Vers un profil UML pour la conception de composants multivues

Mahmoud Nassar*,* — Bernard Coulette* — Jérémie Guiochet*— Sophie Ebersold* — Bouchra El Asri****— Xavier Crégut** — Abdelaziz Kriouile****

** GRIMM-ISYCOM, Université de Toulouse le Mirail, Département de Mathématiques- Informatique, 5, allées A. Machado 31058 Toulouse Cédex*

{nassar, coulette, guiochet, ebersold}@univ-tlse2.fr

*** IRIT-ENSEEIHT, Enseeiht, 2 rue Camichel, 31071 Toulouse Cédex*

cregut@enseeiht.fr

**** ENSAM, B.P. 4024, Béni M'Hamed – Meknès, Maroc*

***** Laboratoire de Génie Logiciel, ENSIAS, B.P. 713 Agdal – Rabat, Maroc*

{elasri, kriouile}@ensias.ma

RÉSUMÉ. *L'objectif de cet article est de présenter un profil UML permettant la construction de composants logiciels multivues. Un composant multivues est une extension de la notion de composant UML permettant de stocker et restituer de l'information en fonction du profil de l'utilisateur (point de vue), et offrant la possibilité de changement dynamique de point de vue. Dans cet article, nous présentons tout d'abord la notion de classe multivues, puis le composant multivues et son intégration dans un profil UML. Nous illustrons à l'aide d'un exemple concret la mise en oeuvre d'une modélisation objet multivues. La transition vers la phase de codage est présentée à travers un patron d'implémentation générique.*

ABSTRACT. *This paper aims to present an UML profile based on multiviews components. A multiviews component allows to encapsulate and deliver information according to the user's point of view and offers mechanisms to manage the dynamic evolution of viewpoints and consistency among views. In this paper, we first present the notion of multiviews component and its integration into a UML profile, and thus detail an object-oriented multiviews modelling applied to a concrete example. The transition to the coding stage is described through a generic implementation pattern.*

MOTS-CLÉS : *Vue, Point de vue, classe multivues, composant multivues, profil UML, patron d'implémentation.*

KEYWORDS: *View, viewpoint, multiviews class, multiviews component, UML profile, implementation pattern.*

1. Motivation et contexte

Dans la société de l'information qui se développe à grande vitesse notamment à travers l'Internet, l'accès aux informations devra être ouvert au plus grand nombre de citoyens. Pour cela, cet accès devra être différencié de façon à respecter les cultures, les niveaux de sensibilisation, les différences de vitesse d'apprentissage, les droits du citoyen et la protection des informations personnelles. Dans cette perspective, nous pensons que le développement et la maintenance de systèmes d'informations "centrés sur les points de vue des acteurs" jouera un rôle stratégique dans le futur. A l'instar des chercheurs travaillant dans l'ingénierie des exigences (Charrel, 2004), nous préconisons de prendre en compte les besoins des acteurs d'un système d'information (développeurs, utilisateurs finals, exploitants...) le plus tôt possible dans le processus de développement.

Lors du développement d'un système complexe (Lemoigne, 1990), la construction directe d'un modèle global prenant en compte simultanément tous les besoins des acteurs est impossible. Dans la réalité, soit plusieurs modèles partiels sont développés séparément et coexistent avec les risques d'incohérence associés, soit le modèle global doit être fréquemment remis en cause, parfois profondément, quand les besoins des utilisateurs évoluent.

L'objectif que nous poursuivons depuis plusieurs années — à l'IRIT tout d'abord, puis actuellement au sein de GRIMM-Isycom et de ses partenaires de l'ENSIAS — est de proposer une méthodologie d'analyse/conception de systèmes centrée sur les points de vue des acteurs interagissant avec le système. Ces concepts de vue/point de vue ont été étudiés dans la plupart des domaines de l'informatique : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de Génie logiciel, etc. (Shilling et al. 1989, Clarke et al. 1999, Finkelstein et al. 1990, Abiteboul et al. 1991, Carré et al. 1990, Coulette et al. 1996, Debrauwer 1998, Mili et al. 2001, Muller et al. 2003, etc.). Dans ce cadre, plusieurs approches ont été menées pour intégrer les vues/points de vue. Ces approches n'emploient pas forcément les termes de point de vue ou de vue mais introduisent des termes sémantiquement proches : sujet (Harrison et al., 1993), rôle (Anderson et al. 1992, Kristensen, 1995), aspect (Kiczales et al. 1997, Szyperski 1999), etc.

La prise en compte des points de vue lors des phases de définition de l'architecture a donné naissance à la norme IEEE 1471 (Hilliard, 2000). Cette recommandation propose un modèle conceptuel général pour décrire l'architecture d'un système. Chaque acteur (participant) du système a un ou plusieurs centres d'intérêt, chacun d'eux étant couvert par un ou plusieurs points de vue. Chaque point de vue est relié à une unique vue sur le système, chaque vue consistant en un ou plusieurs modèles. La dépendance entre les modèles d'une vue peut être explicitée à travers des relations de type UML. Cependant, la norme IEEE 1471 se situe à un niveau de granularité large, ne propose pas de langage de modélisation particulier, et n'offre pas de guide sur la manière de trouver les points de vue.

Les travaux que nous avons menés à partir de 1993 ont donné lieu à la définition du langage VBOOL (Marcaillou et al., 1994), et à la méthode associée VBOOM (Kriouile 1995, Coulette et al. 1996). Parmi les apports principaux de l'approche, on peut citer la modélisation décentralisée selon des points de vue multiples, l'introduction du concept de classe flexible pour supporter les vues de granularité fine, l'instanciation d'une classe flexible selon un point de vue donné, le changement dynamique de point de vue, un mécanisme de gestion de la cohérence entre sous-modèles (Marcaillou, 1995). Par contre, l'expérience a permis de mettre en évidence plusieurs limites de VBOOM ou de sa mise en œuvre : subjectivité dans la définition des vues, rigidité de l'héritage multiple pour prendre en compte l'évolution d'un modèle à bases de vues, complexité d'une programmation avec le langage dédié VBOOL, manque d'outils supports performants, etc.

C'est pour cette raison que nous avons décidé de reconsidérer l'approche développée dans VBOOM en conservant l'objectif d'une modélisation multivues à granularité fine. L'idée est d'associer, d'une manière déterministe, un point de vue à chaque type d'acteur et une vue unique à chaque point de vue. Par ailleurs, prenant en compte l'effet "UML" et sa généralisation comme standard "de facto" dans la modélisation de systèmes logiciels, nous avons opté pour la réutilisation des standards de l'OMG (OMG-site, 2004) et ciblons les langages à objet du marché. La notion de vue proposée dans UML (Unified Modeling Language) (UML, 2001) ne correspond pas à nos besoins car c'est un moyen offert au concepteur pour décrire l'architecture d'un système en fonction des phases de développement (cas d'utilisation, logique, composants, déploiement). UML n'offre pas de mécanisme à granularité fine pour intégrer les points de vue des acteurs au cœur même de la modélisation, à savoir dans les diagrammes de classes.

En outre, l'ingénierie du logiciel basée sur les composants est en pleine effervescence. L'objectif est de développer des systèmes par assemblage de composants à l'instar d'autres disciplines qui proposent des standards de description de composants et des catalogues de composants. Les premiers travaux sur la description d'architectures logicielles à base de composants ont été menés au début des années 90 avec les ADL (Architecture Description Language (Shaw et al., 1996). L'idée est de fournir une structure de haut niveau de l'application plutôt que l'implantation dans un code source spécifique. Medvidovic et Taylor (Medvidovic et al., 2000) ont identifié trois concepts clés utilisés dans les différents langages de description d'architecture : le *composant*, le *connecteur* et la *configuration*. Le composant est une unité de calcul ou de stockage à laquelle est associée une unité d'implantation. Le connecteur correspond à un élément d'architecture qui modélise de manière explicite les interactions entre un ou plusieurs composants. La configuration définit les propriétés architecturales de connectivité et de conformité des composants formant l'application. La configuration structurelle vérifie la correspondance entre les interfaces des composants et des connecteurs. La configuration comportementale modélise le comportement en décrivant l'évolution

des propriétés fonctionnelles entre composants et connecteurs. La configuration non fonctionnelle définit les propriétés de qualité de service et de synchronisation.

En respectant ce cadre, nous proposons d'intégrer le concept de composant multivues dans l'approche VUML. La spécificité d'un composant multivues est la possibilité d'avoir des interfaces dont l'accessibilité et le comportement changent dynamiquement selon le point de vue de l'utilisateur courant. Aussi sommes-nous en train de développer une méthodologie d'analyse/conception dont le noyau est un profil UML, appelé VUML (View based Unified Modeling Language), qui supporte la construction de composants de conception multivues. VUML offre déjà la notion de classe multivues (Nassar et al. 2002, Nassar et al. 2003, Nassar 2004). La représentation d'une classe multivues sous forme de composant permet de décrire une architecture en composants multivues réutilisables ce qui permettra le déploiement d'une application multivues. Le choix d'un profil UML permet de s'appuyer sur les mécanismes d'extension du formalisme UML, notamment les stéréotypes, sans remettre en cause le métamodèle UML existant, ce qui ouvre la voie au support de VUML par les principaux outils UML du marché.

Cet article a pour but de présenter globalement notre méthodologie de développement multivues, mais son apport principal réside dans l'intégration de la notion de composant UML dans notre approche. L'article est structuré de la manière suivante. Nous présentons tout d'abord l'étude de cas sur laquelle nous nous appuyons pour illustrer notre approche (section2). Dans la section 3, nous rappelons le principe d'un profil UML puis la notion de composant dans la version 2.0 d'UML. Dans la section 4, nous présentons en premier lieu les principes associés au concept de classe multivues, puis nous montrons sa représentation sous forme de composant multivues. La section 5 est dédiée à la présentation d'une démarche de mise en oeuvre de la modélisation selon VUML. Dans la section 6, nous montrons comment se fait le passage d'un modèle VUML à du code objet standard à travers un patron d'implémentation générique. Dans la conclusion, nous faisons le point sur le travail effectué, et nous présentons nos principales perspectives de recherche.

2. Etude de cas

Nous nous appuyons sur la modélisation d'un Système d'Enseignement à Distance (SED) tel qu'il en existe dans les universités pour illustrer notre approche. Une présentation plus détaillée de cet exemple peut être trouvée dans (Nassar, 2004). Cet exemple a été choisi comme étude de cas commune par les partenaires du réseau STIC franco-marocain en Génie Logiciel (Coulette et al., 2002).

