architecte logiciel

Xavier Blanc

MDA en action

Ingénierie logicielle guidée par les modèles

Sur le CD-Rom offert avec ce livre
IBM Rational Software Modeler*
Objecteering/MDA Modeler*
Spécifications de l'OMG
* En version d'évaluation

EYROLLES

MDA en action

Ingénierie logicielle guidée par les modèles

CHEZ LE MÊME ÉDITEUR

Dans la collection Architecte logiciel

F. VALLÉE. – **UML pour les décideurs**.
N°11621, 2005, 300 pages.

P. ROQUES, F. VALLÉE. – **UML 2 en action**. *De l'analyse des besoins à la conception J2EE*.
N°11462, 3e édition, 2004, 380 pages + poster.

J.-L. BÉNARD, L. BOSSAVIT , R. MÉDINA , D. WILLIAMS. – **Gestion de projet Extreme Programming**.
N°11561, 2002, 300 pages.

Modélisation et méthodes logicielles

P. ROQUES. – **UML 2 par la pratique, 3e édition**. *Cours et exercices*.
N°11480, 2004, 340 pages.

P.-A. MULLER, N. GAERTNER. – **Modélisation objet avec UML**.
N°11397, 2e édition 2000, 520 pages (réédition au format semi-poche).

P. ROQUES. – **UML : modéliser un site e-commerce**.
N°11070, 2002, 168 pages.

A. COCKBURN. – **Rédiger des cas d'utilisation efficaces**.
N°9288, 2001, 320 pages.

I. JACOBSON, G. BOOCH, J.RUMBAUGH. – **Le Processus unifié de développement logiciel**.
N°9142, 2000, 487 pages.

R. PAWLAK, J.-P. RETAILLÉ, L. SEINTURIER. – **Programmation orientée aspect pour Java/J2EE**.
N°11408, 2004, 460 pages.

B. MEYER. – **Conception et programmation orientées objet**.
N°9111, 2000, 1223 pages.

Développement Java/J2EE, .NET et PHP

K. DJAFAAR. – **Eclipse et JBoss**. *Développement d'applications J2EE professionnelles, de la conception au déploiement*
N°11406, 2005, 656 pages + CD-Rom.

J. MOLIÈRE. – **Cahier du programmeur J2EE.** *Conception et déploiement J2EE 1.4*.
N°11574, 2005, 234 pages.

R. FLEURY. – **Cahier du programmeur Java/XML**.
M*éthodes et frameworks : Ant, Junit, Eclipse, Struts-Stxx, Cocoon, Axis, Xerces, Xalan, JDom, XIndice…*
N°11316, 2004, 228 pages.

J. WEAVER, K. MUKHAR, J. CRUME – **J2EE 1.4**.
N°11484, 2004, 666 pages.

J. GOODWILL. – **Jakarta Struts**.
N°11231, 2003, 354 pages.

L. MAESANO, C. BERNARD, X. LEGALLES. – **Services Web en J2EE et .Net**
N°11067, 2003, 1088 pages.

D. LANTIM. – **.NET par Dick Lantim.**
G11200, 2003, 564 pages.

E. DASPET, C. PIERRE DE GEYER. – **PHP 5 avancé**.
N°11323, 2004, 784 pages.

S. MARIEL. **– Cahier du programmeur PHP 5**. *PHP objet et XML.*
N°11234, 2004, 288 pages.

Xavier Blanc

MDA en action

Ingénierie logicielle guidée par les modèles

Avec la contribution de Olivier Salvatori

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

À ma femme Virginie et à mon fils Arthur

Remerciements

Je tiens à remercier ici tous ceux qui m'ont aidé directement et indirectement à rédiger cet ouvrage :

- Marie-Pierre Gervais, pour son soutien, sa vision académique qu'elle sait partager, ses précieux conseils de fond et de forme et toutes les heures qu'elle a bien voulu passer à me relire, sans lesquels cet ouvrage n'aurait sans doute jamais vu le jour.
- Philippe Desfray, pour son regard profondément novateur sur MDA qu'il a bien voulu partager avec moi, sa disponibilité, sa confiance et les précieux conseils qu'il a su me prodiguer.
- Les membres français de l'OMG, pour m'avoir intégré à leurs meetings et pour les discussions fructueuses que nous avons eues.
- Les membres du groupe Ingénierie des modèles de l'Observatoire français des techniques avancées (OFTA) pour les travaux de synthèse que nous avons menés.
- Les membres du groupe Meta, pour toutes les réunions passionnantes que nous avons eues depuis plus de quatre ans et qui donnent toujours lieu à des débats plus qu'enrichissants.
- Serge Bonnaud, pour son accueil et pour les efforts qu'il a déployés pour que ce projet voie le jour.
- Jean-François Perrot et Gilles Blain, qui m'ont initié à l'ingénierie logicielle guidée par les modèles, pour leur ouverture d'esprit, la démarche scientifique qu'ils ont su m'apporter et l'expérience qu'ils m'ont chaleureusement transmise.
- Salim Bouzitouna, Prawee Sriplakich, Reda Bendraou, Grégory Jevardin De Fombelle, Samir Ammour et Maher Lamari, pour leur curiosité et leurs idées neuves.
- Régis Blanc et Frédéric Vigouroux, pour leurs précieux conseils en PHP.
- Les membres du thème SRC, pour le soutien chaleureux qu'ils m'ont accordé et leurs éclairages scientifiques sur des domaines tels que les techniques formelles.

Préface

Dans notre quête de la puissance d'expression des langages informatiques et du gain de productivité des développements, chaque décennie apporte son lot de techniques nouvelles, qui viennent se compléter les unes les autres. Ainsi avons-nous vu apparaître les langages symboliques, la programmation fonctionnelle, la programmation structurée, les SGBD, les machines abstraites, les L4G, la programmation par objets, les design patterns, les infrastructures de type CORBA, J2EE ou .Net et la programmation par composants.

Les modèles ont suivi une évolution parallèle aux langages et ont été étendus, étoffés et complétés pour offrir une capacité de représentation des problèmes et solutions au niveau le plus abstrait. UML et sa dernière version UML2.0 offrent ainsi une panoplie très complète de techniques de modélisation, permettant de représenter tout type de système et d'application logicielle.

L'apport de MDA est de permettre d'exploiter ces modèles pour simuler, estimer, comprendre, communiquer et produire. Le présent ouvrage présentera de nombreux exemples illustrant l'aspect production, en montrant comment un modèle peut être exploité pour produire différents codes. Le code ne devient dès lors qu'une conséquence, une dérivation mécanique du modèle. MDA contient en germe une progression considérable de la productivité et de la puissance d'expression des développements logiciels.

Initiée en 1999 par l'OMG, l'approche MDA se répand dans le monde du développement logiciel et constitue une évolution importante des pratiques de développement d'applications. Il manquait à ce jour un livre de référence, permettant au plus grand nombre de comprendre et mettre en œuvre les technologies MDA, pour que MDA ne soit plus considéré comme réservé à un cercle de spécialistes.

Xavier Blanc a depuis toujours travaillé dans le domaine de MDA. Nous avons travaillé ensemble dans des projets de recherche liés à MDA, dans des actions de standardisation à l'OMG et actuellement sur le plus gros projet de recherche logiciel européen, appelé *Modelware,* aussi dédié aux extensions des technologies MDA à mettre en œuvre.

Combinant une recherche permanente sur MDA à une expérience professionnelle industrielle et à un poste d'enseignement à l'Université Paris-VI, Xavier Blanc a pu transmettre dans ce livre une partie de son savoir, appuyé par son expérience pratique et une précieuse approche pédagogique.

MDA en action est à ce jour le livre que je recommande à tous ceux qui veulent connaître ce que sont l'approche et les technologies MDA et qui veulent voir des cas pratiques de mise en œuvre.

L'approche MDA sera-t-elle une évolution majeure de notre décennie informatique ? Je vous en laisse juge. On ne saurait trouver meilleure source pour instruire ce passionnant dossier que le présent ouvrage.

Philippe DESFRAY, directeur R&D de Softeam

Table des matières

PARTIE I

PARTIE II

PARTIE IV

PARTIE V

Avant-propos

Les systèmes informatiques n'ont jamais été autant au centre de la stratégie des entreprises qu'aujourd'hui. Les fonctionnalités qu'ils offrent, leur facilité d'utilisation, leur fiabilité, leur performance et leur robustesse sont les qualités reines qui permettent aux entreprises d'être compétitives. L'ingénierie du logiciel fournit sans cesse de nouvelles technologies facilitant la mise en œuvre de ces systèmes tout en accroissant leurs qualités. La dernière en date de ces technologies est certainement l'approche par composant, même si nous voyons déjà pointer l'approche par service.

Le revers de la médaille de ces technologies, aussi appelées plates-formes d'exécution, est qu'elles complexifient énormément les systèmes informatiques. Cela met les entreprises dans une situation inconfortable, car elles hésitent entre adopter une nouvelle plate-forme et subir le coût de la migration ou ne pas l'adopter et prendre le risque de voir des concurrents devenir plus compétitifs grâce au choix inverse.

Cet inconvénient majeur est un frein à l'évolution. Sa source réside dans la complexité des plates-formes d'exécution, pourtant conçues pour faciliter le développement et la maintenance des systèmes. Pour y faire face, il était nécessaire de définir une approche permettant de dompter la complexité. Cette approche se devait d'être flexible et générique afin de pouvoir s'adapter à tout type de plate-forme.

L'ingénierie logicielle guidée par les modèles, ou MDA (Model Driven Architecture), correspond à la définition d'une telle approche. MDA applique la séparation des préoccupations entre la logique métier des systèmes informatiques et les plates-formes utilisées et se fonde sur l'utilisation massive des modèles.

MDA consiste à élaborer les modèles de la logique métier des systèmes de façon indépendante des plates-formes d'exécution puis à transformer ces modèles automatiquement vers des modèles dépendant des plates-formes. La complexité des plates-formes n'apparaît plus dans les modèles de la logique métier mais se retrouve dans la transformation.

Les avantages de MDA sont donc la pérennisation de la logique métier de l'entreprise grâce à l'élaboration de modèles, la productivité de cette logique métier grâce à l'automatisation des transformations de modèles et la prise en compte des plates-formes d'exécution grâce à l'intégration de celles-ci dans les transformations de modèles.

Objectifs de l'ouvrage

Cet ouvrage décrit tous les standards, techniques et frameworks qui composent MDA et présente son architecture socle, qui définit les concepts de modèle, métamodèle et méta-métamodèle.

Après une présentation détaillée des standards phares de MDA, tels que UML ou OCL, nous insistons sur les aspects pratiques de sa mise en œuvre. Nous expliquons en particulier le fonctionnement de JMI et EMF, qui permettent de programmer en Java des traitements sur les modèles, tels que les transformations. Nous présentons ensuite Rational Software Modeler et MDA Modeler, deux outils du marché qui supportent MDA. Pour finir, nous montrons comment MDA prend en compte les plates-formes d'exécution. Nous détaillons en particulier la prise en compte des plates-formes J2EE/EJB et PHP.

Cet ouvrage s'efforce d'offrir une vue complète de MDA et l'illustre par une étude de cas. Cette dernière consiste à développer une application de commerce électronique selon les principes MDA. Cette étude de cas vise à mettre en lumière les avantages de MDA, à commencer par la gestion de la complexité des plates-formes.

Cet ouvrage aura atteint son but s'il aura permis de comprendre la philosophie de MDA, ses concepts fondateurs et son état d'implémentation actuel.

Organisation de l'ouvrage

- Le chapitre 1 est un parcours macroscopique de l'ensemble des standards, techniques et frameworks de MDA. Nous exposons la philosophie de l'approche et expliquons l'intérêt des modèles en précisant leur rôle. Nous définissons en outre les qualités intrinsèques des modèles que sont la pérennité, la productivité et la prise en compte des plates-formes. Ces trois qualités serviront de grille de lecture pour la suite de l'ouvrage.

La première partie est consacrée à la pérennité des savoir-faire offerte par MDA :

- Le chapitre 2 introduit le standard MOF, qui permet d'élaborer des métamodèles. Nous détaillons ce qu'est un métamodèle et expliquons comment les concevoir. Ce chapitre nous permettra ensuite de présenter les différents métamodèles qui constituent MDA.
- Le chapitre 3 est dédié au standard UML, qui est le langage préconisé pour élaborer les PIM. Nous nous attardons sur une partie du métamodèle UML afin de bien faire saisir la place que prend UML dans MDA. Nous introduisons également la notion de profil UML.
- Le chapitre 4 présente les standards AS (Action Semantics) et OCL (Object Constraint Language), qui permettent de spécifier le corps des méthodes UML. Ces standards sont indépendants des langages de programmation. Nous donnons une partie de leur métamodèle et expliquons leur lien avec UML dans MDA.
- Le chapitre 5 est consacré au standard XMI (XML Data Interchange), qui permet de représenter les modèles sous forme de documents XML. Ce standard est d'une

importance capitale pour la pérennité des modèles MDA car c'est grâce à lui que les modèles peuvent être concrètement échangés.

La partie II se penche sur les gains de productivité apportés par les modèles MDA :

- Le chapitre 6 présente les concepts de manipulation de modèles, avec notamment le standard JMI (Java Metadata Interface). Nous introduisons à cette occasion EMF, qui est la plate-forme de métamodélisation d'Eclipse *(http://www.eclipse.org/emf).*
- Le chapitre 7 détaille les concepts à l'œuvre dans la transformation des modèles. Nous présentons les différentes façons d'élaborer des transformations de modèles avec les approches par programmation, par template et par modélisation (MOF2.0 QVT).
- Le chapitre 8 examine les techniques MDA disponibles dans les outils du marché. Nous nous attardons en particulier sur les outils Rational Software Modeler et Softeam MDA Modeler.

La partie III est consacrée à la prise en compte des plates-formes par MDA :

- Le chapitre 9 introduit les concepts de plates-formes tels que définis dans MDA et présente les différentes façons de prendre en compte les plates-formes d'exécution.
- Le chapitre 10 est consacré à la plate-forme J2EE/EJB. Nous expliquons sa prise en compte dans MDA grâce à la définition d'un profil UML. Cette prise en compte sera utilisée au chapitre 12 qui détaille l'étude de cas.
- Le chapitre 11 traite de la plate-forme PHP. Nous expliquons sa prise en compte dans MDA grâce à la définition d'un métamodèle spécifique de cette plate-forme. Cette prise en compte sera elle aussi utilisée au chapitre 12.

La partie IV illustre la mise en œuvre des principes de MDA dans une étude de cas :

- Le chapitre 12 entre dans le détail de l'utilisation de l'approche MDA pour la réalisation de l'étude de cas PetStore. Nous expliquons chacune des étapes de construction de l'application PetStore à la lumière des principes MDA. La fin du chapitre synthétise les principaux apports de l'approche MDA.

La partie V est constituée d'une annexe détaillant le contenu du CD offert avec l'ouvrage.

À qui s'adresse l'ouvrage

Cet ouvrage est destiné à toute personne désirant connaître les avantages et les limites de MDA.

Le développeur trouvera les réponses aux questions qu'il se pose sur l'intérêt des langages de modélisation et sur leurs capacités notamment en terme de production (génération de code).

Le concepteur de modèles, tels que les modèles UML, pourra élargir sa vision quant aux avantages offerts par les modèles. Il pourra notamment découvrir les concepts de

métamodèle, de transformation de modèle et de modèle de plate-forme et ainsi mieux comprendre la place qu'occupe UML dans MDA.

Le décideur et l'architecte pourront comprendre tous les enjeux de MDA. Ils seront à même, grâce à la présentation des outils de support de MDA et à l'étude de cas, de savoir quand et comment s'approprier cette approche.

1

L'architecture MDA

Ce chapitre présente de façon globale l'architecture MDA (Model Driven Architecture) telle que définie par l'OMG (Object Management Group). La suite de l'ouvrage revient en détail sur tous les points introduits ici.

Chacun s'accorde à penser que l'ingénierie logicielle guidée par les modèles est l'avenir des applications. De Bill Gates — *« Modéliser est le futur, et je pense que les sociétés qui travaillent dans ce domaine ont raison »* — à Richard Soley, le directeur de l'OMG, en passant bien entendu par les pionniers du monde UML, comme Graddy Booch, Jim Rumbaugh ou Ivar Jacobson et bien d'autres, tous affirment qu'il est temps d'élaborer les applications à partir de modèles et, surtout, de faire en sorte que ces modèles soient au centre du cycle de vie de ces applications, autrement dit qu'ils soient productifs.

Les modèles

Les modèles offrent de nombreux avantages. Ceux qui pratiquent UML ou tout autre langage de modélisation les connaissent bien. L'avantage le plus important qu'ils procurent est de spécifier *différents niveaux d'abstraction*, facilitant la gestion de la complexité inhérente aux applications.

Les modèles très abstraits sont utilisés pour présenter l'architecture générale d'une application ou sa place dans une organisation, tandis que les modèles très concrets permettent de spécifier précisément des protocoles de communication réseau ou des algorithmes de synchronisation. Même si les modèles se situent à des niveaux d'abstraction différents, il est possible d'exprimer des *relations de raffinement* entre eux. Véritables liens de traçabilité, ces relations sont garantes de la cohérence d'un ensemble de modèles représentant une même application.

La diversité des possibilités de modélisation ainsi que la possibilité d'exprimer des liens de traçabilité sont des atouts décisifs pour gérer la complexité.

Un autre avantage incontestable des modèles est qu'ils peuvent être présentés sous format graphique, facilitant d'autant la communication entre les acteurs des projets informatiques. Les modèles graphiques parmi les plus utilisés sont les modèles relationnels, qui permettent de spécifier la structure des bases de données. La représentation graphique de ces modèles offre un gain significatif de productivité.

Les mauvaises langues prétendent que modéliser est la meilleure façon de perdre du temps puisque, *in fine*, il faut de toute façon écrire du code. De même, au fameux dicton énonçant qu'un bon schéma vaut mieux qu'un long discours, on entend parfois répliquer qu'à un schéma peuvent correspondre plus de mille discours, selon la façon dont on l'interprète.

Ces critiques visent juste en cas d'absence de maîtrise des bonnes pratiques de modélisation, c'est-à-dire de l'ingénierie des modèles. C'est pourquoi il est essentiel d'acquérir de bonnes pratiques de modélisation afin de déterminer comment, quand, quoi et pourquoi modéliser et d'exploiter pleinement les avantages des modèles.

L'OMG a défini MDA (Model Driven Architecture) en 2000 dans cet objectif. L'approche MDA préconise l'utilisation massive des modèles et offre de premières réponses aux comment, quand, quoi et pourquoi modéliser. Sans prétendre être une Bible de la modélisation, répertoriant toutes les bonnes pratiques, elle vise à mettre en valeur les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plates-formes d'exécution. MDA inclut la définition de plusieurs standards, notamment UML, MOF et XMI.

L'OMG

L'OMG (Object Management Group) est un consortium à but non lucratif d'industriels et de chercheurs, dont l'objectif est d'établir des standards permettant de résoudre les problèmes d'interopérabilité des systèmes d'information *(http://www.omg.org)*.

Le principe clé de MDA consiste en l'utilisation de modèles aux différentes phases du cycle de développement d'une application. Plus précisément, MDA préconise l'élaboration de modèles d'exigences (CIM), d'analyse et de conception (PIM) et de code (PSM).

L'objectif majeur de MDA est l'élaboration de modèles pérennes, indépendants des détails techniques des plates-formes d'exécution (J2EE, .Net, PHP ou autres), afin de permettre la génération automatique de la totalité du code des applications et d'obtenir un gain significatif de productivité.

D'autres modèles, comme les modèles de supervision, de vérification ou d'organisation d'entreprise, ne sont pas encore intégrés dans l'approche MDA mais le seront rapidement sans difficulté, compte tenu de son ouverture.

Le modèle d'exigences CIM (Computation Independent Model)

La première chose à faire lors de la construction d'une nouvelle application est bien entendu de spécifier les exigences du client. Bien que très en amont, cette étape doit fortement bénéficier des modèles.

L'objectif est de créer un modèle d'exigences de la future application. Un tel modèle doit représenter l'application dans son environnement afin de définir quels sont les services offerts par l'application et quelles sont les autres entités avec lesquelles elle interagit.

La création d'un modèle d'exigences est d'une importance capitale. Cela permet d'exprimer clairement les liens de traçabilité avec les modèles qui seront construits dans les autres phases du cycle de développement de l'application, comme les modèles d'analyse et de conception. Un lien durable est ainsi créé avec les besoins du client de l'application.

Les modèles d'exigences peuvent même être considérés comme des éléments contractuels, destinés à servir de référence lorsqu'on voudra s'assurer qu'une application est conforme aux demandes du client.

Il est important de noter qu'un modèle d'exigences ne contient pas d'information sur la réalisation de l'application ni sur les traitements. C'est pourquoi, dans le vocabulaire MDA, les modèles d'exigences sont appelés des CIM (Computation Independent Model), littéralement « modèle indépendant de la programmation ».

Avec UML, un modèle d'exigences peut se résumer à un diagramme de cas d'utilisation. Ces derniers contiennent en effet les fonctionnalités fournies par l'application (cas d'utilisation) ainsi que les différentes entités qui interagissent avec elle (acteurs) sans apporter d'information sur le fonctionnement de l'application.

Dans une optique plus large, un modèle d'exigences est considéré comme une entité complexe, constituée entre autres d'un glossaire, de définitions des processus métier, des exigences et des cas d'utilisation ainsi que d'une vue systémique de l'application.

Si MDA n'émet aucune préconisation quant à l'élaboration des modèles d'exigences, des travaux sont en cours pour ajouter à UML les concepts nécessaires pour couvrir cette phase amont.

Le rôle des modèles d'exigences dans une approche MDA est d'être les premiers modèles pérennes. Une fois les exigences modélisées, elles sont censées fournir une base contractuelle variant peu, car validée par le client de l'application. Grâce aux liens de traçabilité avec les autres modèles, un lien peut être créé depuis les exigences vers le code de l'application.

Le modèle d'analyse et de conception abstraite PIM (Platform Independent Model)

Une fois le modèle d'exigences réalisé, le travail d'analyse et de conception peut commencer. Dans l'approche MDA, cette phase utilise elle aussi un modèle.

L'analyse et la conception sont depuis plus de trente ans les étapes où la modélisation est la plus présente, d'abord avec les méthodes Merise et Coad/Yourdon puis avec les méthodes objet Schlear et Mellor, OMT, OOSE et Booch. Ces méthodes proposent toutes leurs propres modèles. Aujourd'hui, le langage UML s'est imposé comme la référence pour réaliser tous les modèles d'analyse et de conception.

Par conception, il convient d'entendre l'étape qui consiste à structurer l'application en modules et sous-modules. L'application des patrons de conception, ou Design Patterns, du GoF (Gang of Four) fait partie de cette étape de conception. Par contre, l'application de patrons techniques, propres à certaines plates-formes, correspond à une autre étape. Nous ne considérons donc ici que la conception abstraite, c'est-à-dire celle qui est réalisable sans connaissance aucune des techniques d'implémentation.

MDA et UML

MDA considère que les modèles d'analyse et de conception doivent être indépendants de toute plate-forme d'implémentation, qu'elle soit J2EE, .Net, PHP, etc. En n'intégrant les détails d'implémentation que très tard dans le cycle de développement, il est possible de maximiser la séparation des préoccupations entre la logique de l'application et les techniques d'implémentation.

UML est préconisé par l'approche MDA comme étant le langage à utiliser pour réaliser des modèles d'analyse et de conception indépendants des plates-formes d'implémentation. C'est pourquoi dans le vocabulaire MDA ces modèles sont appelés des PIM (Platform Independent Model).

Précisons que MDA ne fait que préconiser l'utilisation d'UML et qu'il n'exclut pas que d'autres langages puissent être utilisés. De plus, MDA ne donne aucune indication quant au nombre de modèles à élaborer ni quant à la méthode à utiliser pour élaborer ces PIM.

Quels que soient le ou les langages utilisés, le rôle des modèles d'analyse et de conception est d'être pérennes et de faire le lien entre le modèle d'exigences et le code de l'application. Ces modèles doivent par ailleurs être productifs puisqu'ils constituent le socle de tout le processus de génération de code défini par MDA. La productivité des PIM signifie qu'ils doivent être suffisamment précis et contenir suffisamment d'information pour qu'une génération automatique de code soit envisageable.

Le modèle de code ou de conception concrète PSM (Platform Specific Model)

Une fois les modèles d'analyse et de conception réalisés, le travail de génération de code peut commencer. Cette phase, la plus délicate du MDA, doit elle aussi utiliser des modèles. Elle inclut l'application des patrons de conception techniques.

MDA considère que le code d'une application peut être facilement obtenu à partir de modèles de code. La différence principale entre un modèle de code et un modèle d'analyse ou de conception réside dans le fait que le modèle de code est lié à une plate-forme

d'exécution. Dans le vocabulaire MDA, ces modèles de code sont appelés des PSM (Platform Specific Model).

Les modèles de code servent essentiellement à faciliter la génération de code à partir d'un modèle d'analyse et de conception. Ils contiennent toutes les informations nécessaires à l'exploitation d'une plate-forme d'exécution, comme les informations permettant de manipuler les systèmes de fichiers ou les systèmes d'authentification.

Il est parfois difficile de différencier le code des applications des modèles de code. Pour MDA, le code d'une application se résume à une suite de lignes textuelles, comme un fichier Java, alors qu'un modèle de code est plutôt une représentation structurée incluant, par exemple, les concepts de boucle, condition, instruction, composant, événement, etc. L'écriture de code à partir d'un modèle de code est donc une opération assez triviale.

Pour élaborer des modèles de code, MDA propose, entre autres, l'utilisation de profils UML. Un profil UML est une adaptation du langage UML à un domaine particulier. Par exemple, le profil UML pour EJB est une adaptation d'UML au domaine des EJB. Grâce à ce profil, il est possible d'élaborer des modèles de code pour le développement d'EJB.

Le rôle des modèles de code est principalement de faciliter la génération de code. Ils sont donc essentiellement productifs mais ne sont pas forcément pérennes. L'autre caractéristique importante des modèles de code est qu'ils font le lien avec les plates-formes d'exécution. Cette notion de plate-forme d'exécution est très importante dans MDA car c'est elle qui délimite la fameuse séparation des préoccupations.

Transformation des modèles

Nous venons de passer en revue les trois types de modèles les plus importants pour MDA que sont les CIM, PIM et PSM. Nous avons aussi vu qu'il était important de bien établir les liens de traçabilité entre ces modèles. En fait, MDA établit ces liens automatiquement grâce à l'exécution de transformations des modèles.

Les transformations de modèles préconisées par MDA sont essentiellement les transformations CIM vers PIM et PIM vers PSM. La génération de code à partir des PSM n'est quant à elle pas considérée comme une transformation de modèle à part entière. MDA envisage aussi les transformations inverses : code vers PSM, PSM vers PIM et PIM vers CIM.

Nous ne saurions trop souligner l'importance des transformations de modèles. Ce sont elles qui portent l'intelligence du processus méthodologique de construction d'application. Elles sont stratégiques et font partie du savoir-faire de l'entreprise ou de l'organisation qui les exécute, car elles détiennent les règles de qualité de développement d'applications.

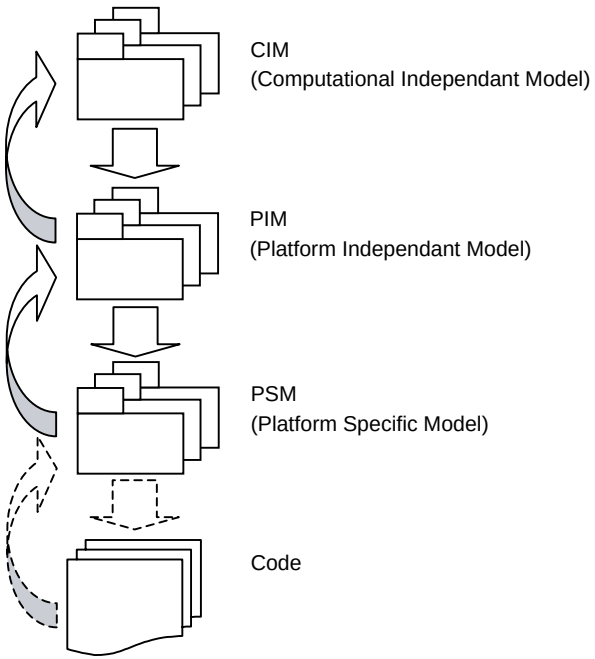
Conscient de cela, MDA préconise de modéliser les transformations de modèles elles-mêmes. Après tout, une transformation de modèles peut être considérée comme une application. Il est dès lors naturel de modéliser ses exigences, son analyse et sa conception et ses modèles de code afin de générer automatiquement le code de la transformation.

Architecture générale de l'approche MDA

La figure 1.1 donne une vue générale de l'approche MDA. Nous constatons que la construction d'une nouvelle application commence par l'élaboration d'un ou de plusieurs modèles d'exigences (CIM). Elle se poursuit par l'élaboration des modèles d'analyse et de conception abstraite de l'application (PIM). Ceux-ci doivent en théorie être partiellement générés à partir des CIM afin que des liens de traçabilité soient établis. Les modèles PIM sont des modèles pérennes, qui ne contiennent aucune information sur les plates-formes d'exécution.

Pour réaliser concrètement l'application, il faut ensuite construire des modèles spécifiques des plates-formes d'exécution. Ces modèles sont obtenus par une transformation des PIM en y ajoutant les informations techniques relatives aux plates-formes. Les PSM n'ont pas pour vocation d'être pérennes. Leur principale fonction est de faciliter la génération de code. La génération de code à partir des modèles PSM n'est d'ailleurs pas réellement considérée par MDA. Celle-ci s'apparente plutôt à une traduction des PSM dans un formalisme textuel.

Figure 1.1
Aperçu global de l'approche MDA



Si la vocation première de l'approche MDA est de faciliter la création de nouvelles applications, elle procure en outre de nombreux avantages pour la rétroconception d'applications existantes. C'est pourquoi les transformations inverses — PSM vers PIM et PIM vers

CIM — sont aussi identifiées. Ces transformations n'en sont toutefois encore qu'au stade de la recherche.

Technologies de modélisation

Nous venons de voir la primauté des modèles dans l'approche MDA. C'est cela qui la différencie principalement des approches classiques de génie logiciel telles que OMT (Object Management Technique), OOSE (Object Oriented Software Engineering) ou BCF (Business Component Factory), qui placent les objets ou les composants au premier plan.

Nous avons vu aussi que MDA préconisait l'élaboration de différents modèles, modèle d'exigences CIM, modèle d'analyse et de conception abstraite PIM et modèle de code et de conception concrète PSM. En réalité, MDA est beaucoup plus général et préconise de modéliser n'importe quelle information nécessaire au cycle de développement des applications. Nous pouvons donc trouver des modèles de test, de déploiement, de plate-forme, etc.

Afin de structurer cet ensemble de modèles, MDA définit la notion de formalisme de modélisation.

Le formalisme de modélisation MOF (Meta Object Facility)

Un formalisme de modélisation est un langage qui permet d'exprimer des modèles. Chaque modèle est donc exprimé dans un certain formalisme de modélisation. Les modèles d'exigences ont leur propre formalisme, qui est différent du formalisme permettant l'expression des modèles d'analyse et de conception abstraite. Rappelons d'ailleurs que le formalisme préconisé pour l'expression des modèles d'analyse et de conception est UML.

Un formalisme définit les concepts ainsi que les relations entre concepts nécessaires à l'expression de modèles. Nous reviendrons largement sur ce sujet dans la suite de l'ouvrage. Le formalisme UML d'expression des modèles d'analyse et de conception définit, entre autres, les concepts de classe et d'objet ainsi que la relation précisant qu'un objet est l'instance d'une classe.

Les notions de modèles et de formalisme de modélisation ne sont pas suffisantes pour mettre en œuvre MDA. Nous avons vu qu'il était aussi très important de pouvoir exprimer des liens de traçabilité ainsi que des transformations entre modèles. Pour pouvoir faire cela, il est indispensable de travailler non pas uniquement au niveau des modèles, mais aussi au niveau des formalismes de modélisation. Il faut exprimer des liens entre les concepts des différents formalismes. Par exemple, il faut pouvoir exprimer que le concept de classe UML doit être transformé dans le concept de classe Java.

Pour cela, MDA préconise de modéliser les formalismes de modélisation eux-mêmes. L'objectif est de disposer d'un formalisme permettant l'expression de modèles de formalismes de modélisation. Dans le jargon de MDA, un tel formalisme est appelé un

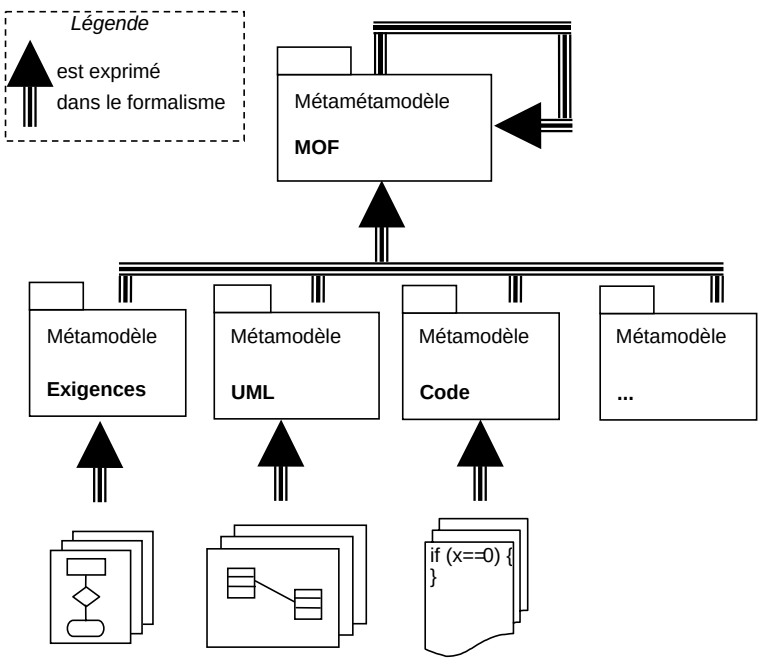
métaformalisme, et les modèles qu'il permet d'exprimer sont appelés des *métamodèles*. Nous pouvons donc faire une analogie entre métamodèles et formalismes de modélisation.

La question qui se pose alors est de savoir s'il est possible de construire un *métaméta-formalisme* permettant d'exprimer des métaformalismes, voire un *métamétaméta…-formalisme*, et ainsi de suite. MDA répond que seuls trois niveaux sont nécessaires : le modèle, le formalisme de modélisation, aussi appelé métamodèle, et le métaformalisme. Pour enrayer la montée dans les niveaux méta, MDA fait en sorte que le métaformalisme soit à lui-même son propre formalisme. Un métamétaformalisme n'est dès lors plus nécessaire.

Dans MDA, il n'existe qu'un seul métaformalisme, le MOF (Meta Object Facility). Aussi appelé *métamétamodèle*, le MOF permet d'exprimer des formalismes de modélisation, ou métamodèles, permettant eux-mêmes d'exprimer des modèles.

La figure 1.2 illustre ces concepts de modèle, de formalisme de modélisation (métamodèle) et de métaformalisme (métamétamodèle). Dans la suite de l'ouvrage, nous utilisons plutôt le vocabulaire MDA, c'est-à-dire modèle, métamodèle et métamétamodèle.

Figure 1.2
Modèle, métamodèle (formalisme de modélisation) et métamétamodèle (métaformalisme)



Le chapitre 2 développe en détail les caractéristiques du MOF. Il précise les relations entre modèles, métamodèles et métamétamodèle et explique comment élaborer un méta-modèle.

Métaformalisme

L'idée d'un métaformalisme n'est pas neuve en soi pour qui est habitué à manipuler des grammaires de langage. Dans le monde XML, un formalisme est représenté sous forme de schéma XML. Un schéma XML définit les concepts ainsi que les relations entre concepts permettant l'expression de documents XML, et les schémas XML sont eux-mêmes des documents XML. Cela n'est possible que parce qu'il existe un schéma des schémas XML, autrement dit un métaformalisme. Dans le monde XML, le schéma des schémas XML est aussi le dernier niveau car il est à lui-même son propre schéma.

La même analogie peut être faite entre les langages de programmation et la BNF (Bachus Naur Form), le langage utilisé pour exprimer les syntaxes des langages. Dans pratiquement tous les manuels de référence des langages (Java, Ada, C++, etc.), on trouve une définition BNF. Par exemple, un programme Java est exprimé dans le formalisme Java qui est défini par une grammaire en BNF, cette dernière se définissant elle-même.

Le métamodèle UML

La section précédente a introduit les notions de modèle, métamodèle et métamétamodèle. Nous avons vu que MDA préconisait l'utilisation de différents modèles et que chacun de ces modèles était conforme à un métamodèle, lui-même conforme au métamétamodèle MOF.

L'univers de MDA est donc partitionné par un ensemble de métamodèles. Chacun de ces métamodèles est dédié à une étape particulière de MDA (exigences, analyse et conception, code). D'un point de vue purement théorique, MDA n'impose aucune contrainte quant à l'utilisation de tel ou tel métamodèle pour chacune de ces étapes. Il n'en va pas de même pour la réalisation, puisque MDA préconise actuellement l'utilisation du métamodèle UML pour l'étape d'analyse et de conception abstraite et conseille de recourir aux profils UML pour élaborer des modèles de code et de conception concrète à partir de modèles UML.

UML pour les PIM

Le métamodèle UML est le métamodèle le plus connu de l'approche MDA. Le sujet de cet ouvrage n'étant pas UML, nous nous contenterons de préciser son rôle dans MDA.

Le métamodèle UML définit la structure que doit avoir tout modèle UML. Il précise, par exemple, qu'une classe UML peut avoir des attributs et des opérations. Le chapitre 3 donne une présentation plus détaillée du métamodèle UML.

D'un point de vue plus conceptuel, le métamodèle UML permet d'élaborer des modèles décrivant des applications objet. UML définit plusieurs diagrammes permettant de décrire les différentes parties d'une application objet. Par exemple, les diagrammes de classes permettent de décrire la partie statique des applications alors que les diagrammes de séquences ou d'activités permettent d'en définir la partie dynamique.

Les modèles UML sont indépendants des plates-formes d'exécution. Ils sont utilisés pour décrire aussi bien les applications Java que les applications C# ou PHP. Plusieurs outils du marché proposent des générateurs de code vers ces différents langages de programmation.

Pour toutes ces raisons, il est évident que le métamodèle UML constitue le métamodèle idéal pour l'élaboration des PIM (Platform Independent Model). Rappelons qu'un PIM est un modèle d'analyse et de conception d'une application et qu'il se doit d'être indépendant d'une plate-forme d'exécution.

UML pour les PSM

Le métamodèle UML définit la notion de *profil UML*. Un profil UML permet d'adapter UML à un domaine particulier. Par exemple, le profil UML pour EJB permet d'adapter UML au domaine des EJB. Les modèles UML réalisés selon ce profil ne sont plus vraiment des modèles UML mais plutôt des modèles d'application EJB.

Les profils UML ciblant des plates-formes d'exécution permettent, par définition, d'adapter UML à des plates-formes d'exécution. Le point intéressant à souligner dans un contexte MDA est que les modèles réalisés selon ces profils ne sont plus des modèles indépendants des plates-formes d'exécution mais, au contraire, des modèles dépendants de ces plates-formes. Ces modèles ne sont donc plus des PIM mais des PSM (Platform Specific Model).

Grâce aux profils ciblant des plates-formes d'exécution, il est possible d'utiliser UML pour élaborer des PSM. Ces PSM sont bien des modèles de code et ne peuvent donc être confondus avec du code. Par contre, étant donné qu'ils sont liés aux plates-formes d'exécution, ils facilitent grandement la dernière étape du MDA, qui est la génération de code.

MDA conseille l'utilisation de profils UML pour l'élaboration de PSM car cela a le mérite de faciliter les transformations PIM vers PSM puisque PIM et PSM sont tous deux des modèles UML.

L'autre approche possible, qui est aussi conseillée par MDA, est de définir les métamodèles propres aux plates-formes. Cette autre approche présente l'inconvénient de ne pas faciliter les transformations PIM vers PSM mais a l'avantage d'offrir une plus grande liberté dans l'expression des plates-formes.

Les profils UML pour EJB et UML pour CORBA ont été standardisés par l'OMG. D'autres profils, tels que UML pour Java ou UML pour C#, ne sont pas standardisés mais sont disponibles dans des produits tels que Rational Software Modeler ou Softeam MDA Modeler *(voir le chapitre 8)*.

UML et CIM

L'élaboration des CIM avec UML fait débat. Rappelons qu'un CIM est un modèle d'exigences d'une application. Ceux qui connaissent UML pourraient argumenter que les diagrammes de cas d'utilisation UML peuvent être considérés comme des CIM. Ces diagrammes permettent en effet de décrire les fonctionnalités qu'offre une application à son environnement. Il existe cependant d'autres approches d'expression des besoins, telles que celles supportées par DOORS ou Rational Requisite Pro. Voilà pourquoi MDA ne fait aucune préconisation quant à l'utilisation d'UML ou non pour exprimer les CIM.

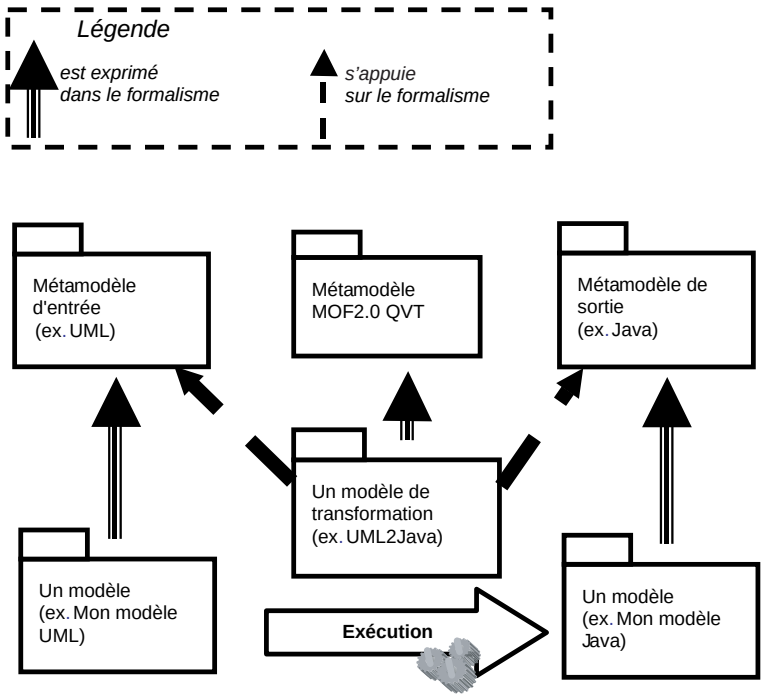
Modélisation de la transformation de modèles avec QVT

Nous avons déjà mentionné le fait que les transformations de modèles étaient au cœur de MDA et qu'il était important de les modéliser. Pour ce faire, l'OMG a élaboré le standard MOF2.0 QVT (Query, View, Transformation). Ce standard définit le métamodèle permettant l'élaboration de modèles de transformation. La transformation d'UML vers la plate-forme Java, par exemple, est élaborée sous la forme d'un modèle de transformation conforme au métamodèle MOF2.0 QVT.

