# Transformations de modèles UML Outillées : Retour d'expériences

**Samir Ammour[*,**], Xavier Blanc[*,**], Mikal Ziane[*]**
[*]LIP6 - Laboratoire d'Informatique de Paris 6
75015 Paris, France
{xavier.blanc, mikal.ziane}@lip6.fr
[**]SOFTEAM
75004, Paris France
{samir.ammour}@softeam.fr

**Résumé :** Dans cet article nous montrons comment le mécanisme de la collaboration paramétrée d'UML 2.0 peut être utilisé pour la spécification visuelle des transformations de modèles UML. Nous présentons les avantages de cette nouvelle technique d'être visuelle et déclarative par rapport aux techniques classiques de spécification de transformations basées sur les langages propriétaires, ainsi qu'au futur standard Q/V/T[1]. Une illustration est faite à travers la réalisation d'un exemple de transformation de models UML vers la plate-forme J2EE. Nous présentons aussi les limitations de la nouvelle technique d'être restreinte aux modèles UML. Notre implémentation est réalisée dans le contexte de l'atelier UML Objecteering.
**Mots Clés :** MDA[2], les transformations de modèles, la collaboration paramétrée, Q/V/T, UML.

## 1 INTRODUCTION

Les modèles sont devenus de plus en plus présents dans le processus de développement logiciel et l'adoption de la l'approche MDA par l'OMG en est l'illustration la plus flagrante (OMG 2003). L'approche MDA implique une manipulation intensive et automatique des modèles. L'idée principale est d'élaborer des modèles indépendants des plates-formes d'exécution et de les transformer en modèles dépendants des plates-formes afin de faciliter la génération de code. La place des transformations de modèles est donc centrale et stratégique.
L'élaboration des transformations de modèles dans les ateliers UML nécessitait, il y a encore quelques années, de connaître les langages de manipulation de modèles proposés par ces ateliers. Il fallait alors, par exemple, connaître soit le langage de scripts VB pour Rational XDE (Rational 2004), soit le langage J pour Objecteering (Softeam 1999; Objecteering 2004) soit enfin le langage Java pour Together (Together 2004). L'écriture des transformations était donc un processus très lourd comme cela a été notamment mentionné dans les conclusions du projet Accord (Accord 2001; Blanc, Caron et al. 2004).
Conscient des difficultés inhérentes à la prise en main et à l'utilisation de ces langages de manipulation de modèles, les éditeurs ont alors tous apportés la même évolution qui est de proposer des langages graphiques pour exprimer les transformations sans avoir à écrire de ligne de code. L'objectif est de faciliter l'élaboration et la maintenance des règles de transformation. Softeam a fait le choix d'utiliser notre approche basée sur les *templates* UML pour élaborer de manière visuelle les transformations de modèles. Ce choix a l'avantage d'être entièrement basé sur des éléments de modèles UML. Lorsque le formalisme graphique choisi est suffisant pour exprimer la transformation, il n'est donc plus nécessaire d'utiliser de langage propriétaire.
De son côté, l'OMG est en train de définir un nouveau standard, nommé Q/V/T, dédié uniquement aux transformations de modèles. Les travaux sur le standard Q/V/T ont commencé à l'OMG en 2002 (OMG 2003). A l'époque huit propositions de solution avaient été soumises. Aujourd'hui, il ne reste que deux propositions candidates. Après environ deux ans de travaux, les concepts de ce futur standard commencent donc à se stabiliser et on sait maintenant qu'une transformation Q/V/T pourra s'écrire soit d'une manière déclarative avec un ensemble de règles de correspondance soit d'une manière impérative avec une succession de règles de construction ou les deux en même temps.
L'approche qu'adoptent les éditeurs d'outils UML semble donc être en léger décalage avec ce que sera le futur standard Q/V/T. Ce décalage soulève inévitablement des interrogations quant à la compatibilité des deux approches. Nous proposons dans cet article de présenter l'approche qu'a choisi Softeam pour élaborer des transformations de modèles et de discuter de cette approche face au futur standard Q/V/T.

[1] Query, View, Transformation (Q/V/T): le futur standard de l'OMG (*Object Management Group*) pour la spécification des transformations de modèles.
[2] Model Driven Architecture (MDA) : une nouvelle démarche et technologie en cour de standardisation à l'OMG, qui favorise l'utilisation des modèles dans le développement logiciel.

L'objectif n'est pas de faire une analyse comparative des deux approches. Cela ne peut pas réellement être fait étant donnée que ces deux approches sont encore en cours d'élaboration. L'idée est plutôt de comparer la philosophie des deux approches afin de voir si elles sont compatibles ou pas.

Pour pouvoir étudier les deux approches, nous avons choisi de les étudier sur un même exemple. L'exemple que nous avons choisi est une partie de la transformation UML vers EJB. Elle est implémentée dans la plupart des ateliers UML en utilisant les langages de transformation propriétaires. Cette transformation utilise le profile EJB (OMG 2004) qui définit un ensemble de stéréotypes et des annotations permettant d'adapter UML au domaine des EJB.

