Evaluation de l'apport des aspects, des sujets et des vues pour la composition et la réutilisation des modèles

Olivier Barais* — Philippe Lahire — Alexis Muller*****
Noël Plouzeau* — Gilles Vanwormhoudt***

** IRISA Rennes, Projet Triskell, IRISA - campus de Beaulieu F-35042 Rennes cedex*
*** I3S Nice-Sophia-Antipolis, Equipe OCL, I3S-UNSA - les algorithmes*
2000 route des lucioles BP 121 F-06903 Sophia-Antipolis cedex
**** Equipe Goal, LIFL-USTL - Bâtiment M3, F-59655 Villeneuve d'Ascq cedex*

RÉSUMÉ. *La réutilisation, l'évolution ou l'adaptation rapide du code d'une application figurent parmi les préoccupations fortes des entreprises. L'ingénierie des modèles tente d'apporter une solution en plaçant le modèle au centre du développement logiciel et en capturant le savoir-faire métier initialement décrit dans le code de l'application. Cette approche a l'avantage de rendre la description indépendante des plates-formes d'exécution. Notre objectif dans cet article est de présenter une synthèse de trois approches différentes pour la composition de modèles, de les évaluer par rapport aux critères mis en évidence par le réseau d'excellence AOSD-EUROPE pour en retirer des enseignements pertinents et ainsi faire de nouvelles propositions. L'évaluation met en évidence la capacité de chacune des approches à composer à la fois des préoccupations fonctionnelles et extrafonctionnelles.*

ABSTRACT. *The reuse, the evolution or the fast adaptation of the code of an application are among the strongest concerns of companies. Model engineering tries to bring a solution putting the model in the center of the software development and by capturing the business know-how initially described in the code of the application. This approach has the advantage to make the description independant of software platforms. Our objective in this paper is to present three different approaches for the composition of models, to evaluate them using the criteria proposed by the AOSD-EUROPE network of excellence in order to extract relevant information. From this evaluation, this paper provides new proposals. The evaluation aims at showing the capacity of each approach to support the composition of both functional and extra-functional concerns.*

MOTS-CLÉS : *ingénierie des modèles, patron d'architecture, composant de modèle, programmation par sujets, développement par aspects, point de vue, modèle paramétré.*

KEYWORDS: *model driven engineering, software architecture pattern, model component, subject oriented programming, aspect oriented software development, views, parameterized model.*

DOI:10.3166/OBJ.13.2-3.177-211

RSTI - L'objet – 13/2007. Points de vue et paradigmes proches, pages 177 à 212

1. Introduction

La réutilisation, l'évolution ou l'adaptation rapide du code d'une application figurent parmi les préoccupations fortes des entreprises. Cependant l'apparition fréquente de nouvelles plates-formes, leur constante évolution et leur disparition éventuelle rendent souvent caduque l'investissement réalisé. L'ingénierie des modèles tente d'apporter une solution en plaçant le modèle au centre du développement logiciel et en capturant le savoir-faire métier initialement décrit dans le code de l'application, dans des modèles le plus souvent en utilisant des langages de surface (Clark *et al.*, 2004).

Un des apports important de cette approche est de rendre la description du savoir-faire la plus indépendante possible des plates-formes logicielles et de s'appuyer ensuite sur des techniques de transformation comme la génération de code pour produire les programmes correspondants. Cependant cela a pour conséquence de faire remonter au niveau du modèle les problèmes de réutilisation, d'évolution ou d'adaptation déjà mis en évidence pour le code. Entre autre, se posent au niveau des modèles, les problèmes de structuration. En effet pour qu'une application soit aisément compréhensible et maintenable, il est crucial de séparer convenablement les préoccupations (Parnas, 1972). Cependant, les approches de spécification d'une application à l'aide du langage UML ou à l'aide de langage de description d'architecture proposent une seule et unique dimension de décomposition autour des fonctionnalités de l'application. Pourtant, différents travaux ont montré la difficulté de structurer une application à l'aide d'une unique dimension ; ce problème a été identifié comme la *tyrannie de la décomposition dominante* (Ossher *et al.*, 1999). En conséquence, certaines facettes ou préoccupations se retrouvent noyées, disséminées et répétées dans différents éléments du modèle. Il est donc intéressant, pour limiter la complexité des modèles et surtout pour augmenter leur degré de réutilisation, de décrire les différentes facettes fonctionnelles (description d'un photocopieur, d'un site d'enchères...) et extrafonctionnelles (synchronisation, sécurité, traces...) séparément et ensuite de les composer.

Notre objectif dans cet article est de situer par rapport à l'état de l'art trois approches différentes pour la composition de modèles, de les évaluer pour en retirer des enseignements pertinents et ainsi faire de nouvelles propositions. Nous nous intéressons aux approches suivantes : *i)* celle de (Barais, 2005) qui repose sur la notion de patron d'architecture et de masque de points de jonction pour favoriser la construction incrémentale d'une description d'architecture logicielle, *ii)* celle de (Muller *et al.*, 2003; Muller *et al.*, 2005a) basée sur les points de vues et le concept de modèle paramétré et, *iii)* celle de (Lahire *et al.*, 2006; Crescenzo *et al.*, 2005) qui s'inspire des approches par sujets et aspects et propose d'associer à tout modèle dit réutilisable, un protocole de composition sous la forme d'un *adaptateur*. Pour évaluer et comparer ces trois approches nous nous appuyons sur un exemple commun d'un système de location de véhicules.

Le reste du document est structuré de la manière suivante. Nous présentons dans un premier temps l'exemple pivot utilisé comme banc de test pour évaluer les trois approches étudiées. Nous présentons ensuite en détail les trois approches : TranSAT, l'approche composant de modèle, et SmartAdapters. La section 6 compare ensuite ces trois approches en se fondant sur les critères de (Chitchyan *et al.*, 2005). Nous

nous intéressons en particulier (mais ce n'est pas limitatif) à l'intérêt de disposer d'un langage de surface, à la facilité de décrire la composition, à sa capacité à être réutilisée mais aussi à l'expressivité qui est nécessaire à la description des points d'intégration des facettes et à la spécification des modifications à mettre en œuvre pour réaliser cette intégration. Finalement, à partir de l'analyse des trois approches précédemment citées, nous identifions, dans la section 7, les caractéristiques d'un canevas générique de composition de modèles. La section 8 présente différents travaux connexes aux approches présentées. Enfin, la section 9 dresse une courte conclusion et présente les perspectives associées à ce travail.

2. Exemple de référence

Afin d'illustrer l'utilisation des différentes approches, nous considérons la construction du système d'information d'une entreprise de location de véhicules.

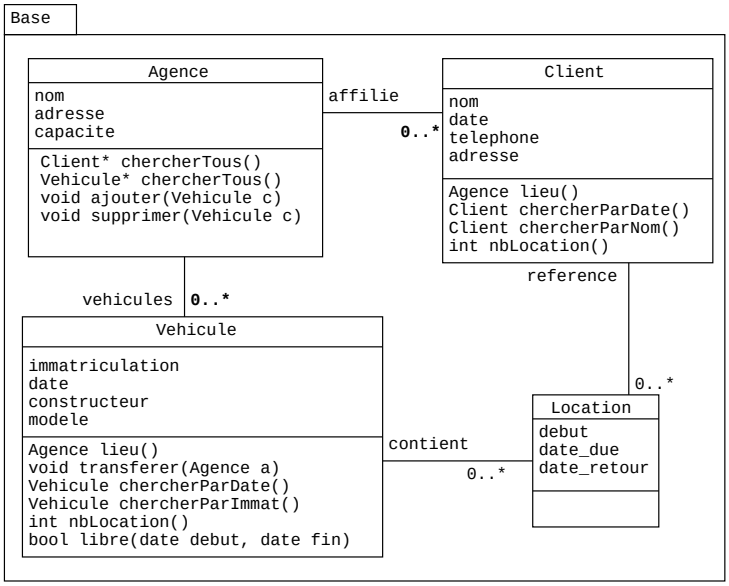

Figure 1. *Location de véhicules*

Un tel système est présenté figure 1. Il est constitué des entités *Agence*, *Client*, *Vehicule* et *Location*. Celui-ci doit répondre à différentes préoccupations métiers : recherche de véhicules et de clients dans les différentes agences, gestion des stocks et des locations de véhicules. Ces différentes préoccupations se retrouvent au travers des différents attributs et opérations répartis et entrelacés dans les entités. On a par exemple, *capacité*, *ajouter*, *supprimer* et *transferer* pour la préoccupation de gestion des stocks, ou encore *nbLocations*, *libre* et l'entité *Location* pour la préoccupation de gestion des locations. Bien que n'étant pas illustrées ici, des préoccupations techniques telles que la gestion d'une interface graphique, par exemple, peuvent également

être considérées. La recherche de ressources, la gestion ou l'allocation de ressources sont des préoccupations qui ne sont pas spécifiques à ce système en particulier. Ces préoccupations peuvent se retrouver dans de nombreux systèmes comme par exemple un système de gestion de bibliothèque (Muller *et al.*, 2003). Les approches présentées par la suite visent à permettre l'expression de solutions génériques à des préoccupations récurrentes et leur intégration pour la construction de différents systèmes. Pour chaque approche, cette possibilité est illustrée par la mise en œuvre de notre exemple de système de locations de véhicules.

3. TranSAT

3.1. *Objectif*

L'objectif de TranSAT (Barais, 2005) est de permettre la construction incrémentale de description d'architecture logicielle à base de composants. TranSAT propose la définition d'un nouveau mécanisme de composition au sein d'une architecture logicielle : le tissage de plans. Il se fonde pour la description d'architecture sur le modèle de composants SafArchie (Barais *et al.*, 2005a) qui est proche du modèle de composants d'UML 2.0 (, *UML 2.0 Infrastructure Specification*, 2005) pour la partie structurelle et permet une description du comportement définie à l'aide d'une algèbre de processus fondée sur le modèle mathématique des automates à entrées-sorties (Lynch *et al.*, 1989). L'approche par tissage de plans vient compléter les mécanismes de modularisation et de composition existants dans SafArchie pour proposer un canevas de conception global de description d'architecture logicielle. Ceci permet de structurer au sein d'un plan les informations relatives à une préoccupation et de définir alors comment ce plan est intégré à une architecture existante.

3.1.1. *Une approche par transformation*

TranSAT propose de décrire les services techniques à l'aide des mêmes entités primaires (composants, connexions et composites) que les services métiers d'une application de manière à pouvoir gérer de manière homogène les applications à tous les niveaux d'abstraction. Le résultat du tissage d'une préoccupation technique au sein d'une architecture logicielle est composé uniquement de ces mêmes entités. Il peut alors être lui-même considéré comme une nouvelle architecture à qui l'on pourra ajouter de nouvelles préoccupations (approche récursive).

Ce choix de l'homogénéité déplace la problématique sur le tissage de deux plans : le plan de base et le plan décrivant la préoccupation à intégrer. TranSAT propose de réaliser ce tissage à l'aide d'une approche par transformation dans laquelle les modifications à apporter au niveau d'une architecture logicielle initiale (aussi appelée plan de base) sont spécifiées de manière explicite afin de permettre l'intégration d'une nouvelle préoccupation. Inspirée par les concepts mis en avant dans la programmation par aspects (Kiczales *et al.*, 1997), l'approche propose une inversion des préoccupations au niveau de la conception des architectures. Ainsi, ce ne sont pas des composants

métiers qui identifient leurs besoins en termes de services techniques, mais ce sont les préoccupations qui viennent transformer l'architecture logicielle. Dans ce sens, le tissage de deux plans est une composition invasive (Abmann, 2003) qui vient transformer la spécification des composants métiers et leur assemblage.

3.1.2. *Architecture de TranSAT*

Le canevas de conception TranSAT est construit comme un système de tissage d'aspects au niveau d'une description d'architecture logicielle. Chaque aspect appelé plan dans notre approche prend en charge une préoccupation et vient modifier une architecture logicielle, appelée plan de base, afin d'y insérer les informations relatives à une nouvelle préoccupation. Le rôle de l'architecte consiste alors à lier le plan de la préoccupation avec une architecture à modifier en précisant les points de jonction où la préoccupation modifie le plan de base pour pouvoir s'intégrer.

