Notion de Patron Multivues : application du profil VUML

B. Coulette[*] — Y. Lakhrissi[*,] — M. Nassar[***] — A. Kriouile[***]**

** GRIMM-ISYCOM Université de Toulouse II*
5, allées A. Machado 31058 Toulouse Cédex, France
coulette@univ-tlse2.fr, younes.lakhrissi@univ-tlse2.fr

*** UFR ACSYS, Faculté des Sciences de Rabat*

**** LGI, ENSIAS BP 713 Agdal, Rabat, Maroc*
nassar@ensias.ma, kriouile@ensias.ma

RÉSUMÉ. *Ce travail fait suite à l'élaboration du profil UML appelé VUML (View based UML) qui supporte la conception multivues. L'objectif de ce papier est d'introduire la notion de « patron multivues ». De tels patrons seront utilisés par les concepteurs de systèmes complexes dans un double but de factorisation du savoir-faire et de réutilisation. Les points de vue à considérer sont dans un premier temps des points de vue prédéfinis représentant les développeurs (analyste, concepteur, programmeur, testeur, mainteneur...), puis des points de vue d'utilisateurs finals propres à chaque domaine d'application. Nous illustrons notre proposition par l'exemple du patron « Commande défaisable » qui permet de décrire de façon générique une commande exécutable éventuellement annulable puis refaisable.*

ABSTRACT. *This work is based on the UML profile named VUML (View based UML) which supports multiview designing. The goal of this paper is to introduce the notion of "multiview Pattern". Such patterns will be used by complex systems designers to both factorize know-how and reuse model fragments. Points of view to consider are firstly predefined ones representing developers (analysts, designers, programmers, testers, maintainers…), secondly final users ones which are specific of each application domain. We illustrate our proposal through the pattern "Undoable Command" which allows to describe in a generic way an executable command that may be undone and redone.*

MOTS-CLÉS : *UML, profil VUML, modélisation multivues, patron logiciel, patron multivues*

KEYWORDS: *UML, profile VUML, multiview modelling, software pattern, multiview pattern*

1. Introduction

Lors de la conception d'un système complexe, l'établissement d'un modèle global prenant en compte simultanément tous les besoins des acteurs est un leurre. Dans la réalité, soit plusieurs modèles partiels sont développés et coexistent avec les risques d'incohérence associés, soit le modèle global doit être fréquemment remis en cause quand les besoins des utilisateurs évoluent. Ce constat s'applique tout particulièrement aux systèmes d'information qui se caractérisent souvent par une forte interaction avec les utilisateurs. Pour répondre à cette difficulté, nous travaillons depuis plusieurs années sur l'intégration de la notion de point de vue dans l'analyse/conception de systèmes logiciels. C'est ainsi que nous avons développé récemment un profil UML, appelé VUML (View based Unified Modeling Language), qui permet d'analyser/concevoir un système logiciel par une approche combinant objets et points de vue (Nassar et al. 2005). Ce profil a été implanté sous l'AGL Objecteering. Nous avons également élaboré un noyau de démarche pour produire un modèle multivues. La singularité de VUML par rapport aux autres approches orientées vues (Debrauwer 1998, Kriouile 1995, Mili et al. 2001, Muller et al. 2003, Barais et al. 2005) réside dans le fait d'associer de façon unique une vue à chaque point de vue sur le système à concevoir. Ce choix simplifie tout d'abord la détermination des vues sur une entité donnée en supprimant le fort indéterminisme qui la caractérise, et réduit aussi les problèmes liés à la composition des vues inhérents aux propositions à base de vues multiples.

Par ailleurs, la notion de patron, introduite par l'architecte C. Alexander (Alexander 1979), devient un « outil » irremplaçable pour le concepteur logiciel. D'une manière générale, un patron décrit un problème récurrent et une solution générique pour résoudre ce problème. Les patrons logiciels sont utilisés partout où il y a besoin de capitaliser de la connaissance et du savoir-faire (analyse, conception, implémentation, etc.). Les patrons associés au développement de logiciel sont par nature multi-points de vues puisqu'ils répondent aux besoins de plusieurs types d'acteur : les acteurs de développement (analyste/concepteur, programmeur, testeur, …) lors de la conception et de l'implantation du patron, et les utilisateurs finals au moment de l'instanciation du patron dans un domaine d'application donné. C'est ainsi que nous travaillons actuellement à définir et formaliser la notion de *patron multivues* afin d'une part de capitaliser la connaissance des concepteurs utilisant l'approche VUML, d'autre part aider ces derniers à réutiliser des patrons multivues archivés et documentés.