Dans cette étude, nous parlerons uniquement des stéréotypes relatifs au *Entity Bean*. Ces *Beans*, dans la technologie EJB, ont pour vocation d'être persistants. Ils sont donc sauvegardés dans des bases de données. Un *Entity Bean* est composé de trois entités : la classe d'implémentation, l'interface *Home* et l'interface *Remote*. La classe d'implémentation contient tous les attributs persistants et la logique métier du *Bean*. L'interface *Home* permet de créer des instances du *Bean* ou de rechercher des instances déjà créées. L'interface *Remote* représente les opérations qui peuvent être appelées à distance sur le *Bean*.

Dans le profile UML pour EJB, les stéréotypes « EJBImplementation » « EJBEntityHomeInterface» et « EJBEntityRemoteInterface », appliqués sur les classes UML, désignent respectivement les classes d'implémentation, les interfaces *Home* et les interfaces *Remote* d'un *Entity Bean*. Le stéréotype « EJBCmpField», appliqué aux attributs UML, désigne un attribut persistant du *Bean*. Le stéréotype « EJBPrimaryKeyField », appliqué lui aussi sur les attributs UML, désigne la clé d'un *Bean*. La clé est un attribut persistent qui permet de retrouver un *Bean* après la création de celui-ci. A une clé doit correspondre une classe appelée classe de la clé. Pour finir, les stéréotypes « EJBCreateMethod », « EJBFinderMethod » et « EJBRemoteMethod », appliqués aux opérations, désignent respectivement les opérations de création et les opérations de recherche dans les interfaces *Home* et les opérations distantes dans les interfaces *Remote*. Appliqués sur un modèle UML, ces stéréotypes permettent donc de spécifier un ensemble d'*Entity Bean*.

La transformation UML vers EJB ne se fait pas en une seule passe. En effet, une étape intermédiaire permet de renseigner le modèle UML avec un ensemble de stéréotypes afin préciser quelles sont les classes du modèle devant être transformées en *Entity Bean*. On appelle « modèle intermédiaire » (Accord 2001) le modèle UML stéréotypé ainsi. La transformation que nous étudions est donc précisément la transformation qui prend en entrée une classe UML du modèle intermédiaire et qui génère le modèle EJB correspondant.

La TABLE 1 présente les stéréotypes du modèle intermédiaire et les règles de transformation associées. L'exemple affiché dans la FIGURE 1 illustre l'application des règles de la transformation « Classe UML vers *Entity Bean* » telles quelles sont implémentées dans l'atelier Objecteering (Objecteering 2004). Dans la partie gauche on trouve la classe Client qui représente la classe d'entrée de la transformation. Elle est composée de trois attributs (Num, Nom et Adresse) et deux opérations (*setNom()* et *setAdresse()*). Le modèle résultat affiché dans la partie droite, est composé de quatre classes : Client_Bean, Client_Home, Client_Object, Client_PK (*Primary Key*) avec l'ensemble des méthodes et attributs d'un *EJB Entity*. Ces classes héritent ou implémentent respectivement les interfaces *EntityBean*, *EJBHome*, *EJBObject* et *Serializable* de Java.

| Méta-Classe | Stéréotype | Transformation |
|---|---|---|
| Class | EJBEntity | Création de la classe d'implémentation du *Bean*.<br>Création de l'interface *Home* du *Bean*.<br>Création de l'interface *Remote* du *Bean*.<br>Sur le stéréotype *EJBEntity* une annotation (*tagged-value*) nommée *EJBSessionType* est définie. Elle prend la valeur *StateFull* ou *StateLess* ce qui permet de déterminer si l'*EJB Entity* est persistant ou pas. |
| Attribut | EJBCmpField | L'attribut est considéré comme étant persistant et sera copié dans la classe d'implémentation du *Bean*. |
| Attribut | EJBPrimaryKeyField | L'attribut sera copié dans la classe d'implémentation du *Bean* et sera considéré comme étant une clé.<br>Une classe clé sera donc créée à partir de l'attribut. |
| Operation | - | L'opération est copiée dans la classe d'implémentation du *Bean*. |
| Operation | EJBRemoteMethod | L'opération sera copiée dans la classe d'implémentation du *Bean* et publiée dans classe *remote* comme une méthode distante. |
| Operation | EJBCreateMethod | L'opération sera copiée dans la classe d'implémentation du *bean* sous le nom « ejbCreate » et publiée dans la classe home comme méthode de création. |
| Operation | EJBFinderMethod | L'opération sera copiée dans l'interface *home* comme méthode de recherche. |