Afin de créer une entité réutilisable au sein de plusieurs architectures, ce canevas de conception est centré autour du concept de *patron d'architecture* (Barais *et al.*, 2005b). Le patron d'architecture logicielle contient l'ensemble des informations relatives à une préoccupation : le *plan*, le *masque de points de jonction* et les *règles de transformation* :

– le *plan* représente un assemblage de composants mettant en œuvre les services liés à une préoccupation ;

– le *masque de points de jonction* décrit un ensemble de contraintes sur le lieu sur lequel le nouveau plan peut être intégré. Le masque de points de jonction a un double rôle dans TranSAT : premièrement, c'est le contrat entre le plan de base et le patron d'architecture. Il précise sous quelles conditions ce plan de base et le plan contenu dans le patron peuvent être tissés. Deuxièmement, ses éléments sont utilisés pour décrire les modifications à apporter pour intégrer une nouvelle architecture ;

– enfin, *les règles de transformation* spécifient les modifications à apporter au niveau du plan de base et du plan de ce patron afin de permettre leur tissage. Le résultat du tissage est une architecture logicielle enrichie de la nouvelle préoccupation.

La liaison entre le plan à intégrer et le plan de base est définie en sélectionnant des points de jonction sur le plan de base. De nombreux fragments de ce plan de base respectant le masque de points de jonction sont des points de jonction potentiels pour la préoccupation prise en charge par le patron d'architecture. La sélection de ces points de jonction est appelée *la coupe* par analogie avec le vocabulaire que l'on trouve dans le domaine de la programmation par aspects. Dans TranSAT, de par la nature complexe de ces points de jonction qui peuvent être constitués de nombreux éléments de l'architecture logicielle à transformer, il a été choisi d'appeler ces points de jonction des *lieux d'intégration*. Ce nouveau nom vise à faire la différence entre un point de jonction au sens AspectJ qui représente un point dans le flot d'exécution d'un programme et un lieu d'intégration qui représente un fragment de modèle d'architecture logicielle à partir duquel TranSAT va venir tisser un nouveau plan.

Le résultat d'une transformation est une nouvelle architecture logicielle contenant l'ensemble des éléments du plan de base et du nouveau plan. Cette architecture peut alors servir à nouveau de plan de base pour l'intégration d'une nouvelle architecture. Une nouvelle itération peut alors être effectuée pour intégrer d'autres préoccupations de manière incrémentale.

Une vue générale de l'architecture de TranSAT est présentée à la figure 2. Cette figure illustre le fait qu'un patron d'architecture est composé de trois types d'information : le plan, le masque de points de jonction et les règles de transformation. La coupe sélectionne un ensemble de lieux d'intégration à partir desquels l'architecte souhaite venir tisser le nouveau plan. Le résultat de la transformation peut lui-même être considéré comme un nouveau plan sur lequel l'architecte pourra venir tisser d'autres plans.

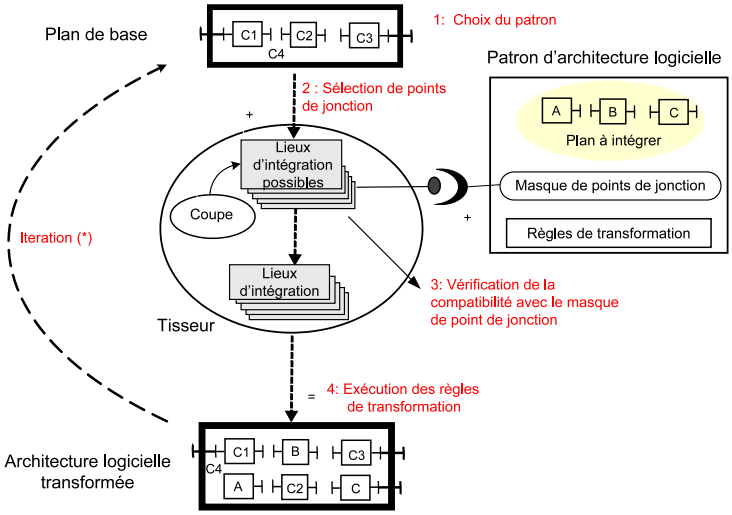

Figure 2. *Vue générale de TranSAT*

3.1.3. *Un langage de transformation d'architecture avec une double personnalité*

La dernière brique constitutive d'un patron d'architecture logicielle concerne les règles de transformation. Ces règles spécifient les opérations à effectuer afin d'intégrer le plan au sein d'une architecture logicielle existante. Elles sont décrites uniquement à partir des éléments du masque de points de jonction et du plan afin de conserver la propriété d'indépendance du patron par rapport au contexte d'intégration. Au niveau des architectures logicielles, la plupart des approches travaillent sur la reconfiguration permettant l'adaptation d'une architecture à son contexte d'exécution ou l'intégration de nouvelles fonctionnalités. Dans ces travaux, des langages dédiés de reconfiguration sont apparus comme Fscript (David, 2005) permettant des adaptations structurelles. Cette focalisation sur les mécanismes de reconfiguration est principalement liée au fait que les plates-formes d'exécution offrent ce seul degré de liberté pour modifier une architecture. Cependant, les apports récents issus du rapprochement entre com-

posants et aspects offrent au développeur des mécanismes de transformation de ces composants bien plus intrusifs permettant la modification locale de l'interface d'un composant. Ces transformations modifient alors aussi bien la structure du composant que son comportement. TranSAT apporte davantage d'abstraction à ces deux types de transformation d'une architecture logicielle : la définition des *reconfigurations* et la définition de l'*introduction*.

D'une manière générale, la reconfiguration considère les composants comme une boîte noire indivisible. Les modifications apportées à l'architecture sont donc des modifications du graphe de composants (ajout, suppression de composants ou de connexions entre composants). La reconfiguration ne casse pas l'encapsulation du composant, elle ne touche ni à son interface, ni à son fonctionnement interne. Au contraire, l'introduction considère qu'il est possible de faire évoluer le composant. L'introduction propose donc des modifications intracomposants. Dans une description d'architecture où le composant est défini par l'intermédiaire de son interface contenant des informations structurelles et comportementales, l'introduction propose des mécanismes pour modifier cette interface en cherchant à maintenir la cohérence entre les informations structurelles et les informations comportementales.

Les règles de transformation spécifient comment tisser un nouveau plan et un plan de base en fonction d'un lieu d'intégration respectant les contraintes définies au niveau du masque de points de jonction. Ces règles définissent les transformations à apporter au plan de base et au nouveau plan ainsi que leur ordre d'application.

Les primitives d'introduction

Le tableau 1 présente les primitives utilisées afin de changer localement la structure d'un composant. Ces primitives permettent de modifier l'interface d'un composant en lui ajoutant de nouveaux ports et de nouvelles opérations, mais aussi en supprimant des ports ou en déplaçant une opération au sein d'un autre port.

	Port	Operation
create	Port *Pr* in *Cp*	Operation *Or* = *op* in *Pr* Operation Or_1 = *op* replaces Or_2
destroy	*Pr*.destroy()	N/A
move	N/A	*Or*.move(*Pr*)

Cp : ComponentRef ; Pr : PortRef ;
Or : OperationRef ; *op* : := *Or* | inverse(*Or*) N/A : Not applicable ;

Tableau 1. *Primitives d'introduction*

Nous ne présentons pas en détail l'ensemble de ces primitives mais la suite de cette section est centrée sur certaines de ces primitives pour illustrer l'esprit du langage.

La primitive $Port\ Pr\ in\ Cp$ permet de définir un nouveau port *Pr* au niveau d'un composant *Cp*. Ce nouveau port ne possède pas d'opération.

La primitive d'ajout d'une opération `Operation Or = op in Pr` insère dans `Pr` une nouvelle opération `Or` possédant la même signature que `op`. Ce type d'ajout d'opération doit nécessairement engendrer une mise à jour du comportement afin de prendre

en compte les nouveaux échanges de messages associés à `Or`. Une règle de transformation modifiant le comportement du composant contenant `Pr` comme celles présentées dans la figure 3 doit donc être définie.

L'ajout d'une nouvelle opération au sein d'un composant est lié à la modification du contrat de comportement de ce composant. Cette modification est spécifiée en utilisant une approche à base de reconnaissance de forme sur le contrat de comportement *patron* $\Rightarrow$ *résultat*. Une telle règle permet d'insérer le message associé avec une nouvelle opération, `op`, avant, après ou autour de l'appel ou de la réponse d'une opération `m`. Le patron spécifie la séquence de messages de `m` qui peut être séparée ou non par une séquence de messages notée `x`. Le résultat spécifie comment les messages associés à l'opération `op` sont insérés par rapport aux messages de `m`. La partie gauche de la règle peut aussi faire référence à des $MessageMask$ définis au sein du masque de points de jonction. La figure 3 montre quelques exemples de telles règles. La première ligne, par exemple, spécifie la réception d'un appel à l'opération `?m` suivie par l'envoi de la réponse à cette requête `!m$`. Le résultat de l'intégration spécifie que juste après la réception de la requête sur `?m`, le composant envoie une requête à `!op` avant d'effectuer le traitement de `m` ($x \rightarrow !m\$$). C'est le cas classique d'une interception avant l'exécution d'une méthode dans une approche objet. Le résultat peut être décrit en utilisant les différents opérateurs du langage SFSP (Barais, 2005) pour la spécification algébrique de comportement. Ainsi la deuxième ligne de la figure 3 spécifie un résultat où l'opération `op` n'est pas systématiquement appelée en cas de réception de requête sur l'opération `m` ($!op \rightarrow ?op\$ \rightarrow x \mid x$).

1 $?m \rightarrow x \rightarrow !m\$ \Rightarrow ?m \rightarrow !op \rightarrow ?op\$ \rightarrow x \rightarrow !m\$$;
2 $?m \rightarrow x \rightarrow !m\$ \Rightarrow ?m \rightarrow (!op \rightarrow ?op\$ \rightarrow x|x) \rightarrow !m\$$;

Figure 3. *Transformation du comportement*

Les primitives `Operation` $\texttt{Or}_1$ `= op replaces` $\texttt{Or}_2$ et `Or.move(Pr)` ajoutent aussi une opération à un port. Dans ces cas-là, cependant, la mise à jour du contrat de comportement est automatique. Dans le cas du `replace`, chaque message associé à $\texttt{Or}_2$ dans le contrat de comportement est remplacé par le message correspondant au niveau de $\texttt{Or}_1$. Les opérations doivent avoir la même polarité afin d'assurer que ce remplacement soit valide au niveau de l'automate de comportement.

Les primitives de reconfiguration

Les primitives de reconfiguration prennent en charge, quant à elles, les modifications des interactions entre composants lors de l'intégration d'une nouvelle préoccupation. Ces modifications concernent la création et la destruction de connexions, la création et la destruction de composites et l'ajout et le retrait de composants au sein d'un composite. Le tableau 2 présente les différentes primitives de reconfiguration que l'on peut utiliser au niveau des règles de transformation.

	Component	Connexion	Composite
create	N/A	Connexion $Br = \{Pr_1, Pr_2\}$	Composite Cr Composite Cr_1 in Cr_2
destroy	N/A	Br.destroy()	Cr.destroy()
move	Cp.move(Cr)	N/A	Cr_1.move(Cr_2)

Cp : ComponentRef ; Cr : CompositeRef ; Pr : PortRef ;
Br : ConnexionRef ; N/A : Not applicable ;

Tableau 2. *Primitives de reconfiguration*

Toujours avec pour objectif de proposer un langage de transformation possédant un fort pouvoir d'expression pour l'architecte, la création des connexions est toujours faite de manière relative dans les règles de transformation. La primitive Connexion $Br = \{Pr_1, Pr_2\}$ connecte un port Pr_1 et un port Pr_2 même si ceux-ci n'appartiennent pas à des composants d'un même composite. Le tisseur se charge de créer les ports délégués et les délégations pour que cette interaction puisse avoir lieu.

3.2. *Application à l'exemple*

L'adaptation de l'exemple au monde des composants nécessite quelques adaptations. En effet, le diagramme d'architecture travaille sur des instances de composants en relation. L'architecture de l'agence de location est alors construite autour de trois composants principaux (cf. figure 4) : l'agence, l'entrepôt de voitures et l'entrepôt de clients. Ces trois composants interagissent au travers de ports. L'agence peut ainsi créer de nouvelles voitures ou de nouveaux clients ou simplement consulter ses stocks. Seul le comportement du composant « entrepôt de voitures » est détaillé. Chaque opération peut s'exécuter alternativement selon un choix indéterministe.