La suite de cet article est structurée comme suit. La section 2 présente un rappel de l'approche VUML. Dans la section 3, nous donnons un exemple de patron de conception classique, le patron *Commande Défaisable*, et dans la section 4, nous examinons comment ce patron peut être transformé en un patron multivues. En conclusion, nous synthétisons les premiers résultats de notre étude et décrivons ses principales perspectives.

2. Principes de VUML (Nassar et al, 2005)

VUML est fondée sur une approche de modélisation centrée points de vue, dont l'objectif est de représenter les besoins et les droits d'accès des utilisateurs de l'analyse jusqu'à l'implémentation. Plus précisément, un **point de vue** est la « vision » d'un acteur (au sens UML du terme) sur le système (ou sur une partie de ce système). Une **vue** est un élément de modélisation (statique) correspondant à l'application d'un point de vue sur une entité donnée. Par simplification de langage, nous disons qu'une vue est associée à un acteur, en considérant comme implicite l'entité sur laquelle le point de vue de l'acteur s'applique. VUML s'appuie sur les trois principes suivants : **(i)** association déterministe d'un point de vue à chaque type d'acteur (utilisateur final ou non) ; **(ii)** association d'une vue unique à chaque point de vue sur un objet donné ; **(iii)** implantation et gestion de l'évolution et de l'activation des vues par l'intermédiaire de la délégation.

Le concept de **classe multivues** est l'élément clé du formalisme VUML. Par rapport à une classe UML "habituelle", une classe multivues est dotée de plusieurs vues représentant les besoins et les droits spécifiques des acteurs. La figure 1 ci-dessous illustre la structure statique d'une classe multivues. Une telle classe est composée d'une base (stéréotype «base») qui a le même nom que la classe UML correspondante, et des vues (stéréotype «view») qui sont reliées à la base via une relation viewExtension. L'activation d'une vue (liaison avec le point de vue de l'utilisateur courant) est faite lors de l'exécution. La relation de dépendance viewExtension exprime le fait que les attributs et les méthodes de la base sont implicitement partagés par les vues de la classe multivues; en outre, une caractéristique de vue peut redéfinir une caractéristique de la base.

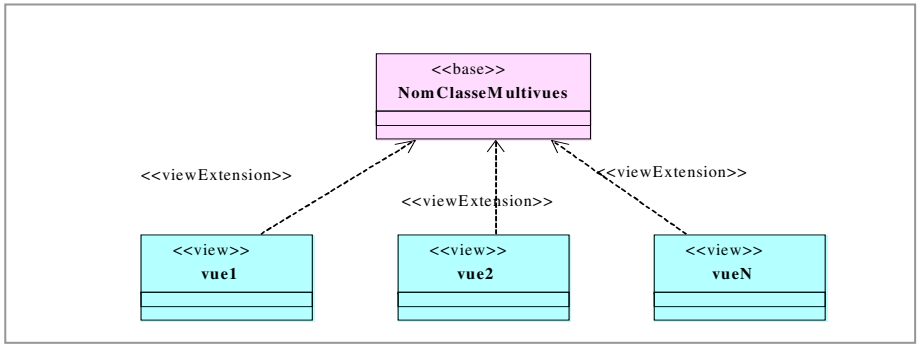

Figure 1. *Structure d'une classe multivues*

Les spécificités de VUML sont décrites dans un méta-modèle qui est une spécialisation du méta-modèle UML, et dont la sémantique est décrite par des expressions OCL (Nassar, 2005). La figure 2 décrit un sous-ensemble du méta-

modèle VUML. Une classe multivues comprend une base et des vues reliées à la base par la relation ViewExtension. La base hérite de *GeneralView* car elle est considérée comme une vue par défaut partagée par tous les points de vue (acteurs) du système. Les vues d'une classe multivues peuvent avoir des dépendances fonctionnelles explicitées par des relations *ViewDependency* décrites en OCL.