Le Système d'Enseignement à Distance (SED) permet à des étudiants de suivre des formations à distance. Pour simplifier, nous considérons que le système ne cible qu'un seul site, géré par un responsable. Un site propose des formations (cours) auxquels peuvent s'inscrire des personnes (alors qualifiées d'étudiants), qui

acquittent des droits en fonction des formations suivies. Afin de réduire la taille des modèles nous nous limitons aux acteurs et aux activités suivants :

- les étudiants s'inscrivent à des formations, suivent des formations à distance, et posent des questions sur les formations.
- les enseignants auteurs de formations produisent des formations et les mettent à jour, produisent les compléments de formation et la liste des ouvrages conseillés pour approfondir une formation, …
- les enseignants tuteurs assurent le suivi des étudiants, répondent à leurs interrogations, posent et corrigent des examens, …
- le responsable de site ajoute de nouvelles formations, gère les inscriptions des étudiants et affecte les enseignants responsables des formations.
- Le conseil pédagogique (doté d'un président) réunit les jurys, fait le bilan pédagogique, définit la stratégie du SED, traite les litiges et les cas particuliers, …
- le directeur du SED définit la stratégie globale et produit les bilans de fonctionnement.

La figure 1 illustre un sous-ensemble des cas d'utilisation identifiés : Gestion de la scolarité, Gestion des formations, Suivi d'une formation, Supervision.

Diagramme de classes UML du SED

Les entités principales du SED sont les suivantes : *Site, Catalogue, Formation, Documentation, Exercice, Question, Examen,* etc. Pour simplifier l'illustration des concepts introduits par VUML, nous choisissons l'entité *Formation*. Après une analyse/conception réalisée en UML, on considère qu'une formation est caractérisée par un enseignant responsable, un numéro d'identification unique, un titre, un nombre de crédits (équivalence ECTS[1]), un niveau de difficulté, un coefficient, une documentation, des exercices, un examen, un prix à payer lors de l'inscription, les étudiants inscrits. Un exercice est caractérisé par un niveau de difficulté, et peut avoir plusieurs solutions. Un étudiant s'inscrit à des formations. Pour chaque formation à laquelle il est inscrit, il peut poser des questions à l'enseignant responsable de cette formation. L'enseignant peut connaître la liste des étudiants suivant sa formation et les auteurs des questions sur cette formation. La figure 2 ci-dessous présente un extrait du diagramme de classes UML du SED. Une description détaillée peut être trouvée dans (Nassar, 2004).

[1] European Credit Transfer System

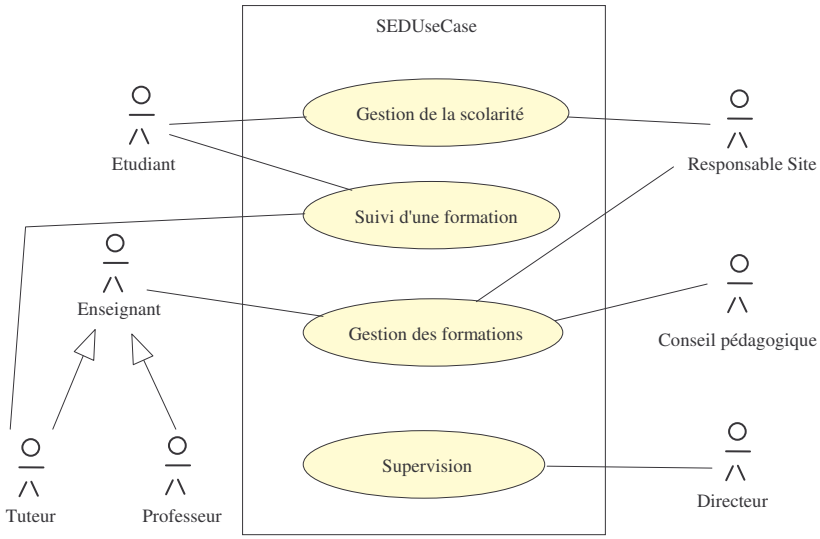

Figure 1. *Cas d'utilisation du SED (extrait)*

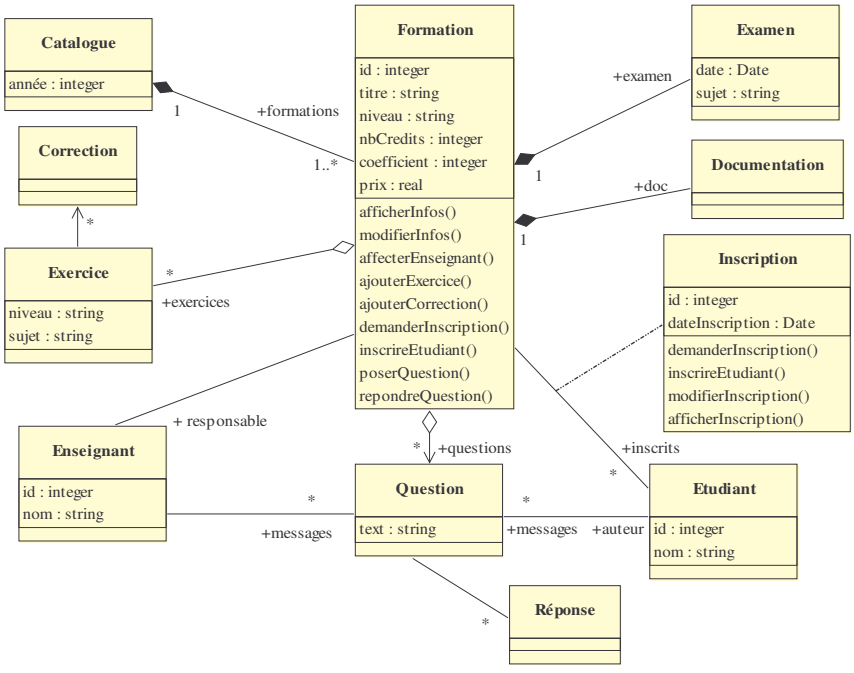

Figure 2. *Diagramme de classes UML du SED (extrait)*

Discussion

Dans un diagramme de classes UML, chaque acteur a potentiellement les mêmes droits d'accès aux informations et aux services encapsulés dans les classes. Cependant, les acteurs n'ont pas les mêmes besoins et les mêmes responsabilités. A titre d'exemple, dans le diagramme de classes de la figure 2, l'étudiant peut accéder à toute la documentation concernant une formation et à toutes les solutions des exercices. Ceci n'est pas acceptable car une partie de la documentation doit être cachée à l'étudiant, et certaines solutions des exercices ne doivent être accessibles qu'à l'enseignant.

Dans UML, le contrôle des droits d'accès ne peut pas être modélisé dans le diagramme de classes mais seulement dans les diagrammes dynamiques (diagramme de séquence, diagramme de collaboration,…), et dans ce cas le contrôle doit être programmé dans les méthodes des classes. Notre objectif est de décrire ces droits d'accès aux informations à un niveau d'abstraction élevé, c'est-à-dire lors de la phase d'analyse.

Afin d'atteindre ce but, nous avons décidé d'introduire un nouveau type de classe, la **classe multivues**, qui permet de définir les vues associées aux profils des acteurs. Le défi est de placer judicieusement les informations (attributs, méthodes, contraintes) dans les vues d'une classe donnée, ce qui permet de définir des droits d'accès. Le principe consiste donc à associer un point de vue à chaque acteur et à lui affecter des droits d'accès aux informations offertes par la classe tout en ayant la possibilité de passer d'un point de vue à un autre d'une manière dynamique. Comme nous allons le voir dans la section 4, les vues ne peuvent pas être modélisées comme des classes descendantes. Nous avons besoin d'une nouvelle relation de dépendance que nous appelons **viewExtension**.

3. Profils et composants en UML

L'objectif de cet article étant de décrire un profil UML pour supporter l'élaboration de composants multivues, nous commençons par présenter les notions de profil et de composant en UML, en prenant en compte les dernières évolutions de ce formalisme (OMG-site, 2004).

3.1. *Profils UML*

Par rapport au formalisme UML standard, les développeurs souhaitent souvent rajouter des caractéristiques supplémentaires pour tenir compte de la spécificité de leur domaine d'application. Afin de satisfaire ce besoin, UML est doté d'un mécanisme d'extensibilité fondé sur les stéréotypes, les contraintes et les valeurs étiquetées. Un tel mécanisme permet de particulariser le métamodèle UML pour qu'il prenne en considération les besoins de modélisation spécifiques. Le résultat est un profil UML. Plusieurs profils UML ont été standardisés ou sont en cours de standardisation, tels que : SPEM (modélisation des procédés logiciels), EDOC

(modélisation des applications distribuées), Scheduling, Performance and Time (modélisation des systèmes embarqués), UML pour CORBA, UML pour les EJB, etc. (cf. OMG-site, 2004).

Un profil UML peut être décrit simplement comme un ensemble d'éléments de modélisation ajoutés au métamodèle UML (OMG, 2003). On peut cependant lui ajouter la notion de règles comme préconisé par SOFTEAM (Softeam, 1999). Ces règles permettent de décrire et d'automatiser un savoir-faire sur UML. De cette façon, les profils UML constituent un moyen efficace pour spécifier et guider le processus de développement UML. Durant le développement, à chaque phase, les profils permettent d'exprimer comment utiliser UML, quels sont les produits de développement attendus, et quelles sont les règles que le modèle doit respecter. En reprenant cette approche (Softeam, 1999), un profil UML peut être décrit par :

- les éléments UML utilisés (éléments d'UML pertinents pour un domaine donné),
- les extensions UML ajoutées (stéréotypes, contraintes, valeurs marquées),
- les règles de validation (règles vérifiant des critères de cohérence sur un modèle pour un profil donné),
- les règles de présentation (les diagrammes UML doivent présenter certaines informations et en cacher d'autres),
- les règles de transformation (règles de génération de code et *designs patterns* permettant d'assister ou d'automatiser le développement).

3.2. *La notion de composant en UML*

Un composant UML est un module logiciel autonome (OMG, 2003) dont le comportement est défini en terme d'interfaces fournies et d'interfaces requises. Un composant peut être considéré comme un type, sa conformité étant définie par ses interfaces. On peut ainsi lui substituer un autre composant qui possède les mêmes interfaces. Il est modélisé lors du passage à l'implémentation dans le diagramme de composants et instancié lors du déploiement, l'instanciation pouvant se faire de façon indirecte via ses constituants internes (*parts*).

Dans le métamodèle d'UML 2.0, un composant est un sous-type de *Class* (cf. figure 3). Ainsi, un composant encapsule des attributs et des opérations, et peut participer à des relations d'association et de généralisation. Il est une abstraction d'un ensemble de *realizingClassifiers* qui réalisent son comportement. De plus, un composant peut avoir une structure interne et un ensemble de ports qui structurent des points d'interaction avec l'environnement extérieur.

