.

Le sous-modèle spécifique contient les fonctionnalités qui seront injectées lors des applications.
La figure 3.5 illustre ces constituants, avec le *template ObserverPattern*. Dans celui-ci, les méthodes d'enregistrement et de mise à jour des *observers* (en italique dans le template) forment le modèle spécifique. Celui-ci peut être bien ou mal formé et nous examinerons en section 3.3 les différents cas possibles lors d'application de templates.

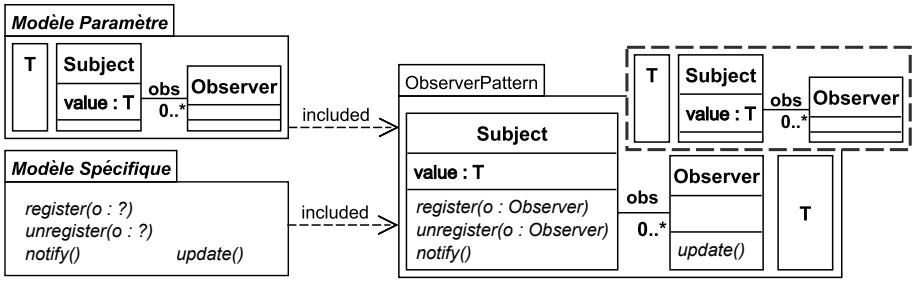


FIGURE 3.5 – Constituants d'un aspectual template :
Sous-modèle paramètre et sous-modèle spécifique

3.1.2.2 Application d'aspectual templates

Pour exprimer l'application d'un aspectual template à un contexte, nous utilisons l'opérateur *apply*, présenté dans Muller [73]. Cet opérateur enrichit le contexte en tissant le modèle spéci-

fique du template, au travers des substitutions déclarées. Cet opérateur part du template vers le contexte.
La figure 3.6 illustre ceci, avec l'application du *template ObserverPattern* sur le modèle *Agency-Car*, le *contexte* et montre son enrichissement dans le modèle résultat (*bound model*). À noter qu'en UML, le *bound model*, en relation *bind* avec le template, désigne à la fois le contexte et le résultat.
Pour les applications d'aspectual templates, il est nécessaire de prendre en compte la contrainte que le paramètre forme un modèle. L'opérateur *apply* permet de garantir ceci. Ainsi, pour qu'une application aspectuelle soit valide, les éléments d'un contexte utilisés comme arguments d'une substitution (*actuals*[1]) doivent former un modèle, le *modèle actuel*, structurellement conforme au modèle paramètre de l'aspectual template. Ce qui signifie que si un paramètre du template est en relation avec un autre de ses paramètres (*e.g.* un attribut contenu dans une classe), la même relation doit être présente entre les éléments actuels auxquels ils sont substitués. Cette conformité structurelle entre le modèle paramètre et le modèle actuel est assurée par un ensemble de contraintes, présentées dans Vanwormhoudt et al. [103].
La figure 3.6 illustre cette conformité. Le *template ObserverPattern* est appliqué sur le contexte *AgencyCar*.

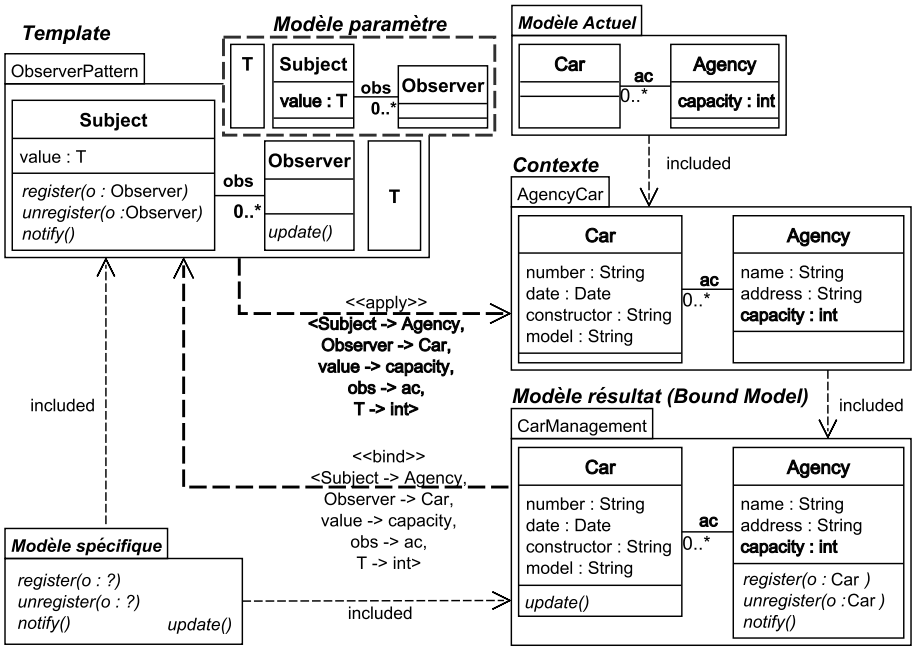


FIGURE 3.6 – Application du pattern Observer
Composition du contexte et du modèle résultat

La structure formée par le modèle paramètre est préservée par celle du modèle contenant les éléments actuels : dans *ObserverPattern*, *obs* est relié à *Subject* et *Observer*, et est substituée avec *ac*, relié à *Agency* et *Car*, tous deux substituants respectivement *Subject* et *Observer*. Il en va de même avec *value*, lié au type *T* et à la classe *Subject* et qui est substitué par *capacity*, lié au type *int* et à *Agency*.
Le *modèle actuel* (en haut à droite de la figure 3.6) peut être extrait du contexte applicatif en utilisant l'opération d'extraction avec les éléments actuels spécifiés dans l'ensemble de

1. Voir *Templates UML*, en chapitre 2

substitutions. Ce modèle actuel sera particulièrement utilisé lors de la définition de règles d'ingénierie concernant l'application d'un aspectual template sur une hiérarchie d'inclusion de modèles (cf. section 3.5).

Structure du modèle résultat

Comme indiqué précédemment, les fonctionnalités ajoutées lors d'une application sont représentées par le (sous-)modèle spécifique. Dans l'exemple de la figure 3.6, le modèle résultat *CarManagement* inclut le modèle spécifique (en bas à gauche) provenant d'*ObserverPattern*, le contexte *AgencyCar* et les contraintes structurelles liant ces deux sous-modèles, *e.g. register* avec *Agency* et *Car*.

Le modèle spécifique est injecté au contexte via un ensemble de substitutions. Il est important de noter que l'opération d'application ne modifie pas le contexte mais génère un nouveau modèle qui est le modèle résultat. Celui-ci inclut :

- Le contexte,
- Le modèle spécifique,
- Les contraintes de dépendances structurelles reliant le modèle spécifique au modèle actuel, selon les substitutions effectuées.

L'injection du modèle spécifique dans ce modèle résultat a pour conséquence l'adaptation des éléments du modèle spécifique (par exemple, l'adaptation du type d'une propriété lorsque le type fait partie du modèle paramètre et est substitué). On voit notamment ceci dans *CarManagement* avec l'opération *register* et son paramètre *Observer*, adapté pour devenir *Car* en figure 3.6.

Composition d'aspectual templates

Les applications aspectuelles présentées jusqu'ici concernaient l'application d'*aspectual templates* sur un modèle, permettant ainsi d'obtenir un modèle enrichi avec les fonctionnalités du template. Tout comme avec les templates (cf. chapitre 2), il est possible de composer des aspectual templates entre eux, *i.e.* appliquer un aspectual template sur un autre, créant ainsi des templates enrichis. L'exemple d'une telle composition est présenté en figure 3.7.

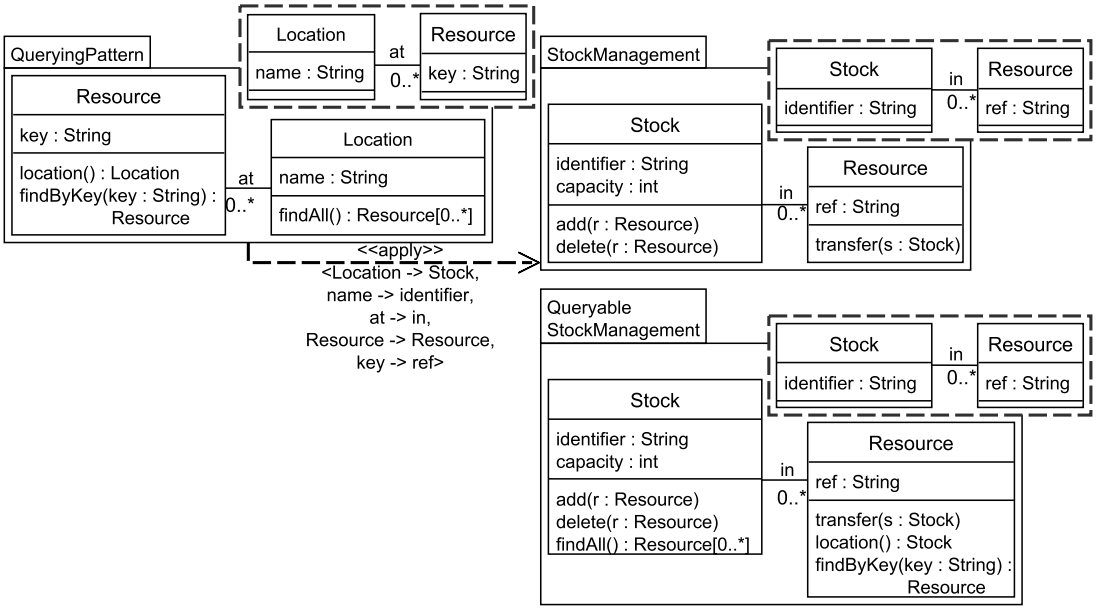


FIGURE 3.7 – Composition d'aspectual templates

L'application de *QueryingPattern* sur l'*aspectual template StockManagement*, représentant un stock et ses ressources, a permis d'ajouter des fonctionnalités de requêtage aux éléments *Stock* (ajout de *findAll*) et *Resource* (ajout de *location* et *findByKey*) qui sont paramètres.

Assemblage de templates

Les templates obtenus par composition peuvent être à leur tour appliqués et constituer des assemblages de templates. La figure 3.8 présente un tel assemblage.

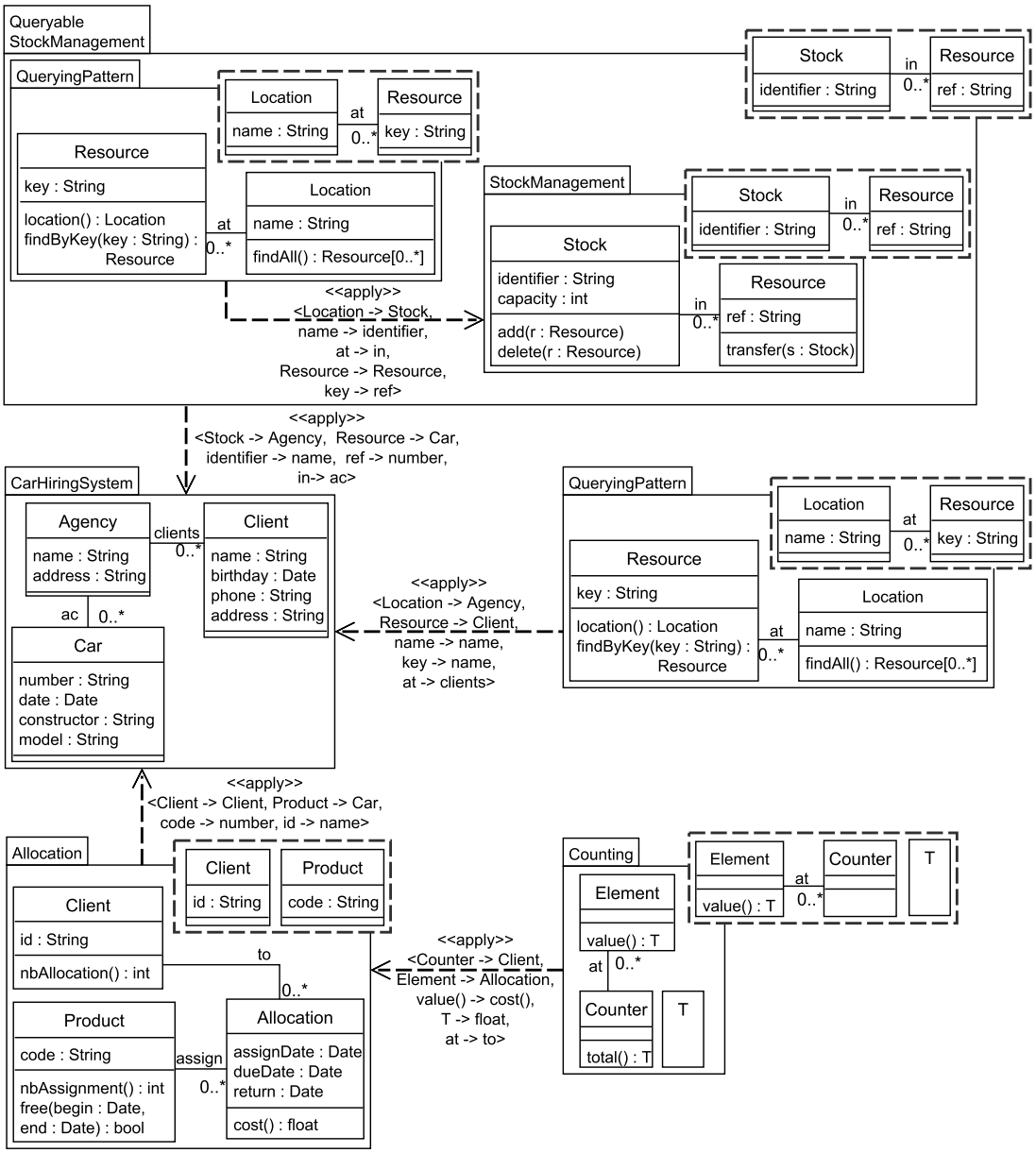


FIGURE 3.8 – Exemple d'assemblages de templates

Le template *QueryableStockManagement*, issu de la composition des templates *QueryingPattern* et *StockManagement*, est appliqué pour injecter la fonctionnalité de gestion des stocks avec recherche au système (*CarHiringSystem*). Une autre composition de templates est illustrée par la séquence d'applications *Counting* sur *Allocation* sur *CarHiringSystem*.

Ordre d'applications des aspectual templates

À partir de l'étude de séquences d'applications de templates, Muller [73] a déterminé des propriétés concernant l'ordre des applications et l'influence de celles-ci sur le modèle résultat. Ces propriétés sont les suivantes :

- Applications successives sur un contexte : Il est possible d'appliquer plusieurs aspectual templates les uns après les autres sur le contexte. L'ordre des applications n'aura aucune influence sur le résultat si les arguments actuels des ensembles de substitutions appartiennent au contexte. Ceci est illustré en figure 3.9. Que le template AT soit appliqué en premier sur le contexte C (*(1)*) ou en deuxième sur ce dernier *(2)*, le modèle résultant de cette séquence d'applications restera identique si les arguments actuels de S et S' appartiennent à C.

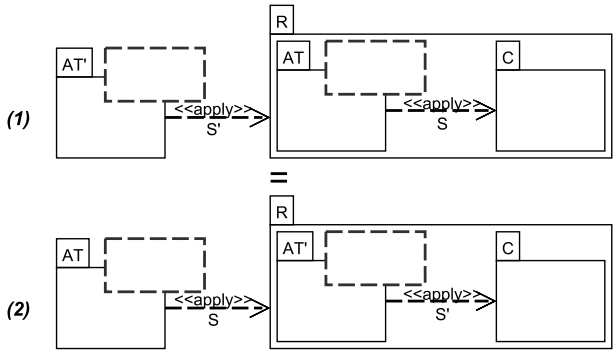


FIGURE 3.9 – Applications de templates sur un contexte

- Composition d'aspectual templates et application à un contexte : Pour que les résultats des deux séquences d'applications suivantes soient identiques :
  1. AT appliqué sur le résultat de l'application de AT' sur C,
  2. AT' composé avec AT, produisant un template appliqué sur un contexte C,

  il faut que les arguments actuels de S soient contenus dans AT'.

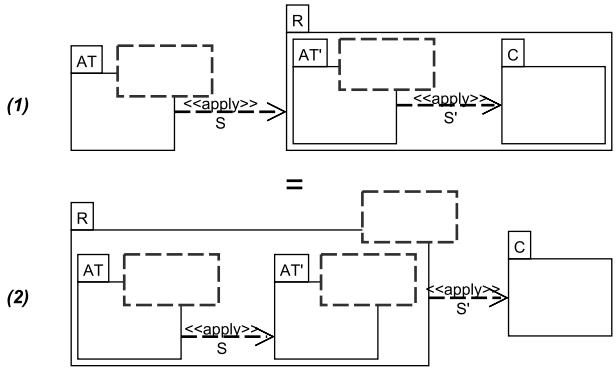


FIGURE 3.10 – Composition de templates et application à un contexte

Application partielle d'aspectual templates

Pour rappel (section 2), le standard définit la notion d'*application partielle*. Nous l'avons enrichie pour tenir compte du statut particulier des aspectual templates. Comme dans le standard, les

paramètres non-substitués sont propagés afin de former le modèle paramètre du template résultat. Ceci permet de définir de nouveaux templates possédant des modèles paramètres plus riches. Pour assurer la cohérence des applications partielles et la bonne formation du modèle paramètre, des contraintes supplémentaires ont été définies (Vanwormhoudt et al. [103]).
Afin d'avoir un aperçu de telles applications, considérons l'aspectual template *ObserverPattern* et son application sur le *template StockManagement*, tels qu'en figure 3.11.

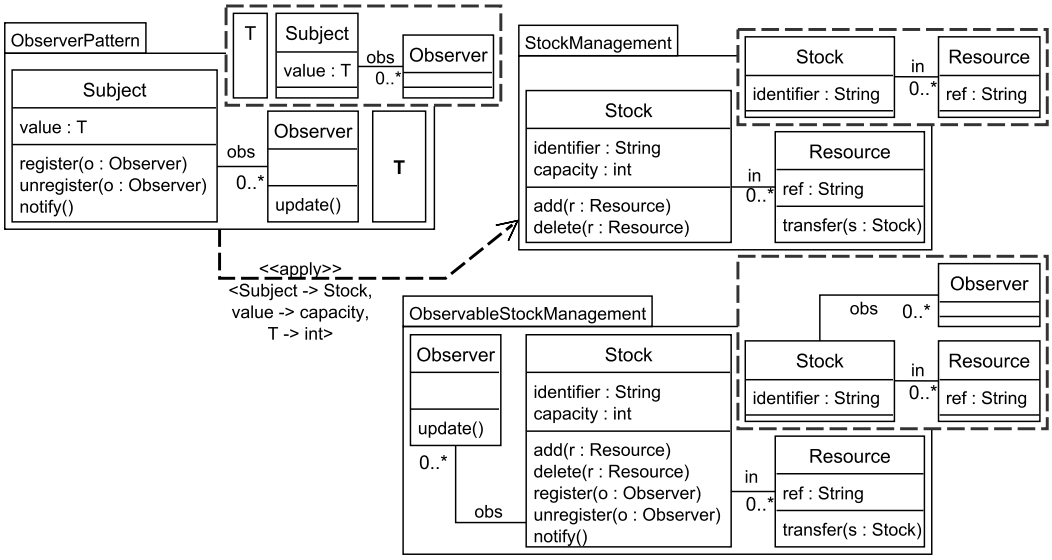


FIGURE 3.11 – Exemple de paramètres (classes et associations) non-substitués

Le but de ceci est d'obtenir un nouveau template exprimant qu'un stock peut être observé. Seul un sous-modèle du modèle paramètre de *ObserverPattern* est substitué : aucun des éléments de *StockManagement* ne joue le rôle d'*Observer*.
La manière dont les paramètres non-substitués sont gérés lors de telles applications a été établie. Plusieurs stratégies sont possibles : les ignorer ; les inclure dans le *core* du template résultat en tant qu'éléments non-paramètres ; les propager dans le modèle paramètre du template résultat (comme spécifié par le standard - cf. chapitre 2).
Comparée aux deux autres, outre qu'elle respecte le standard, la dernière stratégie offre deux principaux avantages. Premièrement, elle respecte le statut d'ordre supérieur des paramètres de template, par rapport aux autres éléments de modèle. Pour rappel, l'objectif des paramètres est d'abstraire les éléments attendus d'un modèle candidat afin d'intégrer les fonctionnalités du template. Les autres stratégies brisent ce principe. Deuxièmement, elle permet d'obtenir de nouveaux templates. En effet, les modeleurs peuvent alors effectuer des assemblages de modèles, *i.e.* réutiliser le template obtenu dans une autre application aspectuelle.
Le template résultat *ObservableStockManagement*, en figure 3.11, illustre cette stratégie. On remarque les deux choses suivantes. Premièrement, l'association *obs* et la classe *Observer* non-substituées ont été insérées dans le *core* du template mais gardent leur statut de paramètres puisqu'elles ont été injectées dans le modèle paramètre de *ObservableStockManagement*.
Deuxièmement, cette insertion a permis d'obtenir un template avec un modèle paramètre plus riche. Ces paramètres ajoutés devront être substitués lors d'applications du template résultat, *ObservableStockManagement*. Plus généralement, en suivant la stratégie du standard, l'ensemble des paramètres du nouveau template est défini par l'union des modèles paramètres du contexte

et ceux non-substitués du template appliqué.
Afin de respecter la sémantique des aspectual templates, il est essentiel que le paramétrage enrichi forme un modèle cohérent. L'exemple d'application partielle précédent illustre un cas où des classes et associations paramètres ne sont pas substitués. D'autres éléments, tels les attributs et opérations contenus dans les classes, peuvent aussi être non-substitués, même lorsque la classe les contenant l'est.
La figure 3.12 illustre ceci. Le *template QueryingPattern* de la figure 3.8 est modifié : les attributs *date* et *adress* sont ajoutés en tant que paramètres, de façon à ce que la fonctionnalité de recherche de ressources par date ou lieu puisse être adaptée. Ce template est appliqué sur *ObservableStockManagement*, afin d'obtenir une gestion de stocks avec des fonctionnalités de requêtage.

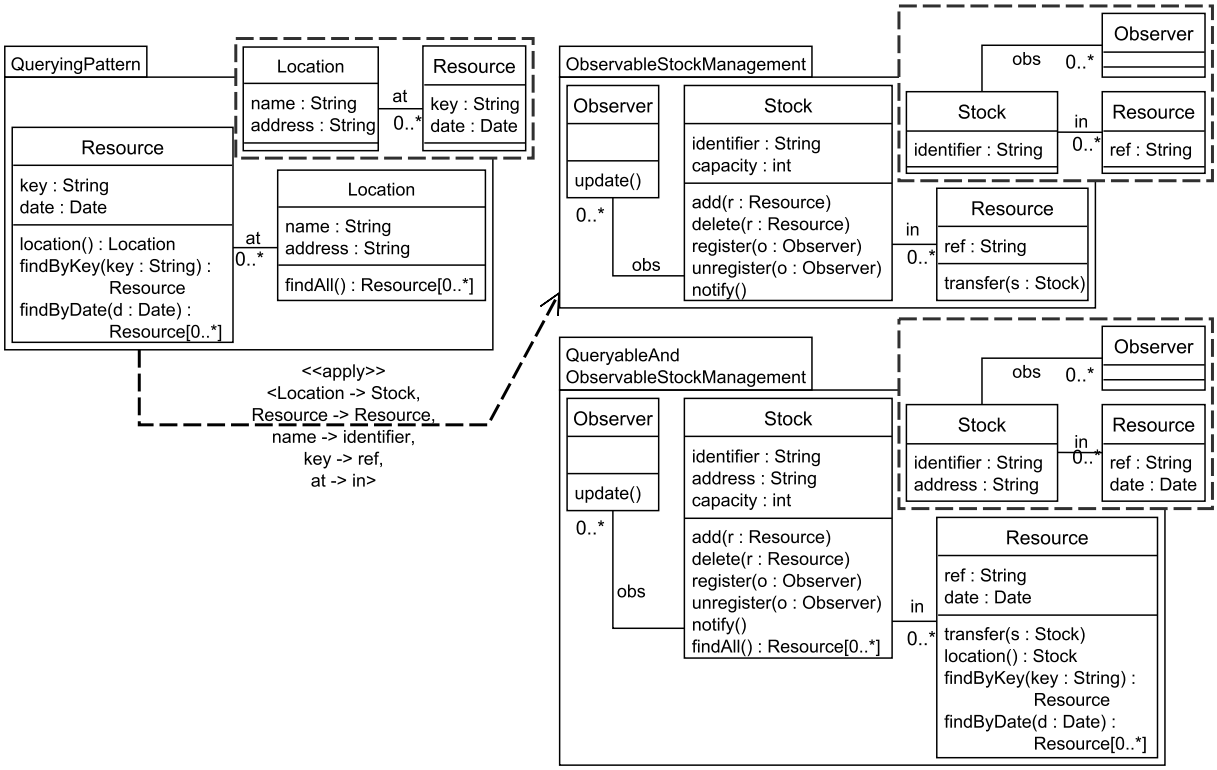


FIGURE 3.12 – Exemple de paramètres (attributs) non-substitués

address et *date* ne sont pas substitués et sont inclus dans le modèle paramètre du template résultat. En effet, ces attributs restent nécessaires afin de permettre la recherche d'une ressource par date ou lieu. En considérant la cohérence du modèle paramètre résultat, les attributs et opérations non-substitués requièrent que les classes les contenant dans le modèle résultat soient paramètres (cf. la substitution de *Location* par *Stock* en figure 3.12). Ceci est essentiel afin d'assurer que de tels paramètres soient dans une classe du modèle paramètre résultat.
Avec ces applications successives partielles (figure 3.11 et 3.12), nous obtenons un aspectual template plus riche. Ce template combine plusieurs fonctionnalités d'observation, de recherche et de gestion de ressources. Il peut être capitalisé dans un dépôt en tant que modèle réutilisable à valeur ajoutée et être par la suite appliqué pour la construction de systèmes.
Dans cette section, nous avons présenté les aspectual templates. Nous avons vu qu'ils possèdent des propriétés intéressantes permettant la construction de systèmes. Partant de là, nous allons étudier une ingénierie basée sur de tels templates.

3.2 Ingénierie Dirigée par les Modèles Basée sur les Templates

Au cours de la section précédente et du chapitre sur l'état de l'art (cf. chapitre 2), nous avons présenté le concept de template et nous en avons montré l'intérêt pour la construction de systèmes. Dans cette section, nous proposons une ingénierie basée sur ce concept en privilégiant celui d'aspectual templates. Pour définir cette ingénierie, nous identifions les artéfacts manipulés et produits par celle-ci, les activités qu'elle sous-tend et les acteurs qu'elle concerne. Cette ingénierie s'appuie sur une structuration en trois parties qui est présentée en figure 3.13.

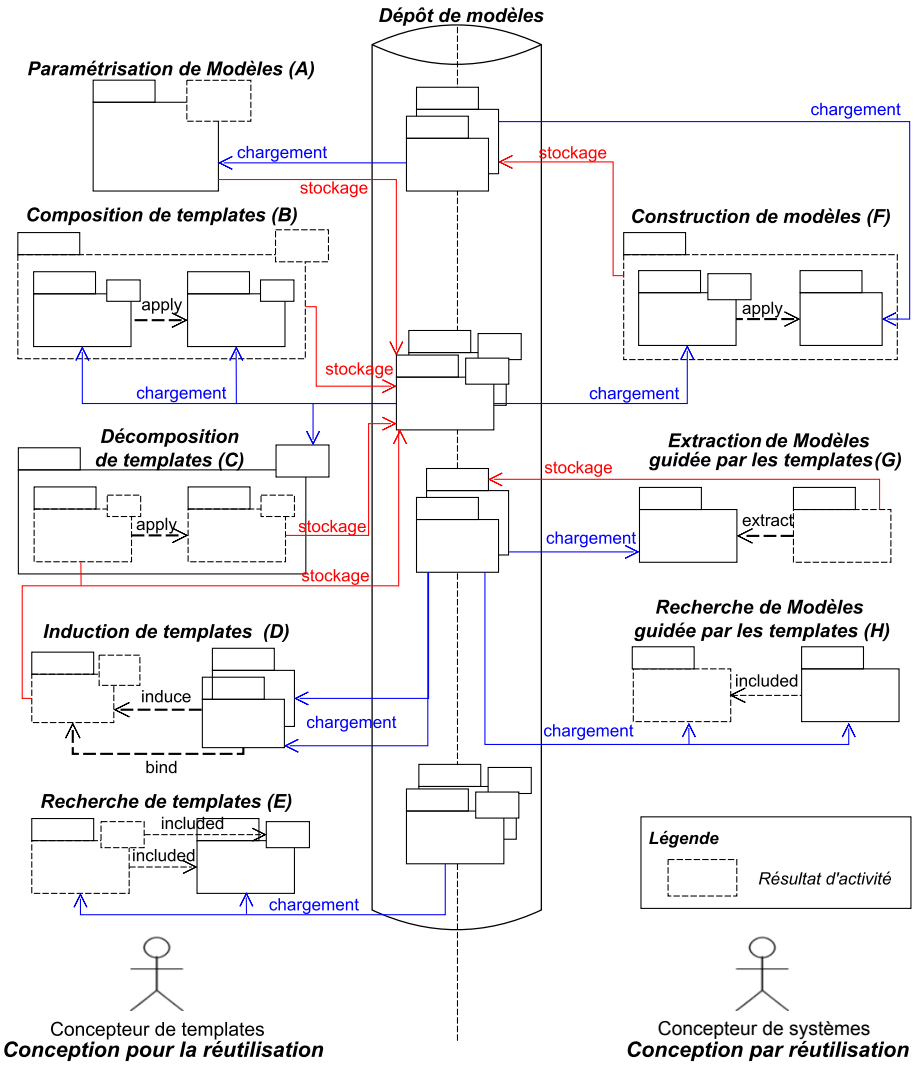


FIGURE 3.13 – Ingénierie Dirigée par les Modèles Basée sur les templates

Au centre, se trouve un dépôt de modèles qui sert à stocker les différents artéfacts de l'ingénierie et qui les rend disponibles à ses acteurs. Ces artéfacts sont des templates, des modèles complets de systèmes mais aussi des morceaux de modèles, qu'il est intéressant de conserver pour certaines activités de l'ingénierie ou de nouveaux projets. Au sein de ce dépôt, ces artéfacts sont organisés pour faciliter leur recherche et la navigation.
Associé à ce dépôt, nous identifions deux espaces de conception. Le premier, situé à gauche de la figure, a comme objectif l'élaboration de templates notamment à partir de modèles ou à partir d'autres templates. Cet espace est destiné à des acteurs qui jouent le rôle de "concepteur de

templates". En général, ces acteurs sont des concepteurs expérimentés ayant déjà eu à résoudre un grand nombre de problèmes de conception par le passé et sont donc plus à même de concevoir des solutions génériques (comme des patrons de conception ou des frameworks) à l'aide de templates. Dans cet espace, les activités se concentrent sur la spécification de templates et leur composition mais aussi leur adaptation à de nouveaux besoins et de nouveaux domaines applicatifs.
Le second espace d'activité, présenté dans la partie droite de la figure 3.13, vise la construction d'applications et de systèmes métiers en s'appuyant sur les templates et les modèles offerts par le dépôt. Les acteurs intervenant dans cet espace sont davantage des spécialistes de domaines métiers et nous les nommerons par la suite "concepteurs de systèmes". Afin de concevoir rapidement leurs applications (ou systèmes métiers), ces acteurs savent comprendre et réutiliser les solutions génériques spécifiées par les templates mis à disposition par la première catégorie d'acteurs. Dans cet espace, les activités sont la création de nouveaux modèles, notamment par application de templates et par composition de modèles ainsi que l'analyse et l'évolution de modèles existants.

Dans les sections qui suivent, nous détaillons chaque partie de l'ingénierie. La sous-section 3.2.1 précise l'utilisation du dépôt de modèles au sein de l'ingénierie. Quant aux sous-sections 3.2.2 et 3.2.3, celles-ci présentent les activités de modélisation spécifiques à chaque espace autour de ce dépôt.

3.2.1 Le dépôt de modèles

Dans un environnement de modélisation, un dépôt de modèles est un support qui permet le stockage des modèles élaborés au cours de projets de modélisation et qui facilite également leur récupération pour de nouveaux projets. Par exemple, *EMF Store* (Koegel & Helming [61]) et *CDO* (Stepper [97]) sont des dépôts qui existent dans l'environnement *EMF* (*Eclipse Modeling Framework* - Steinberg et al. [96]). Parmi les fonctionnalités supportées par les dépôts de modèles, on trouve généralement :

- la persistance des modèles dans des bases de données selon un format optimisé (table-colonnes, graphes, ...),
- le versionnement des modèles afin d'enregistrer les essais successifs d'une modélisation ou des alternatives de modélisation,
- l'association de métadonnées aux modèles pour les identifier plus facilement et les classer,
- l'organisation des modèles en collections afin d'aider à les retrouver plus rapidement selon des critères précis,
- l'indexation et la recherche de modèles pour les retrouver selon des caractéristiques particulières exprimées à l'aide de requêtes,
- la navigation dans les collections de modèles.

Dans notre ingénierie, le dépôt de modèles est un élément central pour la conception dans chacun des deux espaces mais aussi pour l'échange entre ces derniers.
Au sein de chaque espace de conception, ce dépôt sert aux acteurs pour mémoriser et réutiliser leurs efforts de modélisation. Ainsi, dans l'espace dédié aux templates, les concepteurs peuvent sauvegarder les templates qu'ils ont conçus et peuvent les réutiliser pour en construire de nouveaux par composition, par extension ou par copie. Dans l'espace dédié aux systèmes métiers, le dépôt peut être utilisé par les acteurs associés pour mettre de côté des résultats de projets, des modèles intermédiaires des systèmes ou encore des fragments de modèles identifiés comme utiles

dans de futurs projets de modélisation.
Entre les espaces, le dépôt permet aux acteurs d'échanger les artéfacts conçus de part et d'autre. Pour le concepteur de templates, le dépôt permet de mettre à disposition des templates afin que les acteurs métiers puissent les appliquer dans la conception de leurs systèmes. Par exemple, les concepteurs de templates décident de capturer plusieurs variantes du patron *Observer* sous la forme de templates et rendent ces variantes accessibles aux concepteurs de l'autre espace pour qu'ils puissent intégrer facilement différentes capacités d'observation à leurs systèmes métiers. De leur côté, les concepteurs de systèmes peuvent déposer dans le dépôt des exemples de modèles à partir desquels les concepteurs du premier espace vont induire des solutions plus génériques sous la forme de templates.
Dans notre ingénierie, le dépôt de modèles est donc destiné à accueillir aussi bien des modèles que des templates. Ces deux types d'artéfacts sont organisés dans des collections comme mentionné précédemment. Les templates peuvent être regroupés dans des librairies selon les types de problème qu'ils résolvent à la manière par exemple des familles de patrons de conception (Buschmann et al. [22]), tandis que les modèles sont généralement regroupés par projets de modélisation. Pour structurer le dépôt, il est également possible d'exploiter les relations entre les artéfacts. La relation d'inclusion entre modèles introduite en section 3.1 permet de structurer les versions incrémentales des modèles d'un même système. Dans le standard, les relations telles que le *bind*, l'*import* et le *merge* contribuent à la structuration des dépôts.

3.2.2 Espace de conception des templates

Comme nous l'avons déjà indiqué, cet espace vise l'élaboration de nouveaux templates et l'adaptation de templates existants par généralisation des situations applicatives. Nous avons identifié plusieurs activités de conception pour supporter ces objectifs, présentées à gauche en figure 3.13 :

(A) la *paramétrisation*,
(B) la *composition de templates*,
(C) la *décomposition de templates*,
(D) l'*induction*,
(E) la *recherche de templates*

Paramétrisation Cette activité consiste à adapter des templates en ajoutant ou retirant des paramètres. Les concepteurs de templates sont amenés à réaliser ce type d'adaptations pour plusieurs raisons. Il s'agit d'ajuster le paramétrage pour élargir les possibilités d'application d'un template. Une autre raison est d'augmenter les capacités d'enrichissement d'un template. Enfin, le besoin d'avoir plusieurs variantes d'un même template via leur paramétrage motive également cette activité.
La figure 3.14 illustre cette activité avec la paramétrisation du modèle ***(a)*** représentant le patron de conception *Observer*. Avec ce template, la fonctionnalité d'observation peut être installée dans n'importe quel contexte où un observateur et un sujet à observer sont présents et reliés entre eux.
Cependant, au cours d'un projet de modélisation, les spécifications évoluent, menant à de nouvelles façons d'appliquer certaines des fonctionnalités. L'activité de paramétrisation permet de répondre à ce besoin. Ceci est illustré en figure 3.14 avec l'exemple du template *ObserverPattern* ***(variante b)***.
Dans le cas présent, seul le paramétrage du template a changé. Contrairement à la variante précédente ***(a)***, l'association *obs* n'a plus le statut de paramètre dans ce template et fait donc

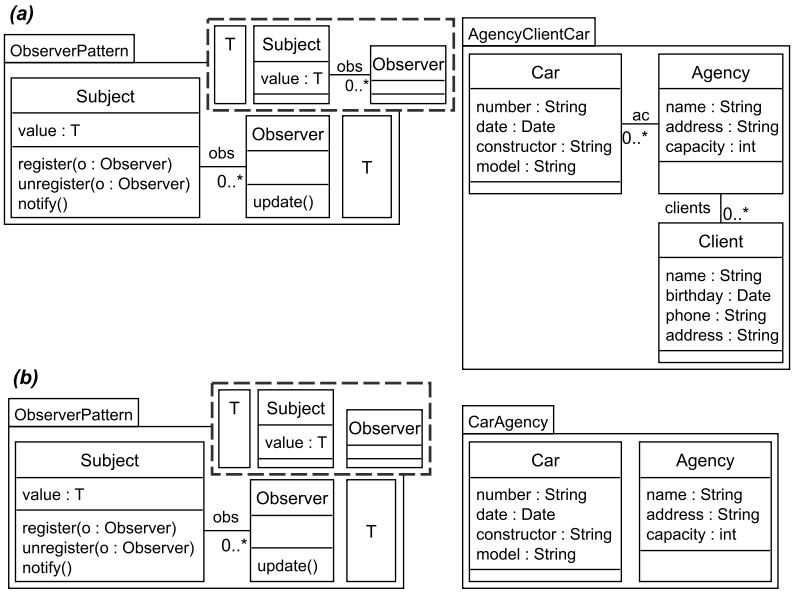


FIGURE 3.14 – Adaptation des paramètres du template

partie des enrichissements qu'il apporte. Ce template est applicable sur l'ensemble des contextes dans lesquels la relation d'observation est absente, tel que dans le modèle *CarAgency* pour l'installer. À travers cet exemple, il est intéressant d'observer que la diminution du nombre de contextes sur lesquels un template est applicable va de pair avec la définition supplémentaire d'éléments du template comme éléments paramètres. A l'inverse, les fonctionnalités sont applicables à des ensembles plus larges de modèles lorsque le statut de paramètre est retiré à des éléments du template.
Dans le cadre de cette activité de paramétrisation, des facilités doivent être offertes aux concepteurs de template afin de contrôler et vérifier le paramétrage. Dans le contexte des aspectual templates, la vérification doit être faite que l'ensemble des paramètres conservent une structure de modèle. Des facilités de recommandation automatique de paramétrage sont attendues. Pour l'exemple du template *Observer*, la sélection de l'association en tant que paramètre requiert également que les classes aux extrémités soient définies en tant que paramètres. Cette recommandation est proposée automatiquement au concepteur lorsqu'il expose l'association comme paramètre.
Dans le chapitre 4, consacré à la mise en œuvre de cette ingénierie, nous présenterons différents opérateurs destinés à supporter cette activité de paramétrisation.

Composition de templates La composition de templates est une activité qui permet de construire un nouveau template à partir d'autres templates. Cela est intéressant quand on souhaite élaborer un template en réutilisant les fonctionnalités génériques fournies par d'autres templates. Cette composition s'effectue en combinant le contenu des templates mais cela requiert également d'examiner la combinaison des paramètres. C'est le cas en particulier quand les éléments à combiner de chaque template ne sont pas tous les deux paramètres. Faut-il conserver ou non ce statut de paramètre pour l'élément correspondant dans le template résultat ?
Dans la section 3.1.2, traitant des aspectual templates, nous avons présenté un mode de composition de templates basé sur l'opération d'application. D'autres modes de composition de templates sont envisageables, notamment par extension des modes de composition qui existent pour les mo-

dèles de manière générale, tels que l'*import* et le *merge* d'*UML* (cf. section 2.1). La figure 3.15 montre un exemple où la fusion de templates se révèle intéressante.

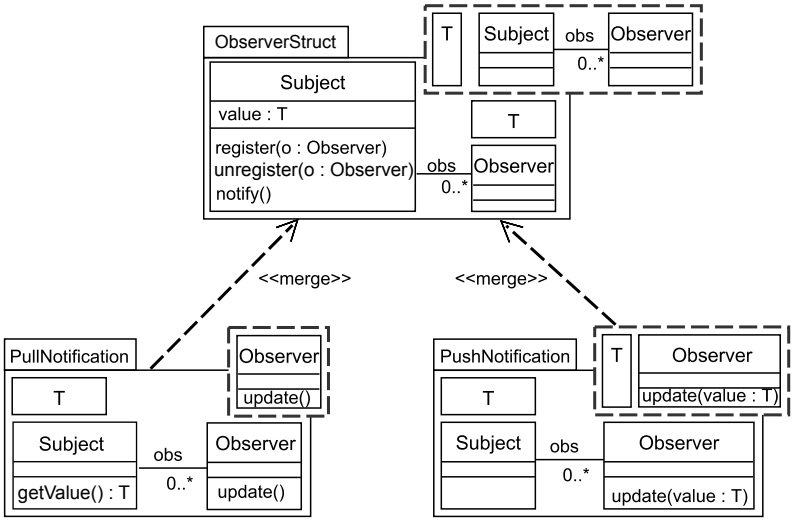


FIGURE 3.15 – Composition par *merge*

Dans cet exemple, deux templates, correspondant aux variantes *push* et *pull* du patron *Observer* (Shvets et al. [95]), sont obtenus en fusionnant un template *ObserverStruct*, exprimant la structure du patron en terme de *Subject* et d'*Observer*, avec un template exprimant la spécificité de chaque notification, respectivement *PullNotification* et *PushNotification*. Comme, on peut le voir sur cette figure, ces templates enrichissent la classe *Observer* du template *ObserverStruct* avec une méthode *update* de notification correspondante qui est également paramètre.
De façon plus générale, quels que soient les modes offerts dans cette activité de composition de templates, il est important que le paramétrage reste cohérent dans les templates produits. C'est ce que nous avons expliqué dans le cadre des aspectual templates où le template résultant d'une application de deux aspectual templates doit toujours rester un aspectual template. Cette contrainte est essentielle pour garantir la composabilité des templates dans les séquences d'application et les assemblages. Nous aurons l'occasion de revenir sur ce point dans la suite de ce chapitre, notamment lorsque nous discuterons d'application de templates à une hiérarchie de modèles (cf section 3.5).

Recherche de templates Lorsque le dépôt contient un grand nombre de templates, des facilités de recherche spécifiques sont nécessaires. Par exemple, en phase de conception d'un template, un concepteur doit être capable de rechercher quels sont les templates incluant déjà certaines fonctionnalités, que ce soit pour les réutiliser ou pour s'en inspirer. De même, lorsqu'un concepteur compose des templates, il doit être en mesure de rechercher quels sont les templates composables avec d'autres. Ces deux situations nécessitent une recherche dans le dépôt de modèles mais, selon la situation, on s'intéressera au contenu du template ou à ses paramètres. Ainsi, dans le premier cas, la recherche se focalisera sur le contenu des templates existants dans le dépôt, tandis que dans le second cas, la recherche sera guidée par les paramètres de ces templates. Pour réaliser ces deux types de recherche, la recherche de sous-modèles proposée dans Carré et al. [25] et Vanwormhoudt et al. [102] et rapportée au contenu ou au modèle paramètre du template, est une solution. Il est à noter que ces capacités de recherche peuvent également être utiles aux concepteurs de systèmes lorsqu'ils ont besoin de trouver un template disposant

d'une fonctionnalité particulière pour l'appliquer à leur système.

Décomposition de templates La décomposition est une activité qui s'avère utile lorsqu'on souhaite extraire une partie d'un template pour en faire un template. Par exemple, dans le template *ObserverPattern* donné en figure 3.14, la notification de changement de valeur s'effectue dans un mode *pull*, ce qui n'est pas forcément adapté à tous les types de situations. En effet, pour les besoins d'un projet, on peut vouloir adopter une notification en mode *push* et par conséquent avoir besoin d'une version différente du template. Pour obtenir cette variante, on peut procéder à partir de rien mais il est également envisageable d'utiliser la structure du template *ObserverPattern* de départ, notamment les classes *Subject*, *Observer* et l'association qui les relie. Cela est réalisable par extraction des éléments concernés à partir du template initial. La difficulté de l'extraction dans le cas des templates porte sur la gestion du statut des paramètres. Ces éléments doivent-ils rester systématiquement paramètres ou cela dépend-il des autres éléments auxquels ils sont associés dans le template initial ? Prendre en charge cette activité de décomposition de templates dans l'ingénierie nécessite approfondissement.

Induction de templates Comme nous l'avons vu à plusieurs reprises, l'élaboration d'un template peut avoir comme source la connaissance de solutions génériques comme les patrons de conception. Cependant, dans des domaines métiers spécifiques, la conception d'un template peut aussi avoir comme source un ensemble de modèles issus de différents projets de modélisation et disposant de structures ou de fonctionnalités proches ou comparables. Dans ce cas, il est intéressant de factoriser ces modèles en un template dont les paramètres prendraient en compte la variété des situations de modélisation rencontrées dans les différents projets. Une façon de mettre en œuvre l'induction d'un template est d'identifier les éléments communs et les éléments variables des modèles, puis de construire le template en transformant les parties variables en paramètres. Cette synthèse de templates par induction soulève de nombreuses questions à étudier.

3.2.3 Espace de conception des systèmes

Rappelons que dans cet espace, les acteurs conçoivent des applications et des systèmes métiers en s'appuyant sur les templates et les modèles disponibles dans le dépôt. Comme dans l'espace précédent, plusieurs activités en lien avec les templates [2] ont été identifiées pour notre ingénierie (partie droite de la figure 3.13) :

(F) la *construction de modèles*,
(G) la *détection et l'extraction de templates*,
(H) la *recherche de modèles guidée par les templates*

On peut ajouter à ces activités celle de recherche de templates, très similaire à celle décrite précédemment pour l'autre espace et qui ne sera donc pas présentée à nouveau. Dans le cas de cet espace, l'usage de la recherche de templates diffère puisqu'elle aide à trouver des templates à réutiliser pour apporter des enrichissements fonctionnels et non de concevoir de nouveaux templates.

Construction de modèles Cette activité correspond à l'usage principal des templates pour la construction d'applications métiers. En effet, les templates sont principalement destinés à la construction de nouveaux modèles par substitution de leurs paramètres dans des contextes applicatifs. Comme indiqué dans l'état de l'art, on recense principalement deux modes d'application

2. On laissera de coté ici les activités qui sont uniquement basées sur les modèles.

des templates pour la construction de modèles : un mode aspectuel et un mode génératif (instanciation). Dans le premier mode, le template est appliqué à un modèle de base pour l'enrichir. C'est le mode que nous avons détaillé et illustré en détail pour les aspectual templates : le modèle résultat contient les éléments du modèle de départ, enrichi des éléments du template après substitutions. Dans le second mode, le template est appliqué à un modèle pour construire un nouveau modèle dont les éléments correspondent au contenu du template avec les paramètres remplacés par des éléments du contexte. Ces deux modes sont illustrés en figure 3.16 : (a) pour l'enrichissement et (b) pour l'instanciation.

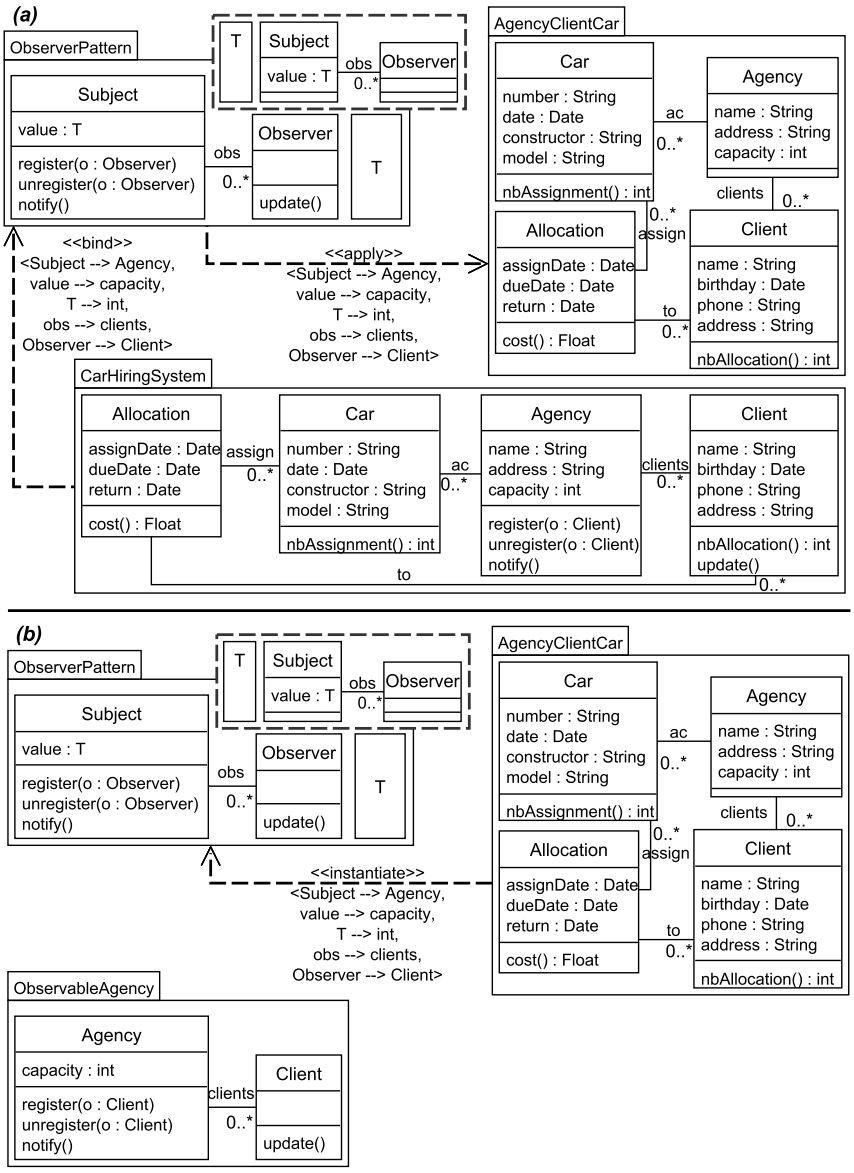


FIGURE 3.16 – Construction de modèles par applications
(a) aspectuelle et *(b)* générative de templates

Comme on peut le noter, les résultats sont différents. Dans le premier modèle résultat, on retrouve la structure du contexte, tandis que pour le second, c'est la structure du template qui est déterminante. Nous reviendrons plus précisément sur le mode génératif et son lien avec le mode aspectuel dans la section qui traite de l'instanciation (cf. section 3.3).

Dans les deux modes d'application, la construction de modèles peut être facilitée par des fonctionnalités liées aux substitutions. On peut citer la vérification que l'application du template est correcte (conformité de structure entre le modèle paramètre et le modèle actuel) et la complétion automatique d'une application (déduction des éléments du contexte à substituer à partir de ceux déjà sélectionnés). Par exemple, en figure 3.16 (***(a)***) lorsque le modeleur a défini la substitution de *value* avec *capacity*, la fonctionnalité de complétion l'assiste en lui proposant toutes les substitutions possibles, restreintes à celles incluant la substitution de *value* avec *capacity*, celle de *Subject* avec *Agency* et *T* avec *int*, puisque *capacity* est reliée à *int* et *Agency* dans *AgencyClientCar*). De telles fonctionnalités ont été mises en œuvre dans notre environnement de modélisation et seront présentées dans le chapitre 4.

Extraction de modèles guidée par les templates Pour les modèles de systèmes issus de l'ingénierie, il est possible de tracer leur construction à base de templates. Pour cela, on dispose de la relation *bind* d'*UML* qui est le moyen de capturer comment un template a été appliqué pour construire un modèle. Certains modèles peuvent avoir été récupérés dans des dépôts en ligne, d'autres peuvent avoir été transmis par des acteurs du projet ne disposant pas de cette ingénierie. Ici, l'objectif est a posteriori de retravailler ces modèles au travers de l'ingénierie à base de templates (ré-ingénierie). Un besoin est de déterminer si le modèle inclut la structure et les fonctionnalités d'un template existant. Cela permet de comprendre un modèle complexe qui intègre des parties de modèles pouvant être appariées à des templates connus. Cela sert à vérifier la conformité d'un modèle vis à vis de patrons de conception qui doivent être respectés dans le projet de modélisation. Enfin, cela est utile pour des besoins d'évolution, notamment lorsqu'il s'agit de faire évoluer une partie d'un modèle en remplaçant la fonctionnalité du template détectée par celle d'un autre template plus adaptée.
La figure 3.17 illustre le besoin de détection de templates dans un modèle.
Dans cette figure, on considère le cas du concepteur d'un système de location de véhicules qui a reçu un modèle (*AgencyCar* à droite) fourni par un collaborateur correspondant à une partie du système. Ce concepteur souhaite vérifier si ce collaborateur a bien respecté le cahier des charges qui stipule que le mécanisme d'observation des véhicules doit se faire en mode *push* avec une fonctionnalité de comptage des notifications. Pour vérifier le modèle en question, le concepteur le confronte à sa bibliothèque de templates conçue pour des capacités d'observations. Ici, trois templates de cette bibliothèque sont présentés. L'identification peut être faite avec le template d'observation en mode *push* mais pas avec celui intégrant en plus la fonctionnalité de comptage : la classe *Car*, correspondant à *Observer*, ne possède pas la méthode *totalNotif()* et la propriété *notifCount*. Le cahier des charges n'étant pas respecté, le concepteur doit donc demander une révision plus complète du modèle à son collaborateur. Identifier un template dans un modèle est déjà en soi une capacité intéressante pour sa compréhension ou sa vérification. S'il faut faire évoluer ce modèle vis à vis des templates, comme suggéré précédemment (par exemple, remplacer la fonctionnalité apportée par le template par une fonctionnalité plus riche), il est également intéressant de disposer de capacités permettant d'extraire automatiquement des templates contenus dans les modèles existants. Une telle capacité d'extraction consiste à retirer du modèle les ingrédients liés au template (détissage - Klein et al. [60]), en tenant compte des substitutions possibles. Pour illustrer cette extraction, revenons sur l'exemple précédent du concepteur ayant reçu un modèle non conforme à ses attentes. À l'aide de l'extraction de templates, le mode d'observation *push* est remplacé par le mode d'observation *push* avec comptage (fourni par *PushCountObserver*) afin de répondre au cahier des charges.