Intégration d'un patron fonctionnel

Nous illustrons TranSAT sur l'intégration d'un premier patron dit fonctionnel car il permet d'ajouter un service de location à l'architecture de l'agence. Ce service nécessite l'intégration de deux composants « gestionnaire location » et « IHM ». Ces composants sont inclus dans le plan du patron, ce sont les éléments d'architecture à intégrer au sein d'une spécification d'architecture. Les contraintes d'intégration de ce patron sont faibles, le patron s'intègre pour deux composants possédant chacun une opération. Ces contraintes sont exprimées sous la forme d'un masque de points de jonction présenté dans la figure 5 (partie gauche). Finalement, les règles d'intégration spécifient la création d'un nouveau port et d'une nouvelle opération sur les composants A et B du masque de points de jonction. Le comportement de A et B est modifié pour l'ajout de la prise en compte des messages de la nouvelle opération.

La liaison entre le plan de base (figure 5) et le patron est définie en ajoutant des contraintes au niveau du masque de points de jonction, dans notre cas

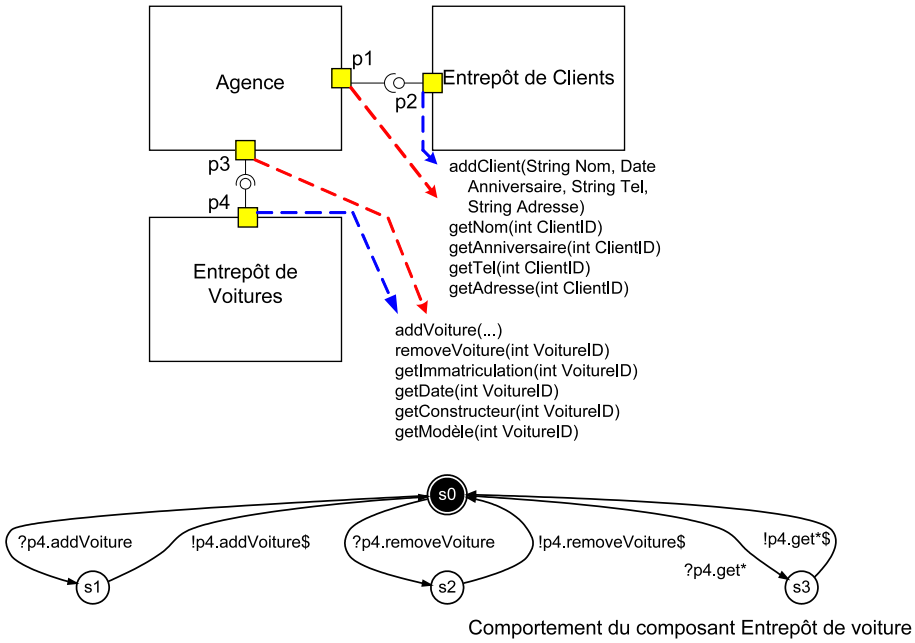

Figure 4. *Exemple d'architecture logicielle à base de composants*

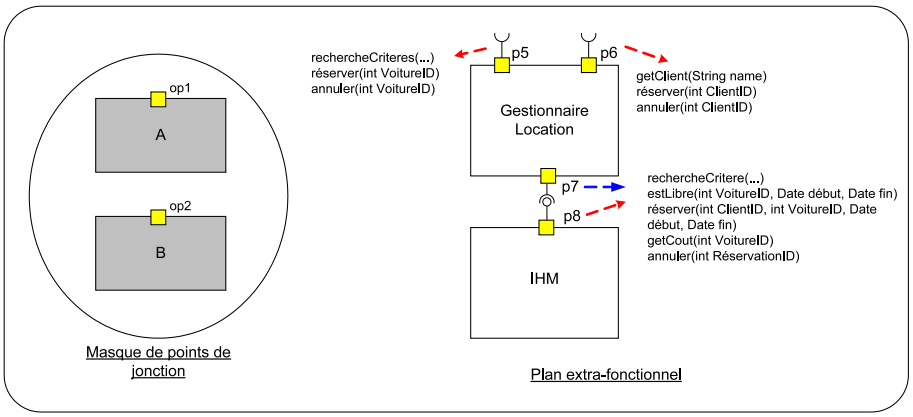

Figure 5. *Exemple de patron d'architecture*

`op1=p4.addVoiture` $\wedge$ `op2=p2.addClient`. Le *framework* TranSAT recherche alors tous les lieux d'intégration compatibles. Pour l'exemple, un seul lieu concorde. Le *framework* exécute alors les règles de transformation pour intégrer le nouveau plan. Le résultat est présenté dans la zone (a) de la figure 6.

Intégration d'un patron technique

TranSAT a pour but d'être un *framework* général pour la construction incrémentale de description d'architecture logicielle. Dans ce sens, il permet l'intégration de préoccupations dites fonctionnelles comme de préoccupations dites techniques ou extrafonctionnelles. Dans l'exemple, le deuxième patron que nous souhaitons intégrer vise à permettre la notification de l'IHM en cas de modifications de la base des clients ou de la base des voitures. Nous n'expliciterons pas précisément ce patron. Cependant, il est intéressant de noter que ce patron a une spécificité : il ne possède pas de plan, c'est-à-dire qu'aucun élément d'architecture n'est ajouté. La modification du comportement se fait en ajoutant un appel à l'opération `MiseAJour`, insérée par le patron, après chaque création d'un nouvel élément. Le masque de points de jonction est identique au masque de points de jonction du patron précédent.

La liaison entre le patron et le plan de base se fait par l'intermédiaire d'une coupe qui identifie deux lieux d'intégration : `(op1=p8.estLibre` $\wedge$ `op1=p4.addVoiture)` $\vee$ `(op1=p8.estLibre` $\wedge$ `op2=p2.addClient)`. Le *framework* TranSAT recherche alors tous les lieux d'intégration compatibles. Pour l'exemple, deux lieux concordent. Il exécute les règles de transformation pour intégrer le nouveau plan. La figure 6 (b1 et b2) présente le résultat de cette intégration. Le nouveau plan est accroché aux deux points de jonction identifiés par l'expression de coupe définie précedemment.

4. Composant de modèle

La deuxième approche étudiée propose le concept de composant de modèle. Elle trouve son origine dans le rapprochement des notions de vues et de composants ; la première apportant le principe de réutilisation et de configuration, la seconde, une méthode de structuration cohérente.

4.1. *Objectif*

L'objectif de cette approche est de permettre la construction de systèmes par assemblage de fonctionnalités, chacune décrite à l'aide d'un modèle générique appelé composant de modèle (Muller, 2006). Dans cette approche, ces modèles sont eux-mêmes paramétrés par un modèle afin de permettre l'expression d'une partie requise complexe. On dépasse ainsi la notion de contrat d'assemblage de composants souvent réduite à une interface de services unitaires.

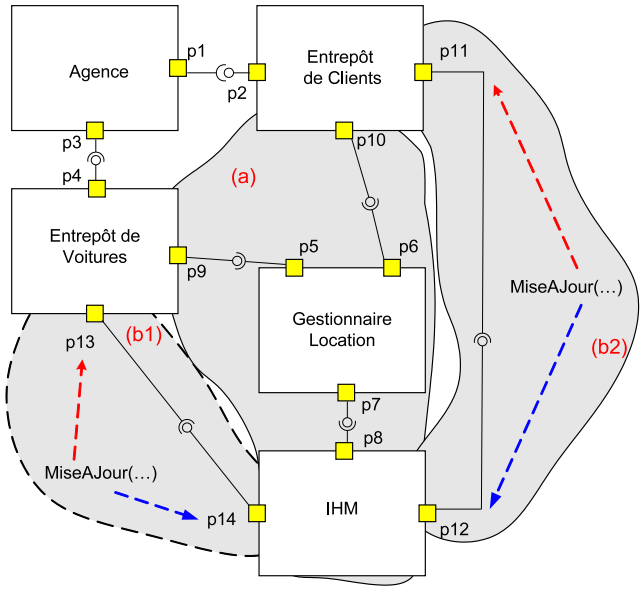

Figure 6. *Résultat du tissage de deux patrons sur une architecture logicielle*

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

4.2. *Application à l'exemple*

Les figures 7, 8 et 9 illustrent un ensemble de composants de modèle pour l'exemple introduit dans la section 2. Chaque composant, exprimé sous la forme d'un paquetage, correspond à une fonctionnalité du système. Le modèle requis de chacun d'eux est explicité dans le coin supérieur droit. Le schéma figuré à l'intérieur du paquetage de chaque composant correspond à son modèle fourni. Celui-ci est paramétré par le modèle requis qui y est ici représenté à l'aide des éléments en pointillés et italique.

La figure 7 illustre un composant pour la fonctionnalité de gestion des stocks. Celui-ci est défini à partir des concepts de stock et de ressource exprimés dans son modèle requis. Le concept de stock doit disposer d'un attribut matérialisant son identifiant et celui de ressource d'un attribut matérialisant sa référence (*ref*). Ces deux

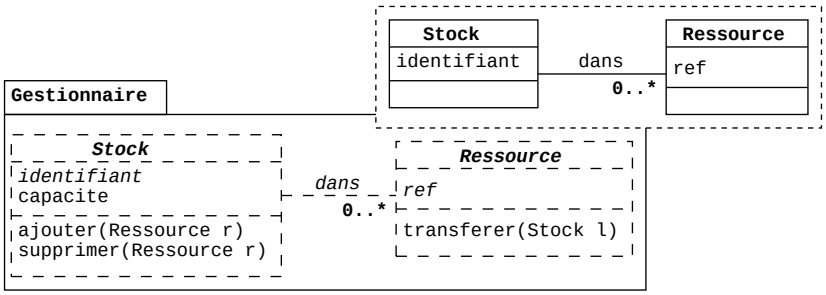

Figure 7. *Composant de gestion des ressources*

concepts doivent également être reliés par une association (*dans*). La fonctionnalité de gestion des stocks est donc exprimée à partir de ce modèle et définit des opérations d'ajout, de suppression et de transfert de ressources. Un attribut permettant de définir la *capacité* dans *stock* est également ajouté.

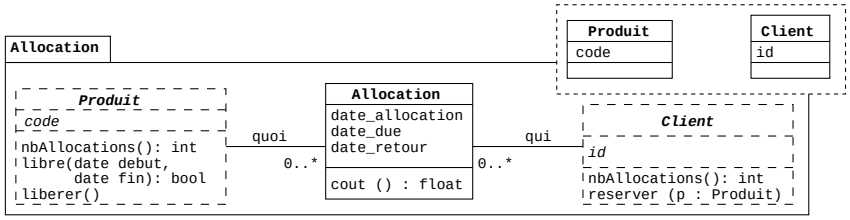

Figure 8. *Composant d'allocation de ressources*

Le composant d'allocation de ressources (cf. figure 8) illustre la possibilité de définir un modèle requis non connexe. Pour appliquer ce composant, il est nécessaire de disposer de produits et de clients, les classes correspondantes à ces concepts pouvant être totalement dissociées. Ce composant a pour but d'allouer des produits à des clients et installe pour cela une relation entre ces entités. Ce composant illustre donc la possibilité de fournir de nouvelles classes, et de nouvelles associations. La classe *Allocation* est ajoutée entre les produits et clients à l'aide de deux associations. Cette classe dispose d'attributs pour la date d'allocation, la date de retour prévue et la date de retour effective, et d'une opération permettant le calcul du coût de l'allocation. Des opérations de gestion des allocations sont également ajoutées aux classes *Produit* et *Client*.

Enfin, notre dernier exemple de composant de modèle (cf. figure 9) définit une fonctionnalité correspondant au patron observateur. Celui-ci définit la structure de ce patron par rapport à son modèle requis, composé de la classe *Sujet* disposant d'un état et d'une classe *Observateur* notifiée du changement de ce dernier.

Lorsque l'on dispose d'un ensemble suffisant de fonctionnalités génériques exprimées à l'aide de composants de modèle, la construction d'un nouveau système peut

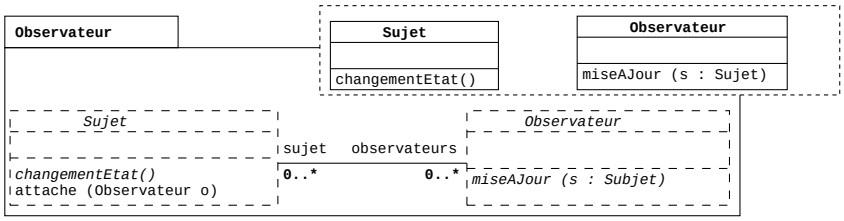

Figure 9. *Composant observateur*

être réalisée grâce à leur assemblage. Cet assemblage doit se baser sur un modèle initial. Celui-ci, appelé modèle de base, permet de définir les concepts et les associations propres au domaine du système à construire.

