UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

MÉTAMODÉLISATION ET TRANSFORMATION AUTOMATIQUE DE PSM DANS UNE APPROCHE MDA

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

PAR

JAMAL ABD-ALI

MAI 2006

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS
Département d'informatique et d'ingénierie

Ce mémoire intitulé :

MÉTAMODÉLISATION ET TRANSFORMATION AUTOMATIQUE DE PSM DANS UNE APPROCHE MDA

présenté par
Jamal Abd-Ali

pour l'obtention du grade de maître ès sciences (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Dr Karim El Guemhioui ……..…..... Directeur de recherche

Dr Luigi Logrippo …….…….... Président du jury

Dr Michal Iglewski …… Membre du jury

Mémoire accepté le : 4 mai 2006

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations, sigles et acronymes

MDA	*Model Driven Architecture* [18] est une méthode de développement de logiciel définie par le consortium *Object Management Group (OMG).*
MOF	*Meta Object Facility* (MOF) [22] est normalisé par l'OMG dans sa version courante 2.0. C'est un le langage utilisé pour définir des langages d'écriture de modèles comme UML et *Common Warehouse Metamodel* (CWM) .
OMG	Organisme de normalisation dans le domaine de l'orienté objet.
PIM	Un *Platform Independent Model* désigne un modèle qui décrit un système à un niveau d'abstraction lui permettant une indépendance par rapport à la technologie d'implémentation.
PSM	Un *Platform Specific Model* désigne un modèle qui décrit un système à un niveau d'abstraction lui permettant de prendre en compte une technologie d'implémentation
QVT	*Query/Views and Tranformations* est une spécification en cours d'approbation pour standardiser un langage de définition de transformations de modèles et de spécifications des vues et requêtes sur les modèles basés sur MOF.
ATL	*Atlas Transformation Language* [33] est un langage de transformation de modèle selon l'approche MDA. Il a été développé principalement par un groupe de chercheurs de l'université de Nantes en France.
XMI	XML *Metadata Interchange Format* (XMI) [25] définit un mécanisme pour la correspondance entre les métamodèles basés sur MOF et le XML et schémas. Il permet aussi la représentation des modèles basés sur MOF dans des fichiers au format XML se conformant aux schémas correspondant à leur métamodèle.

Résumé

Dans cette dernière décennie, la technologie des composants a connu une expansion rapide, initialement avec (Entreprise JavaBeans (EJB), et plus récemment avec la technologie des composants .NET. Beaucoup de compagnies n'hésiteraient probablement pas à embrasser la nouvelle vague technologique des composants .NET si elles pouvaient récupérer (une partie de) leur investissement dans EJB.

Comme il est très probable que les applications existantes, implémentées selon EJB, n'ont pas de modèles indépendants de la plate-forme technologique (PIM), nous proposons un chemin horizontal de migration entre ces deux technologies par la définition d'une transformation qui convertit un modèle spécifique aux composants EJB (PSM) en un modèle spécifique aux composants .Net. Puisque les métamodèles de ces deux technologies sont essentiels à la définition de notre transformation, nous employons le métamodèle d'EJB adopté par l'OMG et, pour les composants .NET, nous proposons un métamodèle qui capture les concepts technologiques de cette technologie et leur structure. L'expression de la transformation est faite dans un langage bien défini dérivé d'une soumission en réponse au *request for proposal* (RFP) publié par l'OMG pour normaliser le langage QVT de définition de transformations de modèles. La praticabilité de l'idée est illustrée par un cas d'étude.

Abstract

In this last decade, component technology has witnessed a fast expansion, initially with Enterprise JavaBeans (EJB), and more recently with .NET Components. Many companies would most likely embrace the new wave of the .NET component technology if they could recover (part of) their investments in EJB.

As it is very probable that the existing applications implemented according to EJB do not have models independent of the technological platform (PIM), we propose a horizontal path of migration between these two technologies by defining a transformation which converts a model specific to EJB components (PSM) into a model specific to the .Net components. Since the metamodels of these two technologies are essential for the definition of our transformation, we use the EJB metamodel adopted by the OMG and, for .NET components, we propose a metamodel which captures the technological concepts of this latter technology and their structure. The transformation is expressed in a well formed language derived from a submission in response to the request for proposal (RFP) published by the OMG to standardize a language for models transformations definition named QVT. The feasibility of the idea is illustrated with a case study.

Chapitre 1

Identification du problème et motivation

1.1. Contexte et problématique

L'industrie du génie logiciel a permis le développement de systèmes informatiques de plus en plus complexes. Mais la pérennité de ces systèmes qui représentent des investissements énormes, est remise en question chaque fois qu'une nouvelle technologie émerge. Cette situation s'est accentuée avec les systèmes à base de composants et roulant sur des middlewares.

Les technologies à base de composants permettent des traitements distribués et profitent de services disponibles au moment de l'exécution (*runtime*) et offerts par les middlewares sous-jacents. Ces technologies sont de plus en plus utilisées pour construire les applications des entreprises. Les services disponibles à l'exécution (*runtime*) sont essentiels pour de telles applications car ils répondent à des besoins réels comme la gestion de la sécurité, le comportement transactionnel, l'interaction avec d'autres composants et ressources de l'environnement d'exécution, la gestion des instances d'un composant, etc.

Les développeurs comptent donc sur le middleware sur lequel roule leur application pour avoir accès aux services requis. En effet, développer ces services spécifiquement pour une application serait très fastidieux et onéreux. En remarquant que les spécifications fonctionnelles et non fonctionnelles des applications réparties modernes dépendent en grande partie de ces services, on comprend que le développeur soit toujours à la recherche de la technologie de composants et du middleware qui fournissent les services les plus adaptés à ses besoins.

Par ailleurs, chaque nouvelle technologie est généralement accompagnée d'un ensemble d'outils qui ne supportent plus les anciennes technologies. De plus, on constate que la nouveauté technologique exerce un attrait important sur les utilisateurs. Tous ces facteurs font que la migration vers une nouvelle technologie est devenue une réalité incontournable et récurrente. Cette migration implique un effort de développement ciblant une technologie différente; cet effort comprend une répétition partielle du travail fait lors du développement de l'application pour l'ancienne technologie. Il en résulte un coût de migration assez élevé vu la complexité des technologies impliquées et des middlewares sous-jacents.

En réponse à cette situation problématique, on est amené à explorer les possibilités de réduire ce coût de migration en minimisant l'effort de développement répété; cet effort concerne le travail de développement indépendant de la technologie visée et qui est relié à la logique métier de l'application.

1.2. Contexte de travail

Notre point de départ est de profiter de l'application existante pour minimiser le coût de la migration vers une autre technologie. On dispose idéalement des éléments suivants : les spécifications informelles de l'application, les modèles de conception de cette application et le code exécutable.

Ces trois éléments représentent différents niveaux d'abstraction de la même application. Les modèles semblent être les éléments les plus exploitables à des fins de migration car : (1) par rapport au texte informel des spécifications, ils ont l'avantage d'être généralement susceptibles d'un traitement automatique, et (2) par rapport au code exécutable, ils sont mieux placés car ils utilisent généralement des artefacts pour représenter la logique métier, différents de ceux utilisés pour la représentation des concepts technologiques d'implémentation. Les modèles sont plus prometteurs au niveau de la séparation de problèmes; ils permettent de faire abstraction des détails qui sont généralement étroitement liés aux technologies d'implémentation.

Notre cadre de travail sera donc celui du développement de logiciels dirigé par les modèles, connu aussi sous le nom d'ingénierie de développement dirigée par les modèles (IDM) ou Model Driven Engineering (MDE). Un cas particulier de l'IDM est le Model Driven Architecture (MDA) [18] qui sera détaillé dans ce mémoire.

En réponse à la problématique déjà exposée et à d'autres situations d'insatisfactions reliées au cycle de vie d'un logiciel (productivité, maintenance, etc.), le consortium Object Management Group (OMG) a défini l'architecture Model Driven Architecture (MDA) [18] qui constitue un cadre ambitieux de développement et qui adopte une séparation entre la logique métier et son implantation technologique. L'approche MDA vise à préserver la partie de l'investissement dépensée pour la mise en œuvre du modèle métier (modèle de la logique d'affaire).

MDA propose de décrire un système par un modèle indépendant de toute plate-forme technologique d'implémentation. Ce modèle est appelé *Platform Independent Model* (PIM). Le PIM subit ensuite une transformation pour produire le modèle spécifique à la plate-forme technologique choisie, appelé *Platform Specific Model* (PSM). Ce dernier décrit le système à un niveau d'abstraction plus détaillé en utilisant les notions et structures spécifiques à la technologie cible.

En remarquant que les différentes catégories de modèles expriment des concepts appartenant à différents niveaux d'abstraction et différentes technologies (domaines), on se rend compte du fait qu'on aura besoin de plusieurs langages de modélisation. Chaque langage de modélisation d'un domaine doit permettre une représentation des concepts de ce domaine avec un ensemble de règles (grammaire) reflétant la structure reliant ces concepts. Un tel langage de modélisation est lui-même défini par un modèle appelé le métamodèle. De même, pour modéliser un métamodèle on a besoin d'un langage d'expression dont le modèle est appelé méta-métamodèle. MDA définit un méta-métamodèle normalisé, le Meta Object Facility (MOF), qui servira de langage d'écriture de tout autre métamodèle. Le fameux Unified Modeling Language (UML) est écrit en MOF et ce couple MOF/UML forme les deux standards de base pour la modélisation dans le cadre de MDA. Une explication plus détaillée des langages de modélisation et de leurs niveaux d'abstraction est présentée dans le chapitre 2.

L'opération clé de l'approche MDA est l'élaboration de plusieurs transformations de modèles permettant le passage d'un niveau d'abstraction à un autre (PIM-PSM, PSM-PSM, PSM-code). Chaque transformation de modèle est définie par un ensemble de règles appelées les règles de la transformation.

Une transformation d'un modèle de départ M1 vers un modèle cible M2 doit être écrite dans un langage bien défini pour permettre son automatisation. D'où le rôle que jouent des outils supportant un ou plusieurs langages de transformation de modèles.

Un outil de génération de code reçoit ensuite le modèle spécifique à la plate-forme technologique PSM pour générer le code approprié.

La figure 1.1 illustre l'architecture MDA en considérant, pour des raisons de simplicité, seulement deux plates-formes technologiques choisies.

À noter que MDA n'est pas encore finalisé, et qu'une grande partie du travail des vendeurs de solutions MDA et des membres de OMG, est concentrée sur l'élaboration et la normalisation d'un langage de définition de transformation de modèles, ainsi que sur la normalisation de profils de modélisations des plates-formes technologiques. Parallèlement, un autre volet de l'effort est orienté vers la définition des transformations et le développement des outils automatisant ces dernières, et ce pour les diverses plates-formes technologiques actuelles.

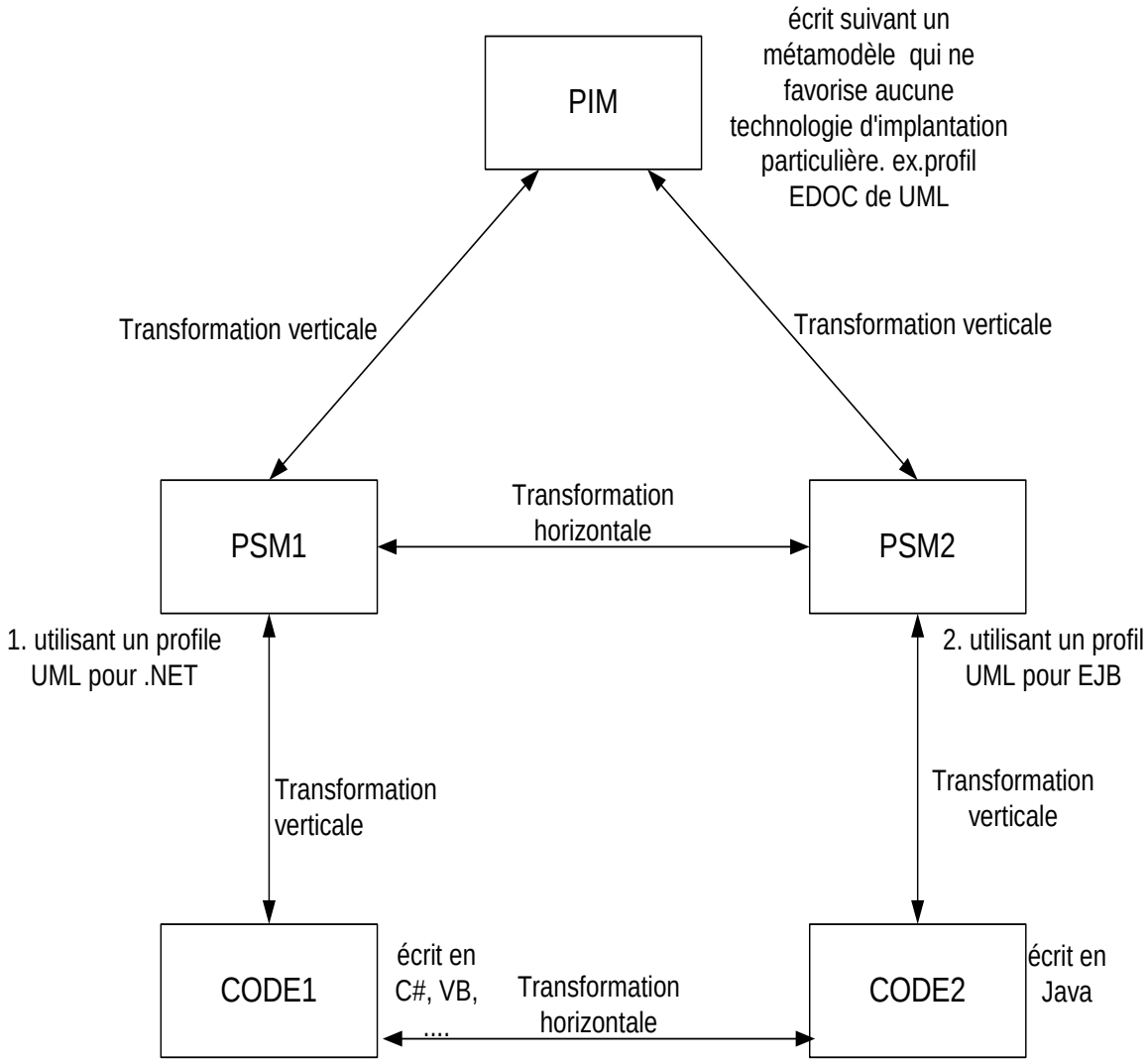


Figure 1.1 - Cycle de vie de MDA

Ayant survolé le cycle de vie de MDA, on peut mieux se situer par rapport à l'IDM. Tout d'abord il importe de signaler qu'en plus d'être un cas particulier de l'IDM, le cadre de travail de MDA jouit des atouts suivants :

- Promotion par l'OMG dont les standards sont largement répandus.
- Existence de langages de modélisation normalisés par l'OMG comme l'UML.
- Normalisation d'un langage d'écriture de langages de modélisation appelé MOF comme un élément de base d'une pile de couches de langages reflétant différents niveaux d'abstraction en modélisation. Cette structure de couches de modélisation constitue un concept commun à tout IDM.
- Existence d'un ensemble d'outils supportant les langages normalisés par l'OMG, en plus des autres langages utilisés en IDM comme XML et WSDL.

Notons que MDA, tout comme l'approche plus générale IDM, considère les modèles comme des entités de première classe. En fait la généralité de l'IDM est principalement due au fait que cette approche ne se limite pas aux standards MOF/UML, contrairement à MDA. IDM se caractérise aussi par un support plus accentué du tissage (Model Weaving) d'exigences non comportementales (tolérance aux fautes, performance, etc.)

1.3. Perspective de solution

On anticipe une longue démarche avant l'émergence d'un cadre de travail stable permettant une migration entre technologies à coût raisonnable. Ceci nous incite à prendre une initiative qui aide à résoudre ou au moins à atténuer, les problèmes de telles migrations tout en restant dans le cadre de MDA. Mais une initiative de notre part ne peut pas couvrir le problème en considérant toutes les technologies existantes; notre contribution considère la migration entre la technologie EJB et celle des composants .NET, qui figurent parmi les technologies les plus répandues de nos jours.

Concrètement, notre effort sera concentré sur l'élaboration d'un métamodèle pour les composants .NET et sur la définition de règles de transformation permettant la conversion automatique de modèles EJB en modèles de composants .NET.

Chapitre 2

État de l’art

2.1. MDA cycle de vie et cadre

Au cœur de l’approche MDA se trouve la création de divers types de modèles d’un même système. Dans cette optique, Le développement du logiciel est guidé par l’activité de modélisation du système, suivant trois étapes schématisées par la figure ci-dessous tirée de [12].

1. Premièrement il faut élaborer un modèle à un niveau élevé d’abstraction, indépendant de toute technologie d’implémentation. Ce modèle est appelé *Platform Independent Model* (PIM).
2. Ensuite, il faut transformer le PIM en un ou plusieurs modèles tels que chacun représente le système en incorporant des éléments spécifiques à une certaine technologie d’implémentation. Un tel modèle est appelé *Platform Specific Model* (PSM).
3. Enfin on génère le code exécutable correspondant à chaque PSM.

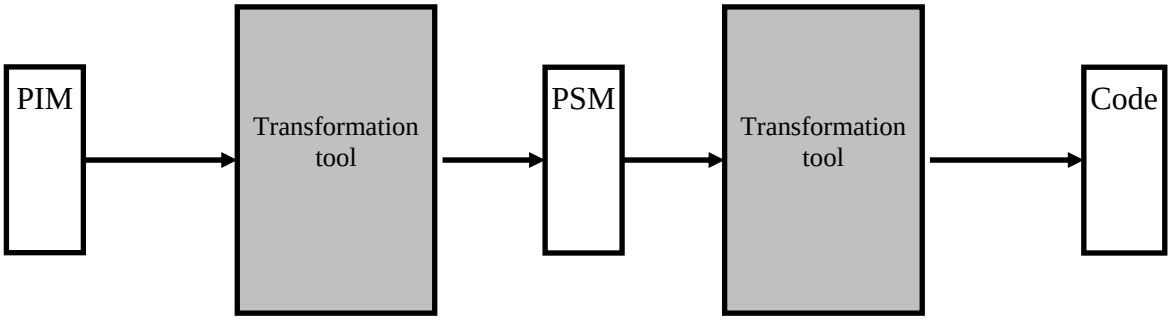


Figure 2.1 - Les étapes principales de MDA

Pour aller plus loin dans l'exploration de l'architecture MDA il importe d'en expliquer la terminologie [12] :

- Langage bien défini : c'est un langage qui a une forme (syntaxe) et une signification (sémantique) susceptibles d'être interprétées automatiquement par ordinateur.
- Un modèle est une description d'un système écrit dans un langage bien défini.
- Une transformation est la génération automatique d'un modèle cible à partir d'un modèle source selon une définition de la transformation formé d'un ensemble de règles applicables aux modèles.
- Une règle de transformation décrit comment un ou plusieurs éléments dans un modèle source peuvent être transformés en un ou plusieurs éléments dans un modèle cible.
- La définition de la transformation est formée par l'ensemble des règles de la transformation.
- Un outil de transformation est un logiciel capable de produire le modèle cible à partir du modèle source selon une définition de transformation.

2.2. Les caractéristiques des transformations

La transformation de modèles est le processus qui fait convertir un modèle en un autre [18]. Dans une approche MDA une transformation est conçue pour être automatisée. Elle est formée d'un ensemble de règles et doit être écrite dans un langage bien défini qu'un outil pourra compiler et exécuter. Cette automatisation sous-entend que les modèles doivent être écrits dans un langage traitable par les machines. La figure 2.2 tiré de [12] illustre ceci tout en utilisant la notion de langage d'écriture de modèle qu'on examinera dans la section suivante.

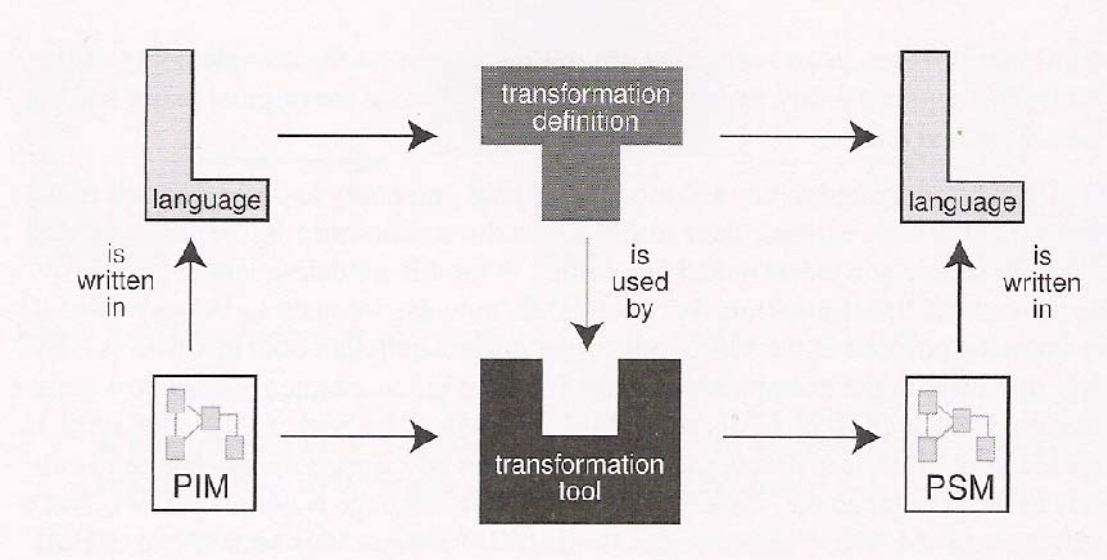


Figure 2.2 - Automatisation des transformations

Une transformation se caractérise par la manière dont ses règles s'appliquent, d'où le besoin de distinguer les propriétés suivantes qui varient selon la définition de la transformation et le langage utilisé [5], [12] :

- L'ordre d'application sur les fragments de modèle correspondant à une règle;
- L'ordre dans lequel s'appliquent les différentes règles;
- La possibilité de composition de règles;
- La relation entre les modèles source et cible (même, différent);
- La traçabilité (définie ci-après);
- La bidirectionnalité;
- L'interactivité permettant la paramétrisation.

Nous allons examiner plus en détail l'ingrédient essentiel de toute transformation, à savoir : la règle. Elle est généralement une expression de correspondances entre un fragment du modèle source et un fragment du modèle cible; et elle se caractérise par :

- L'usage de variables, leur type et leur visibilité;
- La manière de spécifier un fragment source ou cible par des patterns : graphiquement ou par du texte;
- La logique d'exécution : impérative ou déclarative;
- Bidirectionnalité, traçabilité et paramétrisation au niveau de la règle.

Dans la suite de cette section on va éclaircir rapidement quelques notions importantes citées ci-dessus.

2.2.1. Utilisation de paramètres

Il n'y a pas de façon unique pour effectuer une transformation de modèles. Pour spécifier nos choix concernant une transformation, on peut avoir recours à l'utilisation de paramètres dont l'utilisateur spécifie les valeurs. Ces choix peuvent aider à compléter un modèle, ils servent aussi à optimiser le modèle cible produit par l'outil.

Exemple : Dans une transformation Java-SQL, le choix de la longueur 30 du VARCHAR(30) en SQL est amplement suffisant pour remplacer un String de Java conçu pour stocker le nom d'un client.

2.2.2. La traçabilité

Pour tout élément généré dans le modèle cible, la traçabilité permet de retrouver l'élément générateur dans le modèle source.

2.2.3. La cohérence incrémentielle

L'ajout manuel d'information au modèle cible est conservé lors d'une génération ultérieure du modèle cible.

2.2.4. La bidirectionnalité

Elle consiste en la possibilité de retrouver le modèle source par une transformation inverse. Malheureusement ceci est généralement inaccessible, car en passant à un niveau d'abstraction moins élevé, on ajoute des détails moyennant des choix pris par défaut ou même des choix de valeurs de paramètres influençant la transformation.

Notons que la bidirectionalité est garantie, dans le cas où la transformation serait égale à son inverse, ou encore si les deux transformations sont spécifiées de façon à être l'une inverse de l'autre [12].

2.3. Métamodélisation

Un modèle représente un système réel en se basant sur la sémantique et les règles qui conditionnent ses éléments; en d'autres termes il ne doit en aucun cas briser la structure ou les contraintes que les éléments du système réel respectent.
En conséquence, les éléments du langage d'expression d'un modèle doivent satisfaire un ensemble de règles qui leur permet de former un modèle qu'on appelle métamodèle. Le langage est alors dit un langage bien défini et on peut ne plus faire la distinction entre le métamodèle et le langage qu'il définit (figure 2.3 tirée de [12]).

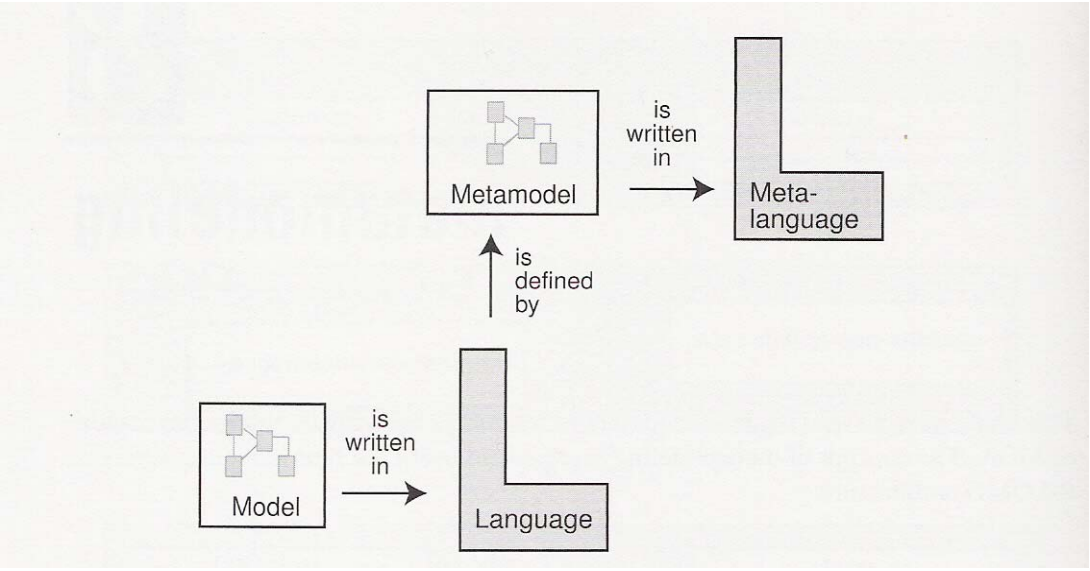


Figure 2.3 - Modèles, langages, métamodèles et métalangages

De même, un métamodèle est écrit dans un langage appelé métalangage et il est instance d'un méta métamodèle qui définit les éléments du métalangage.