**Table 1:** Les règles de la transformation « Classe UML vers *Entity Bean* ».

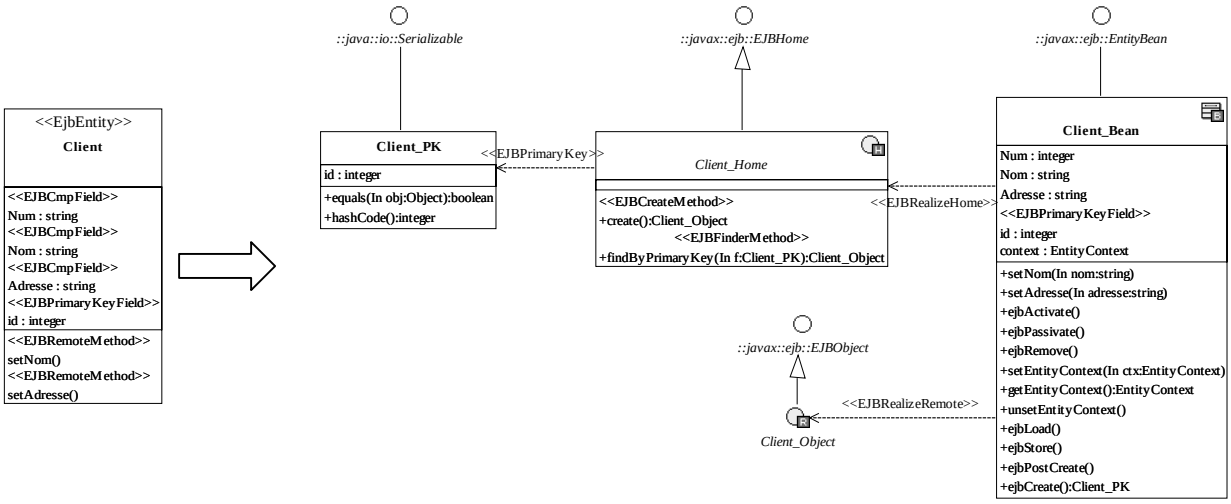


**Figure 1 :** Exemple d'application de la transformation "Classe UML vers *Entity Bean*".

Dans la suite de cet article nous présenterons donc l'approche SOFTEAM et nous l'illustrerons sur cet exemple. La section 2 est donc entièrement dédiée à cette approche qui étend le concept de la collaboration paramétrée UML. La section 3, présente les principes de base de Q/V/T et présente comment peut s'exprimer cette transformation exemple quand on utilise la facette impérative du standard. L'objectif étant de bien comprendre les différences conceptuelles entre les deux approches. La section 4 présente nos conclusions quant au positionnement des deux approches.

## 2 LA NOTION DE LA COLLABORATION PARAMETREE

### 2.1 Définition

Dans UML2.0 (OMG 2003), une collaboration permet de décrire la structure d'éléments (*classifiers*[3] ou leurs instances) en collaboration. Chacun des éléments de la collaboration a une fonction spéciale à accomplir pour réaliser le but global de cette collaboration. Une collaboration permet donc de spécifier les caractéristiques que doivent respecter les éléments prenant part à la collaboration comme les liens entre les classes ou les valeurs de leurs attributs ; mais aussi, il est possible de spécifier des contraintes sur les classes dont les objets sont des instances.

La collaboration paramétrée est une collaboration dont un ensemble de ses éléments sont définis comme des paramètres. Les collaborations paramétrées sont utilisées pour définir la structure des patrons de conception dans UML. En effet, la partie solution d'un patron est assimilable à une interaction entre plusieurs objets (c'est-à-dire à une collaboration).

La possibilité de paramétrer un élément UML est régit par un mécanisme général de généricité appelé « Template ». La **Figure 2** regroupe les méta-classes du méta-modèle UML2.0 qui représentent la notion de *template*. Les éléments qui sont des sous classes de la méta-classe abstraite *TemplateableElement* peuvent être paramétrés. Comme exemple de ces éléments on trouve les *classifiers* (collaborations et classes), packages et opérations.

Paramétrer un élément consiste à identifier l'ensemble des ses paramètres formels (*TemplateParameter*) ce qui représente la signature du *template* (*TemplateSignature*). On ne peut définir comme paramètre que les éléments dits paramétrables c'est-à-dire qui héritent de la méta-classe abstraite *ParameterableElement*. Comme exemple de ses éléments on trouve les opérations, les attributs et les *classifiers*.