Pour spécifier cet assemblage un opérateur d'application de modèles paramétrés est défini. Cet opérateur permet de mettre en correspondance le modèle requis par un composant avec le modèle fourni par un autre (ou par la base). La fonctionnalité exprimée par le composant est ainsi paramétrée par ce modèle requis. Le modèle fourni doit au moins contenir un sous-modèle conforme au modèle requis, il peut cependant contenir d'autres éléments.

C'est cette notion de conformité entre modèle fourni et modèle requis qui permet de garantir la validité d'une composition. Un modèle est conforme à un autre si les éléments qu'il fournit respectent la même structure que les éléments du modèle requis.

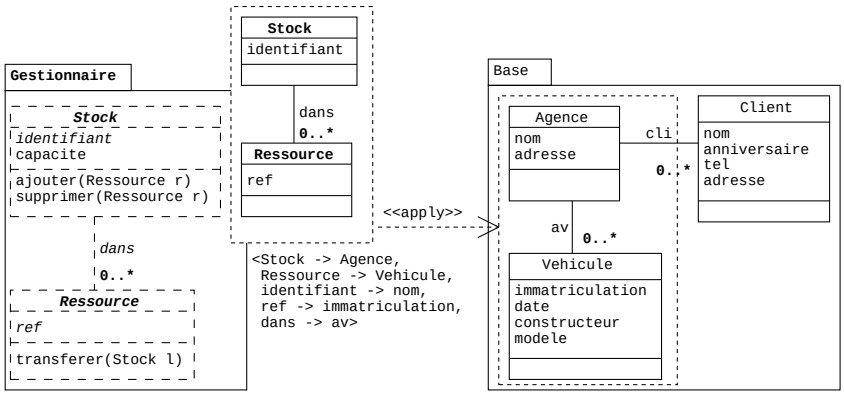

Figure 10. *Application de gestion des stocks au système de location de véhicules*

La figure 10 donne un exemple d'application du composant de modèle gestionnaire au modèle de base pour lui ajouter la fonctionnalité de gestion des stocks. Cette application est spécifiée à l'aide de la relation stéréotypée «apply» qui établit les relations de correspondance entre le modèle requis et le modèle fourni. Les classes *Stock* et *Ressource* sont respectivement mises en correspondance avec les classes *Agence* et

Vehicule. De même, leur attribut *identifiant* et *ref* sont mis en relation avec les attributs *nom* et *immatriculation*.

Une formulation possible du résultat de cette application est présentée par la figure 11. Dans cette formulation, des éléments paramètres de la fonctionnalité sont mis en relation avec les éléments du modèle de base correspondant à l'aide de liens stéréotypés «trace» (permettant de lier des éléments dénotant le même concept). L'approche autorise d'autres formulations du résultat, comme la fusion du composant avec le modèle de base ou l'utilisation de vues (Muller *et al.*, 2003).

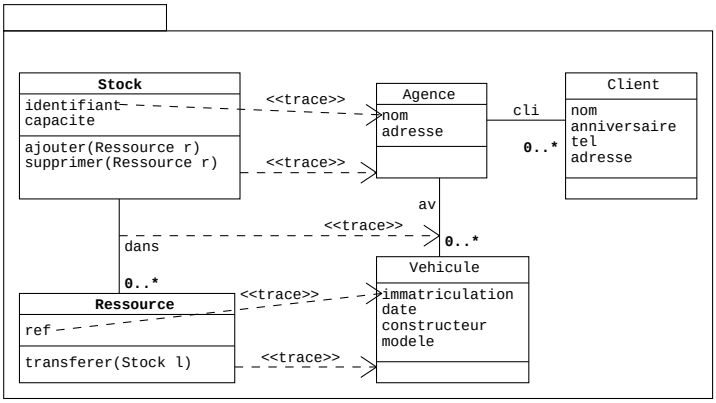

Figure 11. *Système de base avec les fonctions de gestion des véhicules*

Pour supporter l'évolution incrémentale des systèmes, l'approche permet l'application d'un composant de modèle à un autre. Cette possibilité permet la construction de nouveaux composants de modèle complexes à partir de composants plus simples. Elle permet également l'expression de chaînes d'applications. Une telle chaîne est illustrée figure 12 par l'application du composant *Observateur* au composant *Allocation*, lui-même appliqué à la base. Cette première application permettant d'ajouter une fonctionnalité de réservation automatique dès qu'un véhicule devient disponible.

Cette figure illustre également la possibilité d'appliquer à une même base un ensemble de composants de modèle. Un ensemble de propriétés définies pour l'opérateur *apply* permet alors de garantir la cohérence du système résultat quel que soit l'ordre d'évaluation des différentes applications. Les applications des composants *Gestionnaire* et *Allocation* à la base donnent ainsi le même résultat quel que soit leur ordre d'évaluation (Muller *et al.*, 2005a).

L'approche autorise également l'utilisation d'un même composant pour différentes parties du même système. Cette possibilité est ici illustrée par l'application du composant *Gestionnaire* pour la gestion des véhicules et pour la gestion des clients.

Afin de permettre l'expression des composants de modèle et de leur assemblage à l'aide du langage standard UML, une extension du métamodèle UML a été réalisée. Les composants de modèle sont ainsi définis à partir du concept de paquetage *template*

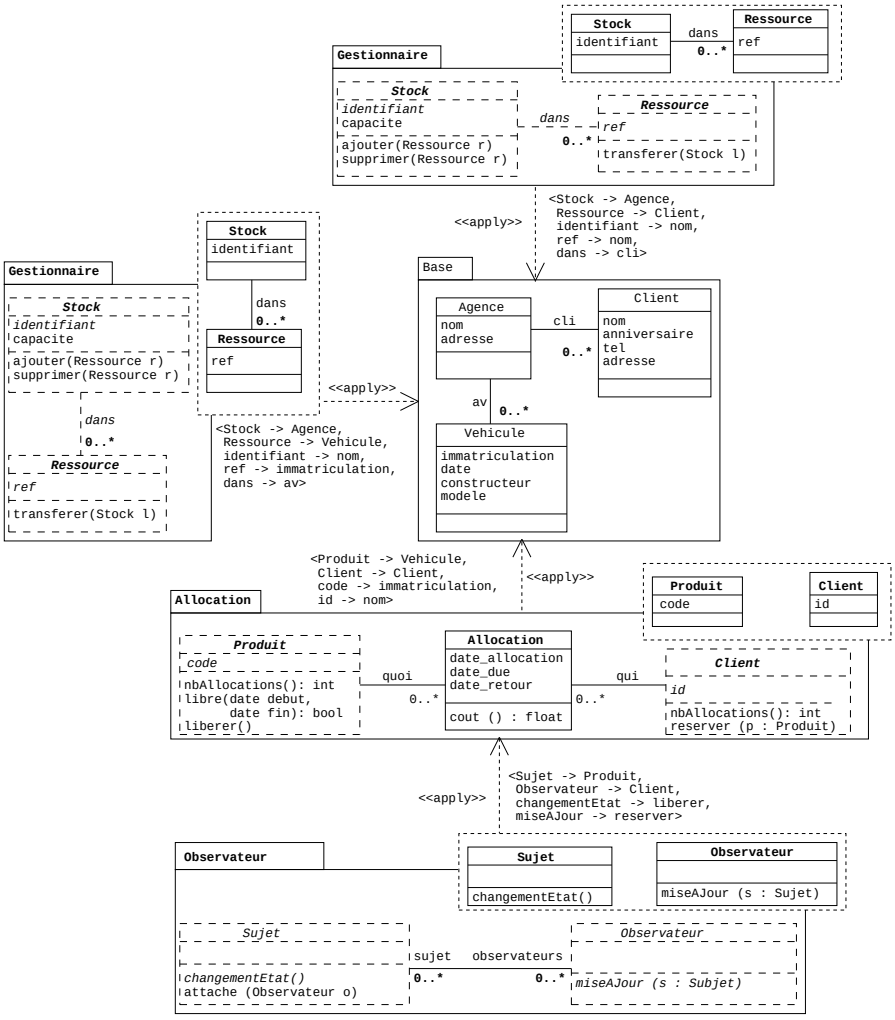

Figure 12. *Assemblage du système de locations de véhicules*

auquel est ajouté un ensemble de contraintes afin de garantir que son paramétrage forme bien un modèle. La relation pour exprimer l'application d'un composant de modèle (*apply*) est également définie par extension du métamodèle standard UML. Les contraintes définies (en OCL) permettent de garantir la conformité entre modèle requis et modèle fourni.

Les principaux avantages attendus de l'approche sont la réduction des temps de conception et l'amélioration de la qualité des systèmes ainsi construits. Ces objectifs sont atteints grâce à la réutilisation de composants de modèle éprouvés. De plus la préservation du découpage selon les différentes préoccupations permet une meilleure

maîtrise de la complexité. Enfin, cette approche permet également l'évolution des systèmes par ajout de nouvelles fonctionnalités sans remise en cause du modèle déjà construit.

5. SmartAdapters

SmartAdapters s'appuie sur des travaux relatifs à la réutilisation de bibliothèques de classes (Quintian, 2004; Lahire *et al.*, 2006). Cette approche a été transposée afin de l'adapter à des modèles (Crescenzo *et al.*, 2005).

5.1. *Objectif*

SmartAdapters a pour vocation d'aider un concepteur à réaliser des modèles par assemblage de modèles ou fragments de modèles existants. L'approche s'appuie sur un métamodèle qui représente une description de la composition et sur un langage dédié à ce métamodèle. Ce dernier repose en particulier sur une réification de la notion d'adaptation. Une adaptation correspond à une ou plusieurs opérations à réaliser pour intégrer une préoccupation dans une application ou un modèle.

Une préoccupation et les adaptations à réaliser pour l'insérer dans une application ou un modèle sont encapsulées dans un conteneur que SmartAdapters nomme paquetage (au sens général d'*ensemble de classes*). Il regroupe la totalité des classificateurs développés pour cette préoccupation. La réification des classificateurs est traditionnelle : chacun possède un nom, d'éventuels modificateurs (*public*, *deferred*, *final*, *frozen*...), des membres (attributs, attributs de classe, méthodes, méthodes de classe, constructeurs, destructeurs, initialiseurs...) et des assertions (invariants de classe). Les méthodes ont une signature (nom, modificateurs, paramètres, éventuel type de retour), des assertions (préconditions et postconditions) et un corps. A l'heure actuelle, le comportement associé à un corps de méthode n'est pas réellement pris en compte.

Chaque adaptation est localisée dans un adaptateur et se trouve donc hors des classificateurs qu'elle adapte, il s'agit bien d'une démarche non intrusive. Une composition est constituée d'un ou plusieurs adaptateurs, en général plusieurs. Chaque adaptateur est identifié par un nom unique ; un adaptateur peut être abstrait ou concret et peut hériter (au sens de la spécialisation) d'un autre adaptateur. Chaque adaptateur référence des cibles d'adaptation (*i.e.* les entités sur lesquelles les adaptations vont s'appliquer). Chaque adaptation est typée en fonction du type de l'entité sur laquelle elle doit s'appliquer. Il y a donc des adaptations pour les classificateurs, pour les méthodes, pour les attributs, les associations, etc.

Ces adaptations peuvent être concrètes ou abstraites : les adaptations abstraites permettent une spécification *a posteriori* des cibles, au moment où l'adaptation est effective, c'estàdire quand la cible réelle est connue. Lorsqu'une cible est abstraite, il est possible de lui associer des contraintes qui devront être vérifiées lors de sa concrétisation. Pour concrétiser une cible d'adaptation, trois options s'offrent aux program-

meurs : citer une cible, citer une liste de cibles ou encore donner une expression régulière qui identifie un ensemble de cibles. Dans les deux derniers cas, cela permet d'appliquer l'adaptation à plus d'une cible.

Du point de vue du programmeur d'applications la description des adaptations à réaliser est faite à travers l'utilisation d'un langage de surface construit au-dessus du métamodèle. C'est ce langage qui sera utilisé pour illustrer l'approche sur un exemple.