On peut même dire qu'un langage est écrit (défini) par un métalangage. Le métalangage doit être écrit encore par un méta-métalangage et ainsi de suite. Pour bien délimiter notre étude des niveaux de modèles, on distingue selon la normalisation de l'OMG quatre couches de modélisation :

M0 c'est la première couche correspondant à un système en exécution (*running*). M0 contient des instances d'objets en cours de traitement par le logiciel.

M1 c'est la couche du modèle exécutable renfermant la structure et le comportement du système. C'est à ce niveau que se situent généralement les modèles que nous manipulons quotidiennement dans nos activités de développement de logiciel (ex. : modèle UML d'un service de gestion de stock, modèle ER d'une bibliothèque, etc.).

M2 c'est la troisième couche de modèles appelés métamodèles, et dont les instances sont des modèles de la couche M1. Le métamodèle du langage UML en est un exemple.
Remarquons qu'une classe X d'un modèle de niveau M1 écrit en UML n'est autre qu'une instance du type Class du modèle UML.

M3 c'est un niveau d'abstraction encore plus élevé, où l'instance d'un modèle de cette couche donne un modèle de la couche M2. Un modèle de M3 donnera une syntaxe d'écriture de métamodèles.

OMG arrête cette suite d'abstraction au niveau 4 en définissant les éléments de M3 comme instances de concepts de la même couche M3. C'est à dire que M3 est définie d'une façon auto-descriptive.

La figure 2.4 illustre les 4 niveaux de modélisation dans un contexte de bases de données relationnelles.

Notons qu'on a limité l'abstraction à quatre niveaux, car ceci répond aux besoins d'écriture et de manipulation de modèles dans le cadre de travail visés par OMG. En effet, l'approche MDA traite des modèles de la couche M1 qui seront validés par rapport à des métamodèles appartenant à M2. Cette approche induit aussi des transformations de modèles qui seront exprimées au niveau M3 car elles s'expriment par des correspondances, entre des fragments des métamodèles ayant servi à l'expression des modèles source et cible.

D'ailleurs, le rôle essentiel des métamodèles pour l'expression des règles de transformation de modèles est discuté plus en détail dans la section 2.5. La figure 2.5 schématise un contexte où les modèles source et cible sont écrits dans différents langages (différents métamodèles) mais avec un formalisme unique au niveau du métalangage (méta-métamodèle unique).

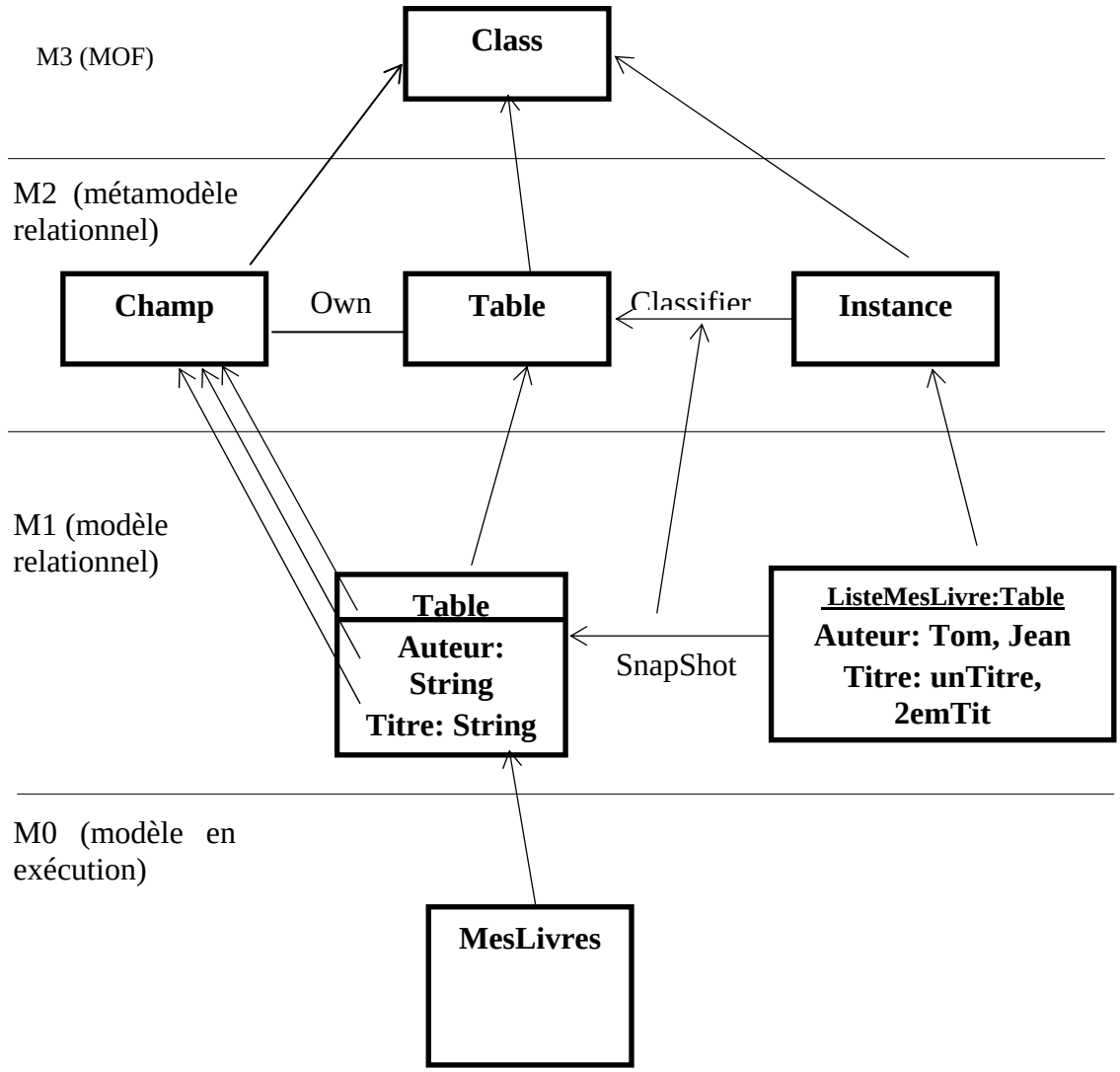


Figure 2.4 - Exemple sur les quatre couches de modélisation

On remarque ainsi qu'un métalangage normalisé peut servir de vocabulaire commun pour manipuler des modèles écrits dans divers langages (divers métamodèles). Ceci permet de créer des outils ayant cette propriété intéressante dans l'environnement pratique qui réserve à chaque domaine informatique un métamodèle spécifique.

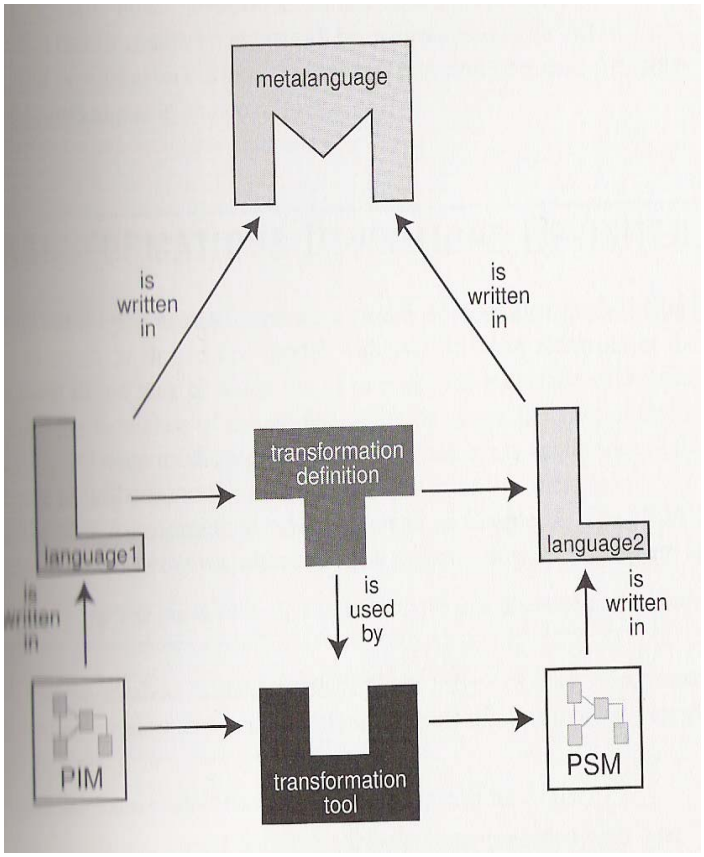


Figure 2.5 - Le métalangage dans le cadre MDA

En effet l'OMG a créé les spécifications de MOF [14] qui est un métalangage normalisé et qui permet de décrire toute une gamme de métamodèles d'usage courant. La figure 2.6 tirée des spécifications de MOF, illustre comment MOF forme un élément commun de spécification de divers métamodèles normalisés par OMG, comme UML et IDL. Les figures ci-dessous illustrent cette hiérarchie à travers un exemple relatif au domaine des bases de données relationnelles.

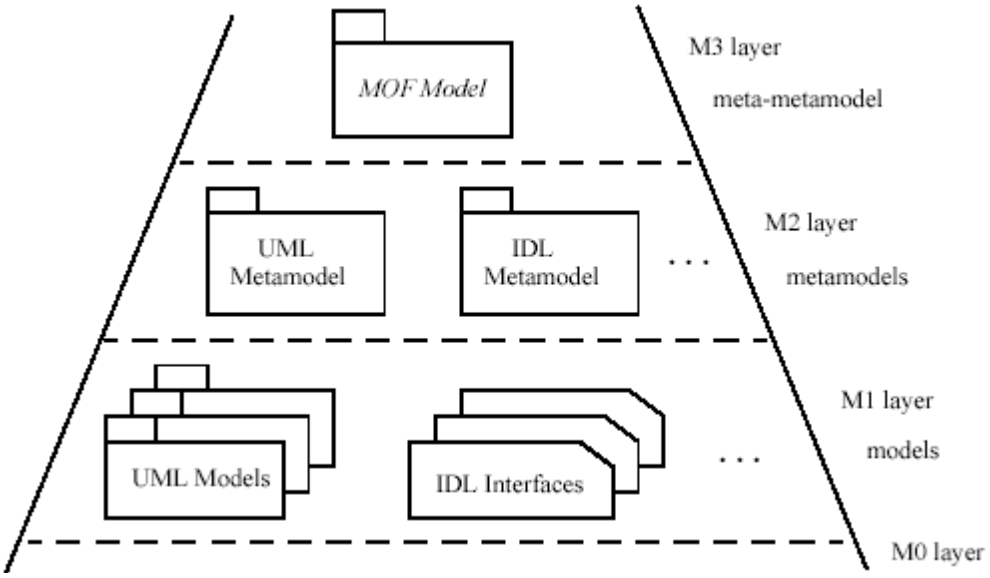


Figure 2.6 - Le MOF à la base de plusieurs métamodèles

2.4. *Technologies normalisées par OMG*

2.4.1. MOF

Le *Meta Object Facility* (MOF) [14] est normalisé par OMG dans sa version courante 2.0. Il se situe au niveau le plus abstrait à la 4ème couche. Il est le langage utilisé pour définir UML et le *Common Warehouse Metamodel* (CWM) qui inclut divers métamodèles comme ceux des bases de données relationnelles, fouille de données et processus des entrepôts de données.

MOF est aussi utilisé pour définir un format d'interchange pour modèles du niveau 2. Il s'agit du format standard XMI qu'on définira ultérieurement.

En ce qui concerne l'architecture MDA, MOF permet la définition des langages de modélisation, ainsi que l'écriture des règles de transformation.

2.4.2. *Query, Views and Transformations* (QVT)

Ce standard est en cours de finalisation [19], et sa version finale sera en principe publiée en juin 2006. Il a été proposé suite à un *Request For Proposal* (RFP) [20] lancé par OMG en l'an 2002. Ce RFP vise à normaliser un langage d'expression des règles de transformation entre les modèles écrits en MOF. Il fera partie de MOF 2 et il comprendra en plus d'autres langages spécialisés pour la création de vues, de requêtes et pour l'écriture de définitions de transformations.

Un survol rapide des soumissions réponses au RFP sur QVT sera fait au cours de la section 2.5.5 traitant les langages de transformation.

2.4.3. Unified Modeling language (UML)

C'est le langage de modélisation le plus largement utilisé. On est déjà rendu à la version 2.0. Son métamodèle [24] est écrit dans MOF et par abus de langage on peut dire que ce métamodèle de UML est instance de MOF.

Le rôle de UML est essentiel comme langage de modélisation normalisé de OMG. UML prend aussi avantage de MOF pour l'écriture des définitions de transformations.

Notons ici que ce langage est adéquat pour l'expression de la structure d'un système, mais souffre d'une insuffisance vis à vis de la modélisation d'un comportement c'est à dire des aspects dynamiques. En effet les diagrammes de machines à états finis de l'UML deviennent très compliqués quand on s'en sert pour décrire les opérations induites par un système relativement simple.

2.4.4. Object Constraint Language (OCL)

C'est un langage d'expression de contraintes. Il est utile pour la définition précise d'un langage ou d'une transformation de modèles; une requête OCL peut spécifier les éléments ciblés par une règle de transformation.

OCL étend l'expressivité de UML. Il permet de spécifier :

Les valeurs initiales des attributs;
Les règles de dérivation des associations et attributs;
Les destinations des messages envoyés;
Les conditions de garde pour une machine à états finis (FSM);
Les requêtes de l'utilisateur sur un modèle.

Le mariage UML-OCL pallie partiellement la faiblesse de UML vis à vis de la définition de post et pré-conditions des opérations et des conditions de garde des machines à états finis (FSM). Notons de plus que OCL est utilisé avec MOF pour décrire un métamodèle.

2.4.5. Action Semantics

Le langage UML Action Semantics (AS), normalisé par l'OMG en 2002, est une extension de UML qui fournit un langage d'expression de l'aspect dynamique d'un système. AS attache le comportement d'un système à des machines à états finis pour être utilisé dans le domaine de logiciels embarqués. Mais AS souffre toujours de l'absence de standardisation concrète de la notation.

2.4.6. Common Warehouse Metamodel (CWM)

C'est un langage de modélisation spécialisé pour les entrepôts de données (*Data Warehousing)*. Il comprend des métamodèles simples pour décrire une base de données relationnelle, de fouilles de données, etc.

2.4.7. Les profils UML

Un profil décrit une extension de UML spécifique à une technologie d'implémentation. Chaque profil est défini par un ensemble de stéréotypes (ex. <<Java Class>>), de contraintes comme la condition d'héritage simple en Java, et des étiquettes (tags) qui agissent comme des attributs des métaclasses de UML permettant ainsi d'attacher des informations à ses instances.

Le profil UML pour *Enterprise Distributed Object Computing* (EDOC) [23] normalisé par l'OMG, constitue un accomplissement important aidant à représenter, indépendamment de toute plate-forme technologique, les notions spécifiques au domaine des applications dites *Enterprise Application.* Ces applications sont caractérisées par le recours à un traitement distribué avec la nécessité d'une gestion du comportement transactionnel et des exigences élevées de performance.

Le Profil UML pour *Enterprise Application Integration* (EAI) [37] est aussi essentiel pour représenter les informations d'accès aux interfaces des applications.

2.4.8. XMI

XML *Metadata Interchange Format* (XMI) définit un mécanisme pour la correspondance entre les métamodèles basés sur MOF et le XML et schémas.
La représentation d'un modèle au format XMI est implémentée en se basant sur l'une des deux approches :

1. Les schémas définissent des formats d'échange pour les métamodèles en offrant une représentation standard de ces derniers.
2. Un document XML conforme à MOF forme une représentation de métamodèle.
    - On utilise alors le document XML pour représenter un métamodèle.

À titre d'exemple, XMI 1.1 [25] a été utilisé pour produire un schéma qui définit la représentation physique du métamodèle d'UML 1.3.

En conséquence, XMI permet de représenter n'importe quel modèle appartenant à la couche M0, M1 ou M2 par un fichier XML en autant qu'il est conforme à MOF.

2.5. Élaboration de transformation de modèles

Dans cette section on va examiner les divers approches et moyens utiles pour la définition des règles d'une transformation et pur leur application.

2.5.1. Détection et traitement de Patterns

Cette approche s'appuie sur la recherche et le traitement des instances d'un ensemble de patterns dans le modèle source. Un pattern est un métamodèle représentant un ensemble d'éléments satisfaisant certaines caractéristiques tout en restant conformes au métamodèle du modèle de départ. Un fragment de modèle est conforme à un pattern P et est dit instance de P si ce fragment satisfait l'ensemble des règles contenues dans P.

Notons ici qu'un langage du niveau M3, comme le MOF, sert à l'écriture du métamodèle de P.

Ainsi le processus de la transformation se réduit à la recherche des instances de tout pattern P pour leur appliquer le traitement correspondant. Il les transforme en instances d'un autre pattern cible P'. En d'autres termes, chaque pattern détermine un ensemble d'éléments qui seront affectés par la transformation selon certaines règles appelées schéma de la transformation. Une telle technique est illustrée dans [16].

Le schéma détermine le traitement que doit subir une instance du pattern P; il identifie les classes à éliminer et celles à créer. Il définit le pattern cible P'. Cela permet de décrire un schéma par le couple de métamodèles P et P'; ou aussi par un seul métamodèle représentant les deux avec un marquage servant à distinguer la partie source de la partie cible ou la partie à éliminer de la partie à créer. La figure 2.7 tirée de [16] montre le rôle des métamodèles comme abstraction des instances des patterns et des règles de transformation.

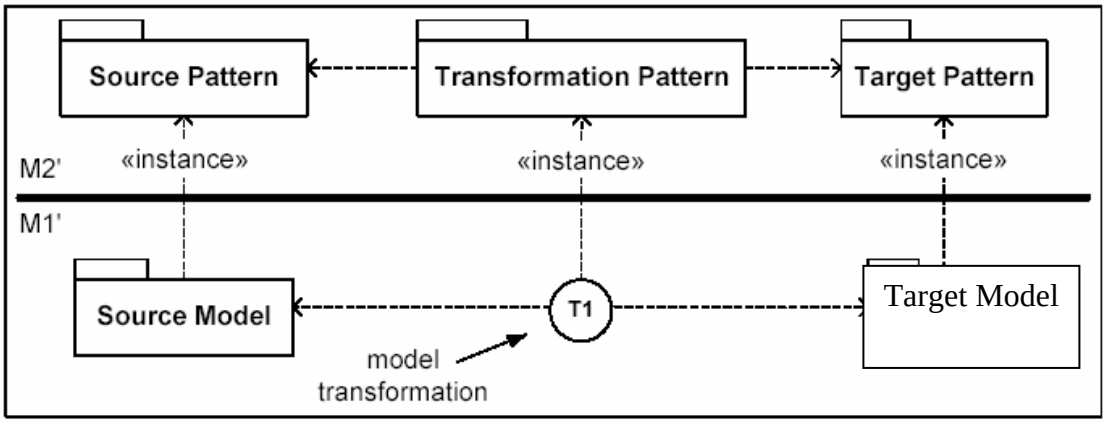


Figure 2.7 - Les patterns d'une transformation au niveau M2

2.5.2. Approche graphique

L'opération clé dans une transformation de modèle réside dans le choix et la détection des éléments à transformer. Les techniques de transformation de graphes peuvent être étendues et réutilisées dans le contexte de UML en utilisant un langage comme Genermorphous (Gmorph) [30].

Les systèmes de transformation de graphes offrent des moyens déclaratifs pour automatiser la détection de graphes satisfaisant des critères de sélection donnés [29].

Les règles de transformation de graphes décrivent des fragments de graphes bien déterminés appelés les *Left Hand Side* (LHS) graphes. La transformation en détecte les occurrences dans le modèle de départ, et ceci en vue de les transformer en des graphes appelés *Right Hand Side* (RHS) graphes dans le modèle cible.

Du fait que UML est un langage de modélisation graphique, un modèle écrit dans ce langage peut être représenté d'une façon ou d'une autre par un graphe. Ceci implique la possibilité de lui appliquer une transformation de graphes pour obtenir un nouveau modèle.

Une extension des transformations des graphes est nécessaire pour les adapter au domaine des modèles écrits en UML. Cette extension comprend un certain nombre de points de généralisation :

- LHS pourra spécifier le type de groupe (meta-type « classifier »), il pourra ignorer les noms des nœuds.
- LHS pourront tenir compte des propriétés des éléments comme les stéréotypes.
- Il faut que la transformation puisse prendre en considération les relations entre les éléments.

On ajoute à ce qui précède, l'usage de OCL comme langage d'expression de contraintes et prédicats.

En effet le langage Gmorph déjà mentionné et qui est en cours de développement [30], implémente ces derniers points. Il va aussi plus loin en représentant les LHS par des métamodèles.

2.5.3. Techniques générales

Quelle que soit l'approche, on peut toujours utiliser plusieurs astuces et techniques pour raffiner ou guider une transformation.

Usages de paramètres : déjà mentionné dans 2.2. On rappelle qu'une transformation peut dépendre d'un certain nombre de paramètres dont le développeur utilise les valeurs pour optimiser son implémentation des caractéristiques et fonctionnalités offertes par une plate-forme cible.

Le marquage : comme étape préliminaire au processus de la transformation, le développeur peut marquer des éléments du modèle de départ pour qu'ils soient traités d'une manière spécifique par les processus automatisés qui seront appliqués. À titre d'exemple, une marque qui fait référence à un concept dans le PSM, est appliquée à un élément du PIM, pour indiquer comment cet élément doit être transformé [15]. Les marques ne font pas partie du PIM mais elles peuvent être comprises comme des informations supplémentaires, spécifiques à une plate-forme et appliquées à une couche transparente superposée au PIM.
Un exemple concret de marquage est l'ajout de la marque 'entité' à une classe pour que la transformation lui applique un traitement spécifique. Les stéréotypes de UML servent aussi d'exemple de marques.

Notons de plus que pour être utiles, les marques doivent être structurées, et leur usage doit répondre à des contraintes comme l'exclusion mutuelle. Elles existent selon un modèle qui structure leur sémantique et coexistence.

Le marquage et les paramètres complètent les informations, qui manquent dans un PIM, d'une façon supportée par le PSM choisi. Les marques peuvent représenter aussi des préférences du développeur et même des spécifications de la qualité de service exigée au niveau du PSM.

2.5.4. Niveau d'automatisation

En principe, quand les modèles décrivent un ensemble fini de concepts affectant leurs transformations, l'automatisation d'une transformation de modèle s'avère possible [12]. Ceci demeure vrai en autant que le modèle de départ est complet, c'est à dire que la transformation ne requiert pas l'ajout d'information indispensable à la production du modèle cible.

Dans le cadre de MDA, le passage du PIM au PSM peut être complètement automatisé dans un contexte où l'on a préparé les éléments suivants pour la réutilisation [14] :

- Un modèle du style architectural;
- Un gabarit (*template*) dans le modèle du style architectural, comme un système type PIM, qui peut être automatiquement transformé vers les diverses plates-formes cibles;
- L'outil nécessaire pour fournir le modèle aux développeurs sous la forme de profil, avec la vérification de la conformité du modèle, les liens à un IDE, un processus de support, etc.;
- Les règles de transformation pour chaque plate-forme cible.

Le travail du développeur se limite alors à l'élaboration du PIM qui sera transformé automatiquement en PSM; et ceci grâce au soutien de l'environnement de développement qui offre des informations de préemballage pour le PSM. Ce préemballage, représenté par les quatre points cités ci haut, guide la définition de la transformation qui produit un PSM à partir du PIM à travers un ensemble de patterns. Le code peut être directement produit mais le PSM reste toujours utile pour les tests et la maintenance.

À la lumière de ce qui précède, on va opter dans notre travail pour une transformation sujette de notre étude selon une approche utilisant les patterns, car elle permet d'exploiter la correspondance entre concepts en les traduisant en correspondance entre patterns dans les modèles source et cible. L'approche graphique souffre toujours du besoin d'une extension qui pourra l'adapter à notre transformation.

Par ailleurs, l'application de notre transformation n'exige aucune opération de préparation du modèle source et fournit un modèle cible qui compense les informations manquantes dans le modèle source par des valeurs prises par défaut pour produire les informations qui y correspondent dans le modèle cible.

2.5.5. Langages de transformations de modèles

Pour définir une transformation de modèles, on peut utiliser un langage non formel, un langage d'action pour représenter l'algorithme de la transformation, ou encore un langage bien défini de *mapping* de modèles.

Conscient du besoin d'un langage bien défini et normalisé pour l'expression de transformation de modèles, l'OMG a publié un *Request for Proposal* (RFP) pour MOF 2.0 Query/View/Transformation (QVT) [20]. On est rendu à une spécification non finalisée [36] qui est inspirée des différentes soumissions réponses au RFP. Comme indiqué dans [36], "une principale exigence de QVT est de fournir un standard pour exprimer des transformations". QVT exige que les transformations de modèles soient définies avec précision en termes de rapport entre un métamodèle de la source et un métamodèle de la cible. Ces deux métamodèles sont conformes à MOF [14].

Notons ici qu'un langage de transformation de modèles forme un élément pivot des trois composantes de Query/Views/Transformations. C'est que chacune prend comme entrée un modèle, pour fournir soit un ensemble de n-uplets *(tuples)* dans le cas d'une requête, soit un autre modèle dans le cas d'une vue ou d'une transformation. Une vue est un modèle qui montre un aspect spécifique d'un modèle.

Le QVT RFP a déjà généré 8 propositions ainsi qu'un certain nombre d'initiatives indépendantes. Nous allons citer quelques-unes rapidement en insistant sur les langages s'avérant utiles à notre future contribution dans le domaine. De plus amples informations sur les outils d'implémentation seront données ultérieurement.

- OpenQVT : [22] soumis par un groupe d'universités et de sociétés françaises en réponse au QVT RFP.
  Ce langage est à la base du développement de l'implémentation de la syntaxe ATL[33] par un outil prometteur de transformation de modèles ADT qui est intégrable dans l'environnement de développement Eclipse [26].

- XMOF de IBM et Compuware :

Comparable à XSLT qui est un langage applicable pour exprimer des transformations de fichiers XML, XMOF est un langage déclaratif qui décrit les résultats voulus d'une transformation plutôt que les processus nécessaires à sa réalisation.
XMOF utilise les standards OMG existants et est considéré par ses auteurs comme une extension de MOF utilisant OCL pour l'expression des requêtes et prédicats [39].
Ce langage utilise le concept de pattern comme modèle répondant à un ensemble de contraintes. Ce langage se base sur la détection des occurrences de chaque pattern dans un modèle pour créer des instances des patterns de sortie correspondants.

XMOF supporte la composition de transformations et la persistance de synchronisation entre modèles d'entrée et de sortie.

  - Outils :

L'outil de modélisation et de développement MDA OptimalJ implémente avec transparence la majorité des concepts de XMOF. OptimalJ est un produit commercial de Compuware.

Mecato de meta-case est un outil open-source MDA avec une infrastructure en MOF, OCL et XMOF. Un prototype de Mecato est présentement disponible, il implémente des requêtes (Queries) et des transformations de modèles.