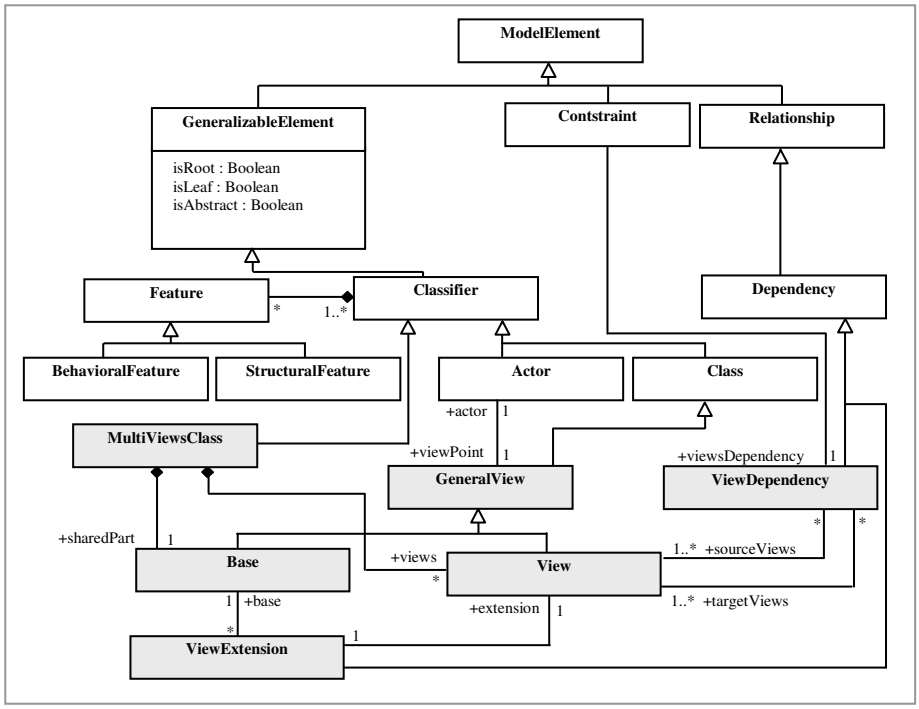

Figure 2. *Fragment du métamodèle VUML (Nassar et al, 2005)*

La démarche de modélisation selon VUML est composée de 3 phases : (i) une phase centralisée de modélisation des besoins pour produire un diagramme des cas d'utilisation, (ii) une phase décentralisée de conception UML selon chaque point de vue, (iii) une phase centralisée de fusion des modèles UML développés séparément, qui aboutit à la réalisation d'un modèle multivues.

3. Patron de conception : exemple de *Commande Défaisable*

Pour illustrer notre propos, nous choisissons un patron de conception classique appelé « Commande Défaisable », et inspiré des descriptions données dans (Gamma et al. 1995) et (Meyer, 1997). Ce patron décrit une commande que l'on souhaite

défaire (annuler) ou refaire selon le contexte. De telles commandes, exploitées par exemple dans un menu, seront défaisables à un niveau quelconque, c'est-à-dire que l'utilisateur pourra défaire la dernière commande exécutée jusqu'à ce qu'il n'y ait éventuellement plus rien à défaire. De même, il pourra refaire (rejouer) les dernières commandes défaites.

A titre d'application de ce patron, nous décrivons un système permettant de créer le catalogue d'une agence de voyage et de gérer des réservations de voyages par des clients. Le catalogue peut être considéré comme une liste de voyages. Le gestionnaire de l'agence doit pouvoir naviguer dans le catalogue, ajouter de nouveaux voyages, modifier ou supprimer un voyage, etc. Un client peut consulter le catalogue, réserver un voyage, afficher les voyages qu'il a réservés, etc. La figure 3 ci-dessous illustre la conception du système *Gestion d'un catalogue de voyages*. La partie générique de cette architecture statique permet d'implanter un menu de commandes, et un historique des commandes défaisables. Les commandes concrètes sont décrites comme des sous-classes de *Commande* ou *Commande Défaisable* en fonction du domaine applicatif. Chaque commande pointe sur un contexte qui est affecté dynamiquement lors de l'instanciation des sous-classes représentant les commandes concrètes. On peut noter que les commandes *Annuler* et *Refaire* ne sont pas considérées comme défaisables.