Le standard MOF2.0 QVT cible les transformations modèle vers modèle. Pour simplifier, disons qu'un modèle de transformation est considéré comme une fonction prenant un modèle en entrée et rendant un modèle en sortie. Étant donné que les modèles d'entrée et de sortie disposent chacun de leur métamodèle, on parle aussi de métamodèle d'entrée et de métamodèle de sortie du modèle de transformation.

La figure 1.3 illustre les relations existantes entre le métamodèle MOF2.0 QVT, un modèle de transformation (UML2Java), son métamodèle d'entrée (UML) et de sortie (Java) et une exécution du modèle de transformation.

Figure 1.3
Transformation de modèles



Les métamodèles d'entrée et de sortie d'un modèle de transformation peuvent être identiques. Un modèle de transformation peut bien évidemment transformer des modèles UML en… modèles UML.

Les profils UML qui peuvent être utilisés pour élaborer les modèles PSM sont considérés comme des métamodèles à part entière. Par exemple, un modèle de transformation peut avoir le métamodèle UML comme métamodèle d'entrée et le profil UML pour EJB comme métamodèle de sortie. Un tel modèle de transformation peut spécifier une transformation UML vers EJB.

Il n'y a pas vraiment de consensus sur le statut de la génération de code. Celle-ci a beau être considérée comme une transformation modèle vers modèle par certains, cela ne fait pas l'unanimité. Il est souvent plus pratique de s'exprimer sous forme textuelle pour manipuler le code que sous forme de métamodèle. Un standard en cours de construction actuellement à l'OMG permettra d'ailleurs de définir un langage spécifiquement dédié à la génération de code à partir de modèles.

Le chapitre 7 détaille les transformations de modèles dans MDA et présente plus précisément les différentes approches possibles pour réaliser une transformation de modèle et les illustre à partir d'un même exemple.

Liens vers XML et Java avec XMI, JMI et EMF

Les modèles sont des entités abstraites qui n'ont pas besoin de représentation informatique pour exister. MDA utilisant les modèles à des fins de productivité, il est cependant nécessaire qu'ils disposent de représentations concrètes afin de pouvoir être manipulés informatiquement.

MDA définit deux façons différentes de représenter les modèles : soit sous forme de documents textuels, soit sous forme d'objets de programmation. La représentation textuelle est plus adaptée au stockage des modèles sur disque dur ou aux échanges de modèles entre applications tandis que la représentation objet est plus adaptée à la manipulation informatique (transformation, exécution, validation, etc.).

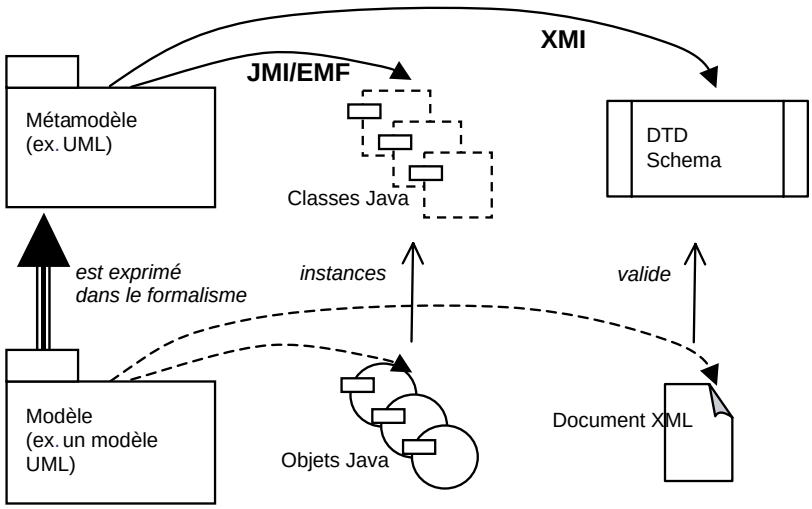
Les standards et frameworks MDA qui définissent la façon de représenter informatiquement les modèles sont XMI, JMI et EMF. Le standard XMI (XML Metadata Interchange) définit la façon de représenter les modèles sous forme de document XML. Le standard JMI (Java Metadata Interface) et le framework EMF (Eclipse Modeling Framework) définissent la façon de représenter les modèles sous forme d'objets Java. EMF, par exemple, permet la manipulation des modèles dans la plate-forme Eclipse.

XMI, JMI et EMF fonctionnent selon le même principe. Que ce soit en XML ou en Java, un format de représentation se définit par sa structure. Définir un format de représentation XML se fait en construisant une DTD ou un schéma XML. Définir un format de représentation objet se fait en construisant un ensemble de classes Java.

Le principe de fonctionnement de XMI, JMI et EMF est de générer automatiquement la structure des formats de représentation des modèles à partir de leur métamodèle. L'idée est de tirer parti de l'analogie qui existe entre la relation entre un modèle et son métamodèle et la relation qui existe entre un document XML et sa DTD ou entre des objets et leur classe.

La figure 1.4 illustre ce principe de fonctionnement. Nous constatons que XMI, JMI et EMF définissent des règles permettant de générer automatiquement les structures des formats de représentation de modèles à partir de leur métamodèle. Par exemple, XMI appliqué à UML permet la génération automatique d'une DTD permettant de représenter les modèles UML sous forme de documents XML. De la même manière, JMI et EMF permettent la génération automatique de classes Java permettant de représenter les modèles UML sous forme d'objets Java.

Figure 1.4
Principe de fonctionnement des standards XMI et JMI



Grâce aux standards XMI et JMI et au framework EMF, il est possible de représenter informatiquement tout modèle. XMI concrétise la pérennité des modèles en ce qu'il offre un format de représentation XML. JMI et EMF sont les socles opérationnels de MDA en ce qu'ils permettent la construction d'opérations de productivité sur les modèles.

Précisons qu'il existe des passerelles entre ces deux standards et qu'il est possible de passer d'une représentation à une autre sans difficulté.

Le chapitre 5 présente en détail les principes de fonctionnement du standard XMI illustrés par un exemple, tandis que le chapitre 6 se penche sur JMI et EMF et explique à partir d'un exemple comment manipuler un modèle à l'aide d'un programme Java.

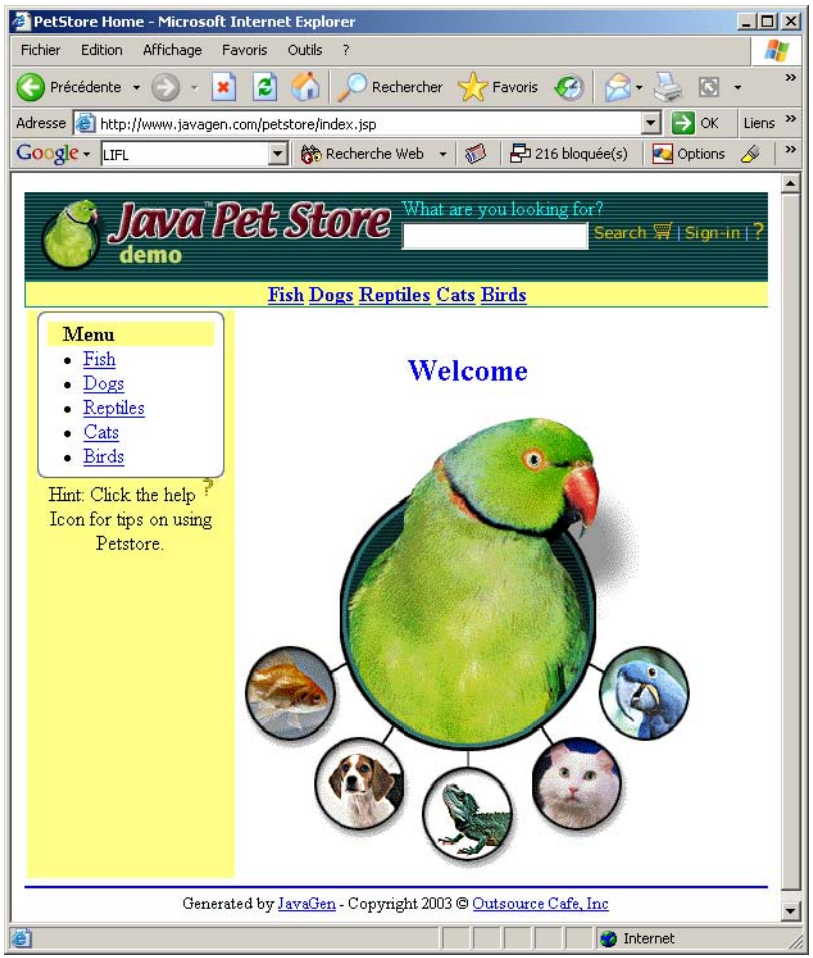
L'étude de cas PetStore

Les sections précédentes ont présenté l'approche MDA dans sa globalité ainsi que les technologies utilisées. Afin d'illustrer concrètement MDA, nous l'avons appliqué à une étude de cas.

L'étude de cas retenue est la fameuse application Java PetStore des BluePrints de Sun. PetStore est une application relativement classique de commerce électronique. Un utilisateur peut naviguer sur un site Web pour voir quels sont les différents articles à la vente afin d'ajouter dans son panier ceux qu'il désire acheter. Quand il le décide, il peut passer commande et régler tous les articles qu'il a mis dans son panier. La figure 1.5 illustre la page d'accueil de cette application.

Nous avons choisi PetStore car c'est une application très documentée sur le Web. Même si l'objectif initial de cette application était de servir à tester les serveurs d'applications J2EE, PetStore est aujourd'hui utilisée pour des objectifs beaucoup plus étendus, tels que la comparaison .Net, Java et PHP. Grâce à cette documentation présente sur le Web, nous pouvons ne pas trop nous attarder sur la description de PetStore et nous concentrer sur notre objectif, qui est d'illustrer l'approche MDA.

Figure 1.5
Page d'accueil du site Web de PetStore



Pour illustrer comment appliquer l'approche MDA et souligner les avantages concrets qu'elle apporte, nous allons détailler les différents modèles et transformations de modèles nécessaires à la construction de cette application selon les principes MDA.

Nous avons fait le choix de présenter les modèles CIM et PIM de cette application sous forme de modèles UML. Nous préférons utiliser l'approche qui consiste à considérer que les CIM peuvent être élaborés en utilisant UML car cela facilite la présentation de l'approche MDA sur une application de la taille de PetStore. Le seul modèle pérenne de PetStore est donc un modèle UML qui conjugue CIM et PIM.

Conformément à l'approche MDA, ce PIM de PetStore a dû être transformé vers différentes plates-formes d'exécution. Nous avons sélectionné les plates-formes J2EE/EJB et PHP. Le fait d'avoir choisi deux plates-formes nous permet de mettre en lumière les solutions qu'offre MDA pour résoudre la séparation des préoccupations entre le métier et la technique d'une application.

Le travail le plus important pour mettre en œuvre l'approche MDA réside dans l'élaboration du PIM et dans la définition des transformations PIM vers PSM. Pour PetStore, nous avons élaboré les transformations UML vers J2EE/EJB et UML vers PHP permettant de transformer le modèle PIM de PetStore vers les modèles PSM correspondant à ces plates-formes.

Concernant ces plates-formes, nous avons fait le choix d'utiliser les deux techniques possibles, à savoir l'utilisation de profils UML et de métamodèles MOF. Le choix d'un profil pour EJB et d'un métamodèle pour PHP permet de bien comprendre les différences existant entre ces deux approches préconisées par MDA pour représenter les plates-formes.

Après avoir transformé les PIM vers des PSM, nous effectuerons manuellement différentes opérations de raffinement sur les PSM afin qu'ils puissent servir à la génération de code. En effet, les transformations PIM vers PSM ne permettant pas d'obtenir un modèle directement exploitable, il est nécessaire de raffiner encore ce modèle afin d'effectuer la dernière opération, qui est la génération de code.

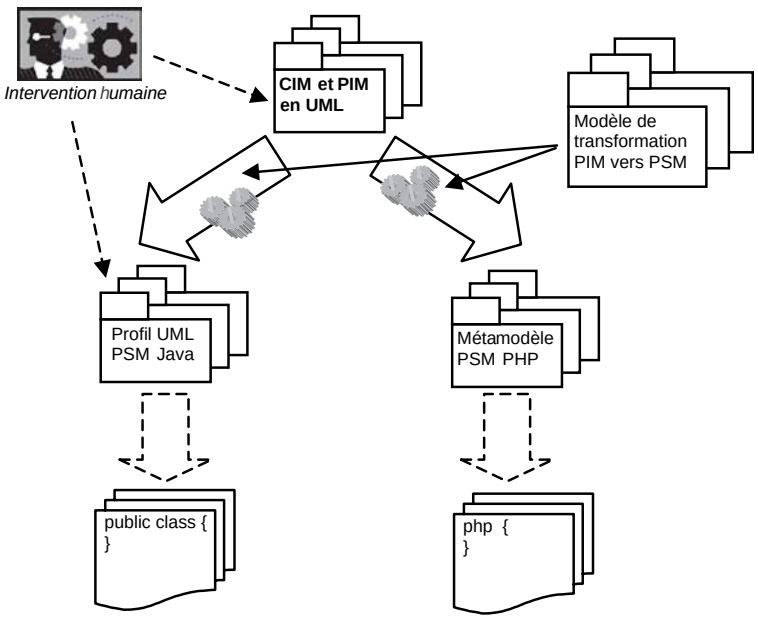
Pour finir, nous avons généré le code des deux PSM obtenus. Cette dernière étape nous permet d'illustrer les solutions qu'offre MDA pour s'abstraire des langages de programmation. Elle nous permet en outre de clarifier la portée de MDA en précisant ce qu'il est possible de faire aujourd'hui avec MDA et ce qu'il n'est pas possible de faire.

La figure 1.6 illustre les différentes étapes de MDA réalisées sur l'étude de cas PetStore, notamment les efforts d'élaboration d'un PIM, la transformation de ce PIM en plusieurs PSM (Java et PHP) et la génération de code à partir de ces PSM.

L'étude de cas PetStore nous permettra de mesurer la taille du travail à fournir pour passer vers une autre plate-forme, ici .Net.

Le chapitre 12 et dernier de l'ouvrage donne tous les détails de la réalisation de PetStore selon les principes de MDA.

Figure 1.6
Application de MDA à l'étude de cas PetStore



Avantages attendus de MDA

Maintenant que nous avons introduit l'approche MDA et que nous avons brièvement décrit la façon dont nous l'illustrerons sur une étude de cas, prenons un peu de recul afin de mesurer pleinement les avantages attendus de cette approche.

Commençons par rappeler brièvement ce qui s'est passé ces dernières années dans le domaine de la construction des applications réparties.

Les premiers travaux effectués à l'OMG pour faciliter la construction et la maintenance des applications réparties ont porté sur l'élaboration de la spécification CORBA. L'objectif de CORBA était de fournir un environnement standard et ouvert permettant à tout type d'application d'interopérer. À l'époque (1990), il fallait résoudre tous les problèmes d'interopérabilité en standardisant les communications réparties et les interfaces des différentes entités composant l'application répartie.

Pour diverses raisons, CORBA n'a pas été un franc succès. D'autres intergiciels ont à leur tour tenté de résoudre le problème de l'interopérabilité en offrant toujours plus de services aux concepteurs d'applications (EJB, DCOM, Web Services).

De cette succession d'intergiciels est né le *paradoxe des intergiciels.* En effet, les intergiciels qui ont été initialement conçus pour faire face à la complexité des applications réparties ont finalement apporté beaucoup plus de complexité qu'ils n'en ont enlevée. Le problème n'est pas tant leur propre complexité, qui est inévitable, que le fait que les

applications qui les utilisent dépendent fortement des services offerts par ces intergiciels. De ce fait, changer d'intergiciel devient un cauchemar tant l'application est engluée dans l'intergiciel. Certains vont jusqu'à appeler *spaghettiware* les applications réparties qui utilisent les intergiciels. L'évolution d'une application a un coût prohibitif, car il faut inévitablement plonger en son sein pour couper tous les liens avec les intergiciels que l'on veut changer.

Pour résoudre ce problème, la solution envisagée a consisté à tout mettre en œuvre pour assurer de manière industrielle la fameuse séparation des préoccupations entre le métier des applications et la technique des intergiciels. De là est né MDA.

Les trois avantages recherchés étaient alors les suivants :

- La pérennité des savoir-faire, afin de permettre aux entreprises de capitaliser sur leur métier sans avoir à se soucier de la technique.
- Les gains de productivité, afin de permettre aux entreprises de réduire les coûts de mise en œuvre des applications informatiques nécessaires à leur métier.
- La prise en compte des plates-formes d'exécution, afin de permettre aux entreprises de bénéficier des avantages des plates-formes sans souffrir d'effets secondaires.

Pérennité des savoir-faire

Le métier des entreprises n'évolue que faiblement par rapport aux technologies qu'elles utilisent pour construire leurs applications informatiques. Fort de ce constat, il est évident que séparer les aspects métier des aspects techniques lors de la construction d'une application est une bonne pratique. Cela permet de rendre les spécifications métier pérennes, indépendamment des spécifications techniques.

L'avantage le plus important qu'offre MDA est précisément celui de la pérennité des spécifications métier. Son ambition est de faire en sorte que les PIM aient une durée de vie de plus de dix ans.

Nous comprenons mieux pourquoi MDA s'appuie si fortement sur la modélisation des spécifications métier. Par nature, les modèles sont des entités facilement pérennes, car ils permettent de représenter des informations à différents niveaux d'abstraction tout en masquant les détails inutiles. C'est donc la modélisation des spécifications métier qui permet de les rendre pérennes.

Le point le plus important pour rendre des modèles métier pérennes est le formalisme utilisé pour les élaborer. Ce formalisme doit être particulièrement stable et sa sémantique largement partagée par les personnes qui utiliseront les modèles.

Sur ce point, remarquons que MDA a pris ses précautions. Tout d'abord, MDA préconise l'utilisation du langage UML. Fruit de plusieurs années de travail collaboratif des différents acteurs du domaine, nous pouvons considérer qu'il est stable et que sa sémantique est largement partagée. De plus, l'OMG ne fait que préconiser l'utilisation d'UML. Il n'est donc pas lié définitivement avec ce langage. Si, demain, un nouveau langage apparaît

pour remplacer UML, il sera toujours possible de transformer les modèles UML vers ce nouveau langage. Les modèles PIM sont donc nécessairement pérennes.

Un autre critère important de pérennité des modèles est le rôle qu'ils jouent dans le cycle de vie de l'application. Rien ne sert de rendre un modèle pérenne si celui-ci n'est d'aucune utilité. N'est pérenne que l'information utile. Cela contredit l'opinion trop répandue selon laquelle seul le code est utile. Certes le code contient l'information nécessaire et suffisante pour faire en sorte que l'application tourne, mais est-il pour autant la seule source d'information utile ?

À cet égard, MDA propose une approche assez novatrice, consistant à faire remonter l'information utile dans les modèles. Le code n'est dès lors considéré que comme une représentation textuelle interprétable par les machines de l'information utile. Dans MDA, l'information utile est localisée pour l'essentiel dans les PIM et, dans une moindre mesure, dans les transformations PIM vers PSM. Insistons une nouvelle fois sur l'importance des liens de traçabilité entre modèles afin que l'approche soit réaliste.

Abstraction de l'information utile

L'approche qui vise à faire remonter en abstraction l'information utile a déjà été utilisée plusieurs fois dans l'histoire de l'informatique. Nous pouvons considérer que cette même approche a été utilisée pour faire remonter dans les langages de programmation tels que le C ou le Pascal l'information utile qui était contenue dans le code assembleur. Nous pourrions même considérer que cette approche a été utilisée pour faire remonter dans le code assembleur l'information qui était contenue dans les cartes perforées.

En ce qui concerne les technologies utilisées par MDA, la pérennité se retrouve quasiment dans tous les standards. Le standard qui porte en lui le plus de pérennité est vraisemblablement UML, qui permet l'expression des PIM. MOF, qui permet d'exprimer les métamodèles, est également gage de pérennité, car c'est grâce à lui que s'accomplit l'indépendance à l'égard des langages de modélisation.

MOF2.0 QVT est lui aussi central pour la pérennité puisque, sans transformation, il ne serait pas possible de faire remonter l'information utile au niveau des PIM. XMI complète ce socle pérenne en permettant de représenter les modèles sous forme de documents XML et en assurant ainsi un stockage « neutre » des modèles.

L'aboutissement de tous ces standards permettra, par exemple, aux développeurs de s'approprier les modèles UML sous une forme de stockage en un « XML neutre », indépendant des outils de modélisation.

Comme le principal apport de MDA est de pérenniser les modèles PIM sur des durées d'environ quinze ans, il lui est souvent reproché de ne concerner que les grosses applications. S'il est vrai que MDA est beaucoup plus profitable à ces dernières qu'aux petites et moyennes, ses principes et technologies n'en restent pas moins intrinsèquement indépendants de la taille des applications. Même des applications constituées de seulement une ou deux classes tirent un bénéfice de l'élaboration d'un modèle UML, lequel peut ensuite être transformé automatiquement vers un modèle de code lié à une plate-forme (CORBA ou J2EE/EJB, par exemple) puis *in fine* vers du code Java ou C#.

Plusieurs outils ciblent d'ailleurs en partie ce marché des petites applications, notamment Poseidon.

Gains de productivité

Il est toujours important pour une entreprise d'augmenter sa productivité afin de rester compétitive. Une façon bien connue d'augmenter la productivité est d'automatiser certaines étapes de production. C'est ce que fait MDA en automatisant les transformations de modèles.

Ce deuxième avantage apporté par MDA de gains substantiels de productivité grâce à l'automatisation des transformations de modèles est très novateur. Avant MDA, les modèles n'étaient que des dessins que l'on accrochait fièrement aux murs de son bureau. Ils permettaient au mieux de communiquer avec les différents membres de l'équipe. À peine imprimés, ces modèles étaient bien souvent en total décalage avec le code de l'application et devenaient inutilisables. Les qualités attendues des modèles relevaient essentiellement de la communication (clarté, lisibilité, simplicité, etc.).

MDA vise une forte intégration des modèles dans le processus de production de l'application puisque les modèles sont directement utilisés pour générer le code de l'application. Les qualités attendues des modèles sont donc tout autres. MDA a besoin de modèles précis, complets et bien définis pour pouvoir les transformer. Le statut des modèles change dès lors du tout au tout, et ils deviennent un élément de production des applications, au même titre que le code ou le binaire.

Pour effectuer ce changement de statut, MDA définit la nature du support informatique (stockage, API) des modèles. Nous avons vu que les standards JMI et le framework EMF permettaient de fournir des représentations concrètes pour les modèles. Ces standards, fondés sur le MOF, permettent d'automatiser les opérations effectuées sur les modèles. Le standard MOF2.0 QVT, par exemple, peut être vu comme un standard qui utilise JMI et EMF en proposant un langage de haut niveau pour exprimer les transformations de modèles.

Les principes à l'œuvre dans ces standards préexistaient à MDA, et certains outils tels que Objecteering ou Rational Rose proposent depuis le début des années 90 d'informatiser les modèles. Ces outils ont défini leur propre format de représentation concret et proposent aux utilisateurs des moyens pour programmer et donc automatiser les opérations sur les modèles.

Ces outils ont été des précurseurs à l'époque de leur conception. Les opérations qu'ils proposent sont principalement de la génération de code et de la génération de documentation sur les modèles UML. Aujourd'hui, de plus en plus d'outils proposent différentes opérations sur différents modèles, comme des opérations de génération et d'exécution de tests, de validation et de vérification, de simulation, d'exécution, de supervision, etc. Ces opérations complexes sont de plus en plus automatisées, engendrant des gains de productivité tout au long des étapes du cycle de vie des applications.

Ce dernier point soulève encore des questions pour savoir quel est le domaine d'application couvert par MDA. À voir se multiplier l'automatisation des opérations sur les modèles, nous pourrions en conclure que MDA est une sorte d'usine à gaz, dans laquelle entre un modèle pour y subir une multitude d'opérations afin qu'en sorte automatiquement une application neuve. En réalité, MDA ne fait aucune préconisation, autre que CIM, PIM et PSM, quant à l'ensemble des opérations à utiliser. Il est donc possible d'utiliser un nombre incalculable d'opérations sur les modèles et de clamer que nous suivons l'approche MDA, alors même que MDA ne préconise pas encore de méthode susceptible de définir clairement quelles sont les bonnes et les mauvaises pratiques.

MDA est une approche, et non une méthode, fondée sur l'utilisation de modèles pour assurer la séparation des préoccupations. Pour ce faire, MDA utilise les modèles comme des éléments productifs. Appliquer réellement MDA nécessite inévitablement de définir une méthode. Une telle méthode, propre à chaque entreprise, permet de bien appliquer MDA dans le contexte de l'entreprise. Aujourd'hui, plusieurs entreprises sont en train de définir leur propre méthode MDA. Toutes suivent l'approche MDA mais diffèrent dans leur degré d'automatisation et dans la diversité des opérations qu'elles proposent.

Dans notre étude de cas, nous utilisons une méthode relativement simpliste, qui consiste à définir en premier lieu les CIM et PIM de l'application sous forme de modèle UML puis à transformer automatiquement ces PIM en PSM, à appliquer manuellement quelques modifications sur ces PSM et à générer automatiquement le code. Cette méthode MDA n'est que partiellement automatisée et donc partiellement productive. Elle permet néanmoins d'obtenir des gains significatifs de productivité.

Prise en compte des plates-formes d'exécution

Les applications réparties d'aujourd'hui s'exécutent de plus en plus sur des plates-formes hétérogènes. Il n'est pas rare de voir une application s'exécuter en partie sur une plate-forme Java et en partie sur une plate-forme .Net, par exemple. Il est important pour les entreprises de dompter cette offre hétérogène afin de tirer parti du meilleur de chaque plate-forme.

MDA prend en compte les plates-formes en intégrant leur description aux modèles lors des transformations PIM vers PSM. Il est dès lors possible d'établir une stratégie propre à l'intégration de telle ou telle plate-forme.

Le troisième avantage qu'offre MDA est d'intégrer le support des plates-formes directement dans les modèles. Cet avantage est peut-être le plus déterminant. L'objectif premier de MDA est de permettre de migrer facilement d'une plate-forme à une autre. Grâce à ce support des plates-formes dans les modèles, nous pouvons penser, d'une part, que la migration de plate-forme sera beaucoup moins coûteuse et, d'autre part, qu'il sera possible de capitaliser le savoir-faire sur les plates-formes. Ces perspectives sont toutefois à conjuguer au futur car elles ne sont pas encore pleinement accomplies à l'heure actuelle.

La prise en compte des plates-formes par MDA se fait en deux endroits : dans les PSM et dans les transformations PIM vers PSM.

Concernant les PSM, souvenons-nous qu'un PSM est un modèle qui dépend fortement d'une plate-forme. Son métamodèle contient toute l'information relative à la plate-forme. Rappelons que ce métamodèle peut être un profil UML tel que ceux que nous avons déjà présentés. Pour l'instant, les métamodèles ou profils de plates-formes ne font qu'énumérer les concepts de la plate-forme nécessaires à connaître pour pouvoir l'utiliser. Par exemple, le métamodèle de la plate-forme J2EE contient les concepts de classe Java, de page JSP, de servlet, d'EJB, etc. Un modèle conforme à ce métamodèle représente une application J2EE.

Il n'existe pas encore de réel métamodèle de plate-forme. Un tel métamodèle permettrait d'élaborer des modèles représentant des plates-formes déployées et d'effectuer toutes sortes d'opérations sur ces modèles, telles que des tests de montée en charge ou de la supervision.

Les transformations PIM vers PSM contiennent toute l'information permettant à un PIM d'utiliser une plate-forme déterminée. Au début de MDA, l'OMG pensait qu'il appartenait aux constructeurs de plate-forme de fournir ces transformations. L'idée était qu'après la sortie d'une nouvelle plate-forme, les transformations permettant de l'exploiter seraient disponibles, engendrant un coût de migration quasi nul.

Peu après la sortie de MDA, de nombreux avis contraires se sont fait entendre, estimant que ce n'était pas aux constructeurs de plate-forme de fournir ces transformations mais plutôt aux utilisateurs, seuls à même de savoir comment exploiter une plate-forme dans leur contexte. Ce deuxième courant s'est avéré beaucoup plus intéressant et beaucoup plus réaliste, chacun capitalisant sa façon d'exploiter les plates-formes.

Ce courant inclut tous ceux qui réalisent les frameworks techniques d'entreprise. Ces frameworks, fondés sur les plates-formes existantes, permettent d'adapter les plates-formes aux besoins des entreprises. Il existe, par exemple, des frameworks techniques d'entreprise sur J2EE, qui empêchent l'utilisation de certaines fonctionnalités de la plate-forme, jugées peu viables pour le contexte de l'entreprise, comme les EJB Entity Container Managed Persistency dans certains frameworks. L'utilisation des frameworks par les applications se retrouve alors dans les transformations PIM vers PSM.

Ce caractère multiplate-forme est certainement l'avantage le plus apprécié par les utilisateurs de MDA. MDA est au demeurant la seule approche qui considère que les applications ne seront jamais plus déployées sur une unique plate-forme.

Synthèse

Ce chapitre a présenté MDA dans ses grandes lignes. Nous avons vu que MDA utilisait fortement les modèles et avons indiqué les différents types de modèles employés par MDA, à savoir CIM, PIM et PSM.

Nous avons présenté brièvement les différentes technologies qui permettent de mettre en œuvre MDA et avons introduit l'application de MDA à l'étude de cas Java PetStore de

Sun. Nous avons dégagé pour finir les principaux avantages de MDA, qui sont la pérennité, la productivité et la prise en compte des plates-formes.

Dans la suite de l'ouvrage, nous détaillons chacun des points soulevés dans ce chapitre. Afin d'organiser ces explications, nous avons choisi d'utiliser les trois avantages de MDA comme grille de lecture.

Partie I

Pérennité des savoir-faire

La pérennité est l'objectif principal de MDA. Il s'agit de faire en sorte que la logique métier des applications ne soit plus mêlée aux considérations techniques de mise en production. Il devient dès lors possible de capitaliser les savoir-faire et d'être beaucoup plus réactif aux changements technologiques.

Pour atteindre cet objectif, MDA vise à représenter sous forme de modèles toute l'information utile et nécessaire à la construction et à l'évolution des applications informatiques. Les modèles apparaissent au centre du cycle de vie des applications.

Pour mettre en œuvre cette pérennité par les modèles, MDA définit une architecture à quatre niveaux, qui pose les concepts de base de l'ingénierie des modèles et positionne chacun des standards qui font MDA, tels que MOF, UML, OCL et AS.

Cette première partie de l'ouvrage consacrée à la pérennité des modèles commence par présenter les concepts fondamentaux de MDA que sont les modèles, métamodèles et métamétamodèles. C'est le standard MOF qui incarne ce socle architectural. Elle introduit ensuite les langages de modélisation tels que UML, OCL et AS, qui permettent de modéliser toute la logique métier des applications informatiques. Pour finir, elle détaille les standards XMI et DI, qui permettent de représenter les modèles sous forme de documents XML et favorisent ainsi leur échange et donc leur pérennité.

2

Modèles et niveaux méta

Si MDA est fondé sur l'utilisation massive des modèles dans toutes les phases du cycle de vie des applications, les métamodèles sont ses éléments de structuration. C'est par leur intermédiaire que MDA assure la pérennité des modèles.

Nous avons vu au chapitre 1 que MDA nécessitait l'utilisation de différents types de modèles (CIM, PIM, PSM, etc.). Chaque type de modèle est élaboré dans un formalisme particulier. MDA a donc besoin de définir de façon précise les différents formalismes qui permettent d'élaborer des modèles à la fois pérennes et productifs.

Conscient de la difficulté inhérente à la définition de formalismes de modélisation, l'OMG a en premier lieu défini le support de définition des formalismes de modélisation. À cette fin, il a conçu le standard MOF (Meta Object Facility), qui apporte le support de définition des formalismes de modélisation sous la forme de métamodèles.

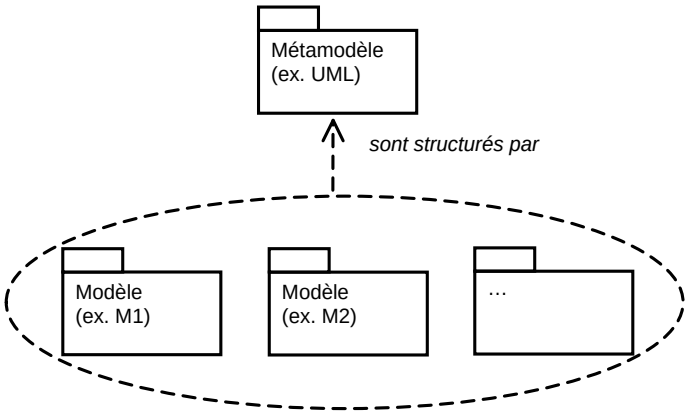
Les métamodèles

Selon MOF, un métamodèle définit la structure que doit avoir tout modèle conforme à ce métamodèle. Autrement dit, tout modèle doit respecter la structure définie par son méta-modèle. Par exemple, le métamodèle UML définit que les modèles UML contiennent des packages, leurs packages des classes, leurs classes des attributs et des opérations, etc.

La figure 2.1 illustre la relation entre un métamodèle et l'ensemble des modèles qu'il structure.

Les métamodèles fournissent la définition des entités d'un modèle, ainsi que les propriétés de leurs connexions et de leurs règles de cohérence. MOF les représente sous forme de *diagrammes de classes.*

Figure 2.1
Relation entre modèles et métamodèles



Rappelons que les diagrammes de classes permettent de représenter les notions d'un domaine et leurs propriétés, que ces notions ou entités soient organisées ou non sous forme d'objets. Cette utilisation des diagrammes de classes a le double avantage de permettre de définir très précisément les métamodèles et de les rendre eux aussi pérennes et productifs.

Un métamodèle est donc une sorte de diagramme de classes qui définit la structure d'un ensemble de modèles. La section suivante de ce chapitre donne différents exemples de métamodèles.

Métamodèle et sémantique d'un modèle

Métamodèle et sémantique d'un modèle sont deux choses différentes. Un métamodèle définit la structure d'un ensemble de modèles mais fournit peu de précision sur leur sémantique. Définir qu'un modèle UML contient des packages, qui, eux-mêmes, contiennent des classes, ne renseigne en rien sur la signification des concepts de package et de classe. MDA ne donne aucune préconisation pour définir la sémantique d'un modèle. Cela se fait principalement par le biais du langage naturel, en l'occurrence l'anglais pour les standards de MDA.

Exemples de métamodèles

Avant de définir précisément et théoriquement ce que sont les métamodèles, il nous paraît important d'en présenter deux exemples. Nous montrerons ainsi à quoi servent les métamodèles et comment ils sont élaborés conceptuellement.

Métamodèle de diagramme de cas d'utilisation

Le premier métamodèle exemple que nous présentons est celui d'une version allégée d'UML. Nous considérerons dans un premier temps que cette version ne contient que des diagrammes de cas d'utilisation.

Notre propos n'est pas d'expliquer comment faire des diagrammes de cas d'utilisation mais de présenter le métamodèle des diagrammes de cas d'utilisation. Nous nous intéressons uniquement à la structure de ces diagrammes.

La définition suivante des diagrammes de cas d'utilisation n'est pas standard, mais nous la proposons afin de faciliter notre présentation :

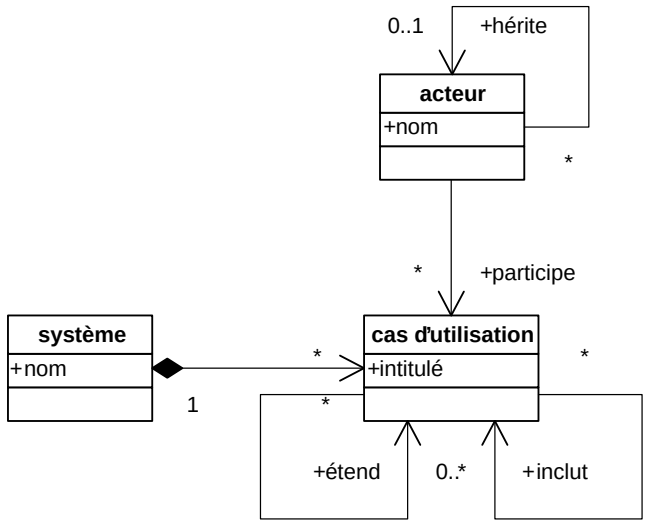
> « Un diagramme de cas d'utilisation contient des **acteurs,** un **système** et des **cas d'utilisation.** Un acteur a un nom et est *relié* aux cas d'utilisation. Un acteur peut *hériter* d'un autre acteur. Un cas d'utilisation a un intitulé et peut *étendre* ou *inclure* un autre cas d'utilisation. Le système a lui aussi un nom, et il inclut tous les cas d'utilisation. »

Les informations représentant les concepts fondamentaux des diagrammes de cas d'utilisation sont en gras et les relations existantes entre ces concepts en italique.

Le métamodèle correspondant à ces diagrammes de cas d'utilisation est une sorte de diagramme de classes, dans lequel chaque concept fondamental est représenté à l'aide d'une classe et chaque relation existante entre concepts à l'aide d'une association.

La figure 2.2 illustre ce métamodèle. Il contient trois classes, `acteur`, `système` et `cas d'utilisation`. La classe `acteur` possède un attribut nommé `nom`, dont le type est une chaîne de caractères. La classe `cas d'utilisation` possède un attribut nommé `intitulé`, dont le type est une chaîne de caractères. La classe `système` possède un attribut nommé `nom`, dont le type est une chaîne de caractères. Ce métamodèle contient une association nommée `hérite`, qui a la classe `acteur` comme source et comme cible. Il contient aussi deux associations, `étend` et `inclut`, qui ont pour source et pour cible la classe `cas d'utilisation`. Il contient enfin une relation d'agrégation entre la classe `système` et la classe `cas d'utilisation`.

Figure 2.2
Métamodèle des diagrammes de cas d'utilisation

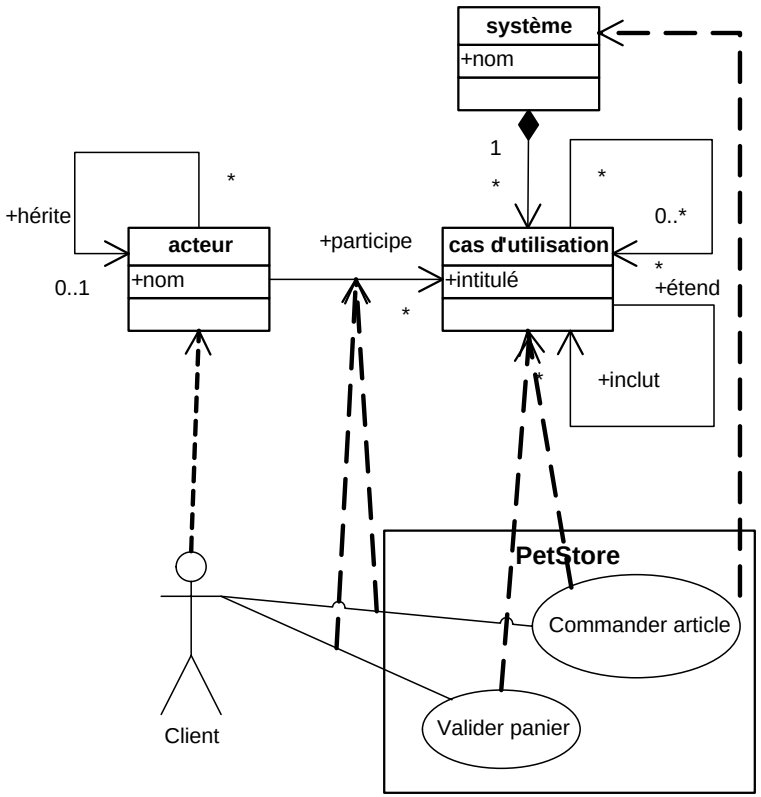


Ce métamodèle définit que les modèles en conformité avec lui ne peuvent contenir que des acteurs, des systèmes et des cas d'utilisation. Un acteur a un nom, un cas d'utilisation a un intitulé et un système a un nom. Ce métamodèle définit aussi que les modèles conformes peuvent faire apparaître des relations d'héritage entre les acteurs et qu'ils peuvent en outre faire apparaître des relations d'extension et d'inclusion entre cas d'utilisation. Les cas d'utilisation de ces modèles doivent en outre être contenus dans un système.

Ce métamodèle définit donc bien la structure des diagrammes de cas d'utilisation telle que nous l'avons énoncée en langage naturel précédemment.

La figure 2.3 illustre la relation qui existe entre ce métamodèle et un exemple de diagramme de cas d'utilisation. Les flèches en pointillés représentent les relations existantes entre les éléments du métamodèle et les éléments du diagramme de cas d'utilisation. Ces relations peuvent être considérées comme des relations d'*instanciation.* On dit alors qu'un modèle est l'instance de son métamodèle.

Figure 2.3
Relations entre un diagramme de cas d'utilisation et son métamodèle



Comme nous le voyons, le diagramme de cas d'utilisation contient un acteur (instance de la classe `acteur` du métamodèle), deux cas d'utilisation (instances de la classe `cas`

d'utilisation du métamodèle) et un système (instance de la classe système du métamodèle). Le nom de l'acteur est client. Les intitulés des cas d'utilisation sont Commander article et Valider panier.

L'acteur de ce diagramme est relié aux deux cas d'utilisation, conformément à l'association présente dans le métamodèle entre les classes acteur et cas d'utilisation. Le système nommé PetStore inclut les deux cas d'utilisation, conformément à l'association présente dans le métamodèle entre les classes système et cas d'utilisation.

Métamodèle de diagramme de classes

Nous allons développer notre exemple en ajoutant à notre version allégée d'UML des diagrammes de classes simplifiés (package, classes, attributs).

L'information qui nous intéresse est la suivante :

> « Un diagramme de classes contient des **packages.** Un package a un nom et contient des **classes.** Un package peut *importer* un autre package. Une classe a un nom et peut *contenir* des **attributs.** Une classe peut aussi *hériter* d'une autre classe. Un attribut a un nom et une visibilité qui peut être soit public soit private. Un attribut *a* un **type** qui peut être soit un **type de base (string, integer, boolean),** soit une classe du diagramme. »

La figure 2.4 illustre le métamodèle des diagrammes de classes. Il est composé de huit classes. La classe package contient un attribut nommé nom, dont le type est une chaîne de caractères. La classe class contient un attribut nommé nom, dont le type est une chaîne de caractères. La classe class hérite de la classe type. La classe attribute contient un attribut nommé nom, dont le type est une chaîne de caractères, et un attribut nommé visibilité, dont le type est une énumération (public ou private). Les classes string, integer et boolean héritent de la classe basicType, qui hérite de la classe type.

Il existe une association nommée import, qui a pour source et pour cible la classe package, ainsi qu'une association nommée super, qui a pour source et pour cible la classe class, une association nommée type, entre les classes attribute et type, et deux associations d'agrégation entre les classes package et class et entre les classes class et attribute.

Ce métamodèle définit que les modèles conformes ne peuvent contenir que des packages qui contiennent des classes contenant des attributs. Packages, classes et attributs ont des noms. Les packages peuvent importer d'autres packages et les classes hériter d'autres classes, tandis que le type d'un attribut peut être soit un type de base, soit une classe.

Ce métamodèle définit donc bien la structure des diagrammes de classes telle que nous l'avons énoncée en langage naturel précédemment.