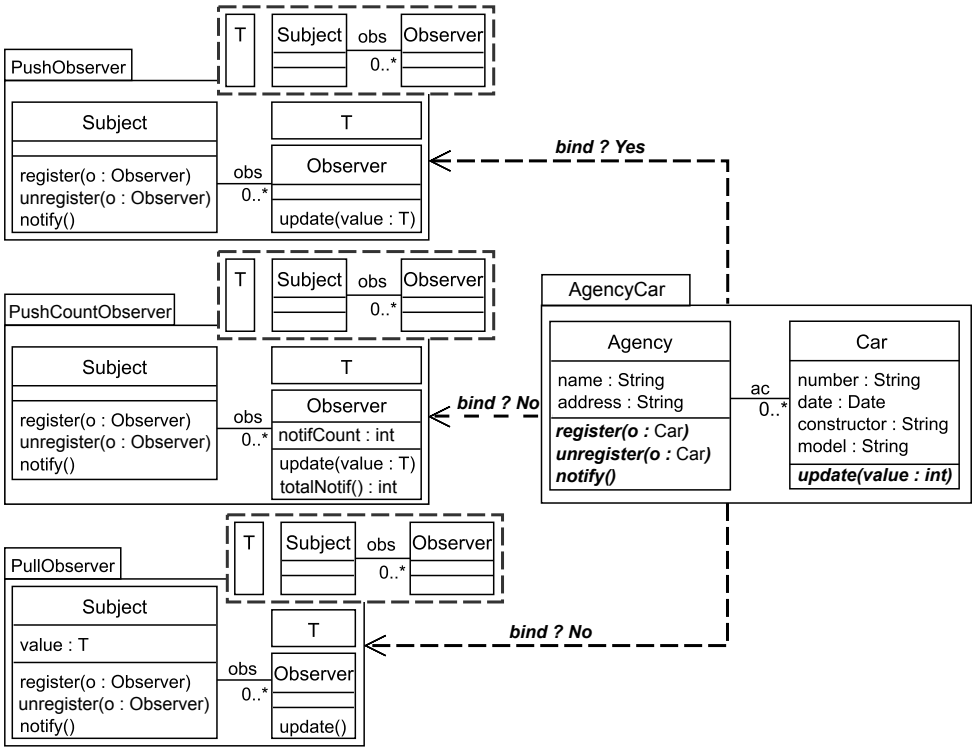


FIGURE 3.17 – Identification et extraction d'un template

Recherche de modèles guidée par les templates Au cours de leurs projets, les concepteurs de systèmes explorent souvent de nombreuses alternatives de modélisation. Ces alternatives peuvent correspondre à des ébauches, à des essais, à des modèles intermédiaires ou encore des versions de modèles, voire des branches complètes répondant à des préoccupations différentes rencontrées au cours du projet ou de projets précédents. Pour modéliser plus vite et de façon plus sûre, les concepteurs peuvent réutiliser ces versions et leur appliquer des templates. Quand un template a été appliqué à un modèle donné, il peut être intéressant de déterminer si ce template reste applicable également aux autres versions du modèle en question. De telles versions peuvent être par exemple une version plus simple ou plus complexe de ce modèle ou encore toute une branche alternative du projet. Déterminer cette applicabilité requiert de pouvoir rechercher des modèles en lien avec celui sur lequel un template est appliqué. Ce type de recherche est illustré à la figure 3.18.

Dans cette figure, le concepteur a appliqué le template *PushObserver* à la version sur laquelle il travaille actuellement et il se demande à quelle autre branche de modélisation (A, B ou C) ce template est également applicable. Il a donc besoin de rechercher dans le dépôt les modèles qui contiennent une structure analogue à son modèle mais également suffisante pour permettre l'application du template. Le résultat de cette recherche est l'ensemble des modèles de la branche A et celui de la branche B. Les modèles de la branche C ne font pas partie du résultat puisqu'il leur manque des ingrédients (la classe *Client* et l'association *clients*). Nous reviendrons de façon plus précise sur les règles qui peuvent guider cette recherche dans la section 3.5, en étudiant l'applicabilité d'un template sur une hiérarchie de sous-modèles.

Les sections suivantes présentent nos contributions à l'ingénierie de modèles à base de templates. Ces contributions se situent principalement dans l'espace dédié à la conception de systèmes. La

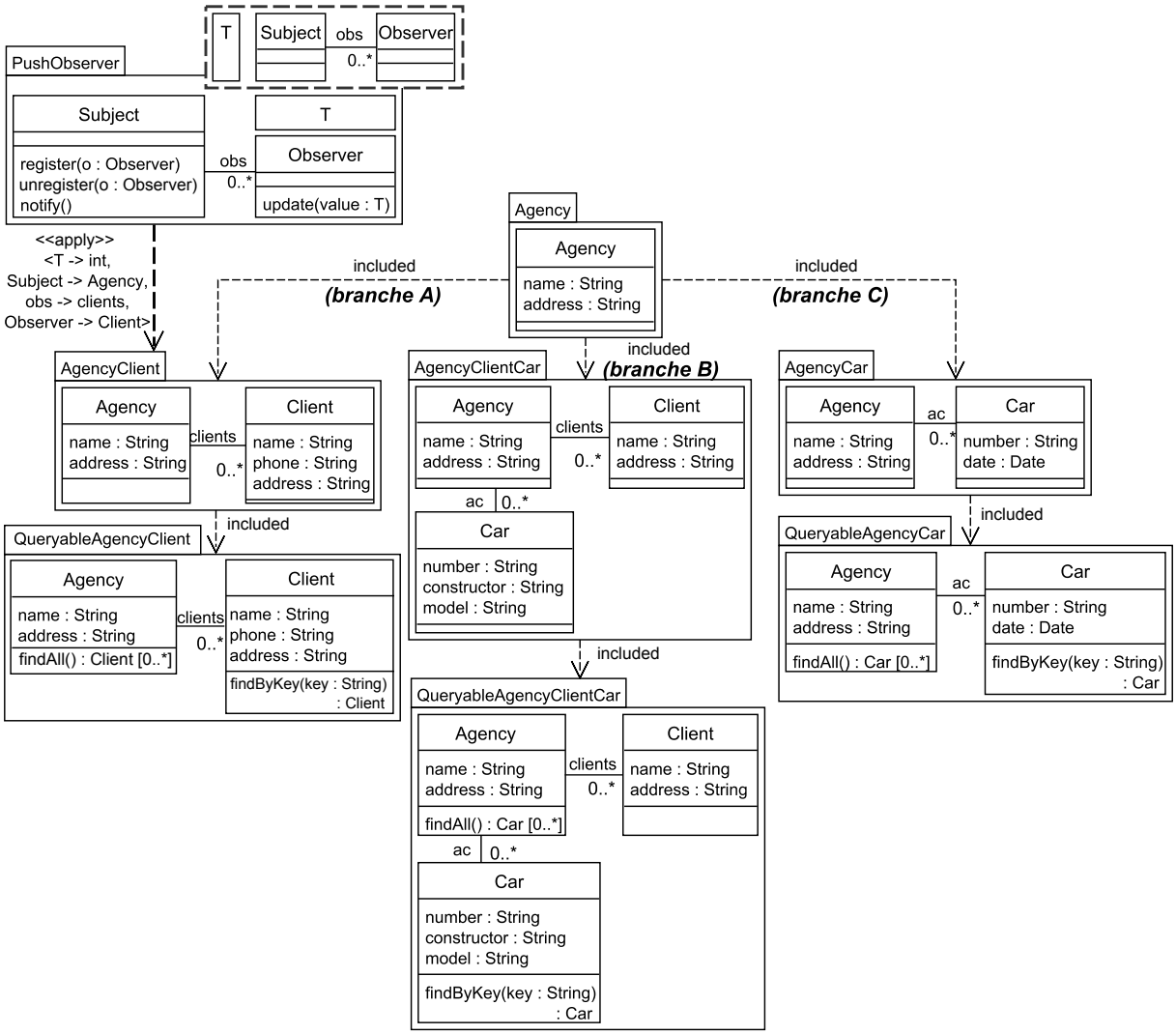


FIGURE 3.18 – Recherche guidée par les templates

section 3.3 est consacrée à l'étude de l'instanciation pour les aspectual templates. Dans cette section, nous prenons comme point de départ la relation *bind* d'UML et son interprétation aspectuelle. Nous analysons cette relation pour isoler l'instanciation de template qu'elle sous-tend. L'instanciation d'aspectual templates est ensuite étudiée comme moyen de construction de modèles à part entière. La section 3.4 s'intéresse à la détection d'instances de template dans des modèles créés manuellement ou par d'autres moyens que les applications de templates ainsi qu'à l'extraction de telles instances. Enfin, la section 3.5 se concentre sur le problème d'application de templates à une hiérarchie de modèles. De telles hiérarchies se rencontrent fréquemment dans le cycle de vie des projets de modélisation de grande taille, notamment dans les pratiques liées à la gestion de versions et à la modélisation en équipe. L'étude de ce problème nous conduira à éliciter des règles pour l'applicabilité de templates à de telles hiérarchies et à étudier la préservation de relations d'inclusion entre les résultats d'application de templates.

3.3 Instanciation de templates

Dans cette section, nous portons notre attention sur l'instantiation de templates, c'est à dire la construction de nouveaux modèles à partir de la structure des templates. Pour ce faire et rester dans la lignée du standard, nous partons du binding et isolons l'instanciation. Ainsi isolée, l'instanciation peut être prise comme opération à part entière(3.3.1). Les conséquences de l'instanciation sur les paramètres du modèle, en particulier lorsqu'ils forment un modèle, doivent alors être examinées. Nous faisons cela en identifiant les constituants du modèle comme des sous-modèles (section 3.3.2). On montrera que l'instanciation peut être appliquée sur tous les modèles aspectuels ou non. De même que pour le binding partiel, une instanciation partielle a lieu lorsque tous les paramètres ne sont pas remplacés (section 3.3.3). Une étude de cette fonctionnalité est proposée et son intérêt pour la production de modèles avec des parties provenant de différents contextes est présenté. Plus généralement, l'isolation des instances de templates contribue à augmenter la réutilisabilité des templates UML et à enrichir les fonctionnalités MDE basées sur les templates [8].

3.3.1 Instanciation et binding

En se référant à la définition de la relation *bind d'UML* (cf. chapitre 2), le *bound model* résultant de l'application d'un template peut être vu comme la fusion d'un contexte applicatif avec une instance du template. Cette fusion est représentée par la relation *merge d'UML*.
La figure 3.19 illustre ce principe : elle explicite le contexte applicatif (en haut à droite) utilisé avec le template (en bas à gauche) afin d'obtenir l'instance (en haut à gauche :*ObserverPatternInstance*).

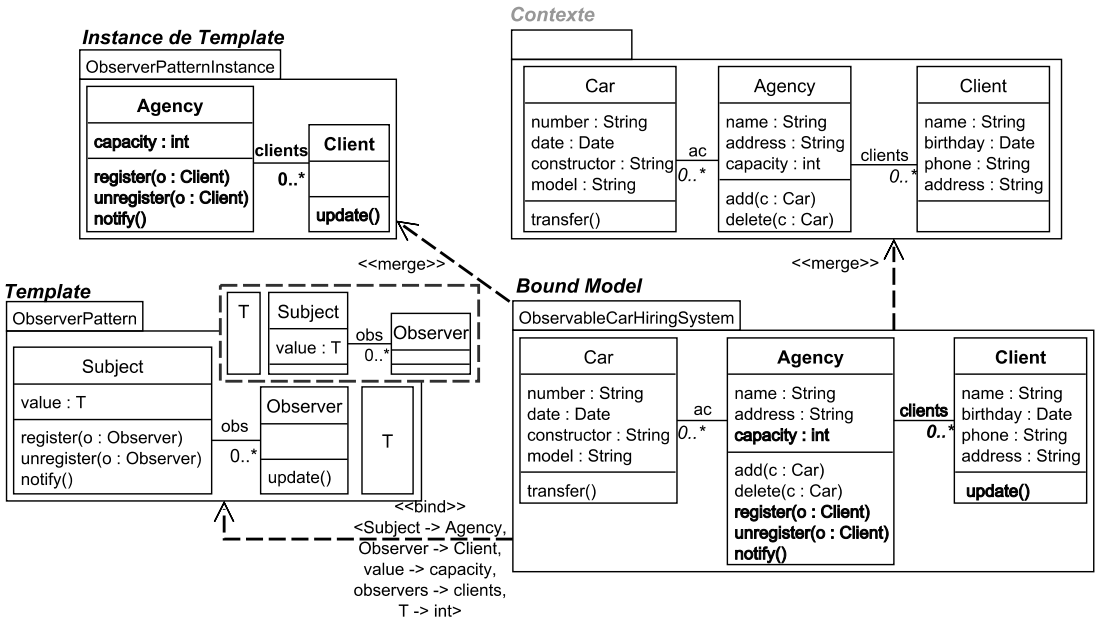


FIGURE 3.19 – Bound model = merge(instance, contexte)

C'est le contexte qui fournit les éléments actuels pour l'application du template. Le *bound model* *ObservableCarHiringSystem* inclut l'instance de template (éléments en gras) et le contexte. Les relations *merge* du standard partent du *bound model* vers l'instance du template et le contexte. Le *bound model* peut donc se formuler comme suit :

Bound model = merge(instance, contexte)

Cette formulation permet d'isoler l'instance et le contexte formant le *bound model*. Isoler l'instance permet de la promouvoir comme artéfact à part entière et de la réutiliser. Une telle isolation est utile lorsque les modeleurs sont davantage intéressés par la construction de nouveaux modèles à partir de templates, plutôt que par l'enrichissement de modèles existants.
Afin de représenter cette instanciation, nous introduisons la relation *instantiate*. Tout comme les relations *bind* et *apply*, cette relation nécessite un template, un modèle de base et un ensemble de substitutions.
L'instanciation consiste à *(1)* copier le contenu du template et à *(2)* remplacer les paramètres par des copies des éléments actuels du contexte.
La figure 3.20 illustre l'instanciation explicite de templates avec la relation *instantiate* et les mêmes template, contexte et instance que ceux de la figure 3.19.

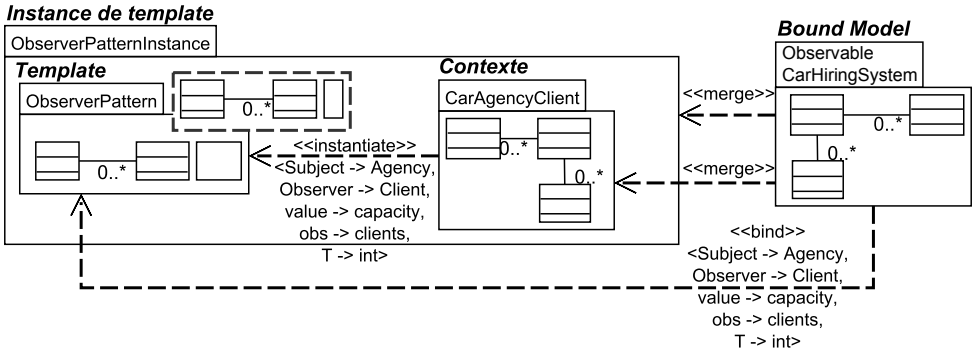


FIGURE 3.20 – Instanciation de template

3.3.2 Instanciation et formes de templates

Pour rappel, un template est constitué d'un sous-modèle paramètre et d'un sous-modèle spécifique (section 3.1). Ces sous-modèles peuvent être bien ou mal formés. Nous étudions tous les cas possibles comme résumé dans le tableau 3.1 pour en déduire les conséquences sur la forme du template et de ses instances.

TABLE 3.1 – Forme possible des constituants de template

		Modèle spécifique	
		bien formé	**mal formé**
Modèle Paramètre	**bien formé**	Cas 1	Cas 2
	mal formé	Cas 3	Cas 4

Considérons les **cas 1 et 2**, correspondant à un sous-modèle paramètre bien formé. Cette situation correspond à celle examinée en figure 3.21.
Insistons sur le fait que, dans le cas 2, la bonne formation du modèle résultat (cf. instance de template en figure 3.19) est plus particulièrement due à celle du modèle paramètre. Ainsi, de la même façon qu'avec l'application aspectuelle, l'instanciation peut être utilisée avec n'importe quel template ayant un sous-modèle paramètre bien formé. Ceci est un résultat intéressant en ce qui concerne la réutilisation de templates. En effet, cela signifie que tout template ayant cette

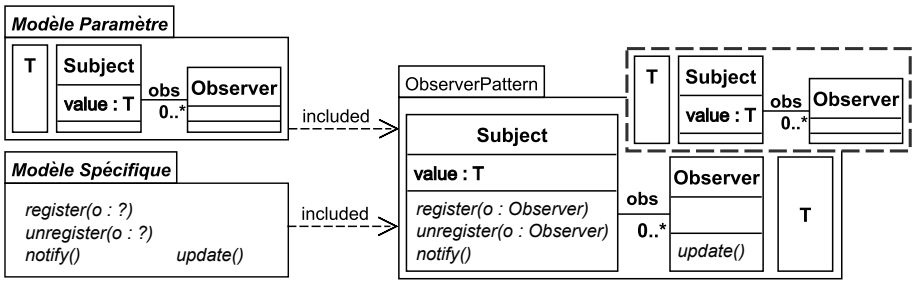


FIGURE 3.21 – Modèle paramètre bien formé et modèle spécifique mal formé

propriété peut être utilisé aussi bien de manière aspectuelle que générative, avec les relations que ces deux pratiques entretiennent (section 3.3.1 et sa figure 3.20 pour l'exemple).
Poursuivons avec les **cas 3 et 4** qui correspondent aux situations où le sous-modèle paramètre est mal formé. La figure 3.22 montre un exemple d'instanciation de templates pour le troisième cas.

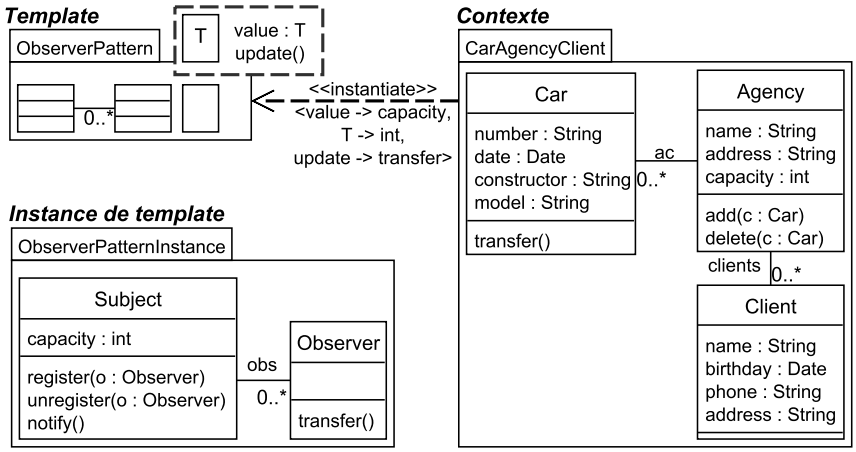


FIGURE 3.22 – Instanciation de template avec un modèle paramètre mal formé

Comme on peut le voir, le sous-modèle paramètre est mal formé, *a contrario* du sous-modèle spécifique. Dans le cas 3, symétriquement au cas 2, la bonne formation du modèle résultat est plus particulièrement due à celle du modèle spécifique.
En ce qui concerne les premier et quatrième cas, les deux sous-modèles sont complémentaires et apportent ensemble la bonne formation du modèle résultat. Un exemple du cas 4 consiste à modifier le statut de la classe *Subject*, dans le template de la figure 3.22, en la définissant comme paramètre, ce qui rend les deux sous-modèles mal formés.

Typiquement, les troisième et quatrième cas se produisent lors de la modélisation de génériques (*e.g.* en C++ ou Java). Il est donc important de les prendre en compte et d'assurer leur cohérence au travers de l'instanciation de templates. Nous verrons notamment dans la section suivante que ces deux cas apparaissent avec le concept d'*instanciation partielle*.

3.3.3 Instanciation partielle

Le standard *UML* permet l'application partielle de templates (tous les paramètres ne sont pas substitués, cf. section 3.1). En suivant un schéma analogue, se pose la question de l'*instanciation*

partielle et les problèmes de la propagation de paramètres et les adaptations de modèle qui en découlent.

L'instanciation partielle permet de produire de nouveaux templates. Comme dans le cas de l'application partielle, nous retenons le principe de propagation de paramètres du standard UML, à savoir la préservation du statut de paramètre pour les éléments non substitués. Il est important de noter que selon les paramètres non substitués, le paramétrage résultat peut constituer un modèle bien formé ou non.
La figure 3.23 donne un exemple d'instanciation partielle entre le template *ObserverPattern* et le modèle *CarAgencyClient*.

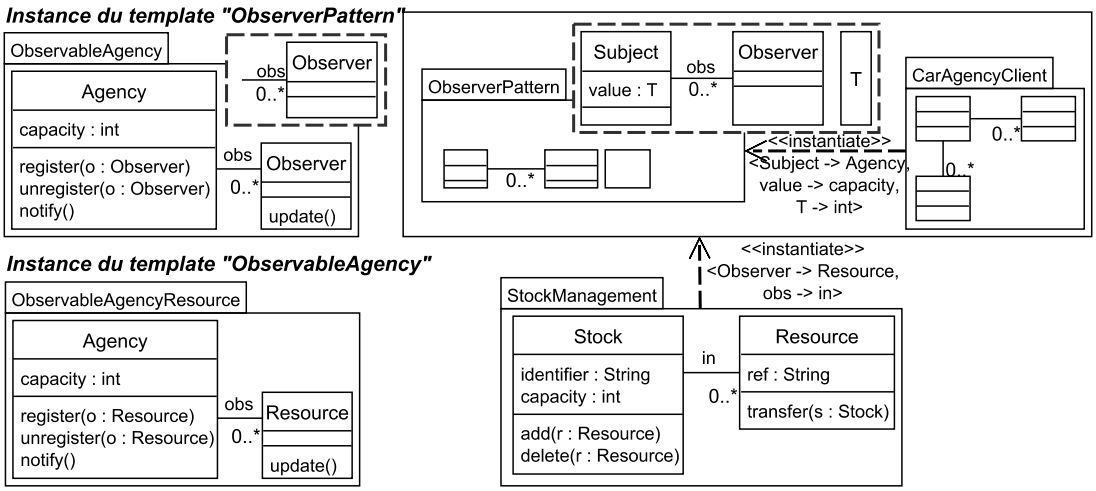


FIGURE 3.23 – Instanciation partielle

Dans cet exemple, les paramètres *Subject, value* et *T* sont présents dans l'ensemble de substitutions de la relation *instantiate*, au contraire des paramètres *obs* et *Observer*. Cette figure montre aussi le résultat de cette instanciation partielle, représenté par le nouveau template *ObservableAgency*. Pour ce template, le modèle paramètre inclut les paramètres non-substitués (*obs*, *Observer*) et est mal formé.

Le template résultat d'une instantiation partielle peut être réutilisé dans un nouveau contexte, ce qui mène à des séquences d'instantiations. En figure 3.23, le template résultat de l'instantiation précédente (*ObservableAgency*) est utilisé sur le contexte *StockManagement*. *Observer* et *obs* sont substitués afin de produire le modèle *ObservableAgencyResource* combinant des constituants des deux contextes *CarAgencyClient* et *StockManagement*. Un tel modèle peut être utile pour observer les changements d'états entre les deux parties. Par rapport à la propagation de paramètres non substitués, on remarque que celle-ci est effectuée en respectant la substitution définie, causant ainsi l'adaptation des méthodes paramètres. Ceci est montré avec la substitution de *Observer* par *Resource* au niveau notamment du paramétrage de la méthode *register* dans la classe *Agency*.
Soulignons que le même résultat peut être obtenu via une séquence d'instanciations alternatives, tel qu'exposé en figure 3.24.
Tout d'abord, l'instanciation partielle du *template ObserverPattern* avec le contexte *StockManagement* et ensuite l'instanciation du template résultat *intermédiaire*, *SubjectResource*, avec le contexte *CarAgencyClient*. Notons que, cette fois-ci, *SubjectResource* dispose d'un modèle para-

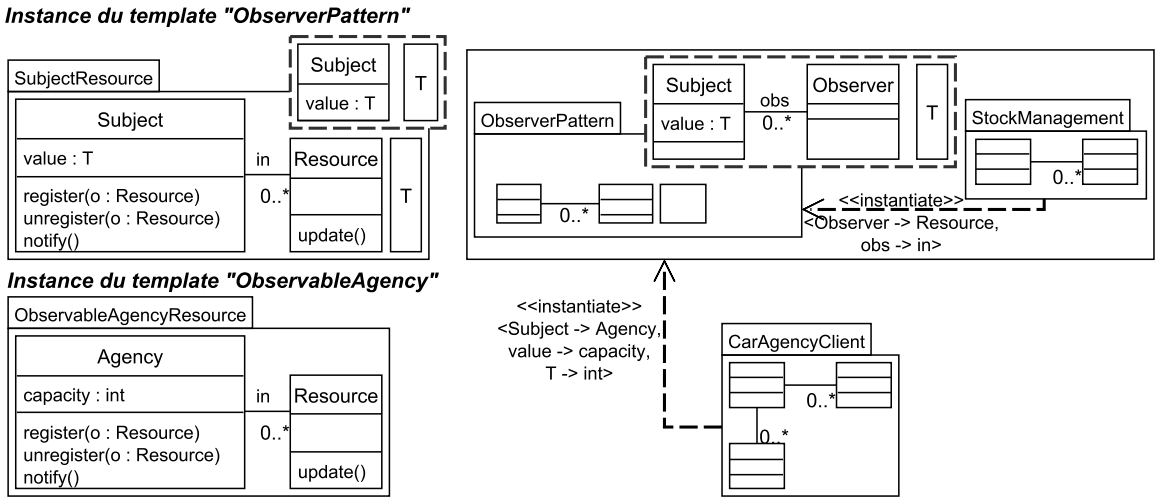


FIGURE 3.24 – Séquences d'instanciations alternatives - Instanciations partielles

mètre bien formé.
Le même modèle *ObservableAgencyResource* peut aussi être obtenu via une instanciation totale, telle qu'en figure 3.25.

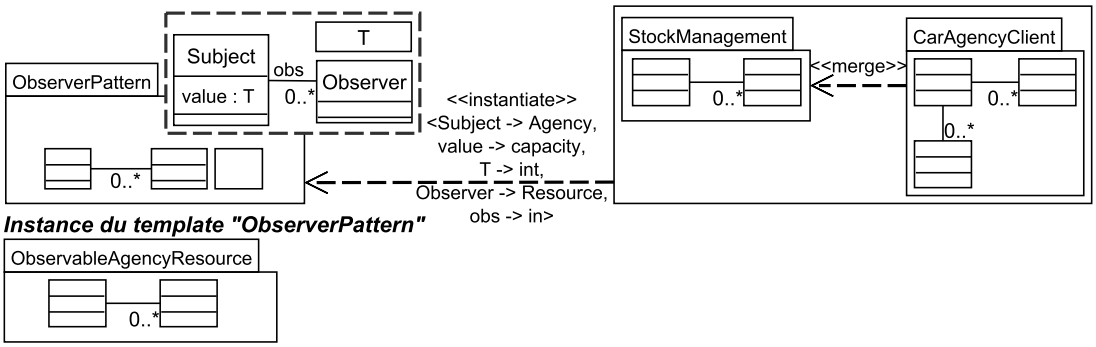


FIGURE 3.25 – Séquences d'instanciations alternatives - Instanciation totale

Ceci nécessiterait la fusion (*e.g.*, via l'opération *merge du standard UML*, présentée en section 3.3) des modèles *CarAgencyClient* et *StockManagement*, suivi d'une instanciation totale du template *ObserverPattern* avec le modèle résultant de la fusion précédente. Cette égalité met en exergue la compatibilité entre l'instanciation totale et partielle et, donc, leur cohérence.

3.3.4 Bilan

De la même manière qu'il existe un lien entre application et instanciation *totales*, nous synthétisons ici les liens qui existent entre application et instanciation *partielles*. La table 3.2 résume les différentes situations étudiées jusqu'ici. Elle présente l'applicabilité des templates sur les contextes et les résultats des opérations étudiées selon qu'il s'agit d'un template aspectuel ou non. Ceci devrait aider à mieux appréhender les templates UML, particulièrement dans les situations où les paramètres forment un modèle.

En complément de ce tableau, insistons sur le fait que la construction du modèle résultat par application ou instanciation partielle obéit aux mêmes règles de propagation de paramètres qu'avec

TABLE 3.2 – Applicabilité de templates et formes des modèles résultats lors de substitutions totales et partielles

	Template			
	Aspectual Template		**Non-Aspectual Template**	
	Applicable ?	*Catégorie du "Bound Model"*	*Applicable ?*	*Catégorie du "Bound Model"*
Application Aspectuelle	Oui	Modèle	**Non**	**Pas de modèle résultat**
instanciation	Oui	Modèle	Oui	Modèle
Application Aspectuelle partielle	Oui	Aspectual Template	**Non**	**Pas de modèle résultat**
Instanciation partielle	Oui	Template	Oui	Template

les applications ou instanciations totales. Ce qui permet de préserver l'équation *Bound model = merge(instance, contexte)* dans le cas des substitutions partielles.
Un exemple est présenté en figure 3.26, avec l'application partielle de *ObserverPattern* sur *StockManagement*, produisant le *bound model ObservableStockManagement*.

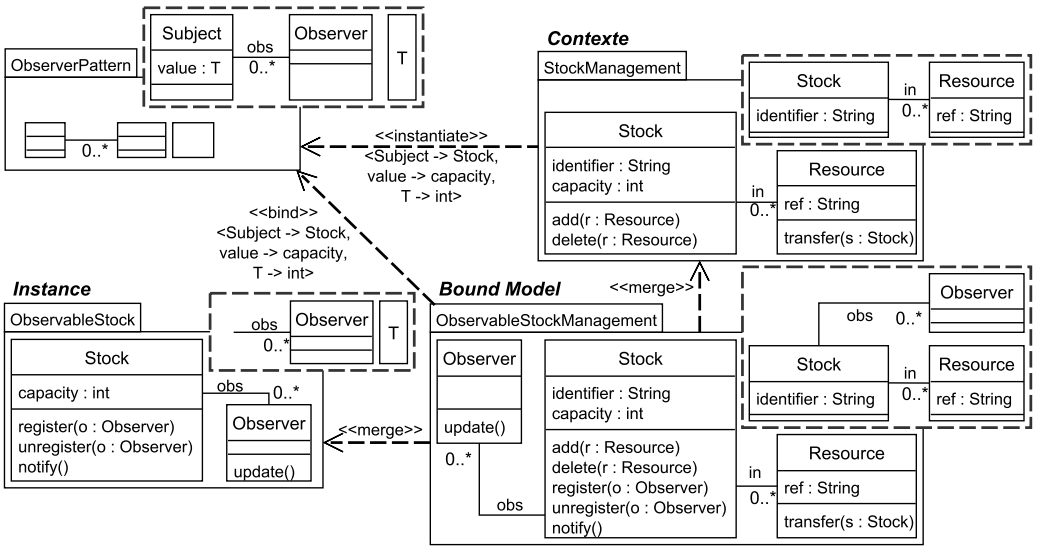


FIGURE 3.26 – Application partielle et instanciation partielle

ObserverPattern est instancié partiellement, générant le *template ObservableStock*. En observant le *bound model*, on note que celui-ci inclut le contexte et l'instance générée.

3.4 Détection et suppression de templates dans un modèle

Le travail effectué dans cette section est lié à l'activité d'extraction de modèles guidée par les templates, présentée en section 3.2. Comme énoncé précédemment, cette activité permet de retrouver des templates dans des modèles existants afin principalement de :

- comprendre ces modèles sous l'angle de templates de référence,
- vérifier que le modèle est conforme à des règles de conception de projets,

- défaire des templates en vue de simplification de modèles,
- remplacer un template par une autre version.

Pour cela, il est nécessaire de :

- *Détecter, dans un modèle, les fonctionnalités équivalentes à celles d'un aspectual template.*
- *Supprimer des fonctionnalités équivalentes à celles d'un aspectual template.* Ce qui revient à supprimer un sous-modèle équivalent au modèle spécifique du template.

Dans les sections suivantes, nous développons ces deux points.

3.4.1 Détection d'un aspectual template dans un modèle

La détection repose sur le test d'inclusion d'instances inférées à partir de l'univers des substitutions possibles :

1. Tous les ensembles de substitutions de paramètres sont calculés (Vanwormhoudt et al. [103]),
2. À partir de ceux-ci, toutes les instances du template avec le modèle sont générées en correspondance,
3. Enfin, un test d'inclusion est effectué entre chacune des instances obtenues et le modèle. Si ce dernier inclut l'une de ces instances, alors le modèle est un *bound model*.

Dans la figure 3.27, on souhaite déterminer si le modèle *CarHiringSystem* inclut les fonctionnalités du pattern Observer en mode push (*ObserverPatternPush*).

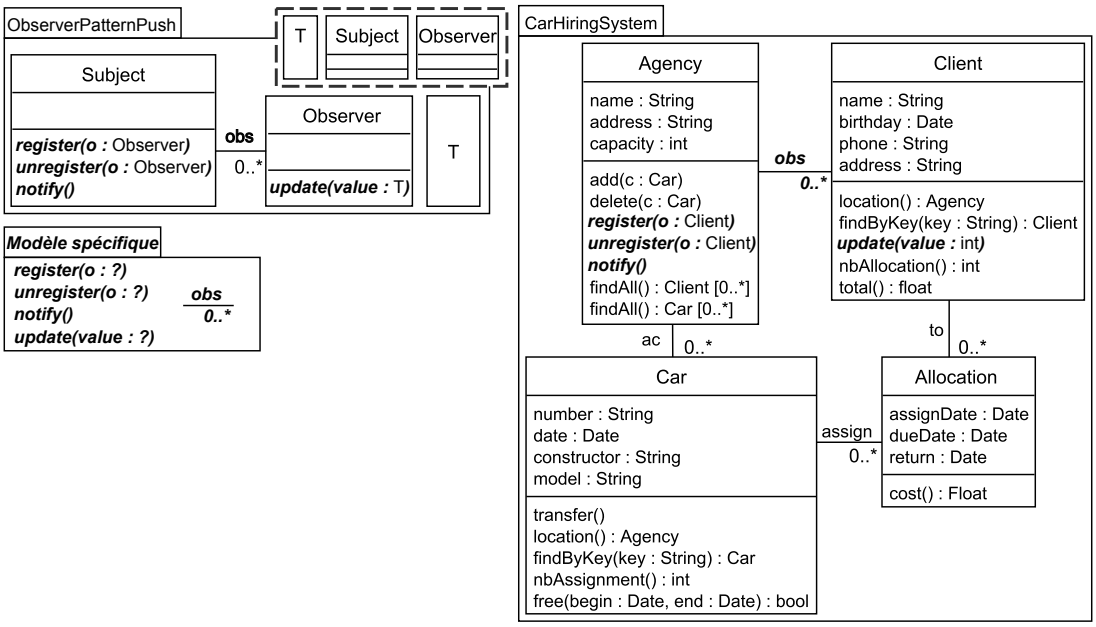


FIGURE 3.27 – Fonctionnalités (*modèle spécifique*) de *ObserverPatternPush* présentes dans *CarHiringSystem*

Selon les étapes présentées plus haut, la détection de *ObserverPatternPush* dans *CarHiringSystem* est la suivante :

1. Calcul des substitutions : La figure 3.28 présente deux des ensembles de substitutions calculés entre *ObserverPatternPush* et *CarHiringSystem*.

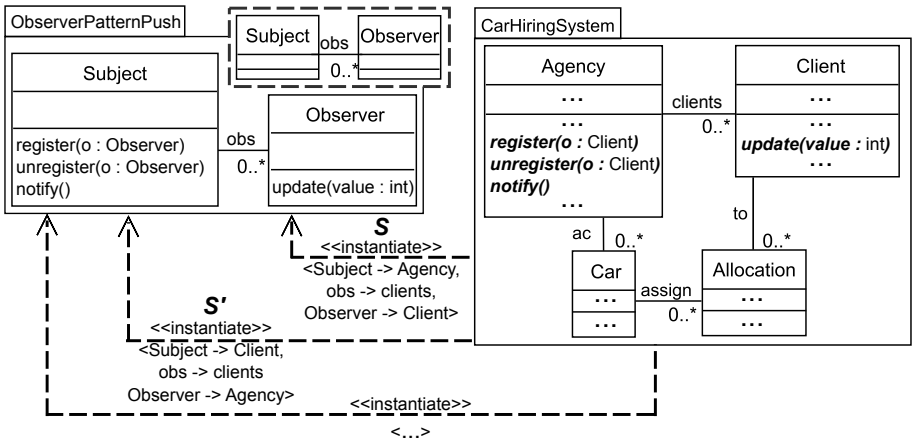


FIGURE 3.28 – Calcul des substitutions entre *ObserverPatternPush* et *CarHiringSystem*

2. Génération d'instances : Pour chacun des ensembles de substitutions précédents, les instances de *ObserverPatternPush* sont générées. La figure 3.29 présente les instances pour les substitutions précédentes.

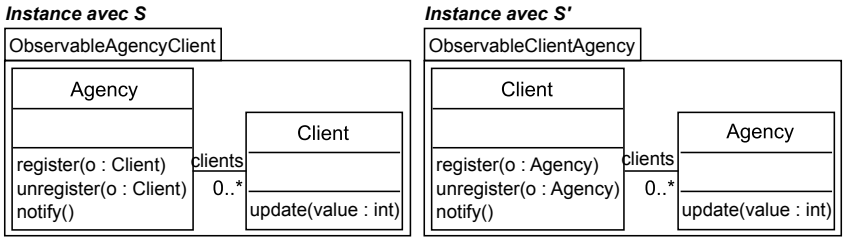


FIGURE 3.29 – Instances de *ObserverPatternPush*

3. Tests d'inclusion : Pour chacune des instances obtenues précédemment, un test d'inclusion est effectué entre elle et *CarHiringSystem*. Comme présenté en figure 3.30, l'instance calculée avec l'ensemble de substitutions S est incluse dans *CarHiringSystem*. L'*aspectual template ObserverPatternPush* est donc détecté au sein de *CarHiringSystem*.

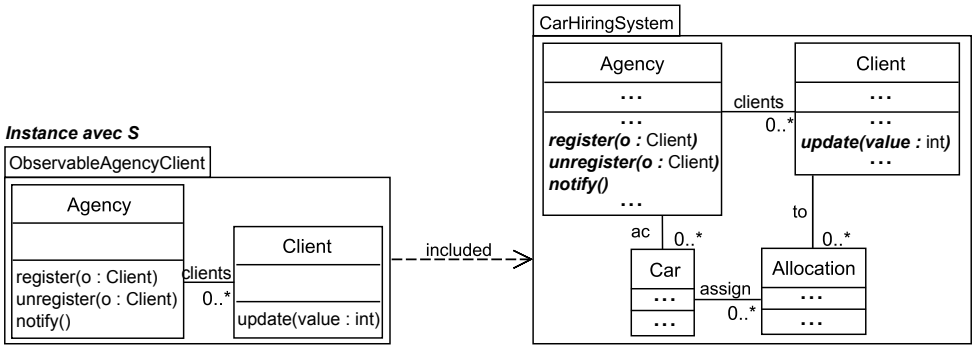


FIGURE 3.30 – *ObservableAgencyClient* inclus dans *CarHiringSystem*

L'opérateur *getBoundSubstitutions*, présenté dans la section 4.1, met en œuvre cette détection de template.

3.4.2 Suppression d'un aspectual template présent dans un modèle

La suppression d'un template dans un modèle consiste en trois étapes :

1. Le calcul de l'instance du template présente dans le modèle pour une substitution valide. Cette instance peut être obtenue via la détection de template décrite en section précédente,
2. Le *modèle spécifique ancré MSC* dans l'instance est déterminé. Ce modèle *MSC* est utilisé afin de déterminer à quels éléments du modèle sont rattachées les fonctionnalités du template,
3. Enfin, les fonctionnalités du template sont retirées du modèle en s'appuyant sur *MSC*.

Dans l'exemple de la figure 3.27, on souhaite remplacer la version *push* du pattern *Observer* par sa version *pull*. Pour cette mise à jour, il est d'abord nécessaire de retirer la version *push* du pattern présente dans *CarHiringSystem*. Ce retrait s'effectue selon les trois étapes citées plus haut :

1. *Calcul de l'instance présente dans le modèle :* Via la détection de templates décrite précédemment, l'instance *ObservableAgencyClient* (cf. figure 3.30), présente dans *CarHiringSystem*, est calculée.
2. *Calcul du modèle spécifique clos MSC dans l'instance :* Afin de retirer les fonctionnalités apportées par un template, c'est-à-dire son modèle spécifique (cf. sous-section 3.1.2.1), il est nécessaire de savoir comment ce modèle spécifique est lié au modèle. Au travers de ces éléments, de façon directe ou indirecte. Ces relations sont présentes dans l'instance du template obtenue lors de l'étape précédente.
   En figure 3.31, le modèle d'instance *ObservableAgencyClient* inclut le modèle spécifique de *ObserverPatternPush* relié aux éléments actuels de *CarHiringSystem*. Par exemple, *register* est relié à l'élément actuel *Agency*.

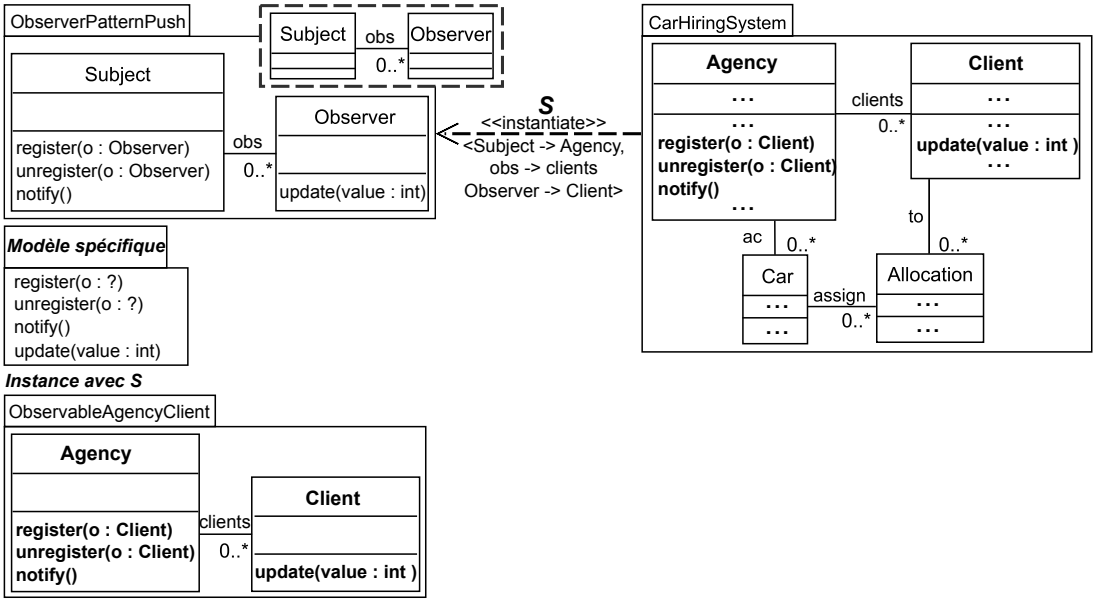


FIGURE 3.31 – Modèle spécifique de *ObserverPatternPush* lié aux éléments actuels de *CarHiringSystem* dans *ObservableAgencyClient*

Pour connaître ces contraintes structurelles entre éléments du modèle spécifique et éléments actuels, les opérations de fermeture d'ensemble et d'extraction de modèles sont

utilisées. Tout d'abord, la fermeture des éléments du modèle spécifique dans l'instance est appliquée[3]. On obtient ainsi l'ensemble des éléments du modèle spécifique munis des éléments actuels auxquels ils sont reliés.
En figure 3.32, la fermeture de l'ensemble des éléments du modèle spécifique de *ObserverPatternPush* ajoute à l'ensemble les éléments *Agency* et *Client*.

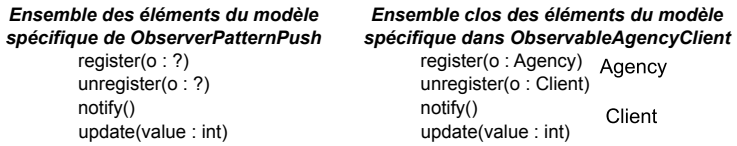


FIGURE 3.32 – Obtention de l'ensemble clos des éléments du modèle spécifique de *ObserverPatternPush* dans *ObservableAgencyClient*

En effet, comme montré en figure 3.31, *register, unregister, notify* et *update* sont liés à ces éléments dans *ObservableAgencyClient*.
Notez qu'il est bien nécessaire de faire la fermeture du spécifique dans l'instance et non pas dans le modèle tout entier. En effet, c'est l'instance qui détermine l'ancrage des éléments pour cette substitution (d'autres applications du même template pourraient être concernées).
Ensuite, afin d'obtenir les contraintes structurelles entre ces éléments du modèle spécifique et les éléments actuels, le sous-modèle de l'instance qui correspond à l'ensemble des éléments du modèle spécifique clos est extrait à l'aide de l'opération d'extraction (cf. section 3.1.1.2). Ce sous-modèle est le *modèle spécifique clos* (MSC) recherché.
La figure 3.33 présente le résultat de l'extraction de l'exemple. Ce modèle spécifique clos possède les contraintes structurelles liant les éléments du modèle spécifique avec *Agency* et *Client*.

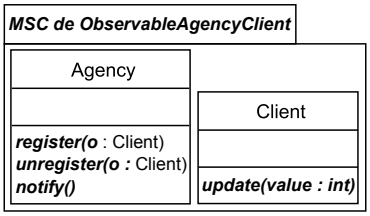


FIGURE 3.33 – Obtention du modèle spécifique clos (MSC) dans *ObservableAgencyClient*

3. *Retrait des éléments spécifiques ancrés du modèle :* En figure 3.34, l'ensemble des éléments spécifiques du MSC de *ObservableAgencyClient* est retiré de *CarHiringSystem*, c'est-à-dire tous les éléments du MSC, exceptés *Agency* et *Client*.
On observe que le MSC d'une instance permet donc de cibler la partie (sous-modèle) d'un modèle de laquelle doit être retirée le modèle spécifique. Ceci est particulièrement utile lorsque le modèle spécifique du même template est présent dans différents sous-modèles d'un modèle et que seul le modèle spécifique d'un des sous-modèles doit être retiré (e.g. lorsque le *pattern Observer* est présent à divers endroits d'un même modèle).

L'opérateur correspondant à cette opération d'extraction est l'opérateur *unbind*, présenté dans la section 4.1 du chapitre 4.

3. Ensemble de ces éléments augmenté des éléments dont ils dépendent dans l'instance et récursivement

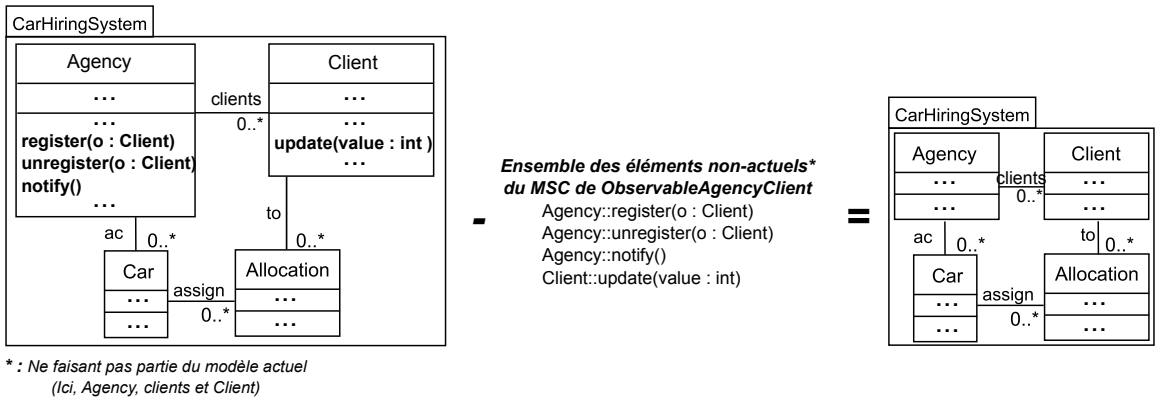


FIGURE 3.34 – Retrait des éléments du modèle spécifique de *ObserverPatternPush* dans le modèle *CarHiringSystem*

3.5 Applications de templates sur des hiérarchies de modèles

Les sections précédentes ont considéré des applications d'un aspectual template sur un modèle. Dans cette section, nous nous intéressons aux applications de tels templates sur des modèles qui sont en relation d'inclusion et forment des *hiérarchies de modèles*. De telles hiérarchies surviennent dans de nombreuses situations de modélisation. Nous nous intéressons aux situations suivantes :

- Conception incrémentale : un modèle de système est construit par ajouts successifs de fonctionnalités [4]. Dans l'exemple de la figure 3.35, M a été enrichi avec des fonctionnalités, produisant un surmodèle M' de M. Le template AT qui est appliqué sur M, peut-il aussi l'être sur M' ?

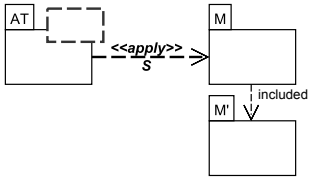


FIGURE 3.35 – Conception incrémentale

- Modélisation en équipe : les modeleurs partagent des modèles et conçoivent ensemble un modèle de système via l'application de templates. En figure 3.36, les modèles M' et M'', sous-modèle de M, sont respectivement détenus par deux modeleurs. Un aspectual template, appliqué sur M, peut-il aussi l'être sur M' et M'' ?

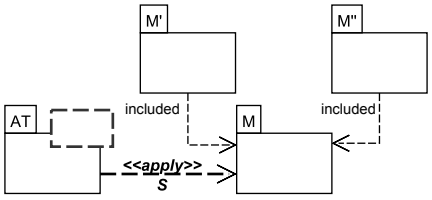


FIGURE 3.36 – Modélisation en équipe

4. La conception incrémentale est notamment décrite par Alam et al. [3]

- Versioning : les étapes produisent des hiérarchies d'inclusion de variantes. Dans l'exemple présenté en figure 3.37, les modèles M' et $M2$ sont des versions alternatives [5] de M. Le template AT, appliqué sur M selon S, peut-il aussi l'être sur ces variantes ? Lorsque AT est appliqué sur cette hiérarchie de variantes, les modèles résultants de cette application sont-ils liés les uns aux autres ?

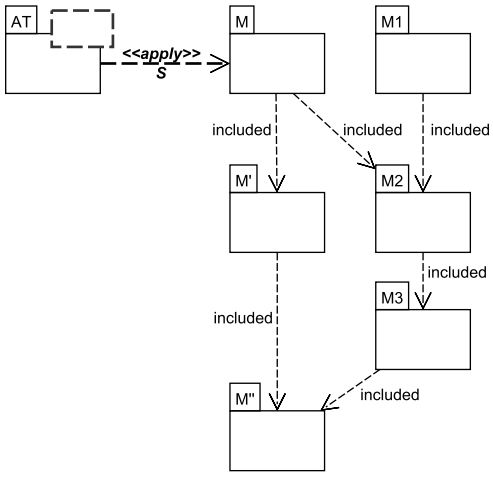


FIGURE 3.37 – Hiérarchie de variantes

De façon plus générale, cela soulève les problématiques suivantes :

- Sur quels modèles d'une hiérarchie, une application de template reste valide (sous-section 3.5.1) ?
- Comment se comportent les résultats vis-à-vis de l'inclusion (sous-section 3.5.2) ?

Les règles d'ingénierie répondant à ces problématiques sont présentées dans les sections suivantes.

3.5.1 Validité d'une application

Dans cette sous-section, nous étudions la validité de l'application d'un aspectual template sur une hiérarchie de modèles et nous en dérivons des règles d'ingénierie.

3.5.1.1 Validité de l'application d'un aspectual template sur un modèle

Un aspectual template peut être appliqué sur un modèle, pour une substitution donnée lorsque le modèle actuel est structurellement conforme au modèle paramètre. Dans ce qui suit, nous représentons la validité de l'application d'un template sur un modèle à l'aide du symbole $|-->$, présenté en figure 3.38.
Un exemple est donné en figure 3.39. On souhaite appliquer l'*aspectual template ObserverPattern* sur *AgencyClient*.
Pour une substitution S donnée, l'aspectual template est applicable sur ce modèle puisque le modèle paramètre et le modèle actuel sont structurellement conformes. Par exemple, l'élément actuel *capacity* est substitué avec le paramètre *value* et ces deux éléments de modèles sont respectivement reliés aux classes *Agency* et *Subject*.

5. À la manière des branches dans les Systèmes de Gestion de Versions (VCS : Version Control Systems), tels que [27, 87, 45]

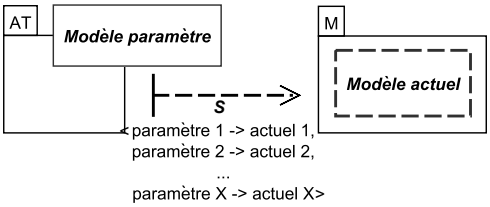


FIGURE 3.38 – Validité de l'application d'un *aspectual template* sur un modèle

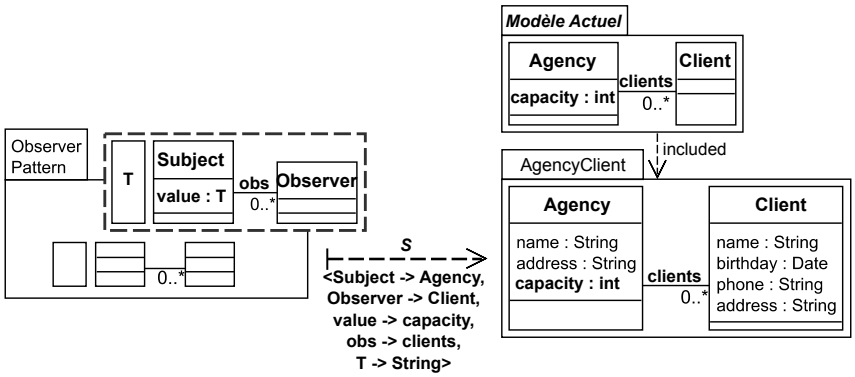


FIGURE 3.39 – Validité de l'application d'*ObserverPattern* sur *AgencyClient*

3.5.1.2 Validité d'une application et surmodèles

Considérons un aspectual template AT appliqué sur un modèle M selon un ensemble de substitutions S. M est un sous-modèle du modèle M'. On cherche à savoir si AT est applicable sur M' selon S en étudiant la relation d'inclusion entre le modèle actuel et M'. Cette situation est représentée en figure 3.40.

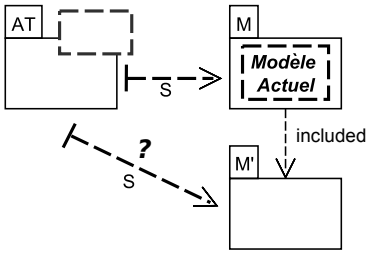


FIGURE 3.40 – M inclus dans M'

Le modèle actuel étant inclus dans M, lui-même inclus dans M', on peut en déduire que le modèle actuel est inclus dans M' et donc que l'application de AT est valide selon S sur M', comme présenté en figure 3.41.
Cette règle est mise en œuvre dans l'exemple de la figure 3.42. On souhaite ajouter des fonctionnalités d'observation entre une agence et ses clients. Pour cela, le template représentant le pattern observateur est utilisé et appliqué sur le modèle *AgencyClient*.
CarAgencyClient est un surmodèle de *AgencyClient* et résulte de la fusion (*"merge"* du standard *UML*) de *AgencyClient* et *CarAgency*. *CarAgencyClient* inclus donc *AgencyClient* et le modèle actuel structurellement conforme au modèle paramètre de *ObserverPattern*.