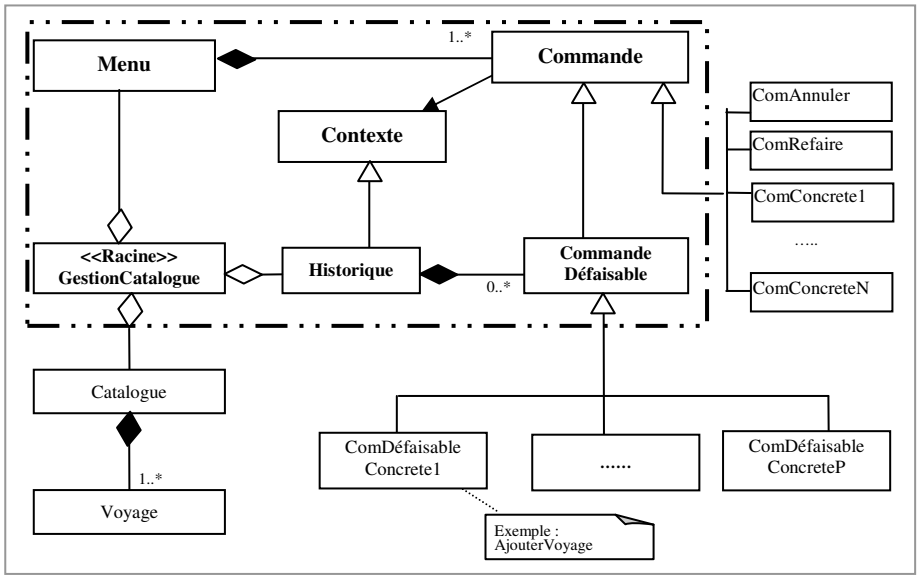

Figure 3. *Conception simplifiée du système « Gestion d'un catalogue de voyages »*

La classe *GestionCatalogue* représente la classe racine du système. Elle dispose d'un menu qui implante par exemple une liste de commandes et d'un historique qui est par exemple une liste de commandes défaisables. Chaque commande référence un contexte qui lui permet d'accéder aux informations nécessaires pour son exécution. L'historique est géré au moyen d'un index. La commande *ComAnnuler* permet de défaire la dernière commande défaisable stockée dans l'historique. Elle déplace l'index de l'historique vers la gauche. La commande *ComRefaire* refait, le cas échéant, la première commande annulée située à droite de l'index et fait avancer l'index. Toute commande autre que *ComRefaire* ou *ComAnnuler* consécutive à des annulations commence par vider l'historique des commandes défaisables situées au-delà de l'index.

La figure 4 illustre la spécification des classes abstraites *Commande* et *CommandeDéfaisable* ainsi que des exemples de spécialisation de ces classes. La classe abstraite *Commande* a un attribut de type *Contexte* dénotant l'objet sur lequel s'exécute la commande et pour méthodes abstraites *executer*() et *activable*(). L'activabilité d'une commande dépend de l'état de l'application à l'exécution. La classe abstraite *CommandeDéfaisable* hérite de *Commande* et contient les méthodes *defaire*() et *refaire*() qui seront concrétisées dans les sous-classes correspondant aux commandes concrètes.

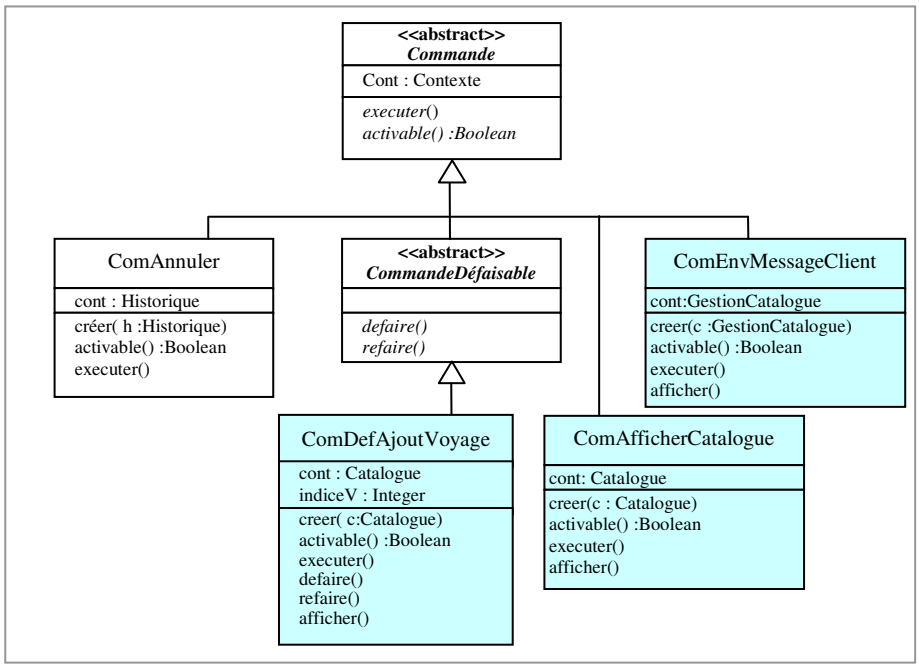

Figure 4. *Exemple de commandes concrètes de « GestionCatalogue »*

ComDefAjoutVoyage est une commande défaisable qui permet d'ajouter un voyage dans le catalogue, le fait de défaire cette commande enlevant ce voyage du catalogue. L'attribut *indiceV* est sauvegardé lors de l'exécution de la commande afin de gérer l'éventuelle annulation. En revanche, *ComEnvMessageClient* ne dérive pas de *CommandeDéfaisable* mais de *Commande*, car si on envoie un message à un client, il n'est pas possible d'annuler cette opération. La commande *ComAnnuler* quant-à elle, n'est pas non plus défaisable et a pour contexte l'historique des commandes défaisables puisqu'elle consiste à exécuter la méthode *défaire*() de la dernière commande stockée dans l'historique.