Ces deux exemples nous ont permis de démystifier les métamodèles. Nous avons vu qu'un métamodèle était une sorte de diagramme de classes permettant de représenter les concepts fondamentaux d'un langage de modélisation sous forme de classes et de représenter les relations existantes entre ces concepts sous forme d'associations.

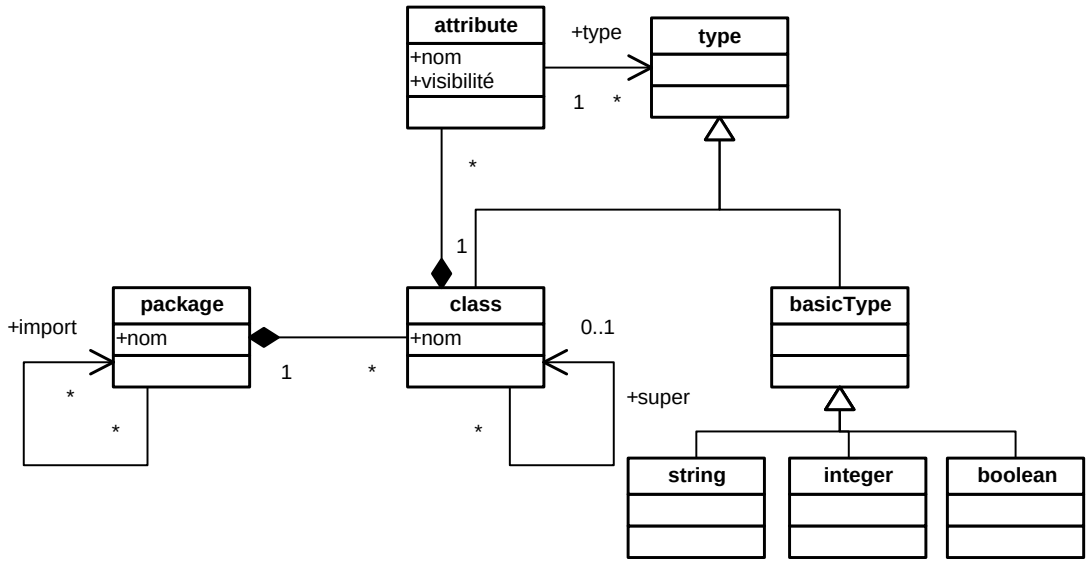


Figure 2.4
Métamodèle des diagrammes de classes

Exemple de métamodèle MOF1.4

Après avoir vu qu'un métamodèle était une sorte de diagramme de classes et en avoir présenté deux exemples, il est temps d'expliquer précisément ce qu'est un métamodèle.

Étant donné qu'il existe plusieurs façons de faire des métamodèles (MOF1.3, MOF1.4, MOF2.0, EMF, etc.) et que nous ne pouvons toutes les présenter, nous avons retenu la façon MOF version 1.4. Tous les métamodèles publics de l'OMG sont réalisés avec cette version, assez simple d'utilisation, contrairement à la 1.3, qui nécessite une connaissance de CORBA, ou à la 2.0, très complexe. Nous présentons la version 2.0 du MOF en fin de chapitre car c'est elle qui permettra d'élaborer les futurs métamodèles.

Pour MOF1.4, un métamodèle définit la structure d'un ensemble de modèles. Cette structuration est semblable à la structuration orientée objet. Les métamodèles MOF1.4 sont donc définis sous forme de classes. Les modèles conformes aux métamodèles sont considérés comme des instances de ces classes.

Afin de discerner les classes constituantes d'un métamodèle des autres classes, telles que les classes Java, MOF1.4 propose d'utiliser le terme *métaclasse.* Un métamodèle est ainsi constitué d'un ensemble de métaclasses. De même, afin de discerner les objets instances des métaclasses des autres objets, MOF1.4 propose d'utiliser le terme *méta-objet.* Ainsi, un modèle est constitué d'un ensemble de méta-objets instances de métaclasses.

Une métaclasse a un nom et contient des attributs et des opérations, aussi appelés *méta-attributs* et *méta-opérations.* Un méta-attribut représente une propriété d'un élément du modèle. Une méta-opération représente un traitement applicable à un élément du modèle.

Pour typer les attributs et les paramètres des opérations, MOF1.4 propose le concept de *type de donnée* (`dataType`). Un type de donnée permet de spécifier un type qui n'est pas un type d'objet. Les types tels que les booléens, les chaînes de caractères, les tableaux et les structures sont des types de données.

Pour ce qui concerne l'expression des relations entre métaclasses, MOF1.4 propose le concept de *méta-association.* Une méta-association est une association binaire entre deux métaclasses. Une méta-association a un nom, et chacune de ses extrémités peut porter un nom de rôle et une multiplicité.

Pour grouper entre eux les différents éléments d'un métamodèle, MOF1.4 propose le concept de *package*. Un package, aussi appelé métapackage, est un espace de nommage servant à identifier par des noms les différents éléments qui le constituent.

Pour résumer, on peut dire que les concepts de base de MOF1.4 sont les suivants (en appellation anglaise) :

- **Class.** Une métaclasse permet de définir la structure de méta-objets. Un ensemble de méta-objets constitue un modèle. Une métaclasse contient des méta-attributs et des méta-opérations.
- **DataType.** Un type de donnée permet de spécifier le type non-objet d'un méta-attribut ou d'un paramètre d'une méta-opération.
- **Association.** Une méta-association permet de spécifier une relation binaire entre deux métaclasses.
- **Package.** Un métapackage permet de regrouper sous un même espace de nommage différents éléments d'un métamodèle.

Métaclasse *(class)*

Comme expliqué précédemment, une métaclasse sert à décrire la structure d'un ensemble de méta-objets.

Une métaclasse a les propriétés suivantes :

- Possède un nom.
- Peut hériter d'une ou de plusieurs autres métaclasses. L'héritage signifie que la sous-classe possède tous les méta-attributs et toutes les méta-opérations de la superclasse. Si une métaclasse hérite de plusieurs métaclasses, il est impératif que ces métaclasses n'aient pas de méta-attributs ni de méta-opérations de mêmes noms.
- Peut être abstraite. Si tel est le cas, aucun méta-objet ne peut être une instance directe de cette métaclasse.

- Peut être considérée comme feuille (`leaf`) ou racine (`root`) d'un arbre d'héritage. Si la métaclasse est feuille, cela signifie qu'aucune autre métaclasse ne peut en hériter. Si la métaclasse est racine, elle ne peut hériter d'une autre métaclasse.

☛ Méta-attribut *(attribute)*

Une métaclasse peut posséder zéro ou plusieurs méta-attributs.

Un méta-attribut a les propriétés suivantes :

- Possède un nom.
- Possède une portée (`scope`). Peut être de niveau méta-objet ou métaclasse. Si la portée est de niveau méta-objet (`instance_level`), cela signifie que le méta-objet porte la valeur du méta-attribut. Si la portée est de niveau métaclasse (`classifier_level`), cela signifie que la métaclasse porte la valeur du méta-attribut pour tous les méta-objets instances.
- Possède un type. Peut être une métaclasse ou un type de donnée *(voir la section suivante sur les types de données).*
- Peut être ou non modifiable (`isChangeable`). S'il est non modifiable, cela signifie en quelque sorte que le méta-attribut a une valeur constante.
- Peut être considéré comme dérivé (`isDerived`). Si le méta-attribut est dérivé, cela signifie que sa valeur peut être calculée, par exemple, grâce aux valeurs d'autres méta-attributs de la métaclasse.
- Possède une multiplicité (`multiplicity`). La multiplicité est spécifiée par trois informations. La première contient les bornes maximale (`upper`) et minimale (`lower`) de la multiplicité. Par exemple, un méta-attribut ayant 0 comme borne minimale et 1 comme borne maximale est un méta-attribut optionnel, tandis qu'un attribut ayant 1 comme borne minimale et l'infini (représenté par le caractère `*`) comme borne maximale est un méta-attribut obligatoire pouvant avoir une infinité de valeurs. Les deux autres informations ne concernent que les méta-attributs ayant une borne maximale supérieure à 1. L'information « ordonné » (`is_ordered`) permet de spécifier que l'ensemble des valeurs du méta-attribut est ordonné et l'information « unique » (`is_unique`) que l'ensemble des valeurs du méta-attribut ne doit pas contenir de doublon.

☛ Méta-opération *(operation)*

Une métaclasse peut contenir zéro ou plusieurs méta-opérations.

Une méta-opération a les propriétés suivantes :

- Possède un nom.
- Possède une portée. Peut être de niveau méta-objet ou métaclasse. Si la portée est de niveau méta-objet, il est possible de demander à un méta-objet de réaliser la méta-opération. Si la portée est de niveau métaclasse, il est possible de demander directement à la métaclasse de réaliser la méta-opération.

- Peut contenir zéro ou plusieurs métaparamètres d'entrée. Un métaparamètre, comme un méta-attribut, a un nom, un type et une multiplicité. De plus, il a une direction (`direction`). Cette direction est soit monodirectionnelle, soit bidirectionnelle. Si elle est monodirectionnelle, elle peut être soit de l'appelant vers l'appelé (`in`), l'appelant donnant la valeur au métaparamètre, soit de l'appelé vers l'appelant (`out`), l'appelé donnant la valeur au métaparamètre. Si elle est bidirectionnelle (`in_out`), c'est indifféremment l'appelant ou l'appelé qui donne la valeur au métaparamètre.
- Peut optionnellement définir un type de retour. Ce dernier peut être soit une métaclasse, soit un type de donnée. Si une méta-opération n'a pas de type de retour, on considère qu'elle ne retourne rien et non pas qu'elle retourne un ensemble vide (`void`).
- Peut jeter une ou plusieurs exceptions. Une exception a un nom et peut contenir zéro ou plusieurs attributs permettant de la renseigner.

Type de donnée *(dataType)*

Les types de données permettent de définir des types non-objet. MOF1.4 permet de définir deux sortes de types de données : les types primitifs et les types construits.

Concernant les types primitifs, MOF1.4 définit les types suivants :

- `boolean` : la valeur est soit vrai (`true`), soit faux (`false`).
- `integer` : la valeur est un entier compris en entre -2^{31} et $+2^{31}-1$.
- `long` : la valeur est un entier compris entre -2^{61} et $+2^{61}-1$.
- `float` : la valeur est définie par la norme ANSI/IEEE 754-1985.
- `double` : la valeur est définie par la norme ANSI/IEEE 754-1985.
- `string` : la valeur est une chaîne de caractères encodée en 16 bits.

Concernant les types construits, MOF1.4 propose les énumérations, les alias et les structures.

Lorsqu'on élabore un métamodèle, on peut soit utiliser les types primitifs existants, soit construire ses propres types grâce aux constructions proposées par MOF1.4.

Méta-association *(association)*

Une méta-association permet de définir une relation entre deux métaclasses. En fait, une méta-association permet de définir la structure des liens entre les méta-objets instances des métaclasses reliées par la méta-association.

Une méta-association a les propriétés suivantes :

- Possède un nom.
- Contient obligatoirement deux extrémités (`associationEnd`). Ce sont les extrémités d'une méta-association qui sont reliées aux métaclasses.

☛ Extrémité d'association *(associationEnd)*

Une extrémité de méta-association a les propriétés suivantes :

- Possède un type. Le type de l'extrémité identifie la métaclasse reliée par la méta-association.
- Possède un nom. Correspond au nom du rôle que joue la métaclasse reliée.
- Possède une multiplicité (même multiplicité que les méta-attributs). Cette multiplicité permet de spécifier le nombre d'instances de la métaclasse identifiée par le type de l'extrémité pouvant être liées à exactement une instance de la métaclasse de l'autre extrémité de l'association. Par exemple, dans le métamodèle des diagrammes de cas d'utilisation *(voir figure 2.2)*, la méta-association `cas` relie les métaclasses `acteur` et `cas d'utilisation`. Une extrémité de cette méta-association a pour type `cas d'utilisation`. La multiplicité de cette extrémité est `*`. Cela signifie que plusieurs instances de `cas d'utilisation` peuvent être liées à une instance d'`acteur` (rappelons que `*` signifie zéro ou plusieurs). Pour savoir combien d'instances d'`acteur` peuvent être liées à une instance de `cas d'utilisation`, il faut regarder l'autre extrémité de l'association (de nouveau `*`).

Une extrémité de méta-association dispose d'une spécification d'agrégation. Celle-ci peut être soit composite (`composite`), soit non existante (`non-aggregate`). Une spécification d'agrégation composite entraîne que les méta-objets (instances de la métaclasse identifiée par le type de l'extrémité) contiennent les méta-objets instances de la métaclasse identifiée par le type de l'autre extrémité (extrémité opposée).

Concrètement, cela signifie que si le méta-objet est détruit, tous les méta-objets qu'il contient sont aussi détruits. Par exemple, dans le métamodèle des diagrammes de cas d'utilisation *(voir figure 2.2)*, l'extrémité de la méta-association qui relie la métaclasse `système` à la métaclasse `cas d'utilisation` et qui a pour type la métaclasse `système` a une spécification d'agrégation composite (illustrée par le losange noir). Dans cet exemple, cela signifie que les méta-objets instances de la métaclasse `système` contiennent les méta-objets de la méta-classe `cas d'utilisation`. Si un méta-objet instance de la métaclasse `système` est détruit, tous les méta-objets instances de la métaclasse `cas d'utilisation` qu'il contient doivent être détruits.

Les contraintes suivantes doivent être respectées lorsqu'il existe des extrémités de méta-association ayant des spécifications d'agrégation composite :

- Il ne faut pas que les deux extrémités d'une méta-association aient une spécification d'agréation composite. Cela empêche que deux méta-objets soient composés l'un de l'autre.
- Il ne faut pas qu'une métaclasse soit reliée par deux méta-associations dont les extrémités opposées (celles dont les types sont les autres métaclasses) aient des politiques d'agrégation composite. Cela empêche qu'un méta-objet soit inclus dans deux méta-objets instances de deux métaclasses différentes.

- Il faut que la multiplicité de l'extrémité ayant une spécification d'agrégation composite ait 1 comme maximum. Cela empêche qu'un méta-objet soit inclus dans deux méta-objets instances de la même métaclasse.

Une extrémité de méta-association peut être ou non modifiable (`isChangeable`). Si l'extrémité est changeable, cela signifie qu'il est possible de modifier n'importe quand les liens des méta-objets correspondant aux extrémités des méta-associations.

Une extrémité de méta-association peut être ou non navigable (`isNavigable`). Si l'extrémité est navigable, cela signifie qu'il est possible d'atteindre les valeurs des méta-attributs du méta-objet lié et de lui appeler ses méta-opérations. Bien sûr, l'accès n'est possible que pour les méta-objets liés par les extrémités opposées de la méta-association. Dans l'exemple du métamodèle des diagrammes de cas d'utilisation, la méta-association `cas` a son extrémité dont le type est `cas d'utilisation`, qui est navigable (symbolisé par une flèche). Cela signifie que les méta-objets instances de la métaclasse `acteur` peuvent atteindre les valeurs des méta-attributs et appeler les méta-opérations des méta-objets instances de la métaclasse `cas d'utilisation` avec lesquels ils sont liés.

☛ Référence *(reference)*

Afin de faciliter les navigations entre méta-objets *via* les méta-associations dont les extrémités sont navigables, MOF1.4 propose le concept de *référence* (`reference`). Une référence est une sorte de méta-attribut permettant essentiellement de naviguer *via* les méta-associations.

Comme un méta-attribut, une référence a un nom, un type, une multiplicité, etc. La différence avec les méta-attributs réside dans le fait que le type de la référence ne peut être qu'une métaclasse reliée par une méta-association avec la métaclasse qui contient la référence. De plus, l'extrémité qui pointe vers l'autre métaclasse (celle qui ne contient pas la référence) doit être navigable. La multiplicité de la référence doit être la même que la multiplicité de cette extrémité. La référence est en fait une sorte d'alias pour atteindre les méta-objets liés.

Package

MOF1.4 propose le concept de *package*, qui permet de grouper différents éléments d'un même métamodèle. L'objectif est, d'une part, de regrouper les éléments d'un métamodèle portant sur un même domaine, par exemple, les éléments relatifs aux diagrammes de classes, et, d'autre part, de gérer plus facilement les méta-objets instances d'un ensemble de métaclasses. Il est ainsi possible de stocker dans un même espace mémoire des méta-objets dépendant les uns des autres.

Un package a les propriétés suivantes :

- Possède un nom.
- Contient zéro ou plusieurs métaclasses, méta-associations reliant ces classes et types de données nécessaires.
- Peut contenir zéro ou plusieurs autres packages.

- Lorsqu'un élément est dans un package (métaclasse, méta-association, type de donnée, package), il dispose d'un nom complet. Ce nom correspond au nom complet du package, suivi du caractère « . », suivi du nom de l'élément. Si, par exemple, la métaclasse `C` est contenue dans le package `B`, qui est lui-même contenu dans le package `A`, le nom complet de la métaclasse est `A.B.C`.
- Un package peut hériter d'un ou de plusieurs autres packages. Dans ce cas, le sous-package acquiert tous les éléments du superpackage (métaclasses, méta-associations, types de données, packages).
- Un package peut importer un autre package. Le package qui importe l'autre package (le package importé) peut utiliser tous les éléments du package importé.

Notation graphique

Étant donné qu'un métamodèle se représente par un diagramme de classes, MOF1.4 ne propose pas de notation graphique spécifique pour l'élaboration de métamodèle et conseille la notation UML.

Cette approche permet d'utiliser les nombreux outils d'élaboration de diagrammes de classes UML. Certains de ces outils proposent des extensions spécifiques à l'élaboration de métamodèles MOF1.4.

Il faut toutefois prendre garde d'induire une confusion entre les diagrammes de classes UML, qui sont des modèles, et les diagrammes de classes MOF1.4, qui sont des métamodèles. Il n'est pas possible d'identifier si un diagramme de classes représente un modèle UML ou un métamodèle MOF. Tout dépend du rôle qu'on lui donne, qui peut être de spécifier la structure d'une application objet ou de spécifier la structure d'un ensemble de modèles.

Les niveaux méta

Nous venons de voir ce qu'était un métamodèle et avons décrit la relation qui existait entre un modèle et son métamodèle. Nous avons vu qu'un métamodèle était une sorte de diagramme de classes dont les règles de construction étaient définies par MOF.

Nous allons maintenant présenter l'architecture à quatre niveaux telle que définie par MDA. Cette architecture pose les bases des relations qui existent entre entités à modéliser, modèles, métamodèles et métamétamodèles.

Entités à modéliser

Il est important de garder en mémoire que l'objectif premier de MDA est de faciliter la construction et l'évolution des applications informatiques. Les entités à modéliser dans le contexte MDA sont donc essentiellement des applications informatiques.

Étant donné que les qualités attendues de MDA sont la pérennité, la productivité et la prise en compte des plates-formes, les méthodologies de développement, les mécanismes de production mais aussi les différentes plates-formes d'exécution existantes sont aussi des entités à modéliser. Le problème est que ces entités (applications, méthodologies, etc.) sont abstraites et qu'il n'est pas facile de distinguer le modèle de l'entité à modéliser.

Par exemple, est-ce le code et uniquement le code qui constitue l'application informatique ou le code est-il le modèle de l'application ? Dans le premier cas, le code serait l'entité à modéliser. Dans le second, à quoi correspondrait l'application informatique ?

En réalité, il s'agit d'un faux problème. Dans le contexte de MDA, seuls les modèles comptent, et ils servent tous plus ou moins à faciliter la production d'applications informatiques. À l'heure actuelle, il est vrai que la production d'une application se réduit à la génération de son code ainsi parfois qu'à celle des fichiers de déploiement.

Nous considérons dans le contexte de cet ouvrage que les entités à modéliser sont les applications informatiques et que les modèles de ces applications contiennent toutes les informations nécessaires à la génération du code. Cela signifie que nous incluons aussi bien les modèles techniques décrivant la conception des applications que les modèles conceptuels décrivant la sémantique du problème adressé par les applications.

Les modèles

Le dictionnaire de la langue française en ligne TLFI (Trésor de la langue française informatisé) donne les définitions suivantes du mot « modèle » *(http://atilf.atilf.fr/tlf.htm)* :

> Arts et métiers : *représentation à petite échelle d'un objet destiné à être reproduit dans des dimensions normales.*
>
> Épistémologie : *système physique, mathématique ou logique représentant les structures essentielles d'une réalité et capable à son niveau d'en expliquer ou d'en reproduire dynamiquement le fonctionnement.*
>
> Cybernétique : *système artificiel dont certaines propriétés présentent des analogies avec des propriétés, observées ou inférées, d'un système étudié, et dont le comportement est appelé, soit à révéler des comportements de l'original susceptibles de faire l'objet de nouvelles investigations, soit à tester dans quelle mesure les propriétés attribuées à l'original peuvent rendre compte de son comportement manifeste.*

En synthétisant ces définitions et en les adaptant au contexte MDA, nous pouvons considérer que les modèles MDA sont des représentations, à différents niveaux d'abstraction, de l'information nécessaire à la production et à l'évolution d'applications informatiques.

Les modèles réalisés dans MDA concourent donc tous plus ou moins selon leur niveau d'abstraction à la production ou à l'évolution d'applications informatiques.

Comme tout modèle, les modèles MDA doivent être fortement connectés avec la réalité, en l'occurrence avec les applications informatiques.

Les métamodèles

Nous venons de voir que les modèles MDA étaient des représentations de l'information nécessaire à la production et à l'évolution des applications informatiques. L'objectif principal de MDA est de faire en sorte que ces modèles soient pérennes, productifs et qu'ils prennent en compte les plates-formes d'exécution.

Ces qualités ne peuvent être obtenues qu'en définissant très précisément la structure des modèles. Nous avons vu que ces définitions de structure des modèles se faisaient grâce à des métamodèles.

Les métamodèles sont au cœur de MDA. Ils permettent de pérenniser la structure des modèles car ils sont eux-mêmes représentés sous forme de modèles (diagrammes de classes). Ils permettent d'associer des traitements aux modèles grâce notamment aux méta-opérations des métaclasses. Les liens entre métamodèles permettent enfin de bien séparer les aspects métier des aspects techniques et donc de prendre en compte les plates-formes d'exécution.

MOF1.4

Nous avons vu que les métamodèles étaient des diagrammes de classes élaborés selon les règles du standard MOF1.4. Le grand avantage apporté par MOF est que, grâce à lui, tous les métamodèles sont structurés de la même manière.

Nous avons vu qu'un métamodèle était un ensemble de métaclasses avec des méta-associations, etc. Cette structuration commune de tous les métamodèles permet de proposer des mécanismes génériques fonctionnant sur tous les métamodèles.

Tout l'intérêt de MOF est de définir des mécanismes génériques sur les métamodèles grâce à une structuration commune. Parmi ces mécanismes génériques, nous verrons dans la suite de cette partie consacrée à la pérennité que le mécanisme XMI (XML Metadata Interchange) permet de construire des grammaires XML à partir de n'importe quel métamodèle afin de stocker les modèles instances au format XML.

MOF est d'un intérêt stratégique pour MDA en permettant de créer des mécanismes génériques, qui peuvent être des mécanismes de production ou de pérennité.

Métamodèle des métamodèles

En y regardant de plus près, nous pouvons remarquer que la relation qui existe entre le standard MOF et les métamodèles est exactement la même qui celle qui existe entre un métamodèle et ses modèles instances. MOF définissant la structure que doit avoir tout métamodèle, il est naturel de vouloir le représenter sous forme de diagramme de classes.

Pour représenter MOF sous forme de diagramme de classes, il faudrait représenter tous ses concepts sous forme de classes et représenter les relations entre ses concepts sous forme d'associations.

La figure 2.5 illustre une partie de ce que pourrait être un diagramme de classes représentant MOF1.4. Les concepts de métaclasse, de méta-attribut, de méta-opération, de type de

donnée et de package y sont représentés par des classes. Les relations entre ces concepts sont représentées par des associations.

Ce diagramme de classes n'est pas complet et n'est fourni qu'à titre d'illustration pour montrer qu'il est parfaitement possible de représenter MOF sous forme de diagramme de classes.

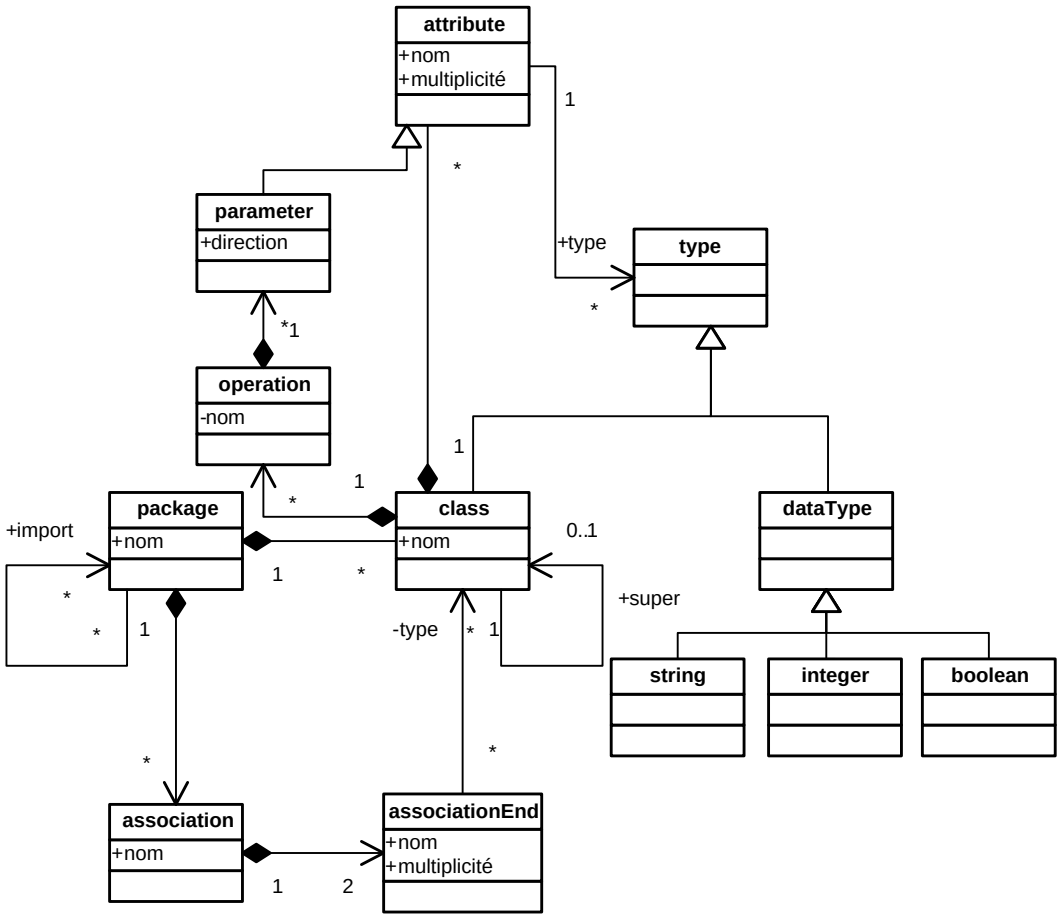


Figure 2.5

Représentation de MOF1.4 sous forme de diagramme de classes

Le diagramme de classes de MOF définit la structure que doit avoir tout métamodèle. Dit autrement, il s'agit du métamodèle des métamodèles, autrement dit du métamétamodèle.

En fait, le standard MOF1.4 est réellement défini comme un diagramme de classes. Ce diagramme de classes est appelé aussi bien métamétamodèle que *MOF modèle.*

Métamétamétamodèle

La découverte du niveau métamétamodèle soulève inévitablement la question de l'existence éventuelle d'un niveau *métamétamétamodèle,* voire celle de savoir si cette succession de niveaux a une fin et un sens.

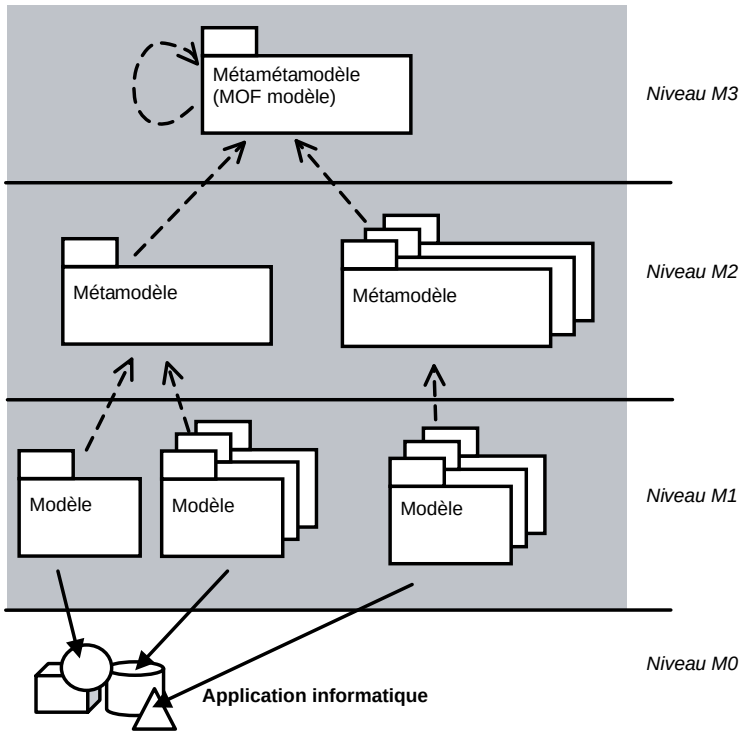
La réponse à ces questions est assez simple. Si nous devions faire le diagramme de classes décrivant la structure du métamétamodèle, nous aboutirions au diagramme de la figure 2.5. La raison à cela est que le métamétamodèle s'autodéfinit et est à lui-même son propre métamodèle. De ce fait, il n'existe que les niveaux modèle, métamodèle et métamétamodèle, aussi appelé MOF modèle.

L'architecture à quatre niveaux de MDA

À partir de ces définitions, nous pouvons nous représenter la fameuse architecture à quatre niveaux de MDA.

La figure 2.6 illustre cette architecture. On y voit le niveau M0, contenant les entités à modéliser, ici les applications informatiques, le niveau M1, contenant les différents modèles de l'application informatique, le niveau M2, contenant les différents métamodèles qui ont été utilisés, et le niveau M3, contenant le métamétamodèle qui a permis de définir uniformément les métamodèles.

Figure 2.6
Architecture à quatre niveaux de MDA



Les relations existantes entre les niveaux M1-M2 et M2-M3 sont équivalentes. M2 définit la structure de M1 tout comme M3 définit la structure de M2. Rappelons que le MOF modèle définit sa propre structure.

Les relations entre les niveaux M1 et M0 ne sont pas faciles à définir. En effet, les modèles du niveau M1 représentent l'information nécessaire à la construction et à l'évolution des applications informatiques et permettent de générer les applications. Nous pouvons considérer que les fichiers source d'une application sont aussi des modèles et qu'ils peuvent eux aussi appartenir au niveau M1.

MDA considère que, quels que soient les niveaux M1, M2 et M3, tous les éléments sont des modèles. Autrement dit, le métamétamodèle et les métamodèles sont aussi des modèles.

Métamodèles et typage des modèles

Nous venons de voir qu'un métamodèle définissait la structure d'un ensemble de modèles. Un ensemble de métamodèles peut donc être vu comme un système de typage des modèles. En prenant pour socle un ensemble de métamodèles, nous pouvons typer les modèles selon leur métamodèle.

MDA utilise fortement les métamodèles comme système de typage des modèles. L'objectif est de préciser quels modèles doivent être élaborés aux différentes phases de l'approche MDA (CIM, PIM, PSM et code). Il est important de comprendre que ce n'est pas l'architecture à quatre niveaux qui structure MDA mais l'ensemble des métamodèles définis par MDA.

Plusieurs métamodèles standards définissent déjà ce système de typage des modèles :

- Comme entraperçu au chapitre précédent, MDA propose d'utiliser le métamodèle UML pour élaborer les PIM. Ce même métamodèle avec ses profils est aussi utilisé pour élaborer les PSM.
- Le métamodèle MOF2.0 QVT est utilisé pour élaborer les transformations de modèles. Ce métamodèle permet de modéliser les passages automatiques entre les différentes phases de MDA.
- Les métamodèles OCL (Object Constraint Language) et AS (Action Semantics) sont aussi très présents dans MDA pour spécifier les comportements des applications, comme nous le verrons au cours des chapitres suivants. OCL et AS sont utilisés principalement pour l'élaboration des PIM.

Nous ne saurions trop insister sur le fait que ce sont ces métamodèles qui structurent l'approche MDA. Cet ensemble de métamodèles n'est évidemment pas complet. MDA n'est qu'une approche, et il est parfaitement envisageable pour une entreprise de définir ses propres métamodèles afin d'adapter cette approche à son contexte.

Les métamodèles qui constituent le système de typage MDA sont non pas indépendants mais liés. C'est grâce à ces liens que nous pouvons maintenir la cohérence entre les modèles élaborés dans les différentes étapes MDA. Cette cohérence est garante de la

qualité de MDA. C'est grâce à elle que les applications construites respectent les exigences des clients.

Liens entre métamodèles

Techniquement, pour pouvoir lier deux modèles instances de deux métamodèles différents et pour faire en sorte que le lien soit modélisé, il faut nécessairement qu'une méta-association existe au niveau des métamodèles. Cette méta-association peut être créée soit directement entre les métaclasses concernées (lien interne) soit par une autre métaclasse, appelée métaclasse de lien (lien externe).

La première solution (lien interne) a l'avantage d'être très simple mais présente l'inconvénient de modifier les métaclasses existantes, en y ajoutant une association et des références si l'on veut naviguer *via* le lien. La seconde solution (lien externe) a l'avantage de ne pas modifier les métaclasses existantes mais présente l'inconvénient de créer une nouvelle métaclasse et donc un nouveau métamodèle.

Prenons comme exemples les deux métamodèles du diagramme de classes et du diagramme de cas d'utilisation. Nous voulons établir un lien entre un cas d'utilisation et un ensemble de classes pour, par exemple, préciser qu'un cas d'utilisation est réalisé par un ensemble de classes. Nous pouvons soit créer une méta-association directement entre la métaclasse `classe` et la métaclasse `cas d'utilisation` et y ajouter les références nécessaires *(voir figure 2.7)*, soit créer une méta-association par le biais d'une nouvelle métaclasse, que nous appellerons `réalisation` *(voir figure 2.8)*.

Ces liens sont omniprésents dans MDA. Par exemple, des liens internes sont spécifiés entre les métamodèles UML, OCL et AS, et des liens externes sont utilisés dans le standard MOF2.0 QVT pour établir les transformations de modèles.

Figure 2.7
Lien interne entre métaclasses

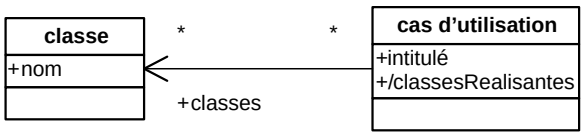
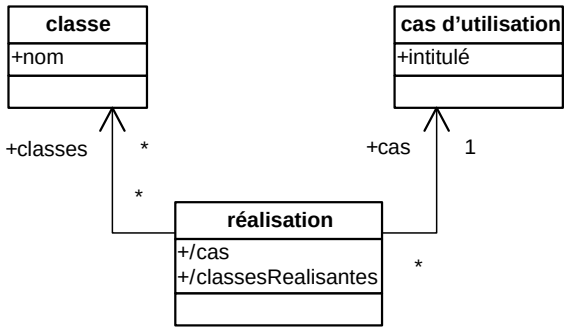


Figure 2.8
Lien externe entre métaclasses



Rappelons qu'il est toujours possible pour une entreprise d'adapter MDA à son contexte en ajoutant, par exemple, ses propres liens.

L'architecture MOF2.0 de l'OMG

Conceptuellement, l'architecture de MOF2.0 ne diffère que très peu de celle de MOF1.4. MOF2.0 est toujours l'unique métamétamodèle, et UML2.0 est le métamodèle dédié à la modélisation d'applications orientées objet.

Techniquement, en revanche, cette version est assez déroutante. L'un des objectifs de MOF2.0 est de capitaliser les points communs existants entre UML et MOF au niveau des diagrammes de classes et d'en expliciter les différences. Pour réaliser cet objectif, les moyens mis en œuvre sont d'une telle complexité qu'ils ne facilitent pas la compréhension de cette version.

UML2.0 Infrastructure

Un des objectifs des versions 2.0 de MOF et d'UML était d'aligner ces standards au niveau des diagrammes de classes. Nous savons qu'un métamodèle MOF ressemble fortement aux diagrammes de classes UML. Il est d'ailleurs impossible de regarder un diagramme de classes et de dire intrinsèquement s'il représente un métamodèle MOF ou un modèle UML. Cela dépend uniquement du sens que lui donne son concepteur. Il est dès lors naturel d'essayer de rapprocher ces deux notions.

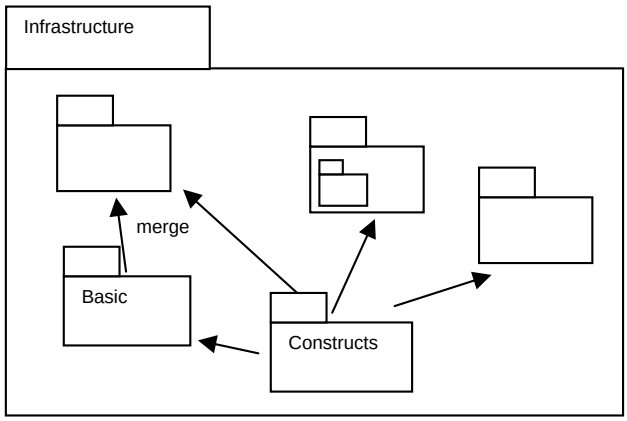
Pour ce faire, l'OMG a construit un métamodèle commun entre UML et MOF. Ce métamodèle contient toutes les métaclasses partagées par UML et MOF au niveau des diagrammes de classes. Il contient donc les métaclasses `package`, `class`, etc. Ce métamodèle commun porte le nom d'*UML2.0 Infrastructure.*

Le métamodèle UML2.0 Infrastructure a pour unique objet d'être réutilisé par les métamodèles MOF2.0 et UML2.0 Superstructure. De ce fait, UML2.0 Infrastructure est conçu d'une façon modulaire. Il est possible de ne réutiliser que certaines parties de l'infrastructure, par exemple, les types de base ou les packages.

Techniquement, pour rendre un métamodèle modulaire, il faut le découper en packages. Le métamodèle UML2.0 Infrastructure est constitué d'une trentaine de packages. Mentionnons parmi eux le package `Basic`, qui contient toutes les métaclasses nécessaires à l'élaboration de diagrammes de classes sans association, et le package `Constructs`, qui contient toutes les métaclasses nécessaires à l'élaboration de diagrammes de classes avec associations *(voir figure 2.9).*

Le package `Constructs` ressemble fortement à MOF1.4. La seule grande différence est que les associations et références de MOF1.4 sont ici considérées indifféremment comme des propriétés de classe (métaclasse `property`).

Figure 2.9
Représentation schématique du métamodèle UML2.0 Infrastructure



Pour faciliter la réutilisation de ses packages, UML2.0 Infrastructure définit le concept de *merge* entre deux packages. Un merge entre packages est une sorte de copier-coller des éléments du package mergé vers le package mergeur. Le merge permet d'intégrer plus facilement dans un nouveau métamodèle les packages proposés pas UML2.0 Infrastructure. Notons que le merge est aussi utilisé par UML2.0 Infrastructure lui-même entre certains de ses packages.

La sémantique du merge est en réalité beaucoup plus complexe. Elle établit une transformation entre les packages et les métaclasses des packages. Nous la présentons plus en détail au chapitre 3.

UML2.0 Superstructure

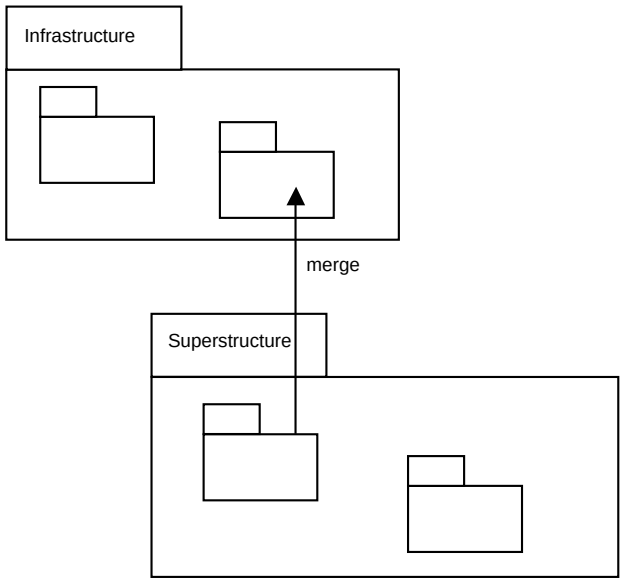
Dans sa version 2.0, le standard UML qui permet de modéliser les applications orientées objet s'appelle UML2.0 Superstructure. Ce nouveau nom permet de le distinguer du métamodèle UML2.0 Infrastructure.

Le métamodèle UML2.0 Superstructure réutilise certaines parties du métamodèle UML2.0 Infrastructure. Cela permet, comme nous l'avons vu, de rapprocher UML et MOF au niveau des diagrammes de classes.

La réutilisation du métamodèle UML2.0 Infrastructure par le métamodèle UML2.0 Superstructure se fait grâce au merge. On peut considérer que le métamodèle UML2.0 Superstructure fait un copier-coller du métamodèle UML2.0 Infrastructure mais sans intégrer la totalité de la trentaine de packages proposée par UML2.0 Infrastructure. Nous revenons plus en détail au chapitre suivant sur ce métamodèle.

En plus de cette intégration, le métamodèle UML2.0 Superstructure définit les autres concepts nécessaires à l'élaboration de tous les diagrammes UML (cas d'utilisation, séquence, déploiement, etc.). Le métamodèle UML2.0 Superstructure est donc beaucoup plus complexe que le métamodèle UML2.0 Infrastructure *(voir figure 2.10)*.

Figure 2.10
Représentation schématique du métamodèle UML2.0 Superstructure



MOF2.0

D'un point de vue conceptuel, MOF2.0 est toujours considéré comme étant l'unique métamétamodèle. Cependant, pour des raisons d'efficacité, il a été convenu qu'un méta-modèle instance de MOF2.0 pouvait ou non contenir des méta-associations. Voilà pourquoi, pour faire la différence entre ces deux types de métamodèles, MOF2.0 a été décomposé en deux parties : EMOF (Essential MOF), pour l'élaboration de métamodèles sans association, et CMOF (Complete MOF), pour celle de métamodèles avec association.

EMOF (Essential MOF)

EMOF a pour source principale le framework de métamodélisation EMF (Eclipse Modeling Framework) proposé par Eclipse *(http://www.eclipse.org/emf).* Ce framework, que nous présentons en détail au chapitre 6, permet de générer automatiquement des interfaces Java à partir de métamodèles Ecore. La particularité des métamodèles Ecore est qu'ils ne contiennent pas de méta-associations entre leurs méta-classes. Pour exprimer une relation entre deux métaclasses, il faut utiliser des méta-attributs et les typer par des métaclasses. EMF impose cette contrainte pour faciliter la génération des interfaces Java. Le concept d'association n'existant pas en Java, il faudrait en effet définir une transcription particulière.