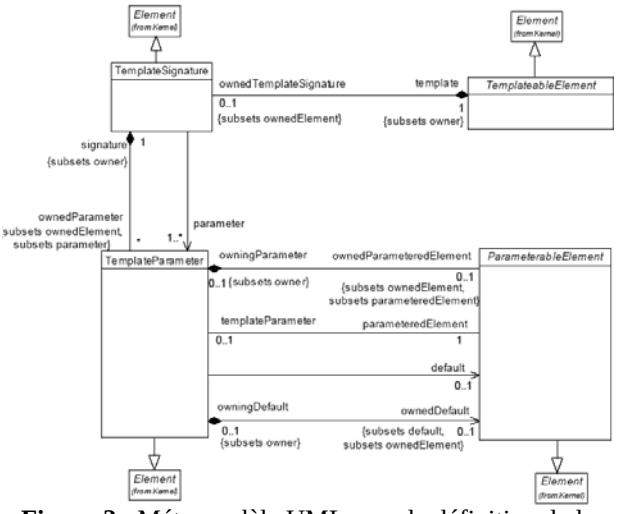


**Figure 2** : Méta-modèle UML pour la définition de la notion de *template*

[3] Un *classifier* est un élément permettant de classifier un ensemble d'objets. La classe est le *classifier* le plus connu mais ce n'est pas le seul ; la collaboration est aussi un *classifier* car elle permet de classifier les objets participant à la collaboration.

Un *template* dans UML est utilisé pour générer d'autres éléments identiques du même type que le *template* grâce à l'utilisation de la relation « TemplateBinding ». L'application des patrons de conception est l'exemple le plus connu de l'utilisation de la notion de *template* lorsqu'elle est appliquée aux collaborations.
Comme illustré dans la Figure 3, un *template binding* est une relation dirigée de l'élément à créer (*boundElement*) vers l'élément *template*. Chaque *template binding* contient un ensemble de clauses de substitution de paramètres (*TemplateParameterSubstitution*). Une clause de substitution permet d'associer à un paramètre formel du *template* un ou plusieurs éléments de substitution.

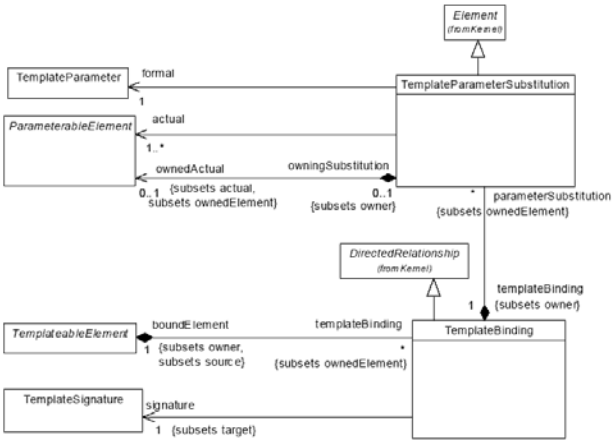


**Figure 3 :** Méta-modèle UML pour l'utilisation des templates

Dans (Ammour, Blanc et al. 2004), nous avons présenté les avantages de la collaboration paramétrée pour le support des patrons dans les outils UML (et plus particulièrement dans Objecteering). L'intérêt le plus important de cette approche est que la spécification et l'application des patrons se fait intégralement en utilisant UML (nul besoin d'un langage propriétaire de manipulation de modèles). L'inconvénient majeur de cette approche réside dans la faiblesse d'expression des *templates bindings*. Tout d'abord, la sémantique des *templates bindings* est spécifiée en langage naturel et est sujette à plusieurs interprétations différentes. En cela, le travail de (Caron, Carré et al. 2004) est très intéressant car il formalisme cette sémantique en utilisant le langage OCL. De plus, les *templates bindings* sont parfois trop contraignants. Par exemple, si un *template* a un de ses paramètres qui est une interface, alors il n'est pas possible d'associer à ce paramètre une classe comme valeur car seul les éléments de même type (même méta-classe) peuvent être associés. Parfois par contre, les binding ne sont pas assez contraignants. Par exemple, si un *template* a un de ses paramètres qui est une opération, alors il n'est pas possible de restreindre l'association de ce paramètre avec des opérations ayant au moins le même type de retour. Nous avons donc proposé dans (Ammour, Blanc et al. 2004) des extensions à la collaboration paramétrée afin d'améliorer la spécification des patrons. Les extensions proposées avaient pour objectifs de mieux préciser les *templates bindings* en ajoutant un support des contraintes OCL.
Toujours dans (Ammour, Blanc et al. 2004) nous avons aussi proposé d'étendre la sémantique du *template binding* pour permettre de faire des modifications sur les modèles sur lesquels s'appliquent les *templates*. En effet, lors de l'application classique d'un *template*, seuls des ajouts peuvent être fait dans le modèle sur lequel s'applique le *template*. Nous avons donc étendu la relation du *template binding* en y ajoutant le concept d'action permettant ainsi de supprimer ou de modifier certaines parties du modèle.
Ces extensions, tant sur la spécification des collaborations paramétrées que leurs applications, ont été entièrement développées dans l'atelier Objecteering et ont été testées sur tous les patrons du GOF (Gamma, Helm et al. 1995). Suite à cette expérience, nous avons songé à utiliser cette approche pour élaborer d'autres patrons de transformations de modèles fréquemment utiles.
Par exemple, nous avons utilisé cette approche pour spécifier et exécuter la transformation qui vise à construire un ensemble d'accesseurs à partir des attributs d'une classe. La **Figure 4** (partie haute) représente une collaboration paramétrée dont l'application génère les accesseurs des attributs d'une classe. La collaboration paramétrée contient deux paramètres : la classe « Class » et l'attribut « Attribute ». Nous avons utilisé les caractères "<%" et "%>" pour définir les actions de modification des éléments du modèle (dans l'exemple, uniquement des créations d'opération avec substitution des noms). Cette syntaxe est présentée plus précisément dans (Ammour, Blanc et al. 2004)