4. Notion de patron Multivues

4.1. Introduction

Au cours du cycle de vie d'un patron, plusieurs acteurs sont amenés à interagir avec lui, que ce soit pendant son développement, ou lors de son application dans un domaine concret. Les acteurs de développement sont relativement stables d'un patron à l'autre, tandis que les utilisateurs finals dépendent bien sûr du domaine d'application. Dans le cas du patron *Commande Défaisable*, chaque acteur de développement a sa propre « vision » d'une commande. Par exemple le concepteur aura besoin de la description de la commande au niveau conceptuel (nom, contexte, opérations, objets à sauvegarder en cas d'annulation,..); le programmeur, par contre, s'intéressera plus spécifiquement au langage de programmation utilisé et au code de chaque opération de la commande ; le testeur, quant à lui, aura besoin de connaître les entrées et sorties de chaque opération de la commande, leurs pré et post-conditions, et devra gérer le programme de test de chaque opération.

Quand on souhaite appliquer (instancier) le patron dans un domaine particulier — par exemple celui de la gestion d'un catalogue de voyages (cf. section 2) —, il faut prendre en compte les besoins des utilisateurs finals. Si l'on considère le Gestionnaire et le Client, il est clair qu'ils n'ont pas les mêmes droits d'accès sur les commandes concrètes. Par exemple, le gestionnaire pourra effectuer une commande d'ajout de voyage dans le catalogue, ce qui sera interdit naturellement au client. Les deux par contre pourront afficher le contenu du catalogue.

On peut donc constater que la démarche multivues peut être appliquée à 2 niveaux : tout d'abord au cours du développement du patron où l'on peut en compte de façon générique les acteurs de ce développement, puis lors de son instanciation dans un domaine applicatif où on peut concrétiser les informations associées aux acteurs du développement, puis introduire les utilisateurs finals. La figure 5 résume la hiérarchisation des points de vue pour notre étude de cas, en montrant qu'il est possible à tout moment d'ajouter un point de vue, ce qui est particulièrement utile quand il s'agit d'un utilisateur final.

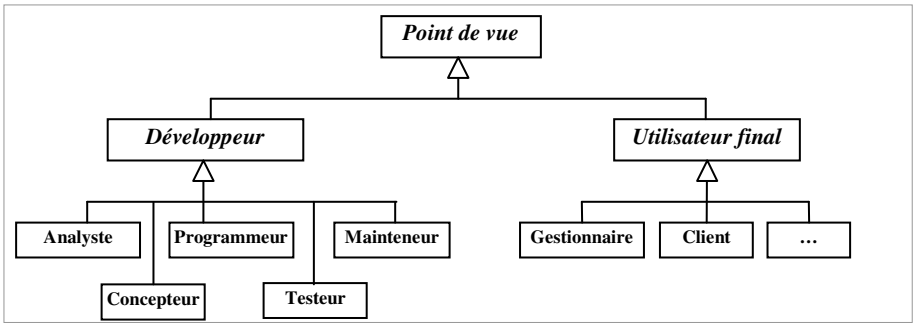

Figure 5. *Hiérarchie des points de vues pour le système GestionCatalogue*

4.2. Description VUML du patron Commande Défaisable

4.2.1. Au niveau du développement du patron

Reprenons (cf. figure 3) la partie générique du diagramme de classes qui décrit la conception UML de notre patron. Plusieurs classes sont manifestement multivues. Pour simplifier, nous nous intéressons ci-dessous aux classes *Commande* et *CommandeDéfaisable*, en prenant en compte les 3 points de vue de développement *Concepteur*, *Programmeur* et *Testeur*.