MOF2.0 intègre lui aussi le métamodèle UML2.0 Infrastructure. EMOF intègre le package `Basic` de l'infrastructure et CMOF le package `Constructs`.

En fait, le package `EMOF` intègre le package `Basic` et le package `CMOF` intègre le package `Constructs` en utilisant le merge *(voir figure 2.11).* De ce fait, les relations d'intégration entre MOF2.0 et UML2.0 Infrastructure sont assez complexes.

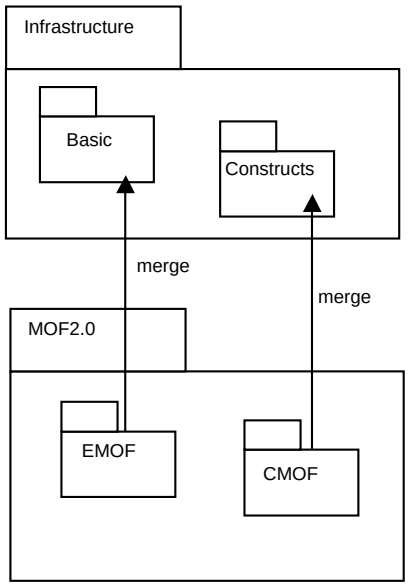


Figure 2.11
Représentation schématique du métamétamodèle MOF2.0

Architecture et niveaux

On pourrait penser que les versions 2.0 de MOF et UML chamboulent assez radicalement la belle architecture à quatre niveaux que nous avons présentée. Pourtant, au niveau conceptuel, cette architecture reste valide et facilite la compréhension des niveaux méta. MOF2.0 demeure le métamétamodèle (niveau M3), UML2.0 Superstructure reste un métamodèle (niveau M2), et les modèles UML (niveau M1) représentent toujours les informations nécessaires à la construction et à l'évolution des applications informatiques (niveau M0).

La difficulté vient principalement d'UML2.0 Infrastructure car il s'agit d'un métamodèle sans niveau fixe. Il peut appartenir à M3 ou M2, voire à M1 selon le bon vouloir de son utilisateur. Lorsqu'il est intégré à MOF2.0, il appartient au niveau M3. Lorsqu'il est intégré à UML2.0 Superstructure, il appartient au niveau M2.

Au fond, ce n'est pas très choquant en soi, car UML2.0 Infrastructure représente la structuration d'un diagramme de classes. Il nous suffit de savoir qu'il n'est pas possible de dire intrinsèquement à quel niveau appartient un diagramme de classes, tout dépendant du sens qu'on donne à ce dernier.

Synthèse

Nous avons vu dans ce chapitre que les métamodèles définissaient la structure d'un ensemble de modèles et non la sémantique des modèles.

Nous avons appris à faire des métamodèles selon le standard MOF1.4 et savons qu'un métamodèle est constitué d'un ou de plusieurs packages contenant des métaclasses reliées par des méta-associations.

Nous avons vu que l'architecture à quatre niveaux de MDA permettait de faire la différence entre les entités à modéliser, les modèles, les métamodèles et les métamétamodèles. Cette architecture nous a permis de souligner que les entités à modéliser dans le contexte MDA étaient essentiellement les applications informatiques.

Nous avons enfin vu que c'étaient les métamodèles et leurs interrelations qui structuraient MDA. Nous savons entre autres que le métamodèle UML permet l'élaboration de PIM et que ce même métamodèle associé aux profils que nous verrons au cours des chapitres suivants permet l'élaboration des PSM. Nous avons brièvement vu les autres métamodèles qui permettent de structurer MDA.

Pour finir, nous avons introduit les versions 2.0 d'UML et MOF et avons vu qu'elles n'ajoutaient pas de nouvelle notion conceptuelle tout en étant techniquement complexes.

Cette connaissance acquise sur les métamodèles va nous permettre d'entrer dans le détail des métamodèles qui font MDA.

3

UML2.0

Après une longue période de discussions techniques et stratégiques, l'OMG a voté, en juin 2003 à Paris, l'adoption du standard UML2.0. S'il est toujours possible d'élaborer les fameux diagrammes de cas d'utilisation, de classes, de séquences et d'états, UML2.0 n'est pas une simple mise à jour d'UML1.4. Il s'agit d'une réelle évolution, qui porte en elle tous les germes de MDA.

Ce chapitre traite de la vision utilisateur d'UML2.0, appelée UML2.0 Superstructure, et introduit les nouveautés radicales d'UML2.0 relatives à MDA. Le métamodèle UML2.0 est décortiqué afin de bien faire comprendre la place qu'il occupe dans MDA.

Plus précisément, le concept de composant UML, tel que défini dans le métamodèle, est présenté en détail. Ce concept de composant permet de mesurer pourquoi MDA préconise l'utilisation d'UML pour élaborer les fameux PIM. Au chapitre 12, nous utiliserons fortement ce concept de composant dans les modèles UML que nous élaborerons pour l'étude de cas PetStore.

Rappelons que cet ouvrage ne se veut pas un guide pour l'utilisateur UML et que les autres évolutions d'UML2.0, telles que les améliorations apportées aux diagrammes de séquences et d'états-transitions, ne sont pas présentées.

Nous nous arrêtons en revanche sur le concept de profil UML, qui permet d'adapter UML à des domaines autres que la modélisation d'applications orientées objet. Nous utiliserons fortement le concept de profil dans la suite de l'ouvrage.

Les objectifs d'UML2.0

Durant les années 70 et 80, on comptait tout au plus dix langages de spécification d'applications orientées objet. Pendant la première moitié des années 90, on en recensait une cinquantaine. Afin d'enrayer la multiplication des langages de spécification, G. Booch et J. Rumbaugh ont décidé en 1994 d'unifier leurs méthodes respectives OOADA (Object-Oriented Analysis and Design with Applications) et OMT (Object Management Technique) au sein de la société Rational Software Corporation.

La première version de ces travaux est sortie en octobre 1995 sous le nom d'UML0.8. Booch et Rumbaugh n'ont eu de cesse depuis d'unifier sous une même bannière toutes les méthodologies existantes. I. Jacobson a rejoint cette initiative vers la fin de l'année 1995 avec sa méthode OOSE (Object Oriented Software Engineering). Les autres méthodes de première génération ont ensuite été unifiées avec UML. C'est aussi à partir de ce moment que les aspects méthodologiques ont été dissociés d'UML et ont été repris dans un autre projet, nommé RUP (Rational Unified Process).

Après 1995, l'initiative de Rational Software Corporation a intéressé d'autres industriels, qui ont vite compris les avantages qu'ils pouvaient tirer de l'utilisation d'UML. C'est ainsi qu'une RFP (Request For Proposal) a été émise à l'OMG en 1996 pour la standardisation d'UML. Rational ainsi que de nombreuses autres entreprises, telles que HP, MCI Systemhouse, Oracle, Microsoft et IBM, ont soumis leurs réponses à cette RFP, et UML1.0 est sorti vers la fin de l'année 1996.

Des débuts de l'initiative UML jusqu'à la version 1.4 élaborée par l'OMG, l'objectif fondamental était de résoudre le problème de l'hétérogénéité des spécifications. L'approche envisagée a consisté à proposer un langage unifiant tous les langages de spécification et à imposer ce langage sur le marché pour qu'il soit très largement utilisé. Cela a permis de tirer parti de l'expérience de tous les langages de spécification mais aussi de convertir assez facilement leurs utilisateurs.

Nous pouvons considérer qu'UML est un succès, puisque c'est désormais le langage de spécification d'applications orientées objet le plus utilisé au monde.

La version 2.0 vise à faire entrer ce langage dans une nouvelle ère en faisant en sorte que les modèles UML soient au centre de MDA. Cela revient à rendre les modèles UML pérennes et productifs et à leur permettre de prendre en compte les plates-formes d'exécution.

La RFP UML2.0 Superstructure

Rappelons qu'UML2.0 est composé de deux standards, UML2.0 Superstructure, qui définit la vision utilisateur, et UML2.0 Infrastructure, qui spécifie l'architecture de méta-métamodélisation d'UML ainsi que son alignement avec MOF. Ces deux standards visent des objectifs sensiblement différents. Ce chapitre se focalise sur la partie UML2.0 Superstructure.

Afin de bien comprendre les objectifs de l'OMG pour la construction d'UML2.0 Superstructure, il est utile de se représenter les différents besoins exprimés dans sa RFP, même si ceux-ci peuvent paraître quelque peu indigestes.

Request for Proposal

Document faisant partie du processus de standardisation de l'OMG et listant tous les besoins devant être traités lors de l'élaboration d'un standard. Les besoins sont soit obligatoires soit optionnels.

Les besoins les plus importants d'UML2.0 Superstructure dans un contexte MDA sont les suivants (la traduction des besoins est écrite en italique afin de la dissocier des commentaires) :

- *6.5.1 a. Le standard devra établir clairement une séparation entre le métamodèle sémantique et la notation en établissant un mapping bidirectionnel entre eux*. Ce besoin montre que la notion de métamodèle sémantique est primordiale. Le métamodèle UML renferme donc toute la sémantique, au sens OMG du terme, de ce langage. La sémantique OMG consiste en la définition des règles de structuration des modèles. La notation graphique n'est ici considérée que comme un mapping vers le métamodèle sémantique. Ce point est très important car, dans les versions UML1.*x*, une partie de la sémantique était définie à l'aide de la notation graphique. Ce n'est plus le cas avec la version 2.0.
- *6.5.1 b. Le standard devra minimiser l'impact sur les utilisateurs d'UML1.x, MOF1.x et XMI1.x et devra spécifier la façon de passer d'UML1.4 à UML2.0*. Ce besoin montre que la notion de compatibilité ascendante commence à prendre sens dans les standards OMG. Les outils devraient donc, en théorie, être capables de transformer un modèle UML1.4 en un modèle UML2.0. L'objectif est de construire une version stable du métamodèle UML2.0 Superstructure, l'OMG souhaitant ancrer définitivement ce standard afin de pérenniser les modèles UML.
- *6.5.1d. Le standard devra définir une DTD pour UML2.0 selon le standard XMI*. Ce besoin montre l'importance de XMI. XMI est considéré comme étant le seul standard capable d'assurer les échanges de modèles. XMI est donc aussi porteur de pérennité.
- *6.5.2.1. Développement à base de composants : le standard devra supporter le paradigme composant, permettre de définir des patrons pour les composants, supporter les plates-formes à composants CCM et EJB et permettre la définition de profils pour les plates-formes à composants*. Ce besoin montre que le spectre d'UML doit s'élargir aux composants. Il montre aussi et surtout que les plates-formes techniques (CCM et EJB) doivent être supportées par UML. UML devient donc potentiellement un langage de modélisation permettant l'élaboration de PIM pouvant cibler les PSM CCM et EJB et doit pour cela être de plus en plus productif.

Cette liste atteste de la volonté de l'OMG de faire d'UML2.0 Superstructure un standard beaucoup plus orienté MDA que ne l'était UML1.4.

L'accent est porté sur le métamodèle, qui est beaucoup plus précis que le métamodèle UML1.4. La notation devient quelque chose de purement syntaxique, qui se greffe sur ce

métamodèle. Vient ensuite l'ouverture qui, grâce au standard XMI, permettra la constitution de passerelles vers les autres standards MDA. Enfin, UML2.0 Superstructure s'élargit vers le paradigme composant.

UML permet l'élaboration de modèles capables de générer des applications objet et composant. Il devient de ce fait le standard idéal pour l'élaboration de PIM. Remarquons qu'UML2.0 Superstructure est actuellement le seul standard MDA orienté utilisateur. Il permet l'élaboration de PIM (objet ou composant) et supporte les plates-formes techniques, ce qui facilite le passage aux PSM. Il est de plus ouvert, ce qui permet son intégration dans une méthodologie MDA.

Les quatre niveaux de compatibilité suivants sont définis pour les outils implémentant le standard UML2.0 Superstructure (malheureusement aucun processus n'a été défini pour vérifier la compatibilité des outils avec les niveaux) :

- Pas de compatibilité : l'outil n'est compatible ni avec le métamodèle, ni avec la notation.
- Compatibilité partielle : l'outil est compatible avec une partie du métamodèle et de la notation.
- Compatible : l'outil est entièrement compatible avec le métamodèle et la notation.
- Compatibilité d'échange : l'outil est entièrement compatible avec le métamodèle et la notation et est capable d'échanger des modèles au format XMI.

Les outils ayant le niveau « compatibilité d'échange » sont donc les seuls intégrables dans des environnements de développement MDA.

Le métamodèle UML2.0 Superstructure

D'énormes efforts ont été apportés à la réalisation du métamodèle UML2.0 Superstructure, qui est de ce fait beaucoup plus complet que ne l'était le métamodèle UML1.4. Sa sémantique est plus précise que celle d'UML1.4, même si elle demeure informelle. La notation graphique est intégralement alignée sur le métamodèle.

Grâce à l'expérience d'UML1.4, la découpe d'UML2.0 en différents packages et sous-packages est beaucoup plus fine. L'envers de la médaille est que ce métamodèle est extrêmement complexe, avec quelque cinquante packages et une bonne centaine de métaclasses.

L'objectif de ce chapitre se bornant à expliquer comment UML2.0 s'intègre dans MDA, nous ne détaillons pas le métamodèle UML dans son intégralité.

Architecture

Au plus haut niveau, le métamodèle UML2.0 Superstructure est découpé en trois parties :

- La partie `Structure` définit les concepts statiques (classe, attribut, opération, instance, composant, package) nécessaires aux diagrammes statiques tels que les diagrammes de classes, de composants et de déploiement.

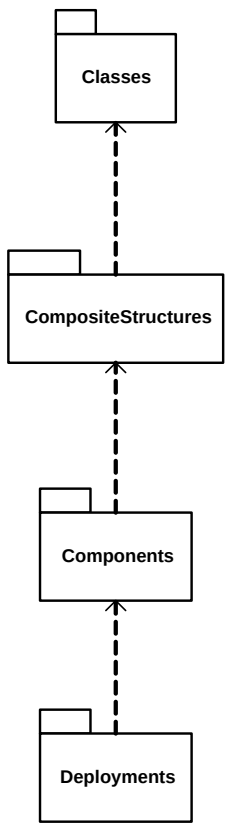
- La partie `Behaviour` définit les concepts dynamiques (interaction, action, état, etc.) nécessaires aux diagrammes dynamiques, tels que les diagrammes d'activités, d'états et de séquences.
- La partie `Supplement` définit des concepts additionnels, tels que les flots de données, les templates et les types primitifs. Cette partie définit aussi le concept de profil.

Chacune de ces parties est constituée de plusieurs packages et sous-packages :

La partie `Structure` comporte quatre packages *(voir figure 3.1) :*

- `Classes`. Contient plusieurs sous-packages, dont le sous-package `kernel`. Ces sous-packages définissent tous les concepts relatifs aux diagrammes de classes. Un concept réellement nouveau et intéressant dans le contexte de cet ouvrage est le concept de *package merge.*
- `CompositeStructure`. Contient lui aussi plusieurs sous-packages, qui définissent tous les concepts permettant de spécifier la structure interne d'une classe. La partie interne d'une classe est un ensemble d'instances qui collaborent.
- `Components`. Dépend fortement du package `CompositeStructure` et ne contient que deux sous-packages. Le package `Components` définit tous les concepts permettant de spécifier des composants.

Figure 3.1
Packages de la partie Structure



- `Deployments`. Contient plusieurs sous-packages, qui définissent tous les concepts permettant de spécifier le déploiement d'une application. Certains sous-packages sont propres au déploiement des applications à base de composants.

La partie `Behaviour` comporte six packages *(voir figure 3.2)*. Nous ne détaillons aucun de ces packages, car même s'ils présentent d'énormes évolutions, ils sortent du cadre de cet ouvrage. Ces packages sont les suivants :

- `CommonBehaviour`. Définit les concepts de base nécessaires aux aspects dynamiques.
- `Activities`. Définit les concepts permettant de spécifier des occurrences de suites d'actions dans une application. L'emphase porte ici sur la synchronisation entre actions et non pas sur les entités qui exécutent les actions.
- `Actions`. Dépend fortement du package `Activities`. Le package `Actions` précise le concept d'action.
- `UseCases`. Définit les concepts permettant de spécifier les cas d'utilisation d'une application.
- `Interactions`. Définit les concepts permettant de spécifier les interactions entre instances dans une application.
- `StateMachine`. Définit les concepts permettant de spécifier les états/transitions d'une application.

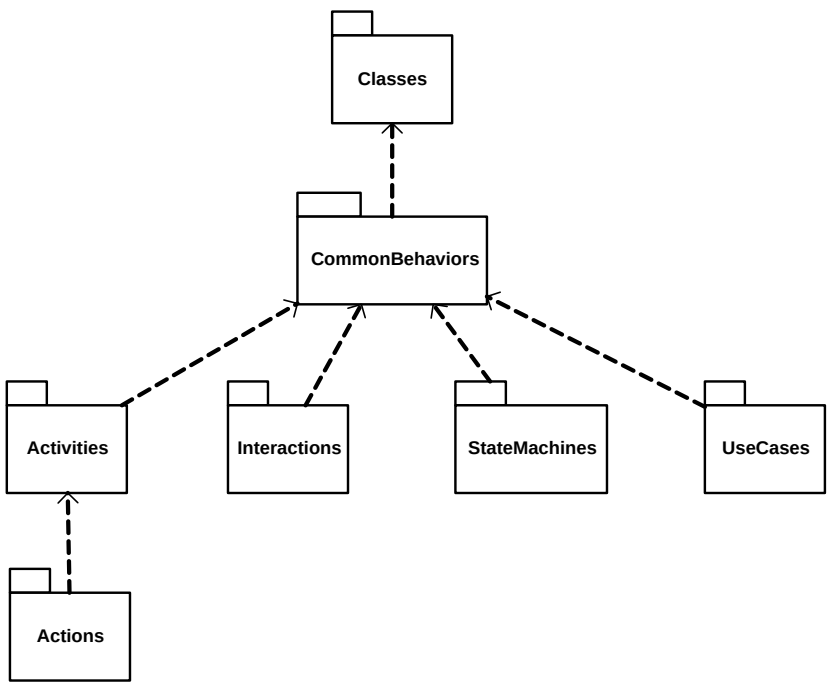
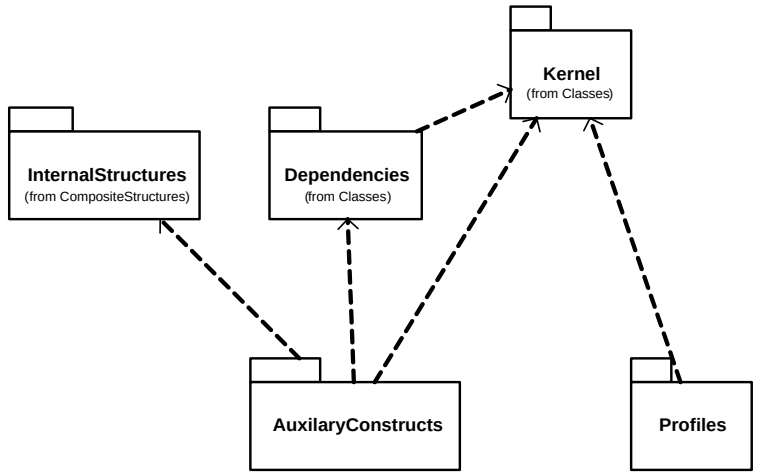


Figure 3.2
Packages de la partie Behaviour

La partie `Supplement` comporte deux packages *(voir figure 3.3) :*

- `AuxiliaryConstructs`. Sorte de fourre-tout constitué d'un sous-package définissant les types de base d'UML, d'un package définissant ce qu'est un modèle UML, d'un package définissant ce que sont les flots de données et d'un package définissant ce que sont les templates. Nous ne détaillons pas ce package car il ne définit pas de concepts entrant dans le cadre de cet ouvrage.
- `Profiles`. Définit les concepts permettant de spécifier des profils. Nous présentons ces concepts en fin de chapitre.

Figure 3.3
Package de la partie Supplement



La découpe en packages d'UML2.0 Superstructure lui confère une plus grande modularité. Les concepts relatifs à certains aspects d'UML étant groupés et rangés par package, la lecture du métamodèle est en principe plus aisée. En réalité, la multiplicité des packages ajoutée à celle des relations qu'ils entretiennent rendent la compréhension de ce métamodèle extrêmement ardue. Cette lecture est encore complexifiée par le fait que les relations entre packages sont des *package merge (voir la section suivante).*

Néanmoins, cette découpe montre aussi qu'UML a gagné en maturité. Il n'est plus envisageable de le considérer comme une simple notation graphique de langages objet. Il s'agit d'un langage à part entière, qui permet de spécifier concrètement n'importe quelle application.

La relation PackageMerge

La figure 3.3 illustre une nouvelle relation entre packages, la relation merge. Cette relation entièrement nouvelle est fortement utilisée par UML2.0 Superstructure. Elle s'établit uniquement entre deux packages, un package source et un package cible. Le package

cible de la relation est appelé package mergé et le package source package mergeant. Une relation merge entre packages implique des transformations sur le package mergeant.

La sémantique des transformations est la suivante :

- Si une classe est définie dans le package mergé, il faut définir une classe qui lui corresponde (même nom, mêmes attributs, etc.) dans le package mergeant, à moins qu'il n'existe déjà une classe correspondante, ayant même nom, mêmes attributs, etc.
- Si un package est défini dans le package mergé, il faut définir un package qui lui corresponde (même nom) dans le package mergeant, à moins qu'il n'y ait déjà un package correspondant (même nom). Après cela, une relation de merge doit être établie entre les packages correspondants.

La relation merge peut être vue comme une sorte de calque entre le package mergé et le package mergeant. En d'autres termes, les éléments du package mergé doivent apparaître dans le package mergeant. Cette relation est très utilisée pour la réutilisation de packages.

Les figures 3.4 et 3.5 illustrent un exemple de relation merge. Le package `R` merge les packages `P` et `Q`. Les classes `B` et `C` doivent donc être définies dans le package `R`. Le package `S` merge le package `Q`. La classe `C` doit donc être définie dans le package `S`.

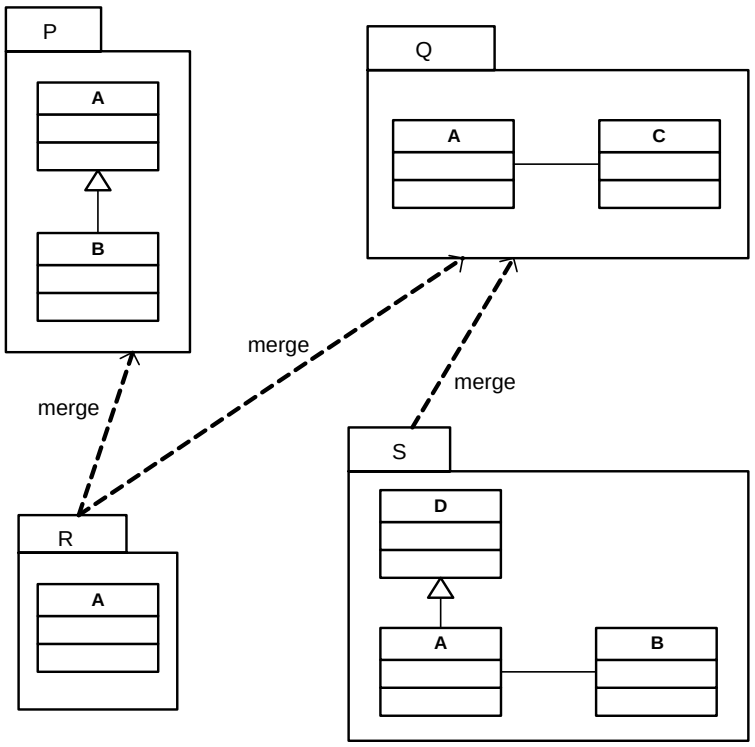


Figure 3.4
Exemples de relations merge avant transformation

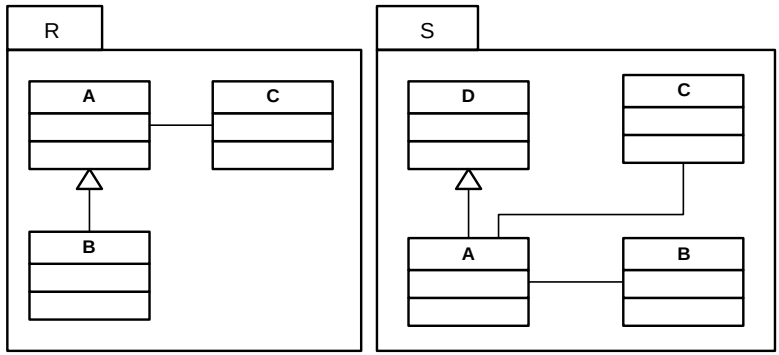


Figure 3.5
Exemple de relation merge après transformation

Comme expliqué précédemment, la découpe en packages et la relation merge rendent la lecture du métamodèle très difficile. C'est malheureusement une difficulté qu'il nous faut affronter si nous voulons saisir la place qu'occupe UML dans MDA.

La section suivante se plonge au sein du métamodèle et du concept de composant.

Le paradigme composant

La grande nouveauté d'UML2.0 Superstructure est le support du paradigme composant. Pour ce standard, un composant est une entité modulaire d'une application, qui peut être facilement remplacée par un autre composant sans que cela ait un impact sur son environnement. La notion d'interface de composant devient dès lors très importante.

Un composant a des interfaces offertes et des interfaces requises. C'est d'ailleurs ce qui le caractérise pour son environnement. La structuration interne d'un composant n'est pas visible de son environnement. Une application à base de composants est donc constituée de composants connectés entre eux. Remplacer un composant consiste à déconnecter le composant à remplacer et à connecter son remplaçant.

Le concept de composant tel que défini dans le standard UML2.0 Superstructure convient aussi bien aux composants logiques (composants business ou de processus) qu'aux composants physiques (EJB, CCM, Web Services).

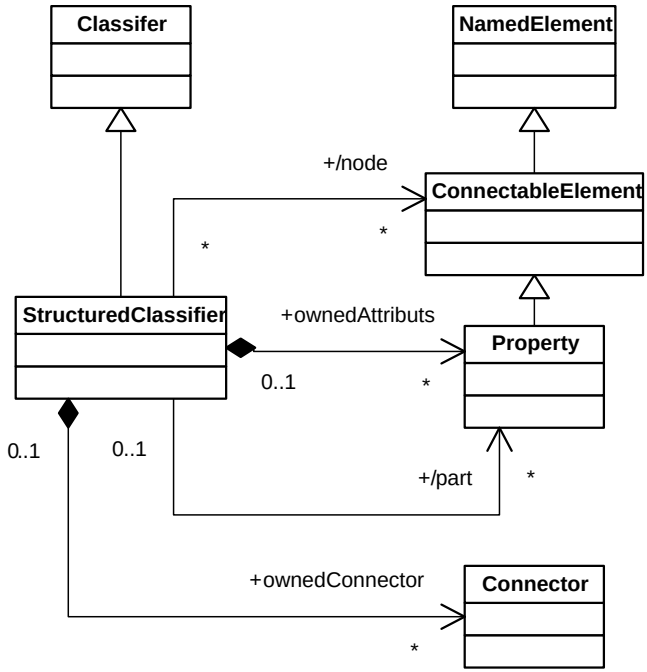
Avant d'entrer dans le détail de la définition de ce concept dans le métamodèle UML2.0 Superstructure, il est nécessaire de présenter le concept de *structured classifier.* Un structured classifier est un classifier qui voit son comportement spécifié par une collaboration d'instances contenues dans le classifier lui-même.

> **Classifier**
> Un classifier permet de classifier un ensemble d'objets. La classe est le classifier le plus connu.

La figure 3.6 illustre la métaclasse `StructuredClassifier`. Cette métaclasse est reliée à la métaclasse `Property` par deux méta-associations (une property peut être aussi bien un attribut qu'une référence).

La méta-association se terminant par `ownedAttribute` précise qu'un structured classifier peut contenir des properties. La méta-association se terminant par `part` précise que certaines des properties seront identifiées comme étant des *parts*. Une part peut être considérée comme une référence vers une instance contenue dans la collaboration qui spécifie le comportement du structured classifier. La métaclasse `StructuredClassifier` est aussi reliée à la métaclasse `Connector` par une méta-association. Celle-ci précise qu'un structured classifier peut contenir des connectors.

Figure 3.6
La métaclasse StructuredClassifier *dans le métamodèle UML2.0 Superstructure*



La figure 3.7 illustre la métaclasse `Connector`. Cette métaclasse est reliée à la métaclasse `ConnectorEnd` par une méta-association qui précise qu'un connector doit contenir au moins deux connector end. La métaclasse `ConnectorEnd` est reliée à la métaclasse `Connectable Element` par une méta-association qui précise qu'un connectable element joue un rôle au regard de son connector end.

En résumé, un structured classifier contient des parts qui sont connectés par des connectors.

En plus du concept de structured classifier, il est nécessaire de présenter les concepts d'*encapsulated classifier* et de *port*. La figure 3.8 illustre la métaclasse `Encapsulated Classifier`, qui hérite de la métaclasse `StructuredClassifier`. La métaclasse `Encapsulated Classifier` est de plus reliée à la métaclasse `Port` par une méta-association qui précise qu'un encapsulated classifier peut contenir des ports.

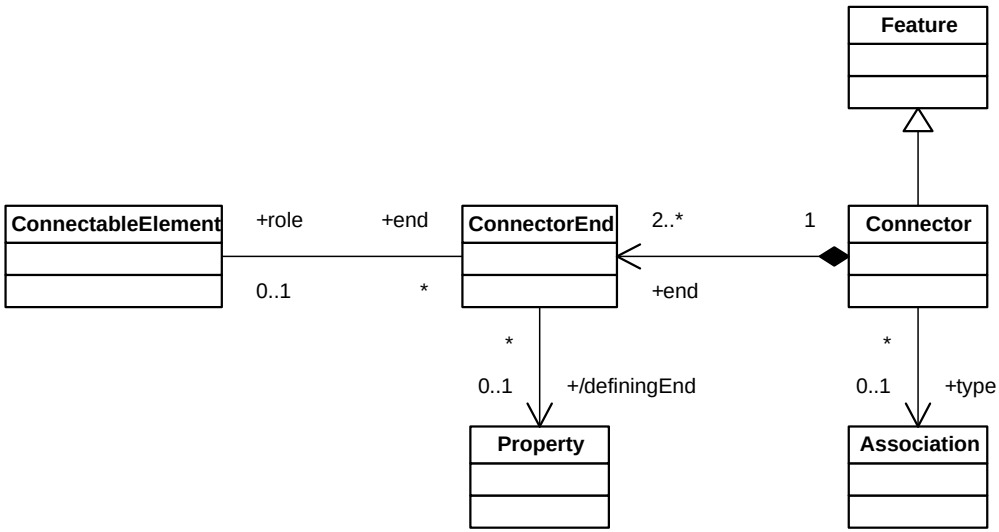


Figure 3.7
La métaclasse Connector *dans le métamodèle UML2.0 Superstructure*

La métaclasse `Port` héritant de la métaclasse `ConnectableElement`, un port peut être connecté par des connectors. La métaclasse `Port` est reliée à la métaclasse `Interface` par deux méta-associations, qui précisent qu'un port peut identifier des interfaces requises ou offertes. Dit autrement, les ports présentent la vision externe d'un encapsulated classifier. Ils sont assimilés à une ou plusieurs interfaces offertes ou requises. Ils peuvent aussi être connectés aux parts de l'encapsulated classifier.

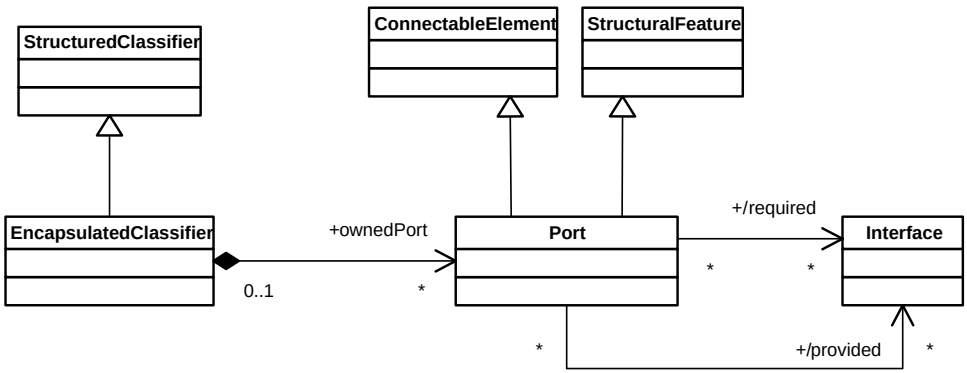


Figure 3.8
Les métaclasses EncapsulatedClassifier *et* Port *dans le métamodèle UML2.0 Superstructure*

UML2.0 Superstructure ne définit que trois concepts propres au support du paradigme composant *(voir figure 3.9 et 3.10) :*

- `Component`. La métaclasse `Component` hérite de la métaclasse `EncapsulatedClassifier`. Plus précisément, la métaclasse `Component` hérite de la métaclasse `Class`, qui hérite de la métaclasse `EncapsulatedClassifier`. Un composant a donc des interfaces offertes et requises (*via* les ports). Un composant peut donc être considéré comme étant un type. Remplacer un composant par un autre n'est possible que si les interfaces offertes correspondent (aucune autre information n'est donnée sur cette correspondance d'interfaces dans UML2.0 Superstructure).

  La métaclasse `Component` est reliée à la métaclasse `Realization` par une méta-association qui précise qu'un composant peut être réalisé par une ou plusieurs realization. La métaclasse `Realization` est reliée à la métaclasse `Classifier` par une méta-association qui précise qu'une realization se voit décrite par un classifier. Autrement dit, un composant est une sorte d'abstraction, dont le comportement est réalisé par un ensemble de classifiers. Ces classifiers correspondent à la vision interne du composant. Les interfaces offertes par le composant sont donc soit implémentées directement par le composant, soit implémentées par les classifiers qui réalisent le comportement du composant. Le lien entre la vision externe d'un composant et la vision interne est établi grâce aux ports du composant. En effet, les ports d'un composant sont liés, d'une part, aux interfaces offertes et requises du composant et, d'autre part, aux parts du composant qui représentent la structure interne du composant.

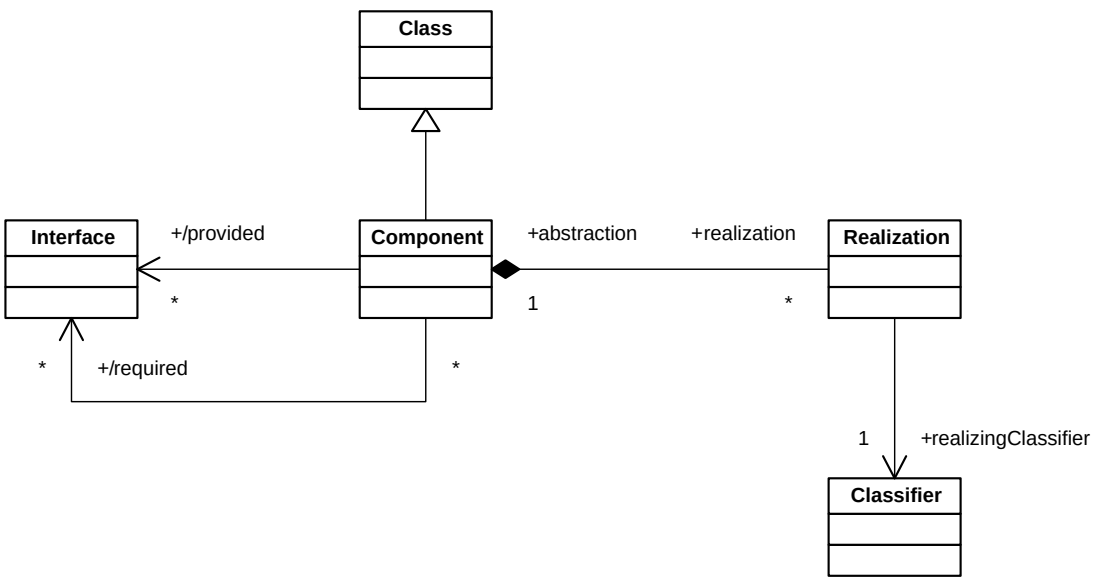
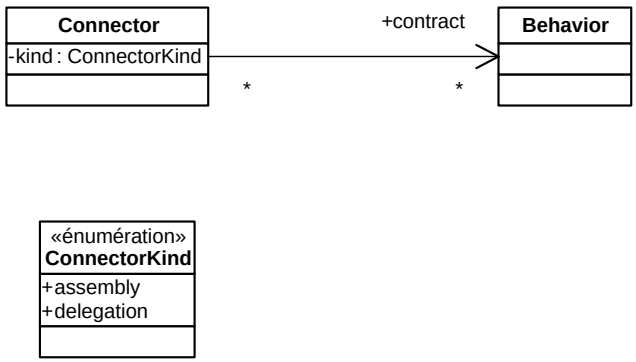


Figure 3.9
Les métaclasses Component *et* Realization *dans le métamodèle UML2.0 Superstructure*

- `Connector`. Le concept de connector spécifique des composants étend le concept de connector présenté précédemment (grâce à un merge) en lui ajoutant les notions de *delegation* et d'*assembly.* Un connector de délégation lie un port d'un composant avec une de ses parts. Un connector assembly lie deux composants *via* leur port ; il faut donc qu'un des composants requière une interface que l'autre offre. La figure 3.10 illustre la partie du métamodèle UML2.0 Superstructure qui définit la métaclasse `Connector`.
- `Realization`. Le concept de realization permet d'identifier les classifiers qui réalisent le comportement du composant.

Figure 3.10
La métaclasse Connector *dans le métamodèle UML2.0 Superstructure*



La figure 3.11 illustre un assemblage de composants qui représente une application de commerce électronique. Les éléments de ce diagramme sont des instances de composant (en UML, le terme composant fait plutôt référence à la spécification d'un composant). On considère ici que ce sont les parts d'un composant « top level » représentant l'application modélisée.

Dans cet exemple, la part `:ShoppingCart` est connectée à la part `:Order` car elle a besoin d'interfaces offertes par celle-ci. La part `:Order` est connectée aux parts `:Customer` et `:Organization` et aux parts `:Service` et `:Product`, et ainsi de suite. Pour compléter ce modèle, il faudrait spécifier les composants `ShoppingCart`, `Order`, `Service`, `Product`, etc. Les spécifications de ces composants contiendraient les spécifications des interfaces offertes et requises de ces composants ainsi que les spécifications des parts de ces composants.

La notation graphique utilisée ici est celle préconisée par l'OMG. Les composants sont représentés par des carrés, les connectors par des traits et les ports par des carrés positionnés sur les contours des composants. Cet exemple illustre parfaitement le fait que la notation graphique n'est qu'une syntaxe qui vient se greffer sur le métamodèle.

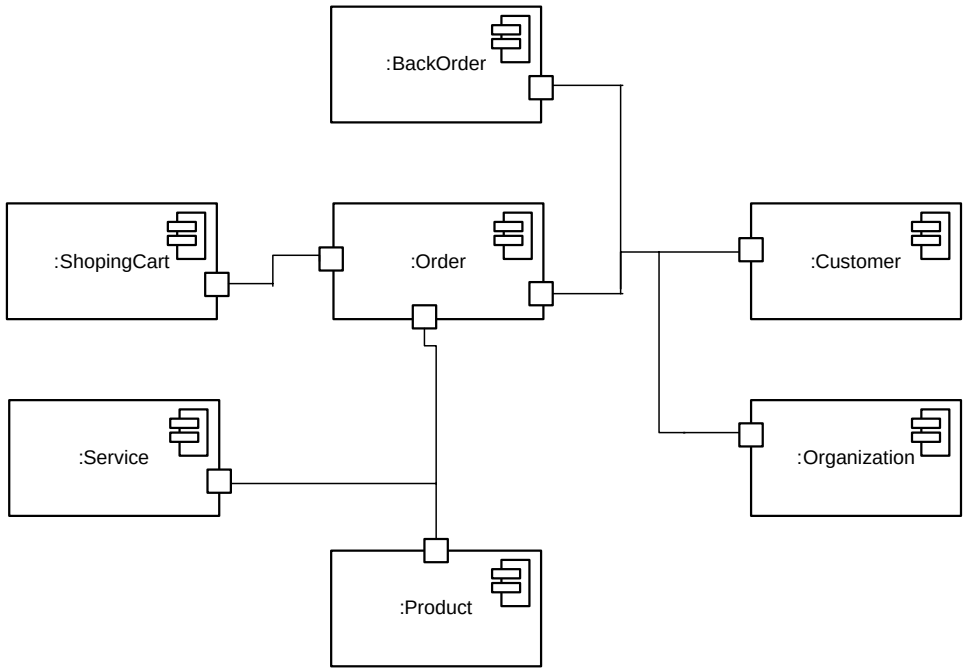


Figure 3.11
Exemple d'assemblage de composants

Déploiement

Le déploiement d'une application était déjà partiellement pris en compte dans la version UML1.4, mais il n'était pas réellement exploitable. Avec UML2.0 et le support des composants, le déploiement est pleinement modélisé. Il est même envisageable de piloter intégralement le déploiement d'application sur des plates-formes d'exécution.

Nous détaillons dans cette section le support du déploiement dans UML2.0 Superstructure afin de bien montrer à quel point UML est le standard central de MDA. UML2.0 permet en effet non seulement de modéliser très précisément une application, mais d'aller jusqu'à son déploiement tout en restant indépendant de la plate-forme sur laquelle elle s'exécute.

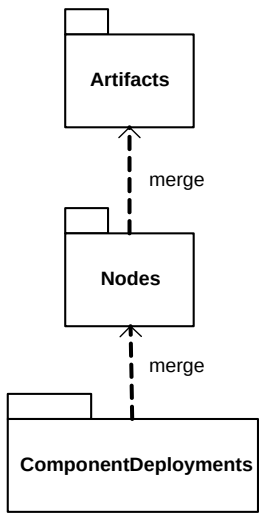
Nous avons vu à la figure 3.1 que le métamodèle UML2.0 Superstructure contenait un package nommé `Deployment`. Ce package comporte toutes les métaclasses nécessaires à l'élaboration de modèles de déploiement. Il est lui-même constitué des trois sous-packages suivants :

- `Artifacts`. Contient principalement la métaclasse `Artifact`, qui permet de représenter n'importe quelle information physique dont a besoin la plate-forme d'exécution sur laquelle l'application sera déployée. Un artefact peut représenter, par exemple, un fichier, un script ou un modèle.

- `Nodes`. Contient les métaclasses permettant de représenter la plate-forme d'exécution sur laquelle l'application sera déployée. Le package `Nodes` merge le package `Artifacts`.
- `ComponentDeployments`. Contient les métaclasses permettant de modéliser concrètement le déploiement de l'application modélisée à l'aide de composants. Le package `Component Deployments` merge le package `Node`.

La figure 3.12 représente les sous-packages du package `Deployment` et les différentes relations qui les lient.

Figure 3.12
Sous-packages du package Deployment



Le package *Artifacts*

Le package `Artifacts` définit les métaclasses `Artifact` et `Manifestation` *(voir figure 3.13)*. Ces métaclasses permettent de modéliser les ressources qui seront utilisées par la plate-forme d'exécution lors du déploiement de l'application. Ces ressources peuvent être des fichiers JAR pour la plate-forme J2EE ou des fichiers PHP pour la plate-forme PHP.