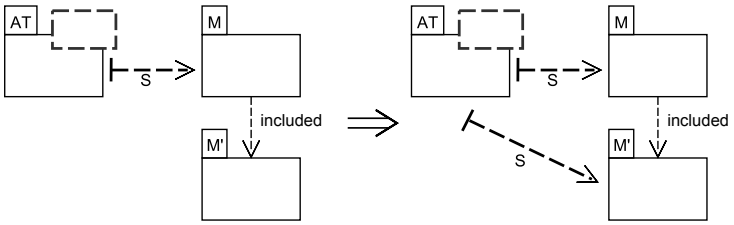


FIGURE 3.41 – *AT* applicable au surmodèle M'

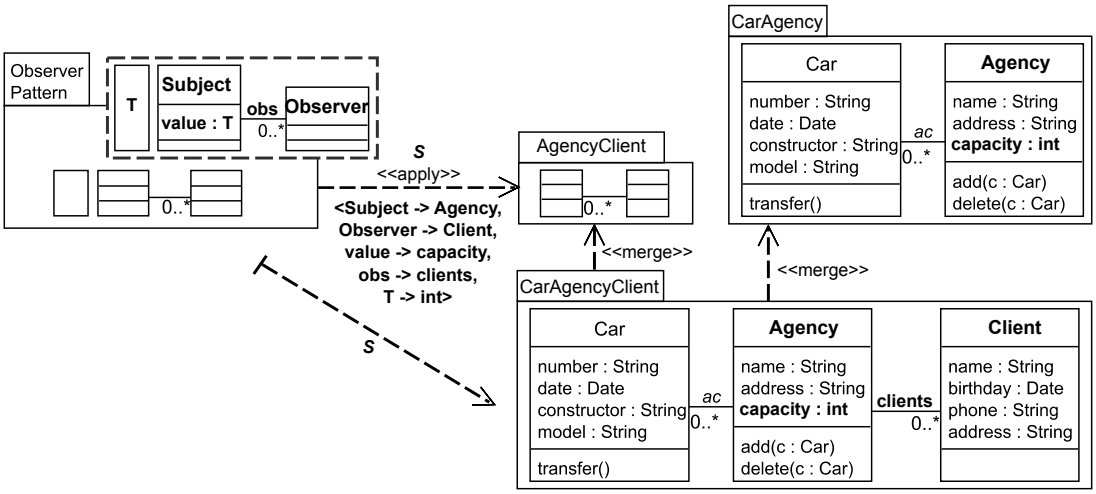


FIGURE 3.42 – *ObserverPattern* applicable sur le surmodèle *CarAgencyClient*

Ainsi, *ObserverPattern* peut aussi être appliqué sur *CarAgencyClient* selon S, afin d'ajouter à ce modèle les fonctionnalités d'observation entre une agence et ses clients, tout en prenant en compte les éléments apportés par la fusion.

3.5.1.3 Validité d'une application et sous-modèles

Considérons un aspectual template AT, appliqué sur un modèle M selon un ensemble de substitutions S, tel qu'exposé en figure 3.43. M possède un sous-modèle M'. On cherche à déterminer si AT est applicable sur M' selon S, en considérant la relation d'inclusion entre le modèle actuel et M'.

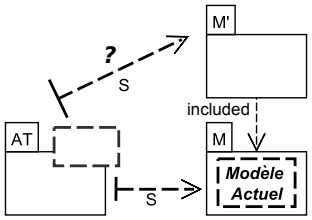


FIGURE 3.43 – M' sous-modèle de M

Afin que AT soit applicable, la règle est que le modèle actuel doit être inclus dans M'. De cette façon, on garantit que M' dispose de la structure requise pour l'application de AT (conformité structurelle). Ceci est présenté en figure 3.44.
Il est important de noter que le modèle actuel est ainsi le plus petit sous-modèle de M pour

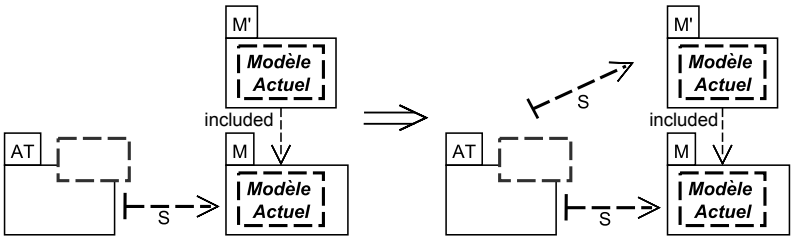


FIGURE 3.44 – *AT* applicable sur le sous-modèle M'

lequel l'application de *AT* est valide.
La validité de l'application sur un sous-modèle est illustré avec l'exemple de la figure 3.45. Le *template ObserverPattern* est appliqué selon S sur *CarAgencyClient*. On porte notre attention sur la partie concernant l'observation de l'agence par ses clients. *AgencyClient* est donc extrait de *CarAgencyCLient*.

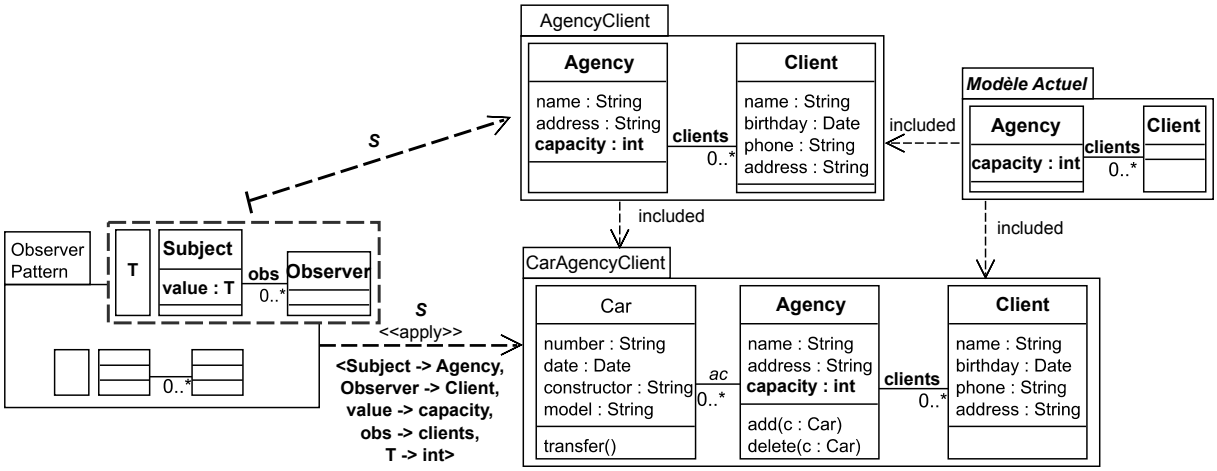


FIGURE 3.45 – *ObserverPattern* applicable selon S
sur le sous-modèle *AgencyClient*

Puisque *AgencyClient* inclut le modèle actuel, *ObserverPattern* peut être appliqué selon S sur ce sous-modèle de *CarAgencyCLient*.

3.5.1.4 Synthèse : Validité d'une application sur une hiérarchie de modèles

En section précédente, nous avons vu qu'un aspectual template est applicable sur un surmodèle M' de M selon un ensemble de substitutions S donné. De façon plus générale, un *aspectual template* est applicable selon S sur toutes les hiérarchies de modèles constituées des surmodèles de M. Cette généralisation est liée au fait que tous les surmodèles incluent, par transitivité, le modèle actuel.
Concernant les sous-modèles, il est possible de tenir un raisonnement similaire. Lorsque l'application d'un aspectual template sur un modèle est valide selon S, le template est aussi applicable selon cette substitution sur l'ensemble des sous-modèles incluant le modèle actuel.
Pour résumer, un aspectual template *AT* appliqué selon un ensemble S sur un modèle M est toujours applicable selon S sur une hiérarchie constituée des surmodèles du modèle actuel de M. Cette règle est synthétisée en figure 3.46.

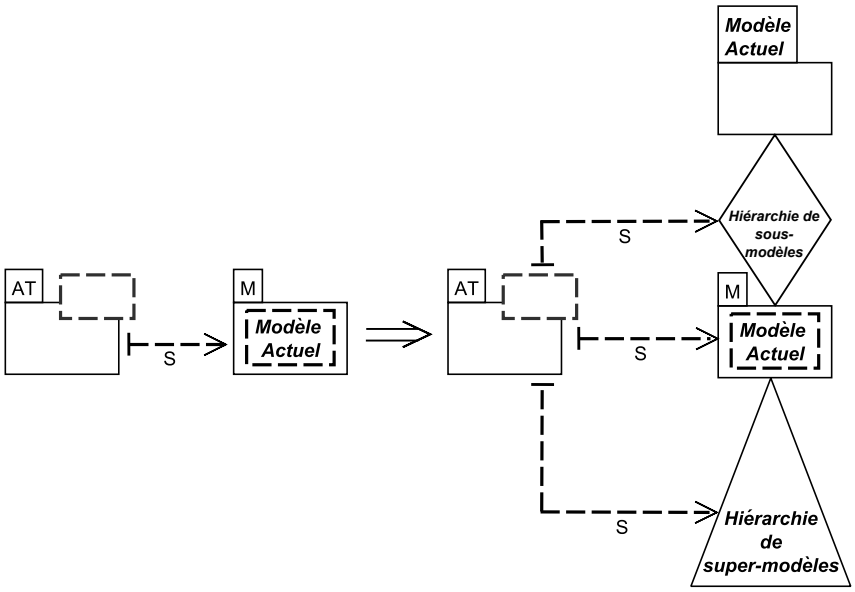


FIGURE 3.46 – *AT* applicable selon *S* sur toute hiérarchie de surmodèles du modèle actuel

Cette règle est illustrée par l'exemple de la figure 3.47.
On souhaite intégrer des fonctionnalités d'observation entre une agence et ses clients à *QueryableCarAgencyClient*. Le *template ObserverPattern* est donc appliqué selon *S* sur celui-ci. Un second modeleur souhaite travailler sur ces fonctionnalités mais sans celles concernant le requêtage. Il extrait donc *CarAgencyClient* de *QueryableCarAgencyClient*. Avec la règle de la validité de l'application d'un aspectual template sur des sous-modèles, il peut appliquer *ObserverPattern*. Deux autres modeleurs s'intéressent respectivement aux parties concernant *(1)* l'agence et ses clients et *(2)* l'agence et les voitures gérées. Les modèles correspondant, *AgencyClient* et *CarAgency*, sont donc extraits de *CarAgencyClient*. Avec la règle de la validité de l'application d'un aspectual template sur une hiérarchie de sous-modèles, le modeleur ayant extrait *AgencyClient* peut intégrer les fonctionnalités du template, alors que celui ayant extrait *CarAgency* ne peut pas appliquer *ObserverPattern* selon la substitution. En effet, bien que les deux sous-modèles de *CarAgencyClient* aient des concepts en commun, *CarAgency* n'inclut pas le modèle actuel.
Enfin, un modeleur débute la conception du système de facturation de l'agence et modélise *AgencyReceipt*, dans la partie droite de la figure 3.47, sans considérer les véhicules. Puisque ce modèle est surmodèle du modèle actuel, *ObserverPattern* peut être appliqué sur *AgencyReceipt*, bien qu'il n'existe aucune relation d'inclusion entre ce modèle et *QueryableCarAgencyClient*.

3.5.1.5 Validité de l'application de plusieurs templates sur un modèle

Après avoir étudié la validité de l'application d'un template sur un modèle, nous terminons par l'étude de validité de l'application de plusieurs templates.
Considérons un template AT applicable sur un modèle M selon une substitution S. D'après la règle d'applicabilité sur des surmodèles, AT est également applicable selon S sur toute application de template (AT' selon S') sur M. Ceci s'explique par le fait que le résultat d'une application de template sur M est un surmodèle de M. Cette situation est présentée en figure 3.48.
Tel que présenté en figure 3.49, deux séquences d'applications sont alors possibles.
De plus, puisque les substitutions S et S' portent uniquement sur le modèle M, la propriété 1

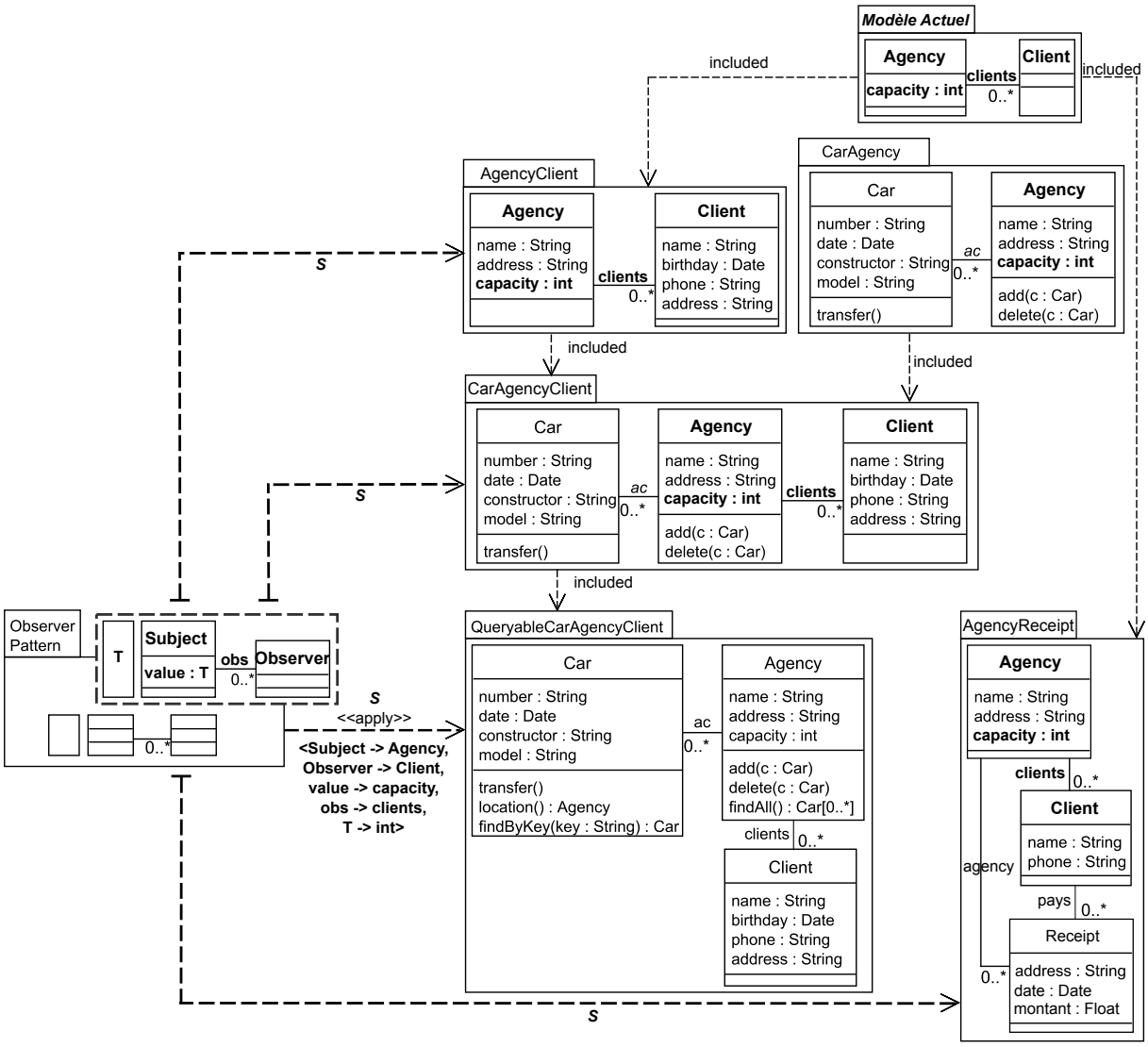


FIGURE 3.47 – *ObserverPattern* applicable selon S sur *AgencyClient*, *CarAgencyClient* et *AgencyReceipt*

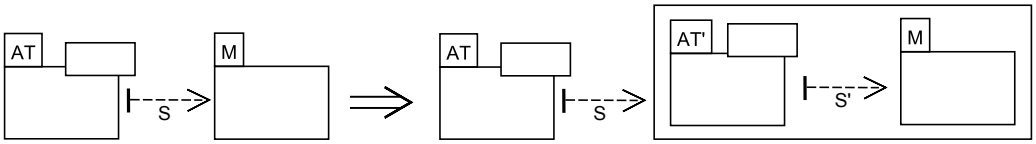


FIGURE 3.48 – Applicabilité et enchaînement d'applications

de Muller [73] permet de déduire que les résultats de :

- l'application de AT selon S sur le résultat de AT' selon S' sur M,
- l'application de AT' selon S' sur le résultat de AT selon S sur M

sont symétriquement les mêmes.

On observe ceci dans l'exemple de la figure 3.50. On souhaite ajouter au modèle *CarClientAgency* des fonctionnalités de requêtage entre une agence et les voitures gérées par cette dernière et des fonctionnalités d'observation entre une agence et les clients auxquels les voitures sont

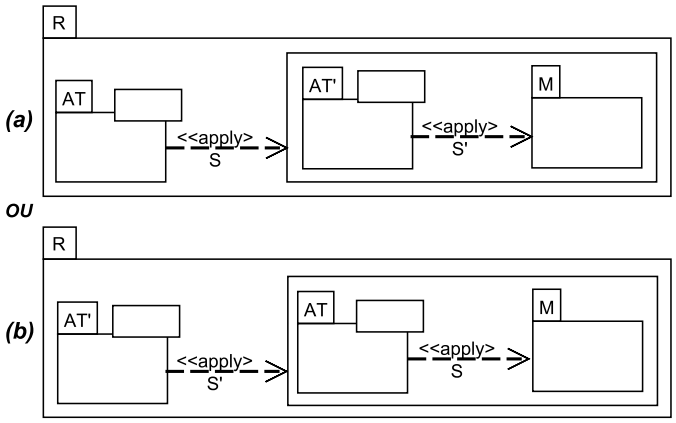


FIGURE 3.49 – Même modèle résultat R quel que soit l'ordre d'applications (Muller [73])

louées. Pour cela, on utilise les templates *QueryingPattern* et *ObserverPattern*. *ObserverPattern* est appliqué sur le résultat de l'application de *QueryingPattern*, résultant ainsi sur le modèle *QueryableCarObservableAgency*.

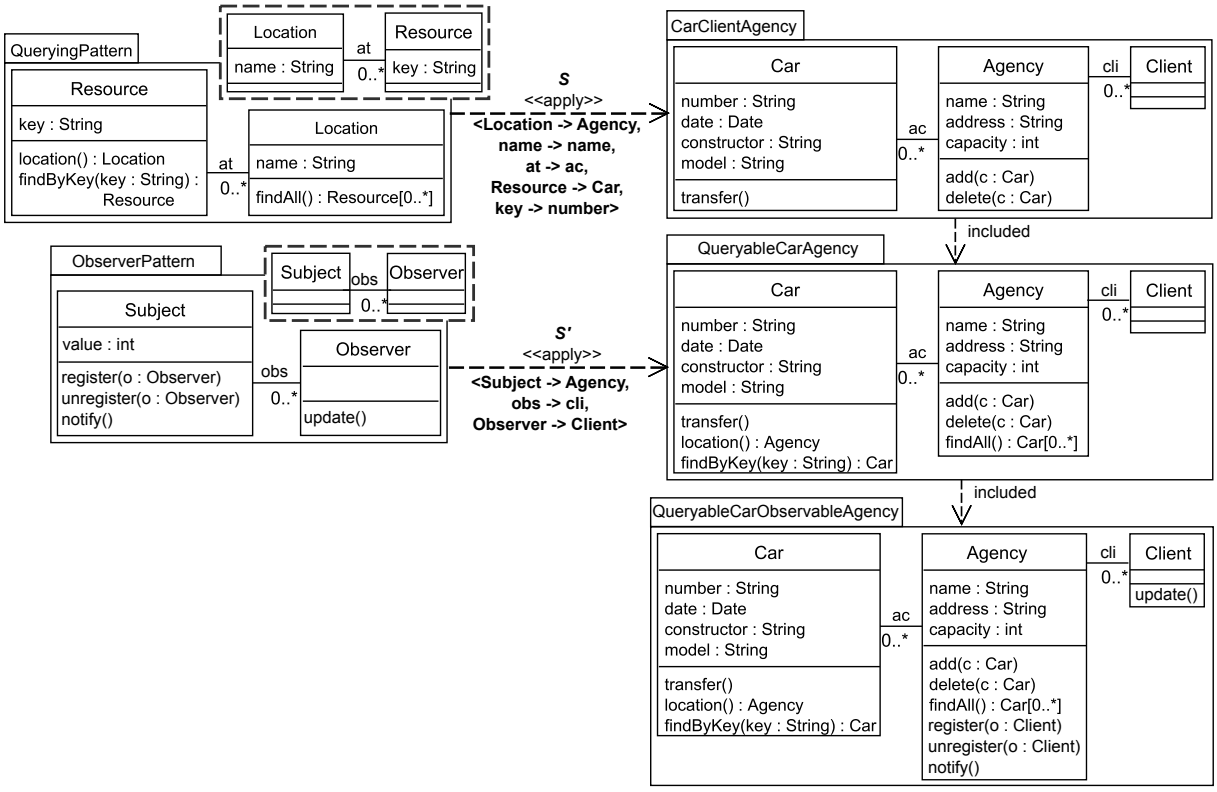


FIGURE 3.50 – Première séquence d'applications possible de *ObserverPattern* et *QueryingPattern*

L'exemple de la figure 3.51 présente la seconde séquence d'applications possible, menant au même modèle résultat *QueryableCarObservableAgency*.

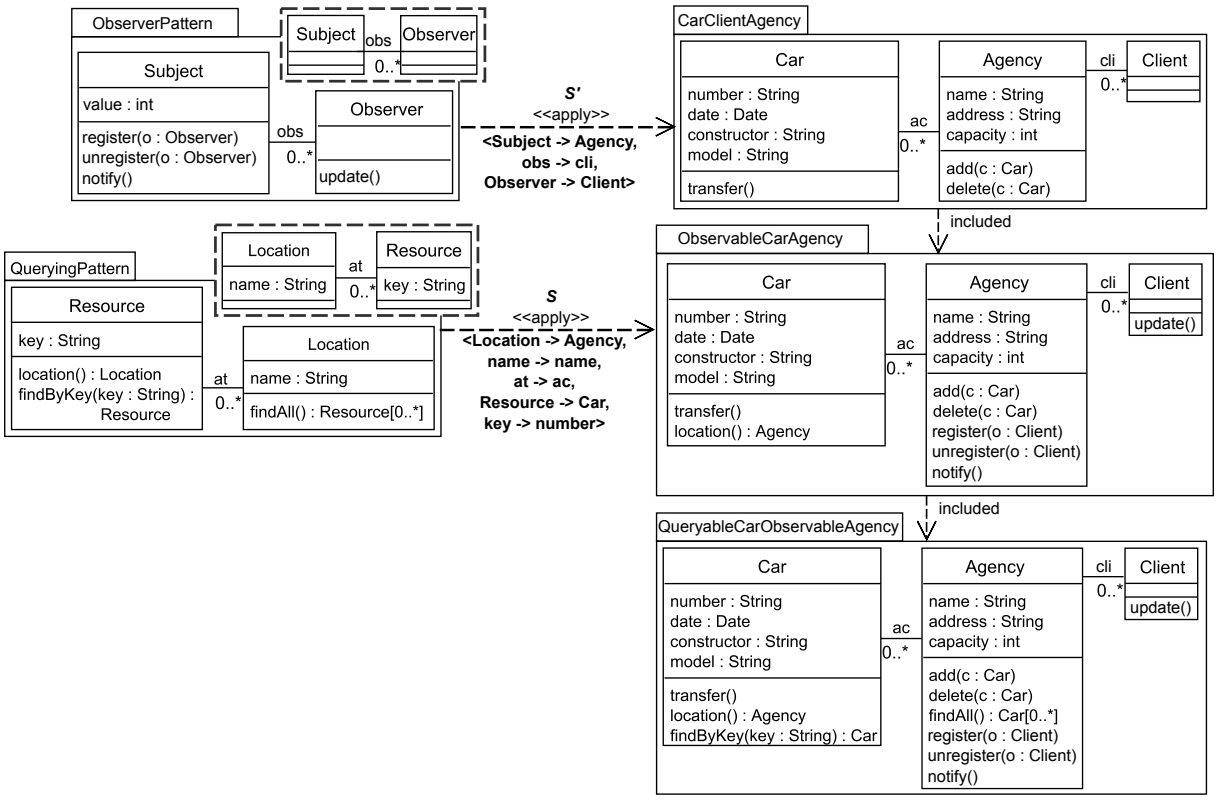


FIGURE 3.51 – Seconde séquence d'applications possible de *ObserverPattern* et *QueryingPattern*

3.5.2 Applications d'un aspectual template sur une hiérarchie : Inclusion entre résultats

Dans la section précédente, nous avons vu l'application d'un aspectual template sur une hiérarchie de modèles. Dans cette section, nous nous intéressons en particulier aux relations entre les modèles résultant de cette application. De telles relations permettent de continuer à tirer profit des règles d'applicabilité pour les hiérarchies formées par les modèles résultats. On peut notamment citer le cas intéressant de l'application de plusieurs templates à une hiérarchie de départ. À chaque application de template, les templates restent applicables sur la hiérarchie de modèles résultats. Nous considérons deux cas principaux d'applications, comme illustré par la figure 3.52 :

- **a)** les applications identiques d'un aspectual template AT sur une hiérarchie de modèles (M et M'), c'est-à-dire lorsque les ensembles de substitutions S et S' sont identiques (sous-section 3.5.2.1),
- **b)** des applications différentes du template, *i.e.* S différent de S' (sous-section 3.5.2.2).

Rappelons que nous étudions ici uniquement l'application totale de template et donc que S et S' sont égaux en paramètres formels.

3.5.2.1 Applications identiques ($S = S'$)

Ce cas est présenté en figure 3.52(a). Dans cette situation, un aspectual template AT est appliqué sur un modèle M avec un ensemble de substitutions S et est aussi appliqué sur un surmodèle M' avec un ensemble de substitutions S' identique à S.
On cherche à déterminer si le résultat de la première application est inclus dans celui de la se-

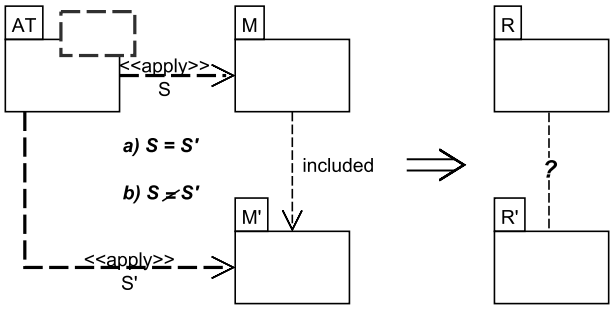


FIGURE 3.52 – Étude de la relation entre R et R' via des applications *(a)* identiques ($S = S'$) et *(b)* différentes ($S \neq S'$)

conde. Afin d'y répondre, nous pouvons appliquer le raisonnement qui suit.
Pour rappel (cf. section 3.3), le modèle résultat (ou *bound model*) R de l'application d'un aspectual template AT sur un modèle M selon S est la fusion de l'instance I de AT avec M (cf. figure 3.53).

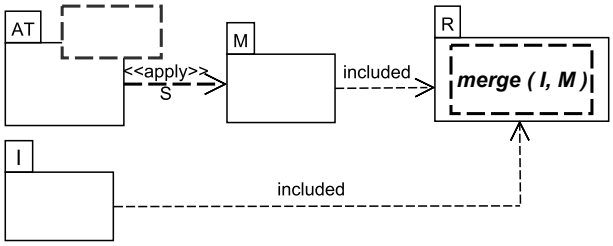


FIGURE 3.53 – Le contexte M et l'instance I sont inclus dans le résultat R

Comme illustré en figure 3.54, l'aspectual template AT est appliqué à l'identique sur M et M' (**a**)). Cela entraîne la fusion de M et M' avec la même instance effectuant les mêmes enrichissements dans R et R' (**b**)). M étant inclus dans M' (**c**)), alors R est inclus dans R'.

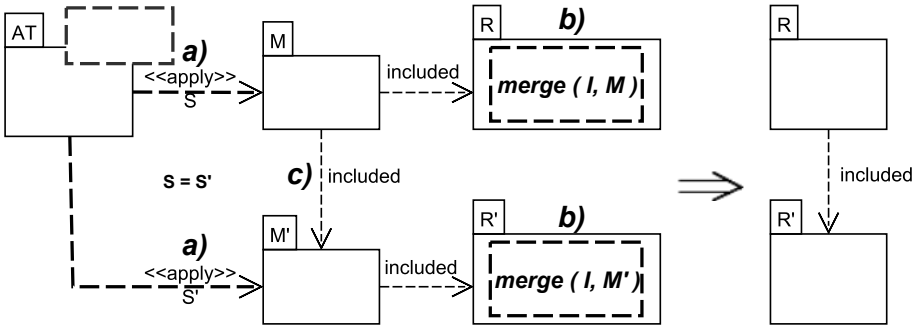


FIGURE 3.54 – R inclus dans R' pour $S = S'$

L'exemple de la figure 3.55 montre l'inclusion du modèle d'instance dans les résultats ainsi que l'inclusion entre ces derniers.

Dans cet exemple, on applique l'aspectual template pour injecter les mêmes fonctionnalités. On observe alors une relation d'inclusion entre les résultats.

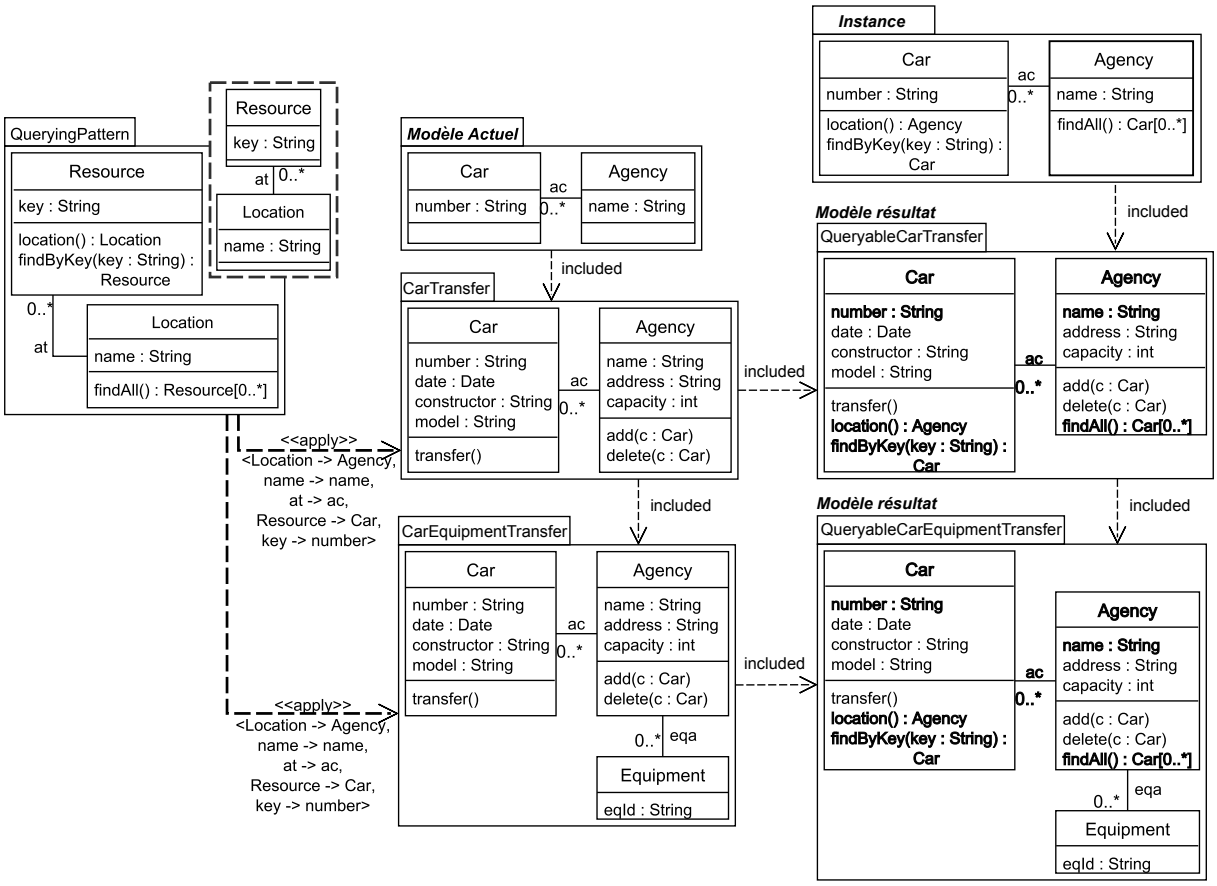


FIGURE 3.55 – *QueryableCarTransfer* inclus dans *QueryableCarEquipmentTransfer*

3.5.2.2 Applications différentes ($S \neq S'$)

Dans cette section, nous étudions le second cas concernant l'application de AT sur M et M', avec des ensembles de substitutions S et S' différents.

Lorsque AT est appliqué sur M et M' selon S et S' l'impact du tissage des éléments spécifiques au travers de ces substitutions peut être différent. La relation d'inclusion entre les modèles résultats dépend de cet impact.

Nous calculons cet impact en calculant la fermeture des ingrédients spécifiques dans chaque instance. Puis, nous extrayons le modèle correspondant. Si cet impact est identique, on peut en déduire une inclusion des modèles résultats.

Dans l'exemple de la figure 3.56, on travaille sur le sous-modèle *CarTrading* qui concerne la manière dont les voitures d'une agence sont commercialisées.

Les voitures étant louées par l'agence aux clients, on applique le *template Allocation* sur *CarTrading*. Quant à la location d'équipements, celle-ci est représentée dans le modèle *CarEquipmentTrading*. Ces équipements étant loués aux clients, on applique donc le même template sur le modèle *CarEquipmentTrading*, avec un ensemble de substitutions différent de l'application précédente.

Les modèles résultats, présentés dans la partie droite de la figure 3.57, sont liés entre eux par une relation d'inclusion. Dans cette figure, les instances sont calculées selon S et S'. L'ensemble des éléments du modèle spécifique est relié aux mêmes éléments actuels dans ces instances, comme on peut le voir dans le modèle spécifique clos, en bas de la figure. Par exemple, l'élément *nbAl-*

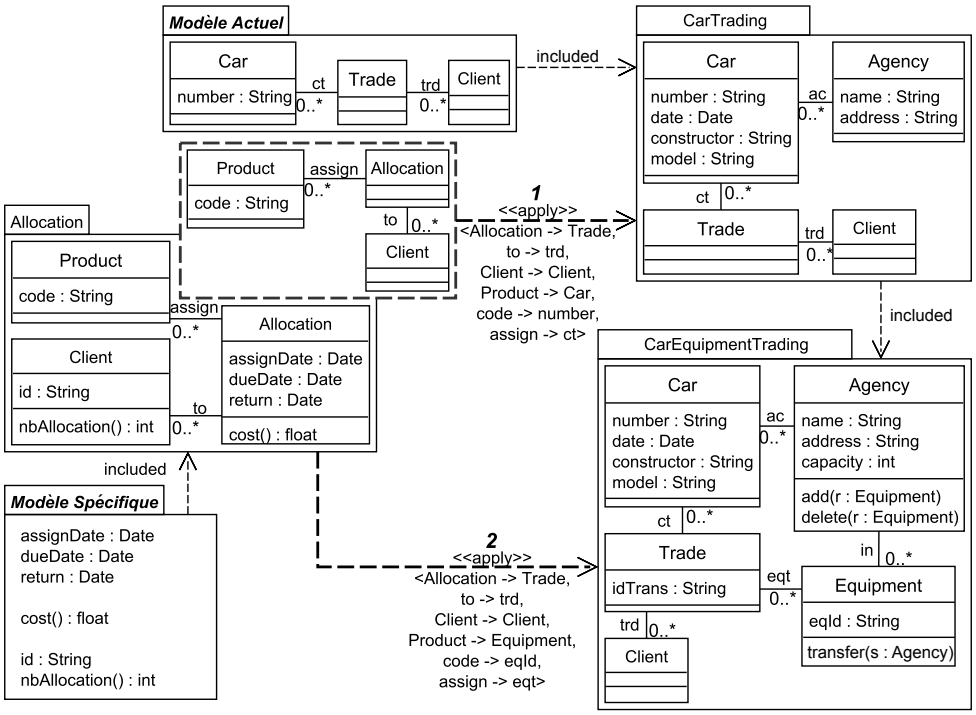


FIGURE 3.56 – $S' \cap S \neq \emptyset$

location, du modèle spécifique, est relié à l'élément actuel *Client*.

On observe ainsi que même lorsque les instances ne sont pas liées entre elles par une relation d'inclusion, les modèles résultats peuvent l'être si les modèles spécifiques clos dans les instances le sont.

Lorsque les éléments du modèle spécifique clos dans les instances sont différents, cela signifie que les éléments du modèle spécifique sont liés à des éléments du modèle actuel *différents*. Les modèles résultats possèdent alors des contraintes structurelles que l'autre n'a pas (celles liant les éléments du modèle spécifique à ceux du modèle actuel). Ainsi, il ne peut y avoir de relation d'inclusion entre les modèles résultats.

L'exemple en figure 3.59 présente une telle situation avec l'application de *QueryingPattern* sur les modèles *AgencyClient* et *AgencyClientCar*. La première application porte sur l'agence et les clients, alors que la seconde concerne l'agence et les voitures louées. Respectivement aux applications, les modèles spécifiques sont liés à *Client* et *Agency* et à *Car* et *Agency*. Ceci conduit donc à des enrichissements différents des contextes, c'est-à-dire des modèles spécifiques clos, dans les instances qui ne sont pas liés. Ainsi, aucune relation d'inclusion n'existe entre les modèles résultats.

Ceci est utile afin de déterminer des relations entre les modèles résultats. La figure 3.58 présente schématiquement ceci.

Un modeleur applique un template AT une première fois sur un modèle M, produisant le modèle R. Un second modeleur applique un autre template AT' sur M, résultant en R'. Le premier modeleur doit ajouter des fonctionnalités de AT sur des éléments apportés par AT' et applique donc AT sur R', produisant R''. Avec la règle définie plus haut, on sait déterminer *a priori*, c'est-à-dire avant de produire R'', l'impact des applications $S1$ et $S3$. Ici, elles ont toutes deux le même impact sur M et R' et on peut donc déduire que le modèle résultat de $S1$, R, est inclus dans R''. En considérant ceci, l'application $S1$ peut être ignorée, réduisant ainsi le nombre des

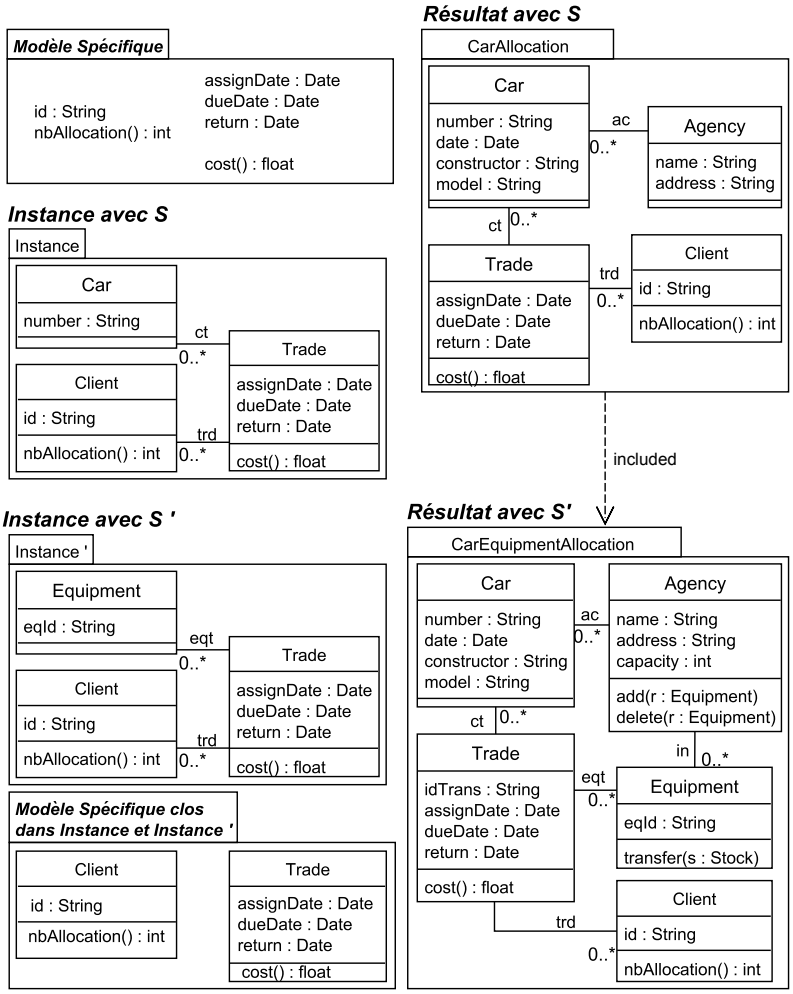


FIGURE 3.57 – Instanciations de *Allocation* et fermeture des éléments du modèle spécifique

applications à effectuer afin d'obtenir R''. L'effort nécessaire à la conception incrémentale d'un système est ainsi diminué, comme représenté dans la partie droite de la figure 3.58.
La figure 3.60 illustre un exemple concret avec les *templates Allocation et EquipmentManagement*. Cet exemple reprend celui de la figure 3.56, à la différence que la partie concernant l'équipement est ajoutée par le *template EquipmentManagement*. On détermine, avec la règle présentée plus haut, que l'application $S1$ de *Allocation* sur M est superflue, puisque les applications $S1$ et $S3$ ont le même impact sur M et R' (les éléments impactés par $S1$ et $S3$ sont en gras dans les modèles résultats R et R''). Ainsi, afin d'obtenir R'', seule l'application de *Allocation* sur R' avec $S3$ est nécessaire.

3.6 Conclusion

Dans ce chapitre, nous avons présenté le cœur de nos propositions. Dans un premier temps, nous avons présenté l'IDMBT, en exploitant ses deux sous-espaces de conception (conception de templates et conception de systèmes), ses activités de modélisation et acteurs associés. Par la suite, nous nous sommes concentré sur plusieurs opérations. Premièrement, nous avons étudié

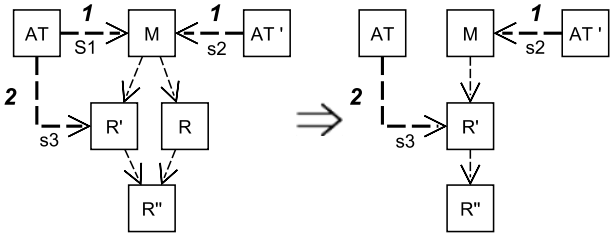


FIGURE 3.58 – Simplification de conception incrémentale

l'instanciation d'aspectual templates, obtenue par isolation du contexte dans le binding. Cette instanciation se positionne sur des activités de construction de modèles. Ensuite, nous avons examiné la détection d'instanciation de templates qui s'avère utile pour la compréhension de modèles et les besoins de traçabilité. Puis, nous avons répondu à des besoins d'évolution guidée par les templates, en étudiant le retrait d'instances présentes dans les modèles. Une dernière partie a été consacrée à l'application de templates dans le cadre des hiérarchies de modèles. Ceci a conduit à examiner l'applicabilité d'un template sur un sous-modèle, un surmodèle et, plus généralement, une hiérarchie dans son ensemble selon un ensemble de substitutions. Le résultat est un ensemble de règles de validité d'applications pouvant être employées afin d'élaborer des outils manipulant des hiérarchies dans des contextes comme le versionnement, la conception incrémentale et la modélisation en équipe. Nous avons aussi étudié les relations entre les modèles résultants des applications d'un template sur une hiérarchie de modèles. À partir de cette étude, nous avons spécifié une règle définissant l'impact des applications sur une hiérarchie lorsque les substitutions sont différentes. Cette règle est utile dans le cadre de la conception incrémentale et en équipe d'un modèle de système, afin de réduire l'effort de modélisation.
Dans le chapitre suivant, nous mettons en pratique ces résultats au travers d'opérateurs, implémentés dans un atelier de modélisation, puis nous expérimentons et évaluons leur utilisation au travers d'un cas concret d'application à une architecture de services web.

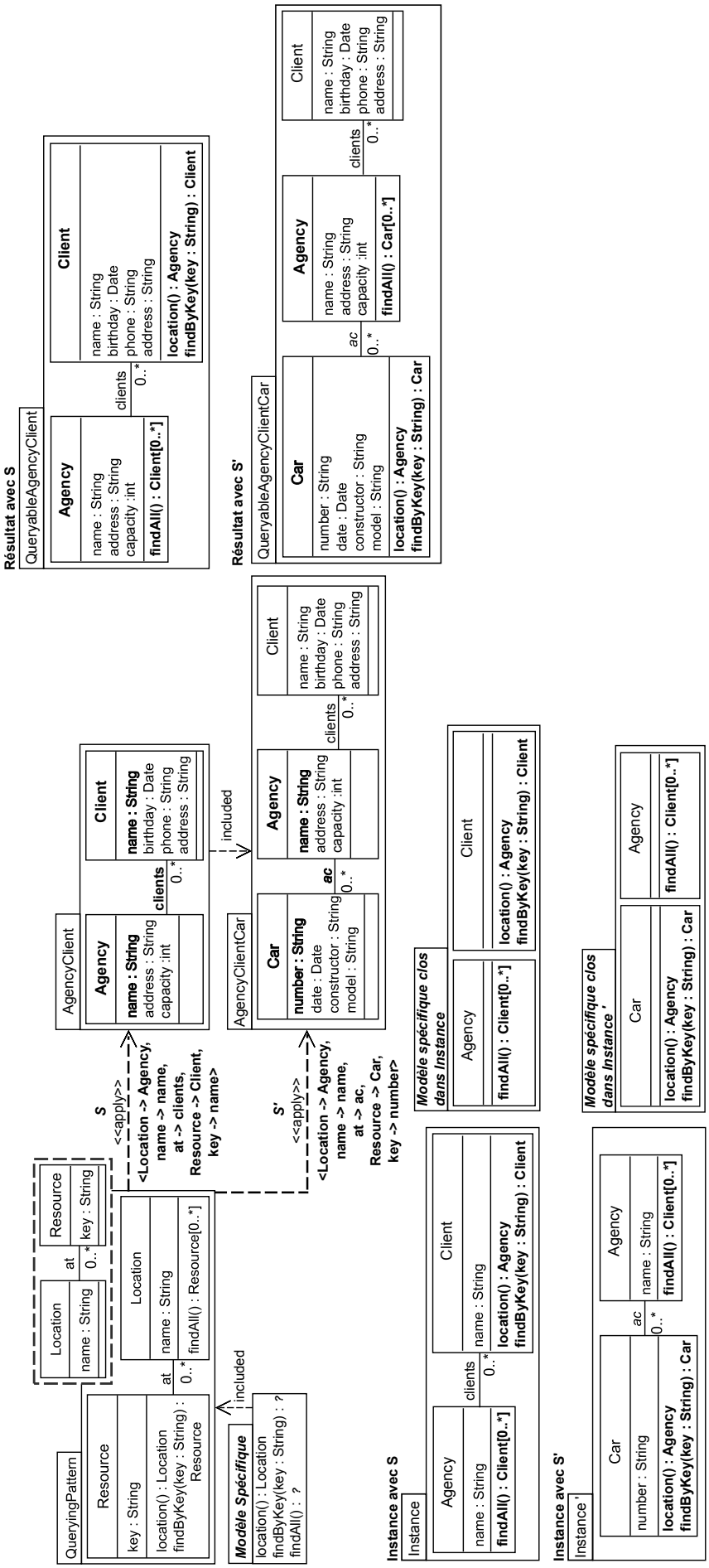


FIGURE 3.59 – Modèles spécifiques clos relatifs aux deux instances - Aucune relation entre les modèles résultats

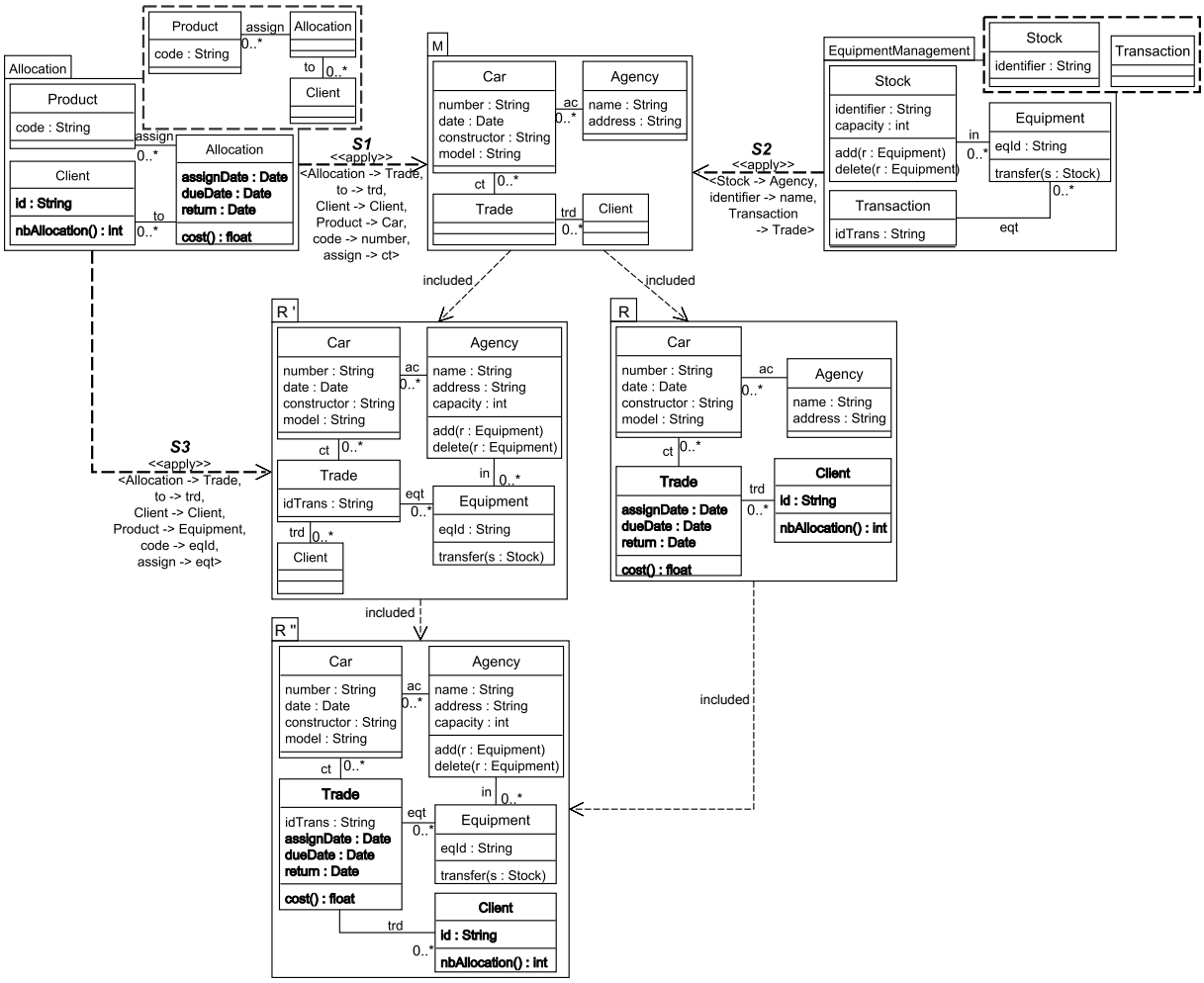


FIGURE 3.60 – Conception incrémentale - Exemple

Chapitre 4

Mise en œuvre

Dans le chapitre précédent, l'approche présentée permet de construire des modèles via la (ré)utilisation de modèles et de modèles génériques que sont les *templates*, au-travers de différentes activités. Les hiérarchies de modèles ont été considérées et leur étude avec les applications de *templates* ont permis de dégager des règles d'ingénierie (*validité d'applications et relations entre modèles résultats*).
Selon ces règles et activités citées plus haut, nous avons défini plusieurs opérateurs de modèles et *templates* que nous présentons dans la section 4.1. Pour chaque opérateur, nous présentons son objectif, sa formulation, sa définition sous la forme d'un algorithme, montrant les relations entre les opérateurs ainsi que sa représentation graphique.
En section 4.2, nous présentons un atelier de modélisation utilisant deux moteurs, permettant de manipuler respectivement des *aspectual templates* et sous-modèles et dans lesquels ont été *implémentés* les opérateurs précédents. Une application de serveur REST d'informations sera présentée au chapitre 5.

4.1 Opérateurs

Dans cette section, nous présentons un jeu d'opérateurs mettant en œuvre l'IDMBT, proposée en section 3.2. Ces opérateurs en utilisent d'autres qui agissent sur les modèles, notamment le modèle *core* d'un *template* et ses sous-modèles *paramètre et spécifique*. Les sous-sections suivantes présentent respectivement la notation utilisée pour formuler ces opérateurs (sous-section 4.1.1), les opérateurs de modèles (sous-section 4.1.2) et les opérateurs de *templates* (sous-section 4.1.3).

4.1.1 Notation et types

4.1.1.1 Notation

Pour tout type T, on note :
- $\mathscr{E}(T)$, son extension, c'est à dire l'ensemble de ses valeurs possibles,
- $x : T$, une valeur quelconque de T, soit $x \in \mathscr{E}(T)$,
- $x : T^*$, n (quelconque) valeurs de T, soit $x \in \mathscr{E}(T^*) = 2^{\mathscr{E}(T)}$. Cette notation permet de désigner n'importe quelle collection de valeurs de T, comme en *UML* (un tableau, une liste, ...),
- $x : T \times T$, un couple d'éléments de T, soit $x \in \mathscr{E}(T) \times \mathscr{E}(T)$. Cette notation permettra notamment de désigner une substitution unitaire spécifié dans un *binding* de *template*. On notera $first(x)$ le premier élément du couple et $second(x)$ son second élément.
- Et par extension :
  - $x : (T \times T)^*$, un ensemble de couples de T, soit $x \in \mathscr{E}((T \times T)^*) = 2^{\mathscr{E}(T) \times \mathscr{E}(T)}$. Cette notation permettra notamment de désigner un ensemble de substitutions spécifié par un *binding* de *template*,

- $firsts(x)$, les premiers éléments des couples de x ($\{y : T \mid \exists z \in x \; avec \; y = first(z)\}$),
- $seconds(x)$, les seconds éléments des couples de x ($\{y : T \mid \exists z \in x \; avec \; y = second(z)\}$).

4.1.1.2 Types et opérations primitives

Pour formuler les opérateurs, nous utilisons les types suivants :

ModelElement Éléments de modèle, typiquement la métaclasse `Element` racine du métamodèle UML.

Model
Un modèle $m : Model$ est vu comme un graphe orienté (V, A) où :
- V (*vertices*) $\subseteq \mathscr{E}(ModelElement)$: les $ModelElement$ du modèle que l'on peut obtenir par l'opérateur :
  $modelElements(Model) : ModelElement^*$
- A (*arrows*) $\subseteq V \times V$: les contraintes de dépendance structurelle entre ses $ModelElement$ que l'on peut obtenir par l'opérateur :
  $modelConstraints(Model) : (ModelElement \times ModelElement)^*$
  avec : $modelConstraints(m) \subseteq modelElements(m) \times modelElements(m)$.