En appliquant la démarche VUML rappelée dans la section 2, nous obtenons le diagramme de la figure 6 ci-dessous, dans lequel nous avons utilisé la notion de vue abstraite (stéréotype « abstractView ») pour regrouper les vues associées aux points de vue de développement. Si l'on considère les classes multivues *Commande et CommandeDéfaisable*, elles sont composées d'une base et de 3 vues correspondant aux 3 points de vue de développement considérés. Ces classes sont abstraites car les méthodes (de la base ou des vues) sont abstraites. L'apport de ce diagramme VUML réside dans la répartition des informations entre les vues qui explicitent la vision d'une commande par chaque acteur de développement considéré. Par exemple, le concepteur de *Commande* dispose dans la vue *ConcepteurCom* d'une description textuelle *desc* et d'une méthode *afficher*(). Le programmeur quant à lui, a accès dans la vue *ProgrammeurCom* au code de la classe dans un certain langage de programmation, et à une méthode d'affichage spécifique. Le testeur a accès dans la vue *TesteurCom* au code des programmes de test des méthodes *executer*() et *activable*() issus de la base, aux opérations de test correspondantes *testerExe*() et *testerAct*(), et à une méthode d'affichage spécifique. Il en est de même avec *CommandeDéfaisable* dont les vues spécialisent les vues de *Commande*, avec d'éventuelles redéfinitions des caractéristiques. Dans la vue *TesteurCD*, il y a de

nouveaux attributs et méthodes pour prendre en compte le test des opérations *défaire*() et *refaire*() provenant de la base *CommandeDéfaisable*.

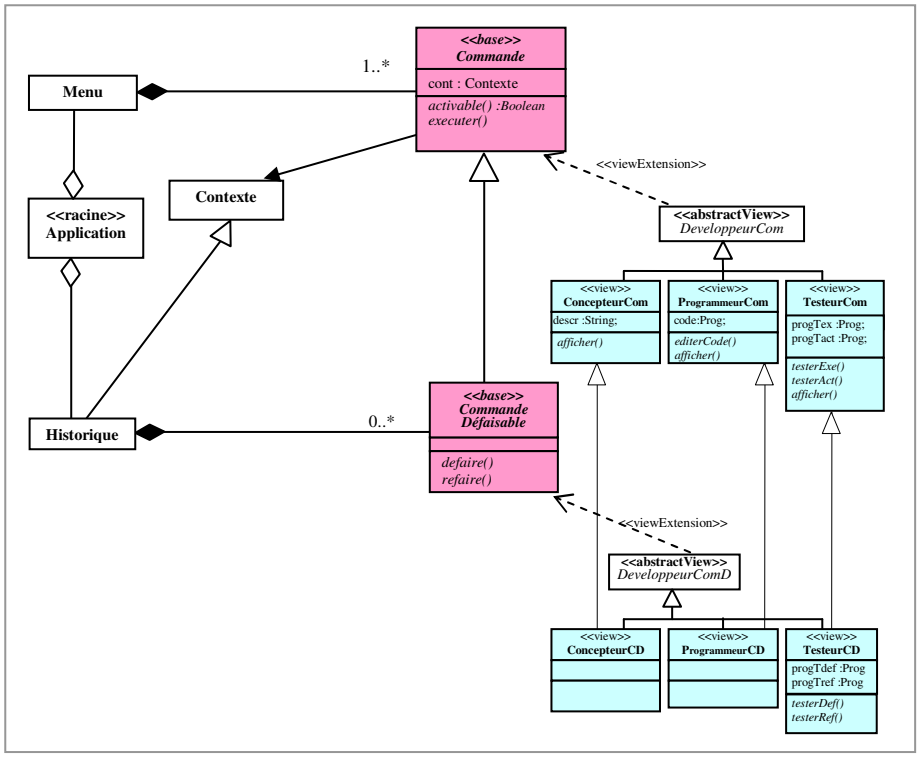

Figure 6. *Partie générique du patron multivues « Commande Défaisable* **»**

4.2.2. Au niveau de l'instanciation du patron

Quand on applique (instancie) un patron « classique » dans un domaine concret, on concrétise dans des sous-classes les méthodes abstraites des classes décrivant le patron. De plus, on prend en compte, généralement implicitement, les besoins des utilisateurs finals. Dans l'approche VUML, les utilisateurs finals ont des points de vue qui influencent le résultat de la conception. Ils doivent donc être explicitement pris en compte. Pour illustrer cela, reprenons le diagramme de classe UML (cf. figure 4) qui décrit les commandes concrètes *ComEnvMessageClient*, *ComAfficherCatalogue* et *ComDefAjoutVoyage*, et intégrons les 2 points de vue utilisateur *Gestionnaire* et *Client*. La figure 7 illustre un extrait significatif du diagramme de classes du système « *Gestion catalogue de voyages* ». Pour simplifier, nous ne redonnons pas la description des vues des classes abstraites, mais décrivons les classes correspondant aux commandes concrètes *EnvMessageClient*, *AfficherCatalogue* et *AjoutVoyage*, la dernière étant défaisable.