La métaclasse `Artifact` hérite de la métaclasse `Classifier`. Cela signifie que, tout comme les classifiers, les artefacts peuvent contenir des opérations et des propriétés et qu'il est possible de les relier grâce à des associations. Dit autrement, un artefact est la représentation dans le modèle d'une information représentée physiquement, sous forme de fichier, par exemple. Le nom de fichier (`fileName`) de l'artefact permet d'identifier la ressource physique.

La métaclasse `Manifestation` hérite de la métaclasse `Abstraction` et est reliée aux métaclasses `Artifact` et `PackageableElement`. Cela signifie qu'une manifestation est une relation d'abstraction entre un artefact et un packageable element. Un packageable element est un élément du modèle qui est contenu dans un package, par exemple, une classe, une association, etc.

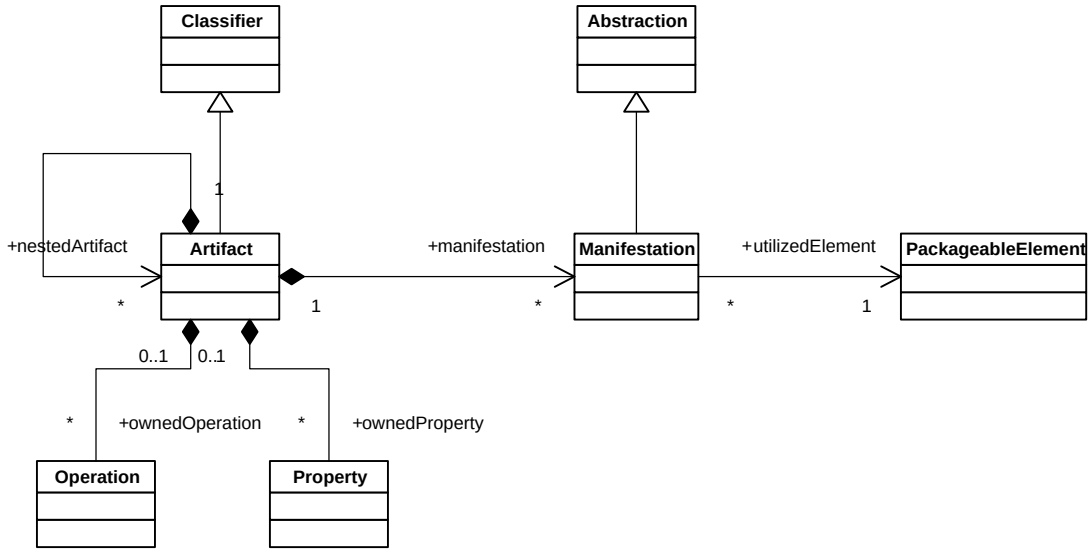


Figure 3.13
Le package Artifacts

Dit autrement, un artefact représente une ressource physique, laquelle dépend de plusieurs éléments du modèle. Nous pouvons donc dire qu'un artefact est, dans le modèle, une sorte d'abstraction de ces éléments de modèle. Cette abstraction, qui existe dans le modèle, se trouve concrétisée par une ressource physique. Par exemple, une archive contenant les fichiers des codes source de plusieurs classes UML est représentée dans le modèle UML par un artefact. Cet artefact peut être considéré comme une abstraction de l'ensemble des classes du modèle.

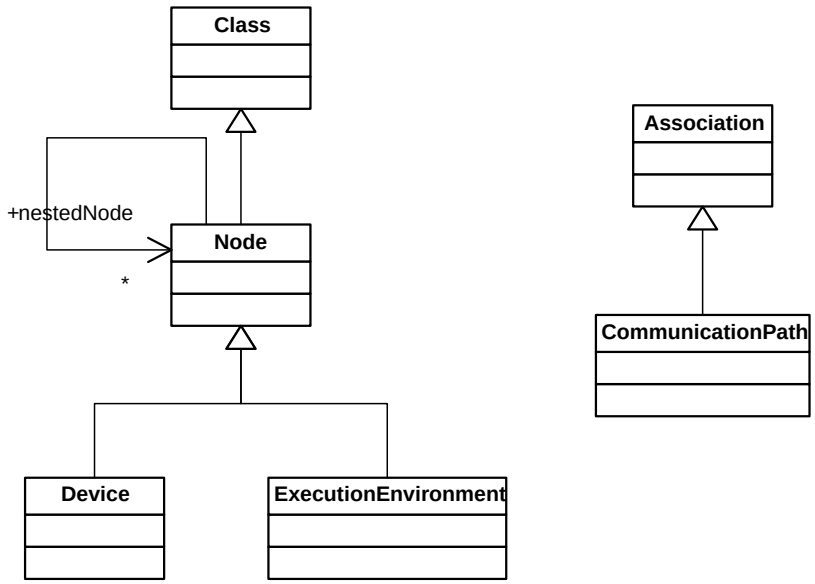
Le package *Nodes*

Le package `Nodes` contient les métaclasses `Node`, `Device`, `ExecutionEnvironment` et `CommunicationPath` *(voir figure 3.14).* Ces métaclasses permettent de modéliser les plates-formes d'exécution.

La métaclasse `Node` hérite de la métaclasse `Class` afin, d'une part, qu'il soit possible de lier les nœuds entre eux grâce à des *communication paths,* qui héritent de la métaclasse `Association`, et, d'autre part, qu'il soit possible de lier un nœud et un artefact.

La métaclasse `Node` est sous-classée par les métaclasses `Device` et `ExecutionEnvironment`. Dit autrement, un nœud représente une ressource informatique (environnement d'exécution ou simple périphérique) pouvant être interconnectée avec d'autres ressources grâce à des communication paths. Un nœud peut être constitué de plusieurs autres nœuds. Le nœud représente de plus l'entité sur laquelle les artefacts peuvent être déployés.

Figure 3.14
Le package Nodes



Le package *ComponentDeployment*

Le package `ComponentDeployment` contient les métaclasses `DeploymentTarget`, `Deployment` et `DeployedArtifact`. Il est partiellement illustré à la figure 3.15.

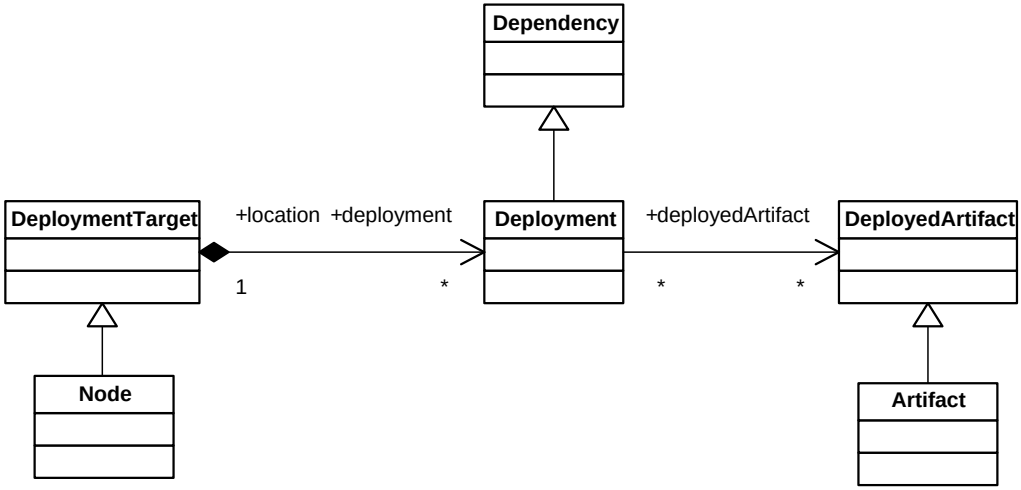
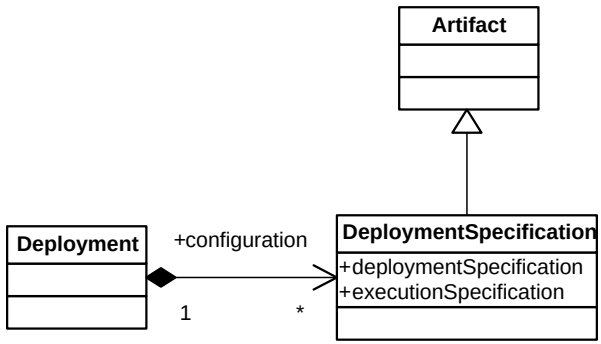


Figure 3.15
Première partie du package ComponentDeployment

La métaclasse `Deployment` hérite de la métaclasse `Dependency` et est reliée à la métaclasse `DeployedArtifact`, dont hérite la métaclasse `Artifact`. La métaclasse `DeploymentTarget`, dont hérite la métaclasse `Node`, est reliée à la métaclasse `Deployment`. Dit autrement, un nœud est une entité sur laquelle les entités peuvent être déployées. Un déploiement peut être vu comme une sorte de dépendance entre le nœud (`DeploymentTarget`) et l'artefact déployé (`DeployedArtifact`).

Le package `ComponentDeployment`, dont l'autre partie est illustrée à la figure 3.16, contient aussi la métaclasse `DeploymentSpecification`. Celle-ci hérite de la métaclasse `Artifact` et est reliée à la métaclasse `Deployment`. Elle possède deux attributs, nommés respectivement `deploymentSpecification` et `executionSpecification`, qui sont de type chaîne de caractères. Dit autrement, un déploiement peut être spécifié à l'aide d'une spécification de déploiement. Cette spécification de déploiement est elle aussi considérée comme un artefact, et il faut donc la déployer.

Figure 3.16
Seconde partie du package ComponentDeployment



Exemple de modèle de déploiement

La figure 3.17 illustre un exemple de modèle de déploiement. On y voit un serveur d'applications sur lequel sont déployés plusieurs artefacts, essentiellement des fichiers de déploiement spécifiques de la plate-forme J2EE. Il existe des relations de composition entre ces artefacts. Par exemple, l'artefact `ShoppingApp.ear` contient les artefacts `ShoppingCart.jar` et `Order.jar`.

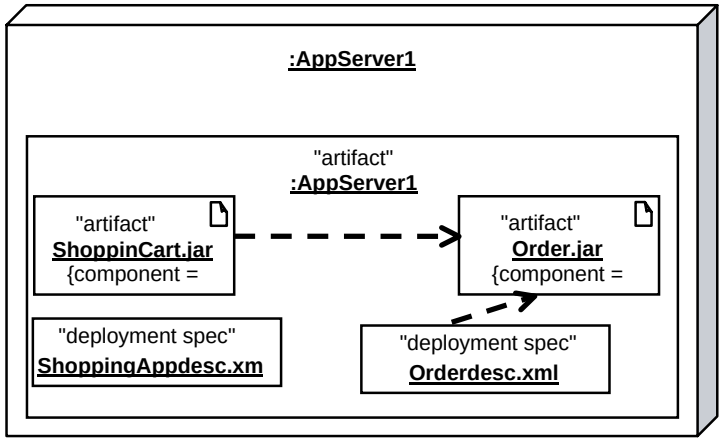
Les artefacts `ShoppingCart.jar` et `Order.jar` sont spécifiés à l'aide des spécifications de déploiement `ShoppingAppdes.xml` et `Orderdesc.xml`, elles-mêmes considérées comme des artefacts et donc déployées sur le serveur d'applications.

Le modèle illustré à la figure 3.17 ne fait pas apparaître les liens d'abstraction entre ces artefacts et les éléments du modèle UML représentant l'application (classes UML, composants, etc.). Ces liens sont toutefois d'une importance capitale car ce sont eux qui permettent de garder la traçabilité entre les différentes étapes du cycle de vie d'une application.

Cet exemple de modèle de déploiement est assez simple, car il ne contient qu'un seul nœud (`Node`). On pourrait imaginer des modèles de déploiement beaucoup plus complexes,

modélisant le déploiement d'applications réparties sur l'ensemble des serveurs du parc de production d'une société, par exemple.

Figure 3.17
Modèle de déploiement



Les profils UML

UML permet de modéliser des applications orientées objet, mais ses concepts sont suffisamment génériques pour être utilisés dans d'autres domaines. Par exemple, les diagrammes de classes, qui permettent en principe de ne modéliser que la partie statique d'une application orientée objet, peuvent être utilisés à d'autres fins, notamment pour modéliser des métamodèles. Ils peuvent même être employés (sans utiliser tous les concepts qu'ils proposent) pour modéliser des structures de bases de données.

UML peut donc être utilisé pour modéliser une multitude de domaines. Le problème est qu'il devient dès lors de plus en plus difficile en regardant un modèle de savoir s'il modélise une application objet, un métamodèle, une base de données ou autre chose.

Pour permettre l'adaptation d'UML à d'autres domaines et pour préciser la signification de cette adaptation, l'OMG a standardisé le concept de *profil UML*. Un profil est un ensemble de techniques et de mécanismes permettant d'adapter UML à un domaine particulier. Cette adaptation est dynamique, c'est-à-dire qu'elle ne modifie en rien le métamodèle UML et qu'elle peut se faire sur n'importe quel modèle UML.

Les profils UML mettent en jeu le concept central de *stéréotype*. Un stéréotype est une sorte d'étiquette nommée que l'on peut coller sur n'importe quel élément d'un modèle UML. Lorsqu'un stéréotype est collé sur un élément d'un modèle, le nom du stéréotype définit la nouvelle signification de l'élément. Par exemple, coller un stéréotype nommé `"Sécurisé"` sur une classe UML signifie que la classe en question n'est plus une simple classe UML mais qu'elle est une classe sécurisée, c'est-à-dire dont les accès sont restreints par mot de passe. Utiliser un profil consiste donc à coller sur un modèle UML un ensemble de stéréotypes.

D'un point de vue graphique, un stéréotype se représente sous la forme d'une chaîne de caractères contenant le nom du stéréotype encadré de guillemets. Si le stéréotype est collé sur une classe, cette chaîne de caractères doit apparaître au-dessus du nom de la classe *(voir figure 3.18).*

Figure 3.18
Utilisation d'un stéréotype sur une classe

Il est évident que le concept de profil ne peut fonctionner que si l'ensemble des stéréotypes utilisés est partagé par les différents intervenants manipulant les modèles. Dans l'exemple du stéréotype `"Sécurisé"`, il faut que tous les intervenants connaissent ce stéréotype ainsi que sa signification.

Pour faire face à cela, l'OMG a défini des profils standards. Ces profils définissent un ensemble de stéréotypes et expliquent leur signification en langage naturel. D'autres organismes de standardisation, tels que le JCP (Java Community Process), proposent eux aussi des profils standards.

L'intérêt principal des profils est qu'il est possible de leur associer des traitements. Spécifiques du domaine couvert par le profil, ces traitements permettent de rendre les modèles UML profilés beaucoup plus productifs que les modèles UML simples, car ceux-ci disposent d'informations supplémentaires grâce aux stéréotypes. Un traitement visant à mettre en place une politique de sécurité, par exemple, prendra une valeur tout autre s'il est attaché à un profil de sécurité.

Utilisation de profils existants

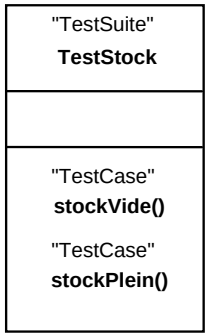
Actuellement, sept profils sont standardisés par l'OMG. Parmi ces profils, nous allons présenter brièvement le profil pour les tests, UML Testing Profile, afin de montrer comment sont utilisés les profils dans MDA. Ce profil permet d'utiliser UML pour modéliser des tests.

Parmi les stéréotypes de ce profil, nous nous bornerons à détailler les stéréotypes `"Test Suite"` et `"TestCase"`. Le stéréotype `"TestSuite"` s'applique à des classes UML. Lorsqu'il est utilisé, la classe UML représente une suite de tests à effectuer. Le stéréotype `"Test Case"` s'applique à des opérations UML. Lorsqu'il est utilisé, l'opération UML représente un cas de test à effectuer. Si un modèle UML contient une classe UML stéréotypée `"Test Suite"` contenant plusieurs opérations stéréotypées `"TestCase"`, ce modèle représente en fait une suite de tests avec plusieurs tests à effectuer.

La figure 3.19 illustre un modèle UML profilé pour les tests.

Le traitement associé à ce profil pour les tests concerne la génération automatique de classes Java pour le framework JUnit. JUnit est un framework Open Source qui permet d'exécuter un ensemble de tests. Ce profil permet donc de modéliser en UML les tests d'une application et de générer automatiquement les classes Java correspondantes.

Figure 3.19
Modèle UML profilé pour les tests



Définition de nouveaux profils

Le succès des profils UML a été tel que chacun a voulu définir des profils pour son propre domaine. Il est aujourd'hui quasiment impossible de dresser la liste de tous les profils définis.

UML2.0 Superstructure facilite la création de nouveaux profils en simplifiant la définition des concepts de profils et de stéréotypes dans le métamodèle.

La figure 3.20 illustre la partie du métamodèle UML2.0 qui contient les métaclasses relatives aux profils. Ces métaclasses correspondent de fait à la définition d'un profil. Un modèle instance de ces métaclasses correspond à la définition d'un nouveau profil et non à l'application d'un profil existant à un modèle UML.

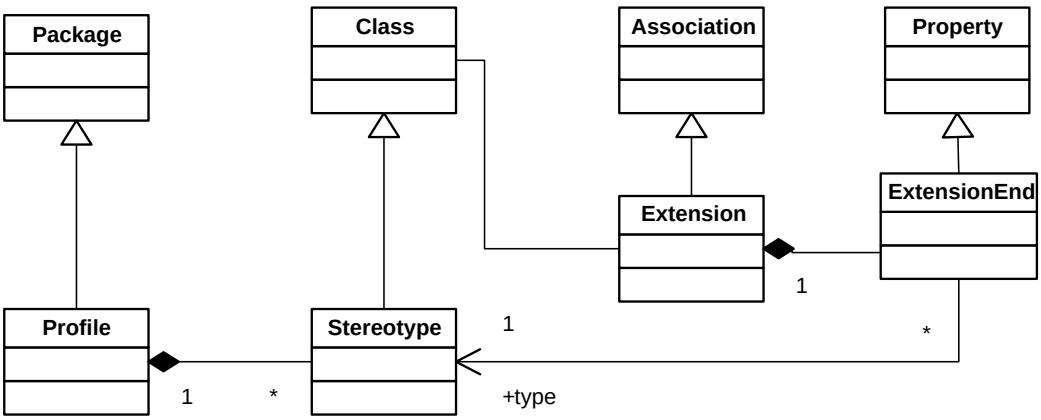


Figure 3.20
Les profils dans UML2.0

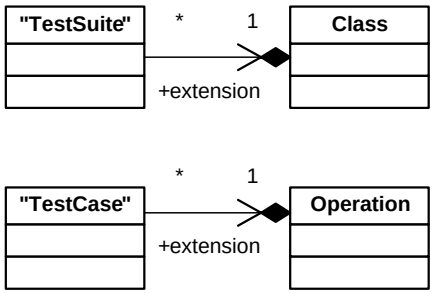
Dans ce métamodèle, la métaclasse `Profile` hérite de la métaclasse `Package`. Cela signifie que les définitions de nouveaux profils sont maintenant considérées comme étant des

modèles UML et plus précisément comme des packages. La métaclasse `Stereotype` hérite de son côté de la métaclasse `Class`. Cela signifie que les définitions de stéréotypes sont considérées comme étant des classes UML. Le nom de la classe correspond au nom du stéréotype.

L'aspect le plus délicat des profils est que la métaclasse `Stereotype` est reliée à la métaclasse `Class` *via* la métaclasse `Extension`, qui hérite de la métaclasse `Association`, et la métaclasse `ExtensionEnd`, qui hérite de la métaclasse `Property`. Cela signifie qu'un stéréotype apporte une extension de signification à une classe. Or, les classes étendues par des stéréotypes doivent être des métaclasses d'un métamodèle, à l'exemple de la métaclasse `Class` ou de la métaclasse `Operation`. Comme la métaclasse `Class` représente aussi bien le concept de classe que celui de métaclasse *(voir le chapitre 2)*, la métaclasse `Extension` est reliée à la métaclasse `Class`.

Si nous reprenons notre exemple de profil de test, celui-ci devrait être défini de la façon illustrée à la figure 3.21. Nous constatons que les stéréotypes `"TestSuite"` et `"TestCase"` sont liés non à des classes normales mais à des métaclasses (`Class` et `Operation`).

Figure 3.21
Profil pour les tests en UML2.0



Étant donné qu'un stéréotype est une classe UML (voir la relation d'héritage entre la métaclasse `Stereotype` et la métaclasse `Class`), il peut contenir des propriétés. On considère alors qu'un stéréotype contient des attributs. Les éléments stéréotypés peuvent attacher des valeurs à ces propriétés. Par exemple, il est possible d'associer une propriété nommée `obligatoire` au stéréotype `"TestCase"`.

Dans un modèle UML, une opération stéréotypée `"TestCase"` pourrait, par exemple, attacher la valeur `oui` à cette propriété. Cela signifierait que le test identifié par cette opération serait obligatoire.

Dans les versions UML1.3 et UML1.4, ces propriétés étaient appelées `Tagged-Value`. Même si la métaclasse `Tagged-Value` n'existe pas dans le métamodèle UML2.0, ce terme reste utilisé lors de la définition d'un profil. Nous l'utiliserons d'ailleurs lorsque nous aurons à présenter des profils.

Synthèse

Ce chapitre a présenté le standard UML2.0 Superstructure et souligné toute l'importance de ce métamodèle dans MDA. L'existence d'un métamodèle est d'ailleurs ce qui fait la plus grande différence entre les versions 1.*x* et 2.0 d'UML. Dans UML2.0 Superstructure le métamodèle est central, la notation graphique ne faisant que s'appuyer sur lui.

Nous avons décortiqué ce métamodèle par le biais du concept de composant et avons vu que la définition de ce concept était indépendante des plates-formes d'exécution. Nous avons notamment détaillé la façon dont le déploiement était modélisé en UML2.0 Superstructure de façon indépendante des plates-formes.

Après ces présentations, il est plus facile de comprendre pourquoi l'OMG préconise l'utilisation d'UML2.0 Superstructure pour élaborer les PIM puisque les modèles UML2.0 Superstructure permettent de modéliser les applications orientées objet indépendamment des plates-formes sur lesquelles elles s'exécutent. C'est en cela que les modèles UML2.0 Superstructure sont pérennes.

Nous nous sommes arrêté sur la notion de profil, qui donne toute sa dimension de flexibilité à UML2.0 Superstructure. Grâce aux profils, il est possible d'appliquer UML2.0 Superstructure à des domaines autres que la modélisation d'applications orientées objet. Nous avons montré comment étaient spécifiés les profils et à quoi ils servaient et avons souligné le fait que les profils avaient surtout vocation à rendre les modèles UML2.0 Superstructure productifs. Nous les utiliserons à la partie III de l'ouvrage, dédiée à la prise en compte des plates-formes d'exécution.

4

Les standards OCL et AS

Il n'est pas évident pour UML d'exprimer précisément ce que fait une opération. Dans le métamodèle UML, une opération n'est définie que par son nom, ses paramètres et les exceptions qu'elle émet. Le corps de l'opération ne peut donc être défini. Le langage OCL (Object Constraint Language) et le standard AS (Action Semantics) ont été précisément définis par l'OMG pour combler cette lacune et permettre la modélisation du corps des opérations UML.

OCL permet d'exprimer des contraintes sur tous les éléments des modèles UML. Il est notamment utilisé pour exprimer les pré- et postconditions sur les opérations. Il est, par exemple, possible de dire que l'opération `crédit` d'une classe `CompteBancaire` a pour post-condition que le solde du compte soit incrémenté du montant passé en paramètre de l'opération.

AS peut être vu comme une solution de rechange à OCL du fait des difficultés d'utilisation de ce dernier pour la majorité des utilisateurs UML, plutôt habitués à écrire des suites d'instructions. Grâce à AS, il est possible de modéliser des constructions d'objets, des affectations de variables, des suppressions, des boucles `for`, des tests `if`, etc.

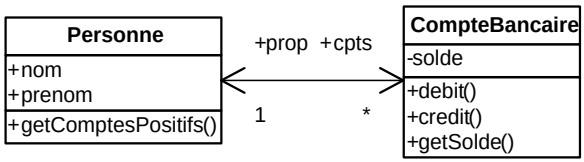
Ce chapitre montre comment modéliser les corps des opérations UML à l'aide des standards OCL et AS. Nous verrons que ces standards sont définis à l'aide de métamodèles et qu'ils se greffent sur le métamodèle UML pour s'intégrer dans MDA.

Ce chapitre ne se veut pas un guide utilisateur d'OCL et AS. Il se contente, une fois encore, de présenter ces standards dans le contexte MDA.

Le langage OCL

En utilisant uniquement UML, il n'est pas possible d'exprimer toutes les contraintes que l'on souhaiterait. Par exemple, il n'est pas possible de dire que l'attribut `solde` d'une classe `CompteBancaire` ne doit pas être inférieur à 1 000 euros *(voir figure 4.1)*.

Figure 4.1
Modèle UML exemple



Fort de ce constat, l'OMG a défini formellement le langage textuel de contraintes OCL (Object Constraint Language), qui permet de définir n'importe quelle contrainte sur des modèles UML.

Le concept d'expression est au cœur du langage OCL. Une expression est rattachée à un contexte, qui est un élément de modèle UML, et peut être évaluée afin de retourner une valeur. Par exemple, une expression OCL dont le contexte serait la classe `CompteBancaire` permettrait d'obtenir la valeur de l'attribut `solde`. Une autre expression OCL, dont le contexte serait la classe `Personne`, permettrait de vérifier qu'une personne dispose ou non d'un compte bancaire.

Les expressions OCL ne génèrent aucun effet de bord. L'évaluation d'une expression OCL n'entraîne donc aucun changement d'état dans le modèle auquel elle est rattachée. Une contrainte OCL est une expression dont l'évaluation doit retourner vrai ou faux. L'évaluation d'une contrainte OCL permet de la sorte de savoir si la contrainte est respectée ou non.

Si OCL permet d'exprimer des contraintes, il n'est pas pour autant un langage de programmation. Il ne permet pas de construire de nouveaux éléments ni même de modifier ou de supprimer des éléments existants.

Pour modéliser le corps d'une opération UML, les contraintes OCL peuvent être utilisées de deux façons :

- Elles peuvent contraindre l'état des objets avant (précondition) et après (postcondition) l'invocation des opérations. Par exemple, pour l'opération `crédit`, il est possible de dire que le solde du compte après invocation est égal au solde du compte avant invocation en ajoutant le montant transmis en paramètre de l'opération.
- Elles peuvent contraindre les données contenues dans les messages échangés lors de l'invocation d'une opération. Il est ainsi possible de dire que le message retourné par un compte bancaire lorsque l'opération `getSolde` est invoquée contient un entier dont la valeur est égale au solde du compte bancaire.

L'objectif du langage OCL est aussi, et surtout, d'être indépendant de toute plate-forme d'exécution. Une fois les contraintes OCL spécifiées sur un modèle UML, il est

envisageable de générer une application conforme en Java, PHP ou .Net. Pour que l'application soit conforme, il suffit que les contraintes OCL soient respectées. Vérifier cette conformité est toutefois une difficulté qui n'a pas encore rencontré de solution industrielle.

Ajoutons qu'OCL est un langage fortement typé et qu'il est possible de détecter si une expression OCL est évaluable simplement en vérifiant la conformité des types qu'elle utilise.

Les expressions OCL

Cette section introduit le concept d'expression OCL.

L'objectif est de décrire brièvement ce concept à l'aide de quelques exemples afin d'introduire le métamodèle qui représente ce concept.

Relation avec UML

Une expression OCL porte sur un élément de modèle UML. Nous montrons ici la relation qui lie OCL avec UML.

☛ Le contexte

Pour être évaluée, une expression OCL doit être rattachée à un contexte. Ce contexte doit être directement relié à un modèle UML. Même si le contexte d'une contrainte OCL peut être n'importe quel élément de modèle, nous ne nous intéressons ici qu'aux classes UML et aux propriétés des classes (opérations ou attributs).

La signification du contexte pour une contrainte OCL est la suivante :

- Si le contexte est une classe UML, la contrainte OCL est évaluée sur toutes les instances de la classe UML.
- Si le contexte est une propriété (opération ou attribut) d'une classe, la contrainte OCL est évaluée sur toutes les instances de la classe UML mais porte uniquement sur la propriété identifiée.

Dans une contrainte OCL, le mot-clé `self` permet de référencer l'élément sur lequel porte l'expression OCL. Grâce à cette référence, l'opérateur de navigation « . » permet de naviguer vers les propriétés de l'élément et vers les autres éléments avec lequel l'élément est associé. De proche en proche, il est possible d'évaluer grâce à d'autres expressions OCL les éléments appartenant à l'entourage du contexte initial.

Par exemple, dans l'expression suivante, si nous considérons que le contexte est la classe `CompteBancaire`, nous voyons qu'il est possible d'obtenir le nom du propriétaire du compte :

```
self.prop.nom
```

Le contexte peut être explicité à l'aide du mot-clé `context`. Le mot-clé `self` devient dès lors implicite. L'exemple suivant est équivalent à l'exemple précédent :

```
context CompteBancaire
prop.nom
```

Les invariants

Une contrainte OCL peut exprimer un invariant (mot-clé `inv`) sur une classe UML. Cela signifie que la contrainte OCL doit être tout le temps vraie pour toutes les instances de la classe.

L'exemple d'invariant suivant exprime que le solde d'un compte bancaire doit être toujours supérieur à – 1 000 euros :

```
context CompteBancaire
inv: solde > -1000
```

Pré- et postconditions

Une contrainte OCL peut exprimer une pré- et une postcondition, *via* les mots-clés `pre` et `post`, sur une opération UML. Ces conditions permettent de contraindre les états des objets avant et après l'invocation de l'opération. Le mot-clé `result` permet d'identifier la valeur de retour de l'opération.

L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur du solde doit être positive avant toute invocation de l'opération `debit` :

```
context CompteBancaire::debit():Integer
pre: solde>0
```

L'exemple suivant permet de spécifier, pour un compte bancaire, que la valeur de retour de l'opération `getSolde` est égale à la valeur du solde du compte :

```
context CompteBancaire::getSolde():Integer
post: result=solde
```

OCL permet aussi, grâce au mot-clé `@pre`, de manipuler dans une postcondition la valeur d'un attribut avant invocation de l'opération. L'exemple suivant spécifie que la valeur du solde lors d'un crédit est égale à la valeur du compte avant invocation augmentée de la somme créditée :

```
context CompteBancaire::crédit(s:Integer)
post: result=solde@pre+s
```

Les opérations de sélection

Étant donné qu'OCL permet de définir des requêtes sur des éléments de modèle, il est possible de l'utiliser pour définir le corps d'opérations qui ne font que sélectionner des éléments de modèle. Nous utilisons pour cela le mot-clé `body`.

Par exemple, il est possible de spécifier en OCL le corps de l'opération permettant de sélectionner tous les comptes bancaires positifs d'une personne :

```
context Personne::getComptePositif():Set
pre: self.cpts.notEmpty()
body: self.cpts->select(c | c.solde>0)
```

Types et valeurs

Cette section introduit le système de types du langage OCL, qui est légèrement différent de celui d'UML. Il est nécessaire de comprendre ce système de types pour appréhender le mécanisme d'évaluation des expressions OCL.

☛ Les types de base

Les types de base du langage OCL sont les booléens (`Boolean`), les entiers (`Integer`), les réels (`Real`) et les chaînes de caractères (`String`).

OCL fournit les opérations de base sur ces types, telles que et, ou, addition, soustraction, concaténation, etc.

☛ Les types UML

Nous avons déjà vu que le contexte d'une contrainte OCL pouvait être une classe UML ou n'importe laquelle de ses propriétés.

Les classes UML sont considérées comme des types par OCL. Les objets instances de ces classes sont donc considérés comme les valeurs de ces types.

☛ Les variables

Il est possible de définir des variables à l'aide du mot-clé `let`. Cependant, OCL n'étant pas un langage à effet de bord, les variables OCL ont une valeur fixe, qui ne peut évoluer. Elles sont, en quelque sorte, des raccourcis qui peuvent être utilisés dans d'autres expressions OCL.

☛ Les opérations ajoutées

Il est possible de définir de nouvelles opérations à l'aide du mot-clé `def` afin de les utiliser dans d'autres expressions OCL. Les opérations ajoutées sont relativement semblables aux opérations existantes sur les classes UML, si ce n'est qu'elles ne peuvent être que des opérations de sélection et qu'elles doivent être entièrement spécifiées en OCL.

☛ Conformité des types

Pour OCL, un type `T1` est conforme à un type `T2` si une instance de `T1` peut être substituée à toute place où une instance de `T2` est demandée.

Pour les types de base, seul le type entier est conforme au type réel. Pour les types UML, c'est l'héritage entre classes qui indique la conformité entre types.

La conformité de types est transitive. Si `T1` est conforme à `T2` et que `T2` soit conforme à `T3`, `T1` est conforme à `T3`.

Une expression OCL est dite valide si tous ses types sont en conformité. En OCL, une expression qui ferait une addition entre une chaîne de caractères et un entier serait non valide, `String` et `Integer` n'étant pas conformes.

Propriétés des classes UML

Cette section montre la façon dont sont considérées les propriétés des classes (attributs, opérations et associations) dans OCL.

☛ Les attributs

Nous avons déjà vu qu'il était possible avec OCL d'obtenir la valeur d'un attribut d'une classe. Plus précisément, il est possible d'obtenir la valeur d'un attribut portée par un objet instance d'une classe.

Le type de la valeur correspond au type de l'attribut tel qu'il est défini dans la classe. Si, par exemple, l'attribut a pour type `String`, dans OCL, le type de la valeur sera `String`. Par contre, si l'attribut a pour type `String[]` (tableau de chaîne de caractères), dans OCL, le type de la valeur sera `Sequence` (collection de valeurs).

En UML, certains attributs sont statiques. Dans ce cas, la valeur de l'attribut est portée par la classe et non par les objets. Il est aussi possible en OCL d'obtenir la valeur de ces attributs. Pour ce faire, il faut utiliser non pas le mot-clé `self` mais directement le nom de la classe.

Rappelons qu'OCL étant un langage sans effet de bord, il n'est pas possible de modifier la valeur d'un attribut.

☛ Les opérations

En OCL, il est possible d'invoquer des opérations sur des objets UML. En fait, seules les opérations de sélection sont invocables car elles ne modifient pas l'état d'un objet (elles ne modifient pas les valeurs des attributs de l'objet). On peut aussi faire des opérations produisant un résultat de test.

Les opérations qui ne sont pas des opérations de sélection ne peuvent être invoquées. Cela violerait le fait qu'OCL est un langage sans effet de bord.

Nous avons déjà vu qu'en OCL il était possible de définir de nouvelles opérations de sélection. Celles-ci doivent être entièrement définies (leur corps doit être spécifié en OCL) et être rattachées à une classe UML.

☛ Les associations

En OCL, il est possible de naviguer *via* les associations UML. Les noms de rôles des associations sont utilisés pour définir comment se fait la navigation par défaut.

La navigation peut se faire de proche en proche. Il est en ce cas possible de chaîner les navigations et ainsi de parcourir une chaîne d'objets.

La navigation *via* les associations ayant une multiplicité « multiple », c'est-à-dire un à plusieurs, elle s'effectue grâce aux collections d'éléments. Les collections sont des types OCL. Elles offrent des fonctions permettant d'itérer sur les éléments de la collection *(voir la section suivante).*

Les collections

Les collections sont des types OCL décrivant des ensembles d'éléments. OCL fournit plusieurs fonctions permettant la manipulation des collections.

☞ Les différentes sortes de collections

Pour manipuler les ensembles d'éléments, OCL propose différentes sortes de collections, appelées `Bag`, `Set`, `OrderedSet` et `Sequence`. Toutes ces collections sont des ensembles d'éléments de même type.

- `Bag`. Collection d'éléments dans laquelle les éléments ne sont pas ordonnés et où les doublons sont possibles (un même élément peut apparaître plusieurs fois dans la collection).
- `Set`. Collection d'éléments dans laquelle les éléments ne sont pas ordonnés et où les doublons ne sont pas possibles (un même élément ne peut apparaître plusieurs fois dans la collection).
- `OrderedSet`. Collection d'éléments dans laquelle les éléments sont ordonnés et où les doublons ne sont pas possibles.
- `Sequence`. Collection d'éléments dans laquelle les éléments sont ordonnés et où les doublons sont possibles.

☞ Manipulation des collections

OCL propose différentes fonctions pour manipuler les collections, quel que soit leur type.

La fonction `select()` permet de sélectionner un sous-ensemble d'éléments d'une collection en fonction d'une condition. Nous pouvons obtenir le contraire de la fonction `select()` grâce à la fonction `reject()`.

L'exemple suivant sélectionne les comptes bancaires dont le solde est positif :

```
context Personne
self.cpts->select(c | c.solde>0)
```

La fonction `forAll()` permet de vérifier si tous les éléments d'une collection respectent une expression OCL. L'expression OCL à respecter est passée en paramètre.

L'exemple suivant permet de s'assurer que tous les comptes bancaires ont un solde positif :

```
context Personne
self.cpts->forAll(c | c.solde>0)
```

La fonction `exist()` permet de vérifier si au moins un élément respectant une expression OCL existe dans la collection. L'expression OCL à respecter est passée en paramètre de la fonction.

L'exemple suivant permet de s'assurer qu'il existe au moins un compte bancaire avec un solde positif :

```
context Personne
self.cpts->exist(c | c.solde>0)
```

Les messages

Lors d'invocations d'opérations, UML considère que des messages sont échangés entre les objets. En OCL, il est possible de contraindre ces messages. Ce mécanisme permet de spécifier les interactions entre plusieurs objets.

L'exemple suivant illustre une interaction entre deux objets, un sujet (`Subject`) et un observateur (`Observer`), spécifiant que lorsqu'on invoque l'opération `hasChanged` sur un sujet, le sujet invoque l'opération `update` sur un observateur, avec `12` et `14` comme valeurs des paramètres de cette opération :

```
context Subject::hasChanged()
post: Observer^update(12,14)
```

Le métamodèle OCL2.0

Avant la version 2.0, OCL n'était spécifié qu'en langage naturel (anglais). Cela présentait l'inconvénient de rendre difficile l'alignement avec le standard UML. De plus, les expressions OCL étaient peu pérennes et peu productives.

L'objectif de la version 2.0 est principalement de spécifier OCL grâce à un métamodèle afin que celui-ci soit facilement intégrable dans MDA. Depuis OCL2.0, les expressions OCL sont des modèles pérennes, productifs et explicitement liés aux modèles UML. Cela offre un gain significatif d'expressivité aux modèles UML.

OCL et UML

Les expressions OCL portent toutes sur des éléments de modèles UML. Il y a donc un fort lien entre le métamodèle OCL et le métamodèle UML.

La figure 4.2 illustre les métaclasses participant à ce lien.

Dans le métamodèle UML, la métaclasse `Constraint` représente n'importe quelle contrainte. Cette métaclasse est reliée à la métaclasse `ModelElement`, qui joue le rôle de `constrainedElement`. Il est ainsi possible d'attacher une contrainte à n'importe quel élément du modèle. La métaclasse `Constraint` est aussi reliée à la métaclasse `Expression`, qui joue le rôle de `body`. Le corps des contraintes UML est de la sorte spécifié à l'aide d'une expression. UML laisse le choix du langage dans lequel les expressions doivent être écrites.

OCL2.0 pour UML1.4 et OCL2.1 pour UML2.0 Superstructure

OCL2.0 est lié au métamodèle UML1.4. Les liens entre OCL et le métamodèle UML2.0 ne sont pas encore finalisés. Cela peut paraître curieux, mais ils seront définis dans la version OCL2.1.

Par souci de transparence, nous préférons montrer dans ce chapitre les liens existant entre OCL et UML1.4, sachant que les métaclasses utilisées dans les liens entre OCL2.0 et UML1.4 appartiennent quasiment toutes au package core d'UML1.4. D'un point de vue conceptuel, ces métaclasses représentent les concepts des diagrammes de classes. Elles sont donc en tout point comparables aux métaclasses du package UML2.0 Superstructure kernel *(voir le chapitre 2)*. Par exemple, la métaclasse `Class` du package core d'UML1.4 est équivalente à la métaclasse `Class` du package kernel d'UML2.0 Superstructure.

Le métamodèle OCL définit quant à lui la métaclasse `ExpressionInOcl`, qui hérite de la métaclasse `Expression`. Grâce à cet héritage, cette métaclasse représente des contraintes UML écrites avec OCL. Cette métaclasse est reliée à la métaclasse `OclExpression`, qui joue le rôle de `bodyExpression`. C'est cette dernière métaclasse qui représente réellement la spécification d'une contrainte en OCL.

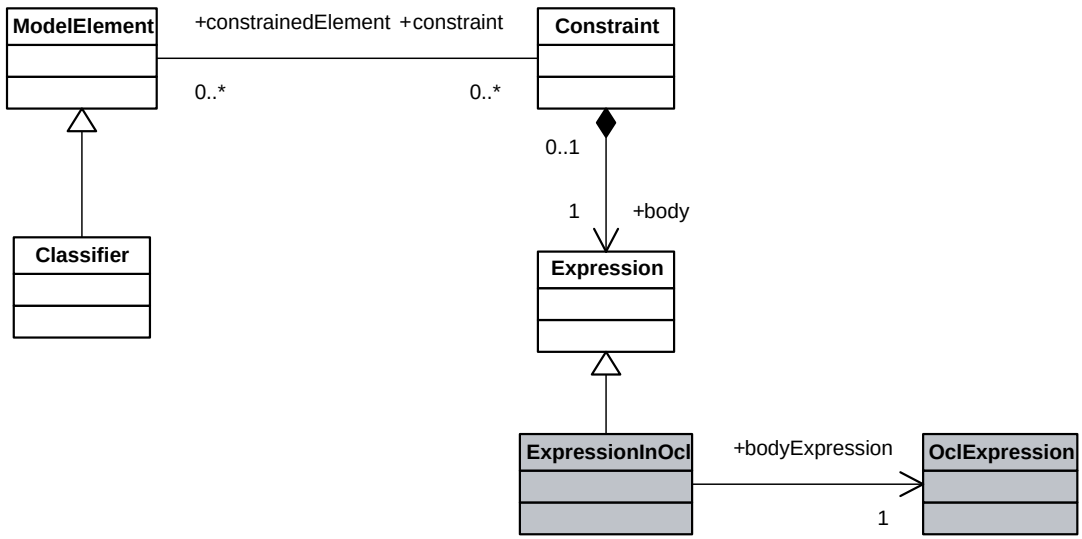


Figure 4.2
Liens entre les métamodèles UML et OCL

Le package *Expressions*

Le package `Expressions` du métamodèle OCL, illustré en partie dans les figures suivantes, contient toutes les métaclasses permettant de représenter les expressions OCL sous forme de modèles.

Nous allons plus particulièrement présenter les métaclasses `OclExpression`, `VariableExp` et `ModelPropertyCallExp`, qui sont les trois classes de base permettant de modéliser une expression OCL.

La figure 4.3 illustre la partie principale du package `Expressions`.