Les adaptations qu'il est possible de réaliser sur le code d'une application sont nombreuses et SmartAdapters propose une hiérarchie qui en décrit une classification dont le degré de précision peut encore être augmenté ; elle regroupe quatre grandes sortes d'adaptation :

– les *interceptions* : il s'agit d'intervenir lors de l'appel ou de l'accès aux membres (attributs, méthodes...) de classificateur. Ces adaptations permettent d'ajouter du code à une méthode au début, à la fin ou autour du code existant, ou encore d'ajouter un comportement dans le cas d'une levée d'exception. Pour les attributs, les interceptions peuvent se faire lors de l'accès en lecture ou lors de l'accès en écriture ;

– les *ajouts* concernent aussi bien les membres de classificateur (ajout d'attribut, de méthode, d'association...) que la possibilité d'indiquer un nouveau superclassificateur. Il est aussi possible d'ajouter un invariant de classificateur ou une assertion (précondition ou postcondition) dans une méthode ;

– les *fusions* pour les méthodes et les classificateurs : les fusions de méthodes sont possibles en choisissant lequel des deux corps doit être exécuté avant l'autre dans la méthode résultante. Les fusions de classificateurs (prendre deux classes pour n'en faire qu'une, par exemple) sont soit simples (aucun conflit n'apparaît) soit doivent être décrites plus précisément lors de la composition en faisant correspondre explicitement les primitives à fusionner ;

– les *redéfinitions* d'attribut, d'association ou de méthode doivent notamment respecter un certain nombre de règles qui limitent les erreurs possibles.

Dans l'approche SmartAdapters, chaque adaptation peut être déclarée *in situ* (dans ce cas, elle est réalisée par modification des classificateurs cibles) ou *ex situ* (de nouveaux classificateurs sont alors créés pour intégrer les éléments de l'adaptation dont les cibles ne sont pas modifiées directement).

5.2. *Application à l'exemple*

Pour illustrer cette approche on considère essentiellement deux préoccupations : la préoccupation « *allocation* » et la préoccupation «*observateur*». Ces préoccupations sont jugées réutilisables et à ce titre un cahier de réutilisation leur est associé. Il a vocation à favoriser leur réutilisation en guidant et contrôlant le concepteur d'applications.

5.2.1. *Préoccupation Observateur*

La figure 13 décrit le protocole de composition associé à la préoccupation *Observateur*. Ce protocole est adapté au contexte dans la section 5.2.3.

```
01 concern bibliotheque.designpattern.observer
02 abstract adapter ObservateurAdaptateur {
03
04   abstract Class target " Classe(s) observable(s) " : observableClass
05   abstract Method target " Notification à l'observateur " : notifyingMethod
06     require notifyingMethod in observableClass .*
07
08   adaptation becomeObservable " Modifie la classe pour la rendre observable " :
09     extend class ImplémentationObservable with observableClass
10   adaptation notifyingObserver " Modifie notifyingMethods pour transmettre
11     les modifications à l'observateur" :
12     extend method notifyingMethod ( ... ) with after { notifierObservateurs() ; }
13 ---------------------------------------------------------------------------------
14   abstract Class target " classe utilisée comme observateur " : observerClass
15
16   adaptation becomeObserver " Modifie la classe pour en faire un observateur " :
17     inherit Observateur in ObserverClass
18   abstract adaptation observerUpdate " Introduit miseAjourObservateur dans l'observervateur " :
19     concretize method public void miseAJourObservateur(Observable o) in observerClass
20 ---------------------------------------------------------------------------------
21   abstract Class target " Classe(s) initialisant les objets
22     observables ou observateurs " : applicationInitClass
23   abstract Method target " Methode(s) créant les objets observables ou
24     observateurs " : applicationInitMethod
25     require applicationInitMethod in applicationInitClass .*
26   abstract Attribute target
27     " Attributs(s) pointant(s) sur des objets observables " : observableInstance
28     require observableInstance in applicationInitClass .*
29   abstract Attribute target
30     " Attributs(s) pointant(s) sur des objets observateurs " : observerInstance
31     require observerInstance in applicationInitClass .*
32
33   adaptation initApplication " Modifie applicationInitMethod pour garantir la présence
34     des observateurs dans la liste des observables " :
35    extend method ApplicationInitClass . applicationInitMethod (...) with post
36      ObservableInstance->forAll( v : ObserverClass |
37          v.observateurs.includes (ObserverInstance))
38 }
```

Figure 13. *Protocole de composition observeur/observable*

Les informations contenues dans la figure 13 sont indépendantes du futur contexte d'utilisation. Les lignes *01* et *02* indiquent que le protocole de composition *ObservateurAdaptateur* est associé à la préoccupation *Observateur*. Le mot-clé *abstract* apparaît avant le mot-clé *adapter* parce que l'adaptateur contient des adaptations abstraites.

L'adapteur est composée de trois parties : *i)* les entités observables, *ii)* les entités qui observent et, *iii)* l'initialisation de la préoccupation qui utilise le patron de conception. Les lignes *04* à *13* décrivent les cibles et les adaptations associées aux entités observables. Les cibles d'adaptation *observableClass* et *notifyingMethod* concernent respectivement les entités observées et les méthodes chargées d'informer les entités qui observent d'éventuels changements. Au moment où le protocole de composition est spécifié, on ne connaît pas encore ces méthodes et classes, c'est pourquoi les cibles sont précédées par *abstract*. Elles deviendront concrètes lors de la composition avec le

modèle choisi (une agence de location dans notre exemple). Deux adaptations sont décrites : une de type *fusion* qui garantit que l'on ne fait qu'ajouter des méthodes (*extend class*) et une autre qui ajoute un traitement complémentaire à une méthode (*extend method*). La première adaptation indique que le contenu de la classe *ImplémentationObservable* doit être inséré dans les classes qui sont observées. La deuxième adaptation décrit l'insertion d'un appel à la méthode identifiée par *notifierObservateurs* à la fin des méthodes qui correspondent à *notifyingMethod*. Ces méthodes doivent appartenir (mot-clé *require*) aux classes observées (*observableClass*). L'appel à la méthode *notifierObservateurs* permet aux classes qui jouent le rôle d'observateur d'être averties des changements éventuels et ainsi de déclencher l'exécution d'un traitement.

Les lignes *14* à *20* décrivent la cible d'adaptation et deux adaptations portant sur des classes dont les instances « observent » celles qui sont concernées par les lignes *4* à *13*. La cible d'adaptation *observerClass* fait référence aux classes choisies pour ce rôle. La première adaptation ajoute la classe abstraite *Observateur* aux classes représentées par la cible d'adaptation *observerClass*. La classe *Observateur* contient une méthode abstraite *miseAjourObservateur* ce qui implique que la seconde adaptation est elle aussi abstraite et qu'un comportement approprié (action à exécuter sur les entités qui observent) devra y être attaché lorsque l'on connaîtra le contexte d'utilisation.

Les lignes *21* à *37* décrivent l'initialisation nécessaire à l'utilisation du patron de conception dans son contexte d'intégration. Cette initialisation concerne le remplissage de la collection des objets qui observent. Cette collection correspond à l'association *observateurs* qui pourra être initialisée par l'utilisation de la méthode *ajoutObservateur* (classe *ImplémentationObservable*). Quatre cibles d'adaptation et une adaptation sont spécifiées pour implémenter l'initialisation. Il est intéressant de noter que l'on peut faire référence aux cibles d'adaptations dans la description d'une adaptation.

5.2.2. *Préoccupation sur l'allocation de produit*

Comme pour la préoccupation décrite dans la section 5.2.1 nous proposons dans la figure 14 un protocole de composition pour la préoccupation « *allocation* »qui est une préoccupation fonctionnelle. Il permet de mettre en évidence que pour certaines adaptations comme par exemple pour la fusion de classificateurs il est nécessaire de proposer une manière de spécifier la correspondance (Clarke, 2001) entre les méthodes, les attributs et les associations de ces classificateurs. Il y a deux situations différentes : soit on est capable d'isoler des classes, des méthodes, des attributs ou des associations qui jouent un rôle particulier et il faut leur associer des cibles d'adaptation dédiées, soit le processus de fusion est plus général et il faut définir des règles de correspondance plus génériques.

Plus précisément, deux classificateurs sont susceptibles de fusionner avec d'autres : il s'agit de *Produit* et de *Client*. Les cibles d'adaptations *produitClass* et *clientClass* (lignes *12* et *13*) permettent de définir les entités qui joueront le rôle des produits et des clients. Cependant il faut maintenant spécifier comment doit s'établir la correspondance à l'intérieur des classes (c'est à dire pour les associations, attributs

et méthodes qu'elles contiennent). Dans notre cas on juge que les critères permettant d'établir la correspondance sont le nom de la méthode (*nameMeth*), le type éventuel de retour de la méthode (*typeMeth*), le nombre de paramètres de la méthode s'il y en a (*nombrePar*) et le cas échéant la position des paramètres (*positionPar*) ou leur type (*typePar*). C'est au moment de l'intégration que le concepteur décidera de la valeur à donner pour chacun de ces critères : *i)* une ou plusieurs valeurs spécifiques, *ii)* la même valeur pour la source et la cible de la fusion, *iii)* la même valeur ou un groupe de valeurs augmenté d'un préfixe ou d'un suffixe ou, *iv)* une valeur quelconque.

```
01 concern bibliotheque.allocation
02 abstract adapter AllocationAdaptateur {
03   abstract Method binding "règle de correspondance" : methodBinding
04     require exists criteria : nameMeth, typeMeth, nombrePar, positionPar, typePar
05   abstract Attribute binding "règle de correspondance" : attributeBinding
06     require exists criteria : nameAtt, typeAtt
07   abstract Association binding "règle de correspondance" : associationBinding
08     require exists criteria : nameAssoc typeAssoc
09
10   abstract Class target "documents à composer" : produitClass
11   abstract Class target " clients à composer" : clientClass
12
13   adaptation buildProduits "ajoute les fonctionnalités du Produit pour l'allocation " :
14     merge class Produit with produitClass
15       require attributeBinding , methodBinding, associationBinding
16   adaptation buildClients " ajoute les fonctionnalités du client " :
17     merge class Client with clientClass
18       require attributeBinding , methodBinding, associationBinding
19 }
```

Figure 14. *Protocole de composition de l'allocation des produits*

5.2.3. *Composition des préoccupations*

On va maintenant s'appuyer sur les deux protocoles de composition définis ci-dessus pour décrire la composition qui permettra de compléter le modèle sur la gestion de l'agence de location (appelé modèle de base dans la figure 1). Le modèle final sera donc constitué de ce modèle de base, du modèle « *Allocation* »(voir figure 8) et du modèle «*Observateur*» (voir figure 9).

Dans la figure 15 on complète la définition du protocole de composition proposée à la figure 13 pour l'adapter à l'agence de location. De même, on complète la définition de l'adaptation, jusque là incomplète car l'adaptation est dépendante du contexte d'intégration. En particulier on veut ici utiliser le patron de conception Observateur pour que l'agence observe les véhicules dans le but d'insérer un véhicule dans une liste de véhicules à revendre lorsque celui-ci dépasse un certain nombre de kilomètres.

Dans la figure 15 le mot-clé *extends* de la ligne *03* montre que l'adaptateur *ApplicationAgence1* est bien une spécialisation (c'est même une concrétisation puisque l'adaptateur n'est plus déclaré *abstract*). Les cibles d'adaptations sont maintenant définies (lignes *04* à *10*). Par exemple on spécifie que les entités observables sont des instances de la classe *Vehicule* alors que les entités qui observent sont des instances de la classe *Agence*. De même, l'adaptation qui n'était pas complètement définie car

```
01 concern application.agence
02 compose bibliotheque.designpattern.observer.{*} with application.agence
03 adapter ApplicationAgence1 extends ObservateurAdaptateur {
04   target observableClass = application.agence.Vehicule
05   target observerClass = application.agence.Agence
06   target notifyingMethod = application.agence.Vehicule.majKilometrage()
07   target applicationInitClass = application.agence.Agence
08   target applicationInitMethod = applicationInitClass.Agence()
09   target observableInstance = application.agence.Agence.vehicules
10   target observerInstance = application.agence.Agence.self
11
12  adaptation observerUpdate :
13      concretize method public void miseAjourObservateur(Observable o) in observerClass with {
14          ajouteVehivuleVieux((Vehicule) o) ;
15      }  post self.vehiculesVieux.includes ((Vehicule) o)
16 }
```

Figure 15. *Adaptation au contexte de la préoccupation allocation de produits*

```
01 concern application.agence
02 compose bibliotheque.location.{*} with application.agence
03 adapter ApplicationAgence2 extends AllocationAdaptateur {
04   binding bindingMethod = nameMeth (=),typeMeth(=),nombreP(=),positionP(=),typeP(covariant)
05   binding bindingAttribute = nameAtt(=), typeAtt(covariant)
06   binding bindingAssociation = nameAssoc(=), typeAssoc(covariant)
07   target produitClass = application.agence.Vehicule
08   target clientClass = application.agence.Client
09  }
```

Figure 16. *Adaptation au contexte de la préoccupation Observeur/Observable*

l'action à réaliser dépend du contexte d'intégration est complétée (lignes *12* à *15*).