Template Templates de modèles dans leur forme la plus générale, telle que spécifiée par UML.
Un *template* t peut être représenté par un couple $(core, parameters)$ où :
- *core* : $Model$ est le modèle offert par le *template*, que l'on peut obtenir par l'opérateur :
  $core(Template) : Model$
- *parameters* : $ModelElement^*$ est l'ensemble des $ModelElement$ de $core(t)$ exposés comme paramètres, que l'on peut obtenir par l'opérateur :
  $parameters(Template) : ModelElement^*$
  avec : $parameters(t) \subseteq modelElements(core(t))$

AT Aspectual Templates.
Pour rappel (cf. section 3.1.2) les *aspectual templates* sont des $Template$ garantissant la propriété que leurs paramètres forment un modèle, plus précisément un sous-modèle bien formé de leur *core*. Un *aspectual template* at peut donc être représenté par un couple $(core, parameterModel)$ où :
- *core* : $Model$ est le modèle offert par le template que l'on peut obtenir par l'opération :
  $core(AT) : Model$
  comme pour tout $Template$
- $parameterModel$: $Model$ est le sous-modèle de $core(at)$ exposé comme paramètre, que l'on peut obtenir par l'opérateur :
  $parameterModel(AT) : Model$
  avec : $parameterModel(at)$ sous-modèle bien formé de $core(at)$.
  Par complément au $parameterModel$ de at, on considère son $specificModel : Model$ qui est le sous-modèle de $core(at)$ non-exposé comme paramètre et que l'on peut obtenir par l'extraction des éléments non-paramètres dans at :
  - $specificElements : ModelElement^*$

avec : $specificElements$ sous-ensemble de $modelElements(core(at))$.
Il est défini à l'aide des opérateurs $core(AT)$, $parameterModel(AT)$ et $modelElements(Model)$:
specificElements = modelElements(core(at)) \ modelElements(parameterModel(at)).

- $specificModel : Model$
  avec : $specificModel$ sous-modèle de $core(at)$.
  Il est défini à l'aide des opérateurs $core(AT)$ et $extract(Model, ModelElement^*)$ (cf. section 3.1.1) :
  specificModel=extract(core(at),specificElements)

4.1.2 Opérateurs de modèles

Pour chacun des exemples illustrant les opérateurs présentés, les modèles *CarHiringSystem* et *AgencyClient* de la figure 4.1 seront utilisés.

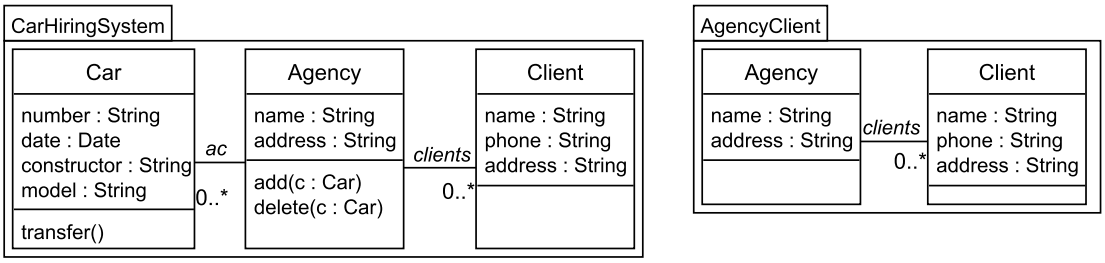


FIGURE 4.1 – *AgencyClient* et *CarHiringSystem*

$CarHiringSystem$:

$$modelElements(CarHiringSystem) = \{Car, Number, date, constructor, model, transfer, Agency, name, address, clients, Client, name, phone, address, ac, add, delete, String, Date\ \}$$

$$modelConstraints(CarHiringSystem) = \{\ (number, Car), (number, String), (date, Car), (date, Date), (constructor, Car), (constructor, String), (model, Car), (model, String), (transfer, Car), (ac, Car), (ac, Agency), (name, Agency), (name, String), (address, Agency), (address, String), (add, Agency), (add, Car), (delete, Agency), (delete, Car), (clients, Agency), (clients, Client), (name, Client), (name, String), (phone, Client), (phone, String), (address, Client), (address, String)\ \}$$

$AgencyClient$:

$$modelElements(AgencyClient) = \{\ Agency, name, address, clients, Client, name,$$

$$
\begin{aligned}
& & & phone, address, String\ \} \\
modelConstraints(AgencyClient) & = & \{\ & (name, Agency), (name, String), (address, Agency), \\
& & & (address, String), (clients, Agency), (clients, Client), \\
& & & (name, Client), (name, String), (phone, Client), \\
& & & (phone, String), (address, Client), (address, String)\ \}
\end{aligned}
$$

4.1.2.1 Extract

Objectif Pour rappel (cf. 3.1.1), l'opérateur *extract* permet de créer un sous-modèle par sélection d'un sous-ensemble d'éléments du modèle d'entrée.

Le sous-modèle obtenu est formé de ce sous-ensemble d'éléments de modèle et des contraintes de structure associées à ces éléments dans le modèle d'entrée. Cet opérateur a été décrit dans Carré et al. [24].

Formulation

extract(m :Model, e :ModelElement*) :Model
Avec : $e \subseteq modelElements(m)$

Algorithme

début
| $result = (e, \{(x, y) \in modelConstraints(m) | x, y \in e\})$
fin

Exemple La figure 4.2 illustre le résultat de l'extraction du modèle *AgencyClient* à partir de *CarHiringSystem* et du sous-ensemble des éléments de modèle suivants : *{Agency, name, address, clients, Client, name, phone, address}*.

$$
\begin{aligned}
result & = extract(& & CarHiringSystem, \\
& & & \{\ Agency, name, address, clients, Client, name, phone, \\
& & & phone, address, String\ \}\ ) \\
& = & (e, & \{\ (name, Agency), (name, String), (address, Agency), (address, String), \\
& & & (clients, Agency), (clients, Client), (name, Client), \\
& & & (name, String), (phone, Client), (phone, String), \\
& & & (address, Client), (address, String)\}
\end{aligned}
$$

Représentation graphique L'extraction est représentée par une relation de dépendance UML stéréotypée *extract* entre deux modèles. Cette relation va du sous-modèle (*AgencyClient*, cf. figure 4.2) vers le modèle dont il a été extrait ($CarHiringSystem$). L'ensemble d'éléments de modèle utilisé pour l'extraction est spécifié au niveau de la relation de dépendance, entre les symboles < et >.

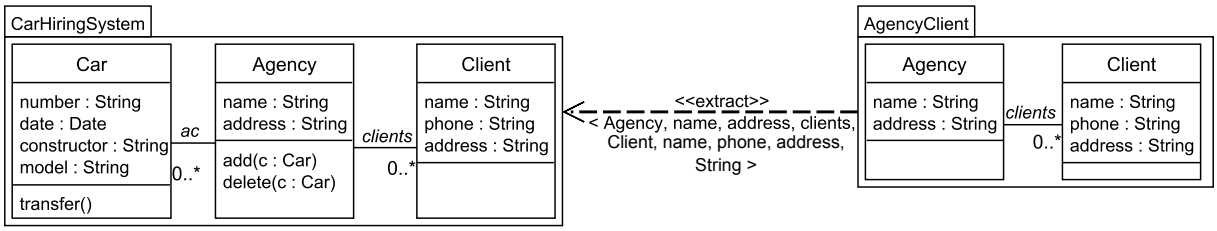


FIGURE 4.2 – Extraction de *AgencyClient* à partir de *CarHiringSystem*

4.1.2.2 Complétion d'ensemble d'éléments de modèle : Closure et Dependents

Les deux opérateurs suivants permettent de compléter un ensemble d'éléments de modèle avec les éléments qui en dépendent ou dont ils dépendent au sein d'un modèle.

Closure

Objectif Pour rappel (cf. 3.1.1), l'opérateur *closure* permet *de compléter un ensemble d'éléments avec ceux dont ils dépendent transitivement dans un modèle.*

Ces dépendances transitives sont déterminées selon les contraintes de structure du modèle. Ceci a été décrit dans Carré et al. [24].

Formulation

closure(m :Model, e :ModelElement*) :ModelElement*
Avec : $e \subseteq modelElements(m)$

Algorithme

début
  $result = e \cup \{y \in modelElements(m) \mid x \in e, (x, y) \in modelConstraints(m)\}$
  `/* Fermeture transitive */`
  si $e \neq result$ **alors**
    $result = closure(m, result)$
  fin
fin

Exemple Avec le modèle *AgencyClient* de la figure 4.2 et l'ensemble *{ clients }*, *closure* permet d'obtenir l'ensemble d'éléments *{ Agency, clients, Client }* dans *AgencyClient* : en effet, *clients* est une association reliée aux classes *Agency* et *Client*.

$$\begin{aligned} result &= closure(AgencyClient, \{\ clients\ \}) \\ &= \{\ Agency, clients, Client\ \} \end{aligned}$$

Dependents

Objectif Symétriquement à l'opérateur précédent, *dependents* permet *de compléter un ensemble d'éléments avec ceux qui en dépendent transitivement.*

Formulation

dependents(m :Model, e :ModelElement) :ModelElement**
Avec : $e \subseteq modelElements(m)$

Algorithme

début
  $result = e \cup \{y \in modelElements(m) \mid x \in e, (y, x) \in modelConstraints(m)\}$
  `/* Fermeture transitive */`
  si $e \neq result$ **alors**
    $result = dependents(m, result)$
  fin
fin

Exemple Pour le modèle *AgencyClient* et l'ensemble *{ Agency }*, *dependents* retourne l'ensemble *{ Agency, name, String, address, clients }* car *name, address* et *clients* sont des attributs et association liés à la classe *Agency* et *String* est relié indirectement à cette classe (via *name* et *address*).

$$\begin{aligned} result &= dependents(AgencyClient, \{\ Agency\ \}\ ) \\ &= \{\ Agency, name, String, address, clients\ \} \end{aligned}$$

4.1.2.3 Included

Objectif L'opérateur *included* permet *de déterminer si un modèle est inclus dans un autre modèle.*

Cet opérateur est défini par l'inclusion de leurs ensembles d'éléments et de leurs ensembles de contraintes de structures.

Formulation

included(m :Model, m' :Model) :Boolean

Algorithme

début
  $result = (modelElements(m) \subseteq modelElements(m')) \wedge (modelConstraints(m) \subseteq modelConstraints(m'))$

La figure 4.3 illustre l'inclusion du modèle *AgencyClient* dans *CarHiringSystem.*

Représentation graphique L'inclusion est représentée par une relation de dépendance *UML* stéréotypée *included* entre deux *modèles.* Cette relation va du sous-modèle (m, cf figure 4.3) à

son surmodèle (m').

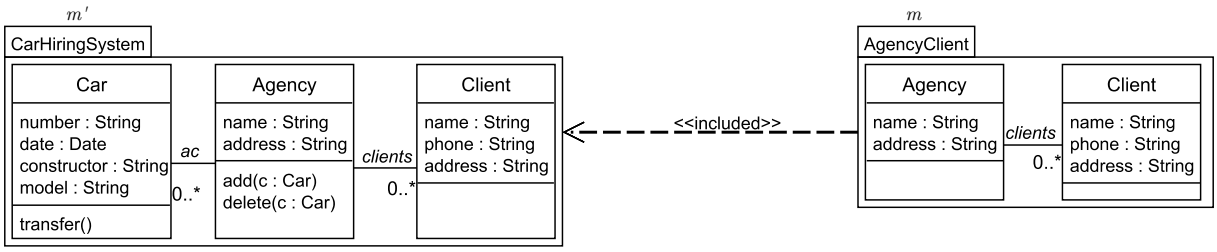


FIGURE 4.3 – Relation d'inclusion entre *CarHiringSystem* et *AgencyClient*

4.1.2.4 Merge

Objectif Il s'agit de l'opérateur *merge* d'UML (OMG [80]) qui *permet de fusionner deux packages.*

Pour rappel (cf. section 2), les deux packages fusionnés sont nommés le *merged package* et le *receiving package*. Le package résultat est quant à lui nommé *resulting package* et correspond au *receiving package* auquel ont été ajoutés les éléments du *merged package* : le *receiving package* est donc à la fois opérande et résultat. Les éléments de même nom sont fusionnés.

Formulation

merge (receivingPackage :Model, mergedPackage :Model) :Model

Comme précisé dans Dingel et al. [39], cet opérateur n'est pas symétrique. L'ordre des packages à fusionner influence le résultat.
Ceci est illustré en figure 4.4. Dans la partie haute, *CarAgency* est le *merged package*. L'attribut *managed* est de type *Car[0..*]* dans le package résultat. A l'inverse, dans le bas de la figure, *CarAgency* est utilisé comme *receiving package* : *managed* est alors de type *Client[0..*]* dans le *resulting package*.

Représentation graphique La fusion est représentée par une relation de dépendance *UML* stéréotypée *merge* entre deux *packages*. Cette relation va du *receiving package* (cf. figure 4.4) vers le *mergedPackage*.

4.1.2.5 Synthèse des opérateurs de modèles

Ces opérateurs de modèles (cf. table 4.5) sont nécessaires à la définition de ceux agissant sur des *templates* et présentés dans la section suivante.

TABLE 4.5 – Opérateurs de modèles

Opérateur	***Objectif***	***Signature***
extract	Crée un sous-modèle en l'extrayant d'un modèle selon un ensemble d'éléments de modèle	*extract(Model, ModelElement*) :Model*

closure	Complète un ensemble d'éléments de modèle avec ceux dont ils dépendent transitivement	*closure(Model, ModelElements*) :ModelElements**
dependents	Complète un ensemble d'éléments de modèle avec ceux qui en dépendent transitivement	*dependents(Model, ModelElements*) :ModelElements**
included	Détermine si un modèle est inclus dans un autre modèle	*included(m :Model, m' :Model) :Boolean*
merge	Fusionne deux modèles	*merge(Model, Model) :Model*

4.1.3 Opérateurs de *templates*

4.1.3.1 Apply

Objectif L'opérateur *apply construit un nouveau modèle en appliquant un template sur un modèle de base.*

Cet opérateur correspond à l'application aspectuelle décrite dans Vanwormhoudt et al. [103] et rappelé dans les chapitres 3.1 et 3.3.3.
La figure 4.5 illustre l'application du *template ObserverPattern* sur le modèle de base *AgencyClient*, permettant d'obtenir le *bound model ObservableAgencyClient.*

Formulation

$$
\begin{array}{l}
\textbf{\textit{apply(at :AT, m :Model, s :(ModelElement} \times \textbf{\textit{ModelElement)}}^*\textbf{\textit{) :Model}} \\
\text{Avec} \quad : \quad seconds(s) \subseteq modelElements(m) \\
\qquad\quad\; firsts(s) \subseteq modelElements(parameterModel(at)) \\
\qquad\quad\; \forall (x,a),\ (y,b) \in s : \\
\qquad\qquad\quad (\ \exists (x,y) \in modelConstraints(parameterModel(at))\ ) \\
\qquad\qquad\qquad\quad \Rightarrow\ (\ \exists (a,b) \in modelConstraints(m)\ )
\end{array}
$$

Le paramètre s représente l'ensemble de substitutions à réaliser. Cet opérateur tient compte de l'application partielle ($firsts(s) \subseteq modelElements(parameterModel(at))$).

Algorithme L'algorithme de l'opérateur *apply* est détaillé dans Vanwormhoudt et al. [103].

Représentation graphique La représentation est celle de Muller [73]. Elle est représentée par une relation de dépendance UML stéréotypée *apply* entre un *template* et un *modèle.* Cette relation va du *template* (cf. figure 4.5) vers le contexte. La substitution utilisée pour l'application se situe entre les symboles < et >.

4.1.3.2 Instantiate

Objectif L'opérateur *instantiate* (décrit section 3.3) *crée une instance de template à partir d'un modèle de base.*

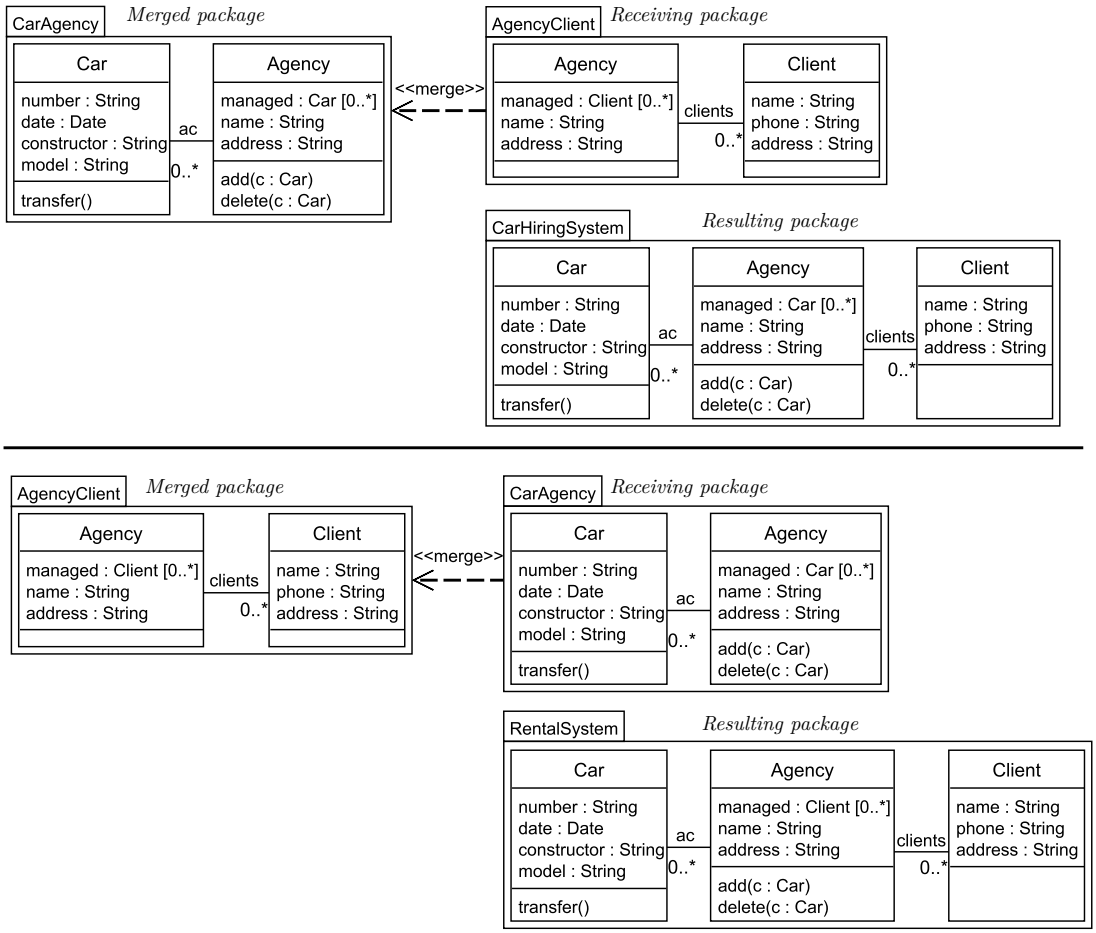


FIGURE 4.4 – Fusion de *CarAgency* et *AgencyClient*

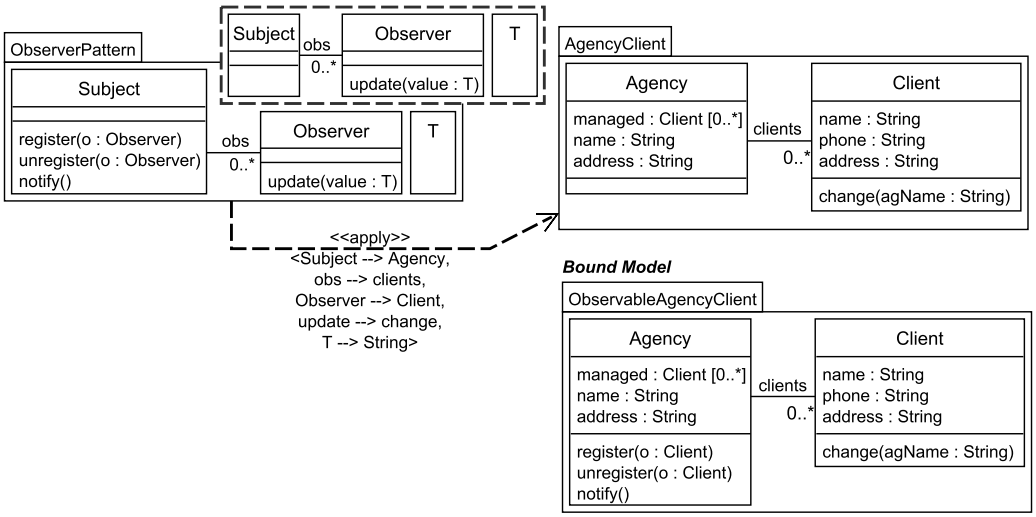


FIGURE 4.5 – Relation d'application aspectuelle entre *ObserverPattern* et *AgencyClient*

L'instanciation est définie avec l'opérateur *apply* quand le modèle de base se confond avec le modèle actuel.

Formulation

$$
\begin{array}{l}
\textit{instantiate(at :AT, m :Model, s :(ModelElement} \times \textit{ModelElement)}^{*}\textit{) :Model} \\
\text{Avec} \ : \quad seconds(s) \subseteq modelElements(m) \\
\qquad\quad firsts(s) \subseteq modelElements(parameterModel(at)) \\
\qquad\quad \forall (x,a),\ (y,b) \in s : \\
\qquad\qquad (\ \exists (x,y) \in modelConstraints(parameterModel(at))\ ) \\
\qquad\qquad\qquad \Rightarrow\ (\ \exists (a,b) \in modelConstraints(m)\ )
\end{array}
$$

Le paramètre s représente l'ensemble de substitutions à réaliser. Cet opérateur tient compte de l'instanciation partielle ($firsts(s) \subseteq modelElements(parameterModel(at))$).

Algorithme

début
  /* $actualModel$ est le modèle actuel défini en section 3.1.2.2 */
  $actualModel = extract(m, seconds(s))$
  $result = apply(at, actualModel, s)$
fin

Exemple La figure 4.6 illustre l'instantiation du template *ObserverPattern* avec le modèle de base *AgencyClient*, permettant d'obtenir l'instance *ObservableAgency*.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{instantiate(ObserverPattern, AgencyClient, \{\ (Observer, Client), (update, change), (T, String),} \\
\multicolumn{3}{l}{\qquad\qquad (Subject, Agency), (obs, clients)\ \}\ ) :} \\
 & & \\
actualModel & = & extract(AgencyClient, \{\ Client, change, String, Agency, clients\ \}\ ) \\
 & = & (\ \{\ Client, change, String, Agency, clients\ \}, \\
 & & \{\ (change, Client), (change, String), (clients, Agency), (clients, Client)\ \}\ ) \\
 & & \\
result & = & apply(at, actualModel, \{\ (Observer, Client), (update, change), (T, String), \\
 & & (Subject, Agency), (obs, clients)\}\ ) \\
 & = & ObservableAgency
\end{array}
$$

Représentation graphique La représentation de l'instanciation est celle présentée en section 3.3. Elle est représentée par une relation de dépendance UML stéréotypée *instantiate* (cf. figure 4.6) entre un *template* et un *modèle*. Cette relation va du contexte vers le *template*. La substitution utilisée pour l'instanciation se situe entre les symboles < et >.

4.1.3.3 Promote

Objectif L'opérateur *promote construit un template aspectuel par sélection d'éléments du template comme nouveaux paramètres.*

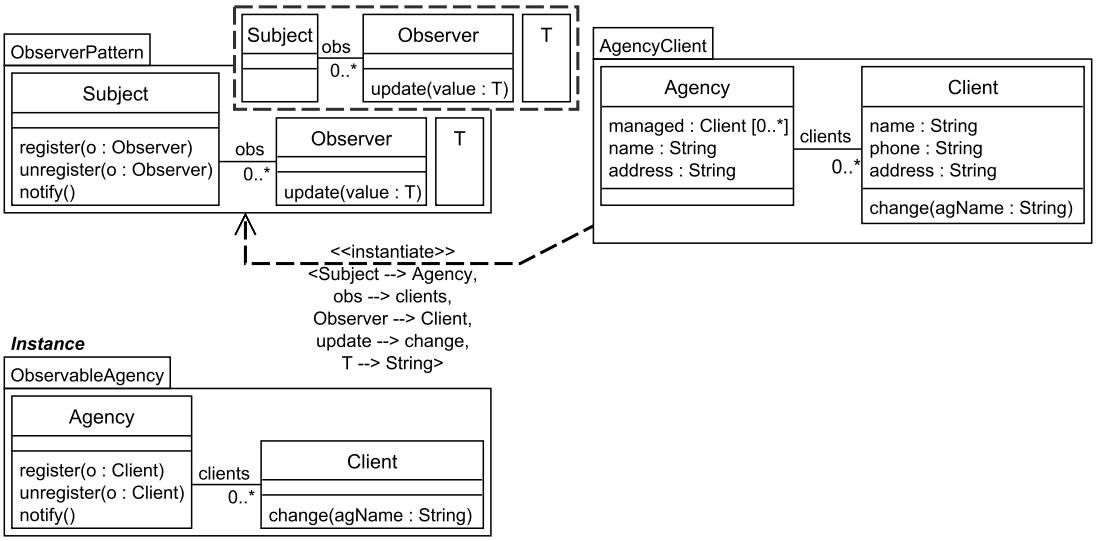


FIGURE 4.6 – Relation d'instanciation entre *ObserverPattern* et *AgencyClient*

Il assure la bonne formation du paramétrage en complétant l'ensemble des éléments du modèle paramètre (Carré et al. [25]). Cette complétion est réalisée via l'opérateur de fermeture (*closure*) présenté en section 4.1.2.2.

Formulation

promote(at :AT, e :ModelElement*) :AT
Avec : $e \subseteq (modelElements(core(at)) \backslash parameters(at))$

e correspond aux éléments de modèle à promouvoir comme paramètres.

Algorithme

début

/* $paramElem$ est l'ensemble des éléments paramètres non complétés */
$paramElem = e \cup parameters(at)$

/* $eClos$ est l'ensemble des éléments paramètres complétés */
$eClos = closure(core(at), paramElem)$

/* $piModel$ est le modèle paramètre bien formé */
$piModel = extract(core(at), eClos)$

/* $result$ est défini avec le core de at et $piModel$ */
$result = (core(at), piModel)$

fin

Exemple La promotion de l'opération *update*, dans le *template SubjectObserver*, est présentée dans la figure 4.7. Le type de donnée T, dont dépend *update*, est lui aussi promu afin d'avoir un modèle paramètre bien formé dans *ObserverPattern*.

$$
\begin{aligned}
&promote(GenericObserver, \{\ update\ \})\\
paramElem &= \{\ update\ \} \cup parameters(GenericObserver)\\
&= \{\ Subject, obs, Observer, update\ \}\\
\\
eClos &= closure(core(GenericObserver), paramElem)\\
&= \{\ Subject, obs, Observer, update, T\ \}\\
\\
piModel &= extract(core(at), eClos)\\
&= (\ \{\ Subject, obs, Observer, update, T\ \},\\
&\quad \{\ (obs, Subject), (obs, Observer), (update, Observer), (update, T)\ \}\ )\\
\\
result &= (core(GenericObserver), piModel) = ObserverPattern
\end{aligned}
$$

Représentation graphique La promotion est représentée par une relation de dépendance *UML* stéréotypée *promote* entre deux *templates*. Cette relation va du *template* résultant de la promotion (*result*, cf. figure 4.7), vers le *template* d'origine (*at*). Les éléments de l'ensemble utilisés pour la promotion sont définis au niveau de la relation de dépendance, entre les symboles < et >.

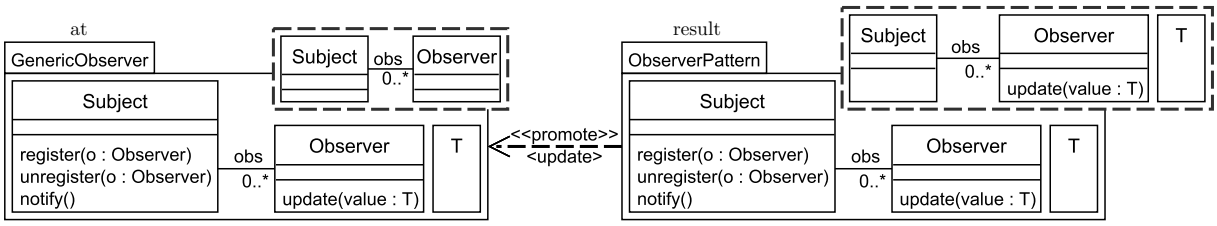


FIGURE 4.7 – Relation de promotion entre *GenericObserver* et *ObserverPattern*

4.1.3.4 Restrict

Objectif L'opérateur *restrict construit un template aspectuel en retirant le statut d'éléments paramètres à des éléments du template.*

Il assure la bonne formation du paramétrage en complétant l'ensemble des éléments de modèle sélectionnés (Carré et al. [25]). Cette complétion est réalisée via l'opérateur de fermeture (*dependents*) présenté en section 4.1.2.2.

Formulation

restrict(at :AT, e :ModelElement*) :AT Avec : $e \subseteq parameters(at)$

e correspond aux éléments à retirer du modèle paramètre.

Algorithme

Début

/* $paramModel$ est le modèle paramètre du *template* at */
$paramModel = parameterModel(at)$

/* $eDepend$ est l'ensemble des éléments paramètres complété */
$eDepend = dependents(paramModel, e)$

/* $paramElem$ est l'ensemble des éléments paramètres sans $eDepend$ */
$paramElem = (modelElements(paramModel) \backslash eDepend)$

/* $piModel$ est le modèle paramètre bien formé */
$piModel = extract(paramModel, paramElem)$

/* $result$ est défini avec le core de at et $piModel$ */
$result = (core(at), piModel)$

fin

Exemple La figure 4.8 illustre la restriction du type de données T dans le *template* *Observer-Pattern*. L'opération *update*, dépendant de T, est elle aussi restreinte, afin d'obtenir un modèle paramètre bien formé dans le *template* résultat *GenericObserver*.

$$
\begin{array}{lcl}
restrict(ObserverPattern, \{T\}) & & \\
paramModel & = & parameterModel(ObserverPattern) \\
 & = & \{\ Subject, obs, Observer, update, T\ \}, \\
 & & \{\ (obs, Subject), (obs, Observer), (update, Observer), (update, T)\ \}\ ) \\
 & & \\
eDepend & = & dependents(paramModel, \{\ T\ \}\ ) \\
 & = & \{\ update, T\ \} \\
 & & \\
paramElem & = & (modelElements(paramModel) \backslash eDepend) \\
 & = & \{\ Subject, obs, Observer\ \} \\
 & & \\
piModel & = & extract(paramModel, paramElem) \\
 & = & (\ \{\ Subject, obs, Observer\ \}, \{\ (obs, Subject), (obs, Observer)\ \}\ ) \\
 & & \\
result & = & (core(ObserverPattern), piModel) \\
 & = & GenericObserver
\end{array}
$$

Représentation graphique La restriction est représentée par une relation de dépendance UML stéréotypée *restrict* entre deux *templates*. Cette relation va du *template* résultat ($result$, cf. figure 4.8), vers le template d'origine (at). L'ensemble d'éléments utilisé est défini au niveau de la relation de dépendance, entre les symboles < et >.

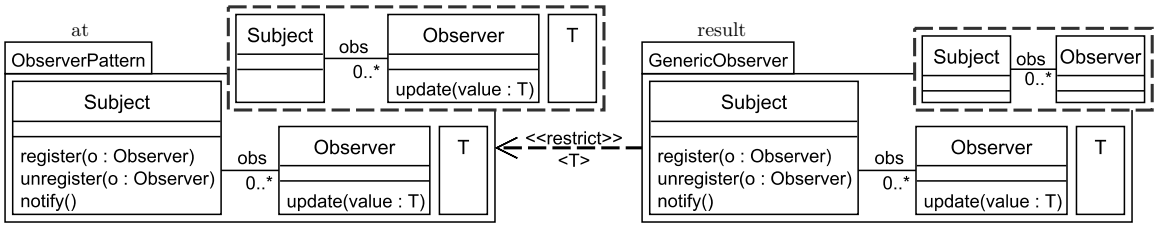


FIGURE 4.8 – Relation de restriction entre *GenericObserver* et *ObserverPattern*

4.1.3.5 FindCompatibleHierarchy

Objectif L'opérateur *findCompatibleHierarchy* retourne, parmi un ensemble de modèles, tous les super ou sous-modèles d'un modèle m compatibles avec l'application d'un *template at* sur m, pour une substitution donnée.

Cet opérateur possède la même contrainte que celle de l'opérateur *apply*, à savoir la validité de l'application de *at* sur m. Lorsque cette contrainte est respectée, les super et sous-modèles compatibles avec l'application de *at* sur m sont ceux sur lesquelles *at* est applicable pour la même substitution, soit *les super et sous-modèles qui incluent le modèle actuel.*

Formulation

findCompatibleHierarchy(at :AT, m :Model, s :(ModelElement* $\times$ *ModelElement)***,
***em : Model***) :Model**

Avec :
$$seconds(s) \subseteq modelElements(m)$$
$$firsts(s) \subseteq modelElements(parameterModel(at))$$
$$\forall (x,a),\ (y,b) \in s :$$
$$(\ \exists (x,y) \in modelConstraints(parameterModel(at))\ )$$
$$\Rightarrow\ (\ \exists (a,b) \in modelConstraints(m)\ )$$

Le paramètre s représente l'ensemble de substitutions de référence. em correspond à l'ensemble des modèles d'entrée.

Algorithme

début

```
/* actualModel est le modèle actuel défini en chapitre 3.5 */
actualModel = extract(m, seconds(s))

/* result inclut tous les sous et super modèles de m incluant */
/* actualModel */
result = {x ∈ em|included(actualModel, x)∧
                  (included(x, m) ∨ included(m, x))}
```

fin

Exemple En figure 4.9 à partir de l'application de *ObserverPattern* sur *CarAgencyClient* et de l'ensemble de modèles *CarModel, AgencyClient, CarAgencyClient et CarHiringSystem*, les

modèles compatibles trouvés sont *AgencyClient, CarAgencyClient et CarHiringSystem* (en gras sur la figure).

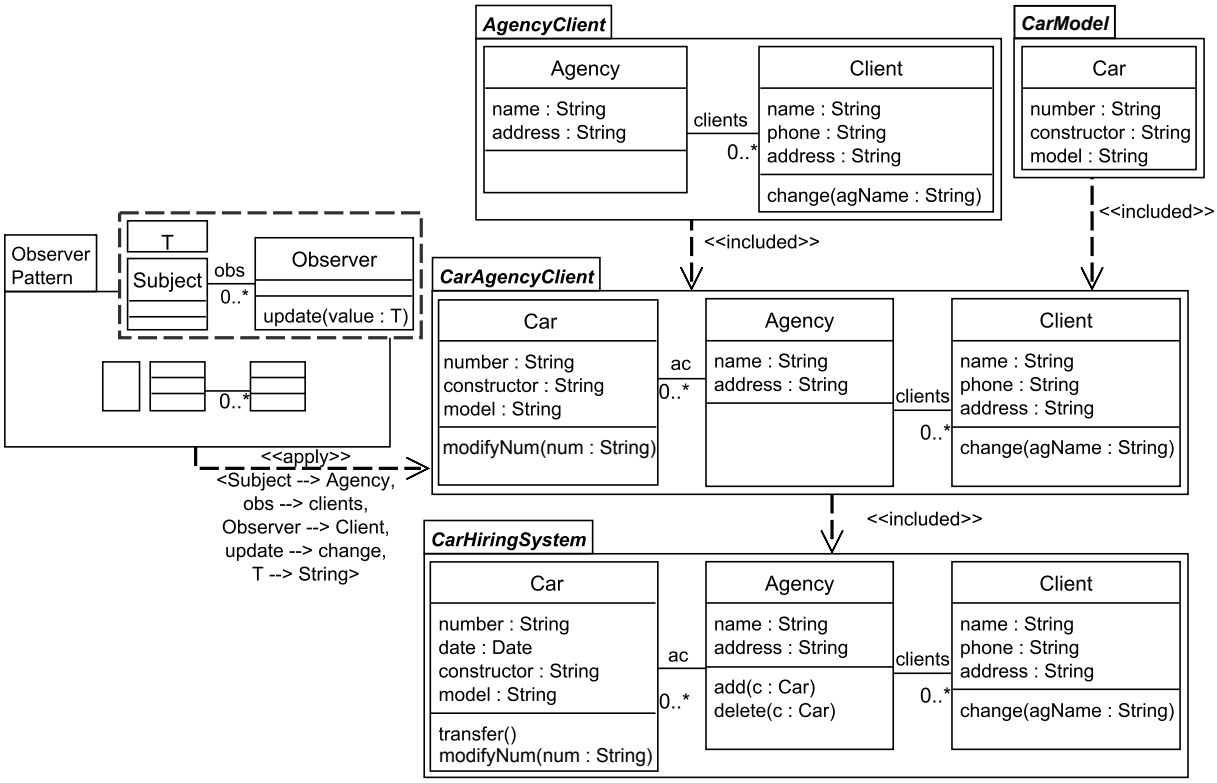


FIGURE 4.9 – Modèles compatibles avec l'application de $ObserverPattern$ sur $AgencyClient$

$$
\begin{aligned}
findCompatibleHierarchy(\quad & ObserverPattern, CarAgencyClient, \\
& \{\ (Subject, Agency), (obs, clients), (Observer, Client), \\
& (update, change), (T, String)\ \}, \\
& \{\ CarModel, AgencyClient, CarAgencyClient, CarHiringSystem\ \}\ )
\end{aligned}
$$

$$
\begin{aligned}
actualModel \quad = \quad & extract(m, seconds(s)) \\
= \quad & (\ \{\ Agency, clients, Client, change, String\ \}, \\
& \{\ (clients, Agency), (clients, Client), (change, Client), \\
& (change, String)\ \}\ )
\end{aligned}
$$

$$
em \quad = \quad \{\ CarModel, AgencyClient, CarAgencyClient, CarHiringSystem\ \}
$$

$$
\begin{aligned}
result \quad = \quad & \{\ x \in em |\ included(actualModel, x) \wedge (included(x, CarAgencyClient) \\
& \vee included(CarAgencyClient, x))\ \} \\
= \quad & \{\ AgencyClient, CarAgencyClient, CarHiringSystem\ \}
\end{aligned}
$$

4.1.3.6 SubsInference

Objectif L'opérateur *subsInference* retourne toutes les substitutions *totales* permettant d'appliquer un template sur un modèle.

En reprenant le template *ObserverPattern* et le modèle *CarHiringSystem* de la figure 4.9, l'ensemble des substitutions possibles est : *{ {(Subject, Agency), (obs, clients), (Observer, Client), (update, change), (T, String)}, {(Subject, Agency), (obs, ac), (Observer, Car), (update, modifyNum), (T, String)} }*

Formulation

$$\boxed{\textbf{\textit{subsInference(at :AT, m :Model) :}}2^{(ModelElement \times ModelELement)^*}}$$

Algorithme L'opérateur *subsInference* correspond à l'inférence de substitutions détaillée dans Vanwormhoudt et al. [103] [1]. L'algorithme utilisé pour cette inférence est celui présenté dans Ullmann [101], qui réalise la détection d'isomorphismes de sous-graphes.

4.1.3.7 GetBoundSubstitutions

Objectif L'opérateur *getBoundSubstitutions* retourne l'ensemble des substitutions totales pour lesquelles un modèle peut être le résultat de l'application d'un template.

Ainsi, à la différence de l'opérateur *subsInference* précédent, les substitutions renvoyées sont celles permettant d'identifier la présence du *template* dans le modèle. Pour cela, nous utilisons le test d'inclusion d'instance pour chaque substitution inférée.

Formulation

$$\boxed{\textbf{\textit{getBoundSubstitutions(at :AT, m :Model) :}}2^{(ModelElement \times ModelELement)^*}}$$

Algorithme

début
  $subsInfer = subsInference(at, m)$

  /* *result* est l'ensemble des substitutions pour lesquelles m est un *bound model* de *at* */
  $result = \{s \in subsInfer | included(m, instantiate(at, m, s))\}$
fin

Exemple Avec le *template ObserverPattern* et le modèle *CarEquipManagement* de la figure 4.10, les substitutions possibles sont les suivantes : *{ {(Subject, Agency), (obs, clients), (Observer, Client), (update, change), (T, String)}, {(Subject, Agency), (obs, ac), (Observer, Car), (update, modifyNum), (T, String)} }* .

1. Les auteurs y considèrent aussi les substitutions partielles.

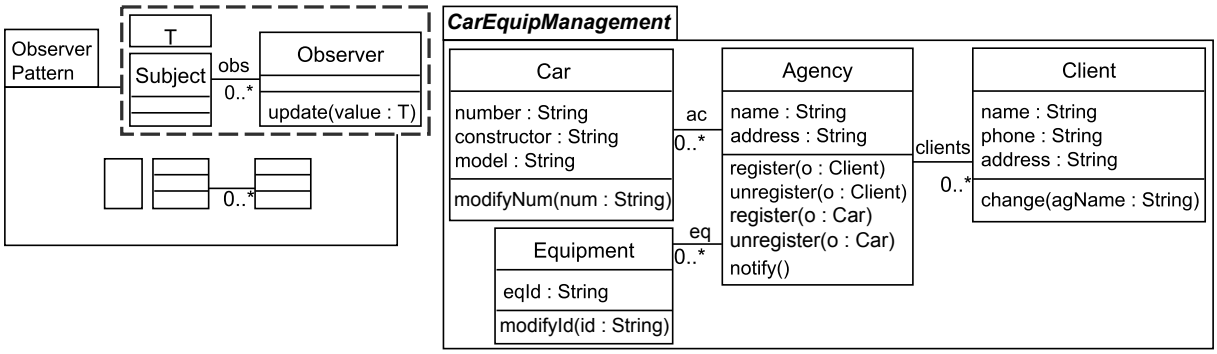


FIGURE 4.10 – $CarEquipManagement$ inclut des instances de $ObserverPattern$

$getBoundSubstitutions(ObserverPattern, CarEquipManagement)$:

$$
\begin{aligned}
subsInfer \quad = \quad & \{\ \{\ (Subject, Agency), (obs, clients), (Observer, Client), (update, change),\\
& (T, String)\ \},\\
& \{(Subject, Agency), (obs, ac), (Observer, Car), (update, modifyNum), (T, String)\},\\
& \{(Subject, Agency), (obs, equip), (Observer, Equipment), (update, changeId),\\
& (T, String)\}\ \}
\end{aligned}
$$

$$
\begin{aligned}
result \quad = \quad & \{\ \{\ (Subject, Agency), (obs, clients), (Observer, Client), (update, change),\\
& (T, String)\},\\
& \{(Subject, Agency), (obs, ac), (Observer, Car), (update, modifyNum), (T, String)\}\ \}
\end{aligned}
$$

4.1.3.8 Unbind

Objectif L'opérateur *unbind* construit un sous-modèle en retirant l'instance éventuellement présente dans un modèle de base.

Cet opérateur met en œuvre la suppression de templates dans un modèle (section 3.4.2) issus soit de la détection grâce à l'opérateur précédent ($getBoundSubstitutions$), soit manuellement, le concepteur ayant détecté la présence d'un template.

Formulation

unbind(at :AT, m :Model, s :(ModelElement* $\times$ *ModelElement)*) :Model
Avec : $included(m, instantiate(at, m, s))$

Le paramètre s représente l'ensemble de substitutions de référence.
Algorithme

début
  `/* Recherche de` s `dans les ensembles retournés par */`
  `/*` $getBoundSubstitutions$ `*/`
  $instanceModel = instantiate(at, m, s)$

/* $instanceModel$ est le modèle d'instance de at */
$instanceModel = instantiate(at, m, s)$

/* $closSpecElem$ est l'ensemble d'éléments clos dans l'ensemble des éléments du modèle spécifique $specificElements$ de at */
$specificElements =$
$modelElements(core(at)) \backslash modelElements(parameterModel(at))$
$closSpecElem = closure(instanceModel, specificElements)$

/* $closSpecModelConstraints$ est l'ensemble des contraintes du modèle spécifique clos $closSpecModel$ dans l'instance */
$closSpecModel = extract(instanceModel, closSpecElem)$
$closSpecModelConstraints = modelConstraints(closSpecModel)$

/* $actuElem$ est l'ensemble des éléments du modèle actuel */
$actualModel = extract(m, seconds(s))$
$actuElem = modelElements(actualModel)$

/* $closSpecElemNotActu$ est l'ensemble des éléments non-actuels */
/* à retirer de m */
$closSpecElemNotActu = (closSpecElem \backslash actuElem)$

/* $result$ est le sous-modèle extrait de m */
$resultRela = modelConstraints(m) \backslash closSpecModelConstraints$
$resultElem = modelElements(m) \backslash closSpecElemNotActu$
$result = (resultElem, resultRela)$

fin

Exemple La figure 4.11 illustre le retrait du template *ObserverPattern* du modèle *ClientManagement*.

$unbind(ObserverPattern, ClientManagement, s)$

$included(\ instantiate(ObserverPattern, ClientManagement, s), ClientManagement\ ) = true$

$$
\begin{aligned}
instanceModel &= instantiate(ObserverPattern, ClientManagement, s) \\
&= (\ \{Agency, register, unregister, notify, clients, Client, change, \\
&\quad String\} \\
&\quad \{(register, Agency), (register, Client), (unregister, Agency), \\
&\quad (unregister, Client), (notify, Agency), (clients, Agency), \\
&\quad (clients, Client), (change, Client), (change, String)\}\ ) \\
\\
specificElements &= modelElements(core(ObserverPattern)) \setminus \\
&\quad modelElements(\ parameterModel(ObserverPattern)\ ) \\
&= \{register, unregister, notify\}
\end{aligned}
$$

$$
\begin{aligned}
closSpecElem &= closure(instanceModel, specificElements) \\
&= \{Agency, register, unregister, notify, Client\} \\
\\
closSpecModel &= extract(instanceModel, closSpecElem) \\
&= (\ \{Agency, register, unregister, notify, Client\}, \\
&\quad \{(register, Agency), (register, Client), (unregister, Agency), \\
&\quad (unregister, Client), (notify, Agency)\}\ ) \\
\\
closSpecModelConstraints &= modelConstraints(closSpecModel) \\
&= \{(register, Agency), (register, Client), (unregister, Agency), \\
&\quad (unregister, Client), (notify, Agency)\}) \\
\\
actualModel &= extract(ClientManagement, seconds(s)) \\
&= (\ \{Agency, clients, Client, change, String\}, \\
&\quad \{(clients, Agency), (clients, Client), (change, Client), \\
&\quad (change, String)\}\ ) \\
\\
actuElem &= modelElements(actualModel) \\
&= \{Agency, clients, Client, change, String\} \\
\\
closSpecElemNotActu &= (closSpecElem \backslash actuElem) \\
&= \{register, unregister, notify\} \\
\\
resultRela &= modelConstraints(ClientManagement) \setminus \\
&\quad closSpecModelConstraints \\
&= \{(number, Car), (number, String), (constructor, Car), \\
&\quad (constructor, String), (model, Car), (model, String), \\
&\quad (ac, Car), (ac, Agency), (name, Agency), (name, String), \\
&\quad (address, Agency), (address, String), (clients, Agency), \\
&\quad (clients, Client), (name, Client), (name, String), \\
&\quad (phone, Client), (phone, String), (address, Client), \\
&\quad (address, String), (change, Client)\} \\
\\
\text{resultElem} &= modelElements(ClientManagement) \setminus \\
&\quad closSpecElemNotActu \\
&= \{Car, number, constructor, model, ac, Agency, name, \\
&\quad address, clients, Client, name, phone, address, change, String\}
\end{aligned}
$$

$$
\begin{aligned}
\text{result} &= (resultElem, resultRela) \\
&= (\ \{Car, number, constructor, model, ac, Agency, name, \\
&\quad address, clients, Client, name, phone, address, change, String\}, \\
&\quad \{(number, Car), (number, String), (constructor, Car), \\
&\quad (constructor, String), (model, Car), (model, String), \\
&\quad (ac, Car), (ac, Agency), (name, Agency), (name, String), \\
&\quad (address, Agency), (address, String), (clients, Agency), \\
&\quad (clients, Client), (name, Client), (name, String), \\
&\quad (phone, Client), (phone, String), (address, Client), \\
&\quad (address, String), (change, Client)\}) \\
&= CarAgencyClient
\end{aligned}
$$

Représentation graphique L'extraction à partir d'un ensemble de substitutions est représentée par une relation de dépendance UML stéréotypée *unbind* allant d'un *template at* (cf. figure 4.11) vers un *modèle m* qui est celui à partir duquel doit être faite l'extraction. Les substitutions utilisées pour cette extraction se situent entre les symboles < et >.

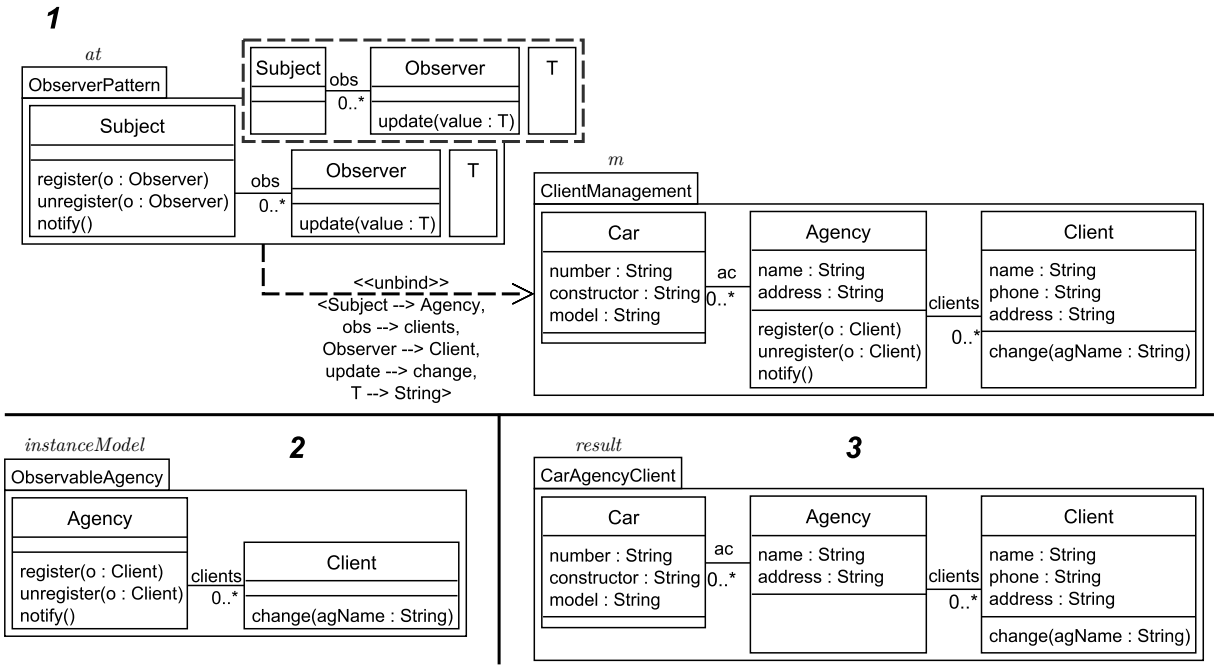


FIGURE 4.11 – Relation *unbind* entre *ObserverPattern* et *ClientManagement*

4.1.3.9 Synthèse

TABLE 4.12 Opérateurs de *templates*

Opérateurs	***Objectifs***	***Signatures***	***Opérateurs utilisés***
apply	Construit un nouveau modèle en appliquant un *template* sur un modèle de base	*apply(AT, Model, (ModelElement × ModelElement)*) :Model*	instantiate et merge
instantiate	Créer une instance de template à partir d'un modèle de base	*instantiate(AT, Model, (ModelElement × ModelElement)*) :Model*	extract
promote	Construire un *template* aspectuel à partir d'un autre par sélection de nouveaux paramètres	*promote(AT, ModelElement*) :AT*	closure et extract
restrict	Construire un template aspectuel à partir d'un autre en retirant le statut de paramètre à des éléments	*restrict(AT, ModelElement*) :AT*	dependents et extract
findCompatibleHierarchy	Retourne tous les super ou sous-modèles d'un modèle m compatibles avec l'application d'un *template at* sur m, pour une substitution donnée	*findCompatibleHierarchy(AT, Model, (ModelElement × ModelElement)*, Model*) :Model**	included et extract
subsInference	Retourne toutes les substitutions *totales* permettant d'appliquer un template sur un modèle	*subsInference(AT, Model)*: $2^{(ModelElement \times ModelElement)^*}$	
getBoundSubstitutions	Retourne l'ensemble des substitutions totales pour lesquelles un modèle applique un template	*getBoundSubstitutions(AT, Model)* : $2^{(ModelElement \times ModelElement)^*}$	subsInference, included et instantiate
unbind	Construit un sous-modèle en retirant un template d'un modèle	*unbind(AT, Model, (ModelElement × ModelElement)*) :Model*	instantiate, closure et extract

4.2 Un atelier de modélisation à base d'*aspectual templates*

Afin de mettre *en application l'IDMBT (cf. section 3.2)*, nous avons conçu un atelier de modélisation dédié [2] qui *assiste les modeleurs en exposant des fonctionnalités de conception et de réutilisation d'aspectual templates*. Nous présentons cet atelier en section 4.2.1. Le profil *UML* défini pour les *aspectual templates* est présenté en section 4.2.2, puis l'architecture à base de *plugins* de l'atelier en section 4.2.3.

4.2.1 Présentation générale

Du point de vue utilisateur, l'atelier (cf. figure 4.12) se compose :

- d'un éditeur de modèles et d'*aspectual templates* (les deux parties supérieures centrales),
- d'une vue graphique du modèle (partie droite). Cette vue peut aider un concepteur à trouver l'*aspectual template* répondant à ses besoins ou à comprendre des assemblages de *templates* et leurs résultats,
- d'une vue permettant de visualiser les relations d'inclusions dans l'espace des modèles (partie basse).

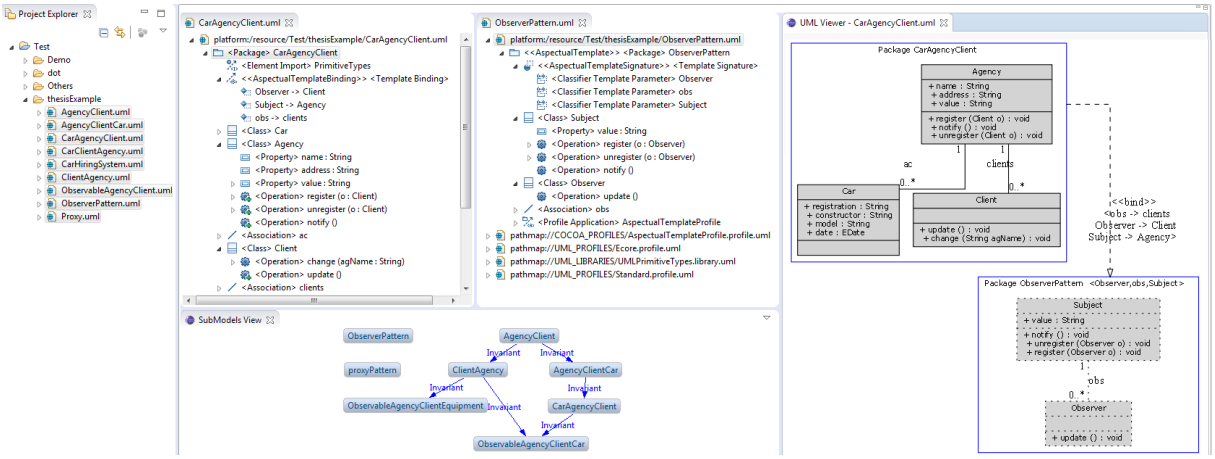


FIGURE 4.12 – Atelier de modélisation à base d'*aspectual templates*

L'éditeur permet de construire les *aspectual templates*. Il spécialise *UML* via un profil dédié, détaillé en section 4.2.2. Pour cette spécialisation, l'éditeur expose un ensemble de fonctionnalités pour *(1)* appliquer les stéréotypes et *(2)* valider les contraintes concernant la bonne formation du modèle paramètre et la cohérence de leurs applications.
Outre l'édition d'*aspectual templates* via le profil, cet éditeur met également des fonctionnalités à disposition (cf. figure 4.13), présentées ci-après.