Un composant possède un certain nombre d'interfaces fournies et requises. Ces interfaces forment la base pour l'assemblage des composants. Une interface fournie est une interface offerte par le composant à son environnement. Cette interface peut être implémentée soit directement par le composant ou bien par l'un de ses

realizingClassifiers, ou bien être le type d'un port fourni par le composant. Une interface requise est une interface que le composant requiert de la part d'autres composants pour réaliser ses services. Une telle interface peut être utilisée par le composant ou ses *realizingClassifier*, ou bien être le type d'un de ses ports requis (OMG, 2003).

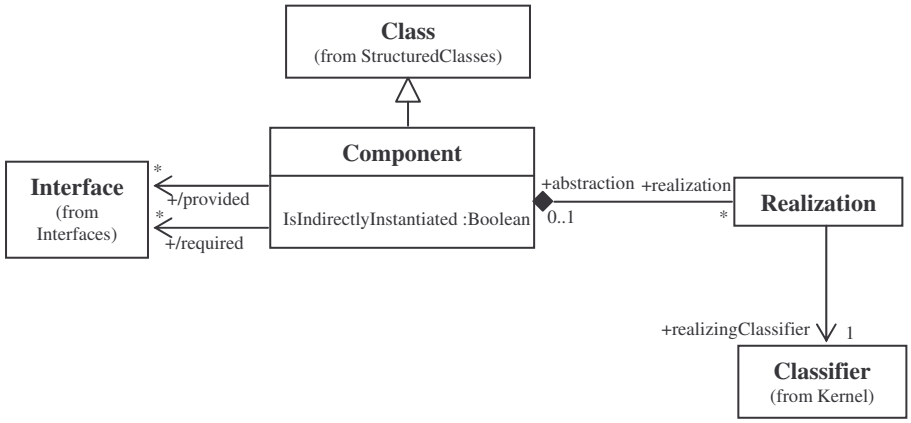

Figure 3. *Métaclasse définissant le concept de composant en UML 2.0*

4. Vers un profil UML supportant les composants multivues

Comme nous l'avons mentionné dans l'introduction, la notion de vue proposée dans UML ne correspond pas à nos besoins. UML n'offre pas de mécanisme de granularité fine pour modéliser les points de vue des acteurs tout au long du développement. En effet, si les cas d'utilisation et les diagrammes de séquence UML sont des moyens efficaces pour modéliser les besoins et les droits d'accès des acteurs, le diagramme de classes — qui joue un rôle central dans la modélisation — n'en garde pas de trace. Notre objectif est de spécialiser UML afin qu'il supporte la notion de vue/point de vue de l'analyse jusqu'au déploiement d'une application. Cette spécialisation se traduit par l'introduction d'un ensemble de stéréotypes que nous regroupons sous forme de profil UML.

Dans cette section, nous présentons tout d'abord les principes de base de VUML, puis nous donnons quelques définitions concernant les concepts de classe multivues et composant multivues. Nous détaillons ensuite le concept de *classe multivues* et les mécanismes liés. La dernière partie de cette section montre le passage d'une classe multivues (élément central de la conception) au composant multivues (élément central d'une application destinée à être adaptable, multi-utilisateurs et distribuée).

4.1. *Principes*

VUML est fondée sur une approche centrée utilisateur, dont l'objectif est de représenter les besoins et les droits d'accès des utilisateurs de l'analyse jusqu'à l'implémentation. VUML a le même objectif d'une modélisation multivues à granularité fine que la méthode VBOOM élaborée antérieurement dans notre équipe (Kriouile, 1995), mais en adoptant les principes suivants : **(i)** association déterministe d'un point de vue à chaque type d'acteur identifié lors de l'analyse des besoins (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 des vues par l'intermédiaire de la délégation (au lieu de l'héritage multiple).

La singularité de VUML réside dans le fait d'associer de façon unique une vue à chaque type d'acteur du système. Cette décision simplifie tout d'abord la détermination des vues sur une entité donnée, et le fort indéterminisme qui la caractérise, et réduit aussi nettement les problèmes liés à la composition des vues inhérents aux propositions à base de vues multiples (Debrauwer 1998, Kriouile 1995, Mili et al. 2001, Muller et al. 2003).

4.2. *Définitions informelles*

Nous définissons informellement les concepts clés de VUML comme suit : Un **acteur** est une entité logique ou physique qui interagit avec le système. Un **point de vue** est la « vision » d'un acteur sur le système (ou sur une partie de ce système). Une **vue** est un élément de modélisation (statique) ; elle correspond à l'application d'un point de vue sur une entité donnée. Par simplification de langage, nous dirons dans la suite qu'une vue est associée à un acteur, en considérant comme implicite l'entité sur laquelle le point de vue s'applique. Le concept de **classe multivues** est l'élément clé de l'approche VUML. Par rapport à une classe "habituelle", une classe multivues est dotée d'une ou plusieurs vues représentant les besoins et les droits spécifiques des acteurs. Un **composant multivues** (CMV) est un module autonome réutilisable représentant une classe multivues.

4.3. *Concept de classe multivues*

Une classe multivues est une structure composée d'une classe de **base** (stéréotype *base*) reliée à un ensemble de **vues** (stéréotype *view*) via une relation de dépendance (stéréotype *viewExtension*) (cf. figure 4). La relation de dépendance *viewExtension* ressemble à la relation d'héritage car elle permet de copier virtuellement les caractéristiques de la base dans les vues avec la possibilité de redéfinir ces caractéristiques ou d'ajouter de nouvelles caractéristiques. La différence ente ces deux relations réside dans le fait que le partage est réalisé de manière dynamique et non statique. Les vues dépendent de la base au sens où les valeurs d'attributs et les méthodes de la base sont implicitement partagées par les vues de la classe multivues. A titre d'exemple, sur la figure 4, chacune des vues *Vue1*, *Vue2* et *VueN* possède, en plus de ses propres caractéristiques, l'attribut *b* et

les méthodes *f()* et *g()* provenant de la base. La méthode *f()* est redéfinie dans les vues *Vue1* et *Vue2*. La vue *Vue2* ajoute la méthode *h()* et la vue *VueN* ajoute les méthodes *i()* et *m()* sans redéfinir les méthodes *f()* et *g()* de la base. Lors de l'exécution, les trois vues *Vue1*, *Vue2* et *VueN* possèderont la même valeur pour l'attribut *b*.

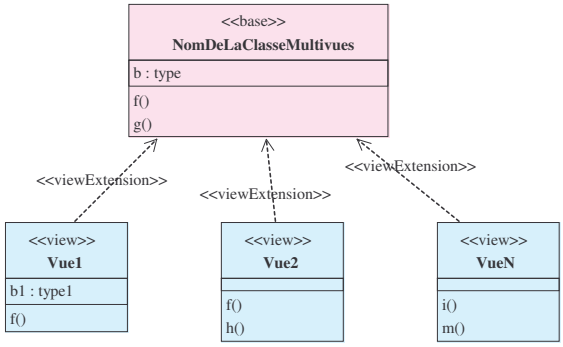

Figure 4. *Structure statique d'une classe multivues*

La figure 5 ci-dessous illustre le diagramme de classes VUML correspondant au diagramme de classes UML de la figure 2. Nous pouvons remarquer que chaque classe du modèle UML peut devenir une classe multivues, selon les droits d'accès aux informations des acteurs sur cette classe. Dans ce modèle VUML simplifié, nous spécifions seulement 3 vues associées aux acteurs *Responsable de site*, *Enseignant* et *Etudiant*. La classe multivues *Formation* est constituée d'une base et de trois vues correspondant aux points de vue de ces acteurs. La base contient les informations (attributs et méthodes) partagées par ces acteurs. La classe *VueEnseignantFormation* décrit la spécificité d'une formation selon le point de vue d'un enseignant, la classe *VueEtudiantFormation* décrit la spécificité d'une formation selon le point de vue d'un étudiant, et la classe *VueResponsableSiteFormation* décrit la spécificité d'une formation selon le point de vue d'un responsable de site. On peut remarquer que la méthode *afficherInfos()* de la base est redéfinie dans les trois vues. Toutes les classes d'un système sont potentiellement multivues car on peut toujours faire évoluer un système en ajoutant un point de vue.

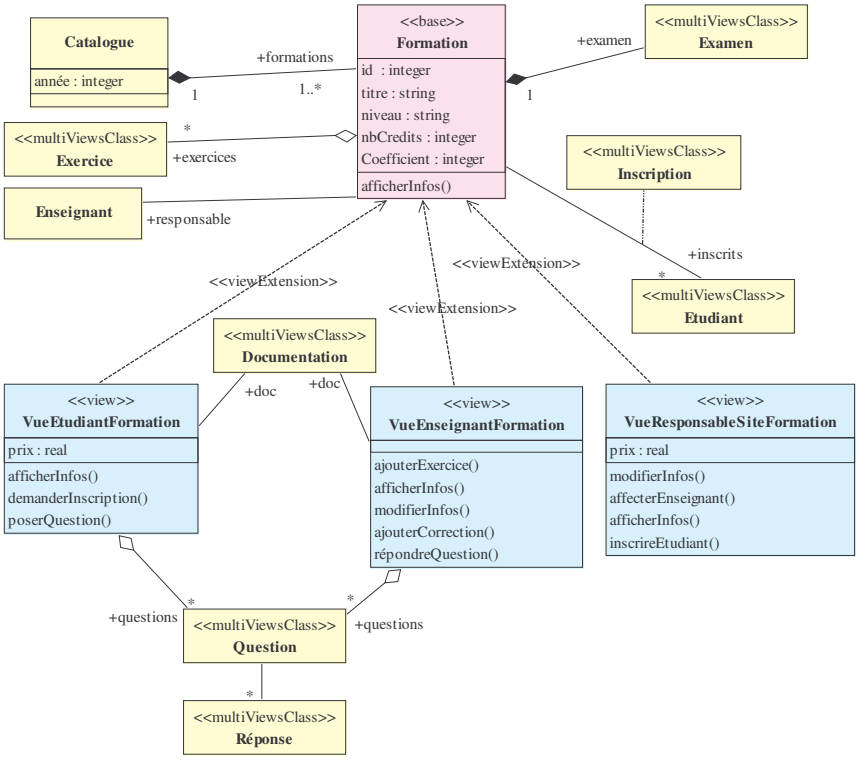

Figure 5. *Modèle VUML du SED (extrait)*

Par rapport au diagramme de classes UML (cf. figure 2), la distribution des attributs et des méthodes sur les classes est changée. A titre d'exemple, l'attribut prix est mis dans la vue *VueResponsableSiteFormation* et dans la vue *VueEtudiantFormation* associées, respectivement, au responsable du site et à l'étudiant, car l'enseignant n'a pas accès à cette information. La méthode *modifierInfos()* de la classe *Formation* est masquée à l'étudiant car ce dernier ne doit pas modifier les informations d'une formation. Les classes *Question* et *Documentation* sont reliées à la vue *VueEnseignantFormation* et à la vue *VueEtudiantFormation* et non pas à la vue *VueResponsableSiteFormation*, car les informations et les services de ces classes ne sont pas dédiés au responsable du site. Par contre la classe *Enseignant* est reliée à la base ; en effet, tous les acteurs ont droit de connaître les informations concernant l'enseignant responsable d'une formation. Nous remarquons sur la figure 5 que les classes *Documentation*, *Inscription*, *Etudiant*, *Exercice*, *Examen*, *Question* et *Réponse* sont stéréotypées par « multiViewsClass ». En effet, pour des raisons de lisibilité des diagrammes de classes VUML, nous avons introduit le stéréotype « multiViewsClass » qui permet de représenter des classes multivues non éclatées (iconifiées).