La figure 16 complète le protocole de composition de la figure 13. On définit les aspects de l'intégration d'un modèle d'allocation de produit en général qui sont propres à la location de véhicules. Les règles de correspondance sont en particulier définies. Par exemple deux méthodes seront fusionnées si leur nom et leur type de retour éventuel sont les mêmes, si le nombre de leurs paramètres est identique et si les paramètres de même position ont le même type. Cet adaptateur modifie la préoccupation décrivant la gestion d'une agence offrant des véhicules à des clients pour y intégrer la seconde préoccupation.

6. Evaluation comparative

L'objectif de cet article est de présenter ces trois approches supportant la séparation des préoccupations en phase de conception mais surtout de proposer un canevas pour l'évaluation de ces trois approches en permettant de mettre en valeur les points communs entre les vues, les sujets et les aspects et en dégageant les points-clés pour le support de la séparation des préoccupations en phase de conception.

Par conséquent, dans cette section nous réalisons une comparaison des différentes

approches présentées ci-avant, selon plusieurs axes et critères. Nous avons choisi de reprendre et d'étendre des critères élaborés par le réseau d'exellence par AOSD-EUROPE (Chitchyan *et al.*, 2005). Les critères proposés sont :

– les capacités d'abstraction des détails de différents domaines ;

– le niveau de séparation des préoccupations ;

– la traçabilité ;

– la généralité des capacités de composition ;

– la facilité d'évolution des architectures ;

– la capacité de maîtrise de la complexité pour des modèles et des applications de taille importante.

6.1. *Capacités d'abstraction*

Par abstraction nous entendons ici la capacité de masquer des détails non pertinents selon un certain point de vue et à un instant donné du processus.

Trois types d'abstraction sont pris en considération dans cette sous-section :

1) l'abstraction de l'architecture de mise en œuvre ; il s'agit de l'indépendance du procédé de composition vis-à-vis des détails d'implantation ; le procédé est-il réservé à la conception ou bien est-il applicable à toutes les étapes ;

2) l'abstraction sur l'environnement du fragment composable : quelles sont les hypothèses minimales sur les propriétés des autres fragments, le procédé impose-t-il une forme particulière aux fragments, non pour être composé mais pour interopérer avec les autres fragments ? Ce critère est particulièrement important pour la réutilisabilité des aspects ;

3) l'abstraction du métamodèle des fragments : le procédé de composition est-il plus ou moins dépendant du métamodèle sous-jacent aux éléments composables ?

6.1.1. *TranSAT*

Dans l'approche TranSAT, la séparation des préoccupations se concentre sur le modèle de l'architecture de l'application. TranSAT est donc principalement utile en phase de conception et ne peut s'appliquer sur d'autres phases du développement logiciel. Concernant le deuxième critère d'abstraction, TranSAT introduit la notion de masque de point de jonction. Un masque de point de jonction est un modèle qui contraint le contexte dans lequel le nouveau fragment peut être intégré. Les règles de transformation du patron qui modifient l'architecture cible afin de permettre l'intégration du nouveau plan sont spécifiées à partir des éléments du masque de point de jonction, ce qui permet la réalisation de patron d'architecture pour plusieurs contextes d'intégration. Actuellement, TranSAT est pleinement lié au modèle de composants SafArchie. Par conséquent les mécanismes de composition comme le masque de point

de jonction ou le langage de transformation sont dédiés au modèle de composants SafArchie.

6.1.2. *Composant de modèle*

Dans l'approche par composant de modèle, la partie requise pour utiliser une fonctionnalité est décrite à l'aide d'un modèle. Cette fonctionnalité est ainsi définie par rapport à ce modèle requis, ce qui permet la réalisation de composants de modèle génériques indépendants d'un contexte d'utilisation particulier. Il est également possible de composer les fonctionnalités tout en conservant leur caractère générique, autorisant ainsi la construction de fonctionnalités réutilisables complexes.

6.1.3. *SmartAdapters*

Concernant le niveau d'abstraction, on peut considérer la modélisation des préoccupations mais aussi la modélisation de la composition. SmartAdapters n'apporte pas d'amélioration par rapport à la modélisation des préoccupations puisqu'il s'appuie sur MOF ou UML. Par contre l'apport est sensible pour la description de la composition. Il fournit d'une part une relation d'héritage entre les adapteurs mais c'est quelque chose que l'on retrouve par exemple dans AspectJ. Les adaptateurs permettent permettent de définir la majeure partie de la composition indépendamment du contexte d'intégration ce qui est particulièrement important pour la réutilisation.

6.2. *Possibilités d'entrelacements*

Une taxonomie très simple dans le domaine des aspects consiste à classer les approches selon le degré de symétrie de l'application d'un aspect sur sa base. On considère généralement deux types de symétrie, la symétrie des éléments et la symétrie des relations. La première s'intéresse à la nature des éléments. Existe-t'il des éléments de natures différentes, éléments d'aspect et éléments de base par exemple (approches asymétriques) ? Ou tous les éléments ont-ils la même nature (approches symétriques) ? Pour le critère d'entrelacement, nous considérerons la symétrie des relations. La symétrie des relations permet d'appliquer de la même manière un aspect sur un autre aspect ou un aspect sur une base. Une approche asymétrique, au contraire, se caractérise par la seule possibilité de mettre en relation un aspect avec une base sans pouvoir mettre en relation directement deux aspects.

6.2.1. *TranSAT*

TranSAT a pour but la construction incrémentale d'une description d'architecture logicielle. Par conséquent, TranSAT est une approche asymétrique, il permet l'intégration de nouvelles préoccupations fonctionnelles ou techniques au sein d'une base. TranSAT différencie ainsi distinctement la notion de plan de base qui contient uniquement des informations architecturales de la notion de patron d'architecture qui contient les fragments à intégrer mais aussi les contraintes sur le contexte d'intégration et les transformations à effectuer pour cette intégration. TranSAT ne permet pas de compo-

ser directement deux patrons d'architecture. L'application d'un patron d'architecture P_1 sur les éléments du plan d'un patron P_2 se fera alors toujours après intégration de P_2 sur un plan de base.

6.2.2. *Composant de modèle*

L'approche par composant de modèle se concentre sur la séparation des préoccupations fonctionnelles d'un système. Ces préoccupations sont spécifiées séparément en utilisant des composants de modèle qui expriment des fonctionnalités génériques, indépendantes de tout contexte applicatif. L'utilisation des composants de modèle consiste à les appliquer à un modèle de base du système ou à d'autres composants de modèle. L'approche ne fait pas de distinction entre ces deux types de modèles ce qui permet de les composer de façon homogène, à partir d'un unique opérateur. Pour le critère d'entrelacement, l'approche par composant de modèle est donc une approche symétrique. Actuellement, l'approche par composant de modèle ne traite que la partie structurelle. Des études sont en cours pour prendre en compte le comportement. Une voie privilégiée est l'utilisation d'assertions (précondition/postcondition, invariant) dans les composants de modèle ce qui pose des problèmes intéressants de compositions d'assertions.

6.2.3. *SmartAdapters*

SmartAdapters est conçu pour permettre à la fois la prise en compte de préoccupations fonctionnelles et extrafonctionnelles (ou même hybrides c'est-à-dire ayant simultanément ces deux propriétés), représentant à la fois des coupes verticales et horizontales d'une entité. Cela permet d'assurer une bonne séparation des préoccupations. La notion de modèle de base n'existe pas, toute préoccupation est représentée par un paquetage et peut être composée avec une autre indépendamment du rôle qu'elle joue. L'existence de deux modes de composition (*ex situ* et *in situ*) permet soit de créer un nouveau modèle qui est la composition des deux, soit de modifier un des deux modèles à composer. Par contre à l'heure actuelle, le comportement est principalement décrit par des invariants de classificateurs et des assertions de méthodes même si pour les besoins de certaines adaptations (adaptations d'interception principalement), SmartAdapters permet de spécifier des séquences d'appels de méthodes. Par ailleurs, une approche analogue (JAdapt (Quintian, 2004)) est dédiée non au modèle mais aux applications Java. Il serait intéressant d'envisager la modélisation du comportement dans SmartAdapters à travers le langage fourni avec Kermeta et de proposer une approche dérivée de JAdapt mais adaptée aux modèles.

6.3. *Traçabilité*

La traçabilité est la capacité à fournir les relations entre les concepts de deux modèles M_1 et M_2 où M_2 est obtenu par un procédé de raffinement de M_1. La traçabilité fournit donc une mesure de la qualité du processus de raffinement.

Dans un cycle de vie de développement logiciel, le système peut être raffiné de deux manières. Le raffinement dit *vertical* consiste à enrichir la spécification du système avec des informations toujours plus en lien avec la mise en œuvre. Le raffinement dit *horizontal* consiste à enrichir le système avec des informations liées à la définition de nouvelles préoccupations. Dans le cadre du raffinement vertical, le critère d'évaluation prend en compte la capacité à retrouver les préoccupations identifiées au niveau conceptuel au niveau de la mise en œuvre.

6.3.1. *TranSAT*

TranSAT se place clairement en phase de conception pour le raffinement horizontal d'une description d'architecture logicielle. Il permet de garder à tout moment deux visions du système, une vision dite éclatée dans laquelle l'architecte connaît la base et les différents patrons d'architecture et une vision dite résultante dans laquelle l'architecte travaille sur le résultat de la fusion des différents plans d'architecture. Au niveau de la mise en œuvre, TranSAT se fonde sur SafArchie qui permet de générer du code vers la plate-forme à composants Fractal[1] ou le langage ArchJava (Aldrich *et al.*, 2002). Pour cette transformation, SafArchie se place sur le modèle d'architecture résultant, au niveau de la mise en œuvre, il n'y a pas de conservation de la structure de l'architecture en préoccupation.

6.3.2. *Composant de modèle*

L'expression des composants de modèle ainsi que leur formalisation sont indépendantes d'une mise en œuvre particulière. Différents modes d'expression du modèle résultat ont cependant été proposés. Il est par exemple possible d'exprimer le système résultat d'un assemblage de composants de modèle à l'aide d'un modèle à vues. Cette possibilité permet ainsi de conserver, dans le modèle résultat, la structuration issue des différents composants de modèle. A partir de ces modèles résultats à base de vues, une architecture de patrons de conception a été également proposée. Cette architecture vise à conserver jusqu'à l'exploitation la structuration introduite lors de la phase de conception, afin notamment de préserver la trace des différentes fonctionnalités.

6.3.3. *SmartAdapters*

SmartAdapters permet de réaliser des compositions *in situ* (c'est-à-dire qui modifient un modèle M1 à partir des informations contenues dans le modèle M2) et *ex-situ* (création d'un nouveau modèle qui est la composition des modèles M1 et M2). Il est donc possible de construire un nouveau modèle qui représente un raffinement d'un modèle existant. Ce raffinement est matérialisé par le modèle de départ, le modèle qui contient la modification et l'adaptateur qui décrit la composition. Il est à noter que grâce à la relation d'héritage qui existe entre adapteurs, il est, d'une part, possible de raffiner la composition sans pour autant raffiner les modèles eux-mêmes et, d'autre part, de raffiner à la fois les modèles et la composition.

---

1. Plus précisément pour Julia, l'implémentation Java de référence.

6.4. *Capacité de composition*

La capacité de composition consiste principalement à évaluer les degrés de liberté offerts pour composer un système par tissage d'aspects. L'évaluation de la capacité de composition peut se décomposer en deux catégories. Premièrement, la capacité offerte par l'approche pour déclarer la composition entre une préoccupation transverse et le modèle de base. Deuxièmement, la capacité offerte à l'utilisateur pour paramétrer la sémantique de composition d'un aspect sur un système de base.

6.4.1. *TranSAT*

Le principal point fort de TranSAT réside dans une bonne expressivité de l'approche pour la gestion de la composition de plans d'architecture. En effet, TranSAT fournit tout d'abord un mécanisme de point de coupe avec deux niveaux d'abstraction : une description abstraite de la coupe à l'aide du masque de point de jonction qui est spécialisée à l'aide d'une expression de coupe dépendante du contexte d'intégration au moment du tissage. Ce langage propose alors le moyen de tisser le même aspect sur plusieurs lieux d'intégration d'un plan de base à l'aide d'un langage expressif. Contrairement à de nombreuses approches, TranSAT épargne à l'architecte la spécification complètement manuelle de la liaison entre un plan de base et la fonctionnalité à intégrer et permet de spécifier des relations *n-aires* entre une préoccupation à intégrer et le plan de base.