- Le langage TRL [28]

Ce langage est une autre proposition en réponse au QVT RFP soumise par Alcatel Softeam, Thales, TNI-Valiosys et Codagen Technologies Corp, et supportée par France Telecom, INRIA/IRISA, Softeam, Université de Paris VI, Université de Nantes, LIFL, CEA.

TRL utilise les standards de l'OMG comme OCL, pour exprimer les contraintes et les *queries* de MOF. De plus, ce langage est basé sur une technique de métamodélisation avec les deux approches déclarative et impérative. Il permet le marquage et la paramétrisation, et il accepte plus qu'un modèle d'entrée à la fois.

- Outils :

SODIFRANCE a développé l'outil MIA-Transformation dans la série Model-In-Action. Cet outil est basé sur TRL et accepte un modèle d'entrée dans divers formats tel que XMI, Rose et autres. Les transformations sont exprimées en utilisant des règles d'inférence. MIA-Transformation a été déjà utilisé dans divers domaines : la transformation de modèles UML d'analyse en modèles UML de conception, la transformation des modèles d'ingénierie de système en modèles UML, la rétro-ingénierie de bases de données relationnelles en modèles UML [28].

Par ailleurs, France Télécom a développé un prototype de compilateur de TRL pour s'en servir dans son projet TRAMs qui traite de la migration de systèmes informatiques en utilisant la transformation de modèles.

- QVT Partners [35]

Est une soumission de proposition QVT qui se distingue par la considération de l'aspect unidirectionnel et bidirectionnel des transformations. Ses LHS et RHS ressemblent à ceux de UMLX, qu'on verra dans cette section, mais sans supporter les multiplicités. De plus ce langage compte sur l'expression textuelle de la relation entre LHS et RHS qui forme une contrainte qui doit être satisfaite après tout passage de LHS vers RHS. Ce langage utilise Action Semantics pour l'expression des *mappings* en plus de OCL.

- MOLA

Le laboratoire IMCS de l'université de Latvia a développé le Model Transformation Language (MOLA) comme une tentative de définition de transformations plus **naturelles** et **lisibles**. On a adopté des structures de contrôle itératives et simples plutôt que récursives puisées dans la programmation structurelle traditionnelle.
MOLA utilise l'approche de réécriture graphique utilisant les LHS et RHS avec une instruction essentielle 'loop' qui cherche tous les fragments de graphes coïncidant avec un LHS.

- UMLX [7]

Développé par GMT Consortium, UMLX est un langage de transformation de modèles UML avec une approche graphique basée sur les LHS et RHS avec un style déclaratif.

- Outil : UMLX 0.0 est gratuitement téléchargeable comme éditeur de modèles UML et compilateur du langage UMLX, mais sous forme de prototype dans une phase embryonnaire.

Citons ici GReAT[3] et ATL [4] qui suivent la même philosophie de métamodélisation de UMLX avec une différence de syntaxe concrète. Les trois langages doivent en principe évoluer vers le standard attendu de QVT.

- DSTC par DSTC Pty Ltd [6]

Une autre soumission de proposition pour QVT RFP qui est conçue de manière à répondre aux exigences de ce dernier RFP.

  - Outil : la gamme d'outils de DSTC Pty Ltd intègre un compilateur de ce langage.

2.6. Choix d'un outil

Pour implémenter notre transformation on a choisi l'outil MIA-Transformation car c'est un outil qui permet une implémentation de notre transformation avec une grande flexibilité. Il répond à nos besoins de tirer profit des deux styles impératifs et déclaratifs de définition d'une transformation de modèle, en plus des caractéristiques ci-dessous :

- Il support le format XMI pour les modèles et métamodèles;
- Il intègre les métamodèles sous forme de modèles UML créés dans Poseidon qui est un éditeur de modèle téléchargeable gratuitement;
- Il implémente plusieurs langages pour définir et exécuter une transformation, y compris une bibliothèque spécialisée de classes de Java offrant des services de manipulation de fichiers XMI en acceptant comme arguments les noms des éléments des métamodèles.
- Il permet une réutilisation à travers des services qui peuvent être invoqués à plusieurs reprises par différentes règles de transformation.
- Il jouit d'une interface graphique permettant le développement convivial de la transformation.
- Il permet un débogage efficace des règles de transformation dans ses différents langages.

Ainsi, MIA[1] se distingue en offrant tout ce qui précède dans un seul outil, alors que les autres outils ne supportent qu'une partie de ces caractéristiques.

[1] Nous tenons à remercier le personnel responsable de la commercialisation de cet outil de la société Sodifrance pour nous avoir octroyé une licence spéciale pour utiliser cet outil sans aucune limitation.

Chapitre 3

Le métamodèle des EJB

Le travail de plusieurs compagnies pionnières dans le domaine de l'orienté objet et du calcul distribué, a donné naissance à un métamodèle de l'architecture des Enterprise JavaBeans (EJB) [21]. Ce métamodèle a été conçu dans le but d'être au service des développeurs d'applications utilisant les composants et exigeant un déploiement. L'OMG qui a adopté ce métamodèle lui a donné une bonne crédibilité.

3.1. Explication préliminaire et terminologie

Avant de passer à l'explication du métamodèle, il importe de fournir une vue d'ensemble très succincte de la technologie EJB. Un composant EJB profite de maints services durant le temps d'exécution. Ces services seront offerts par un serveur d'application via un objet appelé conteneur EJB. Ce conteneur offre ces services, qui sont des services de middleware, selon une déclaration des besoins faite par le développeur dans un fichier appelé descripteur de déploiement. Le serveur traite le descripteur de déploiement pour que les services exigés soient rendus. Ces services concernent principalement des fonctionnalités comme la connectivité entre client et composant, la gestion de la persistance, la gestion de la sécurité, la gestion de la concurrence, la gestion du cycle de vie des instances et la création de réserves de connexions.
Une instance d'un composant EJB est appelée bean; c'est une instance d'une classe implémentant les méthodes que le composant offre à ses clients et qui forment l'interface métier de ce composant. La classe du bean doit implémenter aussi un ensemble de méthodes dont le conteneur aura besoin pour gérer le cycle de vie du bean.
Un bean est de deux types :

- Bean **session** orienté vers des services de traitement offerts au client de ce bean.

- Bean **entité** orienté vers un service de persistance de son état. Il représente un objet du domaine métier intéressant un client de ce bean.

3.2. Les diagrammes

On va expliquer ici les éléments clés du métamodèle à travers les principaux diagrammes, mais une consultation du métamodèle complet [21] et des spécifications des EJB [31] est toujours recommandée.

Les diagrammes présentés vont contenir les différents types de composants selon la technologie EJB ainsi que le *assembly descriptor* où l'on déclare et spécifie les services de sécurité et de transaction requis pour un ensemble de composants appartenant à un même assemblage de déploiement concrétisé par un fichier appelé ejb-jar.

Dans la suite, et par commodité, une instance d'un élément de métamodèle portera le même nom que l'élément à partir duquel elle a été instanciée; la distinction se faisant par le fait que l'instance est précédée par le déterminent ``un`` ou ``des``. Exemple : EnterpriseBean représente un élément du métamodèle EJB, alors que un EnterpriseBean dénote une instance de cet élément. Pour chacun des éléments de ce métamodèle, on expliquera ce que représente cet élément par rapport à la technologie EJB, et on regroupera sous la rubrique éléments possédés ses attributs et les éléments qui lui sont associés par une association de composition. On désigne par éléments reliés, les éléments qui sont associés selon une association de non agrégation à l'élément considéré.

3.2.1. Diagramme principal

Le diagramme principal de la figure 3.1 montre la classe base de tout composant EJB **EnterpriseBean** qui fait partie d'un fichier ejb-jar représenté par l'élément **EJBJar** du diagramme principal**.** Le fichier ejb-jar sert d'assemblage de composants et des informations nécessaires à leur déploiement et aux services offerts par le conteneur. Ces informations sont stockées dans un élément appelé **assembly** appartenant à un fichier au format XML appelé descripteur de déploiement. Les composants de type **Session** offrent des services aux clients appelant, via leurs interfaces, des méthodes implémentant des fonctionnalités propres au domaine métier. Les composant de type **Entity** offre un service de gestion de persistance des états de leurs instances.

EJBJar
Sémantique
EJBJar est l'élément de racine du descripteur de déploiement des composants EJB assemblés dans EJBJar.

Éléments possédés

description : L'élément de description est employé par le producteur de dossier d'ejb-jar pour fournir le texte décrivant l'élément parent. L'élément de description devrait inclure n'importe quelles informations que le producteur de dossier d'ejb-jar veut fournir au consommateur du dossier d'ejb-jar (c.-à-d., au Déployeur). Typiquement, les outils employés par le consommateur de dossier d'ejb-jar montreront la description quand ils traitent l'élément parent.

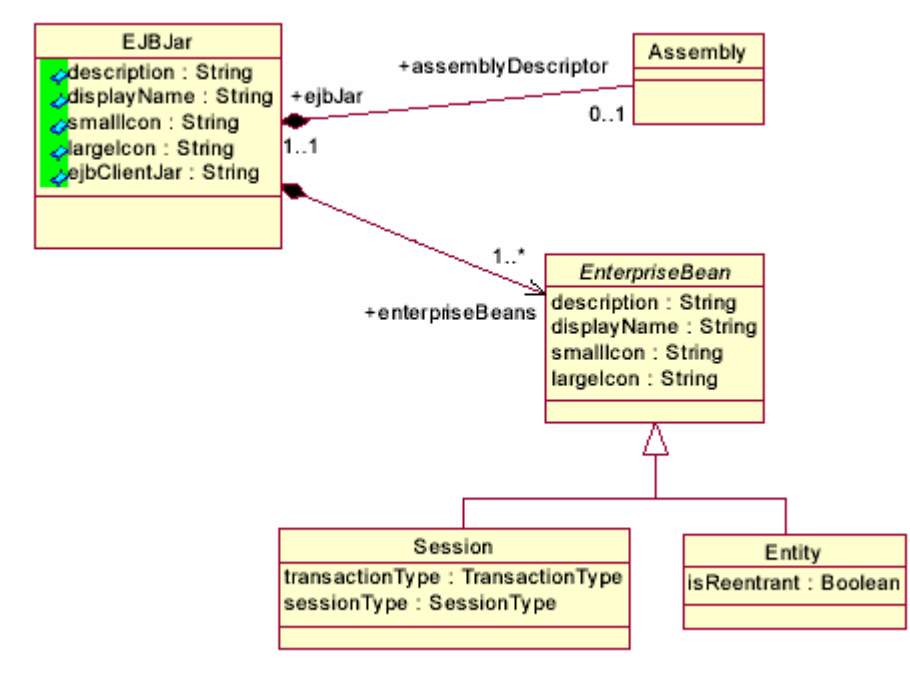


Figure 3.1 - Diagramme principal

displayName : l'élément de displayName contient un nom court qui est prévu pour être affiché par des outils.
smallIcon et largeIcon : ce sont deux attributs pour indiquer les noms des fichiers des petites et grandes icônes.
EjbClientJar : cet attribut indique un nom optionnel pour un fichier ejb-client-jar pour le ejb-jar. Le client d'un composant EJB doit avoir son propre fichier jar qui lui permet d'appeler des composants contenus dans un fichier ejb-jar.

Éléments reliés
Assembly, EnterpriseBean

Assembly
sémantique

Assembly représente l'élément assembly-descriptor de la technologie EJB. L'assembly-descriptor contient l'élément application-assembly du fichier descripteur de déploiement. L'élément application-assembly comprend les pièces suivantes : la définition des rôles de sécurité, la définition des permissions de méthodes, et la définition des attributs de transaction pour les composants représentés par des éléments instances de EnterpriseBean. Toutes les pièces sont facultatives dans le sens qu'elles sont omises si les listes représentées par elles sont vides. Fournir un assembly-descriptor dans le descripteur de déploiement est facultatif pour le producteur de dossier d'ejb-jar.

Éléments reliés
EJBJar

EnterpriseBean
Sémantique
EnterpriseBean est une classe qui représente le composant selon la spécification EJB. Il peut avoir des attributs, des opérations et des associations qui sont dérivés ou filtrés de ses classes et interfaces d'exécution.

Éléments possédés
description : (idem que pour description de la page précédente)
displayName : (idem que pour displayName de la page précédente)
smallIcon et largeIcon : (idem que pour smallIcon et largeIcon de la page précédente)

Éléments reliés
EJBJar : cet élément identifie le descripteur de déploiement à qui appartient le composant EJB.

Session
Sémantique
Un objet qui fournit des traitements sans se préoccuper de la sauvegarde sur un support persistant. Il correspond au bean session dans les spécifications des EJB.

Éléments possédés
transactionType : c'est un attribut qui spécifie le type de gestion du comportement transactionnel.
sessiontype : il indique si la session est de type avec état (*stateless*) ou sans état (*stateful*).

Éléments reliés
EnterpriseBean comme une super classe abstraite.

Entity
Sémantique
Un objet qui est orienté état plutôt que traitement. Il correspond à un bean entité dans la spécification EJB.

Éléments possédés
isReentrant : c'est un booléen qui indique si la réentrance est permise ou non pour un bean entité.

Éléments reliés
EnterpriseBean comme une super classe abstraite.

3.2.2. Diagramme de l'Assembly

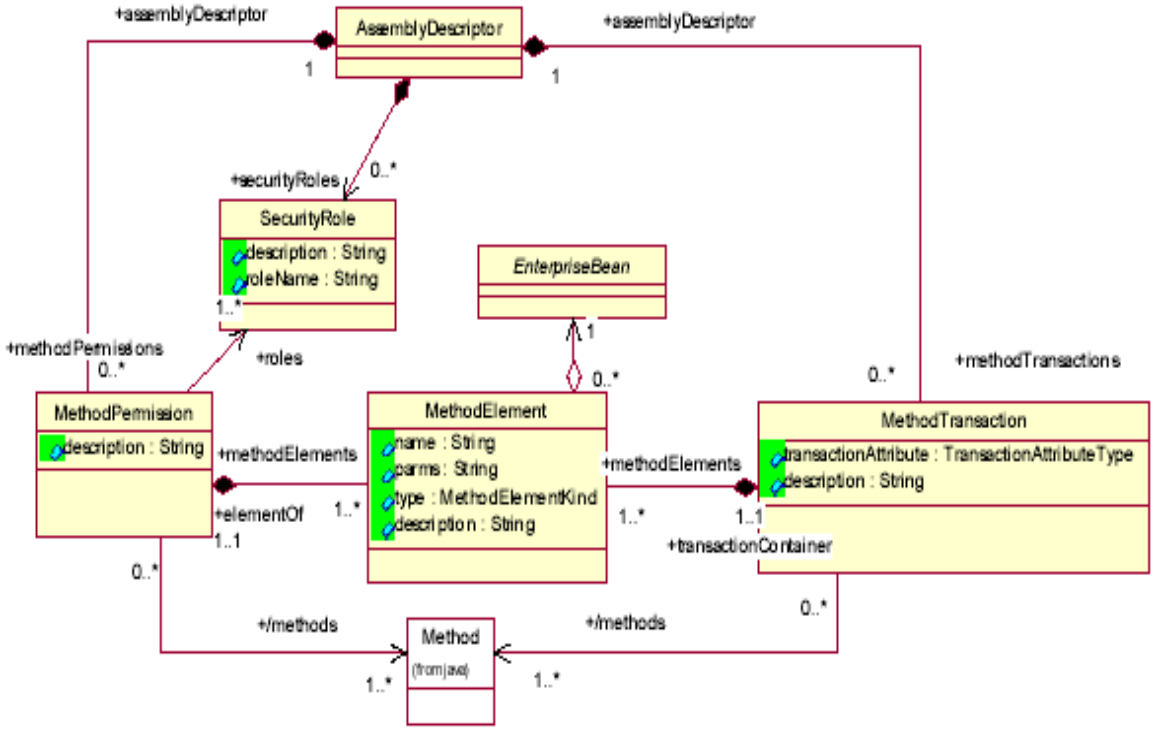


Figure 3.2 - Diagramme de l'Assembly

Le diagramme ci-dessus illustre la structuration des informations de déploiement d'une ou de plusieurs EnterpriseBean. On y trouve les éléments suivants :

MethodElement
Sémantique
Cet élément représente l'élément MethodElement de l'AssemblyDescriptor du fichier descripteur de déploiement selon la spécification EJB.

Éléments possédés
Name : une chaîne de caractères donnant le nom de la méthode à laquelle MethodElement fait référence.
Parms : une chaîne de caractère indiquant les paramètres de la méthode.

Type : un choix d'une énumération {Home, Remote, Unspecified} indiquant à quel type d'interface appartient cette méthode.
Descritption : c'est une chaîne de caractère de description.

Éléments reliés
MethodTransaction, MethodPermission et EnterpiseBean.

MethodTransaction :
Sémantique
Cet élément spécifie le mécanisme de gestion de transaction pour une méthode bien déterminée référencée par une instance de l'élément **MethodElement** appartenant a un composant de type EnterpriseBean.

Éléments possédés
TransactionAttribute : cet attribut indique un choix de type de gestion de transaction à offrir à une méthode.
Descritption : c'est une chaîne de caractères de description.

Éléments reliés
Method de Java et MethodElement.

SecurityRole :
Sémantique
Cet élément représente la définition des rôles de sécurité dans l'Assembly descritpor.

Éléments possédés
name : une chaîne de caractères donnant le nom de rôle de sécurité.
description : une description en texte informel du rôle de sécurité que représente l'élément parent.

Éléments reliés
MethodPermission : Un MethodPermission associé à un SecurityRole *r* et à un MethodElement *mth* donne au rôle de sécurité représenté par *r* la permission d'exécuter la méthode représentée par *mth*.
AssemblyDescriptor (voir l'explication de l'élément Assembly du diagramme principal)

MethodPermission
Sémantique
Une instance de cet élément représente une autorisation d'un rôle de sécurité à une méthode d'un composant de Type EnterpriseBean.

Éléments possédés
description : un attribut décrivant l'autorisation de rôle de sécurité à une méthode.

Éléments reliés
MehtodElement, SecurityRole et Method de Java.

3.2.3. Les diagrammes restants

Comme déjà mentionné, il reste un ensemble de diagrammes qu'on cite comme suit : EJB, Entity Bean, EJB Implementation, References to Resources, Data Types.

L'explication de ces diagrammes sort du cadre de notre travail, mais est disponible dans [21], en plus du métamodèle de Java qui se base sur les spécifications des EJB [31].

Chapitre 4

Proposition d’un métamodèle pour les composants .NET

Ce métamodèle fait ressortir les éléments caractérisant la technologie des composants .NET. On a tenté d’y inclure assez d’éléments et d’informations pour en faire un langage permettant la description d’un système utilisant cette technologie. Ce métamodèle a reçu un feedback positif de la communauté UML/MDA [1]. Une introduction rapide à la technologie des composants .NET ainsi qu’une explication des diagrammes du métamodèle sont présentés dans ce chapitre.

4.1. Introduction à la technologie des composants .NET

4.1.1. Vue d’ensemble

Le .NET Framework est caractérisé par une ouverture à l’intégration de langages objet. Il est basé sur deux composants principaux : le Common Language Runtime (CLR) et la bibliothèque de classes du .NET Framework.

Pour être intégré dans le .NET Framework, un langage doit suivre un ensemble de règles connues sous le nom de Common Language Specification (CLS) et il doit être muni d'un compilateur qui transforme son code source en un langage appelé MSIL qui sera pris en charge par le Common Language Runtime (CLR). Ce runtime peut être compris comme une machine virtuelle qui s'occupe de l'exécution du code compilé en lui assurant des services de temps d'exécution comme le chargement, la transformation en code natif, la gestion de mémoire et l'isolation des programmes, etc. Un code ciblant le CLR sera dit un code managé, car il profite d'une gestion offerte par le CLR pour son exécution.
La bibliothèque de classes est l'autre composant principal du .NET Framework. Elle est formée d'une collection complète orientée objet, de types réutilisables qui répondent aux besoins des développeurs des applications traditionnelles ou des applications Web. Cette bibliothèque forme une API unifiée utilisable par tous les langages intégrés dans le .NET Framework.

Le langage vedette du .NET Framework est C#. Microsoft fournit aussi des compilateurs pour C++ et Visual Basic qui est devenu entièrement orienté objet. Microsoft a signé des accords avec d'autres fournisseurs de compilateurs pour d'autres langages tels que Pascal, Eiffel, Perl et Cobol.

4.1.2. Les composants

Depuis une dizaine d'années, Component Object Model (COM) [9] de Microsoft était le standard pour les composants qui s'exécutent sur des plates-formes Windows. En juillet 2000, le modèle des composants .NET a été lancé par Microsoft sans offrir une compatibilité avec COM. Mais ce modèle en hérite son infrastructure et ses services à l'exécution (*run time*), appelés services COM+.

Au départ, il faut comprendre les frontières logiques qu'un appel à un objet peut rencontrer et éventuellement traverser. Le système d'exploitation Windows maintient des frontières entre ses différents *threads* qu'on appelle les processus. Un processus n'est pas un environnement d'exécution direct pour .NET (du code managé). Il englobe plusieurs processus logiques qu'on appelle domaines d'applications qui forment l'environnement d'exécution direct pour .NET.

D'une façon générale, un composant permet la création d'objets réutilisables et qui interagissent avec d'autres objets. Selon la technologie .NET, pour qu'un objet soit accessible à distance ou hors de son domaine d'application, il faut que cet objet soit une instance d'une classe qui est dérivée de la classe MarshalByRefObject (MBR) de l'espace de nom *System* de l'API de .NET.

Un composant dérivé de MBR offre à un client appelant distant, une référence sur lui-même. L'appelant aura un objet appelé proxy sur lequel il invoque les méthodes de l'interface du composant distant. Le proxy achemine (marshal) les appels des méthodes à l'instance du composant distant et retourne les réponses à l'appelant local. Dans la suite la classe, dérivée de MBR et implémentant l'interface d'un composant sera appelée *type serveur* et ce en vue de simplifier les explications.

Une application .NET peut héberger un composant en déclarant l'hébergement par programmation ou dans un fichier de configuration. Cette opération est appelée enregistrement; elle fournit à l'application des informations spécifiant le protocole de communication, le numéro de port, le nom du type serveur, le nom de l'assembly et le mode d'activation (qu'on expliquera dans la suite). De la même façon une application contenant des appels à un composant devra le déclarer en fournissant les informations pertinentes.

L'enregistrement d'hébergement ou d'appel d'un type serveur par une application .NET spécifie un ensemble de paramètres via des éléments structurés dans un fichier de configuration au format XML, conforme à un schéma bien déterminé. Une représentation fidèle de ce schéma sera illustrée dans le diagramme ConfigurationFile de la section suivante.

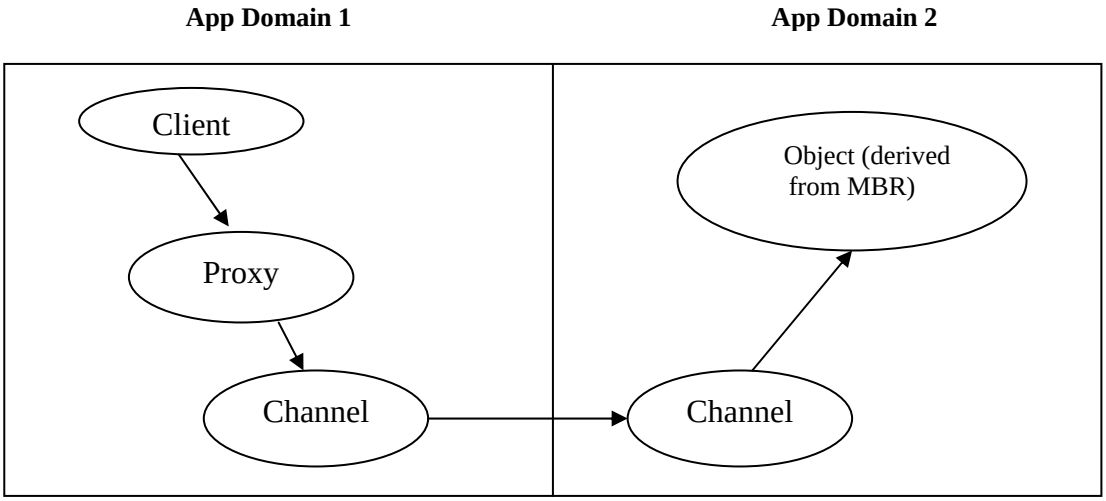


Figure 4.1 - Un objet local offrant l'interface d'un composant à une application cliente

Les principaux éléments de configuration sont en relation avec les canaux de communication et la gestion des instances des composants (figures 4.1 et 4.2). Ils serviront à comprendre les sémantiques des éléments du diagramme de configuration du métamodèle de la section suivante. Il s'agit des éléments suivants :

1. **Channel** : il spécifie le protocole de communication et le numéro de port pour écouter des appels distants aux composants hébergés.
2. **service** : il permet de choisir un des **modes d'activation** suivants pour un composant hébergé :
   - **Client-Activated** : ce choix fait que chaque client aura sa propre instance du composant. La durée de vie de l'instance sera conditionnée par les valeurs offertes dans l'élément lifeTime du même fichier de configuration. Le client appelant jouit d'une plus grande autorité de contrôle sur l'instance du composant.

- **Server-Activated** en mode **SingleCall** : un objet est crée chaque fois qu'une invocation de méthode est acheminée. Cet objet sera relâché et sera candidat pour le ramasse-miettes dés que la méthode retourne l'appel. L'état de l'objet est à sauvegarder par programmation pour une éventuelle future invocation de méthode par le même client qui aura besoin du dernier état de son instance.
- **Server-Activated** en mode **Singleton** : tous les clients se connectent à un seul objet et partagent l'état de cet unique objet.

3. **lifeTime** : ce concept indique les paramètres de gestion de la durée de vie des instances des composants dérivés de MarshaByRefObject et hébergés par une application .NET. Ces paramètres combinés avec le niveau d'activité d'une instance conditionnent la durée de vie de cette dernière.