4.4. *Spécialisation d'une classe multivues*

La relation de spécialisation n'est pas modifiée par l'introduction du concept de classe multivues. Elle correspond dans VUML à un héritage qui implique donc une relation de sous-typage et la possibilité pour la sous-classe de redéfinir les opérations héritées. Une classe multivues peut ainsi avoir des sous-classes qui sont automatiquement multivues. Le lien d'héritage est positionné entre les bases. Les vues de la classe parente deviennent des vues de la classe fille. Il est possible de définir de nouvelles vues pour la classe fille ou de redéfinir une vue parente (Nassar et al., 2003).

Par exemple, nous nous plaçons dans le cas où une entreprise finance pour ses employés une formation du SED appelée alors *formation continue*. Le concept de *formation continue* est différent de celui de *formation* vu précédemment puisqu'il comporte notamment la gestion de dates de formation. Il fait aussi intervenir d'autres acteurs tels que les *professionnels* et les *entreprises*. Elle a bien sûr des informations communes avec une formation classique comme la documentation, les exercices, etc. On modélise cette situation en spécialisant la classe multivues *Formation* en une autre classe multivues : *FormationContinue*. Le diagramme de classes présenté dans la figure 6 illustre à titre d'exemple cette notion d'héritage en VUML. Les différents attributs, méthodes et associations ne sont pas représentés dans leur intégralité. La classe multivues *FormationContinue* hérite des vues de sa classe mère. Ainsi, il existe pour cette classe une vue *enseignant* permettant d'effectuer les mises à jour des exercices, de la documentation, etc.

De plus, il apparaît judicieux de créer deux nouveaux points de vue : celui du *Professionnel* qui suivra une formation continue et celui de l'*Entreprise* dont le professionnel fait partie. Les différences entre les informations concernant un étudiant classique et celles d'un professionnel (tel que le besoin professionnel) justifient le fait de spécifier une nouvelle vue pour une *formation continue* : *VueProfessionnelFormationContinue*. Celle-ci hérite en la redéfinissant de *VueEtudiantFormation* puisque le professionnel — cas particulier d'étudiant — accède aux attributs et méthodes de la vue de l'étudiant comme la consultation de la documentation, la possibilité de poser des questions, etc. De même la vue du responsable du site est redéfinie en *VueResponsableSiteFormationContinue* car un ensemble d'attributs et de méthodes sont rajoutés ou redéfinis (comme *inscrireEtudiant()* qui permet d'inscrire un professionnel étudiant).

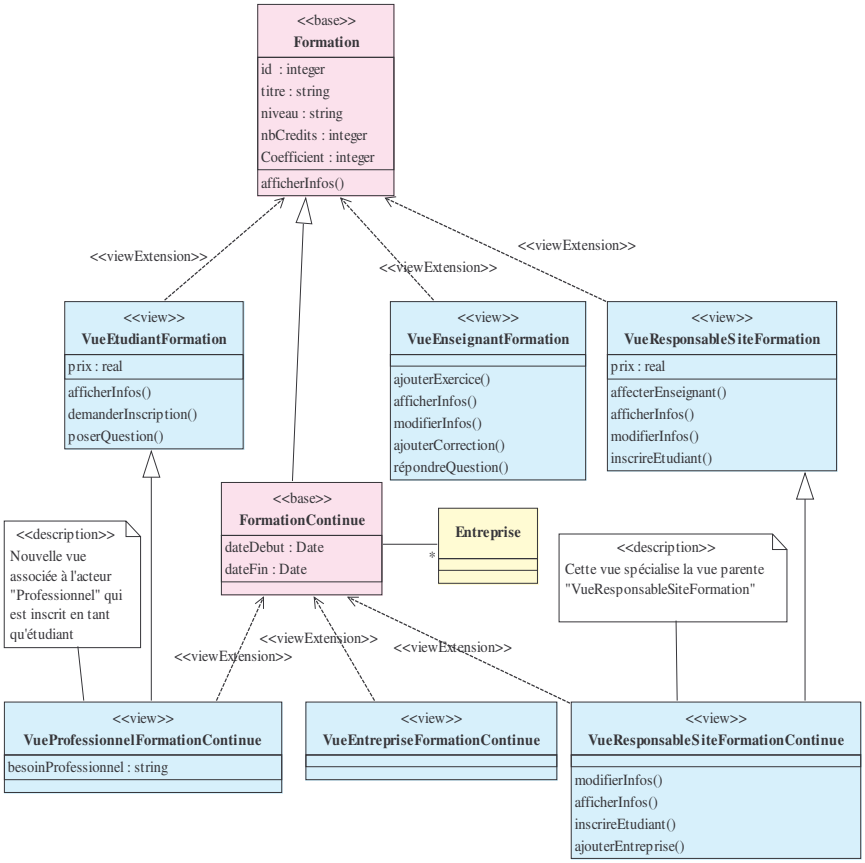

Figure 6. *Illustration de la spécialisation d'une classe multivues*

4.5. *Représentation des dépendances entre les vues*

Les vues d'une classe multivues peuvent naturellement être dépendantes. Il est donc nécessaire de maintenir la cohérence interne d'une telle classe. Autrement dit, les modifications de valeurs d'attributs d'une vue peuvent avoir des répercussions sur les valeurs d'attributs d'autres vues. La gestion de ces répercussions fait généralement partie de la phase d'implémentation au sens où du code doit être introduit pour faire les mises à jour nécessaires. Cependant, nous pensons que ces dépendances — caractéristiques du système modélisé — doivent être explicitées le plus tôt possible dans le développement, c'est-à-dire durant la phase de conception ; pour cela nous utilisons une relation de dépendance stéréotypée par <<*viewDependency*>>, les notes d'UML et le langage OCL. Ces dépendances sont ensuite prises en compte lors de l'implémentation.

Sur l'exemple de la formation (cf. figure 7), nous avons mis en évidence la dépendance entre les vues *Etudiant* et *Responsable* du site de la classe *Formation*. Une relation de dépendance <<viewDependency>> est placée entre les deux vues pour exprimer le faite que le responsable et l'étudiant doivent utiliser la même valeur pour le prix.

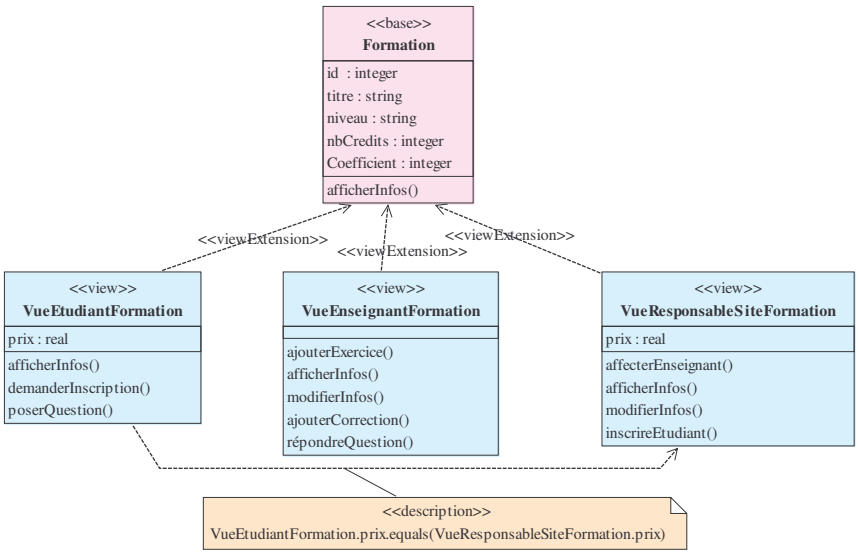

Figure 7. *Illustration d'une dépendance entre deux vues*

4.6. Métamodèle de VUML

Les notions de profil et de métamodèle sont très proches et une utilisation combinée de celles-ci peut être bénéfique pour un projet (Peltier, 2002). Ceci favorise l'émergence de spécifications contenant à la fois la définition de leur profil et leur métamodèle MOF (Meta Object Facility). Dans cette optique, nous avons défini un métamodèle pour décrire la manière dont les concepts de notre profil s'intègrent dans le métamodèle UML. Ce métamodèle étend celui d'UML avec de nouveaux éléments de modélisation : *MultiViewsClass*, *AbstractView*, *Base*, *View*, *ViewExtension* et *ViewDependency*. Le métamodèle défini montre qu'une classe multivues est composée d'une classe de base et d'un ensemble de vues (cf. figure 8). Ces dernières sont reliées à la base via une relation de dépendance particulière (extension) et peuvent être également reliées à d'autres vues par des relations de dépendances classiques (pour gérer la cohérence). Chaque vue est associée à un acteur unique. La table de la figure 9 décrit, pour chaque stéréotype : le nom, l'élément de modélisation associé et une description informelle de sa sémantique.