De plus, TransAT fournit un langage de transformation spécifique au domaine de l'architecture logicielle. Ce langage offre à l'architecte un haut pouvoir d'expression quand il souhaite spécifier l'intégration d'une préoccupation au sein d'un plan de base. Ce langage permet en outre de configurer finement la composition d'un plan en charge d'une fonctionnalité avec un plan de base.

Cependant, TranSAT ne propose pas de mécanisme avancé pour gérer la composition de multiples patrons sur un même plan de base. L'ordre est pour le moment spécifié de manière absolue et il n'existe pas de mécanisme pour détecter l'absence de conflits entre patrons.

6.4.2. *Composant de modèle*

Les capacités de composition de l'approche de composant de modèle sont fournies par l'opérateur d'application. Cet opérateur consiste à mettre en relation le modèle requis par un composant de modèle avec un modèle conforme auquel la fonctionnalité doit être ajoutée. Cet opérateur fonctionne aussi bien pour l'application à un modèle de base qu'à un autre composant de modèle permettant ainsi la construction de fonctionnalités complexes à partir de fonctionnalités plus simples. La conception d'un système peut être réalisée par l'assemblage d'un ensemble de composants de modèle. Les assemblages ainsi exprimés sont vérifiés en testant la conformité des modèles mis en correspondance. L'opérateur proposé possède également des propriétés d'ordre permettant de garantir la cohérence de la composition. Ces propriétés permettent notam-

ment de définir des chaînes d'application et la possibilité d'ajouter une fonctionnalité à un système sans remise en cause des applications précédentes.

6.4.3. *SmartAdapters*

Les possibilités de composition offertes par SmartAdapters s'appuient sur un ensemble d'opérateurs d'adaptation qui reposent sur la définition de cibles d'adaptation et sur des informations complémentaires associées à chaque adaptation comme par exemple des règles de correspondance. Celles-ci permettent à l'utilisateur de mieux spécifier l'effet par exemple d'une opération de fusion. Comme on l'a vu dans la section 5, les opérateurs proposés sont classés en quatre catégories : *i)* ajout d'éléments (par exemple, introduction d'une nouvelle méthode), *ii)* fusion de deux éléments (par exemple, fusion de classificateurs), *iii)* ajout d'un traitement sur interception (par exemple, ajout d'un fragment de code qui s'exécutera en cas d'accès à un attribut particulier) ou *iv)* redéfinition d'éléments (par exemple redéfinition de méthodes) etc. Par ailleurs, comme cela a été noté plus haut, ces opérateurs portent essentiellement sur les aspects structurels et sur les assertions (préconditions, postconditions et invariants).

6.5. *Evolutivité*

L'évolutivité définit la facilité potentielle d'évolution d'un élément. Le facteur-clé est alors la maîtrise de l'impact d'un changement. Dans le cadre du développement d'une application, le besoin d'évolution du logiciel peut être lié à deux facteurs importants : l'évolution des exigences et l'évolution d'un fragment de modèle pour des raisons de maintenance logicielle comme par exemple la réingénierie d'un fragment de modèle.

6.5.1. *TranSAT*

Le concept de masque de point de jonction permet de définir de manière explicite un contrat entre le plan de base à modifier et le patron d'architecture. Ce contrat, défini de manière abstraite sur les relations entre les entités du système, offre un bon support pour l'anticipation du changement. Construit à partir de la topologie structurelle et comportementale de l'architecture mais sans contrainte de nommage, le mécanisme de point de jonction est résistant au renommage des éléments de l'architecture. En outre, la sémantique de certains éléments du masque de point de jonction rend le mécanisme de tissage capable de décrire plusieurs types de transformation structurelle de l'application. Par exemple le masque de liaison définit une contrainte entre deux ports devant être liés quel que soit le nombre de membranes traversées et quel que soit le nombre de ports délégués traversés. En résumé, le mécanisme de masque de point de jonction permet de limiter le couplage entre un plan de base et un patron d'architecture, facilitant ainsi l'évolution de l'un indépendamment de l'autre. En outre la double vision éclatée et résultante du système permet d'évaluer rapidement l'impact d'une modification sur l'architecture résultante.

6.5.2. *Composant de modèle*

Les propriétés définies sur l'opérateur d'application de composants de modèle permettent de prendre en compte le besoin d'évolution des systèmes. Il est notamment possible d'appliquer différentes fonctionnalités au même système sans que celles-ci entrent en conflit et d'obtenir le même résultat quel que soit l'ordre dans lequel elles sont introduites. Il est également possible d'exprimer et d'identifier des chaînes d'application indépendantes. Cette propriété permet de garantir la validité des différentes applications qui composent la chaîne indépendamment les unes des autres. De telles chaînes peuvent ainsi être allongées (ajout de nouvelles fonctionnalités) ou coupées (suppression de fonctionnalités) sans remise en cause de l'intégrité de l'assemblage. Ces possibilités d'évolution, au niveau modèle, peuvent se retrouver au niveau de la mise en œuvre du système, à condition d'utiliser une stratégie préservant la structuration des différentes fonctionnalités.

6.5.3. *SmartAdapters*

L'évolution d'une préoccupation a un impact sur le protocole de composition et sur l'adaptateur spcécifique. Si c'est la préoccupation cible qui change, seul l'adaptateur spécifique devra être modifié. Ce résultat est obtenu principalement grâce aux cibles d'adaptation qui permettent de faire référence à des entités sans les avoir désignées explicitement. Par contre si c'est la préoccupation que l'on réutilise qui est modifiée c'est l'importance de la modification qui va décider de l'impact : si c'est le nom d'une entité qui change alors vraisemblablement seule la description de l'adaptation concernée sera modifiée. Par contre si la structure du diagramme de classes est modifiée, cela risque d'affecter plus largement la description de la composition. L'existence d'une relation d'héritage entre adapteurs devrait favoriser la gestion de l'évolution des préoccupations.

6.6. *Passage à l'échelle*

Le passage à l'échelle est une caractéristique importante de toute approche de modélisation. Cette caractéristique précise la capacité de l'approche à concevoir aussi bien des modèles de grande taille que de petits modèles.

6.6.1. *TranSAT*

La notion de patron d'architecture dans TranSAT n'impose pas *a priori* une granularité pour les fragments d'architecture à intégrer. Cependant, un patron d'architecture ne peut pas être composé à partir de différents patrons d'architecture. C'est une entité non hiérarchique. Ainsi, si les éléments du plan du patron d'architecture peuvent être de granularité très différente, le masque de point de jonction et les règles de transformation doivent rester intelligibles pour l'architecte afin qu'il puisse comprendre rapidement les contraintes liées à l'intégration et l'effet de l'intégration sur son architecture. De plus, un masque de point de jonction trop dédié et des règles de transformation trop intrusives ont tendance à limiter la généricité du patron d'architecture. Par

conséquent, la capacité de TranSAT à supporter le passage à l'échelle reste encore à démontrer.

6.6.2. *Composant de modèle*

L'approche par composant de modèle offre des capacités de passage à l'échelle. En effet, une première caractéristique répondant à ce besoin est la possibilité de former des composants complexes à partir de composants plus simples, ces nouveaux composants pouvant être réutilisés à leur tour pour d'autres composants de plus en plus complexes et répondant à des besoins fonctionnels de plus en plus riches. Cette caractéristique permet aussi d'envisager la conception de bibliothèques de composants de modèle spécifiant des parties fonctionnelles de grands systèmes. Une autre caractéristique permettant de prendre en compte un passage à l'échelle est l'assemblage de chaînes d'application que supporte l'approche. Avec cette possibilité, des chaînes d'application correspondant à des parties plus ou moins complexes d'un système peuvent être conçues séparément (éventuellement par différentes équipes) puis être assemblées pour obtenir le système dans son intégralité.

6.6.3. *SmartAdapters*

SmartAdapters est dédié plutôt à de « petits » modèles correspondant à des préoccupations qui pourront être composées. Ce choix a été fait principalement pour que le métamodèle permettant la représentation des préoccupations reste simple. Par contre grâce à l'expressivité du mécanisme de composition il est possible d'obtenir par ce biais de gros modèles. Par rapport à cette hypothèse de départ le métamodèle et le langage de composition apparaissent donc suffisants ; le séquencement des différentes adaptations est fait dans l'ordre d'apparition. Par contre il n'y a pas actuellement de moyens de décrire un ordre d'exécution précis. Cependant pour une même préoccupation cible, les compositions *ex situ* sont réalisées avant celles qui sont *in situ*. Ce manque d'expressivité pour orchestrer l'exécution des compositions ne permet pas de prendre en compte facilement des compositions qui pourraient interférer les unes avec les autres. Il pourra être intéressant d'étudier pour cela les travaux de Gunter Kniesel relatifs à l'analyse des dépendances entre les préoccupations[2].

7. Bilan et discussion

Cette section discute des similitudes et des différences entre les approches et propose de conserver un ensemble de caractéristiques que devraient respecter les approches de conception par aspects, sujets ou vues.

---

2. http ://roots.iai.uni-bonn.de/research/condor/

7.1. *Similitudes*

La première similitude tient aux objectifs visés par ces trois approches. En effet, ces approches tendent toutes à améliorer la structuration d'un système pour la prise en compte des préoccupations transverses. Ces trois approches peuvent être utilisées dans le cadre d'un processus de construction incrémentale du système. Finalement, ces trois approches mettent en avant la nécessité de travailler sur la réutilisabilité des fragments à intégrer. Une des principales difficultés liées à la réutilisation consiste à pouvoir garantir l'utilisation correcte du fragment logiciel dans un contexte différent de celui pour lequel il a été conçu.

Une deuxième similitude concerne l'introduction d'une entité pour structurer la notion de préoccupation et l'expliciter.

Une troisième similitude de TranSAT, des composants de modèle et de SmartAdapters est la définition d'une abstraction du contexte d'intégration, le masque de point de jonction pour TranSAT, le modèle requis pour les composants de modèle ou les cibles d'adaptation abstraites pour SmartAdapters. Ces concepts sont très proches et permettent de réutiliser une préoccupation pour des contextes d'intégration différents.

Les adapteurs, patrons ou composants de modèle sont dédiés à l'adaptation d'un type de modèle (modèle de classe pour SmartAdapters et les composants de modèle, assemblages de composants pour TranSAT). Il n'est pas possible d'abstraire davantage les adaptateurs.

7.2. *Différences*

Cependant, les choix faits au sein de ces trois approches les font diverger par bien des points. TranSAT et SmartAdapters définissent un langage dédié pour déclarer les modifications à effectuer sur les modèles en vue de la fusion. Au contraire, l'approche par composant de modèle explicite les résultats des modifications au niveau du modèle du contexte.

TranSAT et SmartAdapters d'une part, les composants de modèle, d'autre part, présentent des différences importantes de « philosophie » de la composition car les premières sont des approches asymétriques alors que la seconde est une approche symétrique. La symétrie de l'approche permet d'agréger différents composants de modèle et facilite ainsi le passage à l'échelle.

SmartAdapters et TranSAT ne proposent que de la traçabilité verticale, l'approche composant de modèle fournit quant à elle différents *frameworks* (EJB, Fractal, Corba) de mise en œuvre qui permettent de conserver une vision éclatée du système à l'exécution.

Finalement, SmartAdapters et l'approche composant de modèle propose des opérateurs de composition travaillant exclusivement sur la structure. Au contraire, les patrons d'architecture de TransAT permettent la mise à jour de la structure des com-

posants, de la configuration de l'architecture mais aussi de la spécification comportementale des composants.