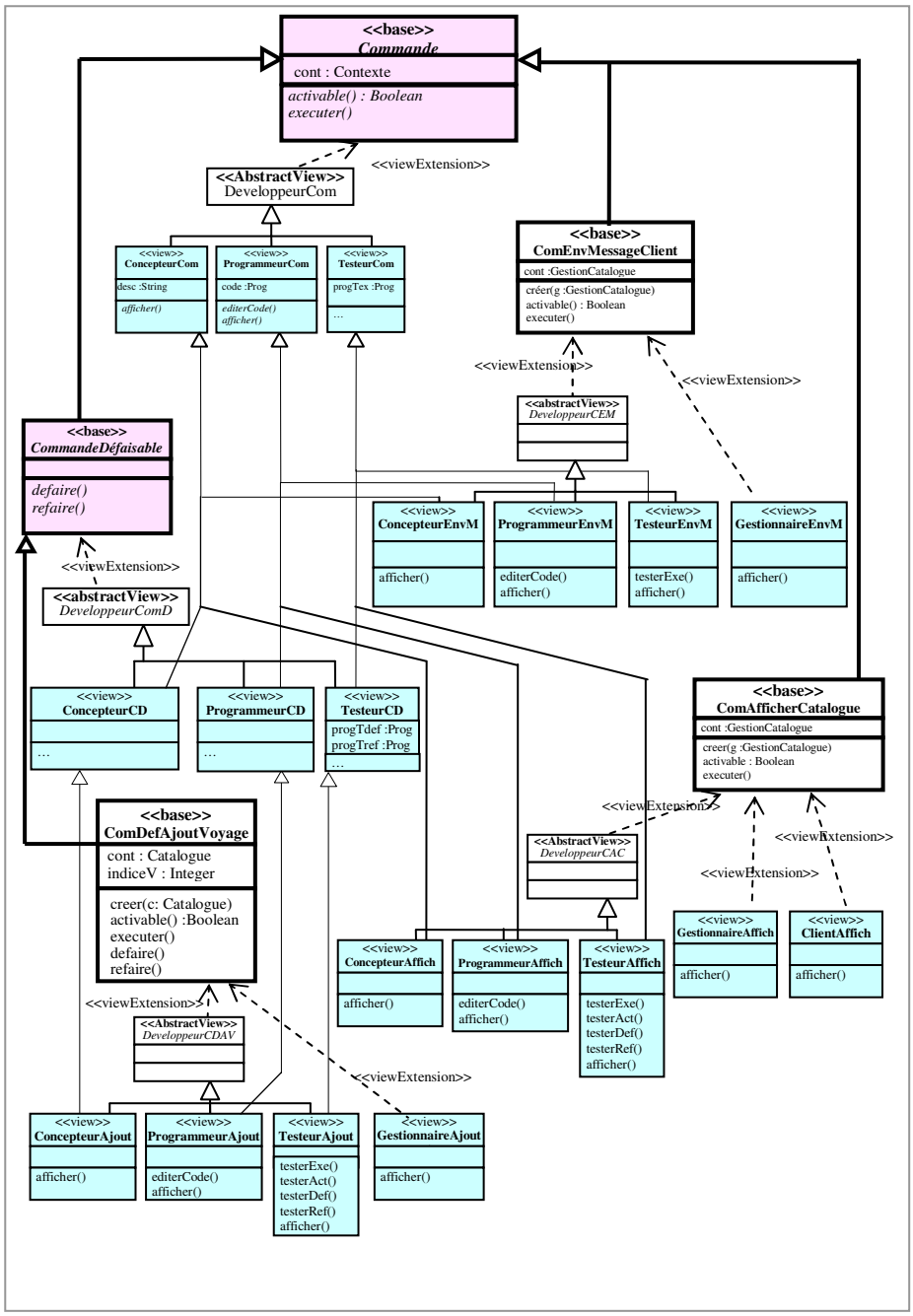

Figure 7. *Application du patron multivues de «Commande Défaisable»*

Les commandes sont décrites par des classes multivues dotées d'une base, de vues de développement analogues à celles des classes parentes, et de vues utilisateur variant selon la commande. Les méthodes des vues de développement sont tout d'abord concrétisées lorsqu'elles étaient abstraites dans les vues parentes. C'est le cas par exemple de *editerCode*() et *afficher*() dans la vue *ProgrammeurAjout* de la commande *AjoutVoyage*. Examinons maintenant les vues associées aux points de vue des utilisateurs finals. Pour la commande EnvMessageClient, le point de vue utilisateur est le *Gestionnaire* seulement puisque l'envoi d'un message à un client lui est réservé. Pour la commande AfficherCatalogue, les points de vue utilisateur sont le *Gestionnaire* et le *Client*. Les bases de ces classes multivues étant concrètes, elles possèdent une méthode de création dont le paramètre spécifie le contexte de la commande. Par exemple, pour la commande AjoutVoyage, le contexte est le catalogue, alors que pour la commande EnvMessageClient, le contexte est la racine de l'application GestionCatalogue à partir de laquelle on accède aux messages. Lors de cette étape d'instanciation du patron, on peut être amené à ajouter des attributs dans les sous-classes multivues. Ainsi, pour la commande *AjoutVoyage*, l'attribut *indiceV* a été placé dans la base car il est partagé par les 4 points de vue qui s'exercent sur cette commande. On peut remarquer qu'il serait possible d'abstraire les vues *utilisateur*, comme cela a été fait pour les vues *développeur* ce qui présente un intérêt quand il y a plusieurs vues et des caractéristiques à factoriser.