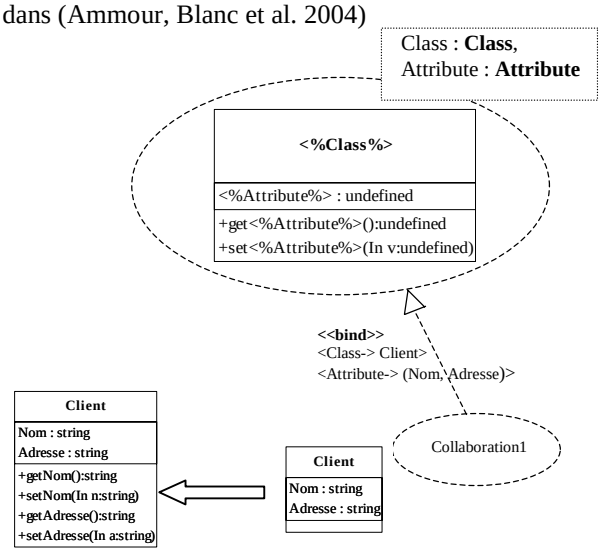


**Figure 4** : Spécification d'une transformation de génération des accèsseurs à l'aide d'une collaboration paramétrée

La partie basse de la Figure 4 représente l'application de la collaboration paramétrée. L'application est représentée par un lien « bind » entre la collaboration définie dans le contexte de la classe « Client » nommé Collaboration1 et la collaboration paramétrée. Le lien « bind » spécifie les valeurs associées aux paramètres « Class » et « Attribute » (ici « Client » pour le

paramètre « Class » et « Nom » et « Adresse » pour le paramètre « Attribute »). Le résultat de l'application est affiché en bas à gauche de la figure. On voit que les méthodes *getNom(), setNom(), getAdresse()* et *setAdresse()* ont été générées.

Ces travaux nous laissent à penser qu'il est possible d'utiliser le mécanisme de la collaboration paramétrée pour élaborer des transformations de modèles UML. L'idée est de représenter la structure du modèle résultat de la transformation à l'aide d'une collaboration paramétrée.

L'élaboration d'une transformation avec les collaborations paramétrées nécessite alors de réaliser :

- la modélisation de la collaboration en y spécifiant les opérations de modification (correspond à définir le résultat attendu de la transformation).
- l'identification les paramètres de la collaboration pour en faire une collaboration paramétrée en y spécifiant les contraintes de *binding* (correspond à définir les paramètres d'entrée de la transformation).

## 2.2 Expression de la transformation « Classe UML vers *Entity Bean* » à l'aide d'une collaboration paramétrée

Les règles de la transformation « Classe UML vers *Entity Bean* » ont déjà été présentée (voir TABLE 1). Nous allons présenter dans cette partie la spécification de cette transformation en utilisant une collaboration paramétrée.

### *2.2.1 Elaboration de la structure des éléments de la collaboration*

La structure des éléments de la collaboration correspond à la structure du modèle attendue après l'exécution de la transformation.

Dans notre exemple, le modèle final doit contenir la classe d'implémentation du *Bean* ainsi que son interface Remote et son interface *Home* et la classe correspondant à la clé du *Bean*. Nous choisissons de nommer ces classes par les noms <%Class%>_Bean, <%Class%>_Home, <%Class%>_Object et <%Class%>_PK (<%Class%> correspond au nom de la classe sur laquelle sera exécuter la transformation).

Ces classes doivent contenir les attributs et les opérations identifiés par les règles de la TABLE 1. Par exemple, si le modèle d'entrée contient une opération stéréotypée « EJBRemoteMethod » alors cette opération devra se retrouver dans la classe d'implémentation du *Bean* et dans son interface *Remote*. Il nous faut donc spécifier dans la collaboration tous les attributs et les opérations possibles. Ceux-ci sont nommés de manière à pouvoir facilement savoir à quoi ils correspondent (par exemple, l'opération nommée <%OperationRemote%> correspond au résultat obtenu après transformation d'une classe contenant une opération stéréotypée « EJBRemoteMethod »).

Pour finir, la collaboration doit spécifier tous les liens entre les classes et les interfaces. Ces liens sont dictés par les règles EJB. Par exemple, une classe d'implémentation d'un *Bean* doit hériter le l'interface *javax.ejb.EntityBean.*

La **Figure 5** représente la structure de la collaboration correspondant à la transformation « classe UML vers *Entity Bean* ».