Paramétrisation de modèles Deux fonctionnalités *"Restrict (ou) Promote Element(s) as Parameter(s)"* permettent de faire varier le paramétrage d'un *aspectual template* à partir de la sélection d'éléments dans le *core* de celui-ci. Elles correspondent respectivement aux opérateurs *promote* (ajout d'éléments au modèle paramètre - cf. section 4.1.3.3) et *restrict* (retrait d'éléments - cf. section 4.1.3.4).
Une autre fonctionnalité, utilisant l'opérateur *closure*, permet quant à elle de compléter le

2. http://www.cristal.univ-lille.fr//caramel/MBE_Template/

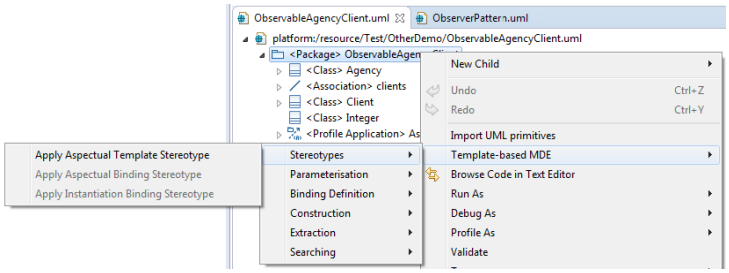


FIGURE 4.13 – Fonctionnalités exposées par l'éditeur d'*aspectual templates*

modèle paramètre afin de respecter les contraintes de bonne formation d'un *aspectual template*.

Définition de bindings A l'aide d'une première fonctionnalité *"Complete Binding (Partially (ou) Totally)"*, utilisant l'opérateur *closure* (cf. section 4.1.2.2), il est possible de compléter les éléments *formals* (*i.e.* ceux provenant du modèle paramètre du *template*) de l'ensemble de substitutions (le *binding*) afin de substituer tous les éléments du modèle paramètre.
Une seconde fonctionnalité *"Infer Binding Substitutions"*, détaillée dans Vanwormhoudt et al. [103] et correspondant à l'opérateur *SubsInference* (cf. section 4.1.3.6), permet d'inférer toutes les substitutions permettant d'appliquer un *aspectual template* sur un modèle.
Ceci est illustré dans l'exemple de la figure 4.14. Pour le *binding* créé dans *CarAgencyClient*, l'attribut *value* n'est pas substitué (valeur "undefined"). L'ensemble des *bindings* possibles (partie inférieure gauche) varient sur la substitution de cet attribut. *value* peut être substitué avec *(a)* tout attribut de type String et *(b)* tout attribut qui appartient à la classe substituée avec *Subject (ObserverPattern)*. *value* peut donc être substitué avec *name*, *address* ou *capacity* qui appartiennent à *Agency*.
Une autre fonctionnalité *"Search a Template in a Model"* utilise l'opérateur *getBoundSubstitutions* (cf. section 4.1.3.7). À la différence de l'inférence de substitutions précédente, les *bindings* retournés sont ceux pour lesquelles les fonctionnalités du *template* sont présentes au sein du modèle.
Ceci est illustré en figure 4.15 : *CarHiringSystem* inclut l'instance de *ObserverPattern* selon la substitution présentée en partie gauche.

Construction de modèles Via la fonctionnalité *"Process Aspectual Binding"* utilisant l'opérateur *apply* (cf. section 4.1.3.1), il est possible d'effectuer une application aspectuelle selon un *binding* stéréotypé *«AspectualTemplateBinding»* (cf. sous-section 4.2.2).
Les instanciations de *templates* sont réalisées avec la fonctionnalité *"Process Instanciation Binding"* utilisant l'opérateur *instantiate* (cf. section 4.1.3.2), au-travers d'un *binding* stéréotypé *«InstantiationBinding»*.

Extraction de modèles Une autre capacité offerte par l'atelier, *"Extract a Model From Another"*, est l'extraction d'un sous-modèle à partir de la sélection d'éléments dans un modèle (utilisation de l'opérateur *extract*, cf. section 4.1.2.1).
Par ailleurs, avec *"Unbind an Aspectual Template in a Model"*, il est possible de retirer un *aspectual template* d'un modèle, *i.e.* obtenir un sous-modèle sans les fonctionnalités du *template* (cf. opérateur *unbind*, en section 4.1.3.8).

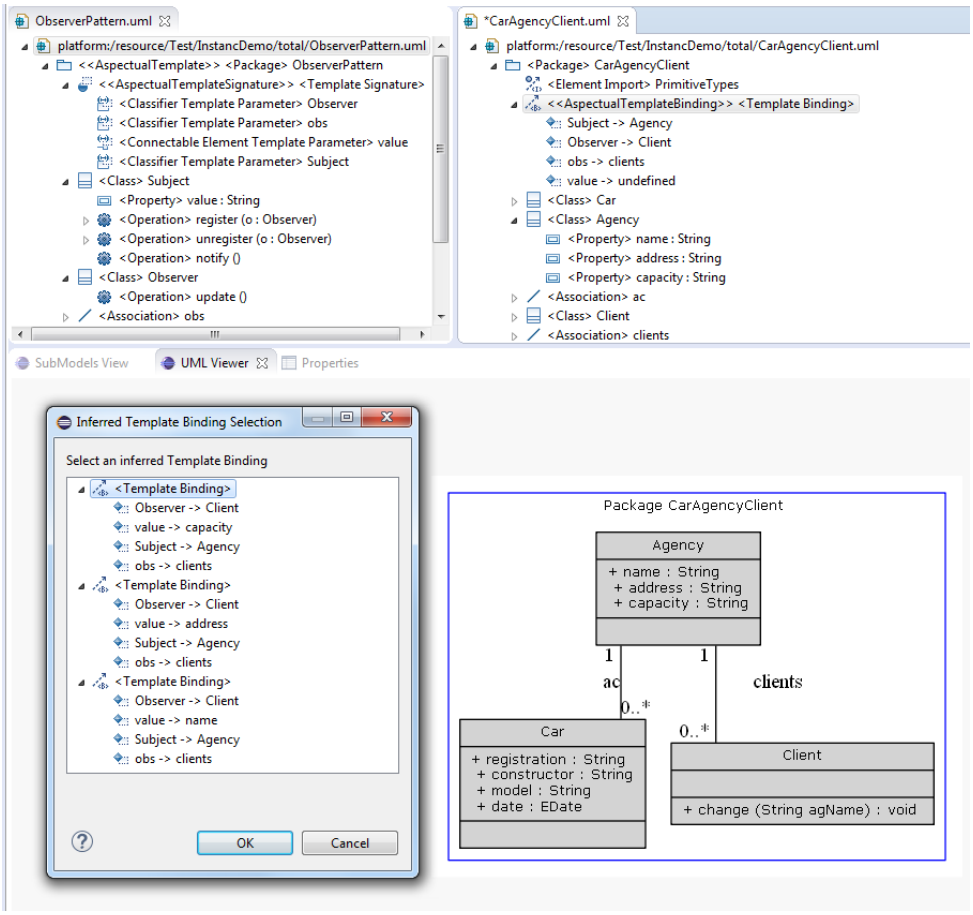


FIGURE 4.14 – Inférence de substitutions

Recherche de modèles À l'aide de la fonctionnalité *"Search Sub or Supermodels Compatible With a Binding"*, il est possible de rechercher, dans le répertoire contenant les modèles du projet, les sous ou surmodèles compatibles avec le *binding* sélectionné. L'opérateur utilisé ici est *findCompatibleHierarchy* (cf. section 4.1.3.5).
En figure 4.16, les sous et surmodèles de *CarAgencyClient* compatibles avec l'application de *ObserverPattern* sur ce modèle sont *CarHiringSystem* et *AgencyClient* (affichés en partie basse).

4.2.2 Extension du métamodèle UML

La prise en charge des aspectual templates nécessite d'étendre le métamodèle d'UML avec de nouvelles contraintes pour assurer la bonne formation du paramétrage et la cohérence du binding. Cette extension est obtenue au moyen d'un profil *UML*, conforme au plugin *UML* d'Eclipse. Un tel profil, non-invasif vis-à-vis du métamodèle par définition, permet d'utiliser les *aspectual templates* sans dépendre d'un outil en particulier. En effet, les stéréotypes peuvent être appliqués sur n'importe quel modèle *UML* utilisant soit l'interface du *plugin UML*, soit provenant d'un outil *CASE*[3] conforme à *UML*.
Ces stéréotypes, utilisant les contraintes *OCL* définies dans Vanwormhoudt et al. [103], sont les suivants :

3. *Computer-Aided Software Engineering* : Génie Logiciel Assisté par Ordinateur

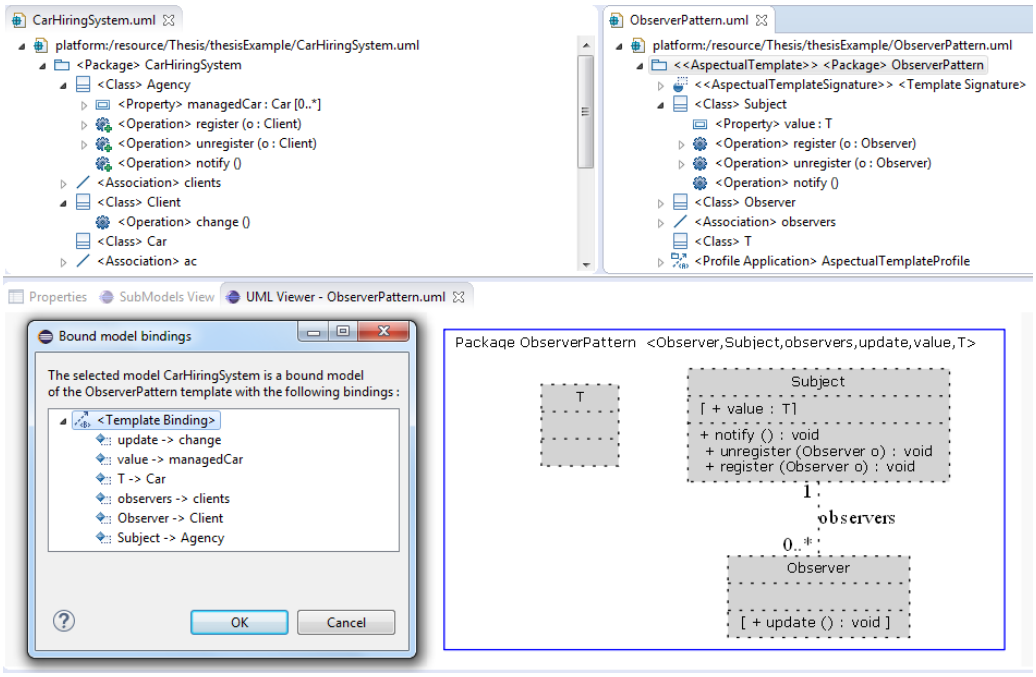


FIGURE 4.15 – *CarHiringSystem* respecte la structure de *ObserverPattern*

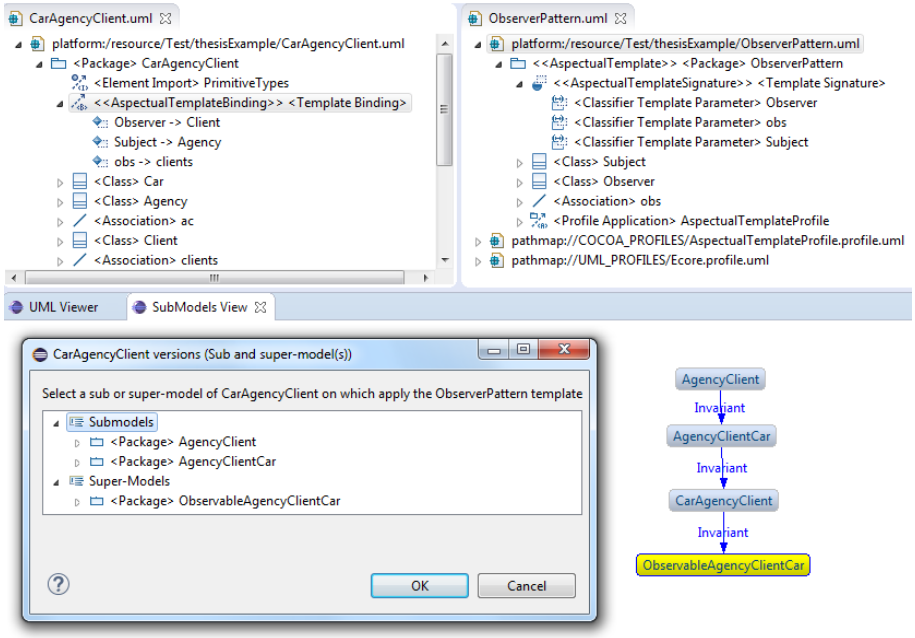


FIGURE 4.16 – Sous et surmodèles compatibles avec l'application de *ObserverPattern* sur *CarAgencyClient*

- *«AspectualTemplate»*, pour interpréter les *templates* de *packages* en tant que *aspectual templates*,
- *«AspectualTemplateSignature»*, pour spécifier les signatures (paramétrage) de *templates* en tant que signatures d'*aspectual templates*. Ce stéréotype est applicable sur des *packages* stéréotypés *«AspectualTemplate»*,

- *«AspectualTemplateBinding»*, pour l'application aspectuelle. Ce stéréotype peut être appliqué sur tout *binding* entre un modèle et un *package* stéréotypé *«AspectualTemplate»*,
- *«InstantiationBinding»* qui étend le *binding* pour l'instanciation. Il est possible d'appliquer ce stéréotype sur tout *binding* entre un modèle et un *template*.

La figure 4.17 présente les stéréotypes en utilisant la notation standard pour les profils (OMG [78]), basée sur l'association *extension* entre la définition d'un stéréotype et la méta-classe étendue.

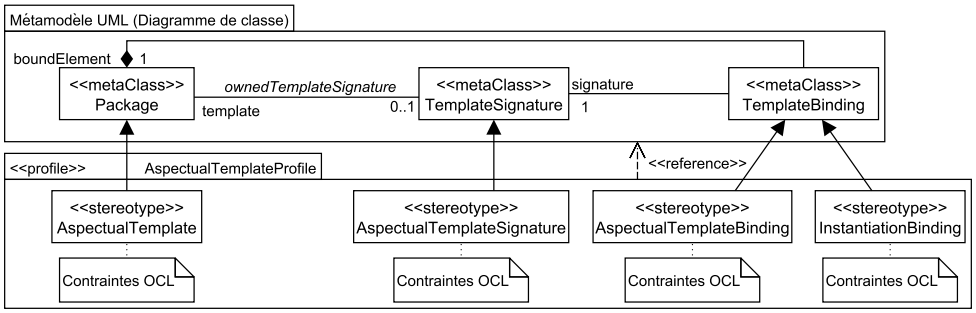


FIGURE 4.17 – Profil *UML*
utilisé par l'atelier

4.2.2.1 Les contraintes *OCL* du profil

Les contraintes utilisées par le profil *UML* sont appliquées aux métaclasses *"TemplateSignature"* et *"TemplateBinding"*. Les contraintes associées à la première métaclasse vérifient la bonne formation du modèle paramètre. Celles portant sur *"TemplateBinding"* sont dédiées à la conformité structurelle entre le modèle paramètre et le modèle actuel.
L'exemple de la figure 4.18 illustre une des contraintes portant sur la vérification de la bonne formation du modèle paramètre d'un *aspectual template*. La signature d'un *aspectual template* impose que l'ensemble des éléments paramètres forme un modèle paramètre bien formé. La contrainte suivante concerne les classes, leurs attributs et opérations. Si l'un de ces deux derniers types d'éléments est paramètre, sa classe doit aussi l'être. La figure 4.18a donne un contre-exemple de ceci, avec l'opération *value()* mis en paramètre sans la classe *Element*.

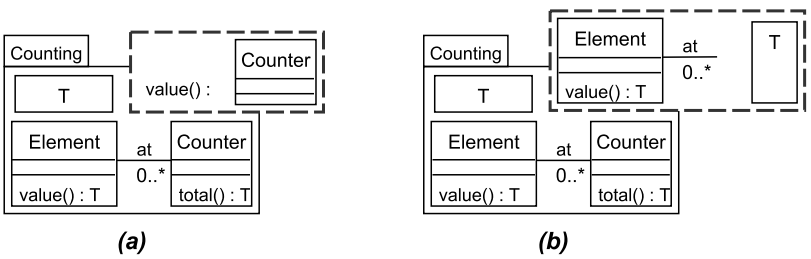


FIGURE 4.18 – Exemples d'*aspectual templates* mal-formés

– *[1] Les classes incluant des attributs ou opérations paramètres doivent aussi être des paramètres :*
context TemplateSignature **inv :**
self.ownedParameter->forAll(
  param : uml : :TemplateParameter |
  let pe : uml : :ParameterableElement = param.parameteredElement **in**
  pe.**oclIsKindOf**(uml : :Feature) **implies**
  let ownerClass : uml : :Class
  = pe.**oclAsType**(uml : :Feature).owner.**oclAsType**(uml : :Class) **in**
  ownerClass.isTemplateParameter ()
)

Les autres contraintes portant sur les applications totales sont présentées en annexe A. Quant aux instanciations totale et partielle, elles utilisent les mêmes contraintes portant sur la conformité structurelle (contraintes 5 à 8 , en annexe). Les contraintes concernant les applications aspectuelles partielles sont détaillées dans Vanwormhoudt et al. [103].

4.2.3 Architecture

L'atelier a été développé sur le principe d'architecture à base de *plugins* d'*Eclipse*. Les *plugins* officiels *EMF, UML et OCL* sont utilisés et de nouveaux *plugins* ont été implémentés. Chacun d'eux contribuent à *Eclipse*, en utilisant les points d'extensions de cette plateforme. Ces *plugins* et leurs relations de dépendances sont montrés en figure 4.19. Il s'agit de :

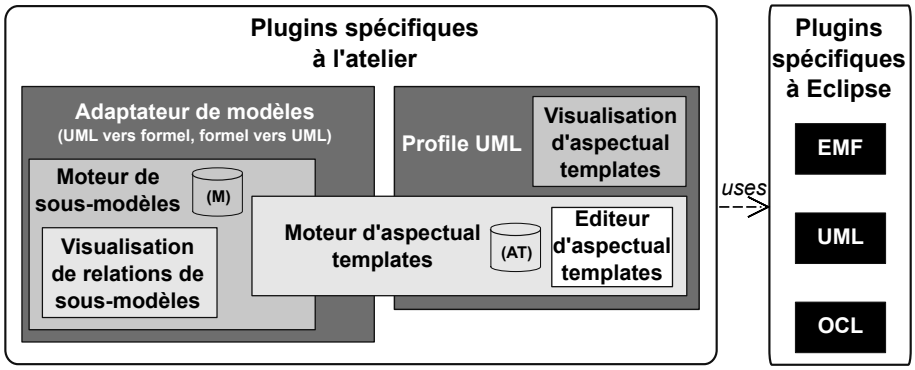


FIGURE 4.19 – Architecture de l'atelier de modélisation

- Un adaptateur de modèles : permettant l'échange de modèles entre les deux moteurs cités plus haut, celui-ci est présenté en sous-section 4.2.3.1.
- Le profil *UML* : il étend le *plugin UML d'Eclipse* avec les contraintes *OCL* et les stéréotypes présentés plus haut,
- Le moteur de sous-modèles : utilisé par le moteur d'*aspectual templates*, il permet de déterminer les relations d'inclusions entre des modèles et d'effectuer des opérations sur ceux-ci (extraction, fermeture d'un ensemble d'éléments de modèles),
- L'outil de visualisation d'*aspectual templates* : il étend *Eclipse* avec une nouvelle vue permettant d'avoir une représentation graphique d'un modèle UML ou d'un modèle UML étendu avec le profil,
- L'outil de visualisation des relations d'inclusion entre modèles : *Eclipse* est étendu avec cette nouvelle vue qui permet d'avoir une représentation graphique des relations de sous-

modèles. Cet outil est utilisé dans le cadre de la recherche de modèles, notamment avec la fonctionnalité de recherche de modèles sur lesquels un *template* est applicable (cf. figure 4.16),

- Le moteur d'*aspectual templates* : implémente les fonctionnalités liées aux *templates*, *i.e.* application des stéréotypes, paramétrisation, inférence et complétion de *bindings* et extraction des constituants d'un *template*,
- L'éditeur d'*aspectual templates* : l'éditeur du *plugin UML* est étendu pour cela, ajoutant des fonctionnalités qui permettent :
  . d'importer ou de spécifier de nouveaux *aspectual templates* et leurs applications, en accédant aux fonctionnalités implémentées dans le moteur de *templates*,
  . de vérifier les contraintes portant sur le modèle paramètre des *aspectual templates* et leurs applications. Pour cela, l'éditeur accède au profil *UML*.

Le bénéfice d'une telle architecture est *la modularité*. En effet, certains *plugins* sont utilisables indépendamment les uns des autres. Cela permet de les intégrer à d'autres outils compatibles avec *EMF* et *UML*, tels des moteurs de construction et de transformation de modèles, des vérificateurs de modèles *UML* ou encore des éditeurs de modèles. Les *plugins* pouvant être utilisés indépendamment des autres sont :

- le moteur de sous-modèles et l'adaptateur de modèles,
- l'outil de visualisation des relations d'inclusion entre modèles, avec le moteur de sous-modèles et l'adaptateur de modèles,
- l'outil de visualisation d'*aspectual templates* et le profile *UML*.

4.2.3.1 Moteur d'*aspectual templates*

Le moteur d'*aspectual templates* est constitué de quatre classes de *helpers*, présentées en figure 4.20. Ces classes permettent de séparer les opérations sur la structure même du *template* (*AspectualTemplateHelper*) de celles sur la signature de celui-ci (*AspectualTemplateSignatureHelper*)

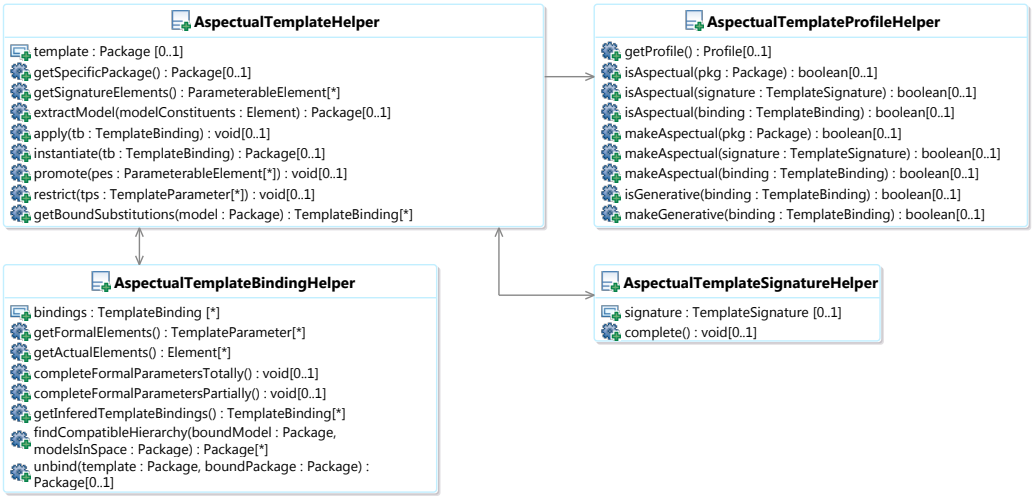


FIGURE 4.20 – Architecture du moteur d'*aspectual templates*

et sur les *bindings* (*AspectualTemplateBindingHelper*). Par exemple, l'opération *"promote"* (cf. section 4.1.3.3), définie dans *AspectualTemplateHelper*, utilise *"dependents"* (cf. section 4.1.2.2) de *AspectualTemplateSignatureHelper*. Ces classes sont détaillées ci-après :

- *AspectualTemplateProfileHelper* : Offre des opérations pour la définition du profil [4].
  . *Application des stéréotypes* : L'application des stéréotypes est effectuée sur les éléments cités précédemment via les opérations *"makeAspectual"* et *"makeGenerative"*.
  . *Test de présence des stéréotypes* : Cette opération teste si les stéréotypes associés au package et à la signature sont cohérents par rapport au binding sélectionné (AspectualTemplateBinding ou InstantiateTemplateBinding).
  Cette opération est notamment utilisée par certaines des fonctionnalités implémentées dans la classe *AspectualTemplateHelper*, présentée ci-après.
- *AspectualTemplateHelper* :
  . *Accès aux constituants d'un aspectual template* : La classe permet d'obtenir les constituants d'un *aspectual template*, à savoir le modèle paramètre et le modèle spécifique ainsi que les éléments de ces derniers (*"getSpecificTemplateElements"*, ...). Ces constituants sont utilisés par les fonctionnalités suivantes.
  . *Opérations sur les aspectual templates* : L'éditeur accède à celles-ci, *e.g.* pour appliquer un *aspectual template* (*"apply(binding : TemplateBinding)"*).
- *AspectualTemplateSignatureHelper* : Offre des opérations pour la définition de la signature d'un *template* (*i.e.* le modèle paramètre). Un exemple d'opération est la complétion de la signature avec *"closure"* (cf. section 4.1.2.2), afin d'obtenir un modèle paramètre bien formé.
  Cette complétion s'effectuant avec des éléments du modèle spécifique, la classe *AspectualTemplateHelper* est utilisée (opération *"getSpecificPackage"*).
- *AspectualTemplateBindingHelper* :
  . *Complétion (partielle et totale) d'un binding* : Les opérations permettant cette complétion (*"completeFormalParametersTotally"* et *"completeFormalParametersPartially"*) font appel à l'opération *"getSignatureParameterables"* de la classe *AspectualTemplateHelperUtil*, puisque le *binding* est complété avec des éléments du modèle paramètre.
  . *Inférence de bindings* : L'opération *"getInferredTemplateBindings"* retourne les *bindings* permettant d'appliquer un *aspectual template* sur un contexte (mise en œuvre de l'opérateur "subsInference", cf. section 4.1.3.6).

Connexion et échange de modèles avec le moteur de sous-modèles Le moteur de *templates* fait appel aux fonctionnalités du moteur de sous-modèles. Celui-ci a été implémenté par l'équipe et intégré à *Eclipse* et son *framework* de modélisation (*EMF*). Ce moteur fournit un ensemble de fonctionnalités permettant :

- de déterminer comment les modèles sont reliés entre eux par des relations d'inclusion,
- de hiérarchiser les modèles selon ces relations,
- de calculer la fermeture (*closure*) d'un ensemble d'éléments de modèle dans un modèle,
- de compléter un ensemble d'éléments de modèle avec ceux qui en dépendent (*dependents*) dans un modèle,
- d'extraire un sous-modèle à partir d'un modèle, via l'opérateur *extract* présenté dans la section précédente.

Ces fonctionnalités sont offertes par le moteur aux autres outils inclus dans l'environnement *Eclipse*, tel le moteur de *templates*. Elles sont valables pour n'importe quel modèle, moyennant l'implémentation d'un adaptateur afin de convertir le modèle au format utilisé par le moteur de sous-modèle.

4. Contrairement aux opérateurs, ces opérations sont non-fonctionnelles : elles modifient le modèle d'entrée.

Afin que le moteur d'*aspectual templates* puisse échanger des modèles avec le moteur de sous-modèles, deux conversions sont effectuées par un *adaptateur de modèles* (cf. figure 4.21) :

- *UML* vers le format de représentation de modèles considéré (a) : Lorsque le moteur d'*aspectual templates* utilise des opérations définies dans le moteur de sous-modèles (*e.g. closure*), il utilise l'adaptateur. Celui-ci prend en entrée un *package UML* et le converti au format requis par le moteur de sous-modèles.
- Format de représentation de modèles vers *UML* (b) : Lorsque le moteur de sous-modèles a effectué toutes les opérations sur le modèle concerné, il le retourne au moteur d'*aspectual templates* via l'adaptateur. Ce dernier transforme alors le modèle en *package UML*.

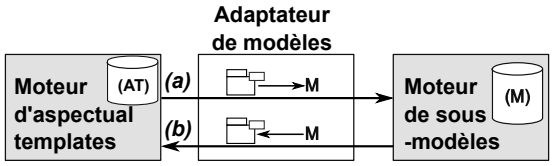


FIGURE 4.21 – Conversions de modèles

Chapitre 5

Application : Modélisation d'un serveur REST d'agrégation d'informations

5.1 Objectif

Le but de cette application est de valider l'approche IDM basée sur les templates (cf. chapitre 3.2) au travers d'un scénario de modélisation incrémentale d'un serveur *REST* d'agrégation d'informations.
Différents opérateurs de construction (*merge, apply, instantiate*), de décomposition (*extract, unbind*), de paramétrisation (*promote, restrict*) ainsi que de recherche de modèles et d'inférence de substitutions (*findCompatibleHierarchy, getBoundSubstitutions*), détaillés en section 4.1, sont utilisés afin de répondre aux spécifications décrites en sous-section 5.2.1.

5.2 Contexte du scénario

5.2.1 Critères du serveur REST

Une agence de presse souhaite pouvoir accéder rapidement et le plus indépendamment possible (*i.e.* contrôler elle-même le moyen d'accès) aux informations provenant d'autres agences et journaux internationaux. Pour cette raison, elle souhaite acquérir une plateforme lui permettant d'aggréger différents flux d'informations. Ces flux de données peuvent provenir de sites web (flux *rss*, *atom*,...) ou de comptes sur des réseaux sociaux. Cette plateforme doit répondre aux critères suivants :

1. *Accès au serveur d'aggrégation* : Pour des raisons d'indépendance de la presse, l'agence souhaite pouvoir accéder et gérer elle-même le serveur,
2. *Accès sécurisé au serveur* : Seuls les journalistes et autres agences doivent pouvoir se connecter au serveur,
3. *Accès performant au serveur (Passage à l'échelle)* : Malgré un nombre important de connexions au serveur, un journaliste doit toujours pouvoir accéder aux données souhaitées,
4. *Serveur extensible* : Dans un second temps, l'agence souhaite pouvoir étendre le serveur avec d'autres fonctionnalités (notamment une messagerie sécurisée),
5. *Un seul moyen d'accès aux données* : Les journalistes de l'agence doivent pouvoir accéder aux données via un seul et même moyen, même en cas d'extension du serveur d'agrégation.

L'agence de presse s'adresse à une *ENL*[1] afin d'obtenir une telle plateforme.

1. Entreprise du Numérique Libre.

5.2.2 Équipe de conception

Composée d'une trentaine de concepteurs et développeurs, *l'ENL* utilise régulièrement, pour la conception des projets sur lesquels elle travaille, un dépôt de modèles et une application autorisant la modélisation en équipe. Cette dernière permet aux concepteurs d'utiliser les *templates* et les modèles du dépôt. Ces concepteurs peuvent endosser deux rôles, présentés en chapitre 3 : les *concepteurs de templates* et les *concepteurs de systèmes*.
Après une phase de définition des besoins avec l'agence de presse, l'équipe de conception affectée au projet conçoit un premier modèle sous forme de diagramme de classes *UML*. Ayant déjà conçu des projets dont l'architecture reposait sur *REST* et d'autres utilisant divers serveurs, l'équipe débute la phase de modélisation en réutilisant deux hiérarchies de modèles issus du dépôt ainsi qu'une bibliothèque de patrons de conception sous forme de *templates*. Ces derniers et les deux hiérarchies de modèles sont présentés en section suivante.

5.3 Existant

5.3.1 Bibliothèque de patrons de conception

Lors des différents projets que *l'ENL* a modélisés et développés, les concepteurs de *templates* ont utilisé plusieurs *patrons de conception* classiques. Dans un but de réutilisation, ceux-ci ont été modélisés sous la forme de *templates*, puis stockés dans le dépôt de modèles.
Les patterns utilisés lors de la modélisation du serveur *REST* sont exposés en figure 5.1 et sont les suivants : *Mandataire* (*ProxyPattern*), Décorateur (*DecoratorPattern*), Adaptateur (*AdapterPattern*), Canal d'événements (*EventChannelPattern*) et Observateur (*ObserverPattern*).

5.3.2 Hiérarchies de modèles

Au fur et à mesure des projets, les modeleurs ont élaboré plusieurs modèles de serveurs concernant la gestion de ressources et des serveurs de gestion de données. Les serveurs modélisés répondent à des besoins différents allant de la sécurité à la gestion de performances. Certains ont été obtenus par extension d'autres modèles de serveurs existant, conduisant à des hiérarchies de modèles respectives.
Les modèles des deux hiérarchies, présentées en annexe C, sont connectés à l'aide de relations d'inclusion, telle que celle présentée dans le chapitre 3. Il s'agit de hiérarchies de modèles représentant respectivement des *serveurs de ressources* et des *serveurs de données*.

5.4 Étapes de modélisation

Comme indiqué en sous-section 5.2.1, le serveur *REST* doit répondre à certains critères, *i.e. accès sécurisé au serveur, accès performant au serveur et extensibilité du serveur* : la phase de modélisation de celui-ci est découpée en trois étapes correspondant à ces critères. Ces étapes sont présentées en annexe D, respectivement au-travers des figures D.1, D.2 et D.3. Ces représentations sont inspirées des *diagrammes de Gantt* (Wilson [104]). Par exemple, pour la première étape (D.1 - *accès sécurisé au serveur*) : la colonne de gauche représentent les modeleurs concernés par la conception d'une fonctionnalité du serveur, alors qu'une telle fonctionnalité est représentée dans la colonne de droite (*e.g.* le concepteur *C* modélise la fonctionnalité d'accès au serveur par mot de passe). Dans cette colonne de droite, chacune des fonctionnalités est modélisée à l'aide de séquence d'opérateurs (représentée par une flèche), *i.e.* un opérateur peut utiliser le résultat d'un

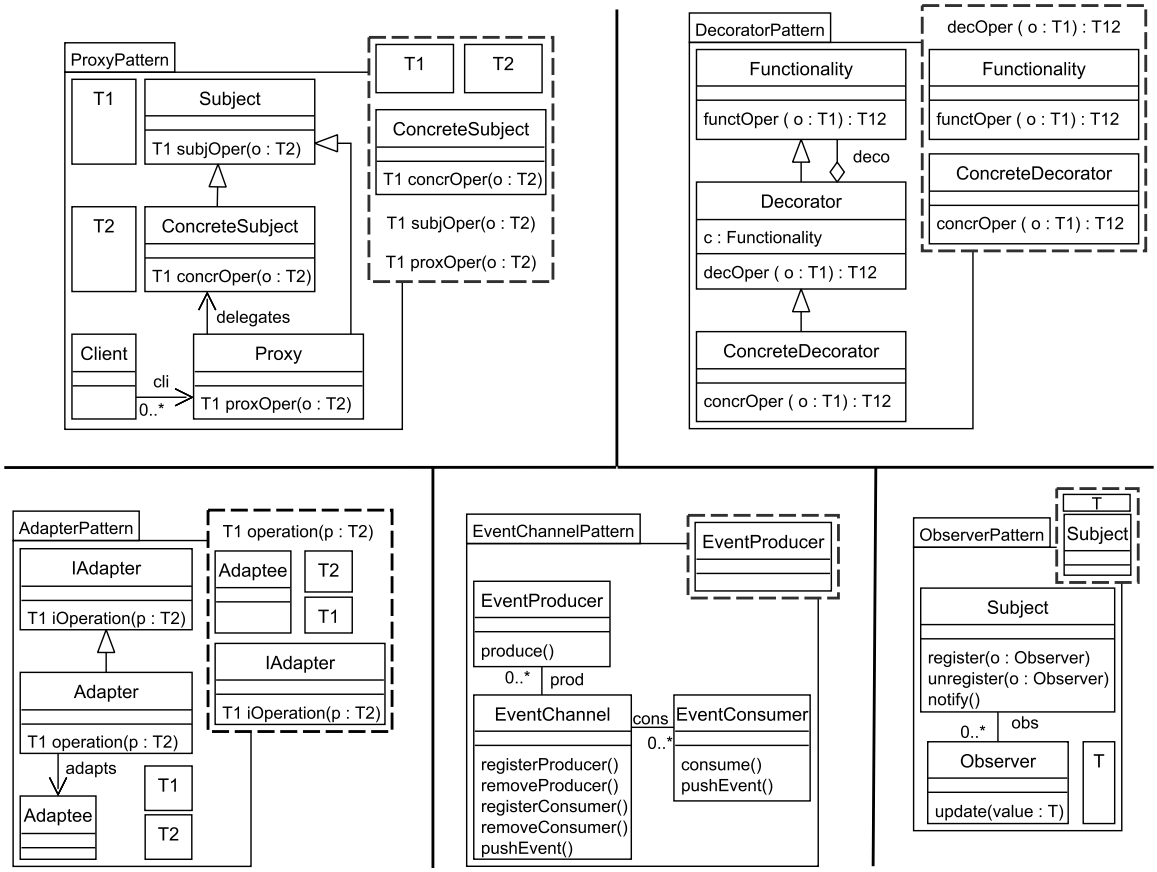


FIGURE 5.1 – Modélisation du serveur *REST* : *patrons de conception* (*templates*) utilisés

autre (*e.g.* pour la fonctionnalité d'accès au serveur par mot de passe, la séquence d'opérateurs est composée de *extract* et de *apply* : le second utilise le résultat du premier).
Chacune de ces trois étapes de modélisation sont présentées dans la sous-section suivante pour les concepteurs de système (identifiés à l'aide du label "CS", suivit d'un identifiant, *e.g. "CS1"*) et en sous-section 5.6 pour les concepteurs de *templates* (identifiés par "CT", suivit d'un identifiant, *e.g. "CT1"*). Concernant ces derniers, leur objectif est de fournir des templates aux concepteurs de système pour modéliser le serveur.

5.5 Première partie : Concepteurs de système

En s'appuyant sur les activités de modélisation, détaillées en chapitre 3 et les opérateurs présentés en section 4.1, les concepteurs de système ont accès aux opérateurs suivants : *(1 - Construction) merge, apply, instantiate, (2 - Extraction) extract, unbind* et *(3 - Recherche de modèles et inférence de substitutions) subsInference, findCompatibleHierarchy et getBoundSubstitutions.*
Les trois étapes de modélisation (Accès sécurisé et performant au serveur et extensibilité de ce dernier) sont respectivement présentées dans les sous-sections 5.5.1, 5.5.2 et 5.5.3.

5.5.1 Accès sécurisé au serveur de ressources

Objectifs du scénario : L'objectif de cette étape est d'étendre le serveur mandataire (Patterns *Serveur mandataire* et *Decorateur*) modélisé par les concepteurs de *templates* (sous-section

5.6) avec des fonctionnalités sécurisant son accès. Pour cela, les modeleurs ont défini les deux fonctionnalités suivantes : *Limitation du nombre de requêtes sur une ressource* et *Accès au serveur par mot de passe.*

Limitation du nombre de requêtes sur une ressource en un temps donné Cette fonctionnalité a pour but d'éviter un nombre trop important de requêtes clientes sur une ressource en un temps donné : un grand nombre de requêtes sur le serveur en un temps très court peut fortement ralentir, voir bloquer l'accès à celui-ci. La fonctionnalité à modéliser doit donc limiter l'impact de ce type d'attaques (*DDOS*[2]). La figure 5.2 décrit la séquence d'opérateurs utilisée pour la représentation de cette fonctionnalité par le concepteur de système *CS1*. Pour rappel (cf. section 5.4), chacun des concepteurs de système sont identifiés dans le scénario par le label "C(*oncepteur de*) S(*ystème*)", suivit d'un identifiant.

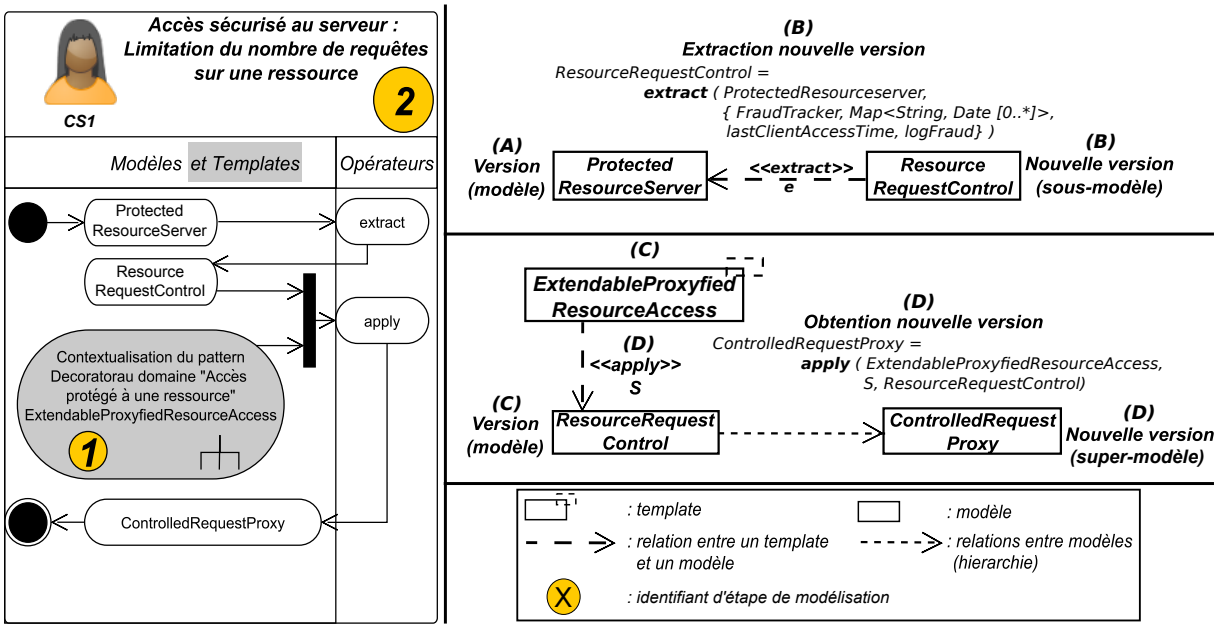


FIGURE 5.2 – Limitation du nombre de requêtes sur une ressource

La numérotation indiquée sur le diagramme d'activité, situé dans la partie gauche de la figure, représente les numéros d'étape de modélisation, exposées dans l'annexe D.1.
Le modeleur sélectionne la fonctionnalité souhaitée dans le modèle *ProtectedResourceServer* (partie supérieure droite de la hiérarchie de modèles, en figure C.1), via l'opérateur *extract*. Dans un second temps, il utilise l'opérateur *apply* afin d'ajouter au serveur mandataire la fonctionnalité limitant le nombre de requêtes en un temps donné. Le modeleur a extrait la fonctionnalité telle que décrite dans la partie supérieure droite de la figure 5.2 : en partant d'une version de modèle existante (*A* sur la figure), le modeleur a utilisé l'opérateur *extract*. Il a passé en paramètres de cet opérateur (*B* sur la figure) le nom de la version de modèle *ProtectedResourceServer* dont il souhaitait extraire la fonctionnalitéet l'ensemble des éléments composants celle-ci, *i.e.* la classe *FraudTracker* et ses éléments. Il a ainsi obtenu une nouvelle version de modèle *ResourceRequestControl*, sous-modèle de *ProtectedResourceServer*. Le résultat de cette extraction est présenté en figure 5.3.
La classe *FraudTracker* et ses éléments ont été extrait car *(1)* les accès à une ressource sont

2. *Denial of Service attack* : Attaque par Déni de Service - Hancock [52]

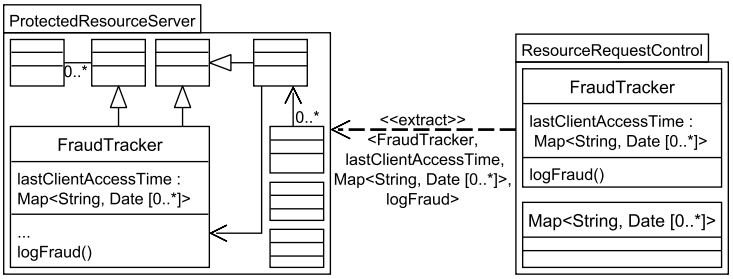


FIGURE 5.3 – Résultat de l'extraction de *ResourceRequestControl* à partir de *ProtectedResourceServer*

stockés dans un dictionnaire (*lastClientAccessTime*) : pour une ressource (son URI, sous forme de chaine de caractères) correspondent toutes les dates et heures d'accès. Et *(2)* selon les dates et heures d'accès à une ressource, l'accès à celle-ci sera bloqué pendant un certain temps (une attaque est détectée quand le temps est trop court entre chaque accès). Cette restriction d'accès lors de la détection est enregistrée via la méthode *logFraud*.

Après avoir ainsi obtenu *ResourceRequestControl* par extraction, le modeleur a étendu le serveur mandataire avec cette fonctionnalité, tel que décrit dans la partie inférieure droite de la figure 5.2 : en partant d'un *template* existant (*C* sur la figure), le modeleur a utilisé l'opérateur *apply*. Il a passé en paramètres de celui-ci (*D* sur la figure) le *template* *ExtendableProxyfiedResourceAccess*, représentant le serveur mandataire extensible [3], le modèle *ResourceRequestControl* extrait précédemment et un ensemble de substitutions. Ceci lui a permis d'obtenir la version de modèle *ControlledRequestProxy*, sur-modèle de *ResourceRequestControl*. Ceci est exposé en figure 5.4.

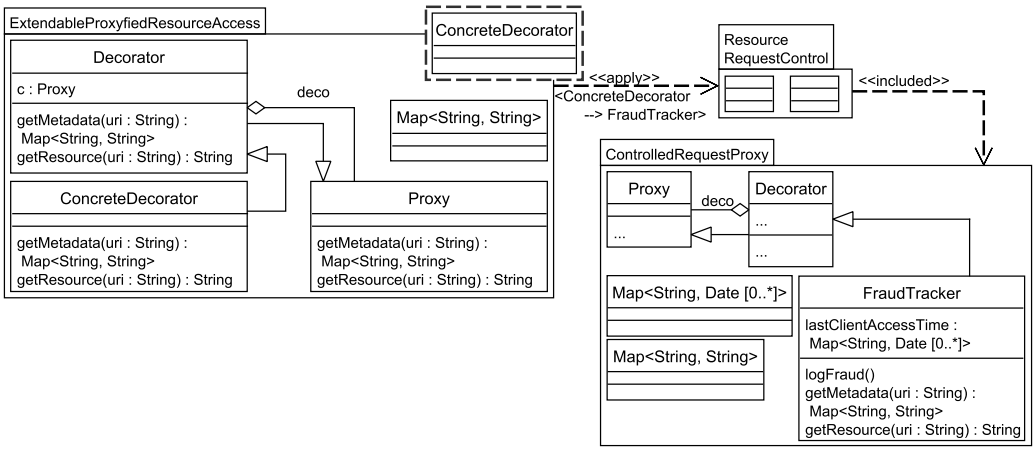


FIGURE 5.4 – Résultat de l'application de *ExtendableProxyfiedResourceAccess* sur *ResourceRequestControl*

Accès par mot de passe L'objectif est ici d'empêcher la connexion de n'importe quel individu ou organisation au serveur de ressources. La fonctionnalité à modéliser doit donc restreindre l'accès au serveur par l'utilisation d'un *login* et d'un mot de passe. Les opérateurs utilisés sont décrits en figure 5.5.

Le concepteur choisit la fonctionnalité souhaitée dans le modèle *PowerControlledServer*, avec l'opérateur *extract*. Ensuite, il utilise *apply* afin d'ajouter au serveur mandataire la fonctionnalité

3. Ce *template* semi-contextualisé a été construit par le concepteur de *template* *CT1*, en partie 5.6.1

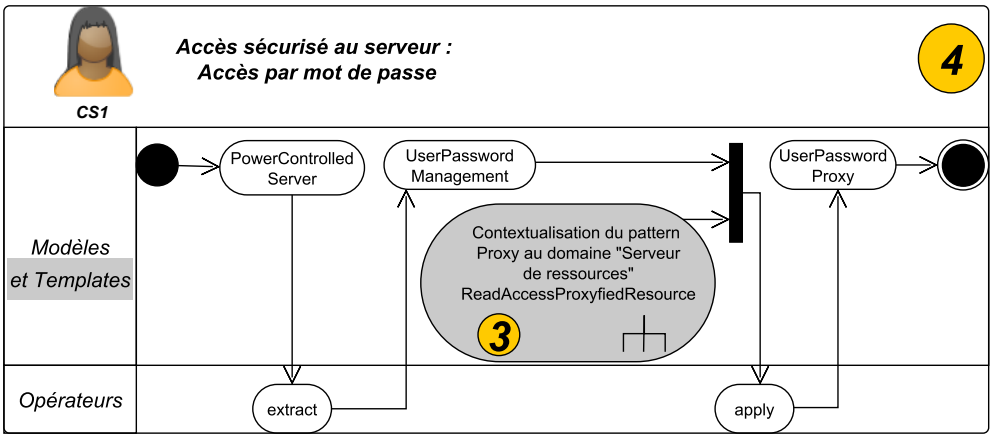


FIGURE 5.5 – Accès par mot de passe

restreignant l'accès par mot de passe et *login*.
Le modèle extrait est *UserPasswordManagement*, sous-modèle de *PowerControlledServer*, tel que présenté dans la figure 5.6.

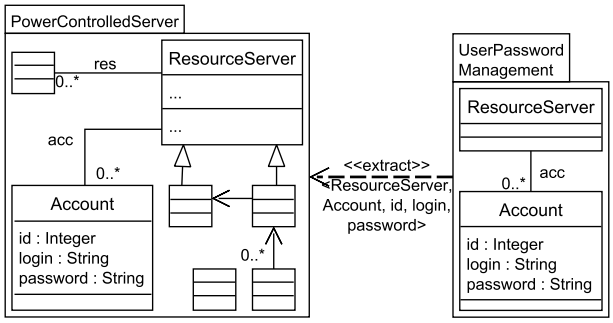


FIGURE 5.6 – Résultat de l'extraction de *UserPasswordManagement* à partir de *PowerControlledServer*

Les classes *ResourceServer*, *Proxy* et *Account* et leurs constituants ont été extraits car *(1)* les *logins* et mots de passes des différents comptes autorisés sont gérés par le serveur de ressources, auquel *(2)* des utilisateurs vont se connecter via un serveur mandataire.
Le modeleur a ensuite appliqué le *template* représentant le serveur mandataire sur *UserPasswordManagement*, obtenant ainsi le modèle *UserPasswordProxy*, exposé en figure 5.7.

Fusion des fonctionnalités Après avoir modélisé les deux fonctionnalités précédentes, le concepteur *CS1* les fusionne afin d'avoir un accès sécurisé au serveur. L'opérateur utilisé est *merge* (cf. figure 5.8), prenant en paramètre les deux modèles résultant des étapes précédentes, *i.e. ControlledRequestProxy* et *UserPasswordProxy* (*A* sur la figure).
Le résultat de cette fusion est la version *SecuredAccessServer* (*B* dans la partie basse de la figure), présentée en figure 5.9.
Le modèle résultat permet ainsi à des clients de se connecter au serveur de ressources mais en sécurisant celui-ci par *(1)* la présence d'un serveur mandataire restreignant l'accès aux ressources en lecture uniquement (utilisation des *templates* construits en sous-section 5.6), *(2)* la limitation du nombre de requêtes sur une ressource en un temps donné et *(3)* une connexion

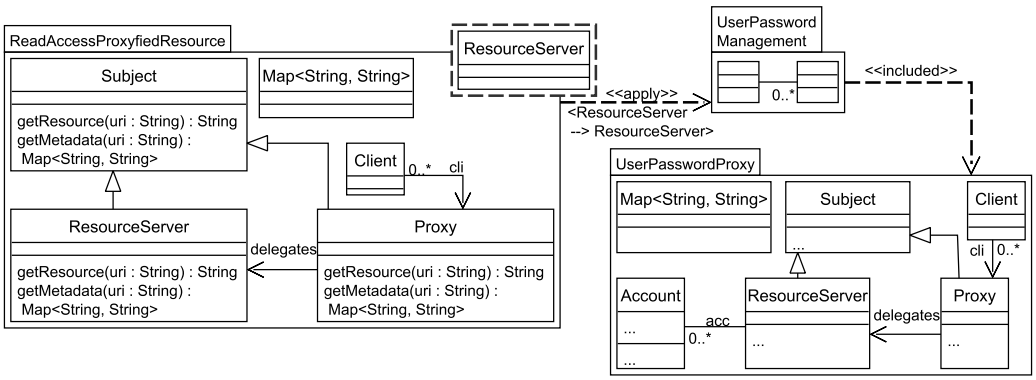


FIGURE 5.7 – Résultat de l'application de *ProxyfiedResourceAccess* sur *UserPasswordManagement*

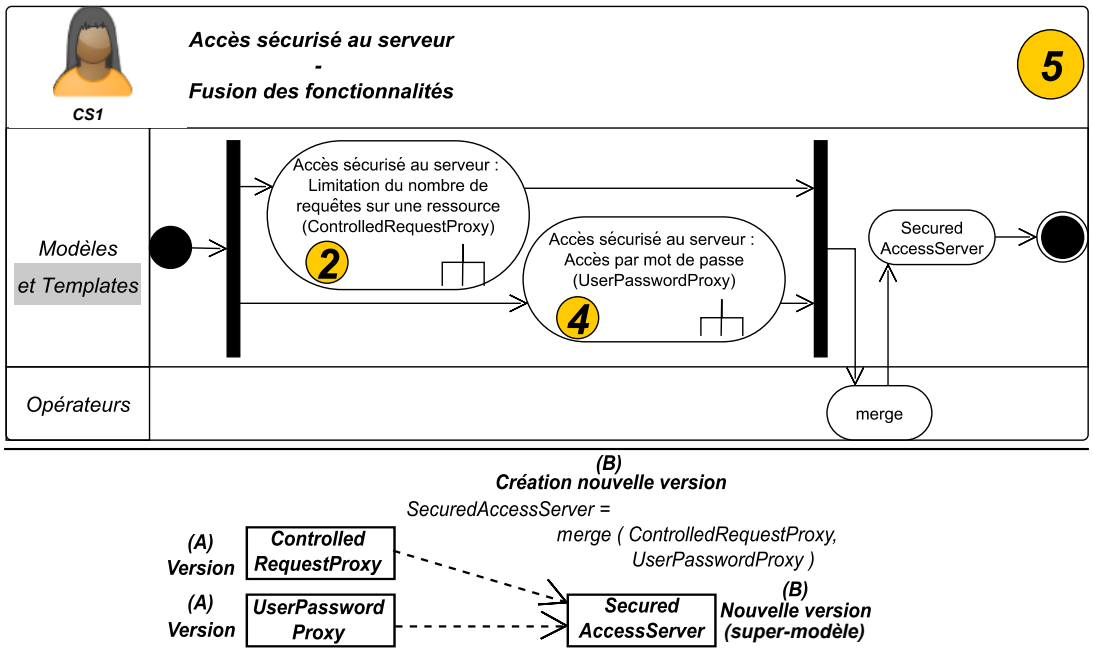


FIGURE 5.8 – Fusion des fonctionnalités

via un *login* et un mot de passe des clients.

Évolution de la hiérarchie de modèles (Serveurs de ressources) La modélisation de l'accès sécurisé au serveur a créé des versions de modèles au sein de la hiérarchie de sous-modèles des figures C.1 et C.2.
Une version simplifiée (i.e. se concentrant sur les modèles créés précédemment) est exposée en figure 5.10.
Chacune des fonctionnalités modélisées (**2** : *Limitation du nombre de requêtes sur une ressource*, **4** : *Accès par mot de passe*) a généré une nouvelle branche dans la hiérarchie.
Comme on peut le voir en figure 5.10, deux branches ont été créées, via les opérateurs *extract* et *apply*. La première branche contient *UserPasswordProxy*, version alternative de *UserPasswordManagement* (l'autre version alternative étant *PowerControlledServer*). La seconde branche contient quand à elle *ControlledRequestProxy*, incluant *ResourceRequestControl*.