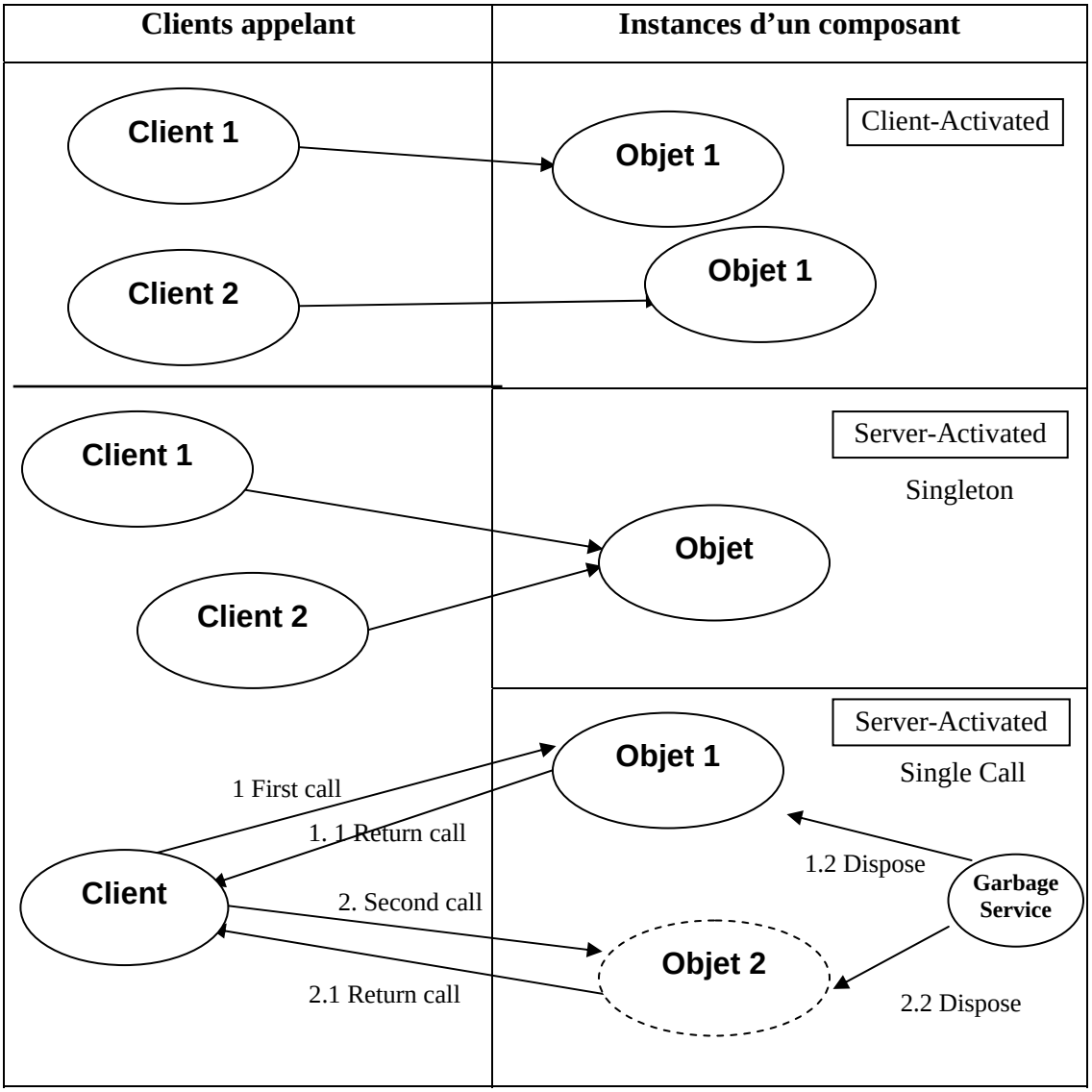


Figure 4.2 - Mode d'activation et gestion des instances d'un composant .NET

Notons que l’application cliente aura besoin des données du type serveur pour créer un proxy implémentant l’interface que le type serveur offre à ses clients, car le proxy aura à accepter des invocations locales des méthodes qu’il achemine à l’objet distant. Ajoutons aussi que l’application hôte doit être déjà démarrée pour qu’un appel à un composant hébergé puisse être retourné.

Par ailleurs, pour profiter des services COM+ hérités de COM, un type serveur doit hériter de la classe ServicedComponent de l’espace de nom System.EnterpriseServices qui hérite à son tour de MarshalByRefObject. Le type serveur doit aussi être enregistré auprès d’une application COM+. Une application COM+ est définie pour créer un groupement logique de plusieurs composants qui partagent une configuration commune de services. Une propriété essentielle d’une application COM+ est son type d’activation qui est un choix entre ‘’Library’’ pour des composants à activer dans le processus de leur créateur sans accès distant, et ‘’Server’’ pour des composants à activer dans un processus de serveur dédié avec la possibilité d’accès à distance. Notons que les services COM+ sont vitaux pour les applications des entreprises car ils couvrent la gestion de la sécurité et le comportement transactionnel.

L’implémentation des services COM+ utilise le mécanisme d’application d’attributs de .NET. Un attribut est une annotation qui associe des valeurs à un bloc de code. Ces valeurs sont accessibles durant le temps d’exécution comme données résultant de la compilation. L’enregistrement pour un service COM+ se fait par application d’attributs de l’espace de nom System.EnterpriseServices. Les attributs d’assembly qui sont caractérisés par une syntaxe commençant par "assembly : " suivi du nom de l’attribut et appliqué au niveau d’un ServicedComponent, visent l’application COM+ pour la configuration des services communs à tous les composants associés à une application. Dans le cas de services non partagés, on applique des attributs complémentaires au niveau du type serveur, de ses méthodes ou de ses interfaces. Les diagrammes expliqués COMplus Security Service et Component COMplus services expliqués dans la section suivante, couvrent les principaux services COM+.

Enfin il importe de signaler q’un composant peut être rendu disponible sans démarrer aucune application hôte si on l’héberge dans Microsoft Internet Information Service (IIS) conçu pour héberger les applications Web ou si l’hébergement est offert par un service du système [10].

Pour de plus amples informations concernant l’implémentation des composants .NET, on recommande la consultation des deux ouvrages de Lowy [9], [10] ainsi que le site de MSDN de Microsoft [34].

4.2. Explication du métamodèle

Notre métamodèle est conçu dans un souci d'y inclure l'essentiel de la sémantique des éléments de la technologie des composants .NET. Pour chacun des éléments de ce métamodèle on expliquera ce que représente cet élément par rapport à la technologie des composants .NET, et on regroupera sous la rubrique éléments possédés ses attributs et les éléments qui lui sont associés par une association de composition. On désigne par éléments reliés, les éléments qui sont associés selon une association de non-agrégation à l'élément considéré. Par commodité, une instance d'un élément de métamodèle portera le même nom que l'élément à partir duquel elle a été instanciée; la distinction se faisant par le fait que l'instance est précédée par le déterminent ``un`` ou ``des``. Exemple : ServicedComponent représente un élément du métamodèle de .NET, alors que un ServicedComponent dénote une instance de cet élément. Les diagrammes de notre métamodèle sont détaillés ci-après.

4.2.1. Diagramme principal

Ce diagramme illustre le fait qu'un type serveur doit hériter de MarshalByRefObject (MBR) et qu'il est généralement en relation avec une application hôte, avec une application cliente et éventuellement avec une application COM+ offrant un ensemble de services.

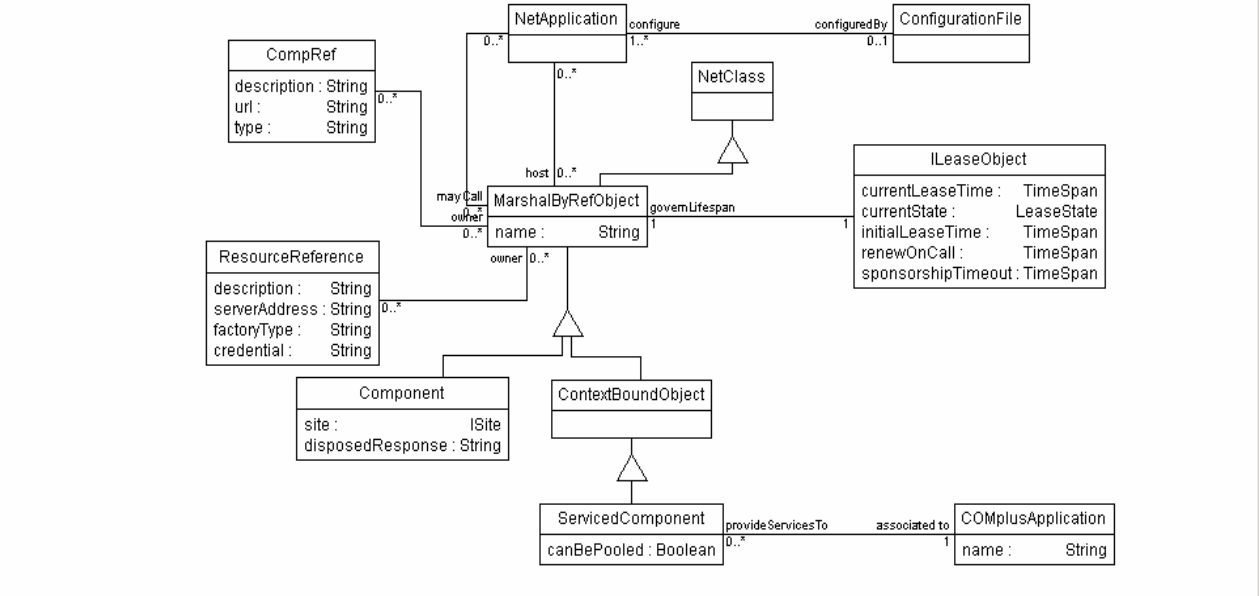


Figure 4.3 - Diagramme principal des Composants .NET

MarshalByRefObject
Sémantique
La classe MarshalByRefObject représente un parent de toute classe accessible à distance et qui offre à son appelant une référence sur elle. Cette référence est exploitée par un proxy dont dispose l'appelant pour son interaction avec une instance de la classe MarshalByRefObject . Dans la suite, on appelle instance d'un composant tout objet instance de la classe héritant de MBR implémentant ce composant. De plus on pourra designer par composant la classe d'implémentation de ce composant et vice versa au cas où ceci ne soulèverait de confusion.

Éléments possédés (attributs)

- Name : Chaîne de caractères désignant le nom de l'objet de type MarshalByRefObject.

Elements reliés
ComPlusApplication, NetApplication, CompRef, NetClass et ILeaseObject .

NetClass
Sémantique
Cet élément représente une classe selon la technologie .NET.

Éléments reliés
MarshalByRefObject via une relation de généralisation. Ceci indique qu'un composant .NET est une classe jouissant de plusieurs autres caractéristiques.

CompRef et ResourceReference
Sémantique
Les objets de ces deux classes représentent des références à d'autres MBR et ressources et donnent des informations sur ces derniers.

ILeaseObject
Sémantique
Un objet ILeaseObject est associé à une instance d'un composant. Il encapsule un ensemble de valeurs de champs que le processus hébergeant utilise pour gérer la durée de vie de cette instance.

Éléments possédés (attributs)

- CurrentLeaseTime: sa valeur représente un laps de temps de type *TimeSpan* selon la technologie .NET. C'est le temps restant pour une instance.
- CurrentState: c'est un attribut dont la valeur dépend d'une énumération définie par le type LeaseState exprimant l'état du cycle de vie d'une instance et dont les membres sont : Active, Expired, Initial, Null et Renewing.
- InitialLeaseTime: de type TimeSpan, cet attribut donne la durée de vie dont une instance dispose au moment de sa création. Cette durée diminue avec le temps et dépend de plusieurs autres évènements.

- RenewOnCall: de type TimeSpan, cet attribut indique l'extension de la durée de vie restante d'une instance suite à une invocation d'une méthode sur cette instance.
- SponsorshipTimeout: de type TimeSpan, cet attribut indique un délai d'attente d'une réponse selon l'explication suivante. Un client d'un MBR peut intervenir explicitement pour empêcher une finalisation d'une instance de MBR qui le dessert; il peut créer un objet appelé sponsor pour cette tâche. Il en résulte qu'avant de finaliser une instance d'un composant, le processus hébergeant doit notifier le sponsor de son client et attendre une réponse durant un temps égal au SponsorshipTimeout.

Éléments reliés
MarshalByRefObject : ILeaseObject fournit les valeurs des paramètres qui conditionnent la gestion de la durée de vie d'une instance d'un composant héritant de la classe MBR de .NET.

NetApplication
Sémantique
Une application .NET peut héberger ou être cliente d'un (ou plusieurs) composant; l'application doit le déclarer par configuration en utilisant un fichier de configuration, ou par programmation.

Éléments reliés
MarshalByRefObject et ConfigurationFile.

ServicedComponent
Sémantique
La classe ServicedComponent (SC) hérite de MBR, mais elle est caractérisée par son aptitude à profiter de .NET Enterprise Services (appelées aussi COM+ services) [9].

Notons que SC hérite de ContextBoundObject dont les instances gardent toujours un contexte de services constant. Dans la suite et en absence de possibilité de confusion, on désigne par SC ou ServicedComponent, un composant dont la classe d'implémentation hérite de la classe ServicedComponent.

Éléments reliés
ComPlusApplication

COMplusApplication
Sémantique
Une application COM+ sert à configurer à la fois plusieurs composants dont le type serveur hérite de ServicedComponent (SC) et qui partagent des services communs. Généralement, ces composants font partie d'une même application.

Éléments possédés (attributs)
- name: c'est une chaîne de caractères désignant le nom de l'application COM+ à laquelle fera référence un SC pour y être associé.

Éléments reliés
ServicedComponent: l'association de ComPlusApplication à cet élément traduit le fait qu'un type serveur héritant de ServicedComponent doit être registré auprès d'une application COM+ pour profiter d'un ensemble de services COM+.

Component
Sémantique
Cet élément représente la classe Component de .NET qui, en plus d'hériter de MarshalByRefObject, est caractérisée par son appartenance logique à un conteneur, et par ses méthodes virtuelles Dispose( ) et Finalize( ) pour relâcher les ressources utilisées avant de détruire une instance.

Éléments reliés
NetApplication : représente une application .NET jouant le rôle d'hébergement ou de client.

ConfigurationFile
Sémantique
Un fichier de configuration associé à une application .NET et qui renferme les informations nécessaires à la configuration d'hébergement ou d'appel d'un composant MarshalByRefObject. Il spécifie, entre autre, le port d'écoute pour les appels et les modes d'activation des composants.

Éléments possédés
Tous les autres éléments du diagramme de configuration expliqué ci-après. Ces éléments représentent les éléments XML du fichier de configuration.

4.2.2. Diagramme de configuration d'applications

Ce diagramme concerne les éléments qui forment la configuration d'une application en vue d'une déclaration d'enregistrement d'hébergement et d'appel d'un composant dérivé de MarshalByRefObject. Ces informations de configuration sont stockées dans un fichier de configuration de l'application, au format XML. Ce fichier est représenté par l'élément ConfigurationFile qui contient tous les éléments du présent diagramme.

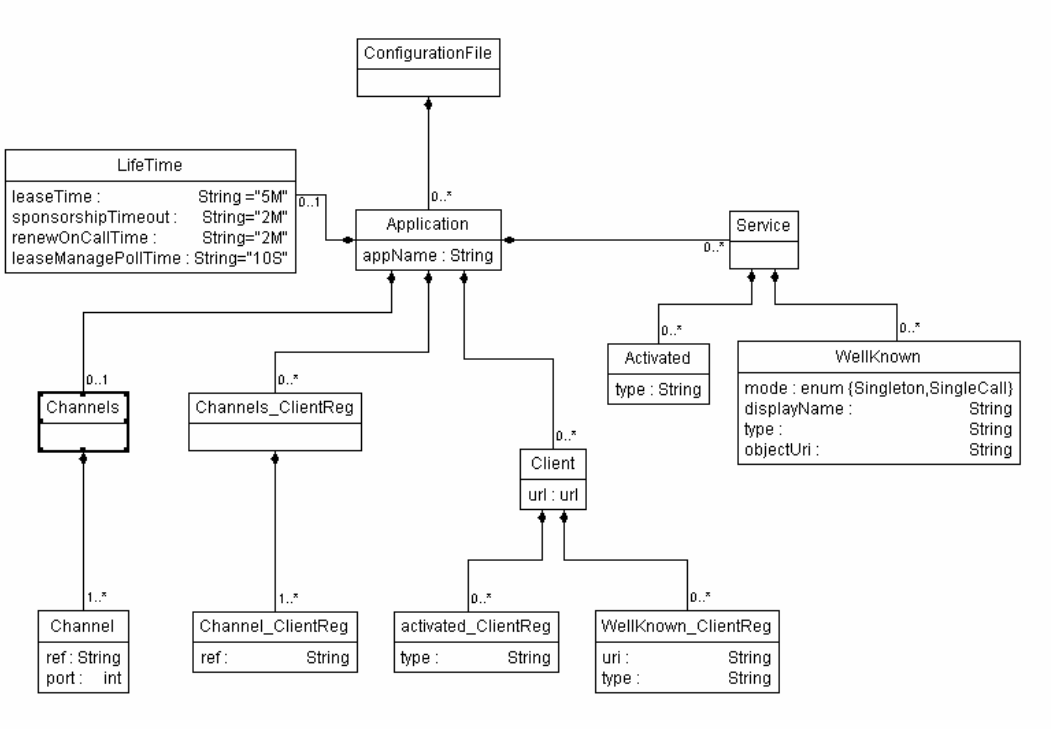


Figure 4.4 - Diagramme de configuration d'une application .NET

Application
Sémantique
Ce concept représente un élément XML appartenant au fichier de configuration d'une application.

Éléments possédés

- appName: cet attribut spécifie le nom de l'application .NET à utiliser dans le l'URL d'activation d'un composant hébergé.

À l'exception de ConfigurationFile tous les autres éléments du diagramme sont possédés par l'élément Application.

Service
Sémantique
L'élément service dans le fichier de configuration déclare l'hébergement et comprend la configuration de cet hébergement pour les composants.

Éléments possédés
Activated et WellKnown.

Activated
Sémantique
Pour chaque composant hébergé selon le mode Client-Activated, le fichier de configuration contient un élément XML appelé activated.

Éléments possédés (attributs)

- type : c'est un string qui donne le nom de la classe du composant et le nom de l'assembly contenant cette classe. Ce string permet à l'application d'identifier la classe implémentant le composant.

WellKnown

Sémantique

L'élément WellKnown dans le fichier de configuration déclare l'hébergement d'un composant par l'application selon une activation de type Server-Activated qui donne à l'application hôte le contrôle du cycle de vie d'une instance du composant.

Éléments possédés (attributs)

- uri : Uniform resource identifier.
- type : voir l'explication de l'élément Activated.
- displayName : donne un nom utilisé seulement par les outils de configuration d'interaction entre objets distants selon la technologie .NET.
- mode : permet de choisir le mode d'activation SingleCall ou Singleton.

LifeTime

Sémantique

L'élément XML nommé LifeTime du fichier de configuration spécifie les valeurs des paramètres de contrôle de la durée de vie d'un composant hébergé par l'application .NET.

Éléments possédés

leaseTime, sponsorshipTimeout, renewOnCallTime et leaseManagePollTime.

Tous ces attributs sont de type String. Toute valeur de ces attributs est formée d'une chaîne de caractère représentant un nombre entier positif avec un suffixe qui est l'une des lettres M ou S; ces lettres indiquent des minutes ou des secondes respectivement.

La sémantique des trois premiers attributs est la même que celle des attributs de l'element IleaseObject (voir diagramme principal). Mais ici ces attributs affectent tous les composant hébergés qui n'ont pas écrasé, par programmation, les valeurs de ces attributs (dans un objet ILeaseObject propre à chaque composant).

Le dernier attribut leaseManagePollTime indique la fréquence de vérification du temps de vie restant (lease time) pour une instance d'un composant. Cette vérification est assurée par le domaine d'application de .NET.

Channel de Channels

Sémantique

Le fichier de configuration spécifie un protocole de communication et éventuellement le numéro de port qui permettent à l'application d'écouter ou de lancer des appels à des composants, dans un élément XML appelé Channel. Les objets de l'élément Channel de notre métamodèle représentent ces éléments XML et sont contenus dans un objet de l'élément Channels.

Éléments possédés

- ref : une chaîne de caractères donnant une référence sur un protocole de communication.
- port : il représente un attribut entier qui donne le numéro de port à utiliser par l'application .NET pour écouter des appels pour les composants qu'elle héberge.

Client et ses éléments

Sémantique

Un élément XML Client déclare que l'application pourra faire appel à un ou plusieurs composants en spécifiant leurs modes d'activation, url et types.

Éléments possédés

- url : c'est une chaîne de caractères spécifiant le protocole, l'adresse de la machine hôte et le numéro de port à utiliser pour l'accès au composant distant appelé.
- activated_ClientReg : il représente un élément XML qui spécifie le mode d'activation Client-Activated du composant et donne la chaîne de caractères dite type qui identifie le nom de sa classe d'implémentation et de son assembly.
- WellKnown_ClientReg : idem que l'élément précédent, avec un mode d'activation Server-Activated et l'uri identifiant la classe du composant.

4.2.3. Diagramme du service COM+ de sécurité

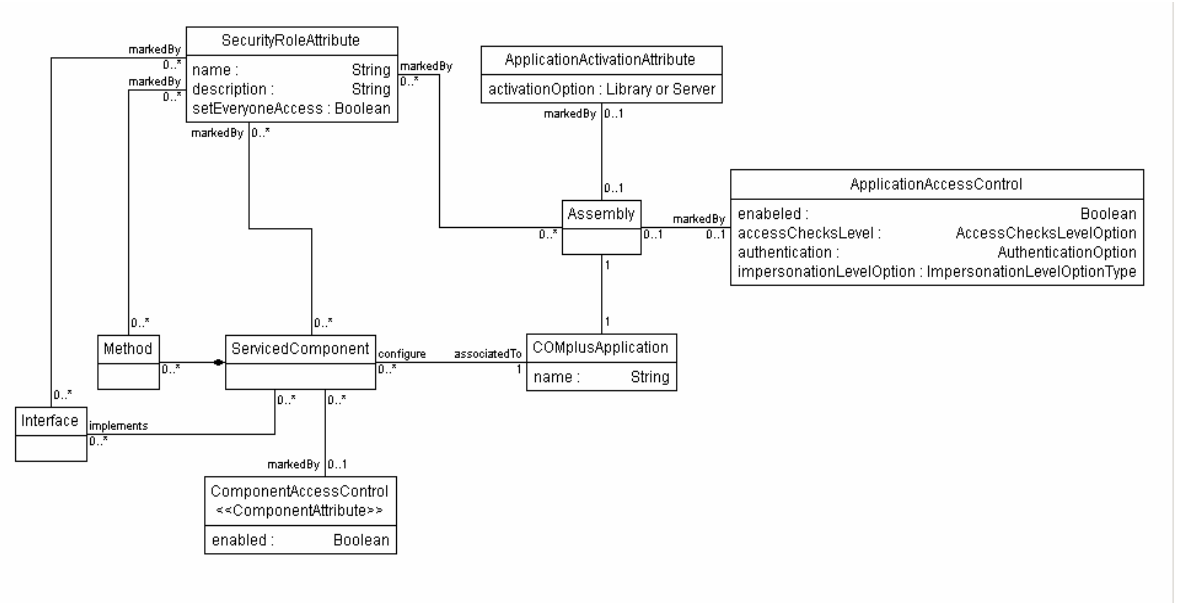


Figure 4.5 - Diagramme du service COM+ de sécurité

Ce diagramme illustre le fait qu'un ServicedComponent associé à une application COM+ peut profiter d'un service de sécurité défini au niveau de l'application COM+.

Assembly
Sémantique
Selon .NET, un assembly représente l'ensemble des fichiers résultant de la compilation d'une application ou d'une librairie de classes. Dans ce diagramme l'élément assembly est présent pour indiquer que des .NET Attributes (ApplicationActivationAttribute) sont des Attributes de .NET appliqués à un composant avec une syntaxe utilisant un suffixe «assembly». Ceci aura pour effet que ces Attributes affectent l'application COM+ à laquelle est associée le composant. Cet attribut sera dit assembly Attribute et il affectera la configuration des services qui concerne tous les composants associés à cette application COM+.

Éléments reliés
COMplusApplication, SecurityRoleAttribute, ApplicationActivationAttribute et ApplicationAccesControl.

SecurityRoleAttribute
Sémantique
L'application d'un objet SecurityRoleAttribute comme assembly Attribute affecte l'application COM+ et y définit un rôle de sécurité gérable par les services COM+ pour contrôler l'accès aux composants associés à cette application COM+.

Éléments possédés (attributs)
- name: donne le nom de rôle de sécurité.
- description: une chaîne de caractères décrivant non formellement des usages et privilèges du rôle.
- setEveryoneAccess: un booléen qui, mis à la valeur *true,* désactive le control d'accès relié à l'instance en question de SecurityRoleAttribute. Ceci est généralement utile pour une phase de débogage.

Éléments reliés
Assembly : Un rôle de sécurité est défini au niveau de l'application COM+ en utilisant l'attribut SecurityRoleAttribute comme un assembly Attribute.
Lorsqu'associé à Interface, ServicedComponent ou Method, SecurityRoleAttribute implique une autorisation de ce rôle de sécurité.

ApplicationActivationAttribute
Sémantique
Un ApplicationActivationAttribute représente un assembly Attribute applicable à un composant. Il spécifie la manière suivant laquelle les composants associés à l'application COM+ seront activés. Il représente un choix entre deux activations dites *Library* ou *Server*.

Éléments possédés (attributs)
activationOption: qui est une énumération {Library , Server}.

Éléments reliés
Assembly.

ApplicationAccesControl
Sémantique
Un ApplicationAccessControl appliqué comme assembly Attribute affecte l'application COM+ en conditionnant le niveau de son contrôle d'accès.

Éléments possédés
- enabled **:** De type booléen, il représente un attribut dont la valeur active ou désactive le contrôle d'accès. La valeur par défaut est *true*.
- accessChecksLevel : Permet d'activer et de choisir le niveau du contrôle d'accès à l'application ou au composant. Par défaut l'attribut est mis à ApplicationComponent.
- authentification : C'est une énumération des valeurs suivantes : None, Connect, Call, Packet, Integrity, Privacy, Default.
  Chacune des valeurs ci-dessus implique un certain niveau de vérification d'identité, de non-altération, de manque et d'encryptage de messages.
- impersonationLevelOption : C'est une énumération des valeurs suivantes :
  Anonymous, Identify, Impersonate, Delegate, Default.
  Ce dernier attribut sert à spécifier le niveau de confiance entre client et composants d'une application COM+ [34].
  .

Éléments reliés
Assembly

Interface et Method

Ces deux éléments représentent respectivement une abstraction d'une interface et d'une méthode de la technologie .NET.

ComponentAccessControl
Sémantique
L'attribut ComponentAccessControl appliqué à un SC active une vérification de privilège d'accès chaque fois qu'un appel au composant associé est lancé.

Éléments possédés (attributs)
enabled : un attribut Booléen qui, mis à *true,* active le service de contrôle d'accès au niveau du composant.

Éléments reliés
ServicedComponent

4.2.4. Diagramme Component COMplus services

Ce diagramme représente les principaux services COM+ [9] offerts à un ServicedComponent (SC).

Notons ici la structure étoilée de ce diagramme ayant pour centre l'élément ServicedComponent qui est relié à chacun des autres éléments avec un rôle d'application d'un service qu'on nomme une marque (*mark)*.