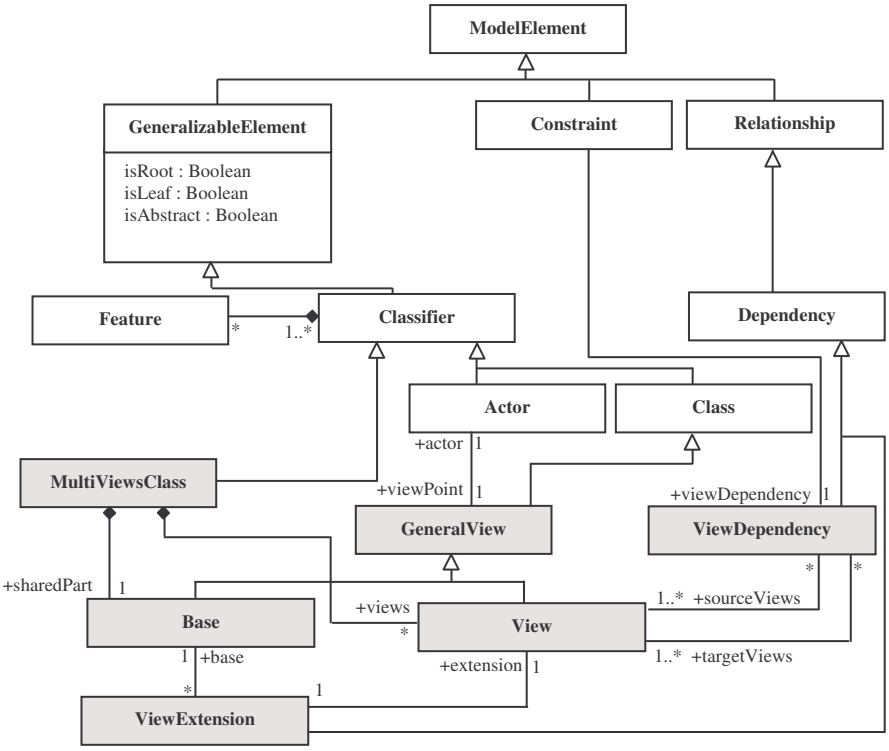

Figure 8. *Fragment du métamodèle associé au profil VUML*

Stéréotype	Elément de modélisation	Sémantique informelle
base	Classe	Classe décrivant la base d'une classe multivues
abstractView	Classe	Classe représentant une vue abstraite. Une telle vue ne peut pas être active.
view	Classe	Classe représentant une vue. Une telle classe doit être non multivues, ne peut hériter que de l'une des vues de sa classe multivues ou des vues de la classe parente de sa classe multivues, et ne peut pas être instanciée directement.
viewExtension	Dépendance	Relation entre une vue et la base d'une classe multivues. La vue étend la base et peut redéfinir ses méthodes.
viewDependency	Dépendance	Relation entre les vues. Des informations de la vue source dépendent d'autres informations de la vue cible.
multiViewsClass	Classe	Classe représentant une classe multivues non éclatée.

Figure 9. *Stéréotypes introduits dans VUML*

4.7. *De la classe au composant multivues*

L'introduction de la notion de classe *multivues* satisfait nos besoins d'analyse/conception jusqu'à l'établissement du diagramme de classes. Pour prendre en compte la transition vers le déploiement, l'évolution d'une architecture et la réutilisation d'éléments d'une application multivues, nous introduisons la notion de *composant multivues*. Un tel composant est la représentation au niveau du diagramme de composants d'une classe multivues. Par rapport aux composants UML "classiques", un composant multivues a la spécificité d'offrir des interfaces dont l'accessibilité et le comportement changent dynamiquement selon la vue correspondant au point de vue courant. En effet, un client accède à un composant selon un point de vue donné. Le client peut activer/désactiver des vues durant l'exécution. Par ailleurs, plusieurs clients peuvent accéder simultanément au même composant selon des points de vues différents (multi-utilisation). Nous travaillons actuellement sur cet aspect multi-utilisation qui sort du cadre de cet article. La figure 10 ci-dessous illustre la métaclasse *MultiViewsComponent* qui définit les éléments formant un composant multivues. Cette métaclasse montre qu'un composant multivues est une extension du composant UML 2.0.

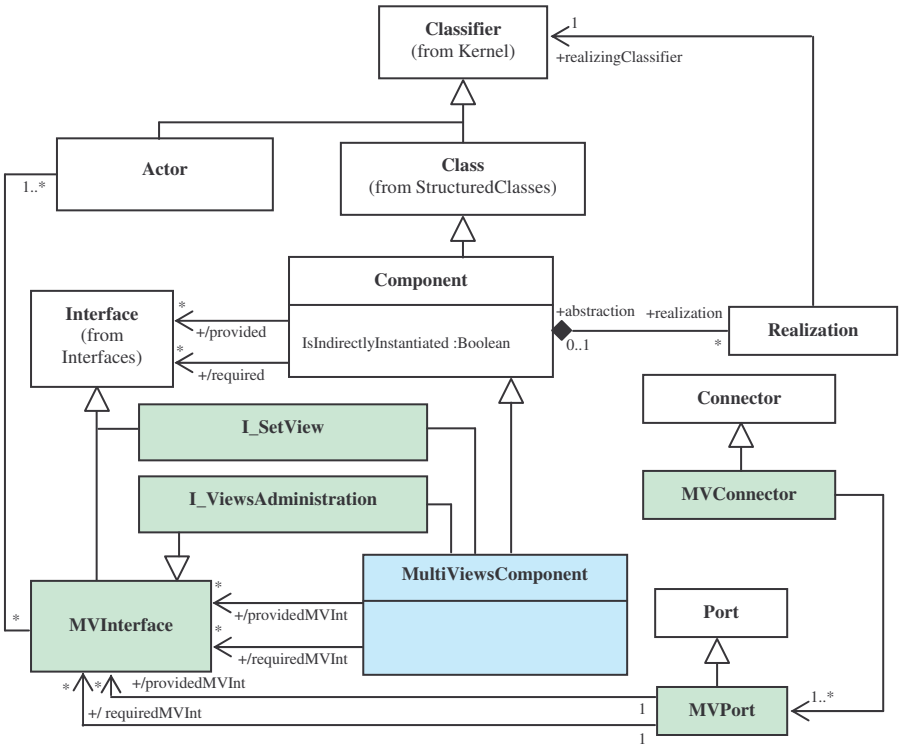

Figure 10. *Métaclasse définissant le concept de composant multivues*

En plus des caractéristiques d'un composant UML 2.0 standard, un composant multivues est ainsi doté d'un ensemble *d'interfaces multivues* fournies et/ou requises qui décrivent le comportement du composant et son interaction avec l'environnement extérieur. Une *interface multivues* (MVInterface) est une interface qui définit, elle aussi, une relation de dépendance "utiliser/fournir" entre le composant et son environnement, mais le comportement des services offerts via cette interface change dynamiquement selon le point de vue actif. Un composant externe doit d'abord activer la vue adéquate avant de pouvoir interagir avec le composant multivues. Cette activation se fait à travers l'interface fournie (*I_SetView*) donnant accès à l'opération *setView().* Chaque composant multivues est doté d'une interface multivues fournie (*I_ViewsAdministration*) qui permet la gestion des vues (ajout/suppression, verrouillage/déverrouillage). Cette interface n'est accessible que par un administrateur (point de vue implicite). Un composant externe requérant cette interface ne peut y accéder que s'il active la vue *Administrateur* à l'exécution.

Un composant multivues doit fournir au moins une interface multivues. Le traitement des messages reçus dépend du point de vue du client. Bien entendu, un composant peut requérir des interfaces multivues sans être lui-même multivues. En effet, l'utilisation par un composant d'une interface multivues signifie que ce composant doit activer une vue donnée avant de solliciter les services d'un composant multivues. Les interfaces multivues doivent être organisées sous forme d'un seul *port multivues* portant le nom *MVPort*. La communication entre le composant et son environnement via un tel port est filtrée et routée vers les constituants internes (*parts*) correspondants selon la vue active. L'interaction via un port mutlivues est réalisée moyennant des *connecteurs multivues*. Ces derniers sont des extensions du connecteur UML classique. Ils permettent les assemblages de composants selon un point de vue déterminé, ce qui permet d'avoir une interaction conditionnée par la vue active (El Asri et al., 2005).

La figure 11 ci-dessous représente un diagramme de composants simplifié. Ce diagramme illustre trois composants multivues (*Formation*, *Inscription*, *Exercice*), et un composant classique WEB. Le composant *Exercice* est un composant multivues qui interagit avec le composant *Formation*. Cette interaction est conditionnée par le point de vue actif. Par exemple l'étudiant s'intéressera à la consultation des exercices alors qu'un enseignant souhaitera la mise à jour des exercices. Par contre, WEB est un composant classique du système car on suppose que tous les acteurs voient ce composant de la même manière.

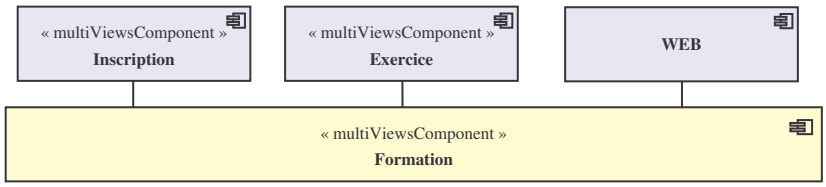

Figure 11. *Exemple de composant multivues « Formation »*

La figure 12 ci-dessous explicite les interfaces (simples et multivues) pour le composant multivues *Formation*. La notation utilisée est celle d'UML 2.0. Le composant multivues *Formation* utilise l'interface multivues *I_Inscription* pour pouvoir traiter les inscriptions des étudiants. Cette interface propose 4 méthodes : *demanderInscription*, *inscrireEtudiant*, *modifierInscription*, et *afficherInscription*. L'accès à chacune de ces méthodes et le comportement associé dépendent de la vue active. A titre d'exemple, la méthode *inscrireEtudiant* ne peut être exécutée que par le responsable du site. L'interface *I_Consultation* est une interface multivues fournie (accessible par tout composant la requérant) pour la consultation des informations sur une formation. Le comportement de la méthode *afficherInfos* change dynamiquement selon la vue active.

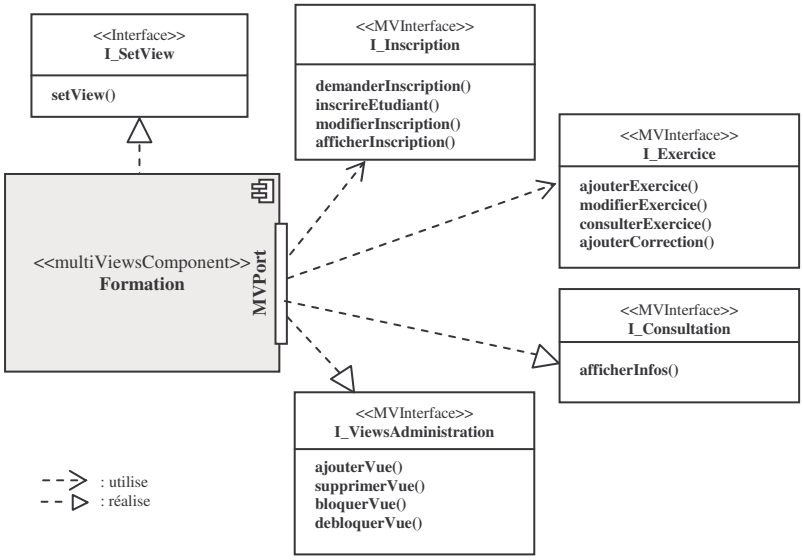

Figure 12. *Représentation explicite d'un sous-ensemble d'interfaces fournies et requises du composant multivues « Formation »*

5. Vers une démarche associée à VUML

Le processus de développement avec VUML n'étant pas l'objectif principal de cet article, nous ne présentons qu'une version simplifiée de la démarche. Nous nous appuyons sur la modélisation de l'étude de cas SED présentée dans la section 2 pour illustrer les phases de la démarche associée à VUML.