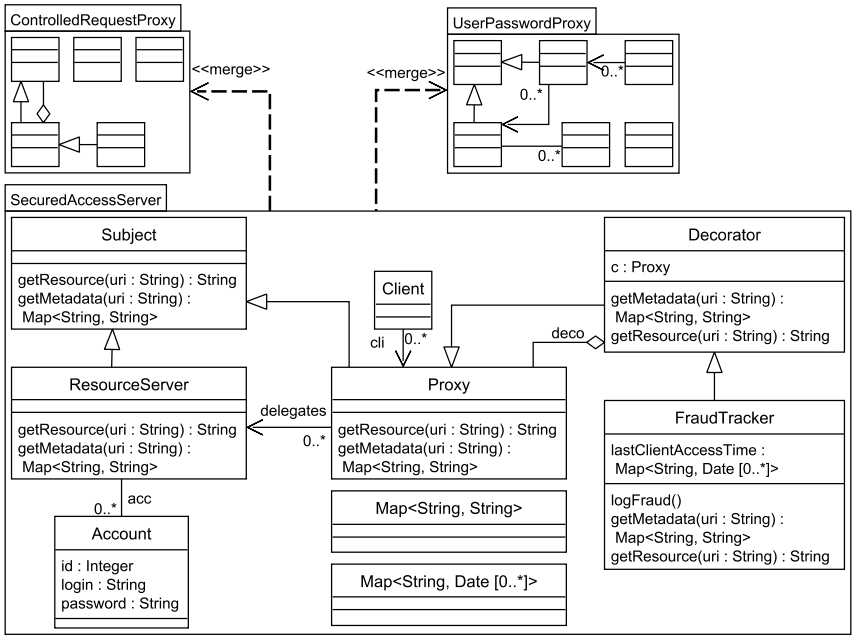


FIGURE 5.9 – Résultat de la fusion de *UserPasswordProxy* avec *ControlledRequestProxy*

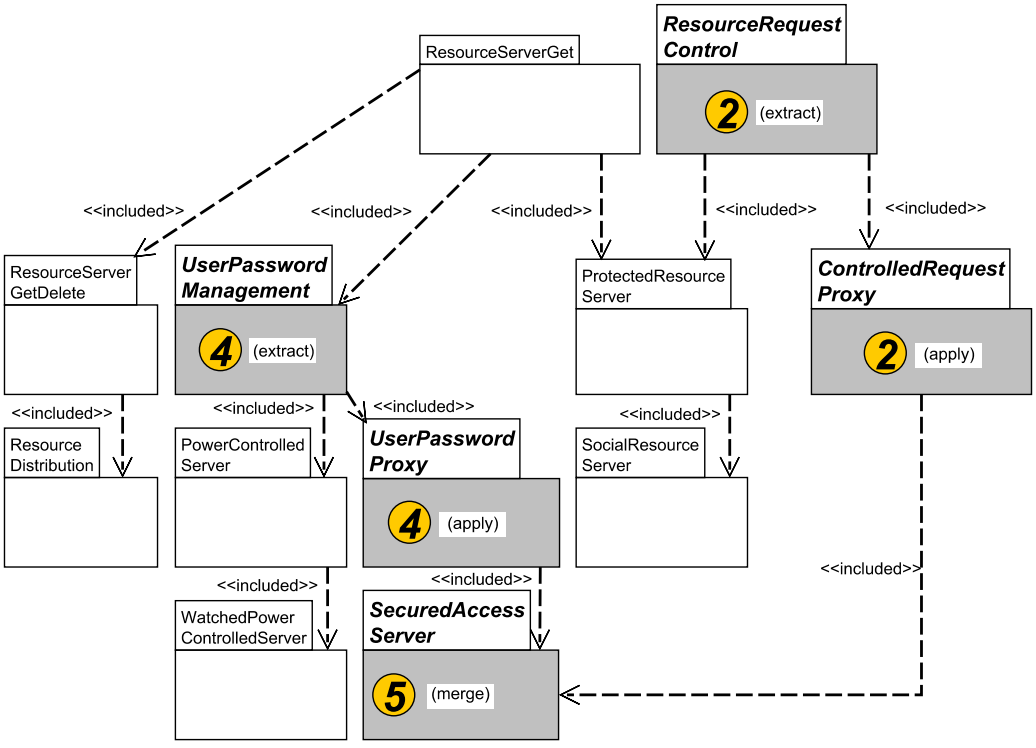


FIGURE 5.10 – Serveur de ressources : Hiérarchie de modèles après modélisation de l'accès sécurisé au serveur

Les versions finales (*ControlledRequestProxy* et *UserPasswordProxy*) des deux branches ont par la suite été fusionnées (5), permettant d'obtenir *SecuredAccessServer*.

5.5.2 Accès performant au serveur

Objectifs du scénario : Cette seconde étape modélise l'accès performant au serveur, via deux fonctionnalités : la *limitation du temps d'existence des ressources* et la *limitation du nombre de ces ressources.*

Limitation du temps d'existence des ressources Avec cette fonctionnalité, les ressources les plus demandées par les utilisateurs ont une durée de vie, dans le cache du serveur, plus longue que les autres. Ceci est exposé en figure 5.11.

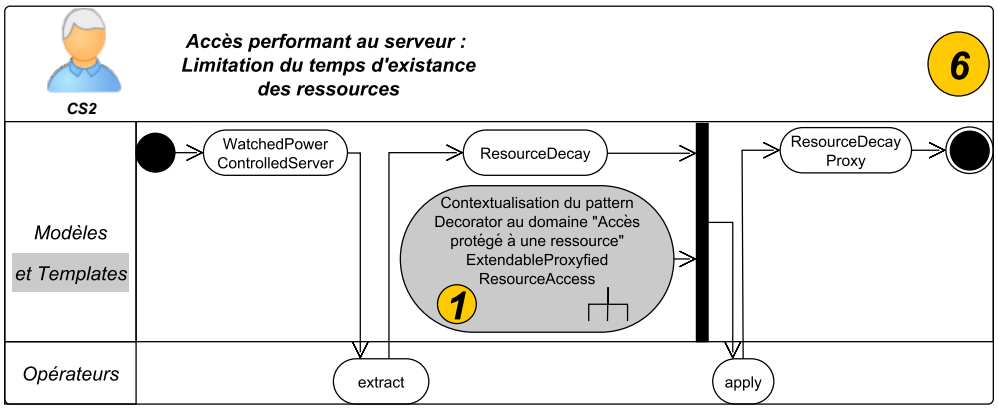


FIGURE 5.11 – Limitation du temps d'existence des ressources

Le concepteur *CS2* sélectionne la fonctionnalité souhaitée dans le modèle *WatchedPowerControlledServer* (cf. hiérarchie en figure C.2), via l'opérateur *extract*. La version de modèle obtenue par extraction (*ResourceDecay*), sous-modèle de *WatchedPowerControlledServer*, est présentée dans la figure 5.12.

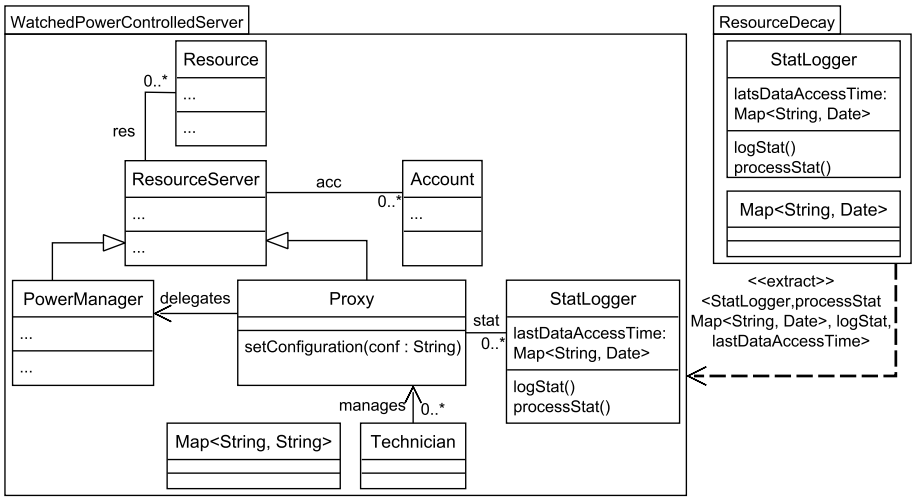


FIGURE 5.12 – Résultat de l'extraction de "ResourceDecay" à partir de "WatchedPowerControlledServer"

La classe *StatLogger* et ses constituants ont été extraits car ils permettent de maintenir à jour un dictionnaire contenant des ressources et les dates et heures des requêtes clientes correspondantes.

L'enregistrement est effectué dans le dictionnaire *lastDataAccessTime*. Lorsqu'une ressource est demandée, une comparaison entre les dates et heures des ressources enregistrées est effectuée : la ressource demandée remplace celle ayant la date la plus ancienne dans le dictionnaire.
Dans un second temps, le modeleur utilise *apply* afin d'ajouter au serveur de ressources la fonctionnalité de mise à jour du cache, extraite précédemment. Le *template ExtendableProxyfiedResourceAccess*, représentant le serveur mandataire extensible, est appliqué sur *ResourceDecay*, comme illustré en figure 5.13.

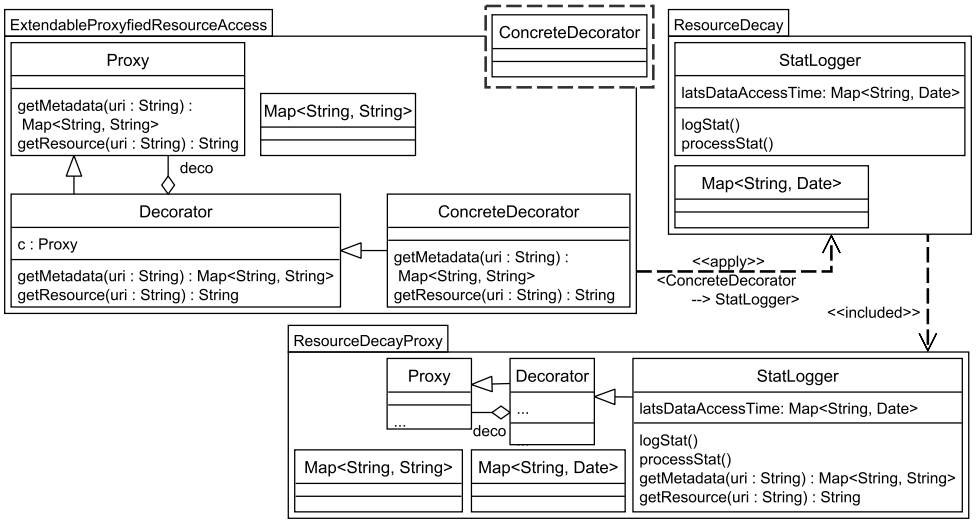


FIGURE 5.13 – Résultat de l'application de "ResourceExtendedProxy" sur "ResourceDecay"

Puisque le *template ExtendableProxyfiedResourceAccess* utilise le pattern *Décorateur*, d'autres fonctionnalités vont pouvoir étendre le serveur mandataire et être composées avec celle de limitation du temps d'existence des ressources.

Limitation du nombre de ressources L'objectif est ici de limiter le nombre de ressources gérées par le serveur, afin d'en améliorer les performances. Ceci est décrit en figure 5.14, avec le concepteur *CS2*.
Ce dernier sélectionne, dans la hiérarchie (voir figure C.2), le modèle *ResourceDistribution*. Dans celui-ci, se trouve des fonctionnalités de gestion du cache du serveur, telles que les opérations *acquireReusable, releaseReusable*, *setMaxPoolSize* et le *pool* de ressources (*reusables*) gérées par le serveur. Quand la requête d'un client pour une ressource arrive sur le serveur, celui-ci va chercher dans le *pool* de ressources celle correspondant à l'*URI* souhaitée via l'opération *acquireReusable* et la retourne au client. Tant que ce dernier utilise la ressource, l'accès à celle-ci est bloquée pour les autres clients. Une fois que le client retourne la ressource, elle est à nouveau disponible et indiquée comme tel via l'opération *releaseReusable*. Quant à la taille du *pool* de ressources, elle est définie par l'opération *setMaxPoolSize*. Ceci permet d'améliorer les performances d'un serveur en limitant le nombre de ressources à stocker dans son cache mais aussi en évitant d'instancier une ressource pour laquelle une instance existe déjà.
Cependant, des constituants inutiles à la fonctionnalité en cours de modélisation sont présents : il s'agit de la classe *Observer* et de son opération *update*, de l'association *obs* et des opérations présentes au sein de la classe *Ressource*. Ces constituants sont ceux du pattern Observer, tel que modélisé dans le dépôt de modèles sous la forme d'un *template* (cf. 5.3.1). Dans le cas présent, ils sont inutiles car les utilisateurs doivent passer par un serveur mandataire afin d'accéder aux

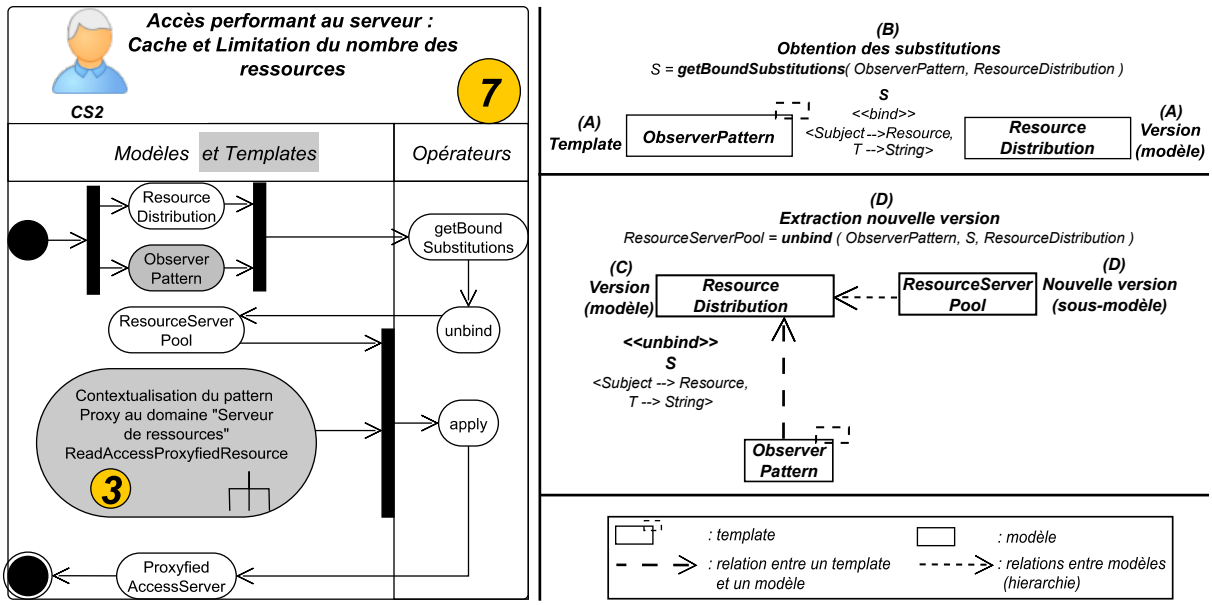


FIGURE 5.14 – Cache et limitation du nombre des ressources

ressources.
Tout d'abord, afin de déterminer précisément comment le pattern Observer est présent au sein de *ResourceDistribution*, le modeleur utilise l'opérateur *getBoundSubstitutions* (*(B)*, en partie droite de la figure 5.14). Il passe en paramètre le modèle *ResourceDistribution* et le *template ObserverPattern* (*(A)*). Il obtient alors l'ensemble de substitutions *{(Subject, Resource), (T, String)}*, pour lequel le modèle inclut une instance du *template*. Ensuite, via cet ensemble de substitutions et l'opérateur *unbind*, le modeleur obtient la version de modèle *ResourceServerPool* (*(D)*), sous-modèle de *ResourceDistribution* (*(C)*), et n'incluant plus le *template ObserverPattern*. Le résultat de cette extraction est présenté dans la figure 5.15.

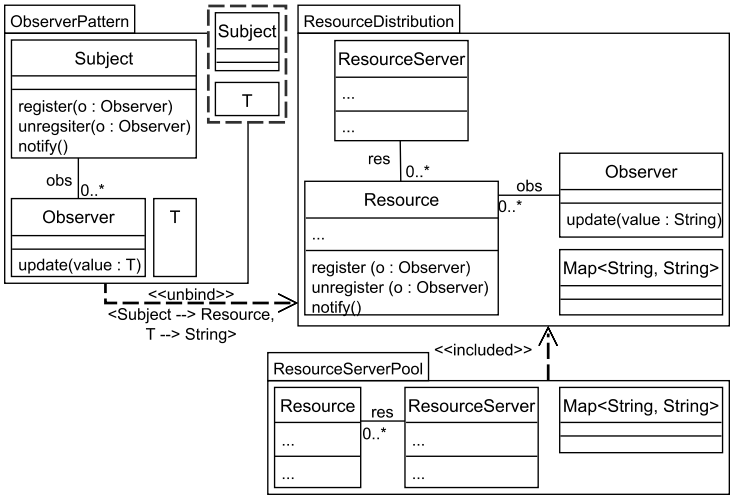


FIGURE 5.15 – Résultat de l'extraction de *ResourceServerPool* à partir de *ResourceDistribution*

Une fois *ResourceServerPool* obtenu par extraction, le modeleur applique le *template ReadAccesProxyfiedResource*, dont le résultat est *ProxyfiedAccessServer*, sur-modèle de *ResourceServerPool*. Ce *template* a été appliqué puisqu'il représente le serveur mandataire, restreignant l'accès aux

ressources en lecture (la définition de ce *template* a été réalisée par un concepteur de *templates* en sous-section 5.6). Le résultat de cette application est présenté dans la figure 5.16.

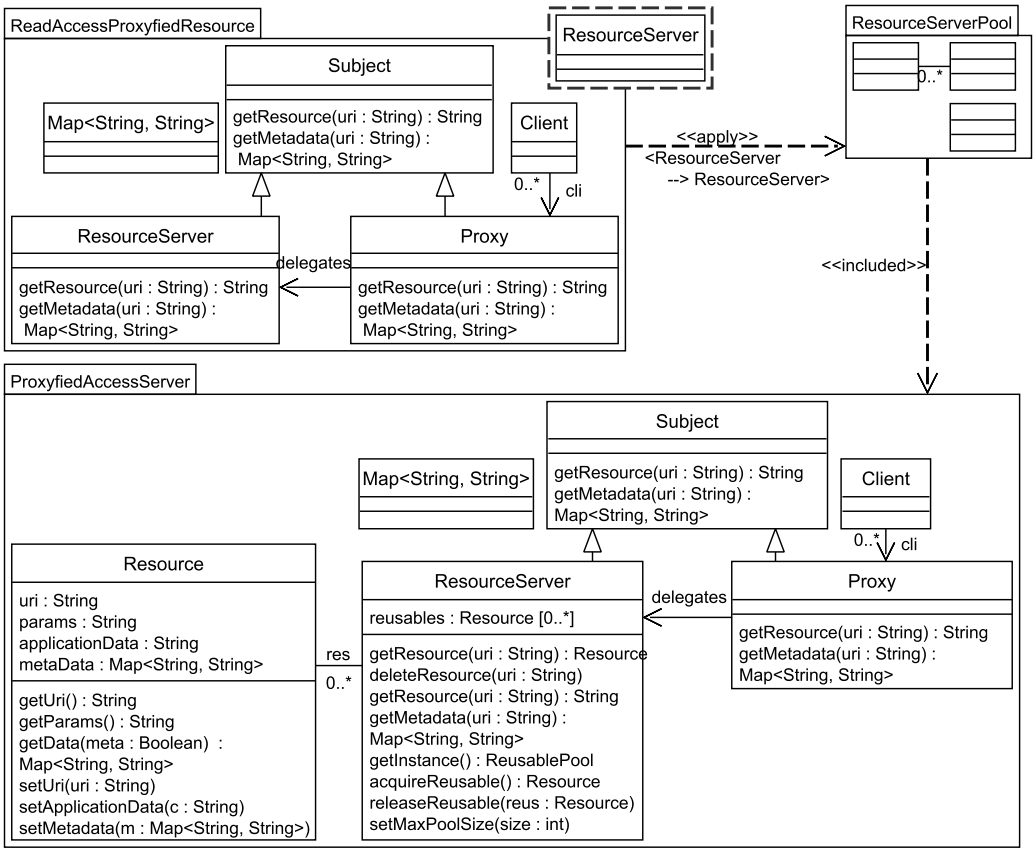


FIGURE 5.16 – Résultat de l'application de *ReadAccesProxyfiedResource* sur *ResourceServerPool*

Fusion des fonctionnalités Le concepteur *CS2* fusionne ensuite les modèles représentant les fonctionnalités précédentes afin d'avoir un accès performant au serveur. La figure 5.17 montre l'utilisation de l'opérateur *merge*.

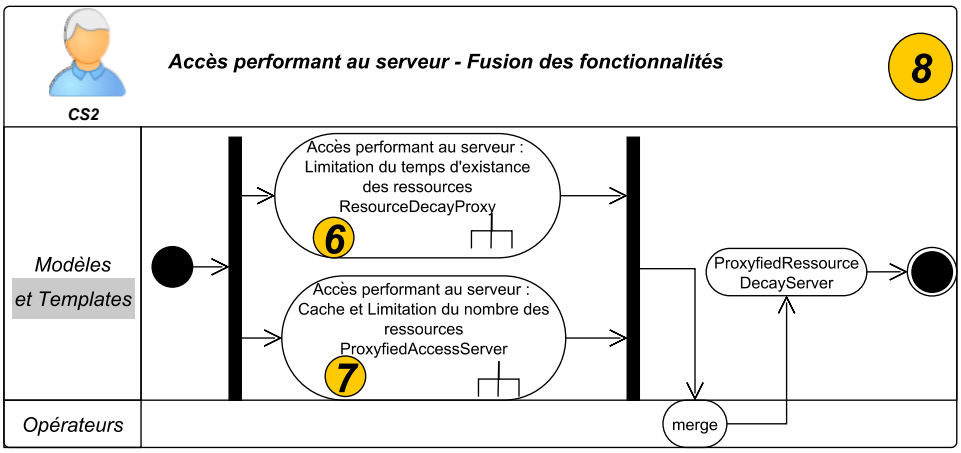


FIGURE 5.17 – Fusion des fonctionnalités

La version de modèle *ProxyfiedResourceDecayServer*, sur-modèle de *ResourceDecayProcy* et *ProxyfiedAccessServer*, est ainsi obtenue. Le résultat de cette fusion est exposé dans la figure 5.18.

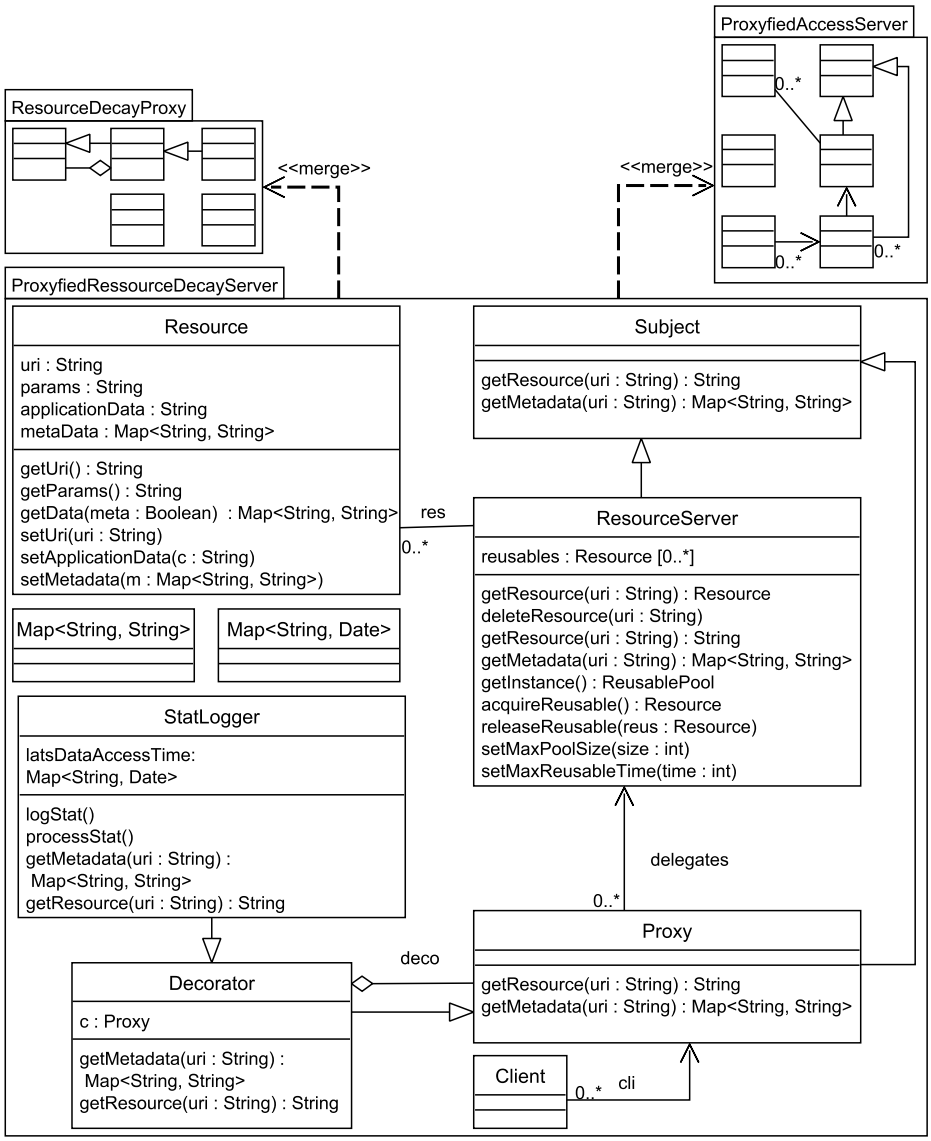


FIGURE 5.18 – Résultat de la fusion de “ResourceDecayProxy” avec “ProxyfiedAccessServer”

Ce modèle résultat permet ainsi à des clients de se connecter au serveur de ressources mais en ayant des temps d’accès aux ressources optimisés par *(1)* la présence d’un système de gestion du cache des ressources [4] et *(2)* la limitation de la durée de vie des ressources dans ce cache.

Évolution de la hiérarchie de modèles (Serveurs de ressources) La modélisation du critère d’accès performant au serveur a créé des versions de modèles au sein de la hiérarchie de sous-modèles des figures C.1 et C.2. Une version simplifiée est présentée en figure 5.19.
Chacune des fonctionnalités modélisées (**6** : *Limitation du temps d’existence des ressources*, **7** :

4. Le mécanisme décrit plus haut correspondant au pattern *ResourcePool* (appelé aussi *ObjectPool*).

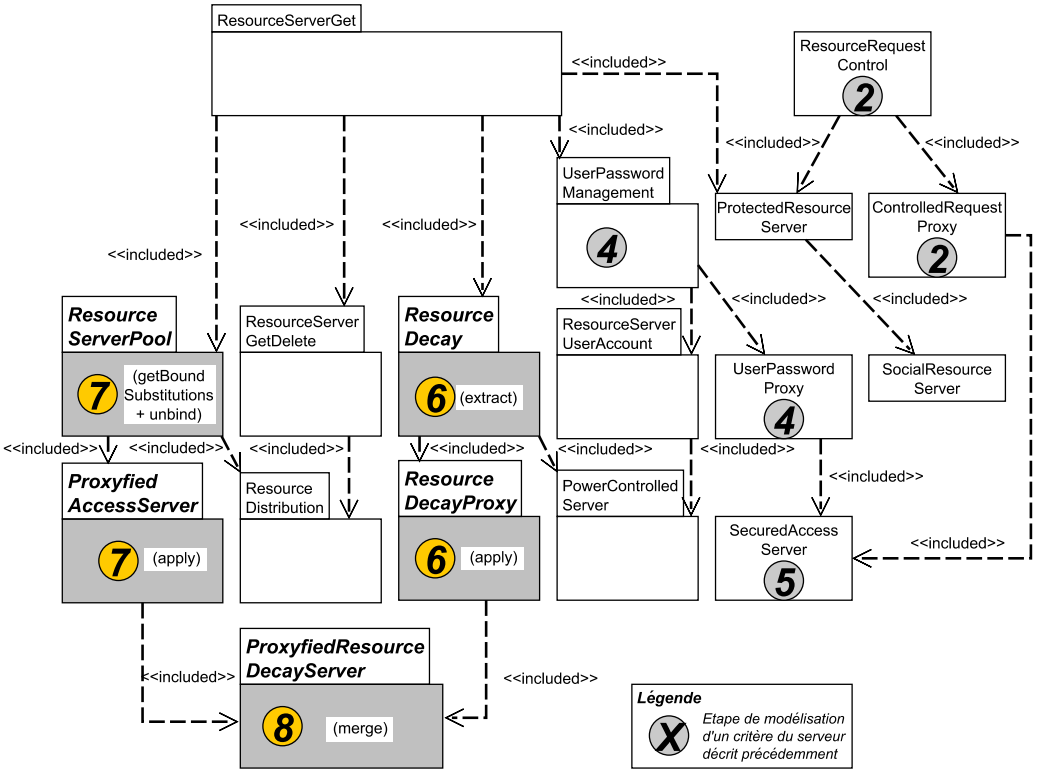


FIGURE 5.19 – Serveur de ressources : Hiérarchie de modèles après la modélisation du critère d'accès performant au serveur

Cache et limitation du nombre des ressources) a généré une nouvelle branche dans la hiérarchie. Comme on peut le voir en figure 5.19, deux branches ont été créées, via les opérateurs *extract* et *unbind*. La première branche contient *ResourceDecayProxy*, version alternative de *ResourceDecay*. La seconde branche contient quand à elle *ProxyfiedAccessServer*, incluant *ResourceServerPool*. La branche de *ResourceDecayProxy* a par la suite été fusionnée (8) avec *ProxyfiedAccessServer* afin d'obtenir *ProxyfiedResourceDecayServer*.

5.5.3 Extensibilité du serveur et accès aux informations

Objectifs du scénario : Cette section comporte deux parties. Premièrement, l'extensibilité du serveur d'informations et l'acquisition des données. L'extensibilité du serveur à pour objectif de donner la possibilité d'ajouter au serveur d'informations d'autres serveurs que le serveur de données. L'acquisition des données concerne la connexion du serveur à différents flux d'informations.
Deuxièmement, l'adaptation au serveur de ressources. Il s'agit de permettre l'accès au serveur de données via le serveur de ressources.

Extensibilité du serveur et acquisition des données Les concepteurs ont pour but de connecter le serveur de données aux différents flux d'informations (flux RSS, Atome (Hammersley [51]), etc...) externes, afin qu'il puisse agréger ces données dans sa base. Le concepteur *CS2* utilise les opérateurs présentés en figure 5.20.
Le modèle *ParallelServerDataObtaining* (en bas et à gauche de la figure C.3) est sélectionné. Il inclut le pattern *EventChannel*, avec notamment les classes *EventChannel* et *EventProducer*

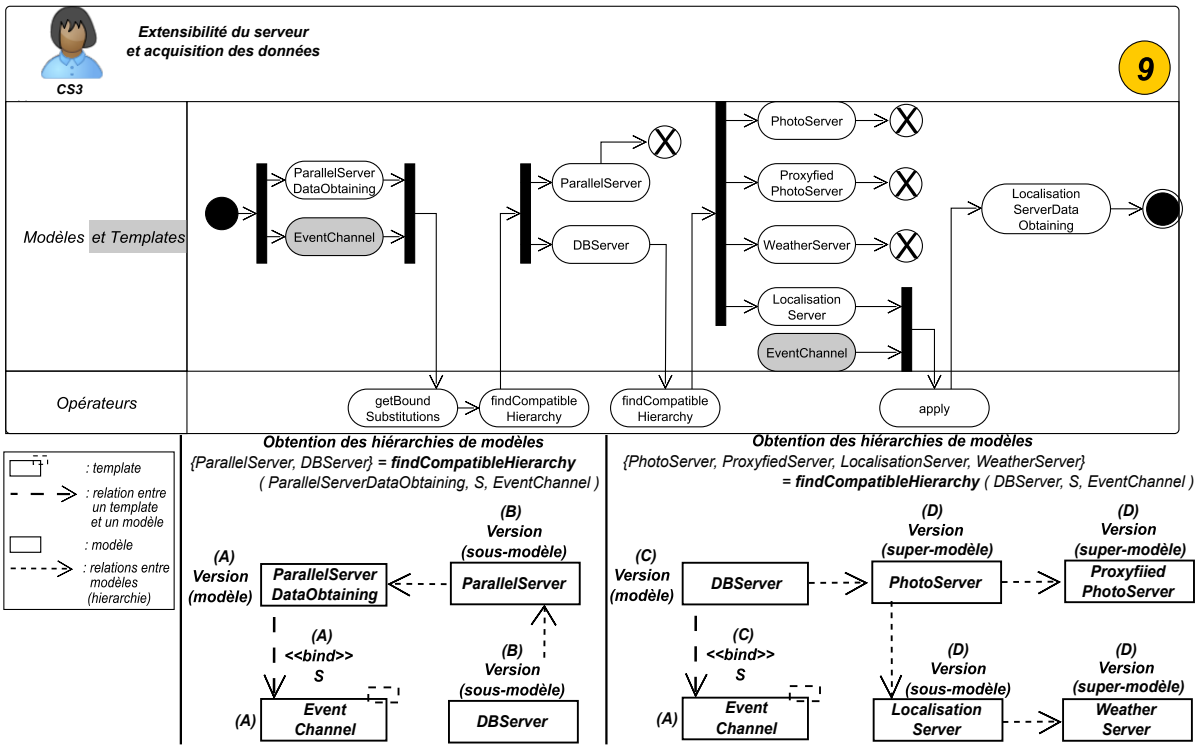


FIGURE 5.20 – Extensibilité du serveur d'information et acquisition des données

(cf. sous-section 5.3). Ce pattern permet la connexion de plusieurs serveurs : ici, seul le serveur de données doit être connecté au canal d'événements (*EventChannel*) mais d'autres serveurs pourraient l'être à l'avenir (tel un serveur de messagerie). Il permet par ailleurs d'ajouter divers moyens externes d'acquisition de données (*EventProducer*). L'autre pattern inclut dans *ParallelServerDataObtaining* est *Composite* (Gamma et al. [50]), soit les classes *AbstractServer*, *Cluster*, l'association *comp* et les généralisations entre *Cluster*, *Server* et *AbstractServer*. Ce second pattern est inutile dans le cadre du projet, car le serveur de données n'a pas à être défini sous la forme d'une grappe de serveurs (*Cluster*).

Le modeleur souhaite ainsi *(1)* retirer le pattern *Composite* et *(2)* ajouter des informations concernant la localisation géographique (photo, description, ...) et la date d'un événementqui sont des éléments absents de la classe *Data* du modèle *ParallelServerDataObtaining*. Le concepteur cherche donc un modèle incluant ces éléments et sur lequel peut être appliqué le *template* représentant le pattern *EventChannel*.

Afin de définir précisément comment le pattern *EventChannel* est présent au sein de *ParallelServerDataObtaining*, l'opérateur *getBoundSubstitutions* est utilisé. Celui-ci retourne l'ensemble de substitutions *{(EventConsumer, Server)}*. Avec cet ensemble, le concepteur de système *CS3* recherche, via *FindCompatibleHierarchy*, tous les super et sous-modèles de *ParallelServerDataObtaining* sur lesquels *EventChannel* est applicable.

Cette recherche de modèles est exposée dans la partie supérieure de la figure 5.20 : en partant de l'application, via l'ensemble de substitutions *S*, du template *EventChannel* sur la version de modèle *ParallelServerDataObtaining*, le concepteur passe ceux-ci en paramètres de l'opérateur *findCompatibleHierarchy* et trouve les sous-modèles de *ParallelServerDataObtaining* (*DBServer* et *ParallelServer* dans la partie basse et gauche de la figure (*(B)*)). Le résultat de cette recherche est présenté dans la figure 5.21.

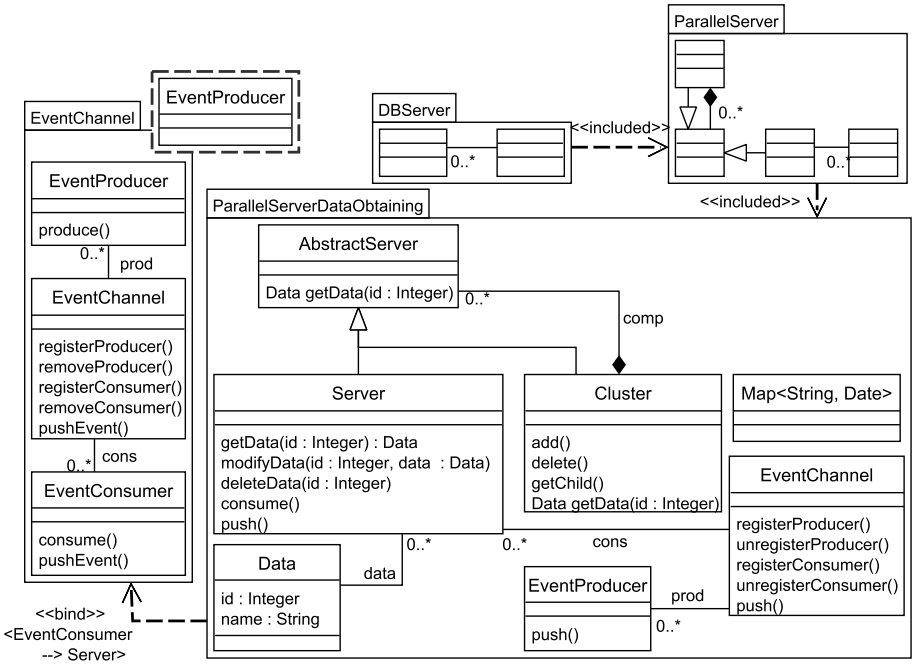


FIGURE 5.21 – Résultat de la recherche des sous-modèles compatibles avec l'applicabilité de *EventChannel* sur *ParallelDataServerObtaining*

Cependant, aucun des sous-modèles trouvés n'inclut les éléments souhaités et décrits plus haut. En utilisant *une seconde fois* l'opérateur *findCompatibleHierarchy* mais cette fois-ci entre le *template EventChannel* et le modèle *DBServer*, le modeleur trouve des surmodèles de *DBServer* sur lesquels *EventChannel* est applicable avec S. Parmi ceux-ci, le modèle *LocalisationServer* possède les informations de localisation souhaitées. Le résultat de cette recherche est exposé en figure 5.22.

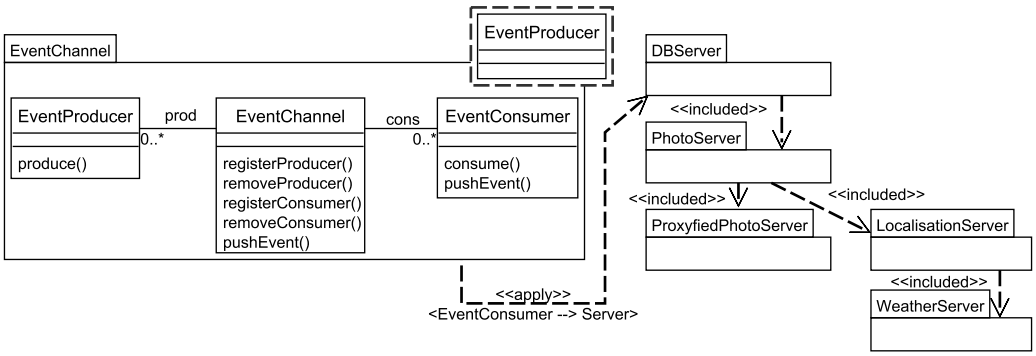


FIGURE 5.22 – Résultat de la recherche des sous-modèles compatibles avec l'applicabilité de "EventChannel" sur "DBServer"

LocalisationServer étant le modèle retenu, le modeleur applique le *template EventChannel* sur ce modèle, obtenant ainsi la version de modèle *LocalisationServerDataObtaining*, sur-modèle de *LocalisationServer*. Le résultat de cette application de *template* est présenté dans la figure 5.23.

Adaptation en serveur de ressources Ici, le serveur de données doit être connecté au serveur de ressources, afin de permettre aux clients d'accéder aux données stockées en base via une URI.

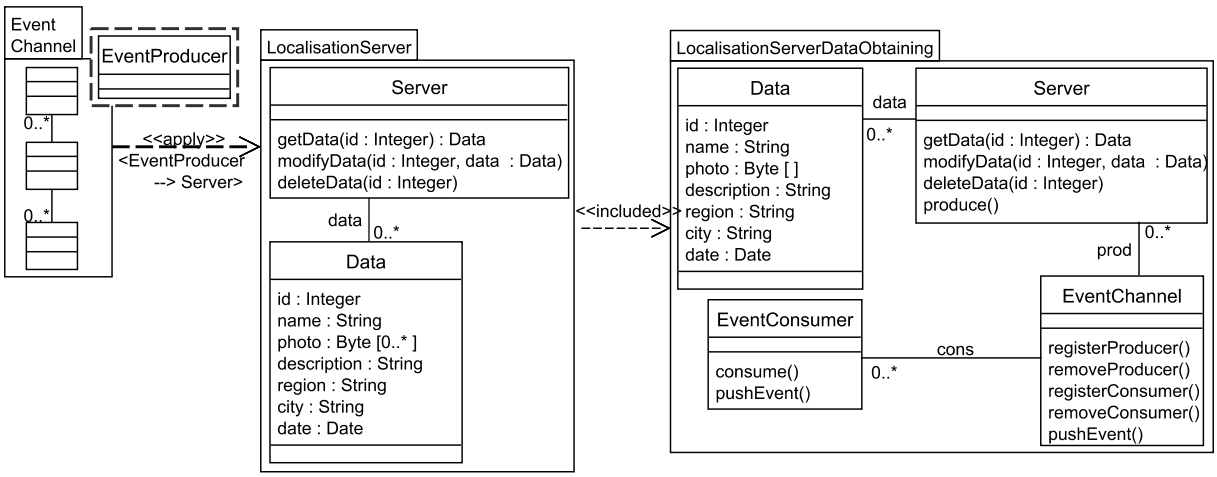


FIGURE 5.23 – Résultat de l'application de *EventChannel* sur *LocalisationServer*

La figure 5.24 présente les opérateurs utilisés par le concepteur *CS3*.

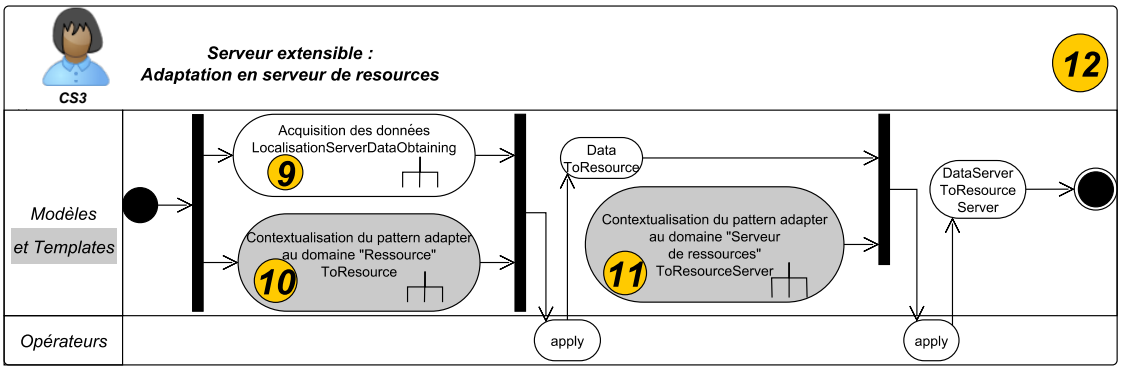


FIGURE 5.24 – Adaptation en serveur de ressources

Ce dernier sélectionne le modèle *LocalisationServerDataObtaining* créé précédemment, afin d'adapter l'accès au serveur de données (*Server*) à celui du serveur de ressources. Pour cela, il sélectionne deux *templates* représentant tous deux le pattern *Adapter* mais adaptés par un concepteur de *template* (cf. sous-section 5.6). Le premier *template*, *ToRessource*, a été modifié pour adapter un objet en ressource et le second, *ToRessourceServer*, afin d'adapter un objet en serveur de ressources. Le modeleur applique ces deux *templates*, permettant ainsi d'utiliser les données comme des ressources et le serveur de données comme un serveur de ressources. Le modèle *DataServerToResourceServer* résulte de ces applications, présentées dans les figures 5.25 (adaptation des données en ressource, modèle résultat *DataToResource*) et 5.26 (adaptation du serveur de données en serveur de ressources).

Évolution de la hiérarchie de modèles (Serveurs de données) La modélisation du critère d'extensibilité du serveur a créée des versions de modèles au sein de la hiérarchie de sous-modèles des figures C.1 et C.2. Une version simplifiée est présentée en figure 5.27.
Les fonctionnalités modélisées (**9** : *Acquisition des données*, **12** : *Adaptation en serveur de ressources*) ont généré une nouvelle branche dans la hiérarchie. En effet, comme on peut le voir en figure 5.10, une branche a été créée via l'opérateur *apply*. La version finale de cette branche

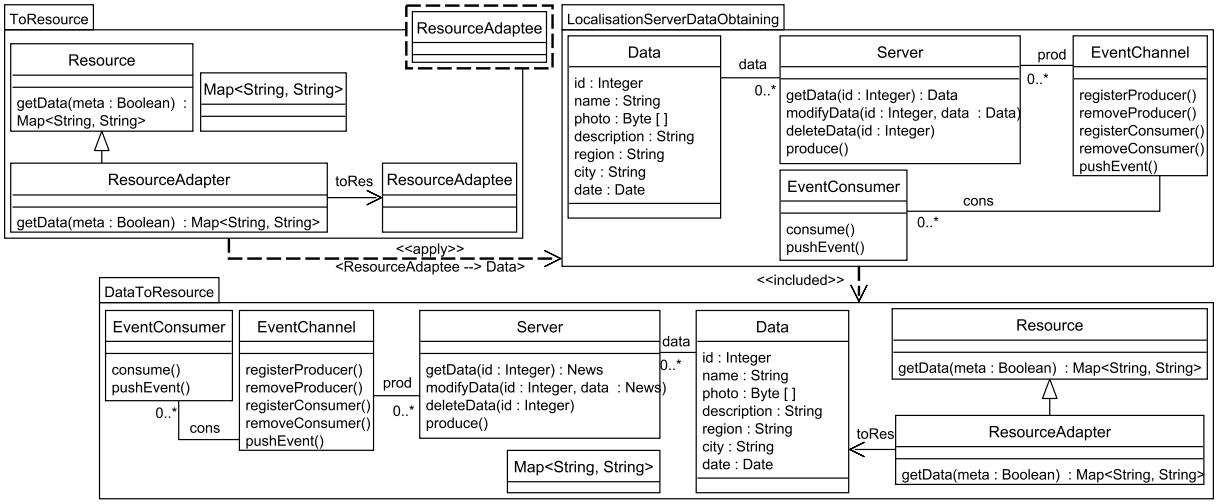


FIGURE 5.25 – Résultat de l'application de "ToResource" sur "LocalisationServerDataObtaining"

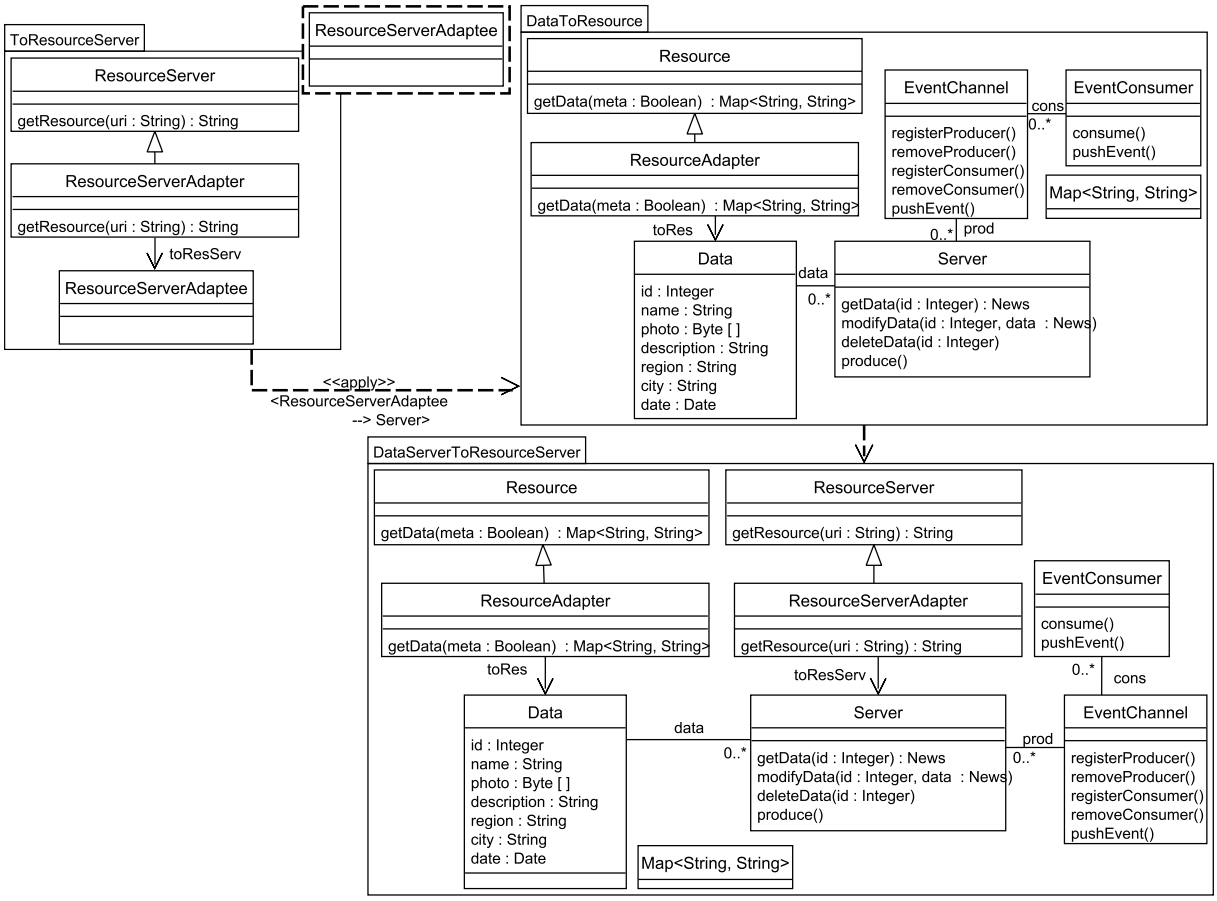


FIGURE 5.26 – Résultat de l'application de "ToResourceServer" sur "DataToResource"

DataServerToResourceServer est une version alternative de *LocalisationServer*.

Fusion des fonctionnalités Suite à la modélisation des trois critères auxquels doit répondre le serveur (Sécurité, performance et extensibilité), le concepteur *CS2* fusionne les versions de

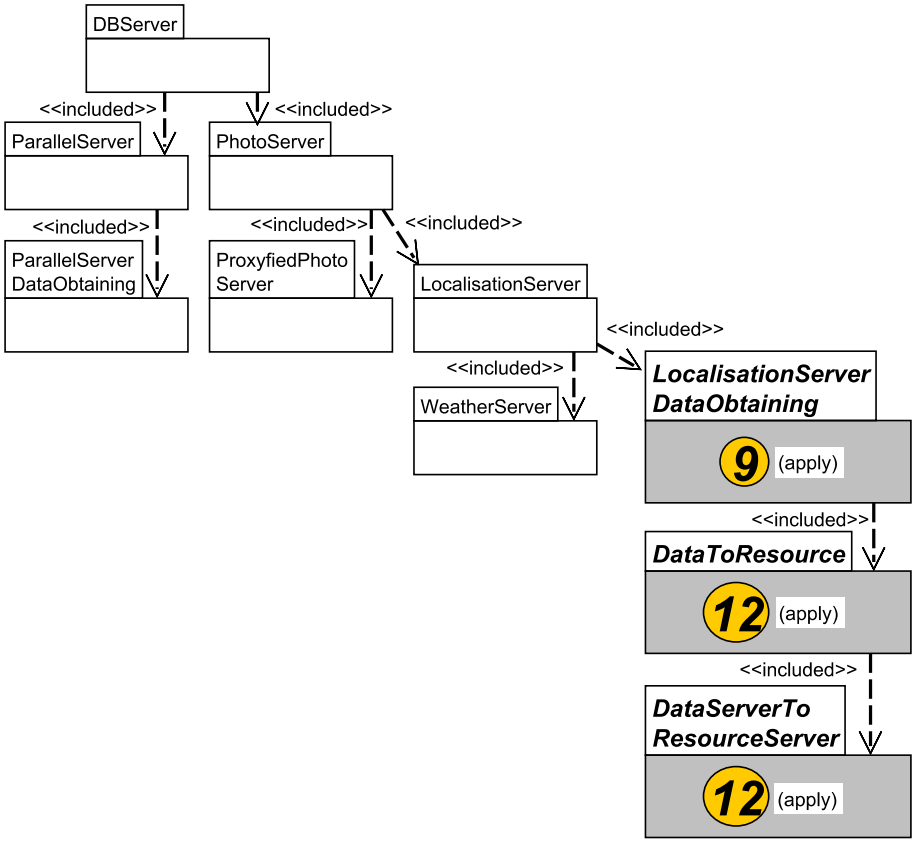


FIGURE 5.27 – Serveur de ressources : Hiérarchie de modèles après la modélisation du critère d'extensibilité du serveur

modèles résultantes de chacune des étapes de modélisation correspondantes, tel que présenté en figure 5.28 avec l'utilisation de l'opérateur *merge*.

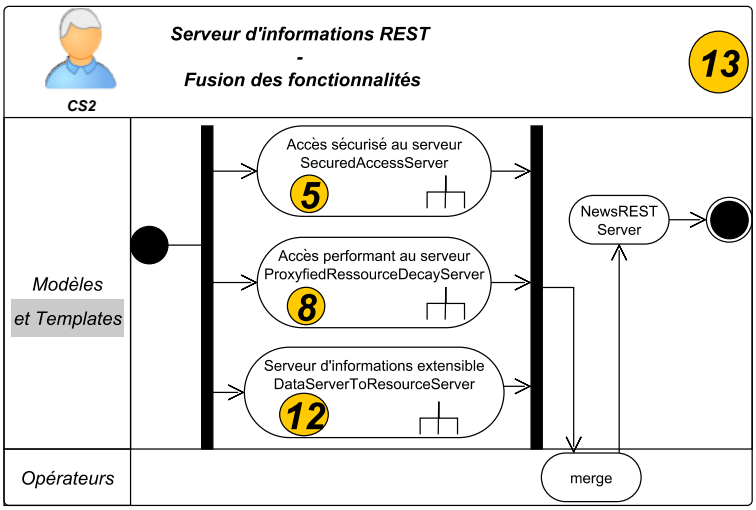


FIGURE 5.28 – Serveur REST : Fusion de l'ensemble des fonctionnalités modélisées

Ainsi, une branche de la hiérarchie de modèles de serveurs de données a été fusionnée avec deux

branches de la hiérarchie de modèles de serveurs de ressources. La figure 5.29 illustre cette fusion, permettant d'obtenir la version finale du serveur REST d'informations, exposée en figure 5.30.