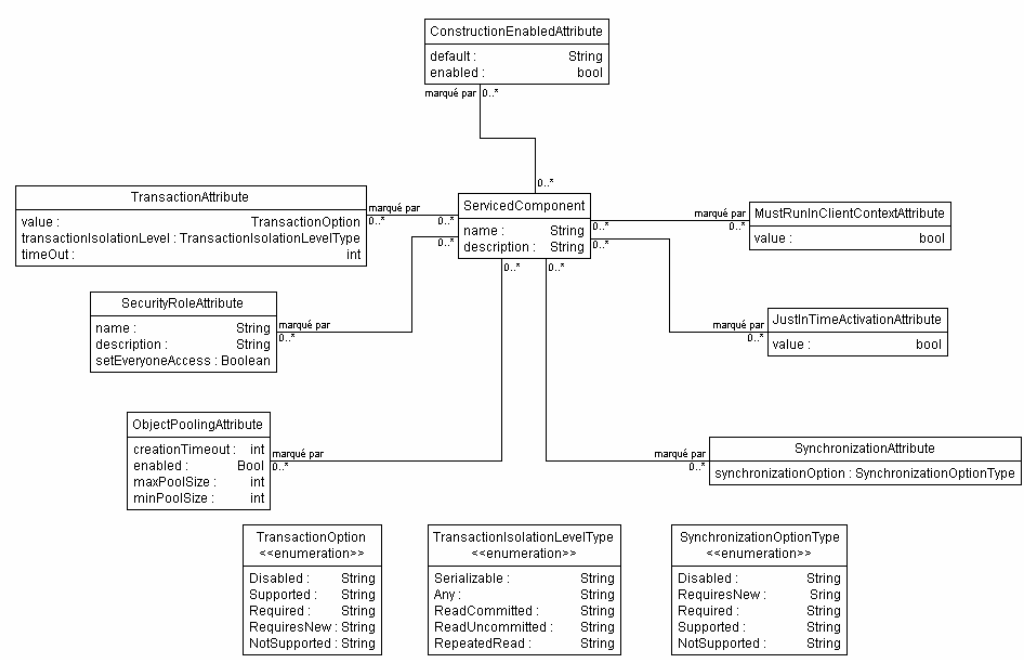


Figure 4.6 - Diagramme des services COM+ liés directement à un ServicedComponent

TransactionAttribute

Sémantique

Un objet de type TransactionAttribute peut marquer un ServicedComponent pour lui offrir un service de gestion de transaction spécifique.

Éléments possédés

- transactionIsolationLevel qui est une énumération servant à un choix de mécanisme d'isolation propre au traitement transactionnel et au verrouillage de données.
  Cette énumération comprend: Any, ReadCommitted, ReadUncommitted, RepeatableRead et Serializable.
- value : c'est une énumération permettant la gestion et la conciliation des transactions des clients et composants. Cette énumération est formée de :
  Disabled, NotSupported, Required, RequiresNfew et Supported.
- timeOut : utilisé pour fixer le timeout d'une transaction.

ObjectPoolingAttribute

Sémantique

Un attribut ObjectPoolingAttribute associé à un ServicedComponent lui permet de profiter du service de *pooling* de COM+.

Cet attribut sert à gérer le *pooling* des instances d'un composant en spécifiant le nombre maximal et minimal de ces instances dans le pool. Le rôle du pool est de garder en réserve des objets prêts à fournir aux futurs appelants. Notons qu'une instance de ServicedComponent sera retournée au pool dès que le client invoquera DisposeObject().

Éléments possédés

- creationTimeout: de type int. Il donne en millisecondes le temps maximal qu'un client peut attendre quand il demande un objet du pool.
- enabled: de type booléen. Cet élément sert à activer ou désactiver le service de pooling.
- maxPoolSize: de type int. Il spécifie le nombre maximal d'instances dans le pool; quand ce nombre est atteint, une instance relâchée par son client sera candidate au ramasse-miettes.
- minPoolSize : de type int. Le service de pooling doit créer des instances de ServicedComponent pour garantir ce nombre minimum d'instances disponibles.

ConstructionEnabledAttribute

Sémantique

L'attribut ConstructionEnabled permet de passer une chaîne de caractères à utiliser par la méthode construct( ) appelée par le constructeur du ServicedComponent. Notons qu'un constructeur de ServicedComponent n'accepte jamais de paramètres.

Éléments possédés

- enabled : de type booléen. Cet attribut mis à true fait que le constructeur du ServicedComponent invoque la méthode construct( ). Autrement, le constructeur n'invoque pas cette méthode.
- default : c'est un String qui donne une valeur à constructString visible par la méthode construct( ).

MustRunInClientContextAttribute

Sémantique

L'attribut MustRunInClientContext appliqué au composant avec le paramètre *true* ou sans paramètre, implique que l'activation du composant doit se faire dans le contexte de son créateur. Notons qu'un contexte signifie l'ensemble des services de temps d'exécution applicables à un composant.

Éléments possédés (attributs)

- value : un booléen qui, mis à *false*, permet de désactiver ce service.

JustInTimeActivationAttribute

Sémantique

L'application de l'attribut JustInTimeActivation (JITA) à un ServicedComponent(SC) avec un paramètre *true* ou sans paramètre permet de désactiver une instance du ServicedComponent par exécution d'une instruction placée à la fin d'une de ses méthodes. La désactivation libère les ressources associées à un objet et le rend candidat au ramasse-miettes ou pour le pooling.

Notons que l'usage simultané de JITA et du pooling fait que l'état d'un objet sortant du pool, sera toujours réinitialisé et ne portera pas le contexte du dernier appelant.

Éléments possédés
value : de type booléen. Sa valeur par défaut est *true* et implique la mise en marche du service JITA pour le SC.

SynchronizationAttribute
Sémantique
Cet attribut détermine les propriétés du service de synchronisation à offrir au SC.

Éléments possédés
value : c'est une énumération formée des valeurs suivantes :
Disabled, NotSupported, Required, RequiresNew, Supported.
La valeur de cet élément détermine la politique de gestion de la synchronisation [34].

4.2.5. Information complémentaire aux diagrammes

En plus des éléments figurant dans les diagrammes précédents, ajoutons l'élément nommé Attribute qui représente le concept de Attribute de la technologie .NET et qui est associé à ServicedComponent, COMplusApplication, méthode et interface avec le nom de rôle *mark*. Notons que Attribute dérive de NetClass.

Ajoutons aussi que ApplicationAccessControl, ComponentAccessControl et tous les autres éléments du métamodèle dont le nom se termine par le suffixe "Attribute'' dérivent de l'élément Attribute.

Un autre élément à ajouter, comme élément du métamodèle de .NET, est NetDefinedAttribute qui dérive de Attribute et qui possède un attribut dont le nom est netType de type String. NetDefinedAttribute représente un Attribute défini dans la librairie de classes de .NET et son attribut netType permet d'identifier quelle classe de la librairie de .NET, une instance de cet élément représente.

4.3. Discussion du métamodèle

Le but principal derrière la création d'un métamodèle est de permettre de modéliser des systèmes appartenant à un certain domaine. Donc c'est plutôt un langage qu'on peut évaluer en fonction de sa richesse en vocabulaire approprié et des règles qui le structure. Tout élément d'un modèle contribue à la description d'un système en profitant de la sémantique d'un élément du métamodèle. Il en est de même pour toute association d'éléments dans un modèle. Le pouvoir expressif des modèles dépend de ce que représentent les éléments du métamodèle.

Concernant notre métamodèle, on a tenté de se conformer à ce qui précède en respectant les critères suivants de bonnes pratiques dans l'art de la métamodélisation :

- Couvrir les concepts : Ce métamodèle fait ressortir les éléments architecturaux de la technologie des composants .NET, leurs interactions de haut niveau ainsi que la structure globale de leurs interactions [17]. Le tableau suivant cite les concepts technologiques et montre leur correspondance aux éléments de notre métamodèle.

Concepts		Éléments du métamodèle
Accès à distance	Une référence vers une instance distante d'une classe exposant une interface pour accomplir des traitements distribués.	MarshalByRefObject
	Usage des Channels (ports, protocoles) pour communiquer des messages.	Channel et Channel_ClientReg.
Sécurité	Utilisation des rôles pour un niveau d'authentification ajustable à : application, composants, méthodes et interfaces.	SecurityRoleAttribute,
	Encryptage, contrôle d'intrusion, d'usurpation d'identité et d'intégrité des messages.	ApplicationAccessControl et ComponentAccesscontrol
Gestion des instances des composants	Pooling, passivation des instances inactives, durée de vie et différents modes d'activation.	ObjectPoolingAttribute, JITA, ILesaseObject, LifeTime, Single-Call et Singleton.
Traitement transactionnel	Gestion du comportement transactionnel entre l'appelant et le composant.	TransactionAttribute
Gestion de la concurrence	Notion d'activité et de causalité spéciale au service de synchronisation de COM+.	SynchronizationAttribute

Tableau 4.1 - Principaux concepts capturés par le métamodèle

- Faire ressortir les relations associant les divers éléments du domaine.
- Éviter la redondance : Si la présence de beaucoup de synonymes est un point fort pour un langage naturel, ce n'est pas le cas pour les métamodèles. Considérons par exemple les deux éléments ILeaseObject et LifeTime qui, quoique manifestement semblables, sont loin de représenter la même sémantique. Le premier conditionne une stratégie de gestion de durée de vie au niveau d'une application hôte alors que le deuxième est associable à une instance, et son état (état de l'objet IleaseObject) conditionne sa durée de vie.

- Ne pas reprendre les éléments de base déjà offerts par le métamodèle de UML car le métamodèle peut être complété par une extension de UML définissant son propre profil.

Ainsi notre métamodèle est supposé offrir un langage approprié pour modéliser la technologie ciblée, et au besoin, il peut intégrer des éléments de UML neutres par rapport aux technologies.

Notre métamodèle est une proposition qui n'est pas émise ou validée par les auteurs de la technologie des composants .NET, mais il pourra tout de même être utile pour les raisons suivantes :

- Il comble l'absence de métamodèle pour la technologie ciblée, en lui offrant un langage de modélisation utilisant ses propres termes.
- Il est indispensable à l'élaboration d'un générateur de code.
- On espère qu'il va stimuler un effort de normalisation d'un métamodèle pour la technologie des composants .NET qui est l'une des exigences les plus urgentes du cadre de travail de MDA.

À noter que ce métamodèle à reçu un feedback très favorable auprès de la communauté scientifique oeuvrant dans le domaine de la métamodélisation et de MDA [1].

4.4. Conclusion

Le métamodèle proposé ouvre la porte à d'autres initiatives de métamodélisation ciblant les différentes branches de la technologie .NET. Il sera d'une valeur pédagogique dans la mesure où il pourra aider à introduire rapidement l'architecture des composants .NET.

Simple et incluant l'essentiel de la technologie visée, ce métamodèle permet de décrire l'implémentation d'une application basée sur les composants .NET. Il peut être considéré comme un premier pas vers un métamodèle normalisé comparable à celui des Enterprise JavaBeans [21]. Cette normalisation de métamodèle est essentielle pour profiter des avantages de MDA car c'est elle qui fournit un langage d'écriture standard des PSMs selon .NET. Cette normalisation est indispensable à la définition des transformations standardisées permettant de passer d'un PIM vers un PSM ciblant la technologie des composants .NET dans un processus de développement conforme à MDA.

Chapitre 5

Définition de la transformation

Ayant à notre disposition les métamodèles de EJB et des composants .NET, nous allons définir, dans ce chapitre, un chemin de migration horizontal entre ces deux technologies par la définition d'une transformation qui permet de passer de tout modèle spécifique aux EJB (c'est à dire un PSM) à un modèle spécifique aux composants .NET. Nous allons utiliser un langage bien défini pour exprimer l'ensemble des règles qui constituent la définition de la transformation. La définition des règles de transformation entre modèles se fera au niveau des métamodèles.

5.1. Langage d'expression et notations

Pour chacune des deux technologies considérées (EJB et .NET), chaque élément d'un PSM est une instance d'un élément unique du métamodèle de cette technologie. Nous allons donc spécifier chaque élément mis en jeu dans une règle de transformation en se référant à son type dans le métamodèle correspondant et en imposant une condition de filtrage portant sur les propriétés de cet élément et du modèle PSM dans lequel il figure.

5.1.1. Le langage utilisé

La transformation qui convertit un PSM spécifique aux EJB en un PSM spécifique à .NET consiste en un ensemble de règles. Pour écrire ces règles, nous utilisons le langage de transformation de modèles Open QVT [22] et son implémentation par la syntaxe concrète de l'outil ATL [33] qui à son tour supporte le langage normalisé *Object Constraint Language* [38].

Rappelons qu'Open QVT est une soumission de proposition en réponse à l'appel à des propositions (RFP) [20] émis par OMG en 2002. Cette proposition et son implémentation par ATL ont été élaborées par un groupe de compagnies internationales et d'universités françaises pionnières dans le domaine.

Au niveau de la syntaxe de base d'ATL, notons:

- L'analyseur d'ATL est sensible à la casse, au niveau des noms de dossiers et du code source.
- Le symbole "!" indique un opérateur de résolution (scope) qui se place entre le nom d'un élément et le nom de son métamodèle.
- Les lignes de commentaires sont identifiés par deux tirets "--".
- Chaque règle doit avoir un nom unique dans son champ de visibilité.
- Une attribution de valeur est effectuée avec l'opérateur de flèche gauche "<-".
- Deux attributions de valeurs, côte à côte, sont séparées par une virgule ",".
- Le symbole " # " au début d'une chaîne de caractères indique que cette chaîne fait référence à un membre d'une énumération.
- Pour se diriger (naviguer) d'un élément vers son attribut, écrire le nom de l'élément, puis "." suivi du nom de l'attribut.
- Quand deux éléments sont associés par un lien, instance d'une association définie dans le métamodèle, on navigue de l'un vers l'autre en écrivant le nom de l'un suivi d'un point et du nom de rôle de l'autre. Ce nom de rôle est généralement spécifié dans le métamodèle, sinon le nom de l'élément ciblé (vers lequel on navigue) peut être utilisé comme nom de rôle mais en le commençant par une lettre minuscule.
- Une relation de composition donne implicitement un nom de rôle 'owner' à l'élément composant.
- OCL est utilisé pour exprimer des conditions de filtrage.
- La concaténation de chaînes de caractères est représentée par l'opérateur +
- Les valeurs littérales sont comprises entre guillemets : " "

La syntaxe de la définition d'une règle de transformation est de la forme suivante :

```
    rule R {
    from e : nom-meta-entrée ! el-e (cond)
    to   s : nom-meta-sortie ! el-s
    ( -- séquence d'attribution de valeurs pour
      -- peupler l'élément créé
      -- ex. title<- e.title, nom<- e.name+"nouveau"
    )
}
```

Avec:

R: nom de la règle.

nom-meta-entrée : nom du métamodèle du modèle d'entrée.

el-e: nom d'un élément du métamodèle. Les instances de cet élément subiront, à tour de rôle, l'application de la règle R.

e: nom de variable locale référençant l'instance de el-e (en cours de traitement par la règle R) rencontrée dans le modèle d'entrée.

cond: condition de filtrage sur les instances d'entrée à la règle.

nom-meta-sortie : nom du métamodèle du modèle de sortie.

s: nom de variable locale référençant l'instance créée en sortie de la règle.

el-s: nom de l'élément du métamodèle dont il faut créer une instance dans le modèle de sortie, chaque fois qu'une instance d'entrée nommée e est rencontrée.

En plus des règles, on peut spécifier des actions à initier au début de la transformation ou au début d'une règle particulière. De telles actions sont définies dans des blocs de code avec en-tête *init*. À noter qu'*init* ainsi que d'autres éléments du métamodèle de Open QVT ne sont présentement pas directement supportés dans ATL.

Les notations suivantes aideront à simplifier l'expression des règles :
EJB : le métamodèle des EJB.
Java : le métamodèle de Java.
Net : le métamodèle des composants .NET.
in : le modèle d'entrée.
out : le modèle de sortie.

Selon OpenQVT, la déclaration " Net::UseDefaultValue := true " implique que la création d'un élément provoque la création de tous ses éléments possédés avec une initialisation suivant les valeurs par défaut.

De plus, on utilisera les noms d'instances qui suivent les mots clés *from* et *to* comme noms de variables visibles dans la même règle où elles sont déclarées. Les noms des instances qui représentent des noms de variables seront écrits en italique dans le texte pour les mettre en relief.

5.1.2. Correspondance entre les éléments de base du métamodèle de EJB et ceux de .NET

En définissant la transformation de EJB vers .NET, on se contente de toucher à un nombre minimum d'éléments du métamodèle de Java indispensable à la transformation partant du métamodèle EJB. On ne rentrera pas dans les détails de correspondance entre les éléments de base de Java (Class, Field, Method, Interface, Parameter) et ceux de .NET, car les nuances à ce niveau entre ces deux technologies, sortent du cadre de notre étude. A titre d'exemple, on adopte les valeurs par défaut des *modifiers* (public, protected, static, etc.) pour Field, Method et Class de .NET, au lieu de se lancer dans une correspondance entre les *modifiers* de Java et ceux de .NET. Ainsi, à une classe de Java on fait correspondre une classe publique de .NET qui n'est ni *final* ni *abstract* car ces deux dernières valeurs ne représentent pas le choix par défaut selon .NET.

Pour la simplicité et en raison de la coïncidence sémantique des concepts de classe, de champ, de type, et de paramètre dans les deux mondes (Java et .NET), on va se contenter du métamodèle de Java pour fournir le langage d'expression pour leurs contreparties dans .NET. Bien que pour éviter n'importe quelle confusion, le type de classe de Java s'appelle JavaClass, et le type de classe de .NET sera appelé NetClass. En outre, les types primitifs de Java ont les mêmes noms dans .NET, à l'exception du type booléen de Java qui s'appelle bool dans .NET.

5.2. Les règles de transformation

Les règles sont expliquées en pseudo code du langage ATL déjà introduit et ce en vue de rendre les règles compréhensibles. Nous utilisons le symbole = pour signifier une affectation de valeur; et lorsque l'égalité se situe entre deux instances appartenant à différents métamodèles, elle implique que la première se voit affectée la valeur de la transformée de l'autre.

Par la suite, les règles sont exprimées dans la syntaxe exacte d'ATL pour, d'une part donner une illustration de cette syntaxe et, d'autre part donner une forme générique à ces règles facilitant leur implémentation dans tout autre outil ou langage.

Règle 1. COMplusApplication. : Cette règle fournit les éléments représentant l'application COM+. L'association à cette application est obligatoire pour permettre à un composant de profiter des services COM+. Cette règle est déclenchée si le modèle source contient une instance de EJBJar représentant un assemblage de composants EJB.

Si le modèle de départ contient au moins une instance de type EJBJar, on crée dans le modèle de sortie une instance de COMplusApplication qu'on note *Comp* avec :

- L'attribut name de *Comp* sera mis à "Comp"
- On crée une instance *appAct* de l'élément ApplicationActivationAttribute avec son attribut activationOption mis à Server.
- on crée une instance as de Assembly. *as* est relié à *Comp* et à *appAct*.
- On crée un élément *appAc* instance de ApplicationAccessControl et on l'associe à as. Il sera peuplé des attributs suivants :
  - enabled = true
  - accessCheckLevel = ApplicationComponent
  - authentication = Connect
  - impersonationLevelOption = Default

```
rule R1 {
   from jar : EJB!EJBJar  ( jar = EJBJar.allInstances( )
   ->asSequence( )->first( ))
   -- le filtre utilisé permet de sélectionner seulement
   -- la première instance de EJBJar.
   to as : Net ! Assembly
```

```
    to  Comp : Net!COMplusApplication  (name <- “Comp”,
    Comp.assembly <-as) ,-- name équivalent à Comp.name
    to appAc : Net!ApplicationAccessControl  (enabled <-
    true,
     accesCheckLevel <- # ApplicationComponent ,
     authentication <- #Connect , impersonationLevelOption
    <- #Default , assembly <- as),
    to  appAct : Net!ApplicationActivationAttribute
    (activationOption <- #Server, assembly <- as)
 }
```

Règle 2. Application, LifeTime, Channels, Channel et Service : Cette règle fournit les éléments représentant l’application qui héberge. Cette application est nécessaire pour écouter les appels à un composant. L’application de cette règle est déclenchée si le modèle source contient au moins un élément indiquant la présence de composants EJB qui seront transformés en composants .NET.

Si le modèle de départ contient au moins une instance de type EJBJar, on crée et on initialise dans le modèle de sortie des instances de NetApplication, ConfigurationFile, Application, LifeTime, Channels, Channel et Service.

```
 rule R2 {
    -- créer les instances de Application, NetApplication
    -- et ConfigurationFile
    from jar : EJB!EJBJar  ( jar = EJBJar.allInstances( )
    ->asSequence( )->first( ))
    -- le filtre utilisé permet de sélectionner la
    --première instance de EJBJar, si elle existe.
    to app : Net!NetApplication  ( ) ,
    to conf : Net!ConfigurationFile  (configure <- app),
    to appElement : Net!Application  (owner <- conf),
    to chs : Net!Channels   (owner <- appElement),
    to  ch :  Net!Channel   (owner <-  chs,  ch.ref <-  “tcp”,
    ch.port <- 8010),
    -- 8010 est un choix arbitraire qui peut être modifié
    -- selon la convenance de celui qui exécute la
    -- transformation.
    to lf : Net!LifeTime  (lf. LeaseTime <- “1 H”,
    -- 1 H indique une heure; c’est un choix de valeur
    –-par défaut qui peut être modifié selon la
    --convenance du    développeur.
        lf.SponsorshipTimeout <- “1 H”,
        lf.RenewOnCallTime <- “1 H”,
        lf.LeaseManagePollTime <- “1 H”,
        owner <- appElement),
    to serv : Net!Service  (owner <- appElement)
}
```

Règle 3. Bean Session avec état ou sans état : Cette règle fait correspondre à tout composant EJB de type Session un composant .NET de type ServicedComponent. Ainsi, pour chaque instance *S* de Session de EJB, elle crée :

- Une instance *SC* de ServicedComponent de .NET, avec :
  - name de *SC* correspondant au S.remoteInterface.name
  - Si *S*.sessionType = Stateful, l'attribut CanBePooled de *SC* est mis à false
  - Si *S*.sessionType = Stateless, l'attribut CanBePooled de *SC* est mis à true
  - *SC*.associatedTo = l'unique instance *Comp* de COMplusApplication déjà créée
  - SC.netApplication = app qui est l'unique instance de NetAppliation
- Une instance *opa* de ObjectPoolingAttribute telle que :
  - enabled = true
  - creationTimeout = 5. C'est une valeur arbitraire choisie pour initialisation.
  - maxPoolSize = 10. C'est une valeur arbitraire choisie pour initialisation.
  - minPoolSize = 0. C'est une valeur arbitraire choisie pour initialisation.
  - *opa*.mark = S
- Une instance *act* de Activated de .NET et on l'associe à l'unique instance de Service déjà créée comme suit :
  - *act*.service = l'unique instance de Service déjà créée par la règle 2
  - *act*.type = *SC*.name + "," + *SC*.name
- Une instance *syn* de SynchronizationAttribute de .NET avec :
  - *syn*.synchronizationOption <- #Required
  - *syn*.mark = SC

Ci-dessous, la formulation de cette règle et de la fonction *pooling* qu'elle utilise:

```
-- on commence par définir une opération sur les instances de Session.
helper context EJB!Session
def : pooling ( ) : Boolean  =  if sessionType =
#Statless  true  else false;

rule  R3 {
   from S : EJB!Session
   to SC : Net!ServicedComponent  ( canBePooled <-
   S.pooling( ), name <- S.remoteInterface.name,
      SC.associatedTo = Comp, SC.netApplication <- app),
   to opa : Net!ObjectPoolingAttribute  ( opa.enabled <-
   true, opa.creationTimeout <- 5,
```

```
      opa.maxPoolSize <- 10, opa.minPoolSize <- 0,
      opa.mark <- SC)
    to act : Net!Activated (owner <-Service.allInstances(),
    -- Une seule instance de Service est créée
    type <- S. remoteInterface.name + ",” +
      S. remoteInterface.name ),
    to  syn : Net!SynchronizationAttribute (
    syn.synchronizationOption <- #Required )
}
```

Règle 4. ContainerManagedEntity - Afin de créer un fragment de modèle dans .NET qui représente ce qu'est un bean entité dont la persistance est gérée par le *container* de EJB, on va appliquer une règle qui crée pour chaque instance de ContainerManagedEntity, un composant *SCE* de type ServicedComponent qui est au service des clients appelants, et qui déclare une classe sérialisable (ayant son nom avec le suffixe"_Serializable") contenant l'ensemble des champs dont on a à gérer la persistance. Une instance *att* de Attribute de .NET contiendra les informations sur les champs clés primaires, et sera associée au composant *SCE* pour lui fournir ces informations.

La gestion de la persistance est alors aisée en utilisant un SGBD (ex. : SQLServer) supportant la gestion de persistance des objets de classes sérialisables au format XML. Notons que selon la technologie .NET, on peut généralement sérialiser des objets, au format XML; on peut même les reconstruire à partir de leur format sérialisé. Le SGBD fournit en réponse à des requêtes, des objets au format XML qui nous permettent d'interroger un ensemble d'objets sérialisés comparables à un ensemble de beans entités de EJB.

Les correspondances expliquant cette règle sont comme suit :

Pour chaque instance *ent* de ContainerManagedEntity de EJB on fait correspondre :

- Une instance *ESerial* de Netclasse avec
  - Création d'une instance *netat* de NetDefinedAttribute
  - *ESerial*.markedBy = *netat*
  - *ESerial*.*netat*.netType = "Serializable"
  - name = ContainerManagedEntity.remoteInterface.name + "_Serializable"
  - Notons que dans la règle 6 on aura : pour tout membre de *ent*.persistentFields ou *ent*.keyFields, créer un *nf* instance de NetField tel que *nf*.owner =*ESerial*
- Une instance *SCE* de ServicedComponent avec
  - name = ContainerManagedEntity.remoteInterface.name
  - SCE.netApplication = app qui est l'unique instance de NetAppliation
  - CanBePooled <- true
  - *SC*.associatedTo = l'unique instance *Comp* de COMplusApplication déjà créée
  - Création d'une instance *act* de Activated de .NET avec :

- *act*.Service = l'unique instance de Service déjà créée par la règle 2
- *act*.type = *SCE*.name + ",'' + *SCE*.name

- Une instance *opa* de ObjectPoolingAttribute avec initialisation des attributs et de l'association *mark* :
  - enabled: true
  - creationTimeout = 5
  - maxPoolSize =10
  - minPoolSize =0
  - opa.mark = SCE
- Une instance kNames de Attribute de Net avec :
  - Création d'une instance *f* de Field de Net
  - *f*.owner = KNames
  - *f*.name = "KeyFieldsNames"
  - *f*.type = String
  - Le champ créé *f* sera utilisé pour stocker l'ensemble des noms des champs clés primaires de l'entité. Cet ensemble de noms peut être représenté par la valeur:
    ent.keyFor ->iterate(*att* : Field, acc : String ="" | acc + *att*.name + ",")
  - *kNames*.mark =*SCE*
    On a créé ainsi un Attribute qui marque *SCE* et qui lui fournit les noms des champs clés primaires séparés par des virgules.
- Une instance *syn* de SynchronizationAttribute de Net avec :
  - *syn*.synchronizationOption = Disabled si ent.isreentrant = true ; autrement elle sera égale à Required
  - *syn*. mark = *SCE*

Règle 5. JavaClass – Cette règle implante la correspondance entre une classe de Java et une classe de .NET. Elle prend en charge les classes qui ne font pas partie des classes d'implémentation, ou des interfaces home ou distantes des composants EJB. Ces dernières classes d'implémentation et interfaces seront prises en charges par les règles traitant les instances de EnterpriseBean.