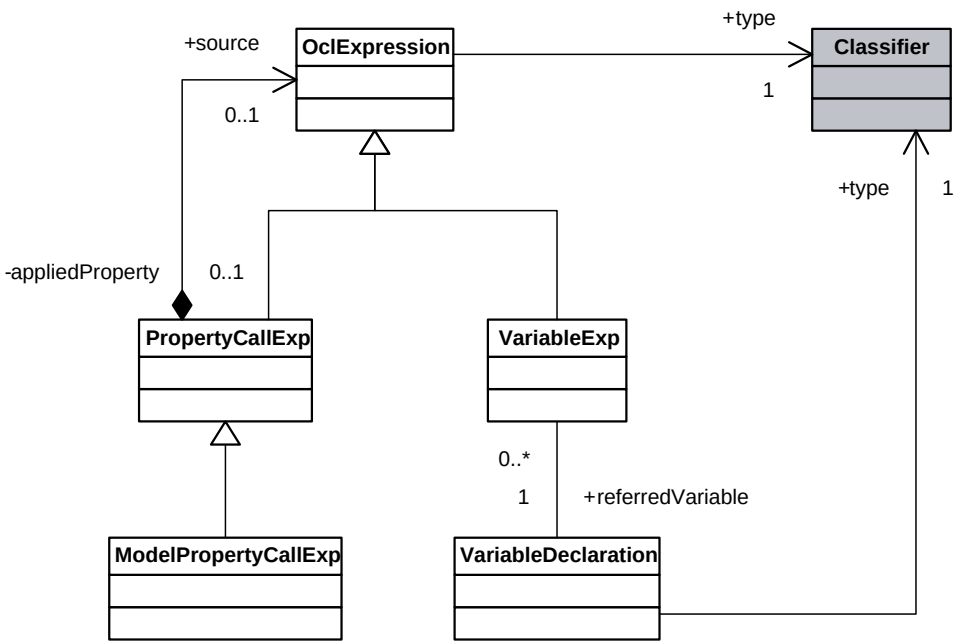


Figure 4.3
La partie OclExpression *du package* Expressions

La métaclasse la plus importante est `OclExpression`. Héritée par toutes les autres métaclasses de ce package, elle représente n'importe quelle expression OCL.

Cette métaclasse est reliée à la métaclasse `Classifier` du métamodèle UML, qui joue dans cette association le rôle de type. Cela signifie qu'une expression OCL a un type qui est un classifier, c'est-à-dire n'importe quel type OCL *(voir le package* `Types` *à la section suivante).* Si le type d'une expression OCL est booléen, l'expression peut être considérée comme une contrainte. C'est d'ailleurs pourquoi la métaclasse `Constraint` n'apparaît pas dans ce package.

La métaclasse `VariableExp` représente une variable. Cette métaclasse est reliée à la métaclasse `VariableDeclaration`, qui représente une déclaration de variable. C'est grâce à cette métaclasse, par exemple, que la variable `self` est déclarée. La métaclasse `VariableDeclaration` est reliée à la métaclasse `Classifier`, qui joue le rôle de type. Cette association a pour but de définir le type de la variable ou le type du contexte si la variable est `self`.

La métaclasse `PropertyCallExp` représente un appel de propriétés. Cette métaclasse est héritée par la métaclasse `ModelPropertyCallExp`, qui représente un appel de propriétés d'une classe UML. Dans OCL, une propriété d'une classe UML peut aussi bien être un attribut, une opération ou une référence.

La figure 4.4 représente les métaclasses permettant de modéliser les expressions OCL qui sont des appels aux propriétés des classes UML (`ModelPropertyCallExp`). Parmi elles, nous ne présentons que la métaclasse `AttributeCallExp`, qui représente un appel à un attribut d'une classe. Cette métaclasse est reliée à la métaclasse `Attribut` du métamodèle UML afin d'identifier l'attribut appelé.

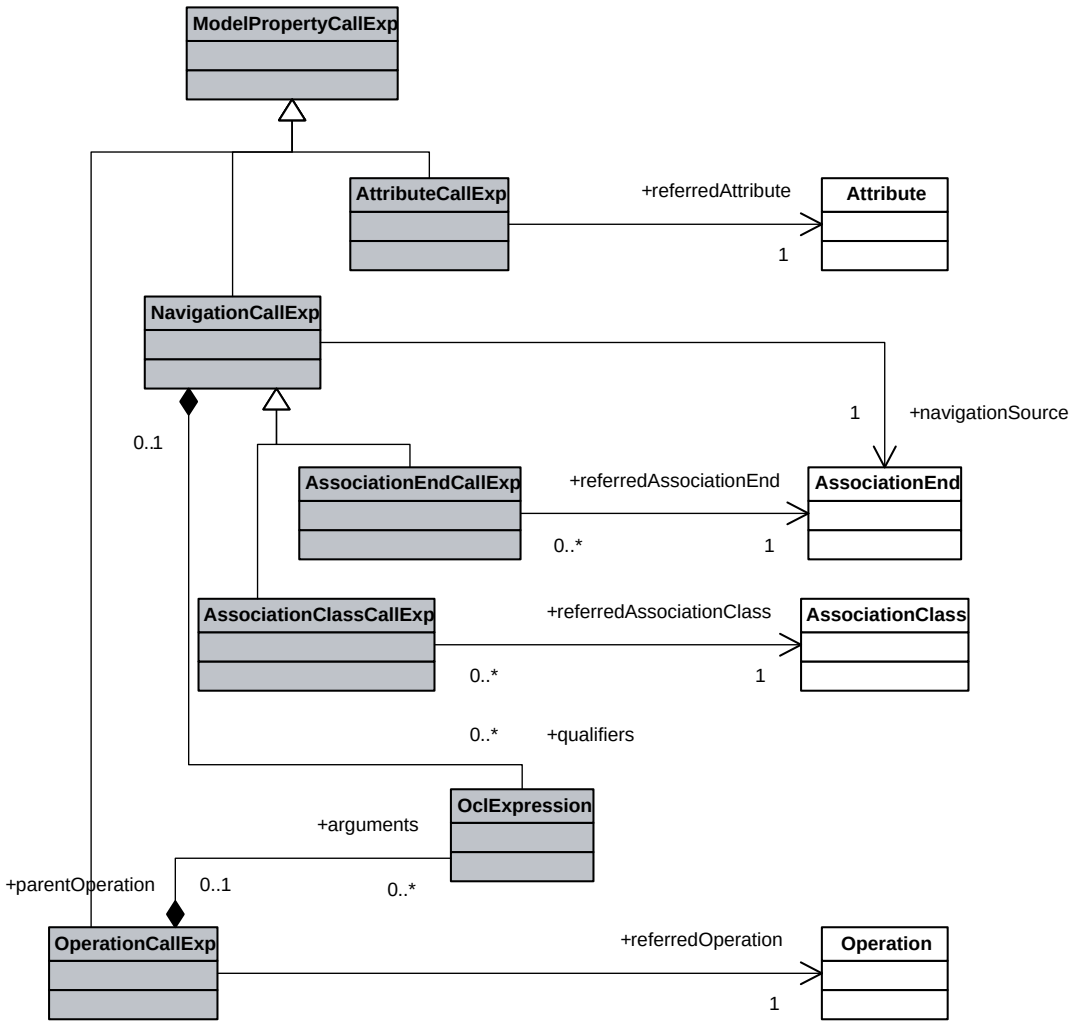


Figure 4.4

La partie ModelPropertyCallExp *du package* Expressions

Le package *Types*

Nous avons déjà indiqué que toute expression OCL était typée. Dans le métamodèle OCL, c'est le package `Types`, illustré à la figure 4.5, qui définit tous les types OCL. Il contient donc toutes les métaclasses nécessaires à l'élaboration des types OCL.

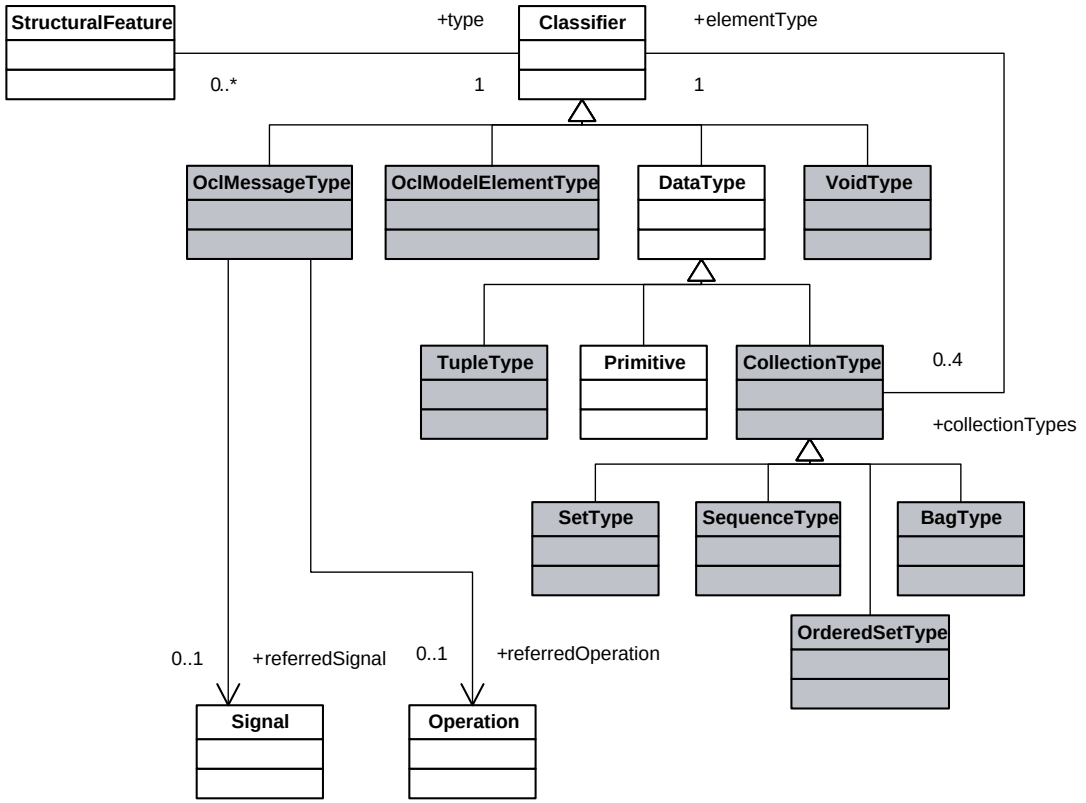


Figure 4.5
Le package Types *d'OCL2.0*

Ce package référence la métaclasse `Classifier`, qui vient du package core d'UML1.4 (semblable à la métaclasse `Classifier` du package kernel d'UML2.0 Superstructure). Toutes les métaclasses qui représentent des types OCL héritent de cette métaclasse. Cela signifie que tous les types OCL sont des classifiers UML, c'est-à-dire des types UML.

Le package `Types` contient la métaclasse `VoidType`, qui permet de représenter le type `void`. Il contient aussi la métaclasse `OclModelElementType`, qui permet de représenter dans OCL les types UML, ceux-ci étant parfois manipulés dans les expressions OCL. Ce package contient enfin la métaclasse `OclMessageType`, qui permet de typer les messages OCL. Cette métaclasse étant reliée aux métaclasses `Signal` et `Operation` d'UML, les messages sont liés soit à un signal, soit à une opération.

Le package `Types` référence aussi la métaclasse `DataType`, qui vient du package core d'UML1.4. Les métaclasses qui représentent les collections (`CollectionType`, `SetType`, `SequenceType`, `OrderedSetType` et `BagType`) héritent de cette métaclasse. Cela signifie que les collections UML sont des types de données UML. Ce package référence aussi la métaclasse `Primitive`, qui représente les types de base (`Integer`, `Boolean`, etc.). Ce package contient enfin la métaclasse `TupleType`, qui permet de mettre dans une même structure différents types OCL.

En association avec chacune de ces métaclasses, OCL définit un ensemble d'opérations. Ces dernières sont celles que l'on peut faire classiquement sur les valeurs de ces types. Nous y retrouvons les opérations d'addition, de soustraction et de comparaison d'entiers, d'égalité de booléens, de manipulation de caractères, etc. Elles sont appelées dans les expressions OCL lorsque nous voulons, par exemple, exprimer une égalité ou une somme.

Modèle *versus* texte

Le métamodèle OCL permet de représenter n'importe quelle expression OCL sous forme de modèle. Cette représentation permet de rendre les expressions OCL pérennes et productives. De plus, le lien fort qui unit les modèles UML et les expressions OCL représentées sous forme de modèles permet d'exploiter pleinement les modèles UML dans une approche MDA. OCL contribue de la sorte à faire des modèles UML des PIM de MDA.

Pour des raisons évidentes de simplicité d'utilisation, OCL reste avant tout un langage textuel. Les contraintes OCL sont donc toujours représentées textuellement, avec la syntaxe que nous avons brièvement présentée en début de chapitre.

Un effort important a été fourni pour définir la façon de passer automatiquement de la forme textuelle à la forme modèle. Cette traduction entre le format textuel et le format modèle n'est pas triviale. Nous allons l'illustrer au travers d'un exemple que nous présentons d'abord sous un format textuel puis sous un format modèle.

L'exemple est le suivant :

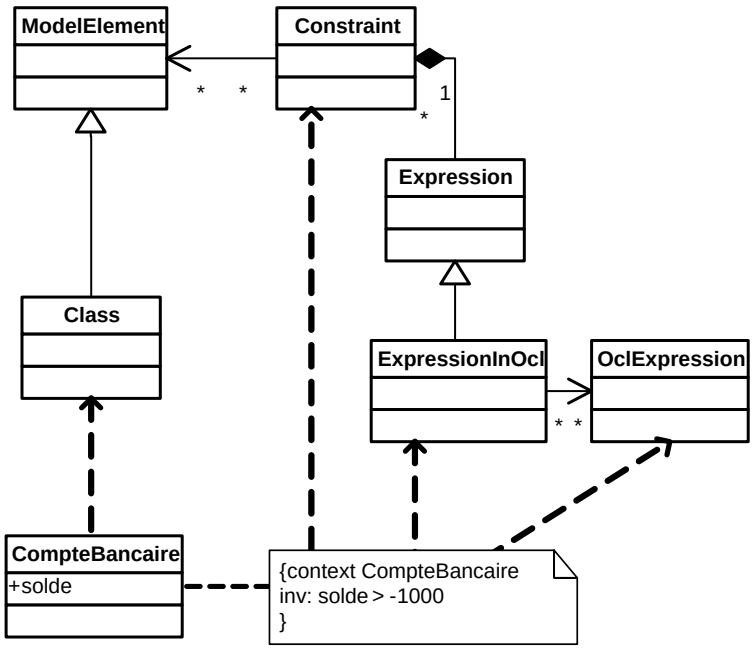
```
context CompteBancaire
inv: solde > -1000
```

Nous avons déjà présenté cet exemple en début de chapitre. Il s'agit d'une contrainte OCL attachée à la classe UML `CompteBancaire`, qui permet d'exprimer le fait que le solde du compte bancaire ne doit pas être inférieur à – 1 000 euros.

La figure 4.6 illustre une partie du modèle de cette contrainte OCL. Nous voyons que la contrainte est une contrainte UML (`Constraint`) et qu'elle est attachée à une classe UML, en l'occurrence un élément de modèle UML. Cette contrainte contient une expression, `ExpressionInOcl`, qui est spécifiée à l'aide de l'expression OCL `OclExpression`.

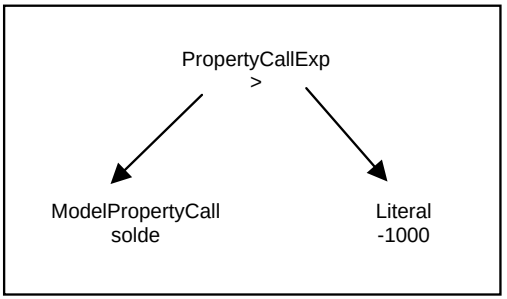
La figure 4.7 détaille cette expression OCL sous forme de modèle. L'expression OCL commence par un appel à opération (`PropertyCallExp`). L'opération appelée est `>`, de la métaclasse `Integer`. Un des paramètres de cette opération est la valeur littérale `-1000`.

Figure 4.6
Contrainte OCL sous forme de modèle (1/2)



L'autre paramètre de l'opération est un appel à l'attribut `solde` de la classe `CompteBancaire` (`ModelPropertyCallExp`).

Figure 4.7
Contrainte OCL sous forme de modèle (2/2)



Cet exemple illustre le fait qu'il est possible de traduire automatiquement les expressions OCL textuelles vers des modèles, même si cette traduction est très difficile à implémenter pour le constructeur d'outils de modélisation. Grâce à cette traduction, les expressions OCL peuvent être facilement saisies par les utilisateurs des outils de modélisation (sous forme textuelle) tout en restant pérennes et productives car transformées automatiquement en modèles.

En résumé

Nous avons vu que le langage OCL permettait de spécifier des contraintes sur les modèles UML. Grâce à ces contraintes, il est possible de spécifier le corps des opérations des classes. Les contraintes OCL sont indépendantes des langages de programmation. La traduction d'une contrainte OCL en un jeu d'instructions dans un langage de programmation est d'ailleurs un problème qui n'est pas encore résolu dans MDA.

OCL permet de préciser les modèles UML tout en faisant en sorte qu'ils soient toujours indépendants des plates-formes de programmation. Il est principalement utilisé dans MDA pour l'élaboration des PIM (Platform Independent Model).

Afin de clarifier l'intégration du standard OCL dans MDA, l'OMG a décidé de le standardiser sous forme de métamodèle. Grâce au métamodèle OCL, les contraintes OCL sont désormais représentées sous forme de modèles.

Le métamodèle OCL est relié au métamodèle UML de façon à exprimer la dépendance entre UML et OCL. Ce lien est un lien interne vers le métamodèle OCL *(voir le chapitre 1)*. Il est donc intégralement défini dans le métamodèle OCL. Ce lien est actuellement établi pour le métamodèle UML1.4. La version OCL2.1 spécifiera le lien vers UML2.0 Superstructure. L'évolution du lien vers UML2.0 Superstructure ne devrait pas poser trop de problème étant donné le rapprochement conceptuel entre les packages core d'UML1.4 et kernel d'UML2.0 Superstructure.

Grâce à ce lien, il est possible de naviguer de manière informatique de la contrainte OCL vers le modèle UML. Il est de la sorte envisageable d'automatiser l'évaluation des contraintes OCL. Au finale, on peut dire que le métamodèle OCL a permis de rendre les contraintes OCL productives.

Pour ne pas rendre l'édition des contraintes OCL trop complexe pour les utilisateurs, l'OMG a préféré conserver la syntaxe textuelle d'OCL. Cette représentation n'a qu'un objectif d'édition. La traduction de la représentation textuelle en une représentation sous forme de modèle (et réciproquement) est si stratégique pour OCL que l'OMG l'a standardisée, même si elle reste difficile à implémenter.

Le langage AS

Jusqu'à sa version 1.4, UML était très critiqué parce qu'il ne permettait pas de spécifier des créations, des suppressions ou des modifications d'éléments de modèles. Ces actions ne pouvant pas non plus être spécifiées à l'aide du langage OCL, puisque celui-ci est sans effet de bord, il était nécessaire de standardiser un nouveau langage. C'est ce qui a donné naissance au langage AS (Action Semantics).

L'objectif d'AS est de permettre la spécification d'actions. Une action, au sens AS du terme, est une opération sur un modèle qui fait changer l'état du modèle. Grâce aux actions, il est possible de modifier les valeurs des attributs, de créer ou de supprimer des objets, de

créer de nouveaux liens entre les objets, etc. Le concept d'action permet de spécifier pleinement le corps des opérations UML.

AS a tout d'abord été standardisé comme un langage à part entière, lié au standard UML, avant d'être inclus dans la version 1.5 d'UML. Dans UML2.0, il est enfin totalement intégré au métamodèle. Cette section présente AS tel que défini dans le standard UML2.0 Superstructure.

Le métamodèle AS

AS n'est standardisé que sous forme de métamodèle, et aucune syntaxe concrète n'est définie, contrairement à OCL. Ce standard laisse donc la liberté à chacun d'utiliser la syntaxe qu'il souhaite. Le fait qu'il n'existe aucune syntaxe de référence est toutefois une lacune assez gênante puisqu'il est impossible de trouver des exemples présentant de manière textuelle des ensembles d'actions.

Dans UML2.0, AS constitue le socle de définition de la dynamique des modèles. Les modèles dynamiques UML (activité, séquence et machine à états) s'appuient dessus. AS assure l'unification et la cohérence de l'ensemble des modèles dynamiques.

Même si l'objectif premier d'AS était de permettre la spécification des corps des opérations UML et leur transformation vers des langages de programmation, il apparaît aujourd'hui que d'autres utilisations sont possibles.

AS peut être utilisé pour des transformations de modèles *(voir la partie II de l'ouvrage)*. Il est possible d'exprimer des transformations de modèles UML avec AS car celui-ci propose des opérations de création, de suppression et de modification d'éléments de modèles.

Une autre utilisation possible d'AS est l'exécution de modèles UML. Il est envisageable de construire des machines virtuelles UML permettant l'exécution des actions directement sur les modèles. Cette utilisation relève encore de la recherche mais semble prometteuse selon l'OMG puisqu'elle est en cours de standardisation.

Les packages constituant AS

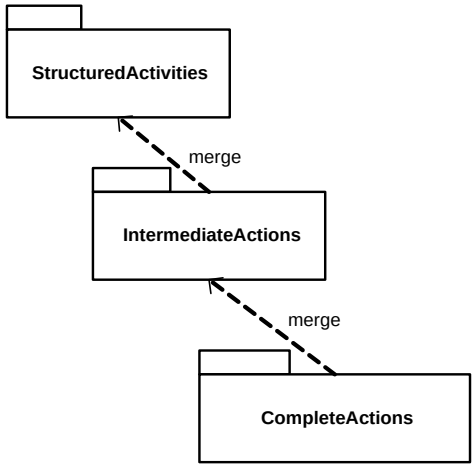
Le métamodèle UML2.0 Superstructure intègre totalement le métamodèle AS. Plus précisément, le métamodèle AS se retrouve dans les packages `StructuredActivities`, `CompleteActions` et `IntermediateActions` du métamodèle UML2.0 Superstructure.

Le package `IntermediateActions` définit les métaclasses représentant les actions classiques de création, de suppression et de modification d'éléments de modèle. Ce package merge le package `StructuredActivities`, lequel contient les métaclasses nécessaires à la représentation d'activités.

Le package `CompleteActions` définit les métaclasses représentant des actions très spécifiques, telles les actions permettant de modifier les liens d'instanciation entre des objets et leur classe. Nous n'utilisons pas ce package ni ce genre d'action dans la suite de l'ouvrage. Notons simplement que ce package merge le package `IntermediateActions`.

La figure 4.8 illustre la découpe des relations entre les packages du métamodèle UML2.0 Superstructure qui incarnent les concepts d'AS.

Figure 4.8
Les packages AS dans UML2.0



Le package *StructuredActivities*

Le package `StructuredActivities` définit principalement le concept d'activité comme étant un enchaînement d'actions *(voir figure 4.9).*

Il contient la métaclasse `Activity`, qui représente la notion d'activité, et la métaclasse `ActivityNode`, qui représente une étape, ou nœud, de l'activité. La métaclasse `Activity` est reliée à la métaclasse `ActivityNode` afin d'exprimer le fait que l'activité est constituée d'un ensemble d'étapes.

Ce package contient aussi la métaclasse `Action`, qui hérite de la métaclasse `ActivityNode`. Cela signifie qu'une étape d'une activité peut être une action. D'autres métaclasses héritent de la métaclasse `ActivityNode`, mais nous ne nous attardons pas sur elles.

Le package `StructuredActivities` contient la métaclasse `ActivityEdge`, qui représente un lien entre deux étapes d'une même activité. Cette métaclasse est reliée à la métaclasse `ActivityNode` par deux méta-associations. Cela permet de structurer les activités sous forme de graphes d'actions.

Le package `StucturedActivities` définit un peu plus précisément la notion d'action comme étant quelque chose d'exécutable, qui accepte en entrée et rend en sortie des données *(voir figure 4.10).* Ce package contient en effet la métaclasse `Pin`, qui représente la notion de donnée. Un pin comporte une valeur. Les métaclasses `OutputPin` et `InputPin` représentent les données en entrée et en sortie d'une action. La métaclasse `Action` est donc reliée aux métaclasses `OutputPin` et `InputPin`. Grâce à ces métaclasses, il est possible de spécifier que les données en sortie d'une action sont consommées en entrée d'une autre action. Dit autrement, cela permet de spécifier des flots de données entre les actions.

Figure 4.9
La métaclasse Activity *dans le package* StructuredActivities

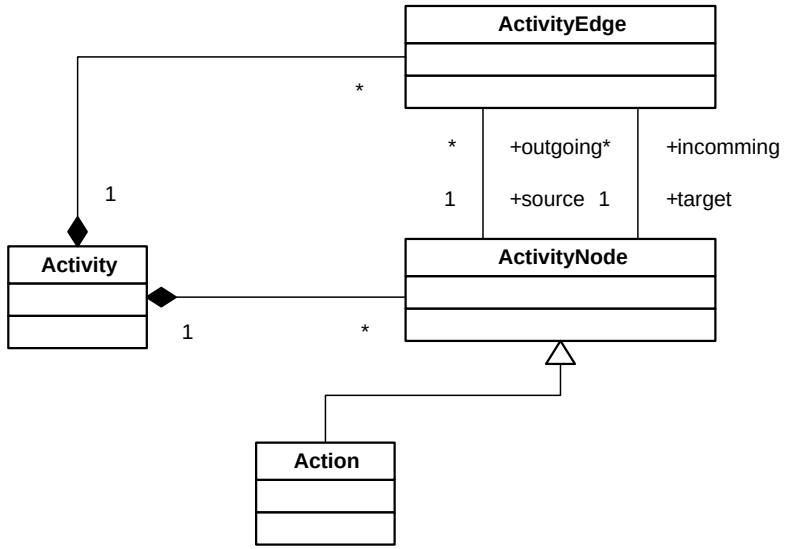
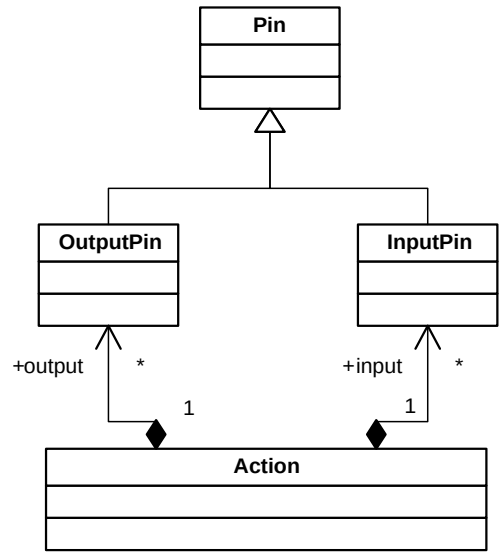


Figure 4.10
La métaclasse Action *dans le package* StructureActivities



Le package *IntermediateActions*

Le package `IntermediateActions` contient toutes les métaclasses permettant de représenter les actions classiques de création, de suppression et de modification d'éléments de modèle. Nous ne détaillons pas ici la totalité de ces métaclasses et ne nous concentrons que sur certaines d'entre elles afin de bien faire comprendre comment élaborer une suite d'instructions en utilisant AS. Nous renvoyons le lecteur curieux au standard UML2.0 Superstructure pour obtenir la liste complète des métaclasses représentant des actions AS.

La figure 4.11 illustre la partie du package `IntermediateAction` contenant la métaclasse `CallOperationAction`. Cette métaclasse représente une action qui est un appel à une opération d'une classe UML. Cette métaclasse est reliée à la métaclasse `Operation` du métamodèle UML afin d'identifier l'opération à appeler. Cette métaclasse est aussi reliée à la métaclasse `InputPin` afin d'identifier l'objet sur lequel sera appelée l'opération.

La métaclasse `InvocationAction` représente une action qui est une invocation. Cette métaclasse est reliée à la métaclasse `InputPin` afin d'identifier les arguments de l'invocation et à la métaclasse `OutputPin` afin d'identifier le résultat de l'invocation.

La métaclasse `CallOperationAction` hérite de la métaclasse `CallAction`, qui hérite à son tour de la métaclasse `InvocationAction`. La métaclasse `CallOperationAction` a donc elle aussi un résultat et des paramètres.

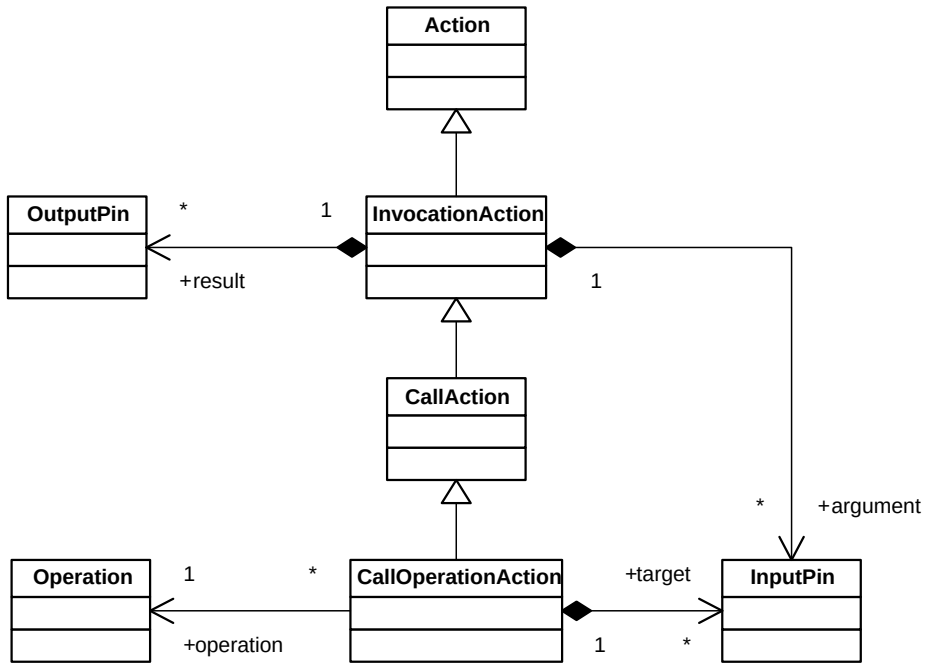


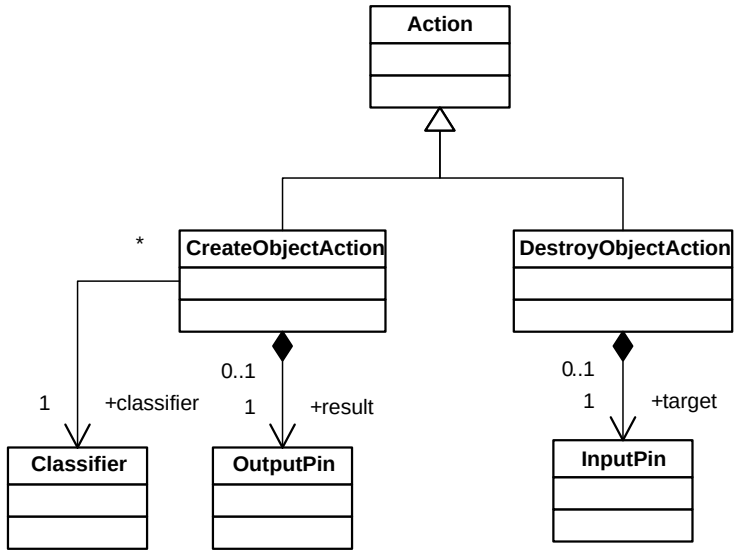
Figure 4.11
La métaclasse CallOperationAction

La figure 4.12 illustre une autre partie du package `IntermediateActions` contenant les métaclasses `CreateObjectAction` et `DestroyObjectAction`. Ces métaclasses représentent respectivement des actions de création et de destruction d'objet.

La métaclasse `CreateObjectAction` est reliée à la métaclasse `Classifier` afin d'identifier la classe de l'objet à construire. Elle est aussi reliée à la métaclasse `OutputPin` afin d'identifier l'objet qui sera créé après l'exécution de l'action.

La métaclasse `DestroyObjectAction` est reliée à la métaclasse `InputPin` afin d'identifier l'objet à détruire.

Figure 4.12
Les métaclasses CreateObjectAction *et* DestroyObjectAction



La figure 4.13 illustre la partie du package `IntermediateActions` contenant la métaclasse `StructuralFeature`. Cette métaclasse représente une action d'accès à une propriété (attribut ou référence, par exemple). Elle est donc reliée à la métaclasse `StructuralFeature` du métamodèle UML afin d'identifier la propriété qui sera accédée.

La métaclasse `ReadStructuralFeatureAction`, qui hérite de la métaclasse `StructuralFeatureAction`, représente l'action de lecture de la valeur d'une propriété. Cette métaclasse est reliée à la métaclasse `OutputPin` afin d'identifier cette valeur retournée.

La métaclasse `WriteStructuralFeatureAction`, qui hérite aussi de la métaclasse `StructuralFeatureAction`, représente une action qui écrit une nouvelle valeur pour une propriété. Cette métaclasse est reliée à la métaclasse `InputPin`, qui représente la nouvelle valeur de la propriété.

La métaclasse `AddStructuralFeatureValueAction`, qui hérite de la métaclasse `WriteStructuralFeatureAction`, représente l'action d'écriture d'une nouvelle valeur pour une propriété qui est un tableau. Cette métaclasse est reliée à la métaclasse `InputPin` jouant le rôle d'`insertAt` afin de définir l'index de la case du tableau dans laquelle se fera l'écriture.

Le package `IntermediateActions` définit d'autres métaclasses représentant toutes les autres actions exécutables sur des éléments de modèle. Il existe, par exemple, des actions pour ajouter des liens entre les objets, émettre des messages, naviguer parmi les instances d'une classe, etc. Le package contient en tout une cinquantaine de métaclasses. Pour une liste complète de ces métaclasses, voir directement le standard UML2.0 Superstructure.

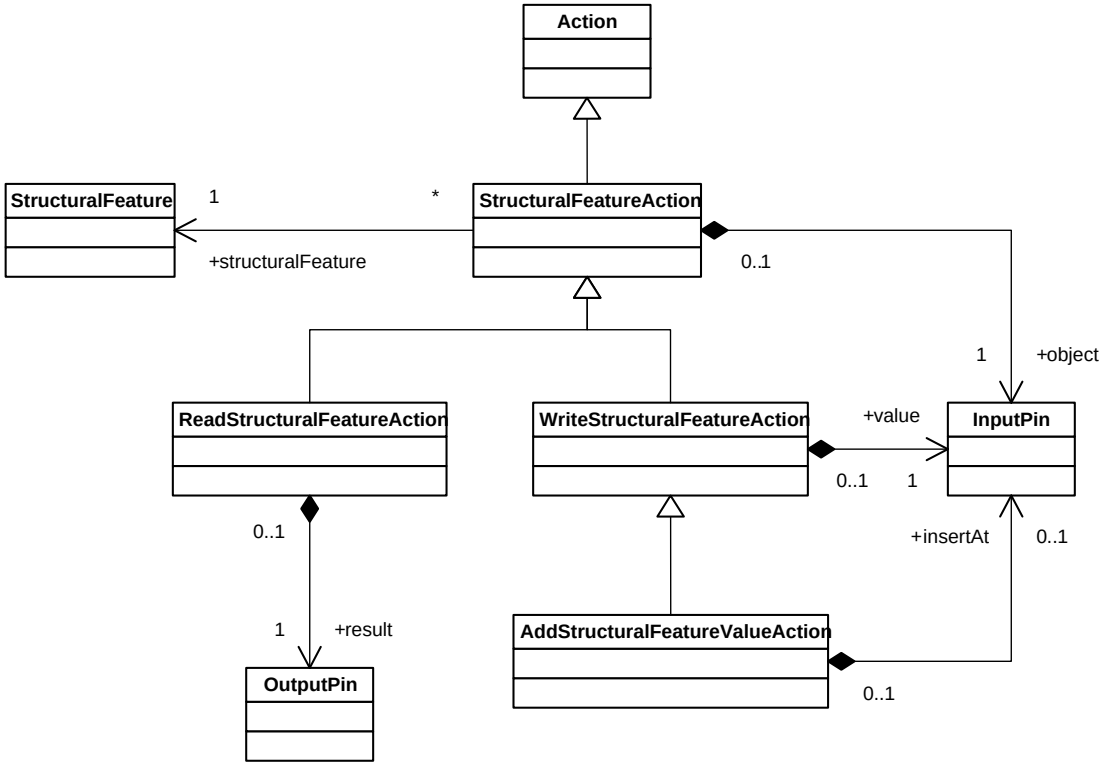


Figure 4.13

Les métaclasses ReadStructuralFeatureAction *et* WriteStructurealFeatureAction

Lien avec la syntaxe concrète

Comme expliqué précédemment, AS n'est défini que sous forme de métamodèle et ne propose pas de format concret. Il n'est donc pas possible de spécifier des activités sous un format concret.

Le standard précise cependant que plusieurs formats concrets peuvent être utilisés et qu'il est du ressort du concepteur d'outil de modélisation de fournir un format concret et d'expliquer comment ce dernier peut se traduire en un format de modèles. Ce travail étant particulièrement difficile à réaliser, aucune proposition de format concret n'a encore réellement séduit les utilisateurs. C'est la raison pour laquelle AS n'est pas encore pleinement exploité, contrairement à OCL.

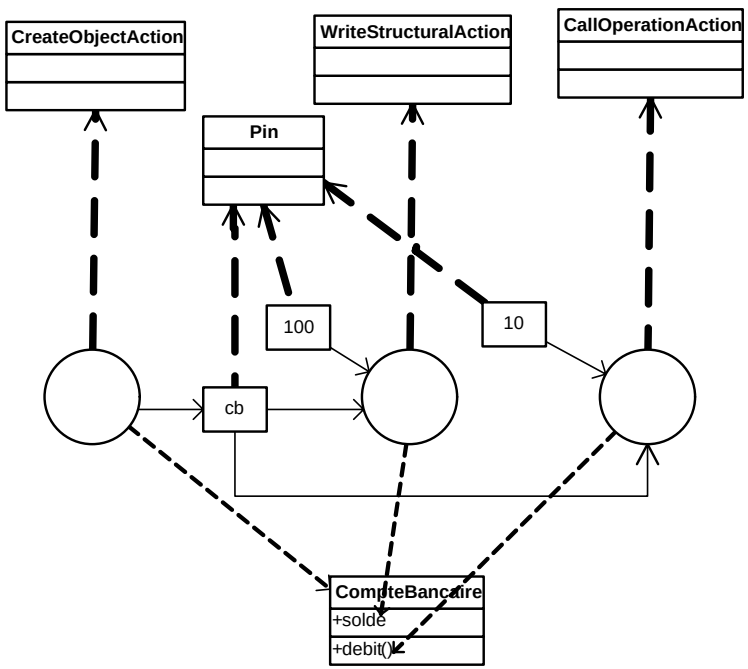
Afin de mieux faire comprendre l'importance d'un format concret, nous allons détailler un exemple simple d'activité. Cette activité contient trois actions : la création d'un compte bancaire, l'affectation d'un montant au solde et l'appel à une opération de débit. Nous proposons d'exprimer cette activité en utilisant une syntaxe proche du langage Java.

Cette activité se représenterait de la façon suivante :

```
CompteBancaire cb = new CompteBancaire() ;
cb.solde=100 ;
cb.debit(10) ;
```

La figure 4.14 donne une représentation de cet exemple sous forme de modèle. Nous avons choisi d'illustrer les actions sous forme de cercle et les flots de données sous forme de flèche. La figure fait apparaître les trois actions et les flots de données. Pour que le modèle soit complet, il faudrait faire apparaître l'activité qui définirait la séquence entre les actions. Cet exemple montre combien il est difficile de passer d'un format concret (ici textuel) à un format de modèle.

Figure 4.14
Les actions sous forme de modèle



En résumé

Nous avons vu que le langage AS permettait de spécifier des actions sur les modèles UML. Il est ainsi possible de spécifier le corps des opérations des classes à l'aide d'une suite d'actions (une activité). Les actions AS sont indépendantes des langages de programmation, l'objectif étant de traduire une suite d'actions AS vers un langage de programmation.

AS permet d'élaborer des modèles UML très précis, quasiment exécutables. Comme ces modèles UML sont toujours indépendants des plates-formes d'exécution, AS est utilisé pour l'élaboration des PIM.

Nous avons vu qu'AS était défini à l'aide d'un métamodèle. Ce métamodèle est entièrement intégré au métamodèle UML2.0. De ce fait, les actions AS sont fortement liées aux modèles UML, ce qui permet de rendre les modèles UML beaucoup plus pérennes et productifs.

Nous avons vu aussi qu'AS ne proposait pas de format concret de représentation. Il s'agit là d'une lacune assez gênante puisqu'il n'est pas possible pour un utilisateur de saisir facilement une suite d'instructions.

Tout comme OCL, AS est utilisé pour élaborer des transformations de modèles. Nous insisterons sur ce point à la partie II de l'ouvrage, qui traite des aspects productifs de MDA.

Synthèse

Même si leurs approches sont différentes, OCL et AS sont des standards relativement proches. Ils permettent tous deux de modéliser les corps des opérations UML.

L'approche OCL consiste à définir des contraintes sur les opérations afin de préciser ce que doit être la sortie d'une opération en fonction de son entrée. Plus proche des langages de programmation, l'approche AS consiste à définir une suite d'actions modifiant l'état d'un modèle. Les deux standards permettent d'élaborer des modèles UML beaucoup plus précis en définissant dans le détail les comportements des objets.

OCL et AS sont entièrement indépendants des plates-formes d'exécution. Grâce à ces standards, les modèles UML peuvent représenter intégralement la logique métier d'une application, et ce indépendamment des plates-formes d'exécution. On peut donc dire qu'ils offrent un gain significatif de pérennité aux modèles UML. Ils sont essentiellement utilisés pour l'élaboration des PIM dans MDA.

Les standards OCL et AS sont définis entièrement sous forme de métamodèles. Ces derniers étant intégralement liés au métamodèle UML, il est possible d'automatiser leurs traitements. Ils sont à ce titre pleinement inscrits dans MDA, même s'il est vrai qu'actuellement peu de produits commerciaux les supportent complètement.

5

Les modèles en XML

Au cours des chapitres précédents, les modèles n'ont été présentés que par le biais de leur métamodèle. Ainsi définis, ils restent des entités théoriques abstraites fortement volatiles et ne peuvent être ni stockés ni échangés sur un support informatique. Il est bien difficile dans ces conditions de parler de pérennité.

Le présent chapitre clôt la première partie de l'ouvrage traitant de la pérennité des modèles en introduisant les standards XMI (XML Metadata Interchange) et DI (Diagram Interchange) de l'OMG, qui permettent le support informatique des modèles pour le stockage et l'échange.

Pour ces standards, l'OMG a choisi de s'appuyer sur le format W3C XML, au succès indéniable et qui est désormais considéré comme le format international d'échange de données.

Il est aussi possible d'utiliser XML comme format de traitement des modèles afin de développer des opérations telles que les transformations de modèles. Nous aurions donc pu insérer XMI dans la partie II de l'ouvrage, consacrée à la productivité des modèles. Cependant, nous considérons que ces traitements sont plus idéalement développables en utilisant les langages objet.

Le format XML

Cette section vise non pas à présenter XML en détail mais plutôt à introduire les concepts importants de ce standard qui sont mis en œuvre dans les standards de l'OMG.

Le lecteur déjà familiarisé avec XML et maîtrisant les concepts de namespace, de schéma XML et de document SVG peut passer directement à la section consacrée à XMI.