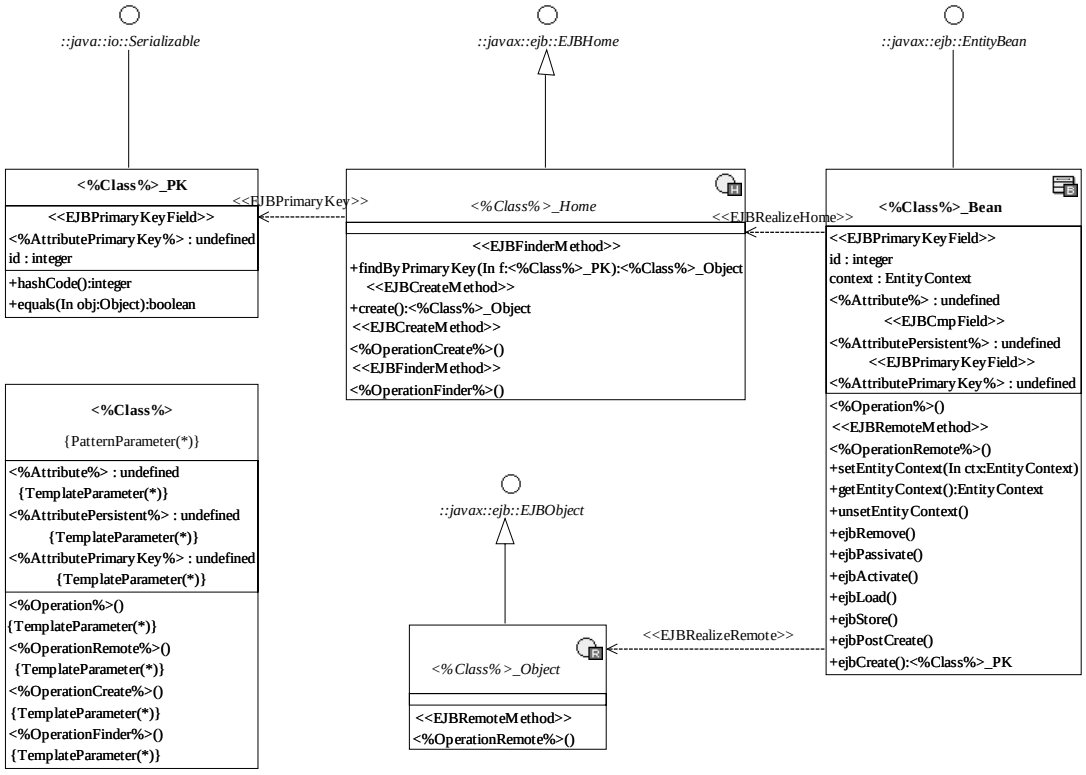


**Figure 5** : La collaboration paramétrée de spécification de la transformation « Classe UML vers *Entity Bean*»

### *2.2.2. Définition des paramètres de la collaboration*

Les paramètres de la collaboration correspondent aux différents éléments que peut contenir le modèle d'entrée. Etant donnée que la transformation accepte en entrée une classe UML, il faut alors qu'un paramètre de la collaboration correspond à cette classe. Les autres paramètres permettent d'identifier les différentes propriétés que peut contenir la classe (attribut persistant, clé, opérations divers, opération distante, opération de création, opération de recherche)

La Figure 6 représente une vue macro de la collaboration « Class2EntityBean» avec ses huit paramètres.

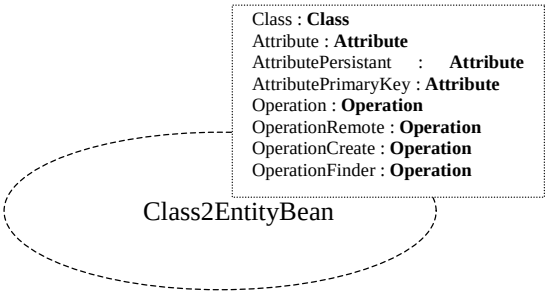


**Figure 6 :** Les paramètres de la collaboration paramétrée "Class2EntityBean"

Afin de contraindre l'application de la collaboration (et donc de la transformation), il est nécessaire de définir quelques contraintes en OCL afin d'assurer une application correcte de la transformation. Ces contraintes, appliquées sur le modèle source, seront évaluées avant l'exécution de la transformation.