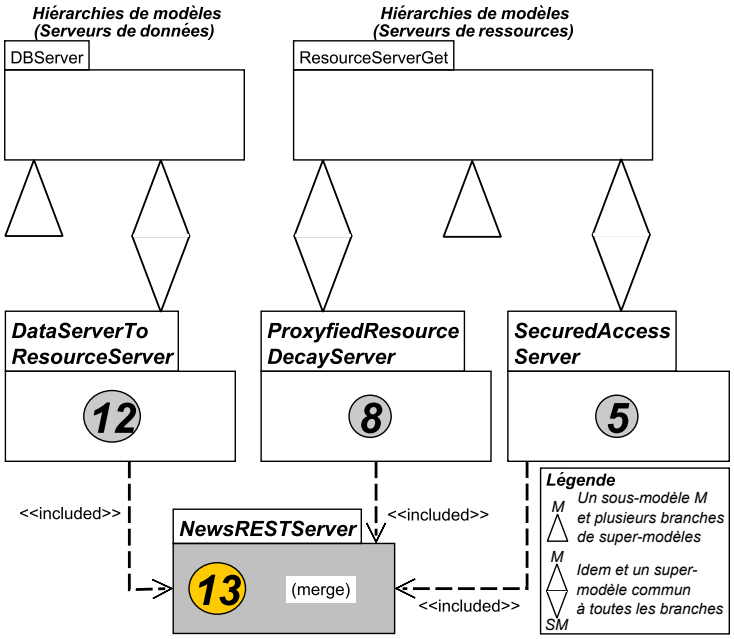


FIGURE 5.29 – Version finale du serveur REST : Fusion de trois branches

5.6 Seconde partie : Concepteurs de Templates

Pour rappel (cf. chapitre 3), le but des concepteurs de templates est de modéliser des templates qui sont ensuite appliqués à des modèles par les concepteurs de systèmes afin de construire un modèle de système. Pour concevoir des templates, les concepteurs de *templates* utilisent les opérateurs de *1 - Composition*, avec *instantiate (instantiation partielle)*, de *2 - Construction*, avec *merge et instantiate (totale)* et de *3 - Paramétrisation*, avec *promote*.
L'étape de modélisation *Accès sécurisé au serveur* est présentée en sous-section 5.6.1 et l'étape *Extensibilité du serveur* en sous-section 5.6.2.

5.6.1 Accès sécurisé au serveur de ressources

L'équipe décide de sécuriser l'accès au serveur par l'utilisation d'un serveur mandataire, restreignant cet accès en lecture seulement (*i.e.* ces ressources du serveur ne doivent pouvoir être modifiées). L'un des concepteurs de *templates* adapte donc le *template Proxy* au contexte concerné, *i.e. un serveur de ressources*. De plus, ce serveur mandataire devra par la suite être étendu par les modeleurs d'applications : un autre concepteur adapte donc le *template Decorator* au contexte *accès protégé à une ressource*.

Adaptation du pattern Proxy au contexte Serveur de ressources Le pattern *Proxy* est adapté au contexte, à savoir *un serveur mandataire restreignant l'accès aux ressources en lecture*. La figure 5.31 décrit cette adaptation qui est effectuée par le concepteur de *templates CT1*. Pour rappel (cf. section 5.4), chacun des concepteurs de template sont identifiés dans le scénario par le label "C(*oncepteur de*) T(*emplate*)", suivit d'un identifiant.

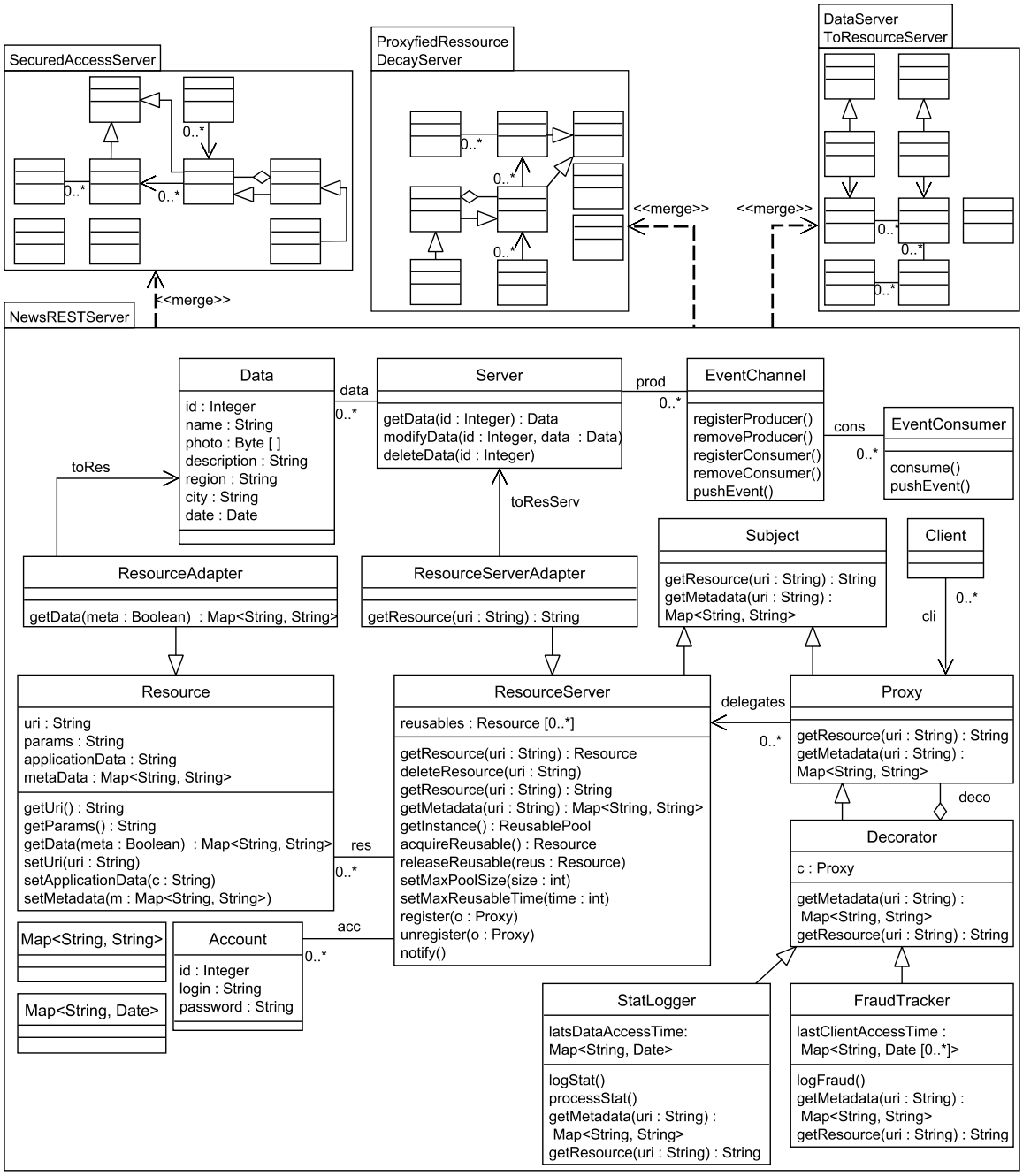


FIGURE 5.30 – Résultat de la fusion de l'ensemble des fonctionnalités

Le concepteur sélectionne les méthodes d'accès en lecture à la ressource et à ses méta-données dans le modèle *ResourceServerGetDelete* (cf. figure C.1). Il instanci ensuite le pattern *Proxy* avec *instantiate* (deux instanciations : une pour l'accès aux données de la ressource, une autre pour l'accès à ses méta-données - cf. figure 5.32) et fusionne les deux instances *ProxyfiedResourceAccess* et *ProxyfiedMetadataAccess* avec *merge* (figure 5.33).

Enfin, le concepteur utilise l'opérateur *promote* afin de mettre en paramètre le serveur de ressources (*ResourceServer*) auquel le serveur mandataire restreindra l'accès.

Cette promotion de *ResourceServer* est décrite dans la partie droite de la figure 5.31 : en partant

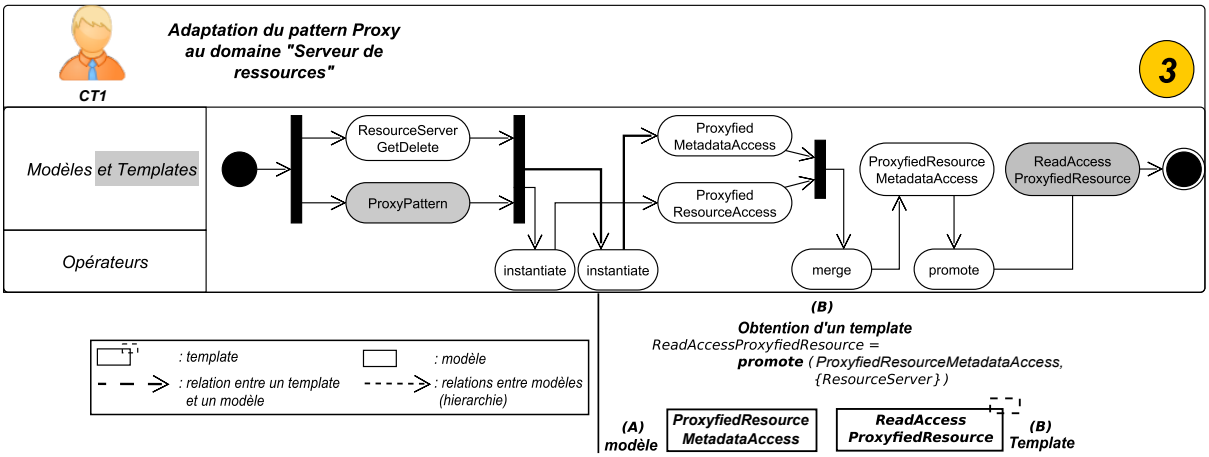


FIGURE 5.31 – Adaptation du pattern Proxy au contexte *Serveur de ressources*

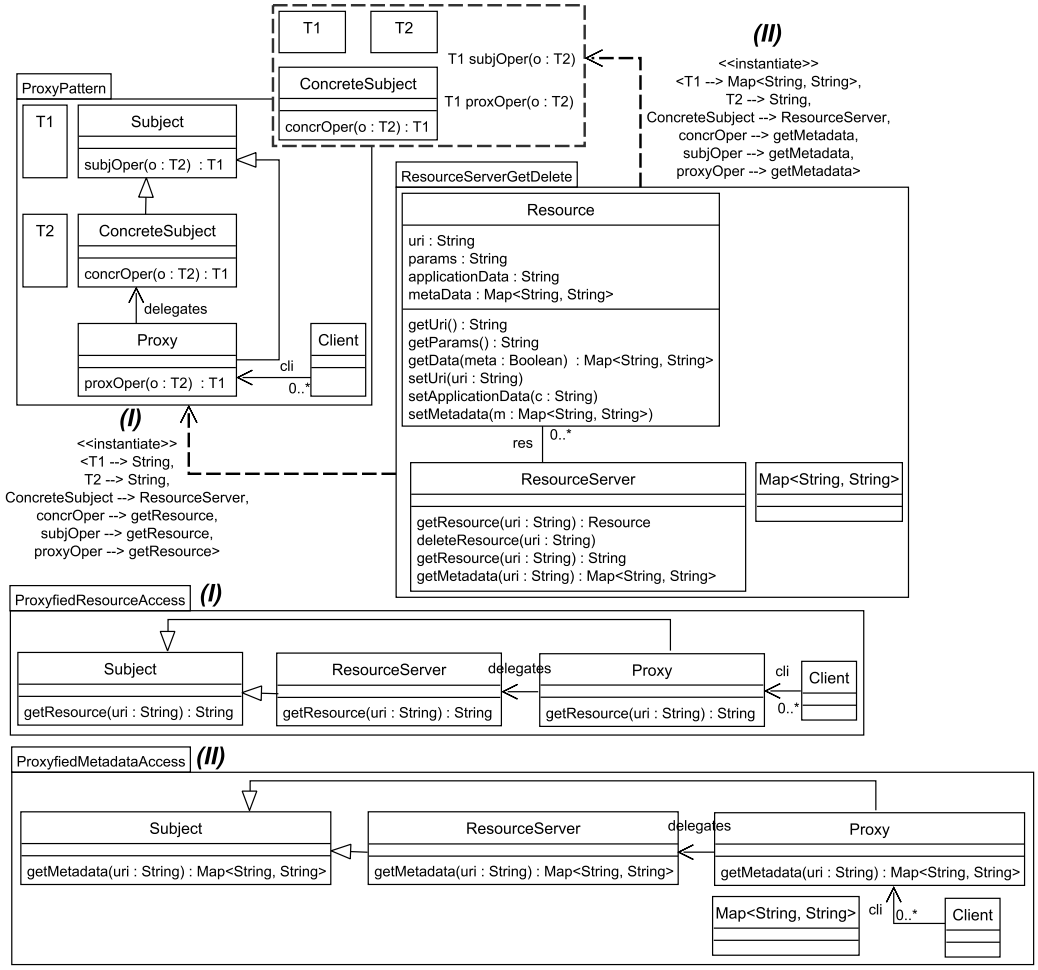


FIGURE 5.32 – Résultat des instanciations de "ProxyPattern" avec "ResourceServerGetDelete"

d'une version de modèle existante (*A* sur la figure), le concepteur a passé en paramètre de l'opérateur *promote* (*B* sur la figure) le modèle *ProxyfiedResourceMetadataAccess* et les fonctionnalités de ce dernier qu'il souhaitait mettre en paramètre, obtenant ainsi le *template ReadAccessProxyfiedResource*. Le résultat de ce paramétrage est présenté dans la figure 5.34.

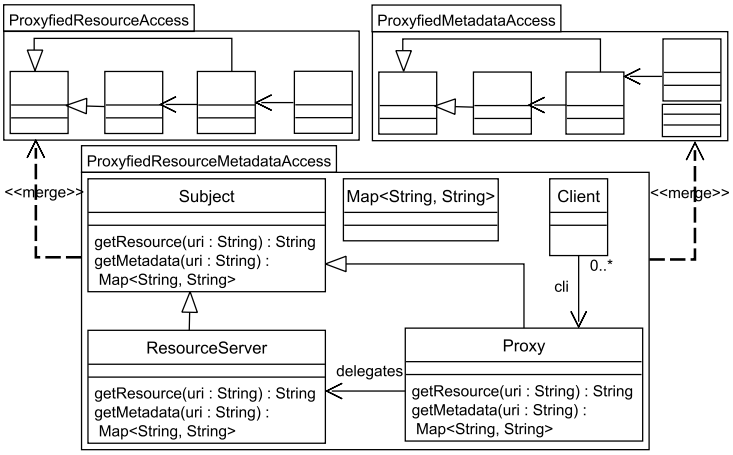


FIGURE 5.33 – Résultat de la fusion des *templates* partiellement contextualisés

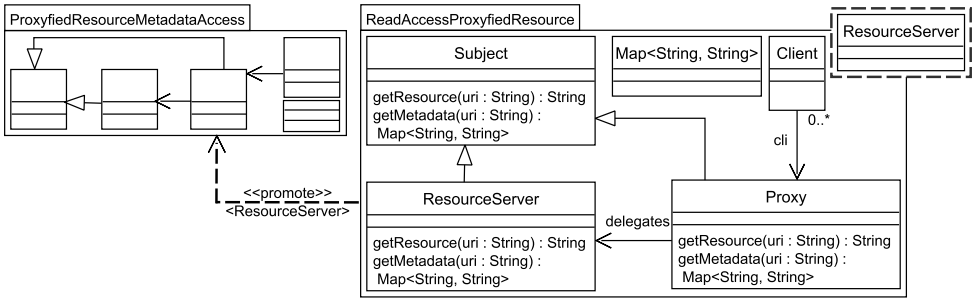


FIGURE 5.34 – Résultat de la promotion de paramètres dans *ReadAccessProxyfiedResource*

Adaptation du pattern Decorator au contexte "Accès restreint à une ressource" Le concepteur de *templates* adapte ensuite le pattern *Decorator* au contexte souhaité, comme présenté en figure 5.35.

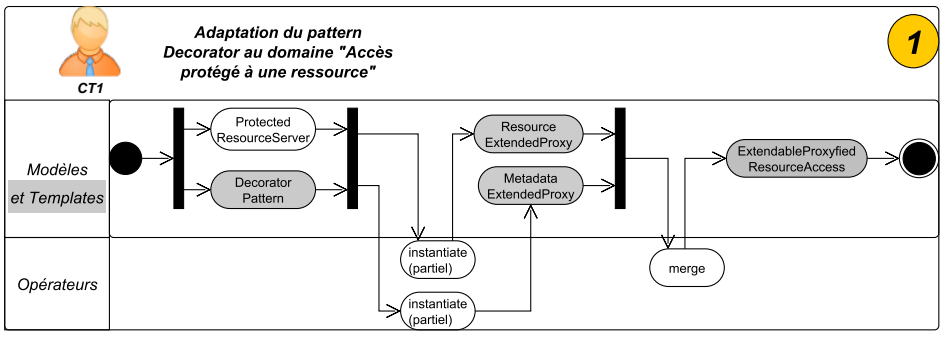


FIGURE 5.35 – Adaptation du pattern Decorator au contexte *Accès restreint à une ressource*

De la même façon qu'avec le pattern *Proxy* précédent, le concepteur sélectionne le serveur mandataire et les méthodes d'accès en lecture aux ressource et à leurs méta-données mais cette fois-ci dans le modèle *ProtectedResourceServer* (cf. figure C.1). Le concepteur effectue deux instanciations partielles afin d'obtenir les méthodes *getResource* et *getMetadata* pour le serveur mandataire. Le résultat de ces deux instanciations est présenté dans la figure 5.36.

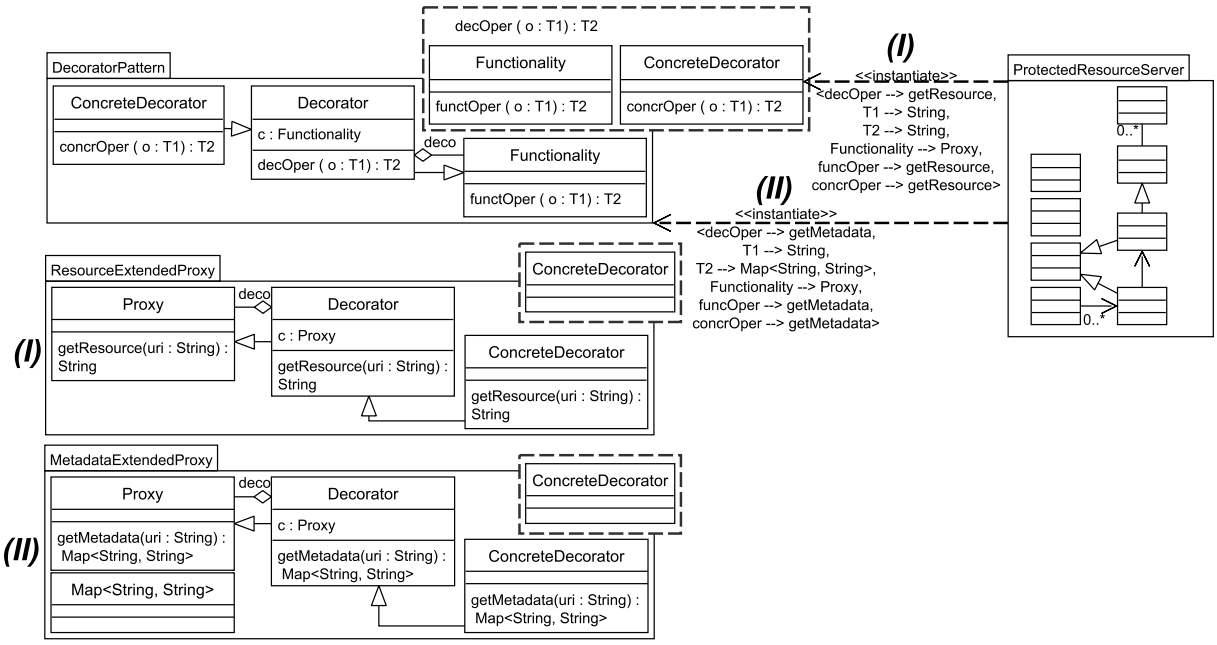


FIGURE 5.36 – Résultat des instanciations de *DecoratorPattern* avec *ProtectedResourceServer*

Avec *merge*, les deux *templates* résultats *ResourceExtendedProxy* et *MetadataExtendedProxy* sont fusionnés, permettant d'obtenir un *template* (*ExtendableProxyfiedResourceAccess*) de serveur mandataire extensible et restreignant l'accès aux ressources. Le résultat de cette fusion est exposé dans la figure 5.37.

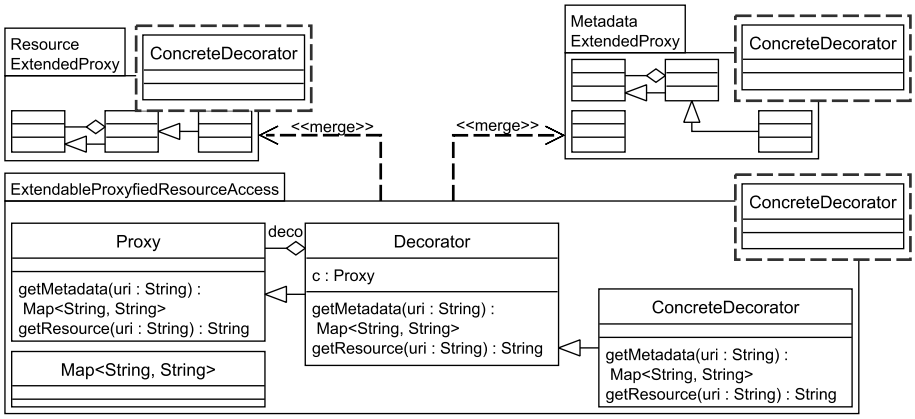


FIGURE 5.37 – Résultat de la fusion des *templates* partiellement contextualisés

5.6.2 Extensibilité du serveur et accès aux informations

Afin de rendre le serveur de ressources extensible, les concepteurs de *templates* décident d'utiliser le pattern *Adapter* : tout serveur modélisé et lié au serveur de ressources via ce pattern pourra être utilisé comme un serveur de ressources. Les concepteurs adaptent donc ce pattern aux contextes *Ressources* et *Serveur de ressources*.

Adaptation du pattern Adapter au contexte Ressources Afin d'effectuer cette adaptation

du *template*, le concepteur *CT2* utilise les opérateurs présentés en figure 5.38.

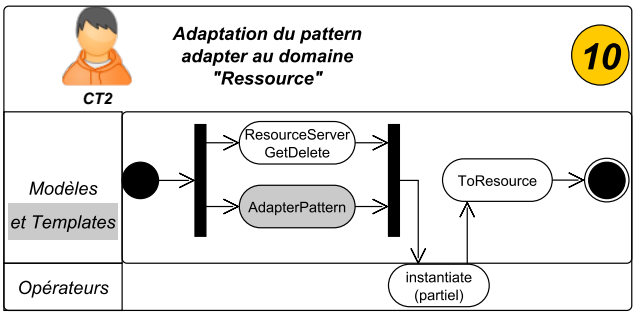


FIGURE 5.38 – Adaptation du pattern Adapter au contexte *Ressources*

Les ressources sont sélectionnées dans le modèle *ResourceServerGetDelete*, puis *AdapterPattern* est instancié avec celui-ci. Cependant, la notion d'adaptateur, représentée par la classe *Adapter* et l'association *adapts*, sont par la suite modifiés pour en faire un adaptateur de serveur de ressources (*ResourceServerAdapter* et *toRes*) : cette contextualisation ne peut se faire cette fois-ci à l'aide d'un opérateur comme l'*instantiate*, puisque le concept d'adaptateur de serveur de ressources est absent de toute version de modèles composant la hiérarchie de serveur de ressources (cf. annexe C). Le modèle résultat est exposé dans la figure 5.39.

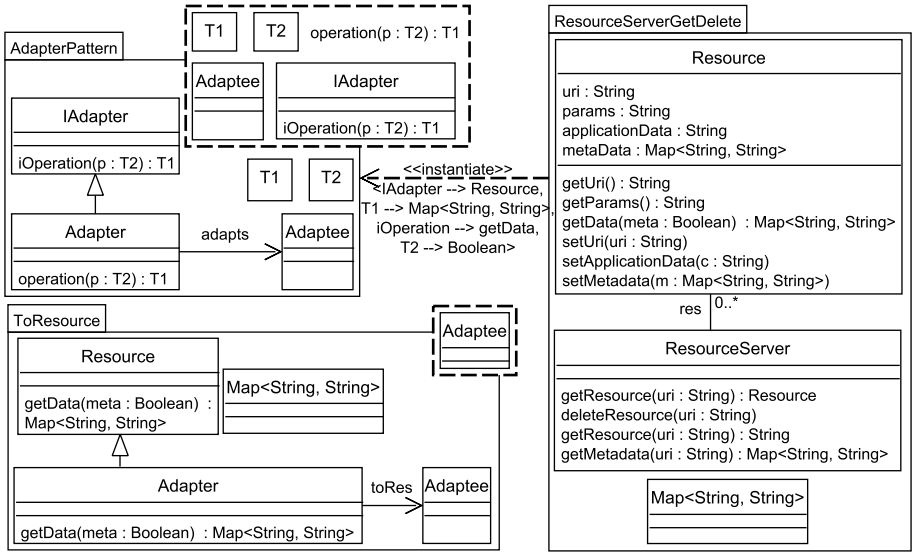


FIGURE 5.39 – Résultat de l'instanciation partielle d'*AdapterPattern* avec *ResourceServerGetDelete*

Adaptation du pattern Adapter au contexte Serveur de ressources Les concepteurs de *templates* adaptent donc le pattern *Adaptateur* au contexte, à savoir *un serveur de ressource*. La figure 5.40 décrit l'utilisation de l'opérateur *instantiate* par le concepteur de *templates CT2*.
La démarche est identique à celle décrite pour l'adaptation du *template* au contexte *Ressources*, décrite précédemment. Le modèle résultat de cette instanciation partielle est présenté dans la figure 5.41.

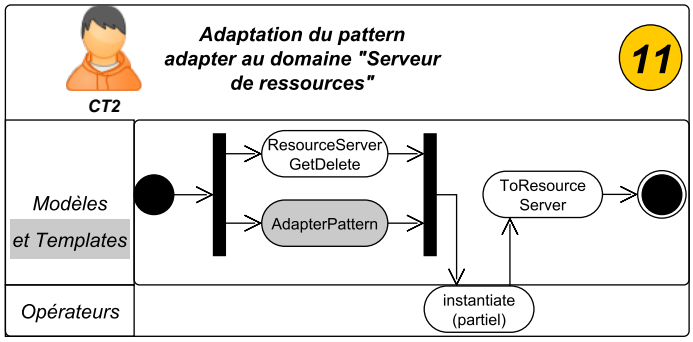


FIGURE 5.40 – Adaptation du pattern Adapter au contexte *Serveur de ressource*

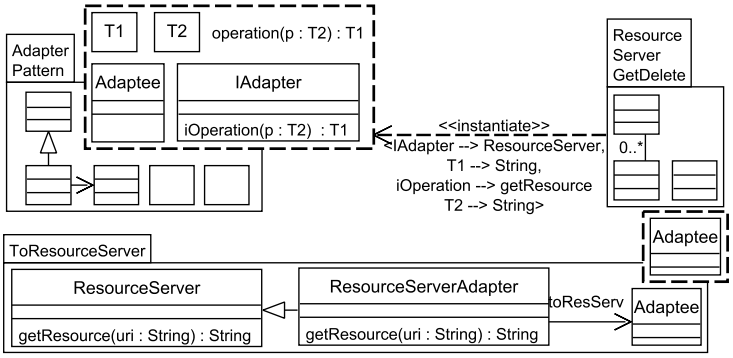


FIGURE 5.41 – Résultat de l'instanciation partielle d' *AdapterPattern* avec *ResourceServerGetDelete*

5.7 Bilan

Ce cas concret de modélisation nous a permis de valider l'IDMBT et ses différents constituants. Tout d'abord, l'application souligne l'intérêt des rôles d'acteurs, définis en section 3.2. En effet, il met en avant la modélisation en équipe du serveur par les concepteurs de templates et les concepteurs de systèmes. On observe notamment cette collaboration au travers de l'utilisation, par les concepteurs de système, des templates construits par les concepteurs de templates.
Cette application met aussi en relief la prise en compte des hiérarchies de modèles, particulièrement lorsque celles-ci sont issues de versioning ou de conception incrémentale. Les concepteurs ont pu effectivement rechercher un modèle qu'ils souhaitaient réutiliser en accédant aux diverses étapes, ou "versions", de projets passés ou de celui en cours d'élaboration.
Enfin, l'intérêt des opérateurs et leur combinaison a été montré dans plusieurs étapes du processus de modélisation du serveur. On peut notamment citer l'instanciation, présentée en section 3.3. Celle-ci a été mise en valeur lors de la construction des templates partiellement contextualisés par les concepteurs de templates. L'instanciation leur a permis d'adapter les templates à un contexte applicatif particulier, en l'occurrence celui des concepteurs de systèmes.

Chapitre 6

Conclusion et perspectives

6.1 Bilan

Afin de faciliter la réutilisation de modèles, nous avons proposé (cf. section 3.2) une approche basée sur les templates, nommée "Ingénierie Dirigée par les Modèles Basée sur les Templates (IDMBT - Allon et al. [7])". Cette approche contribue aux problématiques de réutilisation de modèles. Dans l'état de l'art, certains auteurs ont étudié ces problématiques en considérant des parties de modèles et d'autres en utilisant des modèles paramétrés. Notre approche permet de produire et de combiner ces deux formes de modèles. Elle contribue ainsi à deux espaces de conception : *la conception pour la réutilisation* et *celle par la réutilisation*. L'IDMBT facilite la conception pour la réutilisation en proposant des activités de *paramétrisation, composition, décomposition, induction et recherche de templates*. En ce qui concerne la conception par la réutilisation, notre approche propose des activités de *construction de modèles, de détection et d'extraction de templates*, ainsi que *de recherche de modèles guidée par les templates*. Pour les modèles paramétrés, l'IDMBT s'appuie sur la notion d'"aspectual template" (Vanwormhoudt et al. [103]) qui garantit que le paramétrage forme un modèle de plein droit. Ces derniers étendent les templates du standard UML en ajoutant des contraintes portant sur le paramétrage et le binding.
Nous avons contribué à l'IDMBT en approfondissant trois axes. Tout d'abord, en section 3.3, nous avons étudié l'instanciation totale et partielle de templates UML (Allon et al. [8]). Cette opération d'instanciation permet de produire un modèle à partir d'un template par l'extraction d'une partie du contexte applicatif. Dans l'état de l'art, l'instanciation est souvent intégrée dans l'application d'un template à un contexte applicatif et rarement isolée comme opération à part entière. C'est le cas notamment pour la relation *bind* en UML. L'examen approfondi de cette relation nous a permis de dégager le résultat important suivant :

$$\textit{Template binding} = \textit{instanciation} + \textit{merge}$$

Ce résultat isole l'instance, permettant ainsi de réutiliser cette dernière par la suite. À partir de ce résultat, nous avons proposé l'instanciation totale et l'instanciation partielle de templates comme nouveaux modes de réutilisation. Dans la continuité du binding partiel du standard UML, l'instanciation partielle permet de ne retenir qu'une partie des paramètres à substituer. Le résultat est alors un template qui peut être à son tour réinstancié avec d'autres contextes applicatifs. Il est donc possible d'effectuer des séquences d'instanciations. Dans l'IDMBT, l'instanciation joue un rôle important car elle facilite la définition d'autres opérateurs. Dans le deuxième axe, nous avons proposé les opérateurs de détection et de suppression de templates dans un modèle (3.4) qui s'appuient sur l'instanciation. Le premier ("getBoundSubstitutions") permet de comprendre la structure d'un modèle et de vérifier que celui-ci est conforme à des règles de conception de projets. Pour cela, l'opérateur détecte la présence de l'instance d'un template dans un modèle. L'opérateur de suppression d'instances de templates ("unbind") répond à des besoins d'évolution guidée par les templates, tels que la simplification de modèles ou le remplacement d'une instance

de template par une autre.
En dernier point, nous avons étudié l'application de templates sur des hiérarchies de modèles (section 3.5) qui se rencontrent fréquemment dans des usages comme le versionnement, la conception incrémentale et la modélisation en équipe. Les problèmes posés sont de déterminer sur quel(s) modèle(s) d'une hiérarchie, une application de template est valide et comment se comportent les résultats d'une application vis-à-vis de l'inclusion de modèles. Dans l'état de l'art, aucune approche n'a étudié conjointement les hiérarchies de modèles et l'application des templates. Nous avons défini un ensemble de règles de validité d'application ainsi que des règles qui spécifient les relations entre les résultats d'une application sur une hiérarchie. Ces règles peuvent être utilisées afin de réduire l'effort de modélisation, notamment lors de la modélisation en équipe d'un système.
Afin d'appliquer l'IDMBT, nous avons défini dans le chapitre 4, des opérateurs (Allon [6]), leur mise en œuvre dans une technologie réutilisable ainsi que l'application de cette technologie à un prototype d'atelier [1]. L'intérêt de cette implémentation est de montrer que les concepts de l'IDMBT peuvent être intégrés à des outils de modélisation comme nous l'avons exposé pour l'environnement Eclipse et son framework EMF. Les autres outils pouvant tirer profit des notions que nous avons présentées sont, par exemple, des validateurs de modèles ou encore des moteurs de transformation ou de construction de modèles.
Dans le cadre de notre approche, une bibliothèque d'aspectual templates (Muller et al. [74]) représentant les patrons de conception du GOF (Gamma et al. [50]) a été élaborée. Ces templates peuvent être réutilisés dans diverses situations de modélisation. L'ensemble de notre approche et de ses résultats ainsi que le bibliothèque ont ensuite été appliqués à la conception d'un serveur REST d'agrégation d'informations, à partir d'un dépôt de modèles hiérarchisés et de templates.

6.2 Perspectives

À partir des résultats exposés précédemment, plusieurs perspectives de recherche peuvent être envisagées.
De la même manière que nous avons étudié l'application d'un template sur un modèle et les hiérarchies de modèles générées, il serait utile d'examiner la composition de templates et plus généralement la possibilité de leur hiérarchisation par inclusion. Plusieurs questions se posent alors. Premièrement, selon quels critères peut-on établir une hiérarchie de templates ? Une solution envisageable serait d'utiliser les relations d'inclusion définies par Carré et al. [24]. Il est alors nécessaire de considérer la dimension supplémentaire que sont les modèles paramètres. Dans la continuité de l'étude que nous avons effectuée sur la validité de l'application d'un template à une hiérarchie de modèles (section 3.5.1), une étude similaire pourrait être menée sur les hiérarchies de templates.
Dans cette thèse, nous avons privilégié une approche de l'ingénierie qui va des templates vers les modèles. Il nous semble intéressant d'étudier une vision symétrique allant des modèles vers les templates. Tout d'abord, on pourrait envisager la production de nouveaux templates à partir de modèles. Il est alors nécessaire de déterminer comment identifier les constituants communs et variables de ces modèles pour alimenter la structure des templates à induire. Une telle identification repose-t-elle uniquement sur la syntaxe des constituants de modèles et leurs relations ou doit-on aussi considérer la sémantique de ces constituants ? Des approches basées sur l'intersection, la différence et l'union de modèles (Melnik et al. [68], Alanen & Porres [4])

1. http://www.cristal.univ-lille.fr/caramel/MBE_Template/

ainsi que l'analyse formelle de concepts (Amar et al. [10], Al-Msie'deen et al. [2]) peuvent aider à l'étude d'un tel problème. Ensuite, un autre point à étudier serait la compréhension de modèles, notamment lorsqu'il s'agit de grands modèles. Pour mener à bien ce processus de compréhension de modèles, cela nécessite des moyens de "cartographie" de modèles, c'est-à-dire d'identification de structures particulières, telles que des patrons, ainsi que des moyens de traçabilité permettant de représenter le lien entre le modèle étudié et la structure identifiée. La détection de templates présentée en section 3.4 est une aide à l'identification de telles structures. Alliée à cette opération de détection, la relation bind est un moyen de traçabilité entre les modèles et les templates identifiés. Une piste pour assouplir cette détection serait d'exploiter le binding partiel pour finalement induire des variantes de templates qui pourraient s'installer dans les hiérarchies.

Plus généralement, l'organisation de grands espaces de modèles (dépôts) à l'aide des templates est un sujet important. De la même manière que les métamodèles (Carré et al. [25], Lucrédio et al. [67]) et mégamodèles (Favre [44], Bézivin et al. [15], Hebig et al. [53]) aident à structurer de tels espaces, les capacités de structuration des templates sont à explorer.

Annexes

Annexe A
Contraintes définies pour les applications totales

Application totale

Vérification de la bonne formation du modèle paramètre d'un *aspectual template* :

Vérification de la cohérence des associations

– [2] Les classes auxquelles est liée une association en paramètre doivent être aussi des paramètres :
context TemplateSignature **inv** :
self.ownedParameter->forAll(
param : uml : :TemplateParameter |
let pe : uml : : ParameterableElement = param.parameteredElement **in**
pe.oclIsKindOf(uml : :Association) **implies**
let asso : uml : : Association = pe.oclAsType(uml : :Association) **in**
asso .memberEnd->**forAll**(member | member.type.isTemplateParameter())
)

La figure A.1b ne respecte pas ceci puisque *Counter* n'est pas un paramètre.

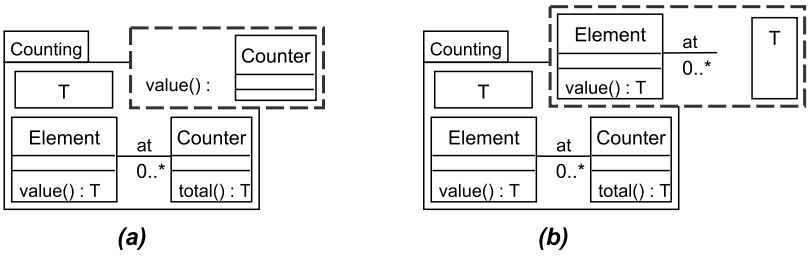


Figure A.1 – Exemples d'*aspectual templates* mal-formés

Vérification du type des opérations et attributs [1]

– [3] Le type d'un attribut paramètre doit aussi être un paramètre :
context TemplateSignature **inv** :
self.ownedParameter->forAll(
param : uml : :TemplateParameter |
let pe : uml : : ParameterableElement = param.parameteredElement **in**
pe.**oclIsKindOf** (uml : : Property) **implies**
pe.oclAsType(uml : : Property) . type . isTemplateParameter()
)

– [4] Les types impliqués dans la signature d'une opération paramètre doivent aussi être des paramètres :
context TemplateSignature **inv** :

1. Ceci ne s'applique pas aux types primitifs.

```
self.ownedParameter->forAll(
    param : uml : :TemplateParameter |
    let pe : uml : :ParameterableElement = param.parameteredElement in
    pe.oclIsKindOf (uml : :Operation) implies
    pe.oclAsType(uml : :Operation).ownedParameter->forAll
    (p : uml : :Parameter | p.type.isTemplateParameter() )
)
```

Par exemple, cette dernière contrainte n'est pas respectée en figure A.1a puisque le type T de l'opération *value()* n'est pas présent dans la signature. La même contrainte est respectée en figure A.1b.

Vérification de la conformité structurelle entre les modèles actuel et paramètre :

Vérification des relations de composition entre paramètres formels et actuels

– [5] Les relations de composition entre les paramètres formels doivent être préservées entre les éléments actuels correspondants :

```
context TemplateBinding inv :
self.parameterSubstitution->forAll (s1, s2 |
    s1.formal.parameteredElement.owner = s2.formal.parameteredElement
    implies s1.actual.owner = s2.actual
)
```

Le *binding* (a) en figure A.2 illustre cette vérification. Le paramètre formel *value()* est contenu par le paramètre formel *Element*. Ainsi, le paramètre actuel associé à *value()* doit être contenu

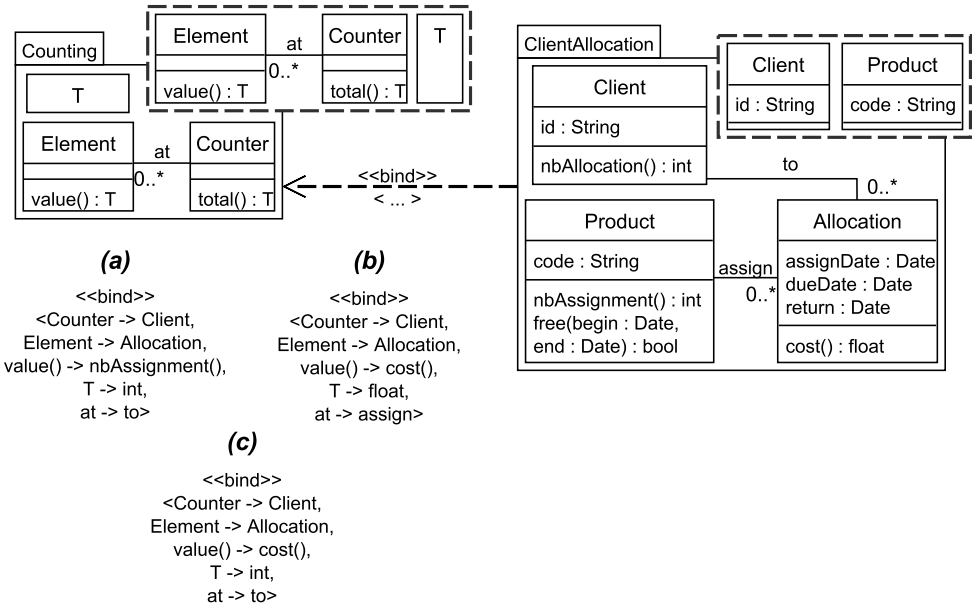


FIGURE A.2 – Exemples de vérification de conformité structurelle des éléments actuels

dans *Allocation* qui est la classe substituée avec *Element*. Puisque *nbAssignment()* est contenu par la classe *Product*, la cinquième contrainte n'est pas respectée.

Vérification de la substitution des paramètres d'une association du modèle paramètre avec les paramètres d'une association du modèle actuel

– [6] Les paramètres d'une association du modèle paramètre doivent être substitués par les paramètres d'une association du modèle actuel :

```
context TemplateBinding inv :
self.parameterSubstitution->forAll (s1, s2 |
    let asso :uml : :ParameterableElement = s1.formal.parameteredElement in
    let cla :uml : :ParameterableElement = s2.formal.parameteredElement in
    (asso.oclIsKindOf (uml : :Association) and cla.oclIsKindOf (uml : :Class)
    and asso.oclAsType(uml : :Association).memberEnd->collect (type)
    ->includes(cla.oclAsType(uml : :Class))
    implies
    s1.actual.oclAsType(uml : :Association).memberEnd->collect(type)
    ->includes(s2.actual.oclAsType(uml : :Class))
)
```

Dans le binding (b) de la figure A.2, les classes liées à l'association *assign* substituée avec *at*, doivent être des classes substituées avec les paramètres *Element* et *Counter*, *i.e. Allocation* et *Product*, au lieu de *Allocation* et *Client*. La sixième contrainte n'est donc pas respectée.

Vérification de la substitution des types Les deux contraintes suivantes se concentrent sur la substitution d'attributs (contrainte 7) et la substitution d'opérations (contrainte 8). Pour un attribut paramètre a, son type doit être substitué avec le type de l'attribut substitué avec a.

– [7] Le type substitué de l'attribut paramètre a doit être substitué avec le type de l'attribut substitué avec a :

```
context TemplateBinding inv :
self.parameterSubstitution->
    select (formal .parameteredElement.oclIsTypeOf(uml : :Property) )->forAll(tps |
    let prop : uml : :Property
    tps.formal.parameteredElement.oclAsType(uml : :Property) in
    let substitutedProp : uml : :Property = tps.actual.oclAsType(uml : :Property in
    self.parameterSubstitution->exists (
    formal.parameteredElement=prop.type and actual=substitutedProp.type)
)
```

Et enfin, la huitième contrainte concerne les opérations : le non-respect de cette contrainte est illustrée avec le *binding* (c) de la figure A.2. Le type du retour de *value()* est T ; puisque T est substitué avec *int*, le type du retour de *cost()* n'est pas compatible : il doit être un flottant au lieu d'un entier. Ainsi, l'opération *value()* ne peut être substitué avec *cost()*.

– [8] Le type substitué de l'opération paramètre o doit être substitué avec le type de l'opération substituée avec o :

```
context TemplateBinding inv :
self.parameterSubstitution->select(formal.parameteredElement.
oclIsTypeOf(uml : :Operation) )->forAll( tps |
let formalOp : uml : :Operation
    tps.formal.parameteredElement.oclAsType(uml : :Operation) in
    let actualOp : uml : :Operation = tps.actual.oclAsType(uml : :Operation) in
    formalOp.ownedParameter->size ()=actualOp.ownedParameter->size() and
```

```
    Sequence1..formalOp.ownedParameter->size ()->forAll( index |
        let memberFormalOp : uml : :Parameter = formalOp.ownedParameter->asOrderedSet()
        ->at(index).oclAsType(uml : :Parameter) in
        let memberActualOp : uml : :Parameter = actualOp.ownedParameter->asOrderedSet()
        ->at(index).oclAsType(uml : :Parameter) in
        self.parameterSubstitution->exists (formal.parameteredElement=memberFormalOp.type and
        actual=memberActualOp.type)
    )
)
```