Pour chaque instance *jClass* de JavaClass de Java, on crée une instance *nClass* de NetClass à condition qu'il n'existe aucune instance *eb* de EnterpriseBean telle que : *eb*.ejbClass ou eb.remoteInterface ou eb.homeInterface soit égale à *jClass*.

En d'autres termes, pour que cette règle soit applicable à une JavaClass il faut que cette dernière ne représente pas une classe d'implémantation ou une home interface ou une remote interface d'un composant EJB.

L'application de cette règle va créer une instance *nClass* de NetClass telle que :

- Les declaringClass de *nClass* correspondent au declaringClass de *jClass*
- Les declaredClass correspondent
- Correspondance de noms

Règle 6. Field - Cette règle implante la correspondance entre un field (champ) de Java et un field de .NET. Ainsi, Pour chaque instance *jF* de Field de EJB (en fait, de Java), on crée une instance *nF* de Field de .NET avec correspondance de name et de type.

De plus :

- Si *jF*.containerManagedEntity = *ent* / *ent* non null alors *nF*.owner = *jF*.ContainerManagedEntity (avec un filtre qui sélectionne la classe serializable produite par la transformation de l'instance ent de ContainerManagedEntity)
- Si *jF*.javaClass.enterpriseBean = *eb* alors *nF*.owner =*jF*.owner (avec un filtre qui choisit seulement une instance de ServicedComponent). Le Field *nf* est possédé (*owned)* par une instance de ServicedComponent.
- Si aucun des deux cas ci-dessus n'est applicable à *jF* alors *nF*.owner = *jF*.owner qui n'est autre qu'une instance de NetClass.

Règle 7. Method - Cette règle implante la correspondance entre une méthode de Java et une méthode de .NET. Elle traite différemment les méthodes appartenant à une des interfaces d'un composant et les méthodes d'une classe de Java dans le cas général.

Pour chaque instance *jMet* de Method de Java, cette règle crée une instance *nMet* de Method de .NET avec correspondance des noms et des paramètres avec leurs types.

De plus :

- Si *jMet*.owner est une remoteInterface d'une EnterpriseBean alors *nMet*.owner = *jMet*.owner.enterpriseBean (avec un filtre qui selectionne seulement l'instance de ServicedComponent)
- Si
  - *jMet*.owner est une homeInterface d'une EnterpriseBean

  alors
    - Le paramètre de retour sera de type void.
    - *nMet*.owner = *jMet*.owner (avec un filtre qui sélectionne seulement l'instance de ServicedComponent)
    - *nMet*.name = *jMet*.name et si *nMet*.name est alors = "create'' *nMet*.name sera = "initializeState''
- Si *jMet*.owner n'est pas en relation avec une instance de EnterpriseBean avec un rôle de homeInterface, remoteInterface ou ejbClass alors *nMet*.owner = *jMet*.owner (le owner de *nMet* est déjà créé dans la règle 5)

Règle 8. Gestion de transaction – Cette règle vise à transmettre les informations des besoins en service de gestion de comportement transactionnel. Ces besoins sont exprimés par le modèle EJB dans les instances de MethodTransaction et leur attribut transactionAttribute et seront traduits par des instances de TransactionAttribute dans le modèle cible.

Pour toute instance bean de Session dont la valeur de l'attribut transactionType est Container, on crée une instance ta de TransactionAttribute de .NET selon la règle suivante.

```
rule R8 {
   from  bean :  Sesssion!EJB  (     (transactionType  =
   #Container)             and       (bean.MethodElement->
   collect(transactionAttribute)   ->   asSet->select   (
   self!null)  ->    iterate(result  :  Integer  =  0  |
   result+1)= 1))
    to tr : TransactionAttribute!Net  ( value <-
      bean.MethodElement->asSequence( )->
      first().MethodTransaction.transactionAttribute.f(),
      -- on cible par cette attribution la première
      -- instance de l'ensemble des instances
      -- de MethodTransaction reliées à un même bean.
      -- f() est une fonction (un helper de ATL)  de
      -- transactionAttribute
      tr.mark <- bean
      -- un filtre sélectionne l'instance de SC à marquer
      -- par tr et ce pour associer tr à la bonne instance
      -- de ServicedComponent correspondant au bean.
      transactionIsolationLevel <- #Serializable,
      -- c'est un choix conservateur
      timeOut <- 2000  --  en millisecondes
    )
}
```

Cette règle utilise un filtre qui la rend applicable seulement au cas où transactionType = Container et où les différentes méthodes du même EnterpriseBean partagent le même attribut de transaction.

La valeur de tr.value est donnée par la fonction f (utilisée dans la règle ci haut) qui est définie suivant le tableau de correspondance suivant:

EJB	.NET
transactionAttribute	**TransactionAttribute.value**
NotSupported	NotSupported
Required	Required
Supports	Supported
RequiresNew	RequiresNew
Mandatory	Dans ce cas, f( ) retourne la valeur nulle et il faut produire un message signalant qu'il faut programmer le lancement d'une exception si un client invoque une méthode de SC sans que le client soit dans un contexte transactionnel.

Never	Dans ce cas, f( ) retourne la valeur nulle et il faut produire un message signalant qu'il faut programmer le lancement d'une exception si un client invoque une méthode de SC tout en étant dans un contexte transactionnel.

Règle 9. Security Role et MethodPermission - Cette règle transmet du modèle source au modèle cible, les informations sur les rôles de sécurité et leur permission sur les méthodes exposées par les composants.

Pour toute instance *srEjb* de SecurityRole de EJB, on crée une instance *srCom* de SecurityRole de .NET avec :

- *srCom*.name = *srEjb*.roleName
- *srCom*.description = *srEjb*.description
- *srCom*.setEveryOneAcces= false
- *srCom*.method correspond à *srEjb*.MethodPermission.method
- *srCom*.assembly.COMplusApplication = l'unique instance de COMplusApplication créée dans le modèle cible.

Règle 10. Références à d'autres composants – Cette règle traduit la dépendance entre composants EJB en dépendance entre les composants .NET correspondant dans le modèle cible.

Pour toute instance *eref* de EJBRef de EJB, on crée une instance *cref* de CompRef de .NET telle que:

- les descriptions correspondent
- *cref***.**type = *eref***.**remote ceci donne le nom de l'instance de ServicedComponent qui est référencée par cref dans le modèle cible.

De plus il faut associer *eref* à l'instance *SC* en spécifiant :

- *cref*.owner correspond à cref.ejb (avec un filtre qui choisit l'instance de SC).

Règle 11. Références à des ressources – Cette règle transmet au modèle cible les informations d'accès d'un composant EJB à des ressources dans son environnement d'exécution (ex. Fabrique de connexion à un SGBD).

Pour chaque instance *resejb* de ResourceRef de EJB, on crée une instance *resnet* de ResourceReference de .NET telle que :

- Les descriptions correspondent
- On définit une fonction f helper qui est fonction de resejb.link et qui en extrait le suffixe de link qui suit "//". Ensuite *resnet***.**serverAddress se verra affectée la valeur f(*resejb*.link)

- *resnet*.factoryType = g(*resejb*.type) où g est une fonction qui fait la correspondance entre un ensemble de types de *factories* de Java et des types de *factories* de .NET (à défaut de correspondance, g(x) sera égale à NetTypefor_java+x)
- *resnet***.**owner correspond à *resejb***.**ejb avec un filtre qui choisit l'instance de *SC* résultant de la transformation de *resejb*.ejb.

Le développeur aura à examiner les résultats de cette règle. Une mise au point des valeurs des attributs des instances ResourceReference est recommandée. Une telle tâche manuelle est généralement faisable car une application utilise normalement un nombre raisonnable de ressources moyennant un nombre plus petit de types de fabriques de ces ressources.

Règle 12. EnvEntry – Cette règle prend en charge les données fournies par le développeur à un composant EJB dans le fichier descripteur de déploiement, telles que les variables d'environnement accessibles au composant et ne faisant pas partie de son code. Elle réinjecte les variables d'environnement sous forme de .NET Attributes appliqués au composant .NET.

Pour toute instance *envent* de EnvEntry de EJB, on crée une instance *envatt* de Attribute de .NET telle que :

- Les composants correspondent : *envatt*.mark = *envent*.ejb (avec un filtre qui choisit l'instance de SC).
- Les noms correspondent : *envatt*.name = *envent*.name

L'ensemble des attributs de envent est à reproduire dans envatt :

- Créer une instance *f1* de Field avec *f1*.name = description; *f1*.owner = envatt ; *f1*.type = String; valeur initiale de *f1*= envent.description
- Créer une instance *f2* de Field avec *f2*.name = value; *f2*.owner = *envatt*; *f2*.type = *envent*.type (ceci implique le transformé de ce dernier par la règle suivante) ; valeur initiale de *f2* = *envent*.value;

Règle 13. Type de EnvEntry - Cette règle sert à convertir les types de variables d'environnement (EnvEntry.type) de EJB en types correspondants dans le métamodèle de .NET.

```
from  b :EnvEntryByte ! EJB (  ) to netb : byte ! Net ( )
from  s : EnvEntryShort ! EJB (  ) to nets : short ! Net ( )
from  i : EnvEntryInt ! EJB (  ) to netint : int ! Net ( )
from  l : EnvEntryLong ! EJB (  ) to netl : long ! Net ( )
from  f : EnvEntryFloat ! EJB (  ) to netf : float ! Net ( )
from  d : EnvEntryDouble ! EJB (  ) to netd : double ! Net ( )
from  b : EnvEntryBoolean ! EJB (  ) to netb : bool ! Net ( )
from  st : EnvEntryString ! EJB (  ) to netst : string ! Net ( )
```

5.3. Discussion de la transformation

En écrivant nos règles de transformation, nous étions conscients du fait qu'un composant EJB ne peut jamais devenir un composant .NET. Nous avons plutôt essayé de produire un PSM selon .NET qui permet de répondre aux besoins fonctionnels et non fonctionnels du système représenté par le PSM source selon EJB. Les divergences entre source et cible doivent être confinées aux détails dont les modèles font abstraction. Nous nous sommes fixés les deux critères suivants pour nous guider dans l'élaboration des règles de transformation.

- Le premier critère est de ne pas altérer la logique métier (*business logic)* contenue dans le PSM source. En effet il est aisé de remarquer que le métamodèle de EJB porte essentiellement sur la partie structurelle du modèle métier, laissant de côté les détails d'implémentation. Ainsi, notre transformation fournit un PSM qui conserve pour chaque composant : les *Fields*, les interfaces et les références aux autres composants ou ressources.
- Le deuxième critère consiste à s'assurer que le modèle cible représente des composants .NET qui jouissent des services de middleware comparables à ceux représentés par le modèle source de EJB.

En d'autres termes, la transformation doit préserver les spécifications fonctionnelles et non fonctionnelles; et comme ces spécifications sont réalisées avec des concepts d'implémentation dans chaque technologie, on a adopté la méthode suivante pour définir les règles de transformation et raffiner ces définitions en se conformant aux critères déjà mentionnés :

1. Faire correspondre à chaque concept d'implémentation une structure d'éléments du métamodèle EJB qu'on appelle P.
2. Au même concept, on fait correspondre un agencement d'éléments du métamodèle .NET qu'on appelle P'.
3. Ensuite on définit une règle qui fait correspondre P à P' d'une façon visant la conformité avec les deux critères ci-dessus. Raffiner cette règle par un travail itératif qui reconsidère les règles en fonction de leurs effets l'une sur l'autre et leur ordre d'exécution.

Concepts EJB		**Éléments correspondants dans le métamodèle des composants .NET**
Accès à distance	Un composant passe une référence à un appelant distant qui, à son tour, disposera d'un proxy qu'il traite comme une instance locale du composant.	MarshalByRefObject
	Un composant possède un nom JNDI	Service, Channel, Activated, WellKnown

Sécurité	Sécurité basée sur les rôles avec un checking au niveau de : l'application, le composant, les méthodes, les interfaces.	SecurityRoleAttribute associable à différents éléments.
Gestion des instances	Pooling, passivation des instances inactives, durée de vie.	ObjectPoolingAttribute, JITA, ILeaseObject, LifeTime, SingleCall, Singleton.
Gestion du comportement transactionnel	Il y a plusieurs choix déterminant la conciliation entre le contexte transactionnel du client et celui du composant	TransactionAttribute et ses attributs
Synchronization et *reentrance*	Le container garantit une mise en file d'attente (en série) des appels ciblant une instance de session. La *reentrance* pour une bean entité n'est pas recommandée mais elle est toujours permise.	SynchronizationAttribute et les options permises via son attribut synchronizationOption. La *reentrance* peut être permise en mettant la valeur de synchronizationOption à Disabled. On obtient alors une absence de synchronisation, ce qui est plus favorable que la *reentrance* à la non cohérence des données qui est contrôlable au niveau du SGBD.
Persistance	ContainerManagedEntity permet de stocker ses attributs sur un support permanent en utilisant un service offert par le container et transparent au développeur.	Le DefinedNetAttribute dont le nom est Serializable représente un attribut permettant la sérialisation des objets d'une classe au format XML. De l'autre coté, SQLServer permet des requêtes ou mises à jour en manipulant des objets sérialisés au format XML. Ainsi on peut aboutir toujours à une persistance gérée par programmation mais qui traite directement le stockage des instances de la classe sérialisable associée au ServicedComponent.

Tableau 5.2 - Principaux concepts technologiques et éléments correspondants dans .NET

Il importe de signaler que notre transformation transmet bien la propriété de composant avec état ou sans état que les EJB supportent. Ainsi un composant .NET représentant un bean Session sans état accepte le pooling alors qu'il ne l'acceptera pas si le bean était avec état.

Citons de plus qu'on a remédié au fait qu'une classe dérivée de ServicedComponent ne supporte pas de constructeur avec paramètres. Ainsi, une méthode *create* avec paramètres de l'interface home d'une Entity de EJB est transformée en une méthode d'initialisation acceptant des paramètres et appelée initializeState.

Le modèle cible comprend une seule application .NET et une seule application COM+ car elles sont suffisantes pour offrir des composants .NET qui correspondent aux composants EJB. Un développeur a toujours le choix d'inclure dans sa conception plusieurs applications .NET et plusieurs applications COM+ selon ses convenances.

D'autre part notre transformation a des limitations et repose sur certaines hypothèses qui sont :

- La transformation ne touche pas aux détails de non corespondance entre les éléments de base Class, Method et Field du métamodèle de java d'une part et ceux de .NET d'autre part. Notons que les auteurs du framework .NET avouent l'avoir créé en se basant sur Java en ce qui concerne la dimension orientée objet de ce langage.
- Le modèle cible ne conserve pas les informations de gestion de transaction de EJB si les instances de MethodTransaction d'un EntrepriseBean ne partagent pas la même valeur de transactionAttribute, c'est à dire que si les méthodes d'un même bean ont différentes gestions du comportement transactionnel. Ceci est dû au fait que la gestion des transactions selon le métamodèle de .NET est configurable au niveau d'un composant plutôt qu'au niveau de chaque méthode.
- Les méthodes d'une classe d'implémentation d'un EnterpriseBean qui ne font pas partie d'une interface home ou remote, ne sont pas prises en charge par la transformation car elles ne représentent pas une fonctionnalité offerte par un composant.
- Les éléments du métamodèle EJB qui concernent des mécanismes d'implémentation spéciaux aux EJB et dont la sémantique ne correspond pas aux sémantiques des éléments du métamodèle .NET, ont été ignorés par les règles de transformation. Notons que l'absence de ses éléments n'affecte aucune spécification fonctionnelle ou non fonctionnelle de l'application. Ces éléments ignorés sont : ceux des icones de représentation, displayName, ejbClientJar et description d'un EJBJar ou d'un EntrepriseBean. Le développeur peut toujours consulter le contenu de ces éléments tout en profitant du fait que les composants transformés gardent les noms des composants du modèle source.

On voit bien que ces limitations et hypothèses demeurent loin de nuire au caractère générique de la transformation et à la valeur des modèles qu'elle produit. En effet, un développeur conscient des limitations de la transformation automatisée, peut toujours en

profiter en ajoutant au modèle cible des annotations qui comblent les informations perdues par chacune des limitations déjà énumérées.

Notons enfin, que notre transformation a reçu un feedback très positif de la communauté MDA [2]. Des chercheurs impliqués dans ATL ont même ajouté à leur site [32] un pointeur vers la version française de notre article.

5.3.1. Perspective de validation de la transformation

Selon l'état de l'art actuel du domaine de méthodes formelles, on ne dispose pas de méthode de preuve formelle assurant que la transformation ainsi définie est exempte d'erreurs, et qu'elle ne cause pas de pertes d'information non signalées dans nos hypothèses et restrictions.

Valider la transformation revient à comparer les modèles d'entrée aux modèles de sortie; or ces deux catégories de modèles sont écrites dans deux langages différents. La comparaison des modèles implique la comparaison des systèmes représentés par ces modèles. On a alors à comparer les sémantiques des modèles. Ces sémantiques sont généralement fournies sous forme de texte informel expliquant les sémantiques des éléments des métamodèles. Ceci rend la comparaison de modèles, sur cette base, une tâche qu'on peut qualifier de compliquée et nécessitant une phase de formulation du problème selon un formalisme convenable.

Le thème de validation des transformations de modèles dans une approche MDA, comme la transformation *class to relational* et la transformation sujette de notre étude, a été évoqué [11] par des chercheurs de renommée qui ont contribué à munir MDA de certains standards comme OCL. Ils ont conclu que résoudre ce problème, nécessite une comparaison de définitions de sémantiques écrites dans différents formalismes, et qu'une telle comparaison n'a jamais été implémentée auparavant. Les auteurs ajoutent même qu'elle est pratiquement impossible [11], mais nous les laisserons seuls juges de cette opinion.

Bref, les transformations connues des modèles et la notre devront attendre la maturité de certains travaux promettant trancher la question de la validation. On s'attend à un cadre de travail approprié muni de :

- Une normalisation d'un langage d'écriture de transformation en réponse au QVT RFP qui peut être suivi par des outils permettant une certaine validation de cohérence et de conformité avec des citations formelles.
- Éventuellement une définition de la transformation inverse qui peut offrir une preuve de conservation des informations des modèles transformés, car elle permet une bijection (correspondance un à un) entre l'ensemble des modèles EJB et celui des composants .NET. Il faut alors prouver que toute composition de ces deux transformations est une identité.

- Définition de l'équivalence entre deux modèles écrits dans deux métamodèles différents.
- Écriture formelle de la sémantique des éléments des métamodèles.
- Langage approprié et outils implémentant les points précédents.

5.4. Conclusion

Le résultat de notre travail trouve sa valeur dans son originalité et son rôle comme première illustration concrète dans le domaine de transformation de modèles appartenant à deux métamodèles de cette envergure dans le cadre de MDA. La transformation est, de ce fait, comparable aux premiers engins de traduction de langues qu'on utilise sans validation, mais dont on reconnaît la production automatique d'un output qui est à compléter et à vérifier par l'utilisateur du traducteur automatique.

Ainsi notre transformation donne un moyen de réécrire n'importe quel PSM de EJB pour fournir un autre modèle en terme du métamodèle des composants .NET. Elle représente une assistance de valeur pour toute migration entre ces deux technologies surtout quand on ne dispose pas de modèle PIM de l'application considérée.

Pour illustrer ceci, on va dans le chapitre qui suit appliquer cette transformation à un modèle PSM selon EJB dans le cas d'une application e-commerce très connue dans le domaine de traitement distribué utilisant les composants. Cet exemple va nous permettre de voir comment les spécifications des composants, contenues dans le modèle source, vont être transmises au modèle cible en utilisant les mécanismes de la technologie ciblée, exprimables au niveau du métamodèle.

Chapitre 6

Étude de cas

En vue d'illustrer l'utilisation de notre transformation, nous allons considérer une application de commerce électronique similaire à l'application largement connue *Pet Store*. Notre exemple, inspiré de [8], consiste en une boutique virtuelle sur le Web permettant à un client distant d'acheter des marchandises exposées dans cette boutique.

Un utilisateur (client), authentifié par le système, peut choisir un ou plusieurs articles offerts. Il peut ensuite commander l'ensemble des éléments déjà choisis (placés dans son panier virtuel).

6.1. Le modèle selon la technologie EJB

Le modèle de notre application selon la technologie EJB est représenté dans son intégralité en annexe. Dans ce chapitre nous focalisons sur quelques aspects de l'application qui aident à mieux comprendre la transformation. La figure 6.1 montre l'instance de Session qui représente le composant principal de l'application dont le nom est EcommerceSessionEJB. Le diagramme de cette figure comprend aussi d'autres éléments qui sont étroitement liés à ce dernier composant (interfaces et EJBJar). Cette figure montre des numéros encerclés marquant des éléments du diagramme, ils indiquent les numéros des règles de transformation qui s'appliqueront à ces éléments.

Au niveau notation, on utilise les stéréotypes pour représenter les instances des différents types d'éléments des métamodèles de EJB et de .NET. Le modèle source écrit selon la technologie EJB comprend essentiellement :

- EcommerceSessionEJB qui représente le bean session avec état qui est prêt à répondre à toute application cliente.

Il authentifie la personne appelant et lui offre divers services lui permettant de choisir les éléments de son panier d'achat et de les commander. Ce bean étant de type Stateful, l'objet bean correspondant garde les informations de ses interactions avec l'utilisateur jusqu'à la fin de sa durée de vie.
Trois instances de JavaClass sont associées à EcommerceSessionEJB avec des rôles de remoteInterface, homeInterface et classe d'implémentation. Les méthodes de remoteInterface figurent pour indiquer les fonctionnalités offertes (services) par le composant.

- Le bean PersonEJB qui est un bean entité, assiste le bean session dans la tâche d'identification des clients et des informations qui leurs sont associées. Le bean PersonBean est au service de notre bean session pour gérer la persistance des informations relatives aux personnes utilisant le système. Il permet la création de nouvelles instances et des requêtes sur les instances déjà existantes.
  On trouve aussi les classes Java représentant les interfaces home et distantes de ce composant ainsi que la classe d'implémentation, avec leurs méthodes. Remarquons que la classe d'implémentation a les méthodes *accessors* get et set des fields persistants. L'interface home possède les méthodes de création d'instances ou de recherche d'instances existantes en utilisant la clé primaire.
- Quatre fields dont le bean précédent assurent la persistance.
- Une instance de EjbRef décrivant la dépendance du EcommerceSession par rapport au bean PersonEJB.
- Une instance de ResourceRef fournissant les informations de connexion à un système de gestion de base de données pour stocker et manipuler les informations concernant les commandes et les lignes de commandes.

La gestion de persistance des informations concernant les commandes, les lignes de commande et les articles, est prise en charge par le bean session interagissant avec un système de gestion de base de données. Cette tâche a pu être assurée par l'usage d'un ensemble de beans entités, mais ceci aura pour effet de gonfler notre modèle sans pour autant apporter une illustration différente de celle du bean entité Person.

6.2. Le modèle .NET produit par la transformation

L'application des règles de transformation au modèle précédent produit un modèle décrivant le même système mais écrit dans le langage défini par le métamodèle .NET. Une représentation complète de ce modèle figure en annexe. La figure 6.2 illustre la partie de ce modèle représentant essentiellement la transformation de l'instance de Session du modèle source. Cette figure montre des numéros encerclés marquant des éléments du diagramme. Ces numéros correspondent aux numéros des règles qui ont crée ou conditionné ces éléments.

Les éléments du modèle .NET sont générés par les règles de la transformation comme suit :

- **Règle 1** : Appliquée à l'instance de EJBJar du modèle de départ, cette règle crée dans le modèle cible une instance de COMplusApplication et des éléments qui lui sont associés. Cette instance de COMplusApplication représente une application COM+ permettant une configuration en commun de l'ensemble des composants desservant une même application et rendant les services COM+ possibles pour ces composants.

- **Règle 2** : Appliquée à l'instance de EJBjar dans le modèle de départ, cette règle crée dans le modèle cible une instance de NetApplication et des éléments composant une instance de ConfigurationFile. Ces instances assurent une configuration par défaut pour l'hébergement des composants .NET.

- **Règle 3** : Appliquée à l'instance de Session qu'on note SES du modèle de départ, cette règle agit dans le modèle cible comme suit :
  - Créer une instance de ServicedComponent, qu'on note SC, représentant un composant selon la technologie .NET. Associer cet élément SC à l'instance de COMplusApplication déjà créé par la règle 1. mettre l'attribut canBePooled, du ServicedComponent SC, à la valeur false car sessionType de SES est stateful.
  - Associer les instances de Method du métamodèle .NET à SC. Les instances de Method concernées sont celles transformées de leur correpondant Method possédées par les instances de remoteInterface et homeInterface associées à la Session SES dans le modèle de départ. Ces instances de Method sont créées grâce à la règle 7.
  - Associer à l'élément SC les instances de Field de .NET transformées des Field de SES.
  - Créer une instance act de Activated dans le modèle cible et associer act à l'unique instance de Service. Ceci indique que le composant .NET représenté par SC profite d'un service d'hébergement d'une application .NET.
  - Créer une instance de SynchronizationAttribute, l'associer à SC et mettre son attribut synchronizationOption à Required.
  - Créer une instance opa de ObjectPoolingAttribute qui s'associe au SC en vu de configurer ce service au cas où un service de pooling est applicable au SC.

- **Règle 4 :** cette règle s'applique suite à la détection de l'instance de ContainerManagedEntity qu'on note Ent. Elle agit dans le modèle cible d'une façon similaire à la règle 3 : elle crée une instance de ServicedComponent qu'on note SCE. Ce qui diffère de l'application de la règle précédente est la prise en charge des persistentFields de Ent. Ceci se concrétise par la création d'une instance ESerial de NetClass de .NET qui est associée à une instance d'un NetDefinedAttribute assurant son caractère serialisable. ESerial est aussi associé à une instance de Attribute où l'on stocke l'information spécifiant la clé des persistentFields de Ent.

- **Règle 5 :** cette règle est responsable de la création des instances de NetClass correspondant aux deux instances de JavaClass du modèle source et qui porteront les mêmes noms dans le modèle source et cible. Leurs noms sont InfosPerson et InfosLigneCommande visibles dans les figures B.1 et B.4 en annexe.

- **Règle 6:** cette règle est responsable de la création des instances de Field dans le modèle cible et correspondant aux instances de Field du modèle source.

- **Règle 7 :** cette règle est responsable de la création des instances de Method dans le modèle cible et qui correspondent aux instances de Method du modèle source.

- Les règles 8 et 9 ne créent rien dans le modèle cible car le modèle source ne contient pas d'élément déclencheur d'aucune de ces règles. Exemple : le modèle source ne contient pas d'instance de l'élément Session associé à une instance de MethodTransaction pour déclencher la règle 8.

- **Règle 10** : elle est responsable de la création de l'élément instance de CompRef qui est le transformé de EjbRef. Cet élément exprime une dépendance du composant transformé du bean session SES par rapport au composant transformé du bean entité appelé Person.