Nous dressons ici une liste de contraintes définies sur les paramètres de la collaboration Class2EntityBean :

[1] La transformation ne doit s'appliquer que sur une classe stéréotypée « EJBEntity » et persistante.

```
context <%Class%>
        inv: (self.constraint.stereotype.name =
        "EJBEntity" and
        self.taggedValue.type.tagType
        ="EJBEntityType" and
        self.taggedValue.dataValue =
        "SatateFull")
```

[2] Le paramètre OperationRemote ne peut être substitué que par des opérations stéréotypées « EJBRemoteMethod » :

```
context <%OperationRemote%>
        inv: (self.constraint.stereotype.name =
        "EJBRemoteMethod")
```

[3] Le paramètre OperationCreate ne peut être substitué que par des opérations stéréotypées « EJBCreateMethod » :

```
context <%OperationCreate%>
        inv: (self.constraint.stereotype.name =
        "EJBCreateMethod")
```

[4] Le paramètre Operation ne peut être substitué que par des opérations qui n'ont pas de stéréotype:

```
context <%Operation%>
        inv: (self.constraint.stereotype.name <>
        "EJBRemoteMethod" and
        self.constraint.stereotype.name <>
        "EJBCreateMethod" and
        self.constraint.stereotype.name <>
        "EJBFinderMethod").
```

# 3 LE LANGAGE Q/V/T

## 3.1 Définition

La Figure 7 représente une partie du futur méta-modèle Q/V/T dans laquelle nous avons représenté les méta-classes de Q/V/T nécessaires à la compréhension de la philosophie de ce standard.

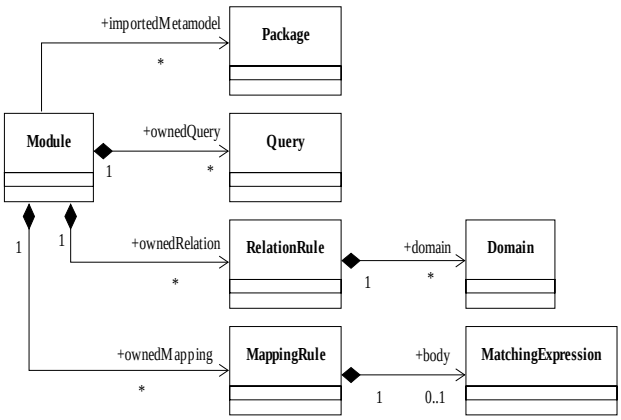


**Figure 7** : Le futur méta-modèle Q/V/T

La méta-classe Module représente une transformation. Cette méta-classe est reliée à la méta-classe Package. Ce lien permet de spécifier sur quels méta-modèles porte la transformation (méta-modèle source et cible). Cette méta-classe est aussi reliée, par des relations d'agrégation, aux méta-classes *Query*, *RelationRule* et *MappingRule*. Ces liens permettent de spécifier l'ensemble des requêtes, règles de correspondance et règles de construction qui composent une transformation.

La méta-classe *Query* représente les requêtes effectuées dans une transformation. Nous n'avons pas représenté ici le détail de cette méta-classe. Les requêtes Q/V/T sont principalement des expressions OCL permettant de naviguer dans un modèle.

La méta-classe *RelationRule* représente des règles de correspondance entre des sous-parties des méta-modèles source et cible. Cette méta-classe est reliée à la méta-classe *Domain* qui représente une sous-partie d'un méta-modèle (aussi appelé patron dans le vocabulaire Q/V/T). Ces règles de correspondance ne définissent pas de construction d'éléments de modèle. Elles sont donc similaires à des règles de programmation déclarative.

La méta-classe *MappingRule* représente quant à elle des règles de construction. Cette méta-classe est reliée à la méta-classe *MatchingExpression* qui représente le concept d'action de construction. Ces règles sont

similaires à des instructions de programmation impérative.
Pour résumer, on peu donc dire qu'une transformation Q/V/T s'écrit dans un module. Ce module peut contenir des requêtes écrites dans un langage proche de OCL. Ce module peut aussi contenir des règles de transformation qui sont soit des règles de correspondance (approche déclarative) soit des règles de construction (approche impérative). Notons que ce standard laisse possible la construction d'une transformation par une approche hybride, c'est-à-dire qui contient des règles de correspondance et des règles de construction.
Associé à ce méta-modèle, le standard Q/V/T propose une syntaxe textuelle concrète. Nous présenterons brièvement cette syntaxe en présentant la réalisation de notre transformation exemple.

## 3.2 Expression de la transformation « Classe UML vers Entity Bean » en Q/V/T[4]

Nous présentons ici quelques lignes de la transformation « Classe UML vers Entity Bean » écrite dans le langage Q/V/T.

```
module EjbEntityToBeanImpl(in ejbModel:UML):
implModel:UML;

main() {
ejbModel->objectsOfType(Class)->
     transformEjbEntity();
}
mapping transformEjbEntity [in Class]()
  : impl:Class, home:Interface,
  remote:Interface, primary:Class
  merges transformPersistentAttribute,
transformCmpField, ...
{
  out impl: Class {
    stereotype :=
self.retrieveStereotype("Bean");
    name := self.name + "_Bean";
    implements :=
self.retrieveFromLib("::Javac::ejb::EntityBean"
);
  }
  out home: Interface {
    stereotype :=
self.retrieveStereotype("Home");
    name := self.name + "_Home";
    generalization := out Generalization {
      superClass :=
self.retrieveFromLib("::Javac::ejb::EJBHome");
  };
}
...
```

Ces quelques lignes contiennent :

- la définition du module qui précise que la transformation transforme un modèle UML (nommé ejbModel) vers un autre modèle UML (nommé implModel).
- la définition du *main* (première règle appelée) qui cherche dans le modèle d'entrée toutes les classes puis elle lance la règle nommée « transformEjbEntity » sur chacune d'entre elles.
- la définition partielle de la règle de construction nommée « transformEjbEntity » qui permet ici de construire l'ensemble des produits, notamment la classe d'implémentation du *Bean* et la fabrique *Home*.