On peut vérifier sur cette étude de cas l'un des atouts de VUML qui consiste à pouvoir ajouter facilement un nouveau point de vue utilisateur. Ce serait le cas par exemple du point de vue *Internaute* qui pourrait accéder au catalogue sans pouvoir réserver un voyage puisqu'il n'a pas les droits d'un client.

5. Conclusion

Dans cet article, nous avons introduit la notion de patron multivues. Le formalisme de description employé est celui du profil VUML développé dans notre équipe (Nassar, 2005). Cette étude s'inscrit dans l'objectif de proposer une démarche globale de développement orienté objet par points de vue qui va de l'analyse jusqu'à l'implémentation.

Nous avons tout d'abord mis en évidence la pertinence de l'approche par points de vue pour la manipulation de patrons, en distinguant le niveau du développement qui requiert des points de vue prédéfinis, et le niveau de l'instanciation dans un domaine d'application particulier qui nécessite les points de vue des utilisateurs finals. Nous avons ensuite validé la notion de patron multivues à travers un exemple significatif.

Nous travaillons actuellement sur la formalisation dans le métamodèle VUML de cette notion de patron multivues, ainsi que sur l'aspect méthodologique, en proposant une démarche détaillée permettant de passer du niveau développement du patron (niveau générique) au niveau instanciation du patron (niveau métier). Cette

démarche peut être en partie automatisable par des transformations entre le modèle VUML générique et le modèle VUML appliqué dans un domaine concret. Le métamodèle enrichi par le concept de patron multivues et la démarche seront intégrés dans un profil P-VUML. Notre but est également de développer une base de patrons multivues réutilisables dans un environnement de conception supportant le profil P-VUML (par exemple Objecteering).

Remerciements

Ces travaux donnant lieu à une thèse en cotutelle dans le cadre du réseau STIC franco-marocain, nous remercions tous les partenaires de ce réseau, en particulier ceux du thème Génie Logiciel.

Bibliographie

Alexander C., The Timeless Way of Building, Oxford University Press, 1979.

Barais O., Muller A., Pessemier N. : « Vers une séparation Entités/Fonctions au sein d'une architecture logicielle à base de composants ». Revue RSTI-L'objet. vol. 11 – n°4/2005, p. 115-139.

Debrauwer L. : « Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME », Thèse de doctorat en Informatique, LIFL, Université des Sciences et Technologies de Lille, Décembre 1998.

Gamma et al. « Design patterns, elements of reusable object-oriented software » Addison-Wesley, 1995.

Kriouile A. : «VBOOM, une méthode orientée objet d'analyse et de conception par points de vue», Thèse d'Etat de l'université Mohammed V, Rabat, Maroc, 1995.

Meyer B., Object-oriented software construction. Prentice Hall, 2nd edition, 1997.

Mili H, Mcheick H., Dargham J., Dalloul S. : « Distribution d'objets avec vues ». Revue L'Objet. Vol. 7, 1-2, 2001, p. 27-44.

Muller A., Caron O., Carré B., Vanwormhoudt : « Réutilisation d'aspects fonctionnels des vues aux composants ». Revue RSTI-L'objet. vol. 9, n°1-2, LMO'2003, 2003, p. 241-255.

Nassar M., Coulette B., Crégut X., Marcaillou S., Kriouile A., « Towards a View based Unified Modeling Language », Proceedings of 5th International Conference on Enterprise Information Systems ICEIS'03, Angers, 23-26 April 2003.

Nassar M. "Analyse/conception par points de vue : le profil VUML", thèse INPT, Toulouse, 28 septembre 2005.

Nassar M., Coulette B., Guiochet J., Ebersold S., El Asri B., Crégut X., Kriouile A, « Vers un profil UML pour la conception de composants multivues ». Revue RSTI-L'Objet, vol.11 –n°4/2005.