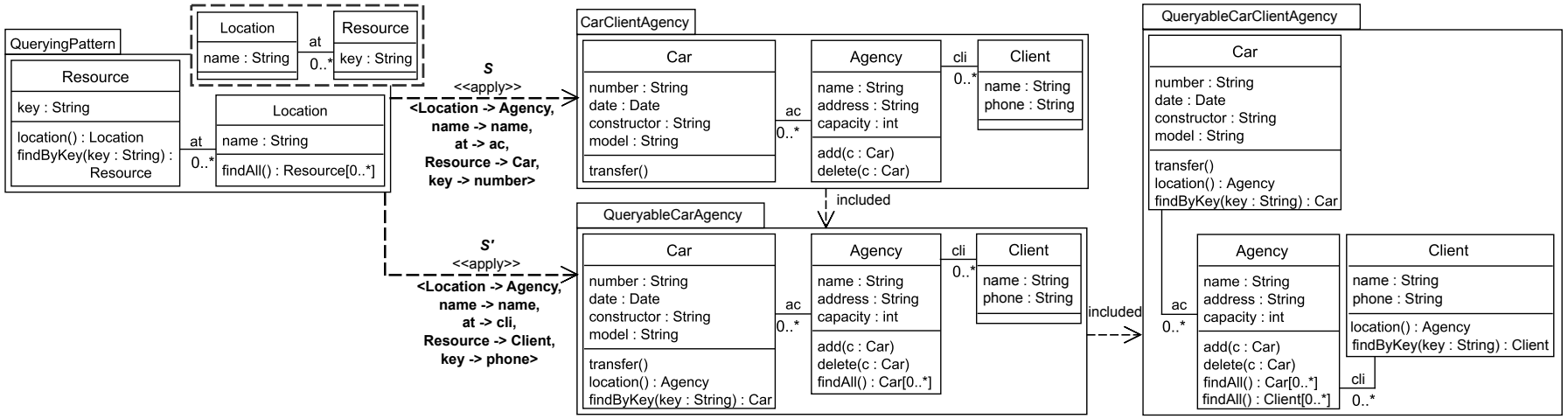


FIGURE B.1 – Première séquence d'applications possible de *QueryingPattern*

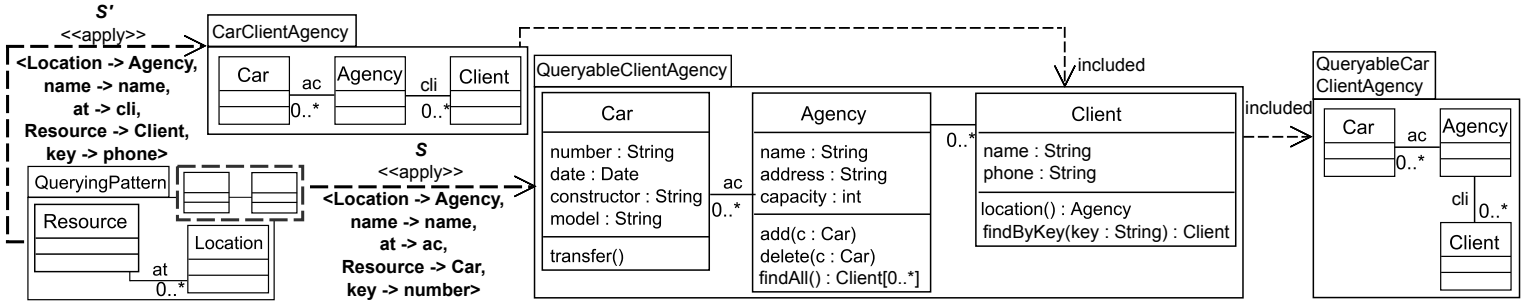


FIGURE B.2 – Seconde séquence d'applications possible de *QueryingPattern*

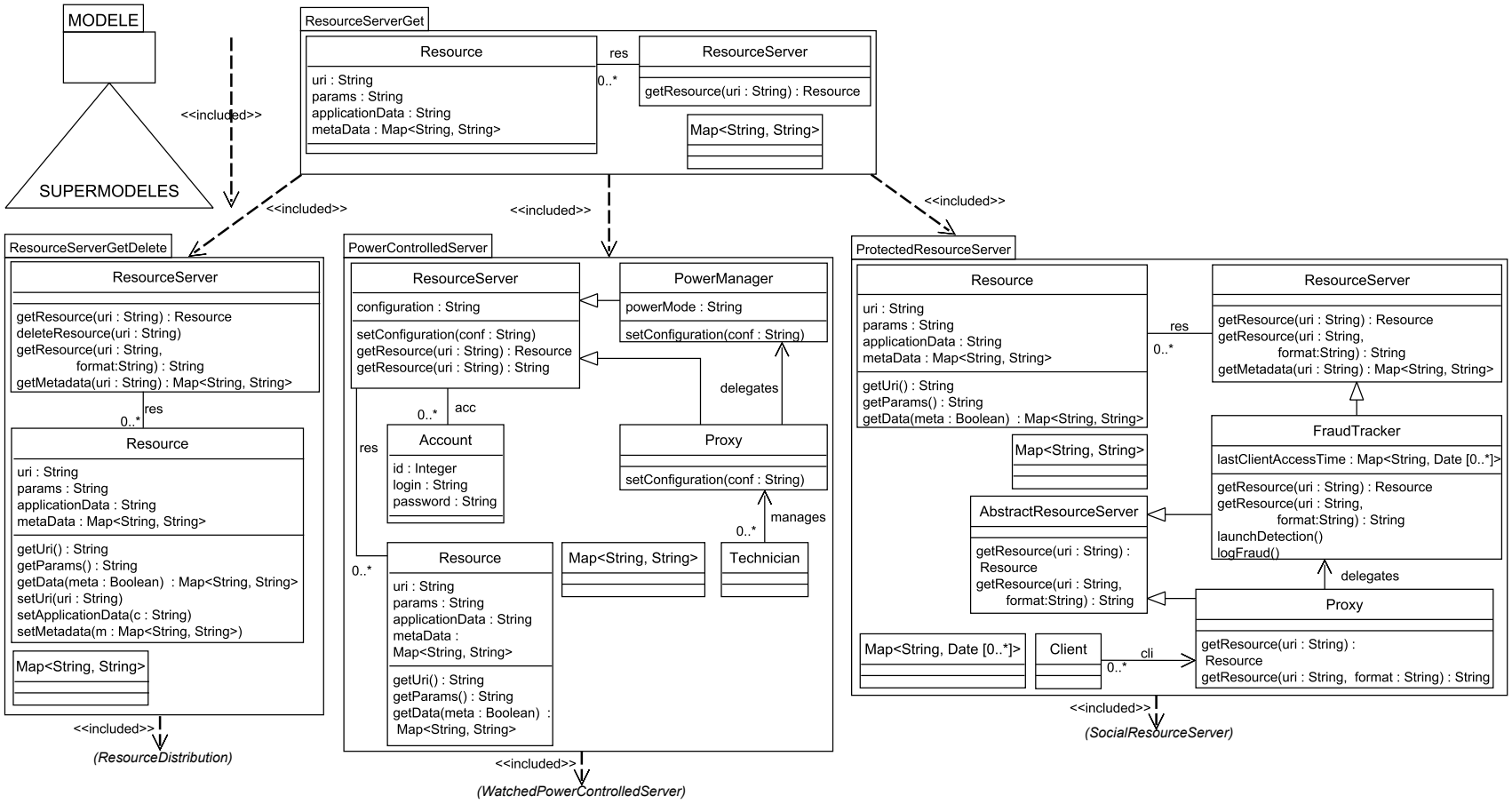


Figure C.1 – Serveurs de ressources - 1\2

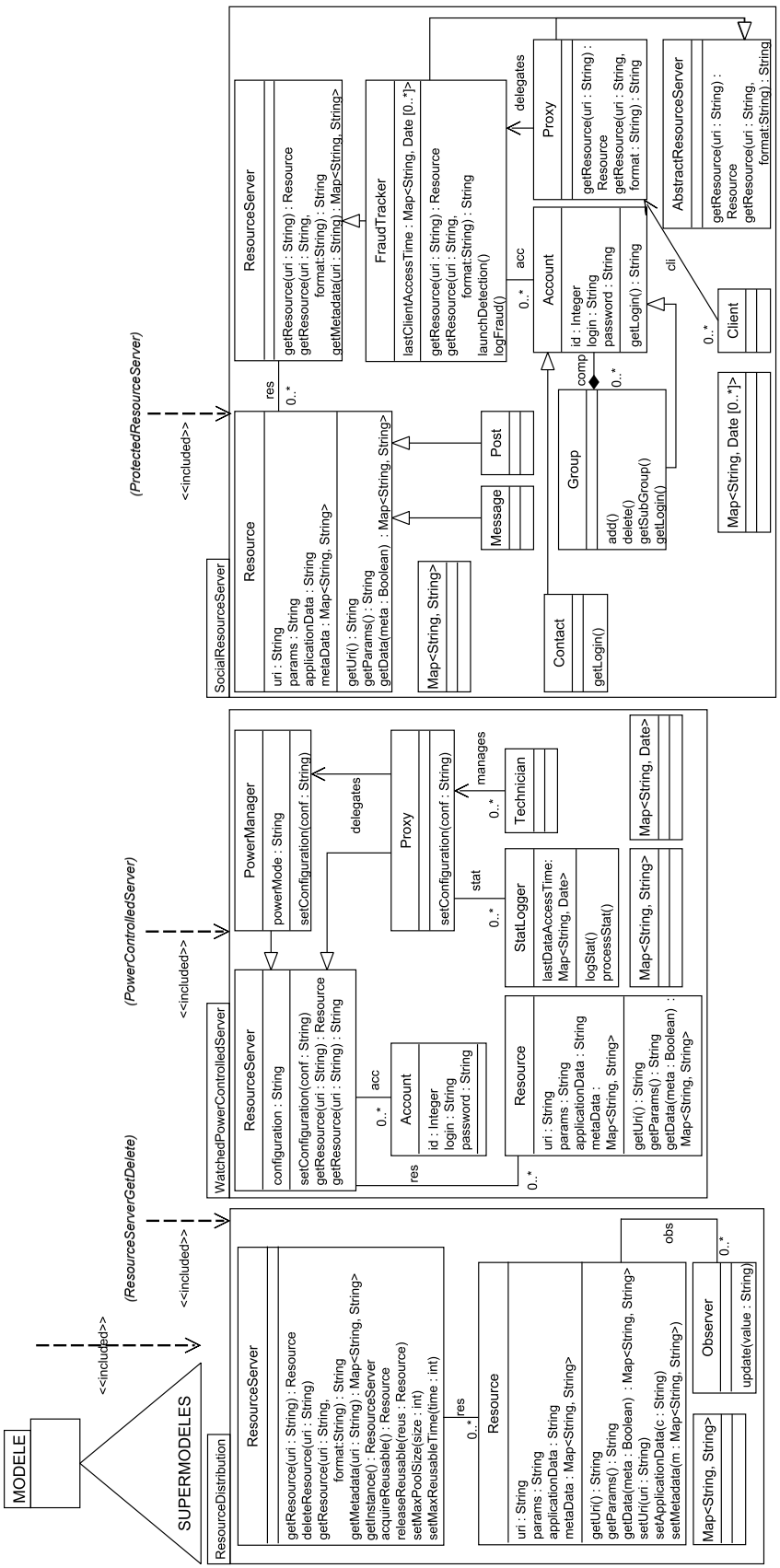


FIGURE C.2 – Serveurs de ressources - 2\2

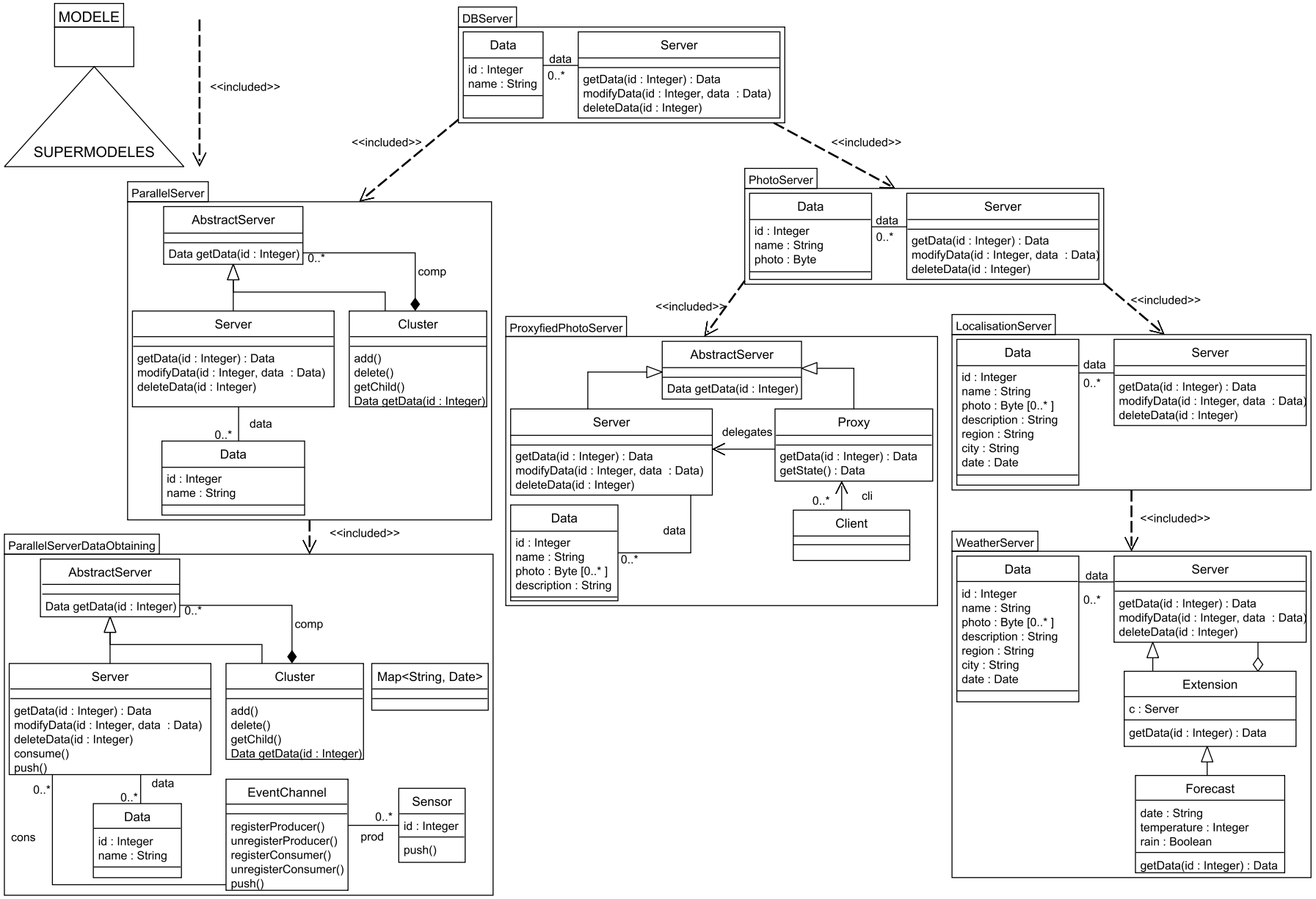


FIGURE C.3 – Serveurs de BDD

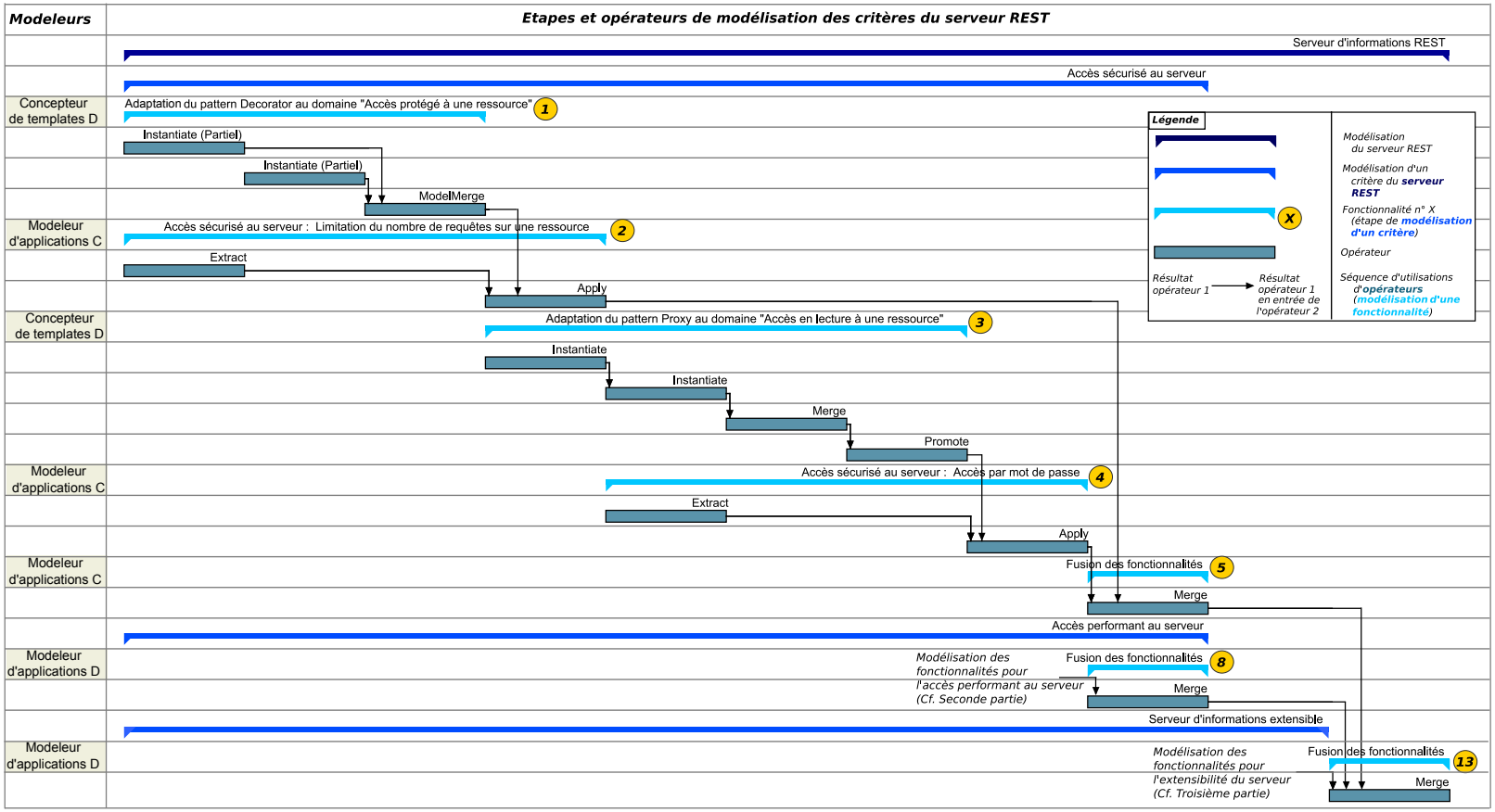


Figure D.1 – Modélisation de l'accès sécurisé au serveur

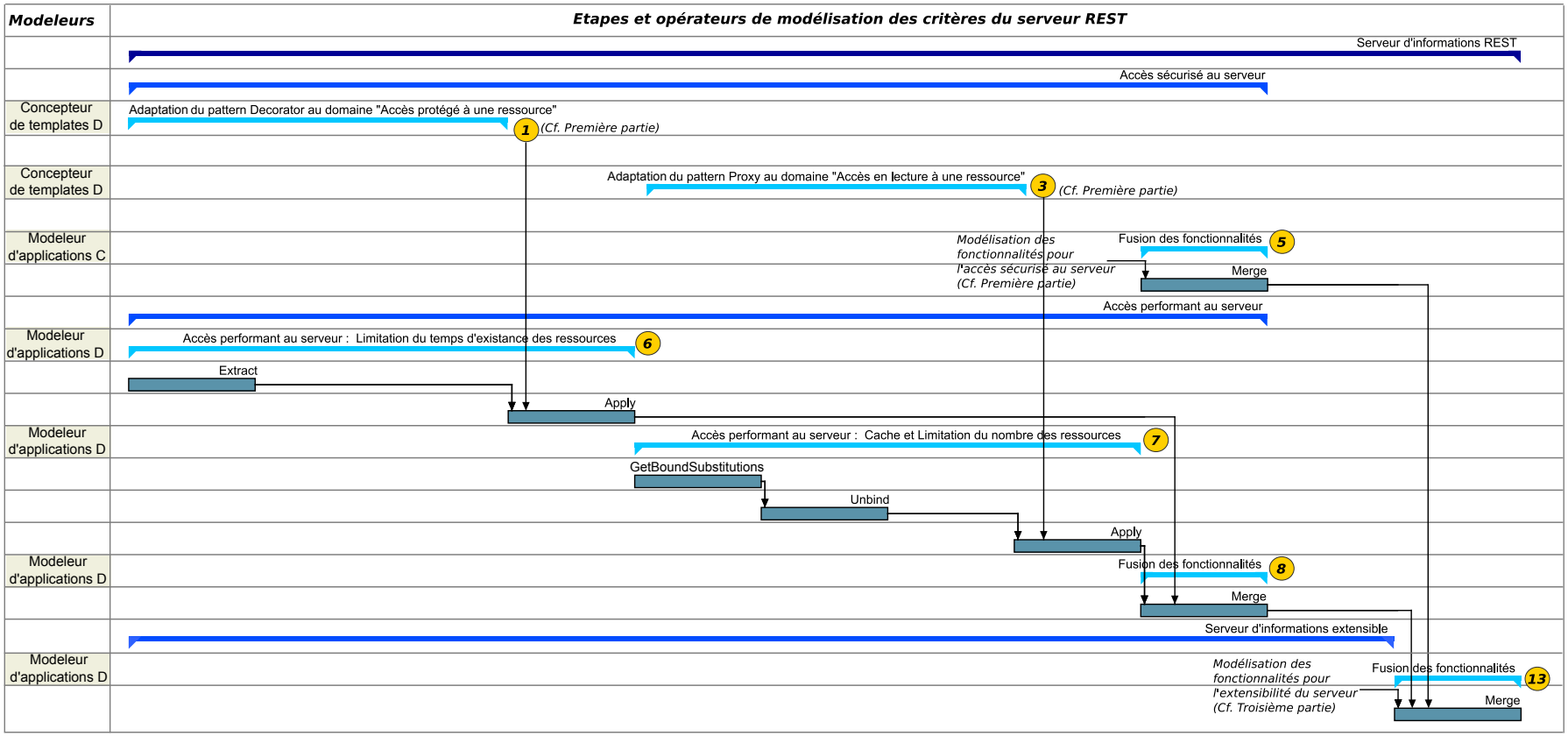


FIGURE D.2 – Modélisation de l'accès performant au serveur

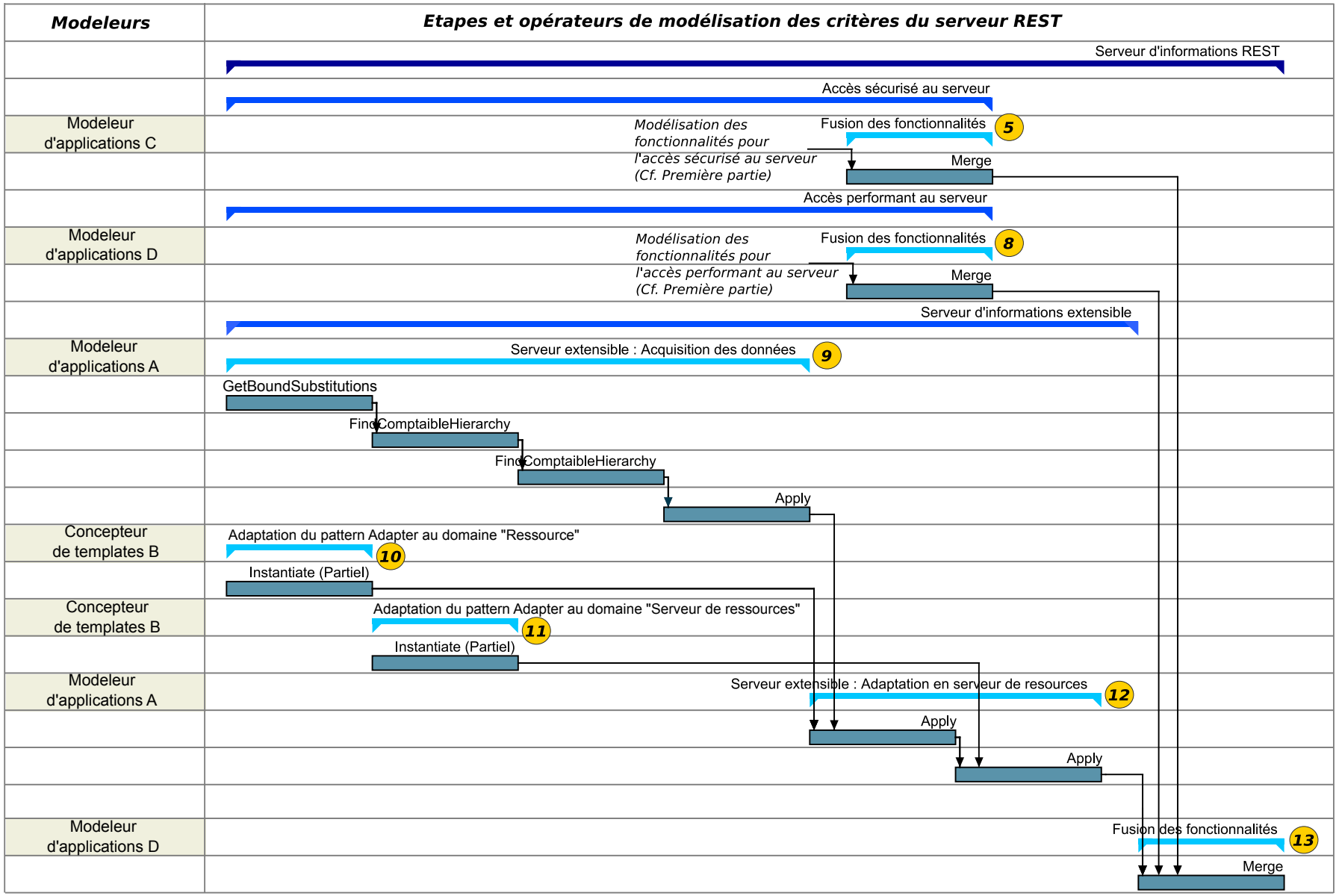


FIGURE D.3 – Modélisation de l'extensibilité du serveur et de l'accès aux informations

Bibliographie

[1] Acher, M., Collet, P., Lahire, P., & France, R. B. (2011). Slicing feature models. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011* (pp. 424–427). (Cité en page 7.)

[2] Al-Msie'deen, R., Huchard, M., Seriai, A.-D., Urtado, C., Vauttier, S., & Al-Khlifat, A. (2014). Concept lattices : a representation space to structure software variability. In *5th International Conference in Information and Communication Systems (ICICS)* (pp. 1–6). : IEEE. (Cité en page 153.)

[3] Alam, O., Kienzle, J., & Mussbacher, G. (2013). Concern-oriented software design. In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings* (pp. 604–621). (Cité en page 79.)

[4] Alanen, M. & Porres, I. (2003). Difference and union of models. In *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings* (pp. 2–17). (Cité en pages 11, 12, 152 et 175.)

[5] Albin-Amiot, H., Cointe, P., Guéhéneuc, Y., & Jussien, N. (2001). Instantiating and detecting design patterns : Putting bits and pieces together. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA* (pp. 166–173). (Cité en page 20.)

[6] Allon, M. (2016). Operators for template-based MDE. In *Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016.* (pp. 80–86). (Cité en page 152.)

[7] Allon, M., Vanwormhoudt, G., Carré, B., & Caron, O. (2015). Template Based MDE. In *4ème Conférence nationale en IngénieriE du Logiciel* Bordeaux, France. (Cité en page 151.)

[8] Allon, M., Vanwormhoudt, G., Carré, B., & Caron, O. (2016). Isolating and reusing template instances in UML. In *Modelling Foundations and Applications - 12th European Conference, ECMFA 2016, Held as Part of STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings* (pp. 173–187). (Cité en pages 2, 69 et 151.)

[9] Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., & Wimmer, M. (2009). Why model versioning research is needed ! ? an experience report. In *Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS*, volume 9 (pp. 1–12). (Cité en pages 1 et 5.)

[10] Amar, B., Guédi, A. O., Miralles, A., Huchard, M., Libourel, T., & Nebut, C. (2012). Finding semi-automatically a greatest common model thanks to formal concept analysis. In *International Conference on Enterprise Information Systems* (pp. 72–91). : Springer. (Cité en page 153.)

[11] Ambrosio, A. P. & Ruggia, R. (1994). Applying reuse to conceptual modelling : an analysis of existing techniques. *AFCET*. (Cité en page 1.)

[12] Bae, J. H., Lee, K., & Chae, H. S. (2008). Modularization of the UML metamodel using model slicing. In *Fifth International Conference on Information Technology : New Generations (ITNG 2008), 7-8 April 2008, Las Vegas, Nevada, USA* (pp. 1253–1254). (Cité en page 8.)

[13] Baniassad, E. L. A. & Clarke, S. (2004). Theme : An approach for aspect-oriented analysis and design. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom* (pp. 158–167). (Cité en pages 31, 32, 33 et 176.)

[14] Bernstein, P. A. (2003). Applying model management to classical meta data problems. In *CIDR 2003, First Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 5-8, 2003, Online Proceedings*. (Cité en pages 6, 11 et 15.)

[15] Bézivin, J., Jouault, F., & Valduriez, P. (2004). On the need for megamodels. In *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. (Cité en page 153.)

[16] Bhalotia, S. (2014). *Aspect-Oriented Modelling with Instantiation Cardinalities illustrated using Software Design Patterns*. PhD thesis, McGill University. (Cité en pages 37, 39, 40 et 176.)

[17] Blouin, A., Combemale, B., Baudry, B., & Beaudoux, O. (2011). Modeling model slicers. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings* (pp. 62–76). (Cité en pages 8 et 175.)

[18] Boehm, B. W. (1987). Improving software productivity. *IEEE Computer*, 20(9), 43–57. (Cité en page 1.)

[19] Bottoni, P., Guerra, E., & de Lara, J. (2010). A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information & Software Technology*, 52(8), 821–844. (Cité en page 20.)

[20] Brooks, F. P. (1986). No silver bullet - essence and accidents of software engineering (invited paper). In *IFIP Congress* (pp. 1069–1076). (Cité en page 1.)

[21] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., & Sabetzadeh, M. (2006). A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management* (pp. 5–12). : ACM. (Cité en pages 6, 11, 15 et 16.)

[22] Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-oriented software architecture, 4th Edition*. Wiley series in software design patterns. Wiley. (Cité en page 61.)

[23] Caron, O., Carré, B., Muller, A., & Vanwormhoudt, G. (2004). An OCL formulation of UML2 template binding. In *International Conference on the Unified Modeling Language* (pp. 27–40). Lisbon, Portugal : Springer. (Cité en pages 27, 28, 29, 47, 175 et 176.)

[24] Carré, B., Vanwormhoudt, G., & Caron, O. (2013). From subsets of model elements to sub-models : A characterization of submodels and their properties. *Software & Systems Modeling (Ed. Springer)*, 14(2), 861–887. (Cité en pages 2, 6, 49, 50, 98, 99 et 152.)

[25] Carré, B., Vanwormhoudt, G., & Caron, O. (2016). On submodels and submetamodels with their relation : A uniform formalization through inclusion properties. *Software & Systems Modeling (Ed. Springer)*, (pp. 1–33). (Cité en pages 63, 105, 106 et 153.)

[26] Carton, A., Driver, C., Jackson, A., & Clarke, S. (2009). Model-Driven Theme/UML. *Trans. Aspect-Oriented Software Development (Ed. Springer)*, 6, 238–266. (Cité en pages 31 et 47.)

[27] Chacon, S. & Straub, B. (2014). *Pro git*. Apress, 2nd edition. (Cité en page 80.)

[28] Chauvel, F. & Fleurey, F. (2007). *Kermeta Language Overview*. `http://diverse-project.github.io/k3/`. (Cité en page 8.)

[29] Chechik, M., Famelis, M., Salay, R., & Strüber, D. (2016). Perspectives of model transformation reuse. In *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings* (pp. 28–44). (Cité en page 1.)

[30] Clarke, S. (2002). Extending standard UML with model composition semantics. *Sci. Comput. Program.*, 44(1), 71–100. (Cité en pages 2, 15, 16 et 175.)

[31] Clarke, S. & Walker, R. J. (2001). Composition patterns : An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada* (pp. 5–14). (Cité en page 31.)

[32] Clarke, S. & Walker, R. J. (2005). *Aspect-oriented software development*, chapter Generic aspect-oriented design with Theme/UML, (pp. 425–458). Addison-Wesley. (Cité en pages 2, 31 et 46.)

[33] Clauss, M. (2001). Generic modeling using UML extensions for variability. In *Workshop on Domain Specific Visual Languages at OOPSLA*. (Cité en pages 19, 20 et 175.)

[34] Cuadrado, J. S., Guerra, E., & de Lara, J. (2012). Flexible model-to-model transformation templates : An application to ATL. *Journal of Object Technology*, 11(2), 4 : 1–28. (Cité en page 44.)

[35] Cuccuru, A., Radermacher, A., Gérard, S., & Terrier, F. (2009). Constraining type parameters of UML 2 templates with substitutable classifiers. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* (pp. 644–649). (Cité en pages 30, 31, 32, 46, 47 et 176.)

[36] de Lara, J. & Guerra, E. (2010). Generic meta-modelling with concepts, templates and mixin layers. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I* (pp. 16–30). (Cité en pages 44, 45 et 176.)

[37] de Lara, J. & Guerra, E. (2013). From types to type requirements : genericity for model-driven engineering. *Software and System Modeling*, 12(3), 453–474. (Cité en pages 2, 44, 46 et 47.)

[38] Del Fabro, D. & Bézivin, J. (2007). Generic model management : from theory to practice. In *First International Workshop on Towers of Models - TOWERS 2007 (co-located with TOOLS EUROPE 2007)* (pp. 1–9). (Cité en pages 6, 11 et 16.)

[39] Dingel, J., Diskin, Z., & Zito, A. (2008). Understanding and improving UML package merge. *Software and System Modeling*, 7(4), 443–467. (Cité en pages 18, 101 et 175.)

[40] D'souza, D. F. & Wills, A. C. (1998). *Objects, components, and frameworks with UML : the catalysis approach*, volume 1. addison-Wesley Reading. (Cité en pages 2, 15, 35, 36, 47 et 176.)

[41] Ellis, M. A. & Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley. (Cité en pages 2 et 5.)

[42] Etien, A., Muller, A., Legrand, T., & Blanc, X. (2010). Combining independent model transformations. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010* (pp. 2237–2243). (Cité en page 1.)

[43] Farinha, J. & Ramos, P. (2015). Extending UML templates towards computability. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.* (pp. 122–133). (Cité en pages 29, 30, 31, 47 et 176.)

[44] Favre, J. (2004). Foundations of model (driven) (reverse) engineering : Models - episode I : stories of the fidus papyrus and of the solarus. In *Language Engineering for Model-Driven Software Development, 29. February - 5. March 2004*. (Cité en page 153.)

[45] Fogel, K. F. & Bar, M. (2001). *Open source development with CVS*. Coriolis Group Books. (Cité en page 80.)

[46] Fondement, F., Muller, P., Thiry, L., Wittmann, B., & Forestier, G. (2013). Big metamodels are evil - package unmerge - A technique for downsizing metamodels. In *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings* (pp. 138–153). (Cité en pages 2, 8, 10 et 175.)

[47] Fowler, M. (1997). *Analysis patterns - reusable object models*. Addison-Wesley series in object-oriented software engineering. Addison-Wesley-Longman. (Cité en pages 2 et 5.)

[48] France , R. B., Ghosh, S., & Turk, D. E. (2003). Supporting effective software modeling. *L'OBJET*, 9(4), 11–29. (Cité en page 1.)

[49] France, R., Georg, G., & Ray, I. (2003). Supporting multi-dimensional separation of design concerns. In *Proceedings of the Third International Workshop on Aspect-Oriented Modeling* : Citeseer. (Cité en pages 5, 41, 42 et 176.)

[50] Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. M. (1993). Design patterns : Abstraction and reuse of object-oriented design. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings* (pp. 406–431). (Cité en pages 2, 5, 23, 139 et 152.)

[51] Hammersley, B. (2005). *Developing feeds with RSS and Atom - developer's guide to syndicating news and blogs*. O'Reilly. (Cité en page 138.)

[52] Hancock, B. (2000). Mass network flooding attacks (distributed denial of service - ddos) surface in the wild. *Computers & Security*, 19(1), 6–7. (Cité en page 128.)

[53] Hebig, R., Seibel, A., & Giese, H. (2012). On the unification of megamodels. *Electronic Communications of the EASST*, 42. (Cité en page 153.)

[54] Herrmannsdörfer, M. & Hummel, B. (2010). Library concepts for model reuse. *Electr. Notes Theor. Comput. Sci. (Ed. Elsevier)*, 253(7), 121–134. (Cité en pages 1 et 5.)

[55] Kagdi, H. H., Maletic, J. I., & Sutton, A. M. (2005). Context-free slicing of UML class models. In *21st IEEE International Conference on Software Maintenance (ICSM2005), 25-30 September 2005, Budapest, Hungary* (pp. 635–638). (Cité en pages 7 et 175.)

[56] Kelsen, P., Ma, Q., & Glodt, C. (2011). Models within models : Taming model complexity using the sub-model lattice. In *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings* (pp. 171–185). (Cité en pages 2, 8, 9 et 175.)

[57] Kent, S. (2002). Model driven engineering. In *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings* (pp. 286–298). (Cité en page 1.)

[58] Kienzle, J., Al Abed, W., Fleurey, F., Jézéquel, J., & Klein, J. (2010). Aspect-oriented design with reusable aspect models. *Trans. Aspect-Oriented Software Development (Ed. Springer)*, 7, 272–320. (Cité en page 37.)

[59] Klein, J. & Kienzle, J. (2007). Reusable aspect models. In *11th Aspect-Oriented Modeling Workshop, Nashville, TN, USA*. (Cité en pages 37, 38, 39, 46, 47 et 176.)

[60] Klein, J., Kienzle, J., Morin, B., & Jézéquel, J. (2009). Aspect model unweaving. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* (pp. 514–530). (Cité en page 66.)

[61] Koegel, M. & Helming, J. (2010). Emfstore : a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010* (pp. 307–308). (Cité en pages 1, 5 et 60.)

[62] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2), 131–183. (Cité en page 1.)

[63] Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., & Jézéquel, J. (2007). Introducing variability into aspect-oriented modeling approaches. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings* (pp. 498–513). (Cité en page 42.)

[64] Lallchandani, J. T. & Mall, R. (2008). Slicing UML architectural models. *ACM SIGSOFT Software Engineering Notes*, 33(3). (Cité en page 7.)

[65] Lano, K. & Rahimi, S. K. (2010). Slicing of UML models using model transformations. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II* (pp. 228–242). (Cité en page 7.)

[66] Lano, K. & Rahimi, S. K. (2011). Slicing techniques for UML models. *Journal of Object Technology*, 10, 1–49. (Cité en page 7.)

[67] Lucrédio, D., Pontin de Mattos Fortes, R., & Whittle, J. (2012). MOOGLE : a metamodel-based model search engine. *Software and System Modeling*, 11(2), 183–208. (Cité en page 153.)

[68] Melnik, S., Bernstein, P. A., Halevy, A., & Rahm, E. (2004). A semantics for model management operators. *Microsoft Technical Report*, (pp. 1–12). (Cité en pages 2, 6, 11, 13, 15, 152 et 175.)

[69] Melnik, S., Rahm, E., & Bernstein, P. A. (2003). Rondo : A programming platform for generic model management. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (pp. 193–204). (Cité en pages 6, 11 et 175.)

[70] Mili, H., Mili, F., & Mili, A. (1995). Reusing software : Issues and research directions. *IEEE Trans. Software Eng.*, 21(6), 528–562. (Cité en page 1.)

[71] Morin, B., Vanwormhoudt, G., Lahire, P., Gaignard, A., Barais, O., & Jézéquel, J. (2008). Managing variability complexity in aspect-oriented modeling. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings* (pp. 797–812). (Cité en pages 42, 43, 44, 47 et 176.)

[72] Muller, A. (2004). Reusing functional aspects : from composition to parameterization. In *Aspect-Oriented Modeling Workshop, AOM*. (Cité en page 33.)

[73] Muller, A. (2006). *Construction de systèmes par application de modèles paramétrés*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I. (Cité en pages 2, 27, 33, 34, 45, 47, 52, 56, 85, 86, 102, 176 et 177.)

[74] Muller, A., Caron, O., Carré, B., & Vanwormhoudt, G. (2003). Réutilisation d'aspects fonctionnels : des vues aux composants. *L'OBJET*, 9(1-2), 241–255. (Cité en page 152.)

[75] Muller, A., Caron, O., Carré, B., & Vanwormhoudt, G. (2005). On some properties of parameterized model application. In *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings* (pp. 130–144). (Cité en page 33.)

[76] Naftalin, M. & Wadler, P. (2006). *Java generics and collections*. O'Reilly. (Cité en page 5.)

[77] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S. M., & Zave, P. (2007). Matching and merging of statecharts specifications. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007* (pp. 54–64). (Cité en pages 2 et 15.)

[78] OMG (2011a). Auxiliary Constructs Templates, chapter 17. UML 2.4.1 Superstructure Specification. (Cité en pages 2, 23, 24, 47 et 120.)

[79] OMG (2011b). UML 2.4.1 - Infrastructure Specification. `http://www.omg.org/spec/UML/2.4.1/`. (Cité en pages 2, 13, 14, 17, 24 et 175.)

[80] OMG (2011c). UML 2.4.1 - Superstructure Specification. http://www.omg.org/spec/UML/2.4.1/. (Cité en page 101.)

[81] OMG (2012). CORBA 3.3 - Common Object Request Broker Architecture. https://www.omg.org/spec/CORBA/3.3/. (Cité en page 1.)

[82] OMG (2014a). MDA 2.0 - Model Driven Architecture. https://www.omg.org/mda/specs. (Cité en page 1.)

[83] OMG (2014b). OCL 2.4 - Object Constraint Language. http://www.omg.org/spec/OCL/2.4/. (Cité en page 27.)

[84] OMG (2016). MOF 2.5.1 - Meta Object Facility. https://www.omg.org/spec/MOF/2.5.1/. (Cité en pages 8, 11 et 37.)

[85] Overstreet, C. M., Nance, R. E., & Balci, O. (2002). Issues in enhancing model reuse. In *International Conference on Grand Challenges for Modeling and Simulation, Jan* (pp. 27–31). (Cité en pages 1 et 5.)

[86] Parnas, D., Clements, P., & Weiss, D. (1989). Enhancing reusability with information hiding. In *Software reusability : vol. 1, concepts and models* (pp. 141–157). : ACM. (Cité en page 1.)

[87] Pilato, C. M., Collins-Sussman, B., & Fitzpatrick, B. W. (2008). *Version control with subversion - the standard in open source version control*. O'Reilly. (Cité en page 80.)

[88] Ramos, R., Barais, O., & Jézéquel, J. (2007). Matching model-snippets. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings* (pp. 121–135). (Cité en pages 40, 41, 46, 47 et 176.)

[89] Reddy, Y. R., Ghosh, S., France, R. B., Straw, G., Bieman, J. M., McEachen, N., Song, E., & Georg, G. (2006). Directives for composing aspect-oriented design class models. In *Transactions on Aspect-Oriented Software Development I* (pp. 75–105). (Cité en pages 41, 42, 46, 47 et 176.)

[90] Reiter, T., Kapsammer, E., Retschitzegger, W., & Schwinger, W. (2005). Model integration through mega operations. In *Workshop on Model-driven Web Engineering, 15Th International Conference on Web Engineering, Sydney, Australia* (pp. 20–29). (Cité en pages 15, 16, 17 et 175.)

[91] Ruhroth, T., Gärtner, S., Bürger, J., Jürjens, J., & Schneider, K. (2014). Versioning and evolution requirements for model-based system development. *Softwaretechnik-Trends*, 34(2). (Cité en pages 1 et 5.)

[92] Sen, S., Moha, N., Baudry, B., & Jézéquel, J. (2009). Meta-model pruning. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings* (pp. 32–46). (Cité en page 12.)

[93] Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., & Jézéquel, J. (2012). Reusable model transformations. *Software and System Modeling*, 11(1), 111–125. (Cité en pages 2, 12, 13 et 175.)

[94] Sendall, S. & Kozaczynski, W. (2003). Model transformation : The heart and soul of model-driven software development. *IEEE Software*, 20(5), 42–45. (Cité en page 1.)

[95] Shvets, A., Pavlova, M., & Frey, G. (2015). Design patterns explained simply. *URL : https ://sourcemaking. com [Online*. (Cité en page 63.)

[96] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF : eclipse modeling framework*. Pearson Education. (Cité en pages 8, 37, 46, 47 et 60.)

[97] Stepper, E. (2014). Cdo model repository. Online. `https://help.eclipse.org/mars/topic/org.eclipse.emf.cdo.doc/html/Overview.html`. (Cité en page 60.)

[98] Sutton, A. & Stroustrup, B. (2011). Design of concept libraries for C++. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers* (pp. 97–118). (Cité en page 44.)

[99] Tessier, P., Gérard, S., Terrier, F., & Geib, J. (2005). Using variation propagation for model-driven management of a system family. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings* (pp. 222–233). (Cité en page 19.)

[100] Tombelle, C., Vanwormhoudt, G., & Renaux, E. (2011). Reusing pattern solutions in modeling : A generic approach based on a role language. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers* (pp. 139–159). (Cité en pages 20, 21, 22 et 175.)

[101] Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 31–42. (Cité en page 110.)

[102] Vanwormhoudt, G., , Carré, B., Caron, O., & Tombelle, C. (2014). Recherche de sous-modèles. In *CIEL 2014, Troisième Conférence en IngénieriE du Logiciel* (pp. 126). : HAL. (Cité en page 63.)

[103] Vanwormhoudt, G., Caron, O., & Carré, B. (2017). Aspectual templates in UML - enhancing the semantics of UML templates in OCL. *Software and System Modeling (Ed. Springer)*, 16(2), 469–497. (Cité en pages 2, 26, 27, 28, 47, 49, 52, 53, 57, 75, 102, 110, 117, 118, 121, 151 et 175.)

[104] Wilson, J. M. (2003). Gantt charts : A centenary appreciation. *European Journal of Operational Research (Ed. Elsevier)*, 149(2), 430–437. (Cité en page 126.)

[105] Ziadi, T., Hélouët, L., & Jézéquel, J. (2003). Towards a UML profile for software product lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers* (pp. 129–139). (Cité en page 19.)

Table des figures

2.1 Extractions à partir de m (Source : Melnik et al. [69]) . . . 6
2.2 Partie d'un modèle de classes UML selon les prédicats définis (Source : Kagdi et al. [55]) . . . 7
2.3 Définition d'extracteurs de modèles avec *Kompren* (Source : Blouin et al. [17]) . . . 8
2.4 Modèle sous forme de graphe et découpage selon les *SCC* (Source : Kelsen et al. [56]) . . . 9
2.5 Treillis de sous-modèles du modèle 2.4(a) selon le graphe acyclique orienté de 2.4(b) (Source : Kelsen et al. [56]) . . . 9
2.6 Utilisation de la relation *unmerge* (Source : Fondement et al. [46]) . . . 10
2.7 Utilisation de la relation *unmerge* (Source : Fondement et al. [46]) . . . 10
2.8 Différence entre m et m' selon *map* (Source : Melnik et al. [68]) . . . 11
2.9 Exemple d'union de deux versions de modèles (Source : Alanen & Porres [4]) . . . 12
2.10 Extrait de l'ensemble des opérations et opérations duales (Source : Alanen & Porres [4]) . . . 12
2.11 Suppression d'éléments dans un schéma relationnel m (Source : Melnik et al. [68]) 13
2.12 Vue d'ensemble de l'approche avec les métamodèles *UML et Java* (Source : Sen et al. [93]) . . . 13
2.13 Référence des éléments *Time* et *String* de *Types* par *Program* via les relations *acces* et *import* (Source : OMG [79]) . . . 14
2.14 Référence des éléments de *Types* par *WebShop* et *ShoppingCart* (*import*) et référence des éléments de *Auxiliary* par *ShoppingCart* (*access*)(Source : OMG [79]) . . . 14
2.15 Exemple d'utilisation de la relation *override* (Source : Clarke [30]) . . . 16
2.16 Exemple d'utilisation de la relation *override* et de *MIC* sur l'attribut *id* (Source : Reiter et al. [90]) . . . 17
2.17 Sémantique de la relation *merge* sur des paquetages (Source : OMG [79]) . . . 17
2.18 Fusion des paquetages *CarAgency* et *AgencyClient* . . . 18
2.19 Exemple de fusion de types (Source : Dingel et al. [39]) . . . 18
2.20 Modèle variable - *Generic UML models* (Source : Clauss [33]) . . . 20
2.21 Modèle (variation) après sélection de *CreditCard_Data* . . . 20
2.22 Modèle de rôles, représentant le pattern *Proxy* (Source : Tombelle et al. [100]) . . . 21
2.23 Modèle de rôles appliqué à un modèle (Source : Tombelle et al. [100]) . . . 21
2.24 Résultat de l'attribution des rôles (Source : Tombelle et al. [100]) . . . 22
2.25 Un template UML représentant le pattern Observer . . . 22
2.26 Instanciation du template de classe *Stack* . . . 23
2.27 Application d'un template de package UML . . . 24
2.28 Métamodèle des *templates UML* . . . 24
2.29 Métamodèle du *binding de templates UML* . . . 25
2.30 Extrait du diagramme d'objets pour le paquetage *CarHiringSystem* . . . 25
2.31 Éléments paramétrables et templatables (Source :Vanwormhoudt et al. [103]) . . . 26
2.32 Aspectual templates de composants (Source :Vanwormhoudt et al. [103]) . . . 27
2.33 Classification des types de modèles UML templatables (Source :Vanwormhoudt et al. [103]) . . . 28
2.34 Erreur concernant la signature d'une opération (Source :Caron et al. [23]) . . . 28

2.35 Erreur concernant une des propriétés d'une association (Source :Caron et al. [23]) 29
2.36 Substitution du type d'un paramètre (Source :Farinha & Ramos [43]) 30
2.37 Substitution du type d'un paramètre - (Contre-exemple) 30
2.38 Exemples de non-conformité à une relation de contenance (Source : Farinha & Ramos [43]) 31
2.39 Spécification de *Iterator* et de la classe définissant le contrat *IterableForward* (Source : Cuccuru et al. [35]) 31
2.40 Réalisation du contrat de *IterableForward* et application de *Iterator* (Source : Cuccuru et al. [35]) 32
2.41 theme d'une fonctionnalité de *log* (Source : Baniassad & Clarke [13]) 32
2.42 Utilisation du Theme de *log* (Source : Baniassad & Clarke [13]) 33
2.43 Application d'un composant de modèle à un modèle de base (Source : Muller [73]) 33
2.44 Application d'un composant de modèle à un autre (Source : Muller [73]) 34
2.45 Séquences d'applications de *Comptage* à *Allocation* à *Base* (Source : Muller [73]) 34
2.46 Framework d'allocation de ressources (Source : D'souza & Wills [40]) 35
2.47 Utilisation du framework d'allocation de ressources (Source : D'souza & Wills [40]) 35
2.48 Défintion du template Sorted_List_Template (Source : D'souza & Wills [40]) . . 36
2.49 Application du template Sorted_List_Template (Source : D'souza & Wills [40]) . 36
2.50 Extrait du métamodèle Ecore 37
2.51 Exemple de définition et de binding de template UML vers Java 38
2.52 Aspect sous forme de templates (Source : Klein & Kienzle [59]) 38
2.53 Instanciation de templates (Source : Klein & Kienzle [59]) 39
2.54 Aspect réutilisable avec des cardinalités d'instanciations (Source : Bhalotia [16]) . 40
2.55 *Template UML* d'une classe (Source : Ramos et al. [88]) 41
2.56 Processus pour dériver un métamodèle de *snippets* de patterns (Source : Ramos et al. [88]) 41
2.57 Source : France et al. [49] 42
2.58 *Primary model* et *Composed model* après utilisation de la directive *"remove"* (Source : Reddy et al. [89]) 42
2.59 Le template aspectuel *TaskScheduling* (Source : Morin et al. [71]) 43
2.60 Composition par héritage (Source : Morin et al. [71]) 44
2.61 *Semantic mixin layers* (template) et *concept* (Source : de Lara & Guerra [36]) . . 44
2.62 Définition et utilisation d'un template de modèle (Source : de Lara & Guerra [36]) 45

3.1 Modèle *Heating System* et son graphe de dépendances 50
3.2 Sous-modèles 50
3.3 Sous-modèle *RegulationSubCircuit* extrait de *HeatingSystem* 51
3.4 Template UML et Aspectual Template 52
3.5 Constituants d'un aspectual template : Sous-modèle paramètre et sous-modèle spécifique 52
3.6 Application du pattern Observer Composition du contexte et du modèle résultat 53
3.7 Composition d'aspectual templates 54
3.8 Exemple d'assemblages de templates 55
3.9 Applications de templates sur un contexte 56
3.10 Composition de templates et application à un contexte 56
3.11 Exemple de paramètres (classes et associations) non-substitués 57
3.12 Exemple de paramètres (attributs) non-substitués 58

3.13 Ingénierie Dirigée par les Modèles Basée sur les templates 59
3.14 Adaptation des paramètres du template 62
3.15 Composition par *merge* 63
3.16 Construction de modèles par applications *(a)* aspectuelle et *(b)* générative de templates 65
3.17 Identification et extraction d'un template 67
3.18 Recherche guidée par les templates 68
3.19 Bound model = merge(instance, contexte) 69
3.20 Instanciation de template 70
3.21 Modèle paramètre bien formé et modèle spécifique mal formé 71
3.22 Instanciation de template avec un modèle paramètre mal formé 71
3.23 Instanciation partielle 72
3.24 Séquences d'instanciations alternatives - Instanciations partielles 73
3.25 Séquences d'instanciations alternatives - Instanciation totale 73
3.26 Application partielle et instanciation partielle 74
3.27 Fonctionnalités (*modèle spécifique*) de *ObserverPatternPush* présentes dans *CarHiringSystem* 75
3.28 Calcul des substitutions entre *ObserverPatternPush* et *CarHiringSystem* 76
3.29 Instances de *ObserverPatternPush* 76
3.30 *ObservableAgencyClient* inclus dans *CarHiringSystem* 76
3.31 Modèle spécifique de *ObserverPatternPush* lié aux éléments actuels de *CarHiringSystem* dans *ObservableAgencyClient* 77
3.32 Obtention de l'ensemble clos des éléments du modèle spécifique de *ObserverPatternPush* dans *ObservableAgencyClient* 78
3.33 Obtention du modèle spécifique clos (MSC) dans *ObservableAgencyClient* 78
3.34 Retrait des éléments du modèle spécifique de *ObserverPatternPush* dans le modèle *CarHiringSystem* 79
3.35 Conception incrémentale 79
3.36 Modélisation en équipe 79
3.37 Hiérarchie de variantes 80
3.38 Validité de l'application d'un *aspectual template* sur un modèle 81
3.39 Validité de l'application d'*ObserverPattern* sur *AgencyClient* 81
3.40 M inclus dans M' 81
3.41 AT applicable au surmodèle M' 82
3.42 *ObserverPattern* applicable sur le surmodèle *CarAgencyClient* 82
3.43 M' sous-modèle de M 82
3.44 AT applicable sur le sous-modèle M' 83
3.45 *ObserverPattern* applicable selon S sur le sous-modèle *AgencyClient* 83
3.46 AT applicable selon S sur toute hiérarchie de surmodèles du modèle actuel 84
3.47 *ObserverPattern* applicable selon S sur *AgencyClient*, *CarAgencyClient* et *AgencyReceipt* 85
3.48 Applicabilité et enchaînement d'applications 85
3.49 Même modèle résultat R quel que soit l'ordre d'applications (Muller [73]) 86
3.50 Première séquence d'applications possible de *ObserverPattern* et *QueryingPattern* 86
3.51 Seconde séquence d'applications possible de *ObserverPattern* et *QueryingPattern* 87
3.52 Étude de la relation entre R et R' via des applications *(a)* identiques ($S = S'$) et *(b)* différentes ($S \neq S'$) 88

3.53 Le contexte M et l'instance I sont inclus dans le résultat R . . . 88
3.54 R inclus dans R' pour $S = S'$ . . . 88
3.55 *QueryableCarTransfer* inclus dans *QueryableCarEquipmentTransfer* . . . 89
3.56 $S' \cap S \neq \emptyset$ . . . 90
3.57 Instanciations de *Allocation* et fermeture des éléments du modèle spécifique . . . 91
3.58 Simplification de conception incrémentale . . . 92
3.59 Modèles spécifiques clos relatifs aux deux instances - Aucune relation entre les modèles résultats . . . 93
3.60 Conception incrémentale - Exemple . . . 94

4.1 *AgencyClient* et *CarHiringSystem* . . . 97
4.2 Extraction de *AgencyClient* à partir de *CarHiringSystem* . . . 99
4.3 Relation d'inclusion entre *CarHiringSystem* et *AgencyClient* . . . 101
4.4 Fusion de *CarAgency* et *AgencyClient* . . . 103
4.5 Relation d'application aspectuelle entre *ObserverPattern* et *AgencyClient* . . . 103
4.6 Relation d'instanciation entre *ObserverPattern* et *AgencyClient* . . . 105
4.7 Relation de promotion entre *GenericObserver* et *ObserverPattern* . . . 106
4.8 Relation de restriction entre *GenericObserver* et *ObserverPattern* . . . 108
4.9 Modèles compatibles avec l'application de *ObserverPattern* sur *AgencyClient* . 109
4.10 *CarEquipManagement* inclut des instances de *ObserverPattern* . . . 111
4.11 Relation *unbind* entre *ObserverPattern* et *ClientManagement* . . . 114
4.12 Atelier de modélisation à base d'*aspectual templates* . . . 116
4.13 Fonctionnalités exposées par l'éditeur d'*aspectual templates* . . . 117
4.14 Inférence de substitutions . . . 118
4.15 *CarHiringSystem* respecte la structure de *ObserverPattern* . . . 119
4.16 Sous et surmodèles compatibles avec l'application de *ObserverPattern* sur *CarAgencyClient* . . . 119
4.17 Profil *UML* utilisé par l'atelier . . . 120
4.18 Exemples d'*aspectual templates* mal-formés . . . 120
4.19 Architecture de l'atelier de modélisation . . . 121
4.20 Architecture du moteur d'*aspectual templates* . . . 122
4.21 Conversions de modèles . . . 124

5.1 Modélisation du serveur *REST* : *patrons de conception* (*templates*) utilisés . . . 127
5.2 Limitation du nombre de requêtes sur une ressource . . . 128
5.3 Résultat de l'extraction de *ResourceRequestControl* à partir de *ProtectedResourceServer* . . . 129
5.4 Résultat de l'application de *ExtendableProxyfiedResourceAccess* sur *ResourceRequestControl* . . . 129
5.5 Accès par mot de passe . . . 130
5.6 Résultat de l'extraction de *UserPasswordManagement* à partir de *PowerControlledServer* . . . 130
5.7 Résultat de l'application de *ProxyfiedResourceAccess* sur *UserPasswordManagement* 131
5.8 Fusion des fonctionnalités . . . 131
5.9 Résultat de la fusion de *UserPasswordProxy* avec *ControlledRequestProxy* . . . 132
5.10 Serveur de ressources : Hiérarchie de modèles après modélisation de l'accès sécurisé au serveur . . . 132

5.11 Limitation du temps d'existence des ressources . . . 133
5.12 Résultat de l'extraction de "ResourceDecay" à partir de "WatchedPowerControlledServer" . . . 133
5.13 Résultat de l'application de "ResourceExtendedProxy" sur "ResourceDecay" . . . 134
5.14 Cache et limitation du nombre des ressources . . . 135
5.15 Résultat de l'extraction de *ResourceServerPool* à partir de *ResourceDistribution* . 135
5.16 Résultat de l'application de *ReadAccesProxyfiedResource* sur *ResourceServerPool* 136
5.17 Fusion des fonctionnalités . . . 136
5.18 Résultat de la fusion de "ResourceDecayProxy" avec "ProxyfiedAccessServer" . . . 137
5.19 Serveur de ressources : Hiérarchie de modèles après la modélisation du critère d'accès performant au serveur . . . 138
5.20 Extensibilité du serveur d'information et acquisition des données . . . 139
5.21 Résultat de la recherche des sous-modèles compatibles avec l'applicabilité de *EventChannel* sur *ParallelDataServerObtaining* . . . 140
5.22 Résultat de la recherche des sous-modèles compatibles avec l'applicabilité de "EventChannel" sur "DBServer" . . . 140
5.23 Résultat de l'application de *EventChannel* sur *LocalisationServer* . . . 141
5.24 Adaptation en serveur de ressources . . . 141
5.25 Résultat de l'application de "ToResource" sur "LocalisationServerDataObtaining" 142
5.26 Résultat de l'application de "ToResourceServer" sur "DataToResource" . . . 142
5.27 Serveur de ressources : Hiérarchie de modèles après la modélisation du critère d'extensibilité du serveur . . . 143
5.28 Serveur REST : Fusion de l'ensemble des fonctionnalités modélisées . . . 143
5.29 Version finale du serveur REST : Fusion de trois branches . . . 144
5.30 Résultat de la fusion de l'ensemble des fonctionnalités . . . 145
5.31 Adaptation du pattern Proxy au contexte *Serveur de ressources* . . . 146
5.32 Résultat des instanciations de "ProxyPattern" avec "ResourceServerGetDelete" . . . 146
5.33 Résultat de la fusion des *templates* partiellement contextualisés . . . 147
5.34 Résultat de la promotion de paramètres dans *ReadAccessProxyfiedResource* . . . 147
5.35 Adaptation du pattern Decorator au contexte *Accès restreint à une ressource* . . . 147
5.36 Résultat des instanciations de *DecoratorPattern* avec *ProtectedResourceServer* . . . 148
5.37 Résultat de la fusion des *templates* partiellement contextualisés . . . 148
5.38 Adaptation du pattern Adapter au contexte *Ressources* . . . 149
5.39 Résultat de l'instanciation partielle d'*AdapterPattern* avec *ResourceServerGetDelete* 149
5.40 Adaptation du pattern Adapter au contexte *Serveur de ressource* . . . 150
5.41 Résultat de l'instanciation partielle d' *AdapterPattern* avec *ResourceServerGetDelete* 150

A.1 Exemples d'*aspectual templates* mal-formés . . . 156
A.2 Exemples de vérification de conformité structurelle des éléments actuels . . . 157

B.1 Première séquence d'applications possible de *QueryingPattern* . . . 160
B.2 Seconde séquence d'applications possible de *QueryingPattern* . . . 160

C.1 Serveurs de ressources - 1\2 . . . 161
C.2 Serveurs de ressources - 2\2 . . . 162
C.3 Serveurs de BDD . . . 163

D.1 Modélisation de l'accès sécurisé au serveur . . . 164

D.2 Modélisation de l'accès performant au serveur 165
D.3 Modélisation de l'extensibilité du serveur et de l'accès aux informations 166

Liste des tableaux

2.1 Caractéristiques des applications de templates par approches 47

3.1 Forme possible des constituants de template 70
3.2 Applicabilité de templates et formes des modèles résultats lors de substitutions totales et partielles 74

4.5 Opérateurs de modèles 101
4.12 Opérateurs de *templates* 115

Ingénierie Dirigée par les Modèles Basées sur les Templates

Résumé :
Pour répondre à la complexité grandissante des systèmes, la réutilisation de modèles est employée dans les phases amont d'analyse et de conception. Dans cette thèse, nous nous intéressons à cette réutilisation en privilégiant les modèles paramétrés que sont les templates. Ceux-ci expriment des connaissances générales applicables à différents contextes. Sur la base des aspectual templates possédant un modèle en paramètre, nous proposons une ingénierie dédiée. Celle-ci est structurée autour d'un dépôt de modèles et de deux espaces de conception : celui des templates et celui des modèles applicatifs, chaque espace supportant des activités de modélisation spécifiques. Nous contribuons à cette ingénierie en approfondissant trois axes. Tout d'abord, en examinant la relation "bind" des templates UML, nous isolons l'instanciation comme opération de plein droit pour construire un modèle basé sur la structure du template. Les questions d'instanciation partielle et de séquences d'instanciation sont aussi examinées. Ensuite, pour répondre à des besoins d'évolution de modèles, nous proposons des opérateurs pour détecter et supprimer des templates dans un modèle. Enfin, nous étudions l'application de templates sur une hiérarchie de modèles dont les usages se rencontrent dans le versionnement et la modélisation en équipe. Pour faciliter ces usages, des règles définissant la validité de telles applications et leurs effets sur les relations entre leurs modèles résultats sont données. Nous appliquons cette ingénierie en proposant des opérateurs, leur mise en œuvre dans une technologie réutilisable et leur expérimentation en modélisant un serveur REST d'agrégation d'informations.

Mots clefs : Ingénierie Dirigée par les Modèles, Templates, Réutilisation, Tissage, Instanciation, Hiérarchies de modèles, UML, Patrons de conception

---

Template Based Model Driven Engineering

Abstract :
Against the growing complexity of systems, model reuse is often used in the analysis and design steps of software development. In this thesis, we explore this kind of reuse by focusing on templates which are parameterized models. Templates capture general knowledge that can be adapted to various application contexts. On the basis of aspectual templates which have a model as parameter, we propose a dedicated engineering. This engineering is structured around a model repository and two design spaces : one for templates and one for application models, each space supporting its specific modeling activities. We contribute to this engineering by studying three topics. First, through the analysis of the UML "bind" relationship, we isolate instantiation as a first-class operation to build a model based on the template structure. Questions about partial instantiation and instantiation sequences are further examined. Then, in order to ease model comprehension and evolution, we provide operators for detecting and deleting templates inside models. Finally, we study the application of templates on model hierarchies which occurs in model versioning and team modeling. To facilitate these uses, rules defining the validity of such applications and their effects on the relations between resulting models are given. We apply this engineering by presenting corresponding operators, their implementation in a reusable technology and their experimentation to model a REST server of data curation.

Keywords : Model Driven Engineering, Templates, Reuse, Weaving, Instantiation, Model Hierarchies, UML, Design Patterns