Documents bien formés

Un document XML bien formé est un document textuel structuré par un ensemble de balises ouvertes et fermées. Les balises XML sont facilement identifiables car elles commencent par le caractère < et finissent par le caractère >. Entre ces deux caractères, une chaîne de caractères alphanumériques commençant obligatoirement par une lettre et ne contenant pas d'espaces représente le nom de la balise.

Une paire de balises ouvrante et fermante doit avoir le même nom. La balise fermante doit contenir le caractère / juste après le caractère <.

Voici un exemple de balises XML :

```
<Livre></Livre>
<MaBaliseNumero1></MaBaliseNumero1>
```

En plus de son nom, une balise ouvrante peut contenir un ensemble d'attributs. Un attribut est caractérisé par un nom et une valeur. Au sein d'une balise ouvrante, les attributs apparaissent après le nom de la balise et sont séparés par le caractère d'espacement. Le nom d'un attribut doit être une chaîne alphanumérique ne contenant pas d'espace. Sa valeur est une chaîne de caractères encadrée par des guillemets ou des apostrophes. Le nom et la valeur d'un attribut sont séparés par le caractère =.

Voici un exemple de balises XML avec des attributs :

```
<Livre numero='1'></Livre>
<MaBaliseNumero1 id="a1"></MaBaliseNumero1>
```

Entre une paire de balises ouvrante et fermante, un document XML bien formé peut contenir soit d'autres balises, soit à peu près n'importe quelle chaîne de caractères. Les caractères < et > ne doivent alors pas être utilisés, à moins d'être protégés.

Un document XML commence obligatoirement par une balise ouvrante. Il n'est pas possible d'imbriquer de façon croisée des balises ouvrantes et fermantes. Par exemple, il n'est pas possible d'ouvrir la balise A puis la balise B et de fermer la balise A avant de fermer la balise B. Il est en revanche parfaitement autorisé d'imbriquer les balises.

Voici un exemple de document XML bien formé :

```
<Livre titre='MDA'>
<Auteur prenom='xavier' nom='blanc'></Auteur>
<Chapitre numero='4' titre='Modèles en XML'>
Les chapitres précédents ont tous insisté sur l'importance de la pérennité dans les
➥modèles MDA…
</Chapitre>
</Livre
```

Le standard XML définit d'autres contraintes syntaxiques sur les documents XML bien formés, mais la présentation précédente suffit à notre démonstration.

Documents valides

Si un document bien formé ne doit respecter qu'un ensemble de contraintes syntaxiques, un document valide doit respecter une structure préétablie d'un ensemble de balises. Une telle structure préétablie peut, en reprenant l'exemple précédent, spécifier que la balise `Chapitre` soit obligatoirement incluse dans la balise `Livre`. Tout document respectant cette structure préétablie serait considéré comme valide.

Le standard XML propose des moyens pour définir ces structurations de balises, notamment *via* les DTD et XML Schema.

DTD (Document Type Definition)

Historiquement, le premier moyen proposé pour définir une structure de balises était la construction d'une DTD. Une DTD est un document textuel qui permet de spécifier une structuration d'un ensemble de balises en définissant leurs relations d'inclusion.

Le mot-clé `ELEMENT` permet de définir une balise. Il doit être suivi du nom de la balise et de l'ensemble des balises contenues. Le mot-clé `ATTLIST` permet de définir un ensemble d'attributs associés à une balise. Ce mot-clé doit être suivi du nom de la balise qui contiendra les attributs puis de la liste des attributs.

L'exemple suivant permet de spécifier que la balise `Livre` doit contenir la balise `Chapitre` et qu'elle doit avoir un attribut nommé `titre` (le mot-clé `CDATA` permet de spécifier que la valeur de l'attribut `titre` est une chaîne de caractères) :

```
<!ELEMENT Livre (Chapitre) >
<!ATTLIST Livre titre CDATA>
```

Pour être complet, cet exemple devrait bien évidemment spécifier d'autres balises, comme `Chapitre`, `Auteur`, etc.

Les DTD fournissent un moyen simple pour définir la structuration d'un ensemble de balises. Il existe d'ailleurs aujourd'hui plusieurs structurations de balises XML réalisées grâce à des DTD. Cependant, cette définition souffre de plusieurs lacunes.

Une DTD doit être réalisée à l'aide d'un format textuel qui n'est pas XML. Il n'est donc pas possible de manipuler les DTD comme nous manipulons des documents XML. De plus, l'expression des DTD est assez limitée. Il n'est pas possible de réutiliser facilement une structuration existante afin de l'enrichir ni de définir des ordres particuliers d'inclusion autres que la séquence.

Pour toutes ces raisons, le W3C a mis au point le standard XML Schema, qui permet de définir des structurations de balises XML en offrant davantage de possibilités.

XML Schema

Les schémas XML ont été standardisés par le W3C pour remplacer les DTD en raison de leur souplesse pour définir des structurations de balises XML.

Contrairement aux DTD, les schémas XML sont définis par des documents XML. Il est donc possible de les manipuler comme n'importe quels documents XML. Les schémas XML permettent en outre de définir des types de structuration. Il est possible de réutiliser ces types afin de les étendre ou de les restreindre. Ils s'appuient enfin sur un système de typage beaucoup plus complet que celui des DTD (composé uniquement du type `String`). Il est possible, par exemple, de dire que la valeur d'un attribut doit être un entier ou un booléen.

L'exemple suivant représente la définition de la balise `Livre` en XML Schema. Ce schéma XML spécifie que la balise `Livre` a un type complexe (`complexType`) et que ce type complexe est une séquence de `Chapitre` (`sequence`, `element`) :

```
<element name='Livre'>
   <complexType>
      <sequence>
         <element ref='Chapitre'></element>
      </sequence>
   </complexType>
</element>
```

Pour être complet, cet exemple devrait bien évidemment fournir la définition des autres balises (`Chapitre`, `Auteur`, etc.).

La définition d'une structuration de balises en XML Schema est toutefois beaucoup plus complexe qu'elle ne l'est en DTD. Cela explique en partie pourquoi il n'existe aujourd'hui que peu de structurations de balises XML réalisées en XML Schema.

Autres techniques XML

Étant donné que XML permet de définir facilement des ensembles de balises, il n'est pas rare de voir des balises ayant les mêmes noms alors qu'elles ont été définies par différents utilisateurs avec différentes significations. Nous pourrions, par exemple, imaginer que la balise `<adresse>` signifierait soit une adresse postale, soit une adresse électronique, soit, pourquoi pas ?, une adresse mémoire. Les balises aux noms très génériques, telles que `<contenu>`, `<extension>`, `<liste>` ou `<element>`, ont bien souvent des significations diverses.

Cette incertitude dans les significations est évidemment problématique si nous souhaitons utiliser ces balises dans un même document XML. Nous pourrions très bien imaginer un document XML utilisant plusieurs fois les balises `<liste>` et `<contenu>` mais avec des significations différentes. Il serait alors impossible de retrouver la signification réelle associée à chacune des balises.

Ce problème est très important pour l'utilisation de XML par MDA. En effet, les balises définies telles que `<Class>` et `<Package>` auront des significations différentes selon qu'elles seront associées à MOF ou UML ou selon les différentes versions de ces standards.

Pour faire face à ce problème de conflit de nommage, le W3C a défini la notion de *namespace.*

Namespace XML

Un namespace permet de regrouper un ensemble de balises et de préciser dans un document XML à quel namespace appartiennent les balises. Les namespace sont donc utilisés pour regrouper différentes balises ayant une signification commune.

Un namespace est défini simplement par une URL. Il est possible d'associer à cette URL une liste de balises, une DTD ou un schéma XML. Par exemple, l'OMG a défini l'URL en cours de discussion *http://org.omg/UML/1.3* pour le namespace des balises UML1.3.

Pour préciser à quel namespace appartient une balise, il suffit de procéder de la façon suivante :

1. Associer un alias au namespace. Cela se fait à l'aide de l'attribut `xmlns`, que nous pouvons associer à n'importe quelle balise ouvrante.
2. Préfixer les balises appartenant à ce namespace avec l'alias du namespace.

L'exemple suivant associe l'alias `uml13` au namespace des balises UML1.3 puis illustre l'utilisation de cet alias pour bien identifier les balises utilisées :

```
<document xmlns:uml13="http://org.omg/UML/1.3">
   <uml13:Class>
   </uml13:Class >
</document>
```

XSLT (eXtensible Stylesheet Language Transformations)

XML permettant de représenter d'une façon structurée n'importe quel type d'information, il est nécessaire de pouvoir transformer les documents XML afin de bénéficier des informations qu'ils contiennent.

Le W3C a proposé le standard XSLT (eXtensible Stylesheet Language Transformations) pour élaborer des transformations de documents XML. L'approche définie dans ce standard permet de spécifier des transformations de documents XML à l'aide de templates.

Un template est composé d'une clause d'applicabilité (clause `match`) et d'un ensemble d'instructions. Le principe de fonctionnement de XSLT consiste à parcourir un document XML donné en entrée d'une transformation afin de détecter quels templates peuvent y être appliqués. Dès qu'un template est détecté comme étant applicable, l'ensemble d'instructions qu'il définit est exécuté. Cet ensemble d'instructions a pour objectif de construire un nouveau document, en l'occurrence le document de sortie de la transformation.

Le standard XSLT est utilisé, par exemple, pour transformer les documents XML en documents HTML afin de rendre les documents XML présentables dans les navigateurs Web.

Voici, par exemple, une partie de la transformation XSLT permettant de générer un document HTML à partir d'un document XML représentant un livre. Cet exemple présente un template applicable aux balises `<Livre>`. La suite d'instructions de ce template permet de

construire les balises <HTML> et <HEAD> tout en insérant un texte précisant le titre du livre (grâce à l'instruction <value-of>) :

```
<template match="/Livre">
<HTML>
<HEAD>
Page correspondant au livre <value-of select="@titre"/>
</HEAD>
</template>
```

SVG (Scalable Vector Graphics)

SVG (Scalable Vector Graphics) est un langage XML standardisé par le W3C pour représenter des figures graphiques vectorielles sous forme de documents XML.

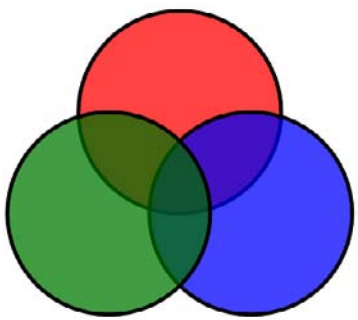
Un des intérêts de SVG est de pouvoir être utilisé avec XSLT pour transformer des documents XML en figures graphiques. Nous verrons que cela intéresse vivement MDA puisque cela permet de représenter les diagrammes graphiques des modèles sous forme de documents XML et ainsi de favoriser leur échange et leur pérennité.

L'exemple suivant est un document SVG représentant trois cercles (balise `circle`) de couleur *(voir figure 5.1) :*

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
              "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns="http://www.w3.org/2000/svg">
  <style type="text/css">
    circle:hover {fill-opacity:0.9;}
  </style>
  <g style="fill-opacity:0.7;">
    <circle cx="6.5cm" cy="2cm" r="100" style="fill:red; stroke:black;
    ➥stroke-width:0.1cm" transform="translate(0,50)" />
    <circle cx="6.5cm" cy="2cm" r="100" style="fill:blue; stroke:black;
    ➥stroke-width:0.1cm" transform="translate(70,150)" />
    <circle cx="6.5cm" cy="2cm" r="100" style="fill:green; stroke:black;
    ➥stroke-width:0.1cm" transform="translate(-70,150)"/>
  </g>
</svg>
```

Figure 5.1

Figure graphique correspondant au document SVG précédent



Nous verrons dans la suite du chapitre que SVG est utilisé pour représenter les parties graphiques, ou diagrammes, des modèles UML.

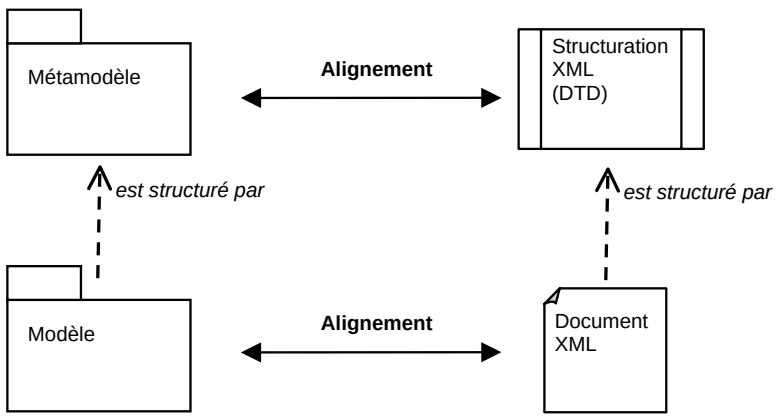
XMI (XML Metadata Interchange)

Les modèles étant des entités abstraites, ils ne disposent pas intrinsèquement de représentation informatique. L'OMG a donc décidé de standardiser XMI, qui offre une représentation concrète des modèles sous forme de documents XML. L'objectif premier de XMI est de définir un moyen permettant de représenter un modèle sous forme de document XML.

Afin de préciser comment faire cette représentation, l'OMG utilise les mécanismes de définition de structure de balises XML DTD et XML Schema. XMI permet en fait de définir les structurations de balises nécessaires et suffisantes à la représentation des modèles au format XML.

Pour définir ces structurations de balises, XMI s'appuie sur l'alignement existant entre les modèles et leur métamodèle, d'une part, et les documents XML et leur structuration, d'autre part. Métamodèles et structurations de balises sont semblables en ce qu'ils définissent les structures des modèles et des documents XML respectivement. La figure 5.2 illustre cet alignement.

Figure 5.2
Alignement entre métamodèle/modèle et DTD/document XML



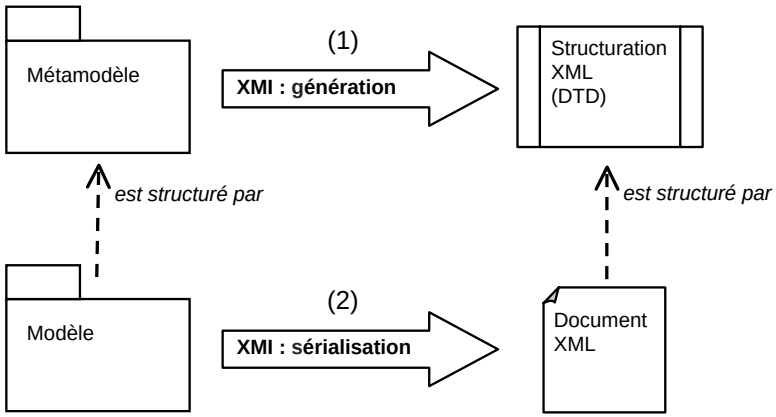
La structuration des balises XML permettant de représenter des modèles sous forme de document XML se fonde sur les métamodèles des modèles. Par exemple, la structuration des balises XML permettant de représenter des modèles UML sous forme de document XML s'appuie sur le métamodèle UML.

Règles de génération des balises XML

L'alignement des métamodèles et des structurations de balises XML permet à XMI de définir un ensemble de règles de génération automatique de structuration de balises XML à partir d'un métamodèle. Ce processus est indiqué en 1 à la figure 5.3.

Grâce à cette structuration de balises XML, il est possible de représenter les modèles sous forme de documents XML. Cette opération est appelée *sérialisation* du modèle au format XML. Indiquée en 2 à la figure 5.3, elle permet de bénéficier du mécanisme de validation proposé par XML et offre une représentation au format XML de meilleure qualité.

Figure 5.3
XMI et la structuration de balises XML



Ces règles de génération de la structuration de balises XML à partir d'un métamodèle sont assez naturelles. Nous présentons ci-après des règles similaires, bien que fortement simplifiées, qui permettent de saisir la philosophie de XMI. Ces règles acceptent en entrée un métamodèle MOF1.4 tel que nous l'avons présenté au chapitre 2. Nous numérotons ces règles afin de les identifier dans la suite du chapitre.

Règles XMI de génération de DTD à partir d'un métamodèle MOF1.4 :

1. Toute métaclasse fournit la définition d'une balise ayant comme nom le nom de la métaclasse. Cette balise doit posséder un attribut XML nommé `id` permettant d'identifier l'instance de la métaclasse afin de la référencer au besoin.

2. Tout méta-attribut d'une métaclasse fournit la définition d'une balise ayant comme nom le nom du méta-attribut. Le contenu de la balise représentera la valeur du méta-attribut. Cette balise doit être contenue dans la balise correspondant à la métaclasse contenant l'attribut.

3. Toute métaréférence d'une métaclasse fournit la définition d'une balise ayant comme nom le nom de la métaréférence. Le contenu de la balise représentera un `id` identifiant

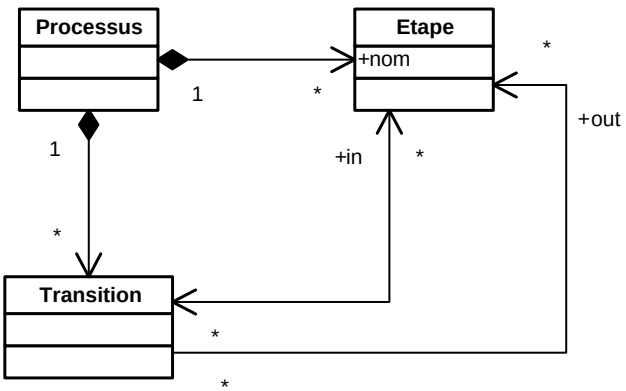
l'instance référencée. Cette balise doit être contenue dans la balise correspondant à la métaclasse contenant la référence.

4. Toute méta-association entre deux métaclasses fournit la définition d'une balise ayant comme nom le nom de la méta-association. Le contenu de la balise contiendra les instances reliées ou les `id` les identifiant. Si la méta-association est navigable, la balise de la méta-association doit être contenue dans la balise correspondant à la métaclasse source de l'association.

Exemple de mise en œuvre sur un métamodèle simple

Nous avons choisi d'appliquer les règles XMI à l'exemple de métamodèle présenté au chapitre 2. Rappelons que ce métamodèle est constitué de trois métaclasses et qu'il permet de modéliser des processus simples *(voir figure 5.4)*. Les métaclasses de ce méta-modèle sont incluses dans un métapackage nommé `MMProcessus`.

Figure 5.4
Métamodèle de processus



Si nous appliquons les quatre règles présentées à la section précédente, nous obtenons la DTD suivante :

```
<!ELEMENT Processus (ProcessusToTransition , ProcessusToEtape) >
<!ATTLIST Processus id ID> /* règle 1 et règle 4*/

<!ELEMENT Etape (nom) >
<!ATTLIST Etape id ID> /* règle 1 */

<!ELEMENT Transition (in, out) >
<!ATTLIST Transition id ID> /* règle 1 et règle 4*/

<!ELEMENT ProcessusToTransition #PCDATA> /* règle 4*/
<!ELEMENT ProcessusToEtape #PCDATA> /* règle 4*/

<!ELEMENT nom #PCDATA> /* règle 2 */

<!ELEMENT in #PCDATA> /* règle 4 */
<!ELEMENT out #PCDATA> /* règle 4 */
```

La règle 1 donne la définition des balises `Processus`, `Etape` et `Transition`. La règle 2 donne la définition de la balise `nom`. La règle 3 n'est pas utilisée. La règle 4 donne la définition des balises `ProcessusToTransition` et `ProcessusToEtape`, dont les noms ont été générés à partir des noms des métaclasses, et la définition des balises `in` et `out`.

Grâce à cette DTD, il est possible de représenter des modèles de processus conformes à ce métamodèle sous forme de documents XML. Par exemple, le modèle de processus composé uniquement de deux étapes nommées `début` et `fin` reliées par une transition est représenté par un document XML de la manière suivante :