- **Règle 11** : Suite à la détection d'une instance de ResourceRef dans le modèle source, cette règle crée une instance de ResourceReference dans le modèle cible. Ces deux instances représentent le recours du composant à un système de gestion de base de données dans les deux modèles.

6.3. Préservation des informations

Le but de la transformation est de transmettre au modèle cible les informations contenues dans le modèle source et qui sont en relation avec les spécifications du système décrit. En effet, on peut voir comment ceci s'applique à l'exemple considéré en examinant d'abord les éléments représentant des composants dans les deux modèles source et cible.

- L'élément Session (SES) est transformé en un ServicedComponent (SC). Un ServicedComponent représente, selon la technologie .NET, un composant qui offre à son appelant, une référence sur une instance lui exposant l'interface de ce composant. Il en est de même pour SES qui est un EnterpriseBean selon la technologie EJB.
  Examinons maintenant le composant représenté par SC à la lumière des concepts qui ont guidé la définition de la transformation et qui sont énumérés dans le tableau 1 du chapitre 5 :

- **Accès à distance** : Ce qui précède traduit bien la transmission du concept d'accès à distance offert par SC et par SES.
- **Fonctionnalités** : SC expose la même interface à un client car il possède les mêmes méthodes exposées par celle de SES : isLog, controlLog, addItem, etc. Notons que SC a le même nom ECommerceSession que celui de l'interface distante du composant source.
- **État et gestion des instances** : SC contient le champ estLog qui correspond à celui de SES et qui représente l'état de l'interaction avec un client. SES étant stateful, SC a son attribut canBePooled mis à *false*. Il ne profitera donc pas du service de pooling qui ne convient pas à un composant qui garde des informations d'état de son interaction avec un client et qui n'est pas appliqué pour un bean session avec état comme SES. Par ailleurs la création de l'élément instance de Activated indique une gestion des instances selon client-Activated de .NET qui correspond bien à un bean session avec état où une instance d'un composant est associée à chaque client; le client contrôle la duré de vie de son instance.
- **Synchronisation** : Pour avoir une synchronization comparable à celle de SES (un appel à la fois), SC est marqué par un Attribute de .NET avec son attribut *synchronizationOption* mis à *Required* qui a pour effet de rendre une instance accessible par un seul thread à la fois.

- Le composant ContainerManagedEntity (Ent), est transformé en un ServicedComponent qu'on note SCE; il est associé à une instance de NetClass qu'on appelle ESerial. ESerial représente une classe sérialisable dont chaque instance renferme les informations d'une personne. SCE représente le composant qui permet les interactions avec un appelant (un autre composant) lui permettant la gestion des informations stockées des personnes, selon la structure de ESerial, au format XML.
  Il est à noter que les champs persistants de Ent (id, name, logInString et accountNumber) sont devenus les champs de ESerial, et que d'autre part les méthodes de recherche et de création des informations d'une personne, *findByPrimaryKey* et *create*, peuvent être trouvées dans SCE avec les mêmes paramètres.
  Ceci étant, examinons les concepts traduits par la transformation de l'élément Ent à la lumière du tableau 1 du chapitre 5 :

  - **Accès à distance :** assurée par l'élément SCE qui représente un ServicedComponent de la technologie .NET.
  - **Fonctionnalités :** d'après le paragraphe précédent et en examinant le modèle cible, on remarque que l'interface de home et distante du bean entité sont transmise à SCE ce qui permet la gestion des objets persistants représentés par les instances de ESerial.

- **Gestion des instances :** Comme pour SC, l'élément instance de Activated créé par la transformation de Ent, implique une gestion des instances Client-Activated de .NET qui convient à un bean entité où une instance d'un composant est associée à chaque client; le client contrôle la duré de vie de son instance. L'attribut canBePooled est mis à *true* pour permettre une réserve d'instances comparable à celle des beans entités.
- **Persistance :** assuré par l'élément ESerial et le SGBD expliqué ci-dessus.
- **Synchronnisation et réentrance :** Pour transmettre le fait que Ent ne permet pas la réentrance, on a associé à SCE un Attribute de .NET avec son attribut *synchronizationOption* mis à *Required*, ce qui a pour effet de rendre une instance accessible par un seul thread à la fois.

Ainsi, la transformation a transmis au modèle cible les interfaces exposées par les deux composants, la gestion de l'état de l'interaction avec les appelants, les champs d'état des instances des composants et les champs persistants dont on doit sauvegarder les valeurs sur un support persistant.

Explorons maintenant les éléments produits par la transformation et en relation avec l'environnement des composants; on trouve alors :

- L'élément instance de RessourceReference dans le modèle cible. Il représente une référence à un SGBD en spécifiant l'adresse du serveur et les informations sur la description de l'usage de cette ressource et sur le type de fabrique (factory) à utiliser pour s'y connecter. Cet élément fait référence à la même ressource référencée dans le modèle source. Le SGBD référencé est à utiliser pour stocker les informations des commandes et lignes de commande.
- La référence à un autre composant est transmise au modèle cible via l'élément instance de CompRef ayant l'attribut type qui fournit le nom de l'instance de ServicedComponent référencé. Cette référence reflète le fait que le composant avec état appelé par un client, utilise un autre composant (Person) pour accéder aux informations d'un client en vue de l'authentifier.
- les services de serveur d'application et du conteneur de EJB sont traduits par l'instance de ConfigurationFile et les éléments qu'elle possède ainsi que par l'instance de Comp et les éléments qui lui sont associés :
  - L'instance de Channel indique qu'une application .NET écoute les appels pour les ServicedComponents avec le protocole TCP sur le port 8010 choisi arbitrairement.
  - Chaque instance de Activated indique le nom de l'assembly, du type et de l'URL pour SC et SCE.
  - En vue d'avoir accès à des services comparables à ceux des EJB l'élément instance de Comp est associé à SC et SCE pour leur permettre d'accéder aux services COM+. Ces services sont configurables au niveau d'une application COM+ pour un groupe de composants ou au niveau de chaque composant séparément.

En résumé, on peut voir que le modèle produit par la transformation décrit la même application que celle du modèle source en conservant la structure des composants et ressources ainsi que la structure de chaque composant, formée par son interface et ses champs.

6.4. Conclusion

On remarque que le modèle produit ne peut pas être déduit par simple correspondance d'élément à élément du modèle source. D'où l'on se rend compte de l'importance de l'automatisation de la transformation surtout quand on traite un grand nombre de composants.

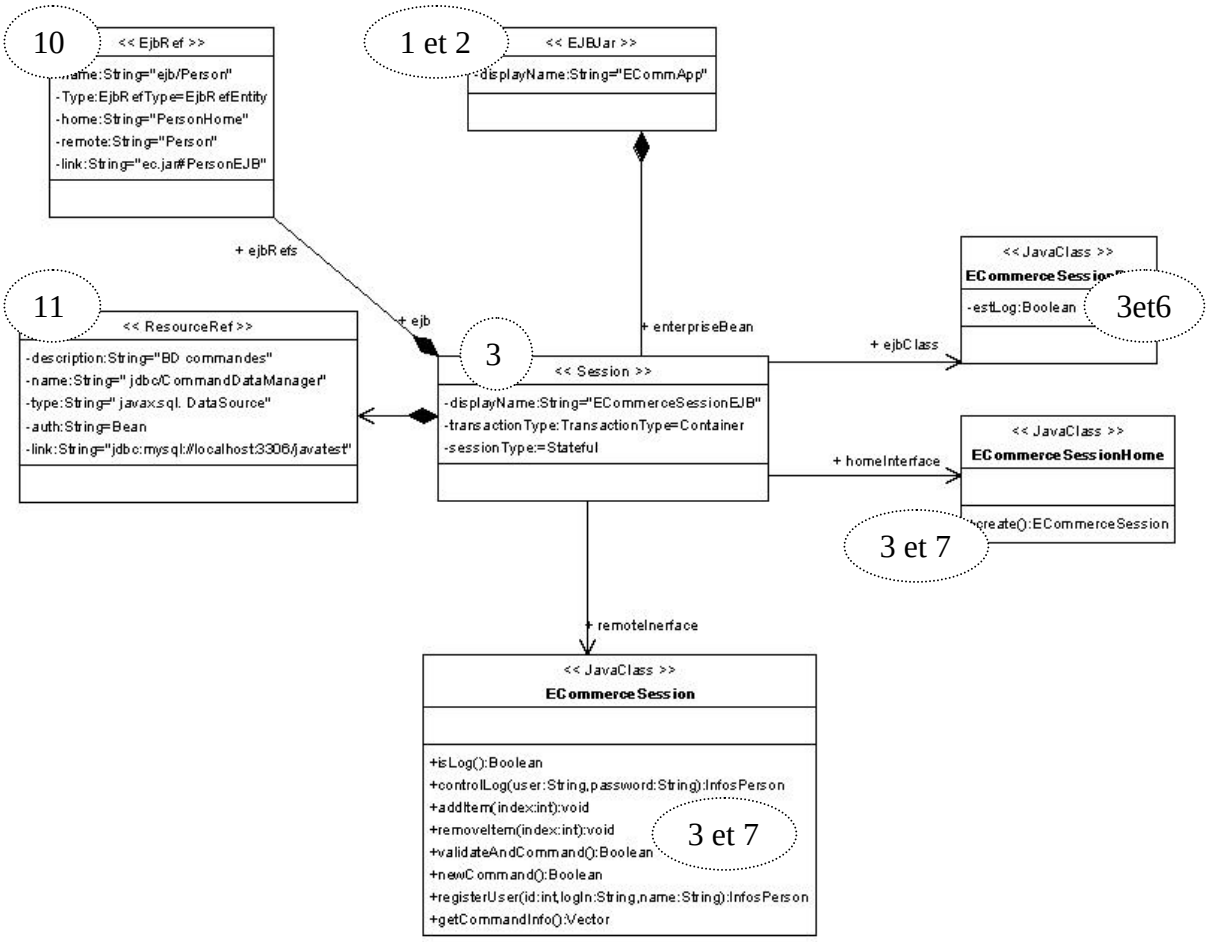


Figure 6.1 - Modèle selon EJB - Diagramme de l'instance de Session

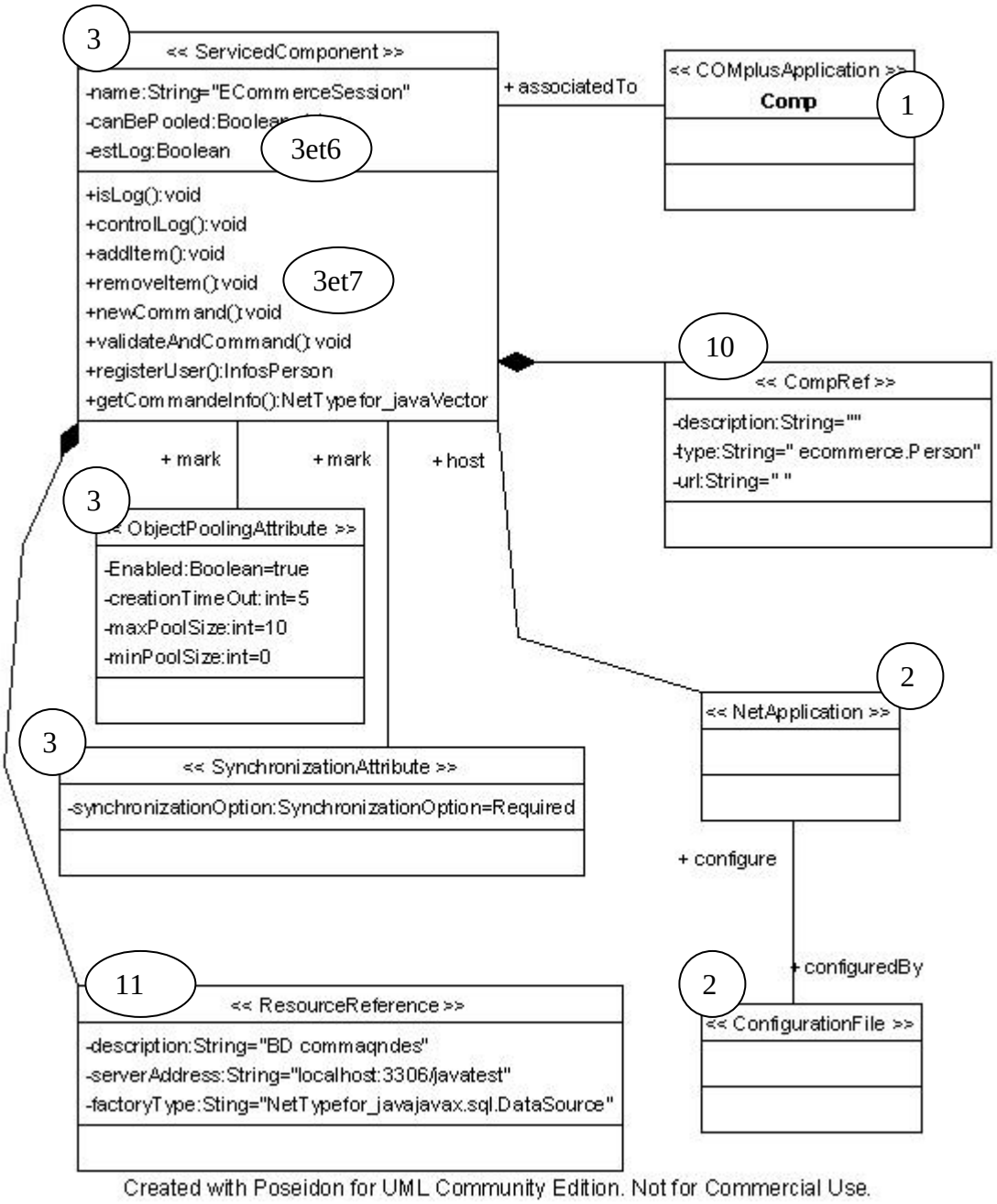


Figure 6.2 - Modèle selon .NET – Diagram de la transforamtion d'une Session

Chapitre 7

Conclusion

Dans ce qui précède, on a vu un ensemble de règles qui définissent une transformation de modèles qui permet le passage d'un modèle écrit selon EJB vers un modèle écrit selon la technologie .NET. On a vu aussi l'application de cette transformation à un exemple illustrant la productivité automatique ciblée par notre étude. Ceci est rendu possible suite à un ensemble d'accomplissements (création de métamodèles, de règles, etc.).

7.1. Travaux accomplis

Le métamodèle proposé pour la technologie des composants .NET peut être vu comme un premier pas vers un métamodèle normalisé comparable à celui des Enterprise JavaBeans. Cette normalisation de métamodèle est essentielle pour une ingénierie des logiciels dirigée par les modèles, car c'est cette normalisation qui unifie le langage d'écriture des PSMs, indispensable pour la définition de toute transformation de modèles ciblant la technologie .NET.

La transformation automatisée qu'on a définie permet de fournir, à partir de tout PSM selon EJB, un autre PSM selon la technologie .NET. Cette tâche n'est pas du tout évidente, et les règles automatisées de transformation offriront une aide de valeur à tout développeur voulant migrer de EJB vers .NET. Cette automatisation sera mieux appréciée quand le PSM selon EJB comprend un bon nombre de composants profitant de divers services, communiquant avec des ressources distribuées et interagissant entre eux pour former une application.

D'autre part, notre initiative élaborée dans le cadre de MDA et profitant de ses standards, est aussi utile dans toute autre architecture de développement dirigé par les modèles (DDM) car MDA est un cas particulier de DDM avec des restrictions en relation avec les standards et étapes de développement propres à MDA.

7.2. Travaux futurs

La définition de la transformation horizontale de PSMs entre les technologies EJB et .NET, devrait ouvrir la porte à d'autres initiatives de métamodélisation et de définition de transformations ciblant d'autres technologies. Une continuation logique de ce travail serait l'élaboration d'un outil de génération de code qui se baserait sur les métamodèles dont on dispose. À ce stade, et même en l'absence de normalisation, notre travail s'avère utile à tout développeur désirant une diversification de sa technologie ou une migration vers une autre technologie à un coût raisonnable.

Une autre extension de ce travail serait la définition des deux transformations verticales qui permettraient de convertir un PIM écrit dans le profil UML pour EDOC [23] en PSMs spécifiques à EJB et .NET, et vice versa. Nous avons aussi mené une réflexion sur la transformation inverse convertissant un PSM selon .NET vers un PSM selon EJB. Remarquons que nos règles sont loin d'être symétriques pour permettre de déduire directement leur inverse, surtout qu'elles font correspondre généralement un élément de EJB à plusieurs éléments de .NET. Ajoutons à ceci, que dans nos premières esquisses de définition de la transformation inverse, on ressent une perte d'information plus notable due essentiellement au fait que le métamodèle de EJB ne couvre pas entièrement la configuration des services offerts par le conteneur des Enterprise JavaBeans qui sont comparables aux services COM+ bien représentés dans le métamodèle .NET (ex. timeOut, maxPoolSize, etc). Ainsi, la taille maximale de réserve d'instances d'un composant peut être spécifiée explicitement dans un PSM selon .NET par la valeur de l'attribut maxPoolSize d'une instance de l'élément ObjectPoolingAttribute; d'autre part, le mécanisme de détention de réserve d'instances est supporté par le conteneur des EJB mais ses paramètres ne sont pas représentés dans le métamodèle EJB.

Notons de plus que la contribution de la modélisation dans le processus de développement de logiciel, est toujours loin de son plein potentiel et ce par manque de standards et d'outils couvrant toutes les étapes de développement dirigé par les modèles. On espère que des travaux futurs fourniront des métamodèles des technologies ainsi qu'un environnement de développement intégré standardisé et ouvert aux métamodèles, qui permet l'édition des modèles et supporte l'automatisation des transformations de modèles.

Concernant la validation de la transformation, on espère que notre travail stimulera des recherches visant l'élaboration d'un cadre de travail permettant la validation d'une transformation de modèles en tenant compte des sémantiques et caractéristiques des métamodèles en plus de la définition de la transformation. Ce sont les sémantiques représentées différemment dans deux langages (métamodèles) qui permettent une traduction (transformation) entre ces deux langages ; ces mêmes sémantiques fournissent la base de comparaison des modèles source et cible d'une transformation.

On espère que le langage normalisé QVT [20] de définition de transformation de modèles sera adopté prochainement, et que les transformations de modèles ne tarderont pas à suivre cet événement qui déclenchera ainsi une succession rapide d'autres accomplissements menant à un cadre de travail selon MDA prêt à répondre aux besoins de l'industrie relatifs au développement de logiciels.

7.3. Conclusion

Enfin, on peut constater comment notre travail permet la réduction du coût de la migration de la technologie EJB vers celle des composants .NET . Cette réduction servait de motivation principale pour notre étude, mais les résultats de cette étude l'ont dépassée pour constituer une contribution visible dans l'élaboration du cadre de travail prometteur de MDA et servir de stimulant à plusieurs autres travaux complémentaires.

Annexe A

Les outils

On peut trouver divers outils de transformation de modèles et de génération de code. Mais généralement, ces derniers implémentent partiellement MDA avec parfois des fonctionnalités sortant de son cadre. Le choix d'un outil sera plus aisé après identification des fonctionnalités offertes telles que :

- Le type de transformation offert (PIM-PSM, PSM-PSM, …) ;
- La génération de code;
- Les technologies cibles;
- La flexibilité permise à l'usager dans la définition des transformations et la construction de générateurs de code. Ex. la création de *Cartridge* dans ArcStyler, choix de valeurs de paramètres, règles de transformation ou génération de code.

Le tableau suivant présente un ensemble d'outils et identifie leurs principaux supports pour un développement selon MDA. On fait la distinction entre les outils commerciaux, ceux à code source libre (*Open Source*) et ceux qui offrent une version binaire, en les indexant par c, o et b respectivement.

Outil & développeur	Technologie	Format d'échange de modèles	Transformation de modèles	Génération de code	% code généré	Remarques Et site
AndroMDA o by http://www.mbohlen.de/ Matthias Bohlen Consulting for IT projects	EJB et autre..	XMI	S/O	UML avec quelques restrictions pour générer le code EJB	75 % de fichiers Java, pas de fichier XML	http://www.andromda.org/index.html AndroMDA assure la synchronisation entre code et modèle
GMT o Contributeurs : Softmetaware, Paris 6, INRIA/ATLAS, Bronstee	J2EE . PHP/MySQL	XMI, compatible avec **Eclipse**	la transformation est à définir par l'usager voir http://www.eclipse.org/gmt/	PIM to code Et PSM to code	complet après insertion des méthodes dans le modèle sous forme de code	comprend **UMLX** qui est un outil de transformation de modèles dans un état embryonnaire.. http://www.eclipse.org/gmt/
Xcoder o by sourceForge	Java, J2EE and C++	XMI, Plug-In Rational Rose	"MDA conforme" PIM to PSM	PSM to code	Pas complet	se caractérise par la trans. PIM-PSM http://sourceforge.net/projects/xcoder/
OpenMDX o SourceForge.NET	J2SE, J2EE, CORBA et .NET.	TOGEHER, Rational Rose, Poseidon, Magicdraw	S/O	PIM to code	Pas complet	http://sourceforge.net/projects/openmdx/
Fujaba o University of Paderborn	java	UML à partir de fujaba directement	**S/O**	PIM to code	Complet	http://wwwcs.upb.de/cs/fujaba/index.html
TOGETHER c for VS.NET/ BORLAND	VS.NET	XMI, Il s'intègre dans VS.NET	S/O	UML to code dans l'environnement VS.NET		http://info.borland.com/04/together/

Outil & développeur	Technologie	Format d'échange de modèles	Transformation de modèles	Génération de code	% code généré	Remarques Et site
ArcStyler c Integrated Object	Java et C#	XMI,eclipse ou son propre IDE	PIM to PSM	PIM to code et PSM to code. Offre comme open source les cartridges de création de générateurs de code.	on s'attend à avoir 75 %	Le style des modèles supporte bien l'aspect dynamique et l'interface graphique http://www.arcstyler.com/
OptimalJ c Compuware	J2EE	XMI	PIM to PSM	PSM to code	Complet java, JSP, XML files	Permet la validation des modèles http://www.klasse.nl/english/boeken/mdaexplained.html
ModFact o OpenSource Paris 6	MOF métamodèles	XMI	QVT Engin de transformation de modèles	S/O	S/O	TRL langage de transformation + DTD Maker http://modfact.lip6.fr/ModFactWeb/index.jsp
ADT Université de Nantes, GMT	MOFet Ecore de EMF métamodèles	XMI. Eclipse Plug-in	Engin de transformation de modèles. Langage ATL dérivé de OpenQVT	Générateur de code : à configurer utilisant EMF de Eclipse	S/O	http://www.sciences.univ-nantes.fr/lina/atl/activities/eclipse/
MIA-transformationc SodiFrance	Pour MOF métamodèles, intégrable dans VS.NET	XMI, Together,rose, Rhapsodie, Poseidon	Engin de transformation de modèles	MIA-Generation pour définir les règles de génération par l'usager	S/O	http://www.mia-software.com/
Rational Rose XDE Developer c **IBM**	J2EE, VS.NET, ADA, CORBA, C++,…	S'intègre avec VS.NET, Eclipse, WebSphere	PIM-PSM	PIM ou PSM To code Avec synchronisation.	Les méthodes sont à implémenter	Supporte plusieurs modèles. http://www-306.ibm.com/software/awdtools/developer/rosexde/features/