Pour obtenir le modèle VUML du SED nous proposons de suivre la démarche schématisée sur la figure 13 ci-dessous. Dans cette section, nous montrons comment obtenir le diagramme de classes VUML ; nous ne présentons pas la phase d'élaboration des composants multivues à partir des classes multivues car ce passage ne présente pas d'intérêt méthodologique particulier.

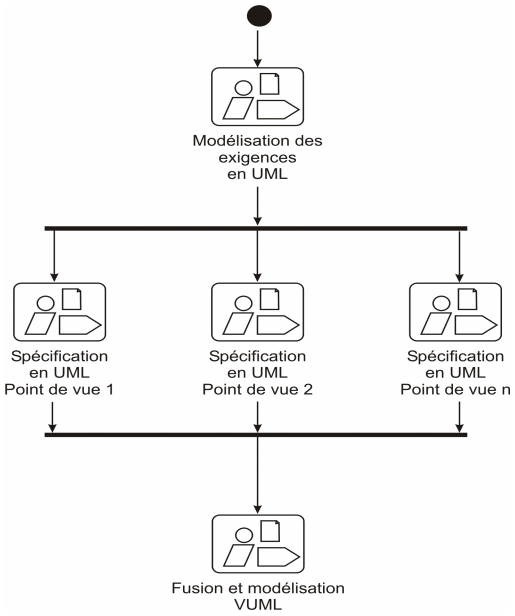

Figure 13. *Démarche d'analyse par point de vue avec VUML*

La modélisation s'effectue en 3 phases : phase centralisée de modélisation des exigences, phase décentralisée de modélisation du système suivant chaque point de vue — en utilisant les différents diagrammes UML —, phase centralisée de fusion et de modélisation produisant le diagramme de classes VUML. Dans un premier temps la modélisation des besoins est principalement réalisée grâce aux cas d'utilisation ; puis, pour chaque acteur identifié, donc pour chaque point de vue, sont spécifiés les scénarios ainsi que les diagrammes de classes associés ; la fusion VUML permet d'identifier les classes multivues et de réaliser un diagramme de classes global permettant de produire le modèle VUML.

5.1. *Modélisation des exigences*

Un résumé de l'analyse des exigences du SED a déjà été présenté dans la section 2. Les acteurs et les cas d'utilisation de ce système sont représentés sur la figure 1. Cette représentation permet de structurer les cas d'utilisation principaux en fonction des buts du système.

5.2. *Spécification par point de vue*

Pour chaque point de vue il est possible de modéliser la dynamique du système en utilisant des diagrammes de séquence ou de collaboration. A titre d'exemple, la figure 14 présente le diagramme de séquence d'un scénario du cas d'utilisation *Gestion des formations*. Les objets identifiés ainsi que les méthodes associées sont reportés dans le diagramme de classes du point de vue *Enseignant* de la figure 15. De plus, ce diagramme fait apparaître des objets et donc d'autres classes identifiées lors des spécifications des autres cas d'utilisation (comme par exemple les classes *Question*, *Reponse* et *Etudiant*).

Une démarche similaire effectuée pour chaque acteur et pour chaque cas d'utilisation est réalisée et donne lieu aux diagrammes de classes des figures 16 et 17. Afin de simplifier ces diagrammes, seuls quelques méthodes et attributs représentatifs sont présentés. On obtient alors un diagramme de classes par point de vue qui montre clairement les informations pertinentes auxquelles peut accéder l'acteur associé. Nous pouvons remarquer par exemple qu'un enseignant (cf. figure 15) doit avoir accès aux informations sur les étudiants inscrits à sa formation, et savoir quel est l'auteur de chaque question reçue. Ce n'est pas le cas de l'étudiant (cf. figure 16) qui peut simplement accéder à la documentation, aux exercices et à l'examen, et poser des questions. Il peut aussi demander de s'inscrire dans une formation. Le responsable de site (cf. figure 17) a besoin quant à lui, de connaître le responsable de la formation, la date de l'examen, et les étudiants inscrits.