La règle principale "transformEjbEntity" déclare l'ensemble des règles à "fusionner". Cette découpe permet de retrouver facilement les règles de transformation telles qu'elles ont été fournies informellement (voir le TABLE 1 dans 1.1).
La transformation complète qui fait environ 150 lignes contient toutes les autres règles de construction permettant de construire le modèle EJB complet.

# 4 CONCLUSION

Dans ce document nous avons présenté l'approche SOFTEAM pour les transformations de modèles qui est basée sur l'utilisation d'un concept assez connu qui est le modèle de la collaboration UML. L'approche a été outillée dans le cadre des travaux sur le support des patrons de conception dans l'atelier UML Objecteering (Ammour, Blanc et al. 2004).
Les collaborations paramétrées sont bien adaptées à la définition de transformations simples de modèles UML vers UML, telles que le passage UML vers EJB. Elle évite l'écriture de "code" comme cela est le cas pour l'approche impérative de Q/V/T de l'OMG ou encore les approches classiques à base de langages de transformations propriétaires. Cette approche offre l'avantage d'être visuelle et déclarative. La spécification des transformations nécessite de maîtriser la technique des collaborations paramétrées mais évite l'apprentissage d'un langage de transformation particulier. Elle se fait tous simplement par la modélisation de la structure d'une collaboration en UML et la définition de ses paramètres en terme de contraintes et requêtes de substitution. À terme, on envisage de réaliser des assistants interactifs pour la construction des contraintes de paramètres et les requêtes de substitution en OCL.
Un autre avantage pour cette approche est quelle supporte les transformations de modèles profilés d'une manière assez simple et transparente. Les transformations de profiles, que ce soit d'un modèle standard vers un profile particulier ou d'un profile vers un autre sont spécifiées de la même façon que les transformations de modèles UML génériques.
Il est important de noter que cette approche est plus simple car elle est beaucoup moins généraliste que l'approche Q/V/T. En effet, contrairement à Q/V/T, cette approche est restreinte aux transformations de modèles UML. Il n'est pas possible de transformer d'autres modèles conformes à d'autres méta-modèles. Cependant, nous pensons qu'il est envisageable de générer du code Q/V/T à partir des collaborations paramétrées. L'approche par collaboration paramétrée peut donc être considérée comme un moyen visuel et ergonomique d'élaborer certaines classes de transformations de modèle (c'est-à-dire, les transformations de modèles prenant en entrée et en sortie les modèles UML et ne

[4] L'élaboration de cette transformation a été réalisée grâce à la participation de Mariano Belaunde.

présentant pas une trop grande complexité). On peut anticiper, que, à grande échelle, la programmation purement "visuelle" de transformations se heurtera aux même difficultés que l'on rencontre quand on tente de représenter graphiquement des comportements complexes de type algorithmique. La possibilité de concilier modélisation graphique et notation textuelle est donc une piste prometteuse pour faciliter l'écriture des transformations de modèles.

## 5 BIBLIOGRAPHIE


Accord (2001). Accord Project Web Site. http://www.telecom.gouv.fr/rntl/AAP2001/Fiches_Resume/ACCORD.htm.

Ammour, S., X. Blanc, et al. (2004). Imporving Patterns Support in UML CASE Tools. Workshop UML 2004, Lisbonne.

Blanc, X., O. Caron, et al. (2004). Transformation de modèles: d'un modèle abstrait aux modèles CCM et EJB. LMO, Lille.

Caron, O., B. Carré, et al. (2004). An OCL Formulation of UML2 Template Binding. UML 2004, LISBONNE.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.

Objecteering (2004). Objecteering Web Site, www.Objecteering.com.

OMG (2003). Final Adopted Specification; "OMG Unified Modeling Language Superstructure Specification". Version 2.0.

OMG (2003). MDA Guide Version 1.0.1 OMG document omg/2003-06-01.

OMG (2003). MOF 2.0 Query/View/Transformation.

OMG (2004). "Metamodel and UML Profile for Java and EJB Version 1.0."

Rational (2004). Rational XDE Web Site, Rational Software Corporation http://www.rational.com/products/xde/index.jsp .

Softeam (1999). UML Profiles and the J language: Totally control your application development using UML.

Together (2004). Borland Together Web Site http://www.borland.com/together/.