Tableau A.1 - Liste non exhaustive des outils implémentant MDA

Annexe B

Diagrammes complets de l'étude de cas du chapitre 6

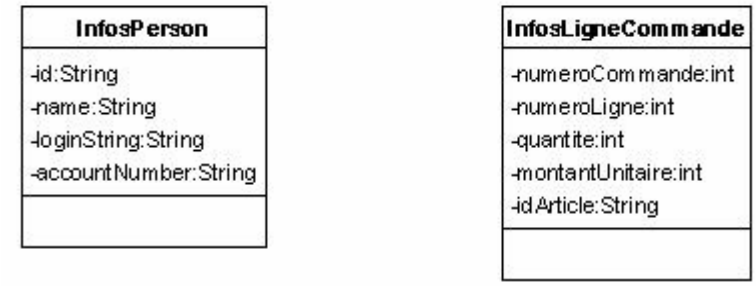
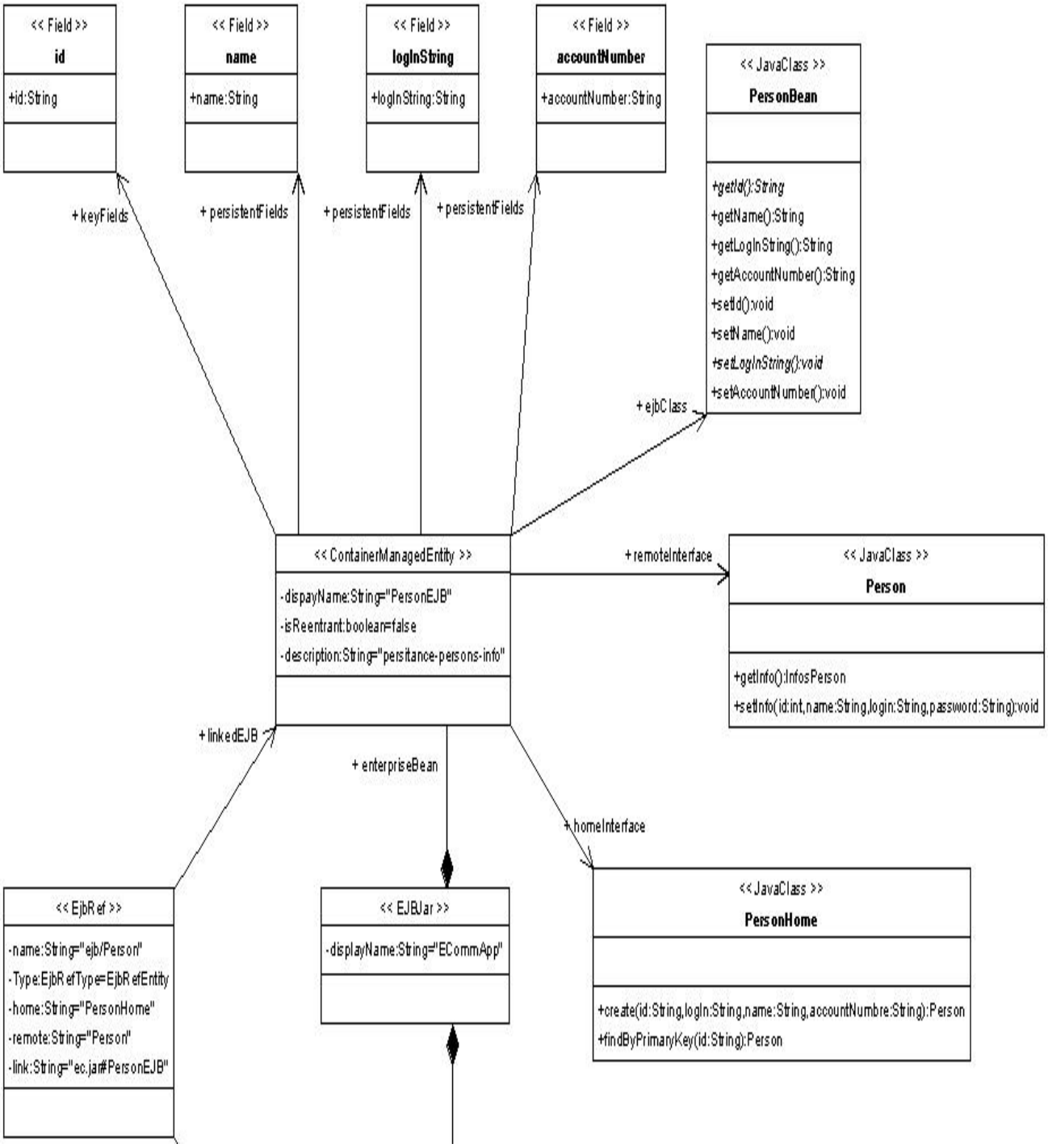
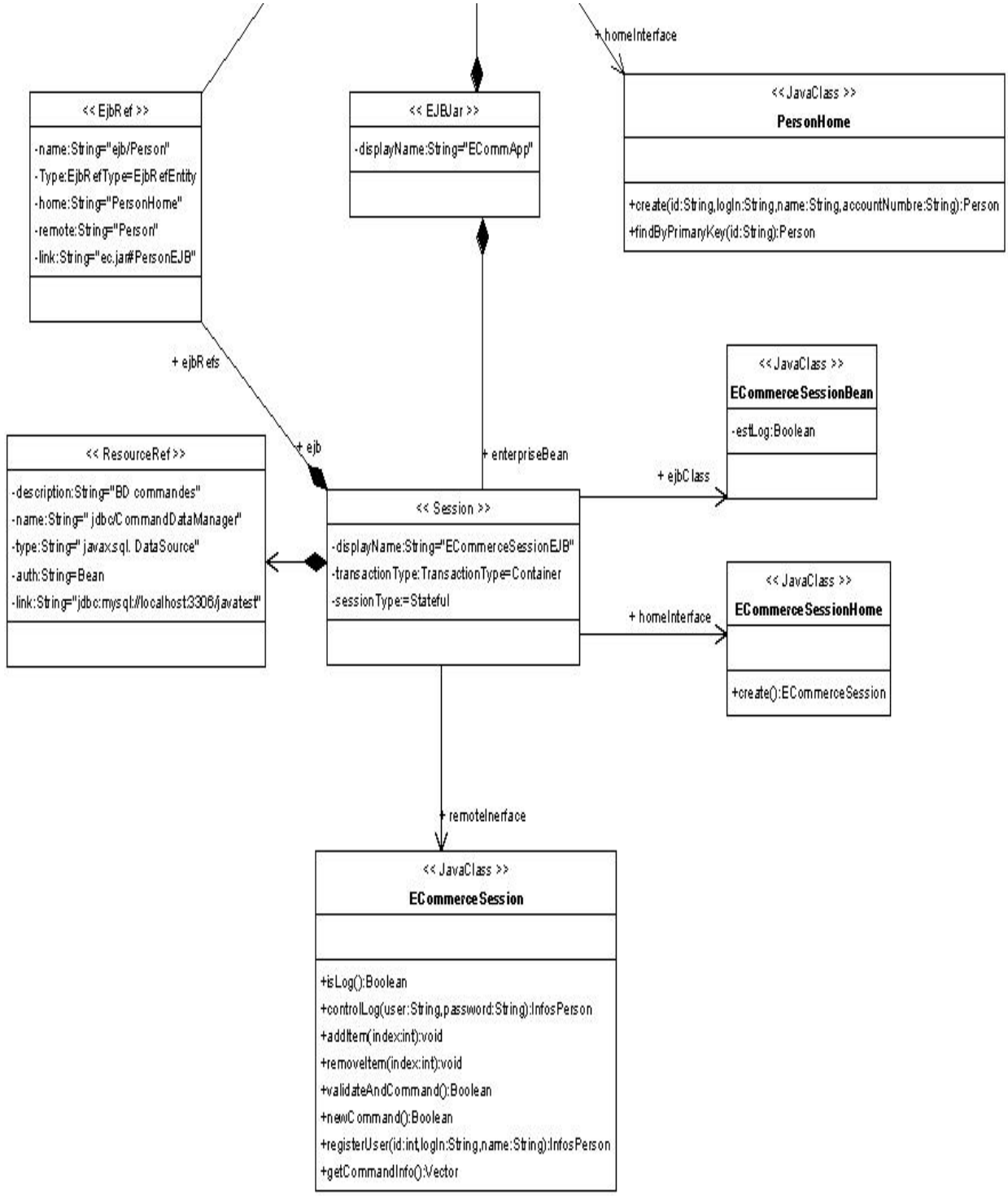


Figure B.1 - Modèle selon EJB, diagramme des Classes Java pour les informations des personnes et des lignes de commandes

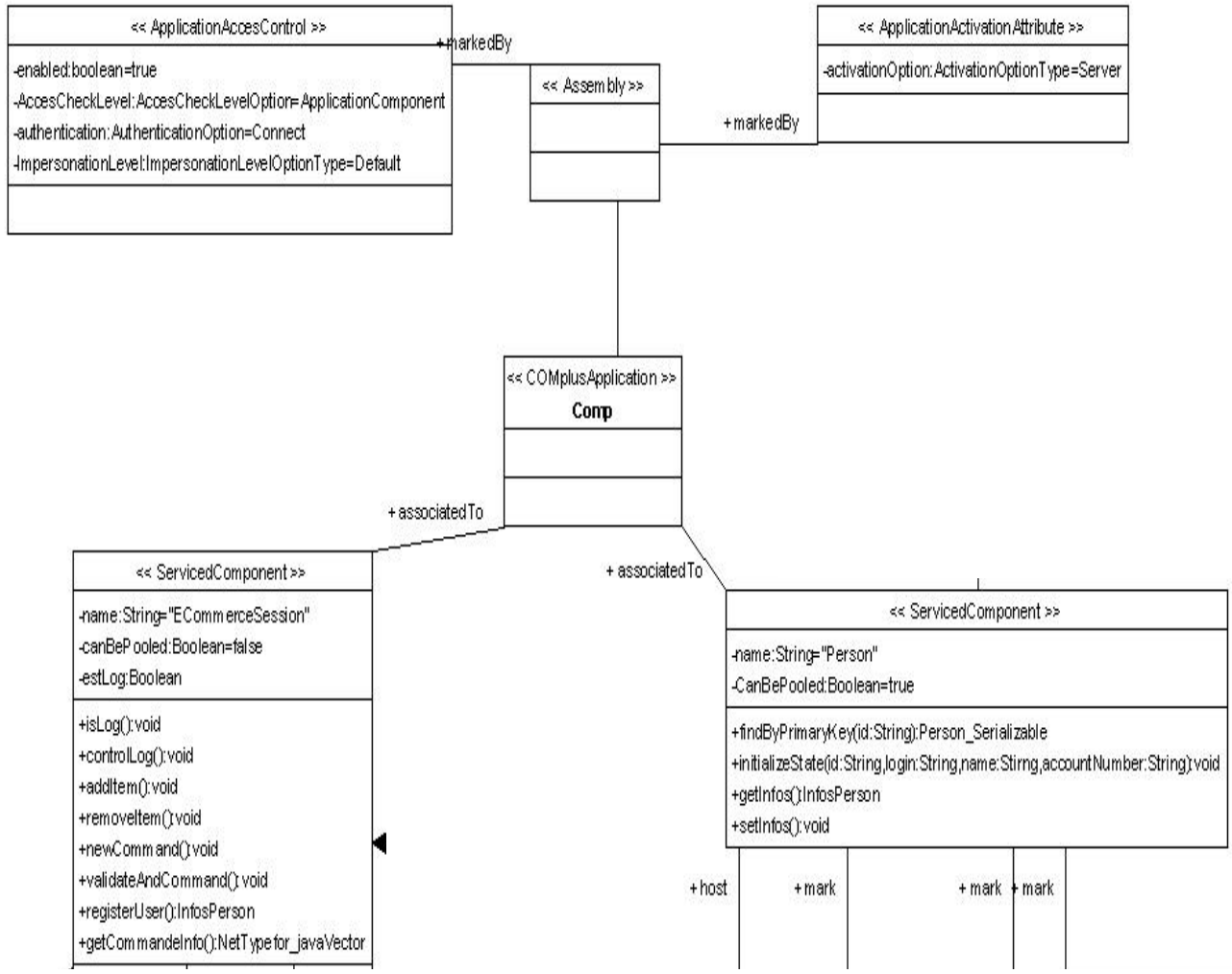


Première partie du diagramme des composants montrant les éléments instances de EJBJar et de ContainerManagedEntity et concernant principalement le composant entité.

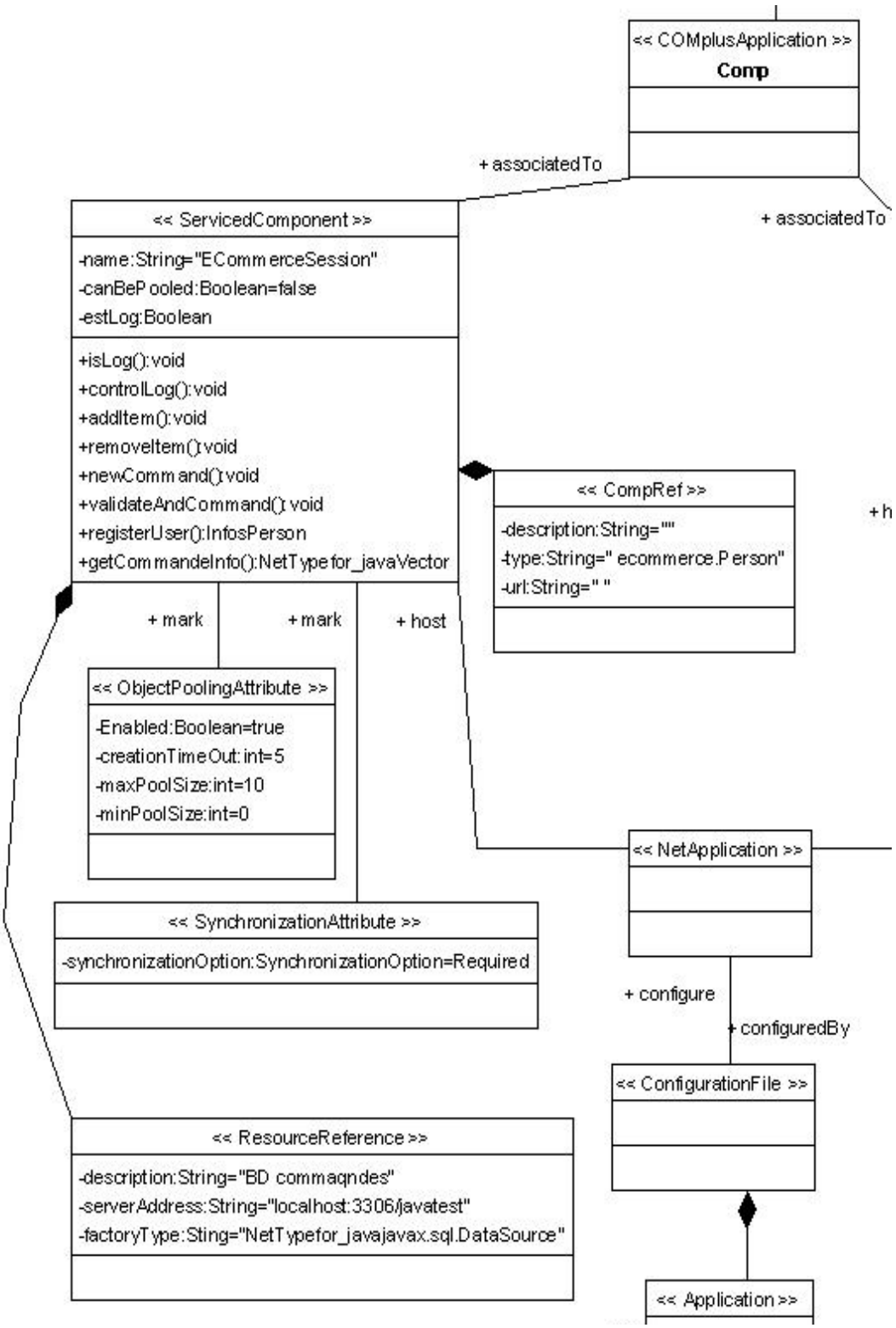


Deuxième partie du diagramme des composants montrant les éléments représentant le bean session (élément central instance de Session)

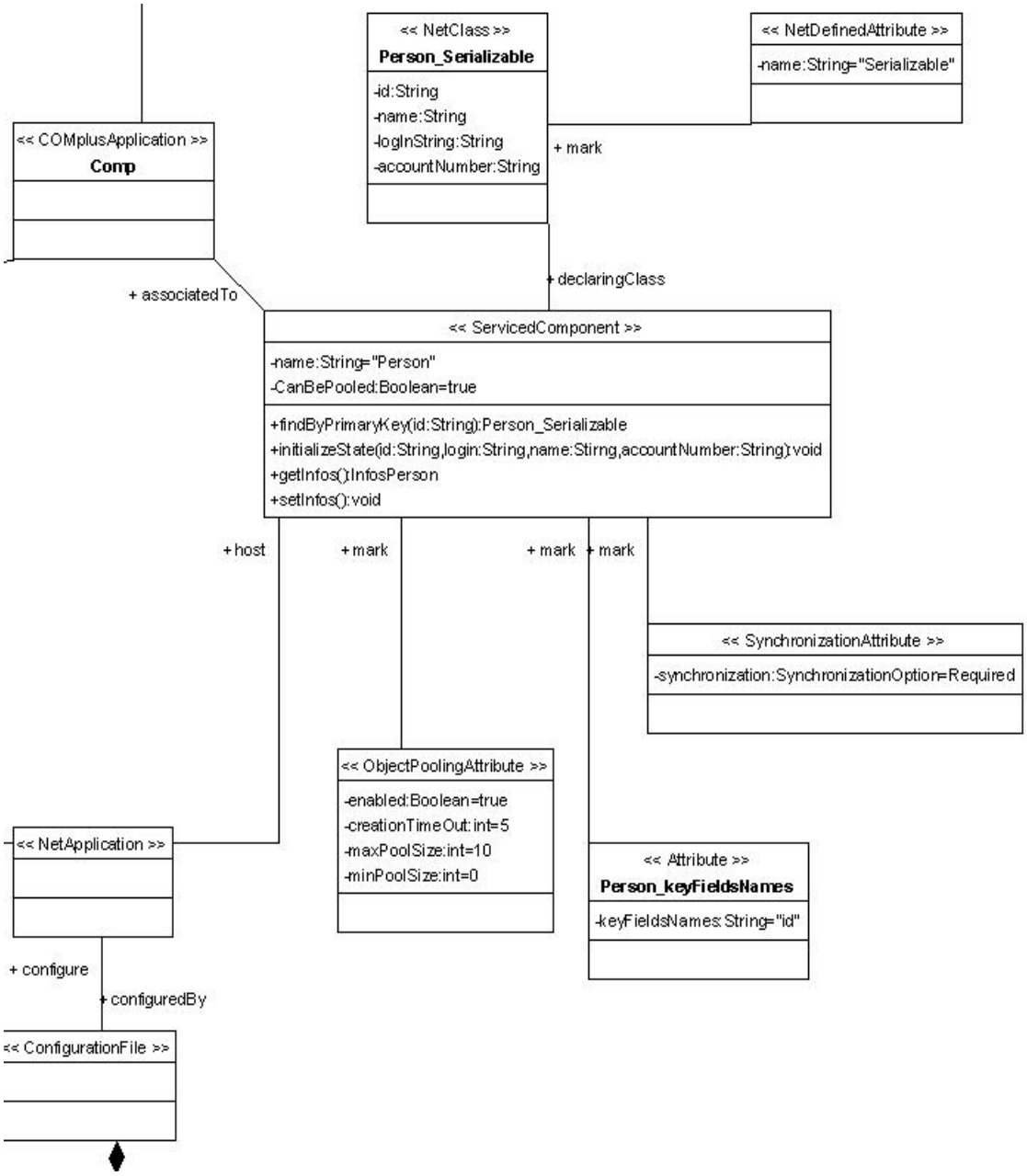
Figure B.2 - Modèle selon EJB, diagramme des composants (divisé sur deux pages)



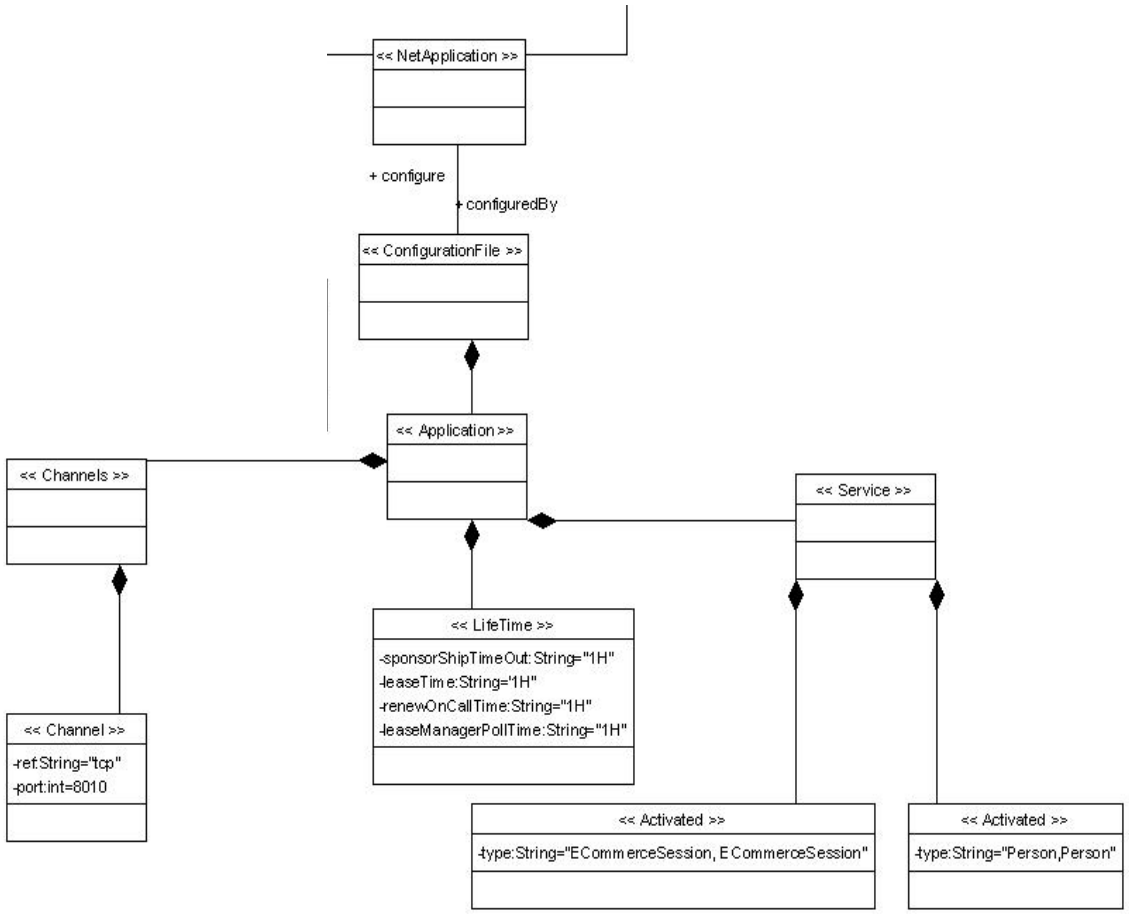
Première partie du diagramme des composants montrant les éléments représentant principalement l'application COM+ et ses associations aux deux instances de ServicedComponent représentant les composants .NET.



Deuxième partie du diagramme des composants montrant les éléments reliés étroitement à l'instance de ServicedComponent qui représente le composant .NET correspondant au bean session du modèle source.



Troisième partie du diagramme des composants montrant les éléments reliés étroitement à l'instance de ServicedComponent qui représente le composant .NET correspondant au bean entité du modèle source.



Quatrième partie du diagramme des composants montrant les éléments représentant le fichier de configuration de l'application .NET hébergeant les composant .NET.

Figure B.3 - Modèle selon .NET, diagramme des composants

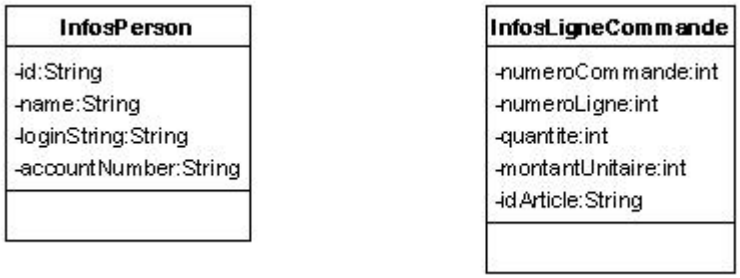


Figure B.4 - Modèle selon .NET, diagramme des classes des informations des personnes et des lignes de commandes

Bibliographie

[1] Abd-Ali Jamal and Karim El Guemhioui. (2005). "An MDA-Oriented .NET Metamodel". Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005), Enschede, The Netherlands. 1541-7719/05:142-153. Disponible à l'adresse : http://doi.ieeecomputersociety.org/10.1109/EDOC.2005.8

[2] Abd-Ali Jamal and Karim El Guemhioui. (2005). "Horizontal Transformation of PSMs**".** *European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA),* Nuremberg, Germany. LNCS 3748 : 299-315. Disponible à l'adresse : http://www.springerlink.com/(qrugxb55zi0jq0nzhxrrnz45)/app/home/contribution.asp?referrer=parent&backto=issue,22,24;searcharticlesresults,6,153

[3] Agrawal Aditya, Gabor Karsai, Feng Shi, "A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations", http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2003_A_UML_base.pdf

[4] Bézevin Jean, Erwan Breton, Grégoire Dupé, Patricx Valduriez, "The ATL Transformation-based Model Management Framework", submitted for publication.

[5] Czarnecki, K., S. Helsen. "Classification of Model Transformation Approaches". OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture.

[6] DSTC, IBM, “MOF Query/Views/Transformations, Initial Submission”, OMG Document ad/2003-02-03, http://www.dstc.edu.au/Research/Projects/Pegamento/publications/ad-03-02-03.pdf.

[7] Edward Willink, “The UMLX Language Definition”, http://www.eclipse.org/gmthome/doc/umlx/umlx.pdf.

[8] Fauré, A., N. Soukouti. *EJB 2.0 Mise en oeuvre.* Dunod. 2002.403 p.

[9] Juval Lowy. *COM and .NET Component Services*, O'Reilly 2001, 384 p. Disponible sur le site de Safari comme ressources électroniques. http://biblio.uqo.ca/ressources-electroniques/bases-donnees/safari.php

[10] Juval lowy. *Programming .NET Components*, O'Reilly 2003, Sebastopol, CA, 459 p.

[11] Kleppe Anneke, Jos Warmer. “Do MDA Transformations Preserve Meaning? An investigation into preserving semantics”. Proceedings *Metamodelling for MDA*, First International Workshop York, UK, November 2003.

[12] Kleppe Anneke, Jos Warmer et Wim Bast. 2002. *MDA Explained, The Model Driven Architecture : Practice And Promise.* Addison-Wesley, 170 p.

[13] Leblanc, Gérard. 2002. *C# et .NET*. Eyrolles. 786 p.

[14] MetaObjectFacility(MOF) Specification http://www.omg.org/docs/formal/02-04-03.pdf

[15] Model Driven Architecture: The several styles of Model Driven Architecture. Joaquin Miller. Lovelace® Computing representing X-Change Technologies http://www.omg.org/news/meetings/workshops/UML%202003%20Manual/02-2_Miller.pdf

[16] Model Transformations at the Metamodel Level Sheena R. Judson, Robert France http://www.metamodel.com/wisme-2003/19.pdf

[17] Nenad Medvidovic, David Rosenblum, David Redmiles and Jason Robbins. Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 1, January 2002, Pages 2–57.

[18] Object Management Group. MDA Guide Version 1.0.1. OMG, 2003. http://www.omg.org/docs/omg/03-06-01.pdf

[19] Object Management Group. MOF™ 2.0 Versioning and Development Lifecycle. OMG 2005 . http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF_QVT

[20] Object Management Group. Request for Proposal: MOF 2.0 Query / View /Transformations RFP. OMG 2002. http://www.omg.org/docs/ad/02-04-10.pdf.

[21] OMG. Metamodel and UML Profile for Java and EJB Specification. February 2004. Version 1.0, formal/04-02-02. An Adopted Specification of the Object Management Group, Inc.

[22] OMG / MOF 2.0, Query / Views / Transformation. ad/2002-04-10, Revised Submission, Version 1.0, 2003/08/18, OpenQVT, disponible à http://www.omg.org/docs/ad/03-08-05.pdf

[23] OMG, UML Profile for EDOC. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_for_EDOC

[24] OMG UML 2.0 Superstructure FTF Rose model containing the UML 2 metamodel
http://doc.omg.org/ptc/2004-10-05

[25] OMG XMLMetadata Interchange (XMI)Specification
http://www.omg.org/docs/formal/02-01-01.pdf

[26] Open source Integrated Development Environment. http://www.eclipse.org/

[27] Poernomo I., J. Jayaputra and H. Schmidt. (2005) “Timed Probabilistic Constraints over the Distributed Management Task Force Common Information Model”. Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005), Enschede, The Netherlands. 1541-7719/05:261-272.

[28] Response to the MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)Revised submission, version 1.0 August 18,2003, OMG Document as/2003-08-05
http://www.omg.org/docs/ad/03-08-05.pdf

[29] Rozenberg G. (ed.); "*Handbook of graph grammars and computing by graph transformation:* Volume I Foundations". World Scientific Publishing, 1997.

[30] Software Modeling and Verification Lab.; "Genermorphous home page". http://cui.unige.ch/smv/gmorph

[31] Sun Microsystems, Enterprise Java Beans. http://java.sun.com/products/ejb/

[32] The ATL Home Page http://www.sciences.univ-nantes.fr/lina/atl/references/refFrench

[33] The ATL language definition page web disponible à http://www.sciences.univ-nantes.fr/lina/ atl/atlProject/languageDefinition/

[34] The Microsoft Developer Network (MSDN) available at http://msdn.microsoft.com/library/default.asp

[35] QVT Partners, "Initial submission for MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2003-03-27, http://www.qvtp.org/downloads/1.0/qvtpartners1.0.pdf.

[36] QVT Partners. QVT: The high level scope, QVT-Partners, 2003. http://qvtp.org/downloads/qvtscope.pdf.

[37] UML Profile for Enterprise Application Integration (EAI) http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_for_EAI

[38] Unified Modeling Language: OCL. Version 2.0. Disponible @ http://www.omg.org/docs/ptc/03-08-08.pdf

[39] XMOF, Queries, Views and Transformations on Models using MOF, OCL and Patterns. Proposal to document: MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10. http://www.omg.org/docs/ad/03-08-07.pdf