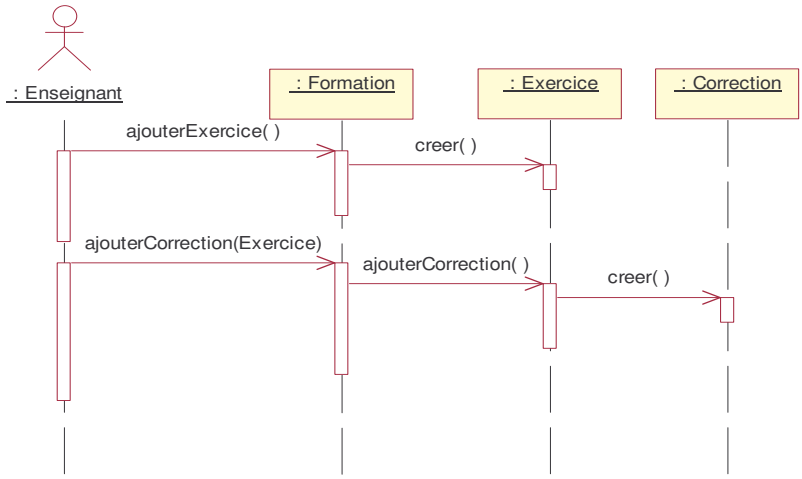

Figure 14. *Exemple de scénario du cas d'utilisation Gestion des formations*

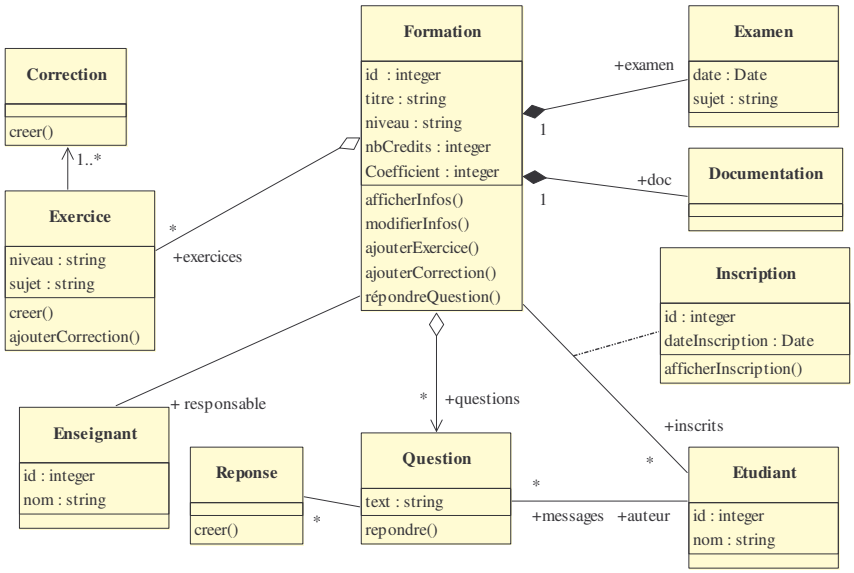

Figure 15. *Diagramme de classes - Point de vue Enseignant*

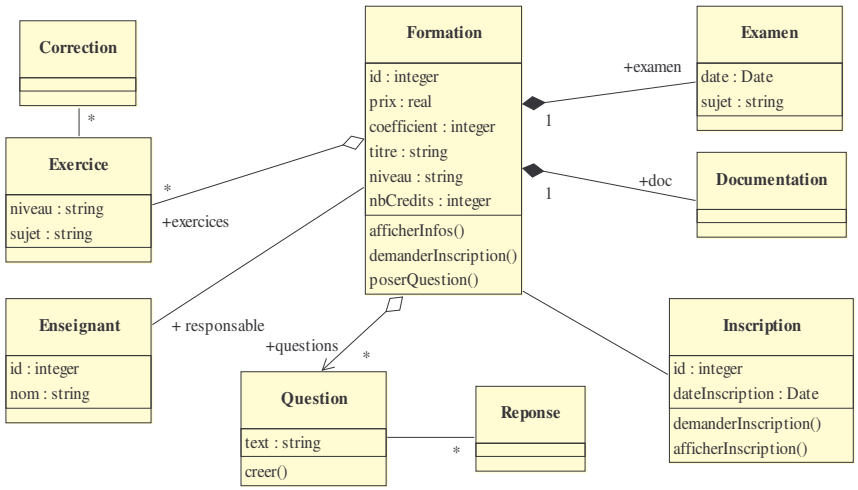

Figure 16. *Diagramme de classes - Point de vue Etudiant*

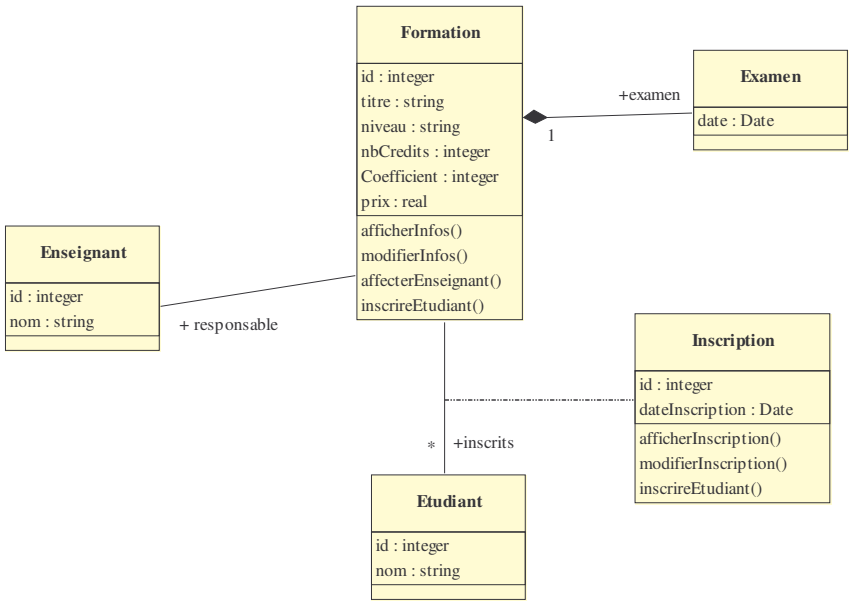

Figure 17. *Diagramme de classes - Point de vue Responsable Site*

5.3. *Fusion et modélisation VUML*

La première étape de la fusion est la comparaison des diagrammes de classes par points de vue qui peut révéler des ambiguïtés ou incohérences sur les classes (nommage, attributs, méthodes) et sur les relations entre classes. Ces problèmes traités, les diagrammes de classes élaborés précédemment peuvent être regroupés ensuite en un seul diagramme. Si l'on faisait cette fusion de manière classique en UML — ce qui est possible — cela reviendrait à perdre les informations liées aux caractéristiques de chaque point de vue : en faisant apparaître simultanément toutes les classes et tous les attributs, on ne spécifierait plus quel acteur peut accéder à ces informations. En revanche, avec la fusion VUML, il est possible de modéliser sur un seul diagramme de classes la totalité des informations présentées dans chaque vue, sans pour autant perdre les spécificités de chaque acteur. Les classes multivues sont des classes apparaissant dans au moins deux diagrammes de classes. Ainsi, *Formation*, *Question*, *Reponse*, *Exercice*, *Correction*, *Examen*, *Documentation*, *Enseignant*, *Inscription*, et *Etudiant* vont donner lieu à des classes multivues. La figure 18 ci-après illustre un diagramme VUML en faisant apparaître les stéréotypes <<*base*>>, <<*view*>>, <<*viewDependency*>> et <<*viewExtension*>> présentés dans la section 4. Afin de simplifier le diagramme seules certaines classes ont été modélisées. Ainsi, les classes multivues qui ne sont pas éclatées en vues sont représentées par le stéréotype <<*multiViewsClass*>>.

La relation de dépendance stéréotypée <<*viewDependency*>> entre les vues de l'*Enseignant* et de l'*Etudiant*, annotée par une contrainte en OCL, exprime le fait que la liste des questions posées par un étudiant relativement à une formation doit être incluse dans la liste des questions posées sur la même formation avec l'ensemble des étudiants inscrits dans cette formation. Les relations *viewDependency* permettent d'exprimer des contraintes sur les associations comme dans notre cas ici mais aussi des contraintes liées aux attributs et aux méthodes. Ces relations seront exploitées lors de la phase d'implémentation pour gérer la cohérence entre les vues.

En poursuivant ce travail de fusion, on obtient un diagramme de classes VUML (cf. figure 18 ci-dessous) où l'on peut, selon le nombre de classes et de vues, présenter les classes multivues soit sous forme éclatée en faisant apparaître chaque vue, soit sous forme "iconifiée" en spécifiant le stéréotype *multiViewsClass*. Cette spécification permet par la suite de développer des composants comme cela a été présenté en section 4 et de les réaliser à l'aide du patron d'implantation décrit dans la prochaine section.

5.4. *Ajout d'un point de vue*

Comme nous l'avons vu dans la section 4, VUML offre la possibilité d'ajouter à tout moment dans le modèle un nouveau point de vue qui donnera lieu à une

nouvelle vue pour une ou plusieurs classes multivues. Cette évolution du modèle ne doit pas remettre en cause le modèle déjà existant. Le mécanisme de spécialisation des vues l'assure. A titre d'exemple, il est possible pour le SED d'effectuer des formations pour des professionnels travaillant en entreprise. Afin de réaliser cette opération, nous avons utilisé le mécanisme de spécialisation pour créer deux nouveaux points de vue : celui du *Professionnel* qui suivra une formation continue et celui de l'*Entreprise* dont le professionnel fait partie (cf. figure 6).

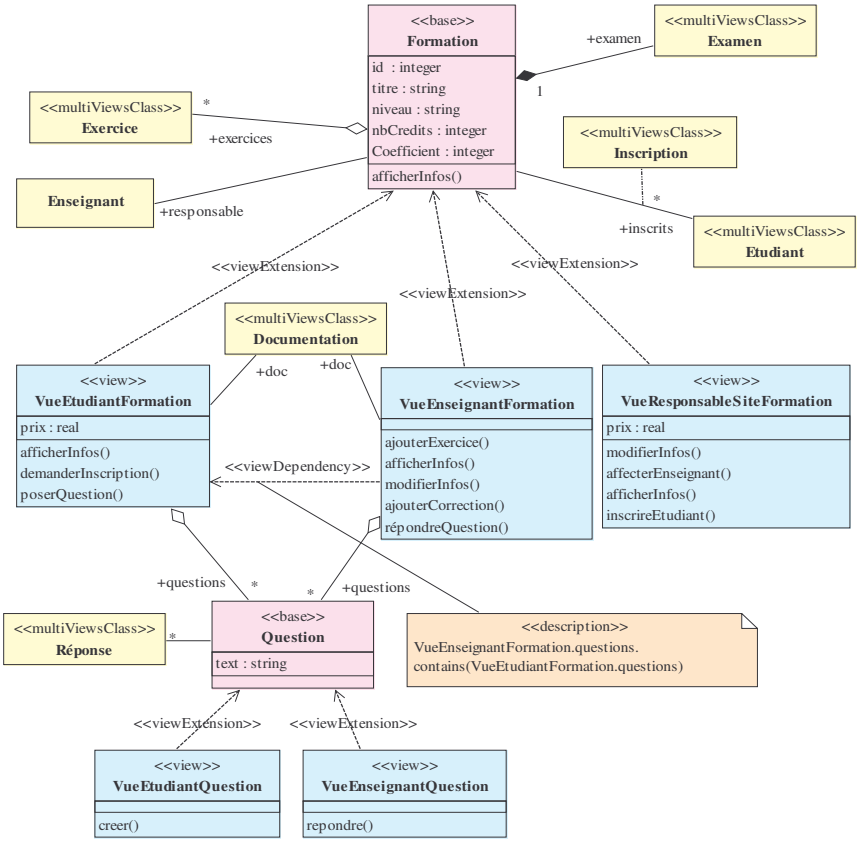

Figure 18. *Diagramme VUML avec les classes multivues Formation et Question*

6. Transition vers la phase de codage

Pour que notre approche soit convaincante et facile à utiliser, il ne faut pas qu'elle pénalise le programmeur objet "classique". C'est ce que nous avons

recherché en ciblant d'une part les langages à objet du marché, et d'autre part en proposant une traduction automatique d'un composant multivues en un modèle d'implémentation. Nous donnons ci-dessous à cet effet le principe d'un patron d'implémentation générique qui permet de produire, à partir d'un composant multivues, un code objet multicibles (Java, C++, Eiffel, …). Ce patron combine le patron *Rôle* pour implémenter la délégation et la technique de la poignée, et le patron *Stratégie* pour l'implémentation des vues avec la surcharge des méthodes (Gamma et al. 1995, Coad 1992). Ce patron est utilisé dans le processus de génération de code relatif à une modélisation VUML. Nous ne donnons que les éléments clés permettant de comprendre ce patron, en laissant de côté l'implémentation de certaines fonctionnalités : propagation des changements de vue (non présentée dans cet article), mise en cohérence des vues, traitement de l'héritage entre classes multivues. Le lecteur pourra trouver une description détaillée de la technique employée dans (Nassar et al., 2002).

La figure 19 ci-après illustre la structure statique du patron proposé. Une classe multivues *CMV* est traduite par l'ensemble de classes suivant : une classe de même nom *CMV* reliée par composition à une classe *Base_CMV* (regroupant les attributs et les méthodes de la base), et à une classe *View_CMV*, et une liste de classes correspondant aux vues. Les vues sont décrites par des sous-classes de *View_CMV*. La classe *CMV* fournit la méthode *setView()* permettant d'activer/désactiver une vue et la méthode *getView*() qui retourne la vue active. La vue active est soit une instance de *View_CMV* (vue par défaut quand aucune vue n'est active), soit une instance de la classe décrivant une vue (*Vuek_CMV*). Les méthodes sont appliquées sur l'instance de la classe *CMV* qui contient toutes les méthodes définies dans la classe multivues. La sélection de la méthode à exécuter s'appuie sur le polymorphisme de *View_CMV*. Considérons une méthode *m(args)* ayant *T* comme type de retour. La définition de la méthode *m()* engendrée dans la classe *CMV* se contente de rediriger l'appel vers la vue courante :

T m(args) {return getView().m(args);}

La définition de la méthode *m()* dans la classe *View_CMV* dépend du lieu de définition de *m()*. Si *m()* est dans la base, l'appel engendré passe par une indirection sur elle :

T m(args) {getCMV().getBase.m(args);}

Si *m()* n'appartient pas à la base, l'exception *AccesInterditException* est levée :

T m(args) {throw new AccesInterditException();}

La classe *Vuek_CMV* est obtenue en dupliquant la vue correspondante *Vuek*. Ainsi, si une méthode *m()* apparaît dans *Vuek*, elle est définie dans *Vuek_CMV* ce qui constitue une redéfinition de la méthode de *View_CMV*.

Ce patron est indépendant d'un langage cible particulier, il a été expérimenté avec Eiffel, puis testé avec Java pour réaliser un prototype du système SED.

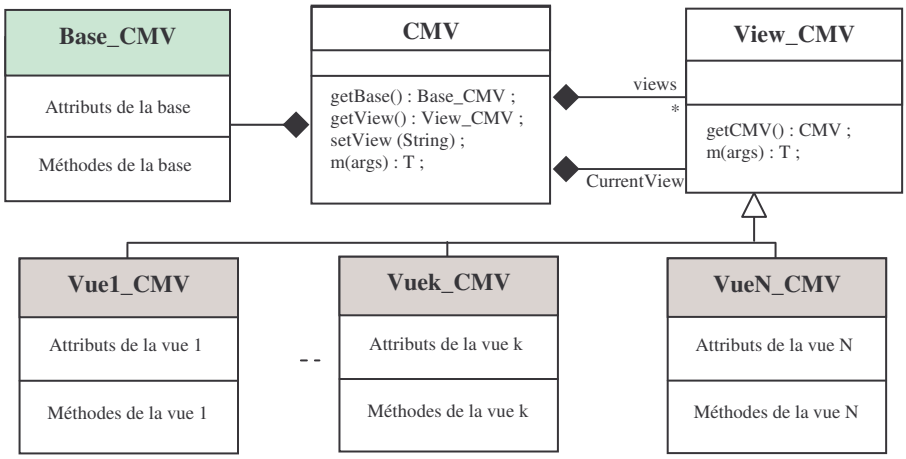

Figure 19. *Patron d'implémentation pour la génération de code multicibles à partir d'un composant multivues*

7. Outil support à VUML

Pour appliquer efficacement l'approche VUML, nous avons réalisé un outil support à VUML. Cet outil a été développé sous l'atelier Objecteering/UML (Objecteering-site, 2004). Le choix de cet atelier comme plate-forme de développement se justifie par le fait qu'il supporte une gestion de profils. En effet, grâce aux profils il est possible d'adapter l'outil *Objecteering/UML Modeler* (qui fait partie de l'atelier *Objecteering/UML*) pour tenir compte des contraintes de modélisation propres à des projets (cf. figure 20). Ceci permet de réduire considérablement le temps nécessaire pour réaliser des outils supportant des extensions d'UML.