7.3. *Caractéristiques idéales d'un modèle de composition*

Nous proposons d'extraire de chaque approche les caractéristiques les plus intéressantes afin d'esquisser les bases d'un métamodèle de composition. Ainsi, un métamodèle de composition doit fournir :

1) une entité primaire pour structurer la notion de préoccupation. Cette entité appelée patron, composant de modèle ou *package* permet de proposer une entité réutilisable ;

2) une abstraction du contexte d'intégration. Cette entité appelée masque de point de jonction, modèle requis ou cible d'adaptation abstraite permet d'expliciter l'interface requise de l'entité en charge d'une préoccupation ;

3) une approche symétrique. Cette caractéristique, proposée par l'approche composant de modèle, offre la possibilité de composer les différentes préoccupations pour faire émerger des préoccupations composites qui résultent de l'assemblage de plusieurs préoccupations. La symétrie facilite alors le passage à l'échelle de l'approche ;

4) un langage dédié pour décrire la composition de manière explicite. Ce langage, proposé entre autre dans TranSAT et SmartAdapters, permet de laisser plus de liberté sur la sémantique de la composition. Cette sémantique est alors spécifiée à l'aide du langage de transformation. D'un autre côté, la représentation du résultat comme dans l'approche composant de modèle fournit une excellente vision de la fonctionnalité introduite par le composant de modèle. Une approche idéale devrait alors proposer un mécanisme hybride ;

5) un langage de coupe pour permettre la création d'une relation *n-aire* entre une préoccupation et un modèle de base. Cette caractéristique proposée dans SmartAdapters et TranSAT fournit un pouvoir d'expression puissant pour lier une préoccupation et un modèle de base. Ce langage est en outre un très bon complément de la spécification du contexte d'intégration ;

6) la possibilité de raffiner une préoccupation. Cette caractéristique, proposée dans SmartAdapters à l'aide d'un lien d'héritage, permet de paramétrer de manière élégante une préoccupation à son contexte d'utilisation. Cette caractéristique va dans le sens de la réutilisation des préoccupations ;

7) la prise en compte de la dimension comportementale dans la sémantique de la composition. Comme proposé dans TranSAT, il est nécessaire de prendre en compte à la fois les informations de structure et les informations de comportement. En effet, les approches purement structurelles semblent trop pauvres pour exprimer réellement la sémantique de la composition. La définition de la sémantique est alors relayée au niveau de la mise en œuvre ;

8) l'inférence de l'indépendance de l'intégration de différentes préoccupations. Comme proposé dans l'approche composant de modèle, la capacité d'inférer l'indé-

pendance de l'application de deux préoccupations autorise l'application simultanée de plusieurs préoccupations sur un même système ;

9) la garantie de la cohérence du système résultant. Finalement, l'approche doit garantir la cohérence du système résultant après l'application d'une préoccupation. Cette garantie, si elle est vérifiée de manière statique à l'aide des informations de la préoccupation, renforce le caractère réutilisable de la préoccupation.

8. Autres approches

Theme (Clarke *et al.*, 2005; Baniassad *et al.*, 2004) est une approche pour l'analyse et la conception orientée aspects. Les aspects sont des préoccupations transverses et réparties dans un système. L'approche Theme se situe donc à deux niveaux : au niveau analyse *Theme/Doc* et au niveau design (modèle) *Theme/UML*. Theme/Doc aborde le problème de l'identification des aspects dans la phase d'analyse des besoins. Il propose pour cela de représenter les relations entre les fonctionnalités du système à l'aide d'un graphe d'analyse.

Au niveau conception, *Theme/UML* permet de modéliser les fonctionnalités et les aspects d'un système, et de spécifier comment ceux-ci doivent être composés.

Un *theme* correspond à un paquetage *template*, c'est-à-dire à un paquetage paramétré. Une relation (nommée *bind*) permet d'exprimer la composition paramétrée de deux *themes*. La vérification de l'assemblage est réalisée en ajoutant au graphe d'analyse (Theme/Doc) les informations relatives aux différentes compositions.

L'approche Theme est conçue pour permettre la définition de traitements génériques, comme des fonctionnalités de trace ou de synchronisation. Chaque paramètre correspond à une classe et à l'ensemble de ses méthodes sur lesquelles la fonctionnalité doit être « tissée ». Ce tissage est décrit à l'aide de diagrammes de séquences. Comparés aux approches précédentes, Theme/UML et l'approche par composant de modèle sont proches. Cependant, Theme/UML n'étudie pas les problèmes d'ordre de composition et la possibilité de composer les fonctionnalités génériques entre elles, qui sont des supports importants pour l'évolutivité et le passage à l'échelle.

Les travaux de (Straw *et al.*, 2004) permettent la représentation de modèles d'aspects génériques dont les paramètres sont identifiés à l'aide de l'élément syntaxique "|" (pipe). Ces modèles d'aspects doivent ensuite être instanciés avant d'être composés à un modèle primaire. Ce modèle primaire correspond au modèle de base du système, tel que présenté précédemment, auquel les différents modèles d'aspects devront être ajoutés. L'instanciation des modèles d'aspects permet d'adapter l'aspect au contexte particulier défini par le modèle primaire. Cela est réalisé en paramétrant les modèles d'aspects par des éléments de l'espace de nommage du modèle primaire.

Les modèles d'aspects contextualisés et le modèle primaire peuvent ensuite être composés. Les éléments de même nom du modèle d'aspect et du modèle primaire sont par défaut fusionnés. Cette stratégie peut être modifiée par des directives de manipu-

lation de modèles : création d'éléments, renommage, ajout et suppression d'éléments dans un espace de nommage, remplacement d'un élément par un autre. Ces directives permettent de résoudre les conflits et incohérences pouvant apparaître dans le modèle composé. Elles permettent également d'imposer un ordre de composition des différents aspects. L'approche ne donne pas de contrainte sur le paramétrage ou sur la composition, l'identification et la résolution de ces incohérences restent à la charge du concepteur.

Cette approche offre ainsi une bonne capacité d'abstraction vis-à-vis de l'implantation et la possibilité d'entrelacer les modèles d'aspect selon un mode asymétrique. Cependant, la trace des modèles d'aspect n'est pas conservée et les modèles d'aspect ne peuvent être composés entre eux indépendamment d'un modèle de base ce qui rend plus difficile le passage à l'échelle.

9. Conclusion et perspectives

Les trois types d'approche de conception d'un système à l'aide de vues, de sujets ou d'aspects permettent une meilleure structuration des systèmes dès les phases amont du développement logiciel. Ce document compare selon les critères de (Chitchyan *et al.*, 2005) trois approches particulières : TranSAT, l'approche composant de modèle et SmartAdapters. La comparaison de ces types d'approche permet de mettre en lumière leurs similitudes et leurs particularités et permet aussi l'identification des atouts et des faiblesses de chaque approche. Finalement à l'aide de cette identification des atouts de chaque approche, nous proposons un ensemble de caractéristiques à respecter pour la construction de canevas de composition de modèles. Ces caractéristiques peuvent alors être vues comme un travail d'analyse en amont de la définition d'un métamodèle de canevas de composition de modèles.

Il existe actuellement deux perspectives à ce travail. La première recherche l'abstraction du mécanisme de composition du métamodèle cible. En effet, l'objectif d'UML (*Unified Modeling Language*) ou de certains langages de description d'architecture (ADL) était de devenir le langage de spécification pivot de tout projet informatique. Cet objectif n'a été que partiellement atteint. Le manque de sémantique associée aux concepts présents au sein d'UML et l'absence de bases formelles obligent chaque domaine d'application à redéfinir sa propre sémantique à travers un profil ou un langage spécifique (DSL : *Domain Specific Language*). L'effort dans la construction de ces langages est alors principalement mis sur la précision sémantique des concepts et l'obtention d'une bonne adéquation entre les moyens et les besoins. Dès lors, les problématiques d'extension ou de raffinement associées à ces langages sont souvent mises en retrait. Nous proposons comme première perspective à ce travail de synthèse, un modèle d'aspect générique adaptable à différents DSL. L'objectif est de pouvoir construire rapidement, à partir des concepts présents dans le DSL cible et de notre modèle d'aspect générique, un cadre de raffinement fiable par aspects pour les spécifications exprimées à l'aide de ce DSL. Les critères définis dans cet article doivent permettre d'esquisser les caractéristiques de ce modèle d'aspect abstrait.

La deuxième perspective de ces travaux porte sur la définition du modèle de contexte d'intégration d'une préoccupation. En effet, ce modèle est exprimé pour le moment à l'aide des concepts du métamodèle du modèle à transformer (diagramme de classes pour SmartAdapters et l'approche composant de modèle, diagramme d'architecture pour TranSAT). Il serait pertinent, pour ce modèle de contexte d'intégration, d'utiliser le concept de *type de modèle* introduit dans (Steel *et al.*, 2005) pour permettre la réutilisation de préoccupations pour différents modèles possédant des métamodèles différents. Ainsi une fonctionnalité de *location* devrait, à l'aide de ce modèle de type, pouvoir s'intégrer de manière transparente sur un modèle UML, sur un programme Java ou sur un modèle Kermeta (Muller *et al.*, 2005b).

10. Bibliographie

, *UML 2.0 Infrastructure Specification*, http ://www.omg.org/cgi-bin/apps/doc ?formal/05-07-05.pdf, 2005.

Abmann U., *Invasive Software Composition*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

Aldrich J., Chambers C., Notkin D., « ArchJava : connecting software architecture to implementation », *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, ACM Press, New York, p. 187-197, May 19–25, 2002.

Baniassad E., Clarke S., « Theme : An Approach for Aspect-Oriented Analysis and Design », *ICSE '04 : Proceedings of the 26th International Conference on Software Engineering*, IEEE Computer Society, p. 158-167, 2004.

Barais O., Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants, Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille, Lille, France, novembre, 2005.

Barais O., Duchien L., *SafArchie Studio : An ArgoUML extension to build Safe Architectures*, Architecture Description Languages, Springer, p. 85-100, 2005a.

Barais O., Duchien L., Meur A.-F. L., « A Framework to Specify Incremental Software Architecture Transformations », *31st EUROMICRO CONFERENCE on Software Engineering and Advanced Applications (SEAA 2005)*, IEEE Computer Society, September, 2005b.

Chitchyan R., Rashid A., Sawyer P., Garcia A., Mónica, Alarcon P., Bakker J., Tekinerdogan B., Clarke S., Jackson A., Survey of Analysis and Design Approaches, Technical report, AOSD-EUROPE, 2005.

Clark T., Evans A., Sammut P., Willans J., *Applied Metamodelling, A Foundation for Language Driven Development*, Xactium, 2004, 2004.

Clarke S., Composition of Object-Oriented Software Design Models, PhD thesis, School of Computer Applications - Dublin City University, january, 2001.

Clarke S., Walker R. J., « Generic Aspect-Oriented Design with Theme/UML », Addison-Wesley, Boston, p. 425-458, 2005.

Crescenzo P., Lahire P., « Une approche pour améliorer la réutilisabilité des modèles métiers », *Actes de la 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, Lille, p. 51-73, septembre, 2005.

David P.-C., Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation, Phd thesis, Université de Nantes / École des Mines de Nantes, July, 2005.

Kiczales G., al., « Aspect-Oriented Programming », *European Conference on Object-Oriented Programming (ECOOP)*, vol. LNCS 1241, Springer-Verlag, June, 1997.

Lahire P., Quintian L., « New Perspective To Improve Reusability in Object-Oriented Languages », *Journal Of Object Technology (JOT)*, vol. 5, n° 1, p. 117-138, 2006.

Lynch N. A., Tuttle M. R., « An Introduction to Input/Output Automata », *CWI Quarterly*, vol. 2, n° 3, p. 219-246, 1989.

Muller A., Construction de systèmes par application de modèles paramétrés, Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Université de Lille, Lille, France, Juin, 2006.

Muller A., Caron O., Carré B., Vanwormhoudt G., « Réutilisation d'aspects fonctionnels : des vues aux composants », *Langages et Modèles à Objets (LMO'03)*, Hermès Sciences, p. 241-255, January, 2003.

Muller A., Caron O., Carré B., Vanwormhoudt G., « On Some Properties of Parameterized Model Application », *First European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05)*, vol. 3748 of *LNCS*, Springer, p. 130-144, November, 2005a.

Muller P.-A., Fleurey F., Jézéquel J.-M., « Weaving Executability into Object-Oriented Metalanguages. », *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, vol. 3713 of *Lecture Notes in Computer Science*, Springer, p. 264-278, 2005b.

Ossher H., Tarr P., Murphy G. (eds), *Proceedings of the Workshop on Multidimensional Separation of Concerns at OOPSLA'99*, Denver, Colorado, USA, November, 1999.

Parnas D. L., « On the criteria to be used in decomposing systems into modules », *Commun. ACM*, vol. 15, n° 12, p. 1053-1058, 1972.

Quintian L., JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet, Thèse de doctorat, Université de Nice-Sophia Antipolis, France, juillet, 2004.

Steel J., Jézéquel J.-M., « Model Typing for Improving Reuse in Model-Driven Engineering. », *in* L. C. Briand, C. Williams (eds), *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings*, vol. 3713 of *Lecture Notes in Computer Science*, Springer, p. 84-96, 2005.

Straw G., Georg G., Song E., Ghosh S., France R., Bieman J. M., « Model Composition Directives », *Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2004)*, vol. 3273 of *LNCS*, Springer, p. 84-97, 2004.