Le langage J (Objecteering-site, 2004) est l'un des constituants essentiels de la puissance de *UML Profile Builder*. C'est un langage qui permet de paramétrer et de piloter l'atelier *Objecteering/UML*. J est un langage objet interprété, à la syntaxe Java, dédié à manipuler les modèles. Son code peut ainsi être modifié et testé rapidement. J s'exécute au niveau métamodèle et manipule les éléments de modélisation créés par l'utilisateur.

Figure 20. *Adaptation d'Objecteering/UML Modeler par des Profils UML réalisés sous Objecteering/UML Profile Builder (figure tirée de la documentation d'Objecteering/UML)*

Afin de mette en oeuvre l'outil support à VUML, nous avons développé deux profils Objecteering :

- Le profil Objecteering VUML qui permet de mener une modélisation selon VUML et de vérifier la conformité des diagrammes avec la sémantique de VUML. En fait, la sémantique statique de VUML est définie par le métamodèle, des règles de bonne modélisation (well-formedness rules) exprimées en langage formel OCL (Object Constraint Language) et des descriptions textuelles informelles. Vu que l'atelier Objeceetring ne supporte pas le langage OCL, nous avons traduit les règles de bonne modélisation en langage J.
- Le profil Objecteering « Générateur de code Java » qui permet la génération du code Java à partir d'une modélisation VUML. Cette génération de code s'appuie sur le patron de génération de code présenté dans la section 6.

8. Conclusion

Cet article s'inscrit dans le cadre des travaux de recherche que nous menons sur l'élaboration d'une méthodologie d'analyse/conception par points de vue. Nous avons tout d'abord présenté la notion de classe multivues qui permet d'intégrer les concepts de vue et point de vue dans l'analyse/conception d'un système. Une classe multivues est une entité de modélisation "flexible" qui permet de décrire l'information en fonction des points de vue des acteurs concernés. Chaque classe multivues est statiquement composée d'une base (vue par défaut) et d'un ensemble de vues étendant cette base. Une classe multivues peut être spécialisée en une classe multivues, sa base héritant de celle de la classe parente. Nous préconisons la mise en évidence des dépendances entre les vues d'une classe multivues pendant la phase de conception à travers des déclarations de dépendances (explicitées en OCL).

L'apport de cet article se situe plus précisément dans l'introduction de la notion de composant multivues qui permet de représenter la classe multivues au niveau d'un diagramme de composants. Structurellement, un composant multivues étend le

composant UML 2.0 en proposant des interfaces appelées *interfaces multivues* et une interface simple permettant de positionner la vue active. Il est aussi doté d'une interface multivues destinée à la gestion des vues. L'ensemble des stéréotypes introduits pour élaborer des composants multivues est regroupé dans le profil VUML.

Notre soucis étant de faciliter la transition vers le codage, nous avons aussi défini un patron d'implémentation générique qui décrit la manière de passer d'un composant multivues à du code objet standard (testé avec Java et Eiffel pour l'instant).

Ce travail a été validé par l'intermédiaire d'un outil support à VUML. Il a été implémenté en particularisant l'atelier *Objecteering/UML* grâce à la technique des profils. Cet outil supporte complètement la modélisation basée sur les vues et points de vue. Il offre également deux fonctionnalités supplémentaires : une vérification de la cohérence des modèles VUML qui doivent être conformes à la sémantique VUML ; une génération de code objet à partir d'une modélisation VUML. Cette génération s'appuie sur le patron d'implémentation générique proposé par VUML.

Nous travaillons actuellement à l'affinage et à l'expérimentation sur des exemples significatifs de la démarche esquissée dans la section 5 (en particulier la phase de fusion en VUML de modèles UML développés séparément), et le développement de composants multivues réutilisables. Certains éléments concernant l'aspect dynamique de VUML nécessitent encore des approfondissements. Dans cette optique, nous travaillons actuellement à l'intégration de la notion de point de vue au sein des diagrammes dynamiques (notamment les diagrammes d'activités et les diagrammes Etats-Transition). Concernant l'outil support à VUML, nous sommes en train d'enrichir le générateur de code afin qu'il prenne en compte les dépendances fonctionnelles entre les vues.

Parmi les pistes de travail d'ores et déjà explorées, nous pouvons signaler l'intégration dans VUML d'un modèle de gestion de cohérence entre les vues basé sur un patron. Une autre piste intéressante consiste en l'intégration des travaux menés autour de VUML dans l'approche MDA (Model Driven Architecture). Ceci nous permettra de profiter des avantages de cette architecture : pérennité de la logique métier grâce à l'élaboration de modèles, productivité de cette logique métier grâce à l'automatisation des transformations de modèles, et prise en compte des plates-formes d'exécution grâce à l'intégration de celles-ci dans les transformations de modèles.

Remerciements :

Cet article présente des résultats de recherches menées au sein du réseau franco-marocain STIC en Génie Logiciel lancé en 2002. Tous les partenaires du réseau sont vivement remerciés pour leurs remarques et suggestions qui nous ont permis d'affiner et de valider notre approche.

9. Bibliographie

Abiteboul S., Bonner A.: « Objects and Views », *Proceedings of ACM SIGMOD,* may 1991, p. 238-247.

Andersen E. P., Reenskaug : « System Design by Composing Structures of Interacting Objects». *Proc. of the 6th European Conference on Object-Oriented Programming (ECOOP'92)*, LNCS, Vol. 615. p. 133-152, Utrecht, The Netherlands. Springer-Verlag, 1992

Carré B., Geib J.M. : « The Point of View Notion for Multiple Inheritance », *Proc. of ECOOP/OOPSLA*, 1991.

Charrel P.J. : « An Information System Designing as a Persistent Multi-Viewpoints Requirements Elicitation Process », *6th Workshop on Organisational semiotics*, Reading University, Royaume Uni, 11-12 juillet 2003 . Kluwer Academic Publishers, 2004.

Clarke S., Harrison W., Ossher H., Tarr P. : « Separating Concerns throughout the Development Lifecycle». *Proc. Of ECOOP'99 Workshop on Aspect-Oriented Programming*, Lisbon, Portugal, 1999.

Coad P. : « Object-oriented Patterns ». *Communications of the ACM*, 1992.

Coulette B., Kriouile A., Marcaillou S. : « L'approche par points de vue dans le développement orienté objet des systèmes complexes », *Revue l'Objet*, vol. 2, n°4, février 1996, p. 13-20.

Coulette B. et al. *Réseau STIC franco-marocain en Génie Logiciel*, 2002. http://www.univ-tlse2.fr/grimm/isycom/reseauSTIC/reseauSTIC.html

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.

El Asri B., Nassar M., Coulette B., Kriouile A. : « Assemblage de composants multivues par contrats », *Actes du XXIII*^{ème} *Congrès INFORSID (INFORSID'2005)*, Grenoble, France, 24-27 mai, 2005. p. 29-44.

Finkelstein A., Kramer J., Goedicke M. : « Viewpoint Oriented Software Development », *Proc. of Software Engineering and Applications Conference*, Toulouse, décembre 1990, p. 337-351.

Gamma E. et al. : « Design Patterns, Elements of reusable Object-oriented Software », Addison-Wesley, 1995.

Harrison W, Ossher H. : « Subject-oriented programming : a critique of pure objects», *Proceedings of OOPSLA'93*, Washington D.C., Sept. 26-Oct 1, 1993, p. 411-428.

Hilliard R. : « EEE1471-std-2000 : Recommended Practice for Architectural Description for Software-Intensive Systems», November 2000. http://www.enterprise-architecture.info/Images/Documents/IEEE%201471-2000.pdf

Kiczales G., Lampng J., Mendhekar A., Maeda C., Lopes C. V. : « Aspect-Oriented Programming». *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*. Finland. Springer-Verlag LNCS 1241, 1997.

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.

Kristensen B. B., and Osterbye K. : « Roles : Conceptual Abstraction Theory & Practical Language Issues». *Theory and Practice of Object Systems (TAPOS)*, 143-160, Special Issue on Subjectivity in Object-Oriented Systems, 1996.

Le Moigne J.L. : « La modélisation des systèmes complexes» . Dunod, 1990.

Marcaillou-Ebersold S. : « Intégration de la notion de points de vue dans la modélisation par objets – Le langage VBOOL », *Thèse de l'Université Paul Sabatier de Toulouse*, 1995.

Marcaillou S., Coulette B., Kriouile A. : « Visibility : A new relationship for complex system modelling », *International conference TOOLS USA'94*, Santa Barbara, August 1-5 1994, Prentice Hall.

Medvidovic N., Taylor R. N.: « A Classification and Comparison Framework for Software Architecture Description Languages », *IEEE Transactions on oftware Engineering*, Vol 26, no1, Janvier 2000, p. 70-93.

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. *VUML :* « une extension UML orientée point de vue ». *Thèse de doctorat en Informatique*, Rabat : ENSIAS, Université Mohammed V-Souissi. 2004.

Nassar M., Coulette B., Kriouile A.: « Vers un langage de modélisation unifié supportant les vues», *rapport IRIT 02-29-R*, Toulouse, octobre 2002.

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.

Objecteering-site. http://www.objecteering.com, 2004.

OMG, UML 2.0 Superstructure Final Adopted specification, Document - ptc/03-08-02, 2003, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02

OMG - site, Object Management Group, 2004, http://www.omg.org

Peltier M., « Transformation entre un profil UML et un métamodèle MOF », *Revue l'Objet*, vol. 8, n°1-2/2002, LMO'2002, p. 25-40.

Shaw M., Garlan D., « Software Architecture - Perspectives on an Emerging Discipline », Prentice Hall, 1996.

Shilling J., Sweeny, P.: « Three Steps to Views ». *In OOPSLA'89*. New Orleans, LA, 1989, p. 353-361.

Softeam : «Profiles UML et langage J : Contrôlez totalement le développement d'applications avec UML», *White Paper*, 1999.

Szyperski C.*, Component Software*, Addison-Wesley, 1999.

UML, Unified Modeling Language : «UML Specification version 1.5», OMG documentformal/03-03-01, 2001 http://www.omg.org.