```
<Processus id='p1'>
    <ProcessusToEtape>
        <Etape id='e1'>
            <nom>début</nom>
        </Etape>
        <Etape id='e2'>
            <nom>fin</nom>
        </Etape>
    </ProcessusToEtape>
    <ProcessusToTransition>
        <Transition>
            <in idref='e1'/>
            <out idref='e2'/>
        </Transition>
    </ProcessusToTransition>
</Processus
```

État actuel de XMI

Les sections précédentes de ce chapitre ont explicité le principe de fonctionnement de XMI, qui consiste à permettre la génération automatique de la définition de structuration de balises XML à partir d'un métamodèle. Nous avons montré que ce mécanisme permettait de représenter tout modèle au format XML.

Chaque nouvelle version de XMI améliore l'ensemble de ces règles afin de générer des définitions de structuration de balises toujours plus adéquates à la représentation des modèles sous forme de document XML.

La première version de XMI, XMI1.0, ne permettait que des générations relativement sommaires de DTD à partir de métamodèles MOF1.3. Cette version souffrait inévitablement des lacunes des DTD. Ces dernières ne disposant que d'un système de typage sommaire (uniquement des chaînes de caractères), il n'était pas possible de faire profiter pleinement les documents XML des types des métamodèles (système de typage complet).

Les DTD générées manquaient en outre de flexibilité quant aux possibilités d'ordonnancement des balises XML. Il fallait suivre un ordre préétabli totalement arbitraire, qui n'était pas imposé par le métamodèle. Il fallait, par exemple, pour les modèles UML, que les cas d'utilisation apparaissent avant les classes dans les documents XML.

De plus, le nom associé à la balise XML correspondant à chaque métaclasse d'un métamodèle devait intégrer entièrement les noms de la hiérarchie des packages contenant la métaclasse. La balise correspondant à la métaclasse `UseCase` s'appelait alors `Behavioral_Elements.Use_Cases.UseCase` et celle correspondant au nom du cas d'utilisation s'appelait `Foundation.Core.ModelElement.name`, car le nom d'un cas d'utilisation était hérité de la métaclasse `ModelElement`. De ce fait, les documents XML représentant des modèles étaient particulièrement verbeux et ne contenaient que peu d'information utile.

Fondée sur les métamodèles MOF1.4, la deuxième version de XMI, XMI1.1, améliore considérablement l'ensemble des règles XMI1.0. Outre le gain notable de flexibilité qu'elle apporte à l'ordonnancement des balises XMI, elle bénéficie de la standardisation du mécanisme de namespace XML *(voir la section précédente)*, qui permet de réduire considérablement la taille des noms des balises en leur associant un même namespace.

XMI1.1 définit un mécanisme permettant de représenter des sous-parties de modèles sous forme de documents XML. Ce mécanisme est d'une grande utilité pour effectuer, par exemple, des mises à jour de modèles et ne transmettre que les sous-parties des modèles ayant changé. Malgré toutes ces évolutions, cette version souffre inévitablement des lacunes des DTD quant au typage des données.

La troisième version de XMI, XMI1.2, n'offre pas d'avancée majeure mais corrige l'ensemble des petites erreurs de la version 1.1. Remarquablement stable, cette version est actuellement en cours de standardisation à l'ISO (International Standardization Organization).

Une extension de XMI1.2 permet de bénéficier des schémas XML. L'objectif de cette extension est de générer non plus des DTD mais des schémas XML à partir des métamodèles. L'apport de XML Schema à XMI est très important en ce qu'il fournit des mécanismes de typage très intéressants *(voir la section précédente)*. Grâce à cette extension, XMI n'a plus à souffrir des lacunes des DTD.

La version XMI2.0 s'appuie non pas sur les métamodèles MOF2.0, contrairement à ce que laisse supposer son nom, mais sur ceux de MOF1.4. Cette version reprend les règles de l'extension à XMI1.2 permettant de générer des schémas XML et les améliore. Ce standard propose aussi de construire automatiquement un métamodèle à partir d'un schéma XML. Cette approche n'est toutefois applicable pour l'instant qu'à des schémas XML respectant des contraintes fortes.

Il faudra attendre la version XMI2.1, en cours d'élaboration, pour retrouver les métamodèles MOF2.0 et permettre la génération de schémas XML.

Le problème de l'échange de modèles UML entre outils

Lorsque la première version de XMI a été publiée, ce standard était censé permettre à tous les outils du marché de s'échanger des modèles UML. C'est du moins ce qu'avait annoncé l'OMG, et cela ne semblait pas présenter de difficulté majeure. Plus de cinq ans

après la publication de ce standard, force est de constater qu'il n'est toujours pas possible d'échanger des modèles UML entre les outils du marché en utilisant XMI.

La première raison de cet échec vient de la multitude des versions des standards entrant en jeu pour l'échange des modèles UML entre outils. La sérialisation d'un modèle UML sous forme de document XML dépend en effet des versions de XMI, de MOF et d'UML. Or il existe quatre versions de XMI, quatre d'UML et trois de MOF. Étant donné que chaque version de XMI est fortement liée à une unique version de MOF, seules les versions de XMI et d'UML entrent en considération pour l'échange des modèles UML.

Le tableau 5.1 montre les différentes combinaisons permettant la représentation des modèles UML sous forme de document XML.

Tableau 5.1 Combinaisons possibles entre XMI et UML

	XMI1.0 DTD	XMI1.1 DTD	XMI1.2 DTD	XMI1.2 Schema	XMI2.0 Schema	XMI2.1 Schema
UML1.3	X	X	X	X		
UML1.4	X	X	X	X	X	
UML1.5	X	X	X	X	X	
UML2.0						X

Il est possible d'échanger les modèles UML1.3, UML1.4 et UML1.5 selon quatre versions différentes de XMI. Comme il n'existe aucune recommandation officielle de l'OMG préconisant une combinaison plutôt qu'une autre, chaque outil choisit une ou plusieurs possibilités. De ce fait, la possibilité d'échange de modèles UML entre outils est tout sauf évidente.

Le tableau 5.1 montre aussi qu'il ne sera possible d'échanger les modèles UML2.0 que selon une seule version de XMI. L'échange des modèles UML2.0 devrait donc être grandement facilité.

La seconde raison de l'échec vient de la façon dont fonctionne XMI à l'égard de la flexibilité d'ordonnancement des balises XML. Chaque nouvelle version de XMI offre davantage de flexibilité dans l'ordonnancement des balises XML. Cette flexibilité est intéressante puisqu'elle offre plus de choix de représentation des modèles UML sous forme de documents XML. En contrepartie, elle présente l'inconvénient majeur d'accroître considérablement la complexité de lecture des documents XML représentant les modèles UML.

Pour pouvoir lire convenablement un document XML représentant un modèle UML, il faut être capable de lire toutes les possibilités d'ordonnancement. Cet ensemble d'ordonnancements est très important dans le cas d'UML. Cela explique que certains outils ne sont même pas capables d'importer des modèles UML qu'ils ont pourtant exportés eux-mêmes sous forme de documents XML.

Cette seconde raison reste d'actualité avec UML2.0. Il semble cependant que les outils de manipulation des documents XML soient plus faciles d'emploi et plus puissants et

qu'ils permettent de mieux gérer cette complexité. Malheureusement, nous ne pourrons le vérifier que lorsque seront disponibles sur le marché un grand nombre d'outils supportant UML2.0.

En résumé

Le standard XMI permet de représenter n'importe quel modèle sous forme de document XML. Le principe de fonctionnement de XMI consiste à générer automatiquement une spécification de structuration de balises XML (DTD ou XML Schema) à partir d'un métamodèle. Il est ainsi possible de bénéficier du mécanisme de validation des documents XML.

XMI a été employé sur le métamodèle UML pour permettre l'échange des modèles UML. Cela n'a pas eu le succès escompté, et il demeure aujourd'hui impossible d'échanger des modèles UML entre les différents outils du marché. Nous avons vu que les raisons de cet échec tenaient principalement à un problème de maturité, du fait du nombre important de versions des standards entrant en jeu. Le principe de fonctionnement de XMI n'est donc pas remis en question.

Actuellement, tout nouveau standard OMG qui propose un métamodèle doit proposer la DTD ou le schéma XML correspondant. XMI est donc considéré comme le standard assurant l'interopérabilité minimale pour l'échange des modèles entre outils.

Les spécifications de structuration de balises XML générées par XMI à partir de métamodèles ne remplacent pas les métamodèles. Toutes les informations contenues dans les métamodèles n'apparaissent pas intégralement dans ces spécifications de structuration de balises. Par exemple, nous avons vu au chapitre 1 qu'un attribut d'une métaclasse pouvait être un attribut dérivé. Cette notion n'est pas reproductible dans le monde XML. De fait, la validation XML ne peut remplacer la validation d'un modèle par son métamodèle.

DI (Diagram Interchange)

Il n'est pas tout à fait exact de dire que le standard XMI permet de représenter les modèles sous forme de document XML. XMI ne permet de représenter sous forme de document XML que les informations dont la structure est définie dans un métamodèle. Autrement dit, les informations dont la structure n'est pas définie dans un métamodèle ne peuvent être représentées sous forme de document XML selon les règles du standard XMI.

Nous avons vu que, dans UML, la partie graphique des modèles, c'est-à-dire les diagrammes, n'était pas définie dans le métamodèle UML. De ce fait, il n'est pas possible de représenter les diagrammes UML dans les documents XML.

Cela ne va pas sans poser problème. Même s'il est envisageable de redessiner un diagramme de classes, il est beaucoup plus problématique de redessiner des diagrammes d'états ou de séquences, par exemple. De plus, un modèle UML obtenu à partir d'un document XML ne permet pas de savoir combien de diagrammes il faut dessiner. Dire qu'il faut un

diagramme par package est trop simpliste et trop arbitraire. Le nombre des diagrammes et leur contenu sont pourtant d'une importance capitale pour l'échange d'information.

Pour faire face à cette difficulté, l'OMG a défini un standard spécifique sous le nom de DI (Diagram Interchange). Ce standard a pour objectif de permettre la représentation au format XML des parties graphiques des modèles UML.

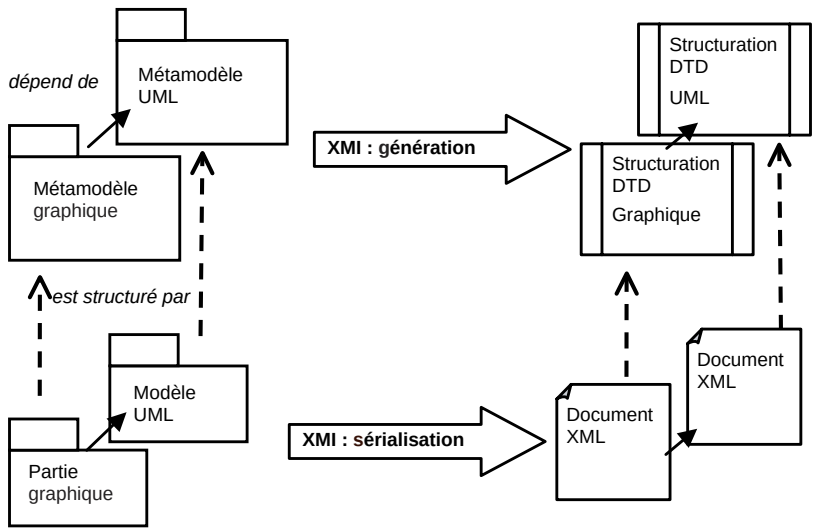
Principe de fonctionnement de DI

L'approche définie par DI pour permettre la représentation au format XML des parties graphiques des modèles UML s'appuie fortement sur XMI. L'idée est de définir un nouveau métamodèle, lié au métamodèle UML, représentant sous forme de métaclasses les idiomes graphiques nécessaires et suffisants à la représentation de toutes les parties graphiques d'UML.

L'objectif est d'appliquer XMI à ce métamodèle afin d'obtenir la spécification de structuration des balises XML permettant de représenter ces parties graphiques en XML.

La figure 5.5 illustre ce principe.

Figure 5.5
Principe de fonctionnement de DI



En plus de ce principe de fonctionnement, DI définit une transformation XML permettant de générer automatiquement des documents SVG à partir des documents XML représentant les modèles UML et leurs parties graphiques. Cette génération permet de rendre visible les modèles UML dans n'importe quel outil supportant le format SVG. Cette génération étant standard, il est de plus possible d'échanger complètement des modèles UML, parties graphiques comprises.

La figure 5.6 illustre l'utilisation de SVG dans le standard DI.

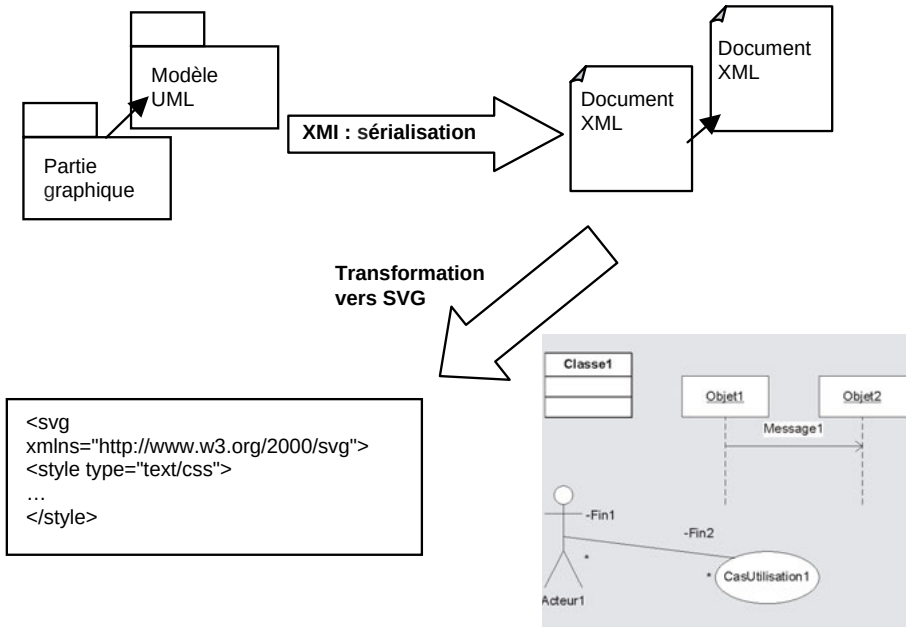


Figure 5.6
Utilisation du standard SVG dans DI

Nous pouvons nous représenter DI, d'une part, comme l'application du standard XMI à un nouveau métamodèle représentant la partie graphique d'UML et, d'autre part, comme la génération automatique d'un document SVG affichable graphiquement dans les outils du marché.

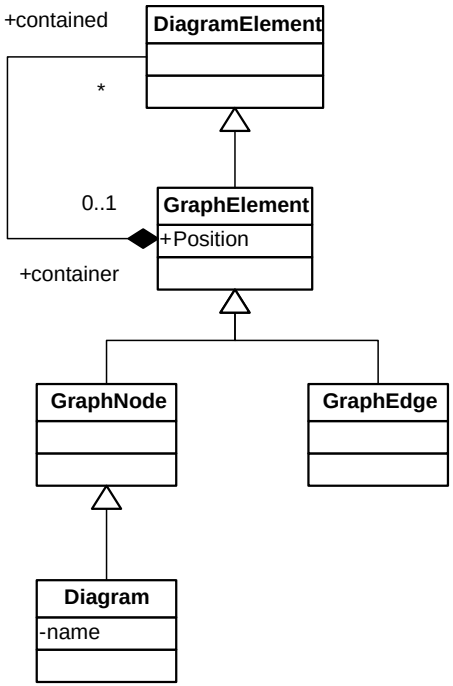
Le métamodèle représentant la partie graphique est d'assez bas niveau. Plutôt que de définir des métaclasses relatives aux différents diagrammes UML (diagrammes de classes, de séquences, de composants, de déploiement, etc.), il définit les métaclasses permettant de représenter n'importe quel élément graphique sous forme vectorielle, c'est-à-dire essentiellement sous forme de nœuds connectés par des arcs.

Ce métamodèle est relativement simple. Les métaclasses principales qu'il définit sont les métaclasses `GraphNode` et `GraphEdge`. Grâce à ces deux métaclasses, il est possible de représenter n'importe quel élément graphique. Ces deux métaclasses héritent de la métaclasse `GraphElement`, qui permet de représenter n'importe quel élément graphique. Cette métaclasse a d'ailleurs un méta-attribut nommé `Position` permettant de localiser l'élément sur une grille 2D. Cette métaclasse hérite de la métaclasse `DiagramElement` et lui est reliée par une méta-association d'agrégation. Cela permet d'exprimer qu'un élément graphique peut être un élément complexe s'il est défini par la composition de plusieurs autres

éléments graphiques. Le métamodèle définit la métaclasse `Diagram`, qui hérite de la métaclasse `GraphNode` et représente un diagramme graphique.

La figure 5.7 illustre ces différentes métaclasses.

Figure 5.7
Métaclasses principales du métamodèle DI

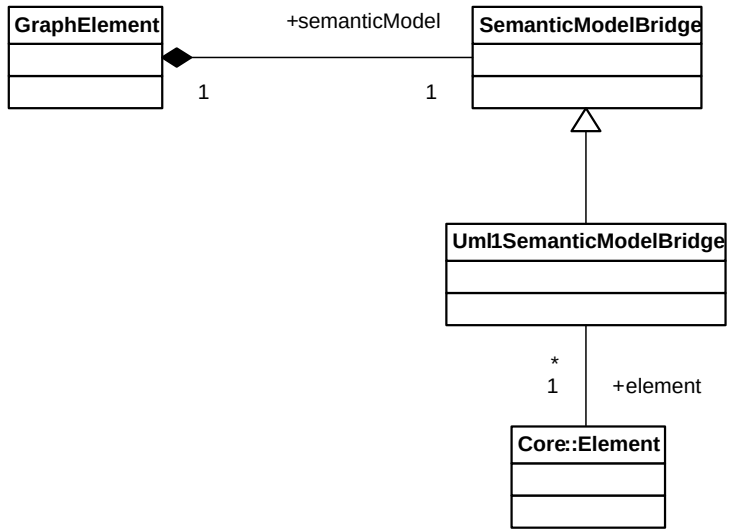


La dépendance entre le métamodèle DI et le métamodèle UML s'effectue par le biais des métaclasses `GraphElement` et `SemanticModelBridge`. La métaclasse `GraphElement` est reliée à la métaclasse `SemanticModelBridge` par une méta-association d'agrégation. Cela permet d'exprimer une dépendance sémantique entre un élément graphique et un élément d'un modèle. La métaclasse `SemanticModelBridge` est héritée par la métaclasse `Uml1SemanticModelBridge`, qui est reliée à la métaclasse `Core::Element` du métamodèle UML1.4 représentant n'importe quel élément du modèle UML.

Grâce à ces liens, il est possible de spécifier qu'un élément graphique (`GraphElement`) est le représentant graphique d'un élément sémantique d'un modèle UML (`Core::Element`). Dans la version actuelle de DI, cette relation est uniquement possible entre les éléments graphiques et les éléments des modèles UML1.4. Il n'est donc pas encore possible d'utiliser DI pour les modèles UML2.0. Cependant, étant donné que le métamodèle UML2.0 définit lui aussi la métaclasse `Element`, il n'y aura que peu d'améliorations à apporter à DI pour qu'il supporte UML2.0.

La figure 5.8 illustre les différentes métaclasses qui permettent une association sémantique entre un élément graphique et un élément d'un modèle UML1.4.

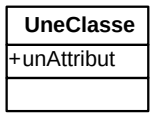
Figure 5.8
Métaclasses permettant d'exprimer un lien sémantique entre un élément graphique et un élément de modèle UML1.4



Mise en œuvre dans un exemple

Pour terminer cette présentation du standard DI, nous allons illustrer sa mise en œuvre à travers l'exemple de la figure 5.9. Cet exemple relativement simple n'est constitué que d'un diagramme de classes ne contenant qu'une seule classe.

Figure 5.9
Diagramme UML à sérialiser au format XML grâce à XMI et DI



La figure 5.10 illustre le modèle conforme au métamodèle DI représentant la partie graphique du modèle UML de la figure 5.9. Ce modèle étant relativement complexe, nous avons choisi de le présenter en utilisant une notation semblable à celle des diagrammes d'instances UML. Chaque instance d'une métaclasse y est représentée par un carré. Le texte associé au carré est constitué du nom de l'instance suivi du nom de sa métaclasse, les deux noms étant séparés par le caractère deux-points. Les liens entre les carrés représentent les liens entre les instances.

Ce modèle est composé de huit instances. L'instance `diag1` de la métaclasse `Diagram` représente le diagramme graphique. L'instance `class` de la métaclasse `GraphNode` représente la partie graphique de la classe `UneClass`. Les instances `operationCompartment`, `attributeCompartment` et `nameCompartment` représentent respectivement les parties graphiques de la classe `UneClasse` correspondant aux opérations, aux attributs et au nom (leur métaclasse est la métaclasse `CompartmentGraphNode`). C'est pourquoi ces instances sont liées à l'instance `class`. L'instance `attributeCompartment` est liée à une instance `attribut`, qui est une instance de la métaclasse `GraphNode`. Cette instance incarne la représentation graphique de l'attribut.

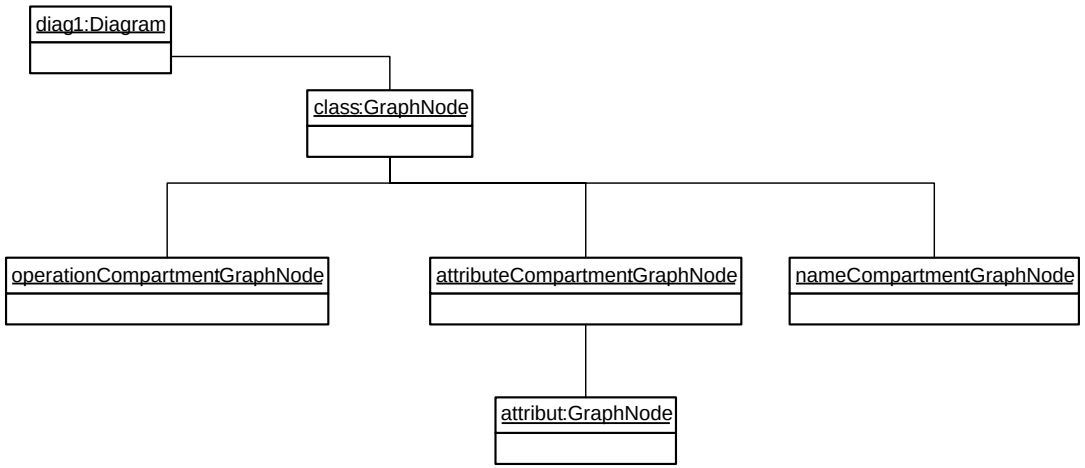
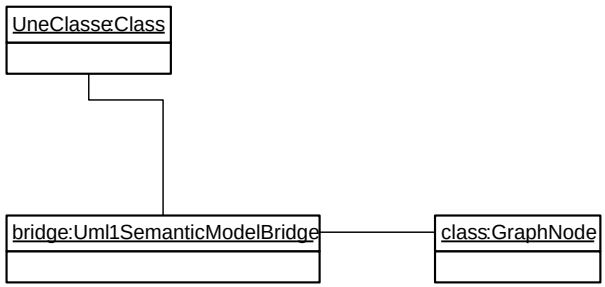


Figure 5.10
Représentation du modèle de la partie graphique

À ce stade du modèle, il faut ajouter les instances permettant de faire le lien sémantique avec le modèle UML. Deux liens doivent être établis dans cet exemple, l'un vers la classe UML et l'autre vers l'attribut, car seuls ces deux éléments ont une existence dans le métamodèle UML.

La figure 5.11 illustre le lien sémantique entre la classe UML et sa partie graphique. Nous constatons que l'instance de la métaclasse `Class`, qui représente la sémantique de la classe UML, est liée à l'instance `bridge` de la métaclasse `Uml1SemanticModelBridge`, elle-même liée à l'instance `class` illustrée à la figure 5.10. Ces instances représentent la modélisation du lien entre la partie graphique du modèle UML et sa partie sémantique.

Figure 5.11
Liens sémantiques entre DI et UML



Nous avons volontairement masqué une grande partie de ce modèle, qui est en réalité beaucoup plus complexe. Il conviendrait en effet d'ajouter toutes les instances permettant de modéliser les quatre traits qui forment le carré graphique de la classe ainsi que les instances permettant de représenter toutes les chaînes de caractères, etc. Tel qu'il est représenté ici, cet exemple est cependant suffisant pour comprendre l'approche DI.

Le modèle illustré en partie aux figures 5.10 et 5.11 peut donc être sérialisé en un document XMI. En tant que représentation graphique et sémantique d'un modèle UML, ce document XMI garantit une totale pérennité du modèle UML. De plus, il est possible d'y appliquer la génération d'un document SVG afin de visualiser la partie graphique du modèle dans un outil supportant le standard SVG.

En résumé

Le standard DI permet de représenter les parties graphiques des modèles UML sous forme de documents XML. L'approche définie par DI est très intéressante en ce qu'elle réutilise l'approche XMI en définissant un métamodèle graphique et en appliquant la génération de spécifications de structuration de balises.

Le métamodèle DI étant de très bas niveau, il permet de modéliser n'importe quel élément graphique. Actuellement, ce métamodèle est lié au métamodèle UML1.4. Cependant, ce lien ne concernant que deux métaclasses, il ne devrait pas être trop difficile de le faire évoluer pour le positionner en regard du métamodèle UML2.0 ou d'un autre métamodèle.

Le standard DI n'est pas encore totalement déployé dans les différents outils du marché, et ceux qui l'implémentent sont encore peu nombreux. Cette carence du marché s'explique principalement par l'état actuel du support de XMI. En effet, il n'est pas possible de supporter DI sans supporter XMI. Comme XMI2.1 devrait être plus largement déployé dans les outils du marché, il est légitime d'espérer que DI suivra.

Ajoutons que la plupart des éditeurs d'outils UML ont salué la qualité du standard DI lors de sa publication par l'OMG. Nous pouvons donc raisonnablement escompter un large support de ce standard dans un futur proche.

Synthèse

Ce chapitre a présenté les standards XMI et DI, grâce auxquels il est possible de représenter les modèles au format XML. Les modèles gagnent ainsi un support informatique largement répandu, gage de leur pérennité.

Même si les premières versions du standard XMI n'ont guère rencontré un franc succès pour la pérennité des modèles UML, nous avons vu que cela provenait principalement du manque de maturité du standard. La version XMI2.1 devrait connaître une meilleure fortune.

Le standard DI, qui reprend l'approche définie dans le standard XMI, n'est pas encore réellement déployé. La généricité des concepts qu'il propose devrait cependant le voir rapidement adopté par les outils qui implémenteront XMI2.1.

Malgré des erreurs de jeunesse, XMI reste le standard mis en avant par l'OMG pour résoudre tous les problèmes d'interopérabilité. L'OMG préconise en effet son utilisation exclusive pour l'échange des modèles. Chaque standard OMG qui propose un nouveau métamodèle doit donc fournir la spécification de structuration de balises correspondante générée par XMI.

Partie II

Gains de productivité (frameworks et outils)

MDA est très innovant en ce qu'il considère les modèles non plus comme des éléments contemplatifs mais comme des éléments productifs. Les modèles sont ainsi au centre du cycle de vie de développement et d'évolution des applications informatiques. La productivité des modèles renforce considérablement leur pérennité car elle les rend indispensables. Sans productivité des modèles, MDA n'est qu'une approche théorique, incapable de fournir un moindre gain, tant qualitatif que quantitatif.

Les opérations de production sur les modèles les plus connues et les plus demandées aujourd'hui sont principalement des opérations de génération de code et de génération de documentation. L'objectif est d'assurer une cohérence entre code et documentation en leur donnant une base commune. L'automatisation de ces opérations offre un gain considérable de productivité et de qualité.

Les opérations de production sur les modèles de demain devraient être des opérations de transformation, de simulation et d'exécution de modèles. L'objectif est de bénéficier des avantages des modèles à toutes les étapes du cycle de vie des applications. Ces opérations étant particulièrement complexes, elles nécessitent la définition de nouveaux mécanismes et solutions.

La mise en place d'opérations de production sur les modèles nécessite la définition et la construction d'une ingénierie. Cette partie se penche sur tous les aspects de cette ingénierie, depuis ses bases jusqu'aux concepts les plus avancés en passant par son implémentation actuelle dans deux outils du marché, Rational Software Modeler et Softeam MDA Modeler.

6

Manipuler des modèles avec JMI et EMF

Ce chapitre présente les différents moyens de manipuler des modèles avec les langages de programmation orientée objet. L'objectif est de montrer comment il est possible d'utiliser les langages de programmation objet pour coder des opérations sur les modèles, telles que la génération de code et de documentation ou la transformation de modèles.

Nous avons vu à la partie précédente qu'il était possible de représenter les modèles sous forme de documents XMI. Cette représentation XMI est certes très intéressante pour stocker les modèles ou pour faire des échanges entre outils, mais elle n'est pas vraiment adaptée au développement d'opérations sur les modèles.

La représentation XMI souffre encore des lacunes du standard XMI *(voir le chapitre 5)*. La manipulation des documents XMI est de surcroît délicate, ce qui rend ardu le développement d'opérations sur les modèles, et les mécanismes de production XML ne sont pas pleinement adaptés à toutes les opérations sur les modèles que nous avons déjà identifiées (génération de code, de documentation, etc.).

Un autre format de représentation des modèles permettant leur manipulation dans les langages de programmation orientée objet était donc nécessaire pour développer les opérations sur les modèles de la même manière que pour développer n'importe quelle application informatique.

L'OMG a proposé un format de représentation permettant la manipulation des modèles dans les langages de programmation orientée objet. Ce format était défini initialement dans le standard MOF1.3 et utilisait le langage de définition d'interface CORBA/IDL. Actuellement, c'est le standard MOF2.0 to IDL qui définit ce format. De son côté, le JCP (Java Community Process) a défini le standard JMI (Java Metadata Interface), une API Java

permettant la manipulation des modèles. Enfin, le framework EMF (Eclipse Modeling Framework) a été conçu pour la manipulation des modèles dans l'environnement ouvert Eclipse.

Quel que soit le standard ou le framework (MOF, JMI ou EMF), l'approche est sensiblement la même. L'idée est de fournir un ensemble d'interfaces offrant les opérations nécessaires à la manipulation des modèles. Le développement d'une opération sur les modèles consiste simplement à développer une application utilisant ces interfaces de manipulation des modèles.

Les concepts clés de la manipulation des modèles

Les interfaces de manipulation de modèles proposées par ces standards sont de deux types : les interfaces dites *taylored*, c'est-à-dire taillées sur mesure pour un métamodèle donné, et les interfaces dites réflectives, qui permettent l'accès au niveau du métamodèle.

Les interfaces taylored sont parfaitement adaptées à la manipulation d'un type de modèle, c'est-à-dire à un ensemble de modèles instances d'un même métamodèle. Par exemple, les interfaces taylored pour UML2.0 offriront les opérations permettant d'obtenir les attributs d'une classe ou les connexions entre composants UML. On parlera alors d'interface taylored pour le métamodèle UML2.0.

Les interfaces réflectives sont utilisables sur tous types de modèles, les opérations qu'elles proposent étant totalement indépendantes de la structure des modèles. Le point fort de ces interfaces est qu'elles permettent d'obtenir des informations sur le métamodèle d'un modèle et ainsi de connaître dynamiquement la structure du modèle. Les sections suivantes présentent plus en détail ces deux sortes d'interfaces de manipulation des modèles.

Les interfaces taylored

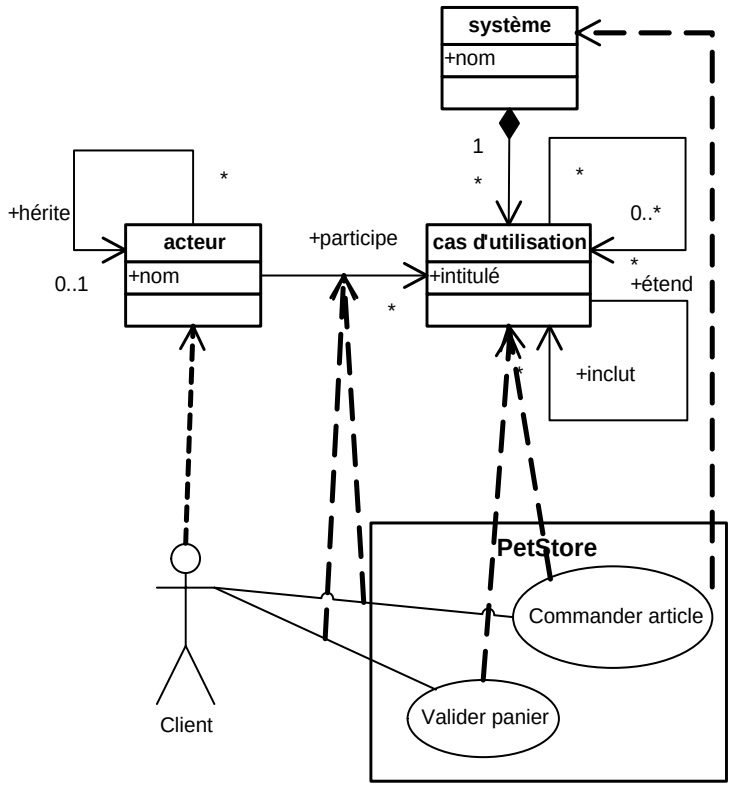
Comme expliqué précédemment, les interfaces taylored sont adaptées à un unique type de modèle. Elles offrent des opérations spécifiques permettant une navigation dans les modèles instances d'un unique métamodèle.

Les interfaces taylored relatives au métamodèle représentant la structure des diagrammes de cas d'utilisation *(voir partie haute de la figure 6.1)* permettent d'effectuer les opérations suivantes :

- Connaître le nombre d'acteurs contenus dans un modèle, ajouter ou supprimer des acteurs, connaître le nom d'un acteur et le modifier, connaître les liens d'héritage entre acteurs et les modifier et connaître et modifier les cas d'utilisation dans lesquels participe un acteur.
- Connaître le nombre de cas d'utilisation contenus dans un modèle, ajouter ou supprimer des cas d'utilisation, connaître l'intitulé d'un cas d'utilisation et le modifier, connaître les cas d'utilisation étendus ou inclus dans un cas d'utilisation et les modifier.

- Connaître le nombre de systèmes contenus dans un modèle, ajouter ou supprimer des systèmes, connaître le nom d'un système et le modifier et connaître l'ensemble des cas d'utilisation contenus dans un système et les modifier.

Figure 6.1
Métamodèle représentant les diagrammes de cas d'utilisation associés à un exemple de modèle



L'ensemble des opérations de ces interfaces et la façon dont nous les avons présentées laisse déjà entrevoir ce que nous développerons dans les sections suivantes de ce chapitre, à savoir que les interfaces taylored sont fortement liées aux métaclasses du métamodèle.

Appliquées au modèle exemple de la partie basse de la figure 6.1, ces interfaces permettraient de savoir que le modèle est constitué d'un système et d'un acteur relié à deux cas d'utilisation.

Les interfaces réflectives

Contrairement aux interfaces taylored, les interfaces réflectives sont utilisables sur n'importe quels modèles, quel que soit leur métamodèle. L'idée sous-jacente est de considérer tous les modèles comme des ensembles d'éléments (instances de métaclasse) reliés entre eux. Par exemple, le modèle exemple de la figure 6.1 est composé de quatre éléments, deux

instances de la métaclasse `cas d'utilisation`, une instance de la métaclasse `acteur` et une instance de la métaclasse `système`, reliés entre eux.

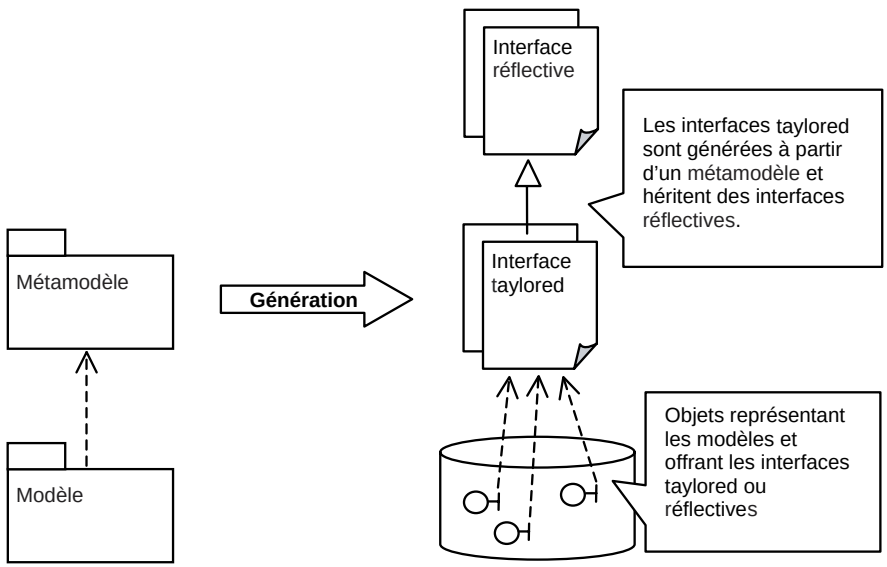
L'intérêt des interfaces réflectives est qu'elles offrent des moyens d'accéder aux métaclasses des métamodèles. À partir d'un élément d'un modèle, il est possible d'accéder à sa métaclasse et ainsi d'obtenir toutes les informations qui le structurent (attributs, références, etc.). Par exemple, sur le modèle de la figure 6.1, à partir de l'élément correspondant au système, il est possible d'accéder à la métaclasse `système` et ainsi de savoir que cette métaclasse a un méta-atttibut nommé `nom` et des métaréférences vers des éléments instances de la métaclasse `cas d'utilisation`. Grâce à ces informations, nous pouvons connaître le nom du système ainsi qu'obtenir les cas d'utilisation qui le composent.

Les interfaces réflectives ont ceci de très intéressant qu'elles permettent de développer sur les modèles des opérations indépendantes des métamodèles. De telles opérations peuvent être, par exemple, des opérations de génération de documentation ou de sauvegarde. Leur inconvénient est bien entendu leur grande complexité, qui rend difficile l'élaboration des opérations sur les modèles.

En résumé

L'approche que suivent les standards et framework MOF, JMI et EMF est sensiblement la même. Ils définissent des règles de génération d'interfaces taylored à partir de métamodèles et définissent des interfaces réflectives permettant la manipulation de tout type de modèle. Tous ces standards font en sorte que les interfaces taylored héritent des interfaces réflectives. Cela permet d'utiliser les capacités de navigation dans les métamodèles des interfaces réflectives à partir des interfaces taylored.

Figure 6.2
Les interfaces taylored et réflectives



En suivant ce principe, les modèles sont représentés par des ensembles d'objets qui présentent les interfaces taylored et réflectives. Ce principe est illustré à la figure 6.2.

Les sections qui suivent se penchent en détail sur le standard JMI et le framework EMF, les deux approches les plus utilisées et les plus outillées.

JMI (Java Metadata Interface)

JMI est un standard du JCP (Java Community Process) de Sun, qui définit un moyen de représenter les modèles sous forme d'objets Java.

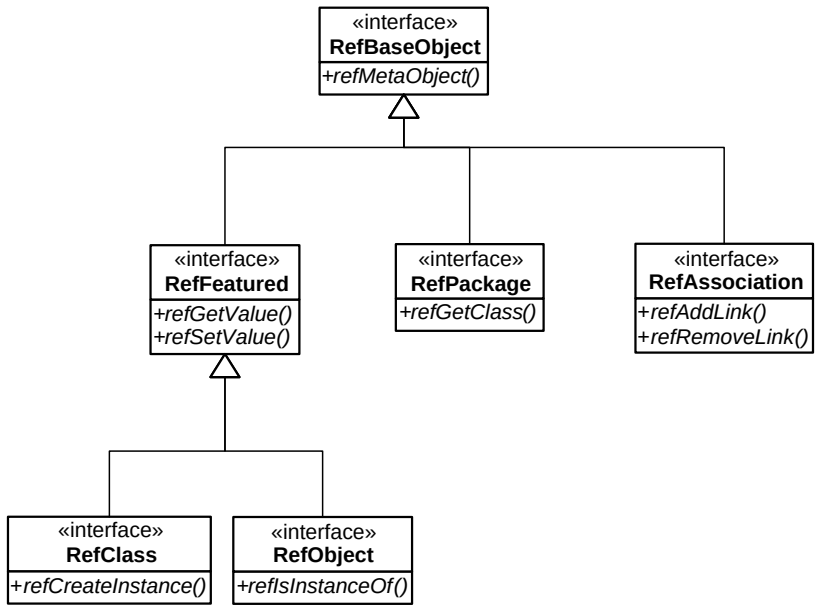
L'approche JMI suit celle que nous avons présentée à la section précédente. L'idée est de générer des interfaces taylored à partir d'un métamodèle et de faire en sorte que ces interfaces héritent des interfaces réflectives.

Les interfaces réflectives de JMI

Comme expliqué précédemment, les interfaces réflectives ne sont pas dédiées à la manipulation d'un unique type de modèle et ne dépendent pas d'un métamodèle particulier.

La particularité de ces interfaces est de permettre de naviguer dans les niveaux méta. Il est possible, à partir d'un élément de modèle, de connaître sa métaclasse et ainsi de savoir dynamiquement comment celui-ci est structuré. La figure 6.3 illustre toutes les interfaces réflectives de JMI.

Figure 6.3
Les interfaces réflectives de JMI



RefBaseObject

L'interface réflective la plus importante de JMI est `RefBaseObject`. Celle-ci représente indifféremment n'importe quel élément, qu'il fasse partie d'un modèle ou d'un métamodèle.

Cette interface offre l'opération `refMetaObject()`, qui retourne la métaclasse de l'élément. La métaclasse retournée est de type `RefObject`. Cela permet d'obtenir la métaclasse d'une métaclasse, ou métamétaclasse, et ainsi de remonter tous les niveaux méta jusqu'aux métaclasses de MOF, lesquelles s'autodéfinissent et s'autoréférencent.

RefFeatured

L'interface `RefFeatured`, qui hérite de l'interface `RefBaseObject`, contient toutes les opérations permettant d'accéder aux propriétés d'un élément (attributs, référence et opération).

Cette interface offre les opérations `refGetValue` et `refSetValue` permettant respectivement de lire et d'écrire les valeurs d'une propriété. Ces opérations prennent en paramètre une chaîne de caractères permettant d'identifier la propriété.

RefClass

L'interface `RefClass`, qui hérite de l'interface `RefFeatured`, représente la notion de *factory* à éléments. Une factory est une sorte d'usine, qui permet de construire les instances des métaclasses.

Cette interface offre donc l'opération `refCreateInstance`, qui permet de créer des instances d'une métaclasse.

RefObject

L'interface `RefObject`, qui hérite de l'interface `RefFeatured`, représente la notion d'un élément instance d'une métaclasse. Grâce à son lien d'héritage avec l'interface `RefFeatured`, cette interface dispose de toutes les opérations permettant d'accéder aux propriétés de l'élément.

Cette interface offre aussi l'opération `refIsInstance`, qui permet de savoir si l'élément est bien l'instance d'une factory particulière.

RefAssociation

L'interface `RefAssociation`, qui hérite de l'interface `RefBaseObject`, représente la notion de liens entre éléments.

Cette interface offre les opérations `refAddLink` et `refRemoveLink`, qui permettent respectivement l'ajout et la suppression des liens entre éléments.

RefPackage

L'interface `RefPackage`, qui hérite de l'interface `RefBaseObject`, représente la notion de package.

Cette interface offre donc l'opération `refCreateInstance`, qui permet de créer des instances d'une métaclasse.

Règles de génération d'interfaces taylored

Le standard JMI1.0 génère des interfaces taylored Java à partir de métamodèles MOF1.4 *(voir le chapitre 1)*. Nous présentons ici un ensemble de règles de génération JMI relativement simplifiées.

Règle de métaclasse

Une métaclasse d'un métamodèle donnera lieu à la création de deux interfaces : une interface dite `Instance`, qui sera portée par les objets représentant les instances de la métaclasse, et une interface dite `Factory`, offrant les opérations de création des instances de la métaclasse.

L'interface `Instance` présente les caractéristiques suivantes :

- A pour nom `nom_métaclasse`.
- Offre des opérations de lecture et d'écriture pour chaque méta-attribut de la métaclasse.
- Offre des opérations de navigation pour chaque métaréférence de la métaclasse.
- Hérite de l'interface réflective `RefObject`.

L'interface `Factory` présente les caractéristiques suivantes :

- A pour nom `nom_métaclasseClass`.
- Offre des opérations de création des instances de la métaclasse.
- Hérite de l'interface réflective `RefClass`.

Règle de méta-association

Une méta-association d'un métamodèle donnera lieu à la création d'une interface qui présente les caractéristiques suivantes :

- A pour nom `nom_méta-association`.
- Offre des opérations de création des instances de la méta-association (création de liens entre les instances des métaclasses reliées par la méta-association).
- Offre des opérations de navigation sur les liens (permettant d'obtenir des instances des métaclasses liées entre elles).
- Hérite de l'interface réflective `RefAssociation`.

Règle de métapackage

Un package d'un métamodèle donnera lieu à la création d'une interface qui présente les caractéristiques suivantes :

- A pour nom `nom_packagePackage`.
- Offre des opérations permettant d'obtenir toutes les interfaces `Factory` des métaclasses contenues dans le métapackage.
- Offre des opérations permettant d'obtenir toutes les interfaces correspondant aux méta-associations contenues dans le métapackage.
- Hérite de l'interface réflective `RefPackage`.

Ces règles nous font mieux comprendre la philosophie de JMI. Leur application à un métamodèle génère un ensemble d'interfaces Java permettant la manipulation intégrale de modèles instances de ce métamodèle. Ces interfaces permettent la création et la suppression d'éléments de modèle, la navigation parmi les attributs et les références d'un élément et la création et suppression de liens.

Exemple de mise en œuvre

Avant de commencer à illustrer JMI par un exemple, il est important de noter que le standard JMI ne définit que des interfaces Java. Pour pouvoir l'utiliser, il faut disposer de classes implémentant ces interfaces. Celles-ci sont fournies par les différentes plates-formes (*http://mdr.netbeans.org/* ou *http://modfact.lip6.fr*). Les moyens de démarrer ces plates-formes sont donc propriétaires, et c'est pourquoi nous ne les présentons pas ici.

L'exemple que nous avons choisi est celui de la figure 6.1.

Exemple avec les interfaces taylored

L'application des règles de génération des interfaces JMI sur le métamodèle exemple a permis la génération des interfaces suivantes. Les noms un peu particuliers de certaines de ces interfaces, tels que `AHRiteActeur` ou `ATendCasDUtilisation`, viennent principalement de la présence de caractères accentués dans les noms des métaclasses et méta-associations de notre métamodèle et de la traduction de ces caractères vers Java :

- `ACasSystMe.java`. Générée à partir de la méta-association existant entre les métaclasses `système` et `cas d'utilisation`.
- `Acteur`. Générée à partir de la métaclasse `acteur` (interface `Instance`).
- `ActeurClass`. Générée à partir de la métaclasse `acteur` (interface `Factory`).
- `AHRiteActeur`. Générée à partir de la méta-association ayant comme source et cible la métaclasse `acteur` (relation `hérite`).
- `AInclutCasDUtilisation`. Générée à partir de la méta-association ayant comme source et cible la métaclasse `cas d'utilisation` (relation `inclut`).
- `AParticipeActeur`. Générée à partir de la méta-association entre les métaclasses `cas d'utilisation` et `acteur` (relation `participe`).

- `ATendCasDUtilisation`. Générée à partir de la méta-association ayant comme source et cible la métaclasse `cas d'utilisation` (relation `étend`).
- `CasDUtilisation`. Générée à partir de la métaclasse `cas d'utilisation` (interface `Instance`).
- `CasDUtilisationClass`. Générée à partir de la métaclasse `cas d'utilisation` (interface `Factory`).
- `CasPackage`. Générée à partir du métapackage contenant toutes ces métaclasses.
- `SystMe`. Générée à partir de la métaclasse `système` (interface `Instance`).
- `SystMeClass`. Générée à partir de la métaclasse `système` (interface `Factory`).

Grâce à ces interfaces, il est possible de construire notre modèle exemple à l'aide du programme Java suivant :

```
[1]CasPackage extent = //implémentation propriétaire
[2]SystMe sys = extent.getSystMe().createSystMe("PetStore");
[3]Acteur ac = extent.getActeur().createActeur("Client");
[4]CasDUtilisation ca =
‰extent.getCasDUtilisation().createCasDUtilisation("Commander panier");
[5]CasDUtilisation ca2 =
‰extent.getCasDUtilisation().createCasDUtilisation("Valider panier");

[6]ac.getParticipe().add(ca);
[7]ac.getParticipe().add(ca2);
[8]sys.getCas().add(ca);
[9]sys.getCas().add(ca2);
```

La première ligne du programme permet d'identifier un objet représentant un package. L'identification de cet objet n'est pas standard et dépend de la plate-forme JMI utilisée.

Les deuxième et troisième lignes permettent respectivement d'obtenir les factory des métaclasses `système` et `acteur` et de créer une instance de `système` (nommée `PetStore`) et une instance d'`acteur` (nommée `client`).

Les quatrième et cinquième lignes permettent par la même approche d'obtenir la factory de la métaclasse `cas d'utilisation` et de créer deux instances dont les intitulés sont `Commander article` et `Valider panier`.

Les lignes suivantes permettent d'établir les liens entre ces instances.

Exemple avec les interfaces réflectives

L'utilisation des interfaces réflectives ne nécessite pas d'étape de génération d'interfaces. Elles peuvent donc être utilisées directement.

Le modèle de la figure 6.1 peut être directement construit à l'aide du programme Java suivant :

```
[1] RefPackage p = //propriétaire
[2] RefObject act = p.refClass("Acteur").refCreateInstance(null);
[3] act.refSetValue("nom","Client");
```

```
[4] RefObject ca1 = p.refClass("Cas d'Utilisation").refCreateInstance(null);
[5]ca1.refSetValue("intitulé","Commander Panier");
[6]RefObject ca2 = p.refClass("Cas d'Utilisation").refCreateInstance(null);
[7]ca2.refSetValue("intitulé","Valider Panier");
[8]Collection col = (Collection) act.refGetValue("participe");
[9]col.add(ca1);
[10]col.add(ca2);
[11]RefObject sys = p.refClass("Système").refCreateInstance(null);
[12]sys.refSetValue("nom","PetStore");
[13]Collection cas = (Collection) sys.refGetValue("cas");
[14]cas.add(ca1);
[15]cas.add(ca2);
```

La première ligne du programme permet d'identifier un objet représentant un package. L'identification de cet objet n'est pas standard et dépend de la plate-forme JMI utilisée.

La deuxième ligne identifie l'objet jouant le rôle de factory pour la métaclasse `acteur` afin de créer une instance de cette métaclasse. La troisième ligne permet de spécifier le nom de cette instance.

Les quatrième et sixième lignes permettent, *via* le même mécanisme d'identification de factory, de créer deux instances de la métaclasse `cas d'utilisation`. Les cinquième et septième lignes permettent de spécifier l'intitulé des instances.

Les huitième, neuvième et dixième lignes permettent de spécifier les liens entre les cas d'utilisation et l'acteur.

Les dernières lignes permettent de créer une instance de la métaclasse `système` et de spécifier les liens entre cette instance et les cas d'utilisation.

Cet exemple montre que la manipulation des interfaces réflectives est plus délicate mais permet une manipulation des modèles indépendamment des métamodèles.

EMF (Eclipse Modeling Framework)

EMF est un framework Open Source fondé sur les mêmes principes architecturaux que JMI, si ce n'est qu'il est fortement couplé à la plate-forme Eclipse. Dans EMF, il est possible de définir un métamodèle et de générer les interfaces taylored dédiées à ce métamodèle afin de pouvoir manipuler les instances du métamodèle dans Eclipse. EMF dispose, tout comme JMI, d'interfaces réflectives.

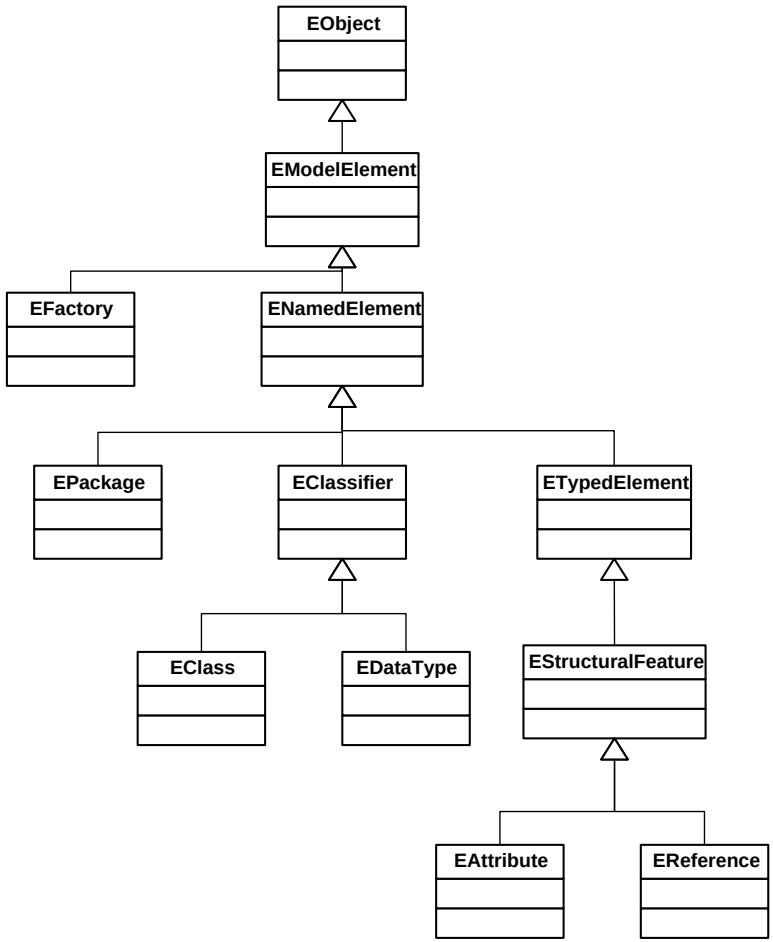
Nous commencerons par présenter le métamétamodèle d'EMF, lequel est différent du métamétamodèle MOF, puis nous présenterons les interfaces réflectives et taylored d'EMF avant de les illustrer par un même exemple. Nous finirons cette section en présentant les fonctionnalités supplémentaires offertes par le framework EMF.

Le métamétamodèle d'EMF

La particularité d'EMF est qu'il se fonde sur la version MOF2.0 et non MOF1.4 comme JMI. En fait, EMF propose son propre métamétamodèle, le métamétamodèle Ecore, qui ressemble fortement au métamétamodèle EMOF *(voir le chapitre 1)*, car il ne supporte que la notion de métaclasse sans méta-association. Ecore est légèrement différent du standard EMOF en ce qu'il est entièrement intégré à la plate-forme Eclipse.

La figure 6.4 illustre les métaclasses du métamétamodèle Ecore. Nous n'allons pas présenter ici l'intégralité de ce métamétamodèle. Il est simplement important de savoir que les métamodèles conformes à ce métamétamodèle sont composés d'EClass contenant des EAttribute et des EReference.

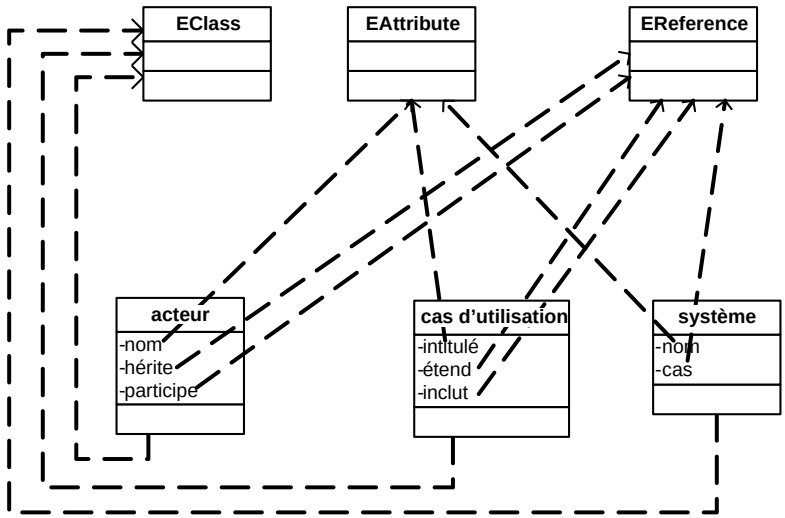
Figure 6.4
Sous-ensemble du métamétamodèle Ecore



La figure 6.5 illustre le métamodèle exemple des cas d'utilisation sous EMF *(voir figure 6.1)*. Ce métamodèle est constitué de trois EClass (le terme *EClass* exprime la notion de méta-classe dans le framework EMF) : `acteur`, `système` et `cas d'utilisation`. Les méta-associations entre ces EClass ont disparu au profit d'EReference. On voit que l'EClass `acteur` possède deux EReference, `hérite` et `participe`, qui remplacent respectivement les méta-associations présentes dans le métamodèle de la figure 6.1.

Le fait de remplacer les méta-associations par des EReference ne restreint pas les capacités d'expression des métamodèles. Il est d'ailleurs possible de transformer tout métamodèle MOF en un métamodèle Ecore (une transformation automatique est fournie dans le framework EMF).

Figure 6.5
Le métamodèle exemple sous EMF



Les interfaces réflectives d'EMF

Tout comme JMI, EMF fourni des interfaces réflectives. Celles-ci permettent la manipulation des modèles d'une façon indépendante de leur métamodèle.

La particularité d'EMF est que toutes les interfaces réflectives qu'il propose sont spécifiées dans le métamétamodèle Ecore.

EObject

L'interface réflective la plus importante est `EObject`. Elle représente n'importe quel élément, qu'il appartienne à un modèle ou à un métamodèle.

Cette interface offre l'opération `eClass()`, qui permet d'obtenir l'EClass de l'élément. Cette opération retourne un objet Java de type EClass.

L'interface `EObject` offre aussi les opérations `eGet()` et `eSet()`, qui permettent respectivement de lire et d'écrire les valeurs des différentes propriétés de l'élément (attributs et références).

EClass

L'interface réflective `EClass` représente une métaclasse d'un métamodèle.

Cette interface offre les opérations `getEAttributes()` et `getEReferences()`, qui permettent d'obtenir la liste respectivement de tous les attributs et références contenus dans la méta-classe. Cette interface offre aussi l'opération `getEStructuralFeature()`, qui permet d'obtenir une propriété (attribut ou référence) d'une EClass à partir de son nom.

EPackage

L'interface réflective `EPackage` représente le moyen d'accès à toutes les EClass définies dans un package.

Cette interface offre l'opération `getEClassifier(String qname)`, qui permet de récupérer la référence vers une EClass d'un métamodèle à partir de son nom.

EFactory

L'interface réflective `EFactory` représente le moyen de créer des instances des EClass définies dans un package.

Cette interface offre l'opération `create()`, qui permet de créer une instance d'une EClass.

Règles de génération d'interfaces taylored

Les règles de génération d'interfaces taylored d'EMF sont similaires aux règles de génération JMI. Elles sont même un peu plus simples car elles ne souffrent pas du problème délicat de la traduction en Java des méta-associations, puisque ces dernières n'existent pas.

Nous présentons ici une version simplifiée de ces règles.

Règles EClass

Une EClass donne lieu à la création d'une seule interface (contrairement à JMI qui génère deux interfaces par métaclasse).

Cette interface présente les caractéristiques suivantes :

- A pour nom `nom_EClass`.
- Offre des opérations de lecture et d'écriture pour chaque EAttribute de l'EClass.
- Offre des opérations de lecture et d'écriture pour chaque EReference de l'EClass.
- Hérite de l'interface réflective `EObject`.

Règles EPackage

Un EPackage donne lieu à la création de deux interfaces : une interface `Factory`, permettant la création de toute instance des EClass contenues dans l'EPackage, et une interface

`Package` offrant les opérations de navigation entre toutes les EClass d'un package d'un métamodèle.

L'interface `Factory` présente les caractéristiques suivantes :

- A pour nom `nom_pakcageFactory`.
- Offre une opération de création pour chaque EClass contenue dans l'EPackage
- Hérite de l'interface réflective `EFactory`.

L'interface `Package` présente les caractéristiques suivantes :

- A pour nom `nom_packagePackage`.
- Offre une opération de navigation pour chaque EClass contenue dans l'EPackage permettant d'obtenir l'EClass correspondante.
- Hérite de l'interface réflective `EPackage`.

Génération des classes d'implémentation

Contrairement à JMI, EMF propose, en plus de la génération des interfaces taylored, une génération des classes d'implémentation réalisant ces interfaces. Les règles de génération de ces classes d'implémentation sont très complexes car elles supportent la cohérence des modèles. Si, par exemple, un élément référence un autre élément et que cet autre élément soit supprimé, les classes d'implémentation supportent la mise à jour de la référence. Ces règles de génération assurent aussi la cohérence du modèle Java par rapport au modèle EMF. Elles définissent notamment un moyen de supporter l'héritage multiple des EClass alors que l'héritage multiple entre les classes Java est non supporté.

Exemple de mise en œuvre

Comme nous l'avons fait pour JMI, nous allons illustrer les interfaces EMF à partir de l'exemple de modèle de cas d'utilisation de la figure 6.1.

Utilisation des interfaces taylored

L'application des règles de génération des interfaces EMF à notre métamodèle exemple permet la génération des interfaces suivantes :

- `Acteur`. Générée à partir de l'EClass `acteur`.
- `CasdUtilisation`. Générée à partir de l'EClass `cas d'utilisation`.
- `CasFactory`. Générée à partir de l'EPackage représentant l'intégralité du métamodèle. Cette interface est l'interface `Factory` de l'EPackage.
- `CasPackage`. Générée à partir de l'EPackage représentant l'intégralité du métamodèle. Cette interface est l'interface `Package` de l'EPackage.
- `Système`. Générée à partir de l'EClass `système`.

Pour chacune de ces interfaces, EMF génère la classe d'implémentation correspondante.

Grâce à ces interfaces, il est possible de construire notre modèle exemple à l'aide du programme Java suivant :

```
[1] CasFactory fact = CasFactory.eINSTANCE;
[2] Acteur ac = fact.createActeur();
[3] ac.setNom("Client");
[4] CasdUtilisation cau1 = fact.createCasdUtilisation();
[5] cau1.setIntitulé("Commander Panier");
[6] CasdUtilisation cau2 = fact.createCasdUtilisation();
[7] cau2.setIntitulé("Valider Panier");
[8] ac.getParticipe().add(cau1);
[9] ac.getParticipe().add(cau2);
[10] Système sys = fact.createSystème();
[11] sys.setNom("PetStore");
[12] sys.getCas().add(cau1);
[13] sys.getCas().add(cau2);
```

La première ligne du programme permet d'obtenir une référence vers la factory correspondant au package. Cette factory sera utilisée pour créer les instances des EClass.

La deuxième ligne permet de créer une instance de l'EClass `acteur`. La troisième ligne permet de spécifier le nom de cet acteur.

Les quatrième, cinquième, sixième et septième lignes permettent de créer les deux instances de l'EClass `cas d'utilisation` et de spécifier leur intitulé respectif.

Les huitième et neuvième lignes permettent de spécifier les liens entre les acteurs et les cas d'utilisation.

Les dixième et onzième lignes permettent de créer l'instance de l'EClass `système` et de spécifier son nom.

Les douzième et treizième lignes permettent de spécifier les liens entre le système et les cas d'utilisation.

Utilisation des interfaces réflectives

Comme pour JMI, l'utilisation des interfaces réflectives ne nécessite pas d'étape de génération d'interface. Ces interfaces peuvent donc être utilisées telles quelles pour construire le modèle exemple. Le programme suivant illustre l'utilisation de ces interfaces pour construire le modèle exemple à l'aide d'un programme Java :

```
[1] EFactory fact = //obtention de la référence
[2] EPackage pa = //obtention de la référence
[3] EClass acEC = (EClass) pa.getEClassifier("Acteur");
[4] EObject ac = fact.create(acEC);
[5] ac.eSet(acEC.getEStructuralFeature("nom") , "Client");
[6] EClass cuEC = (EClass) pa.getEClassifier("CasdUtilisation");
[7] EObject cu1 = fact.create(cuEC);
[8] cu1.eSet(cuEC.getEStructuralFeature("intitulé") , "Commander Panier");
[9] EObject cu2 = fact.create(cuEC);
[10] cu1.eSet(cuEC.getEStructuralFeature("intitulé") , "Valider Panier");
```

```
[11] Collection parti = (Collection) ac.eGet(acEC.getEStructuralFeature
➥("participe"));
[12] parti.add(cu1);
[13] parti.add(cu2);
[14] EClass sysEC = (EClass) pa.getEClassifier("Système");
[15] EObject sys = fact.create(sysEC);
[16] sys.eSet(sysEC.getEStructuralFeature("nom") , "PetStore");
[17] Collection cas = (Collection) sys.eGet(sysEC.getEStructuralFeature("cas"));
[18] cas.add(cu1);
[19] cas.add(cu2);
```

La première ligne permet d'obtenir une référence à un élément représentant le package. Cet élément sera utilisé pour obtenir toutes les informations des EClass du métamodèle.

La deuxième ligne permet d'obtenir une référence vers un élément représentant la factory du package. Cet élément sera utilisé pour créer toutes les instances des EClass.

Dans ces deux premières lignes du programme nous avons masqué la façon dont nous pouvons obtenir les références par souci de simplicité. L'obtention de ces références est en effet un mécanisme lourd et très complexe dans EMF.

La troisième ligne permet d'obtenir une référence vers l'EClass `acteur`. La quatrième ligne permet de créer une instance de l'EClass `acteur`.

La cinquième ligne permet de spécifier le nom de l'acteur. Pour pouvoir spécifier le nom de l'acteur, il faut naviguer dans l'EClass `acteur` et récupérer l'EAttribute correspondant.

La sixième ligne permet d'obtenir une référence vers l'EClass `cas d'utilisation`. La septième ligne permet de créer une instance de l'EClass `cas d'utilisation`. La huitième ligne permet de spécifier l'intitulé de cette instance (`Commander un article`).

Les neuvième et dixième lignes permettent de créer la deuxième instance de l'EClass `cas d'utilisation` et de spécifier son intitulé (`Valider un panier`).

Les onzième, douzième et treizième lignes permettent d'établir les liens entre l'acteur et les cas d'utilisation.

Les quatorzième et quinzième lignes permettent de créer une instance de l'EClass `système`. La seizième ligne permet de spécifier le nom de cette instance (`PetStore`).

Les dix-septième, dix-huitième et dix-neuvième lignes permettent de spécifier les liens entre le système et les cas d'utilisation.

Là encore, cet exemple montre que les interfaces réflectives permettent d'effectuer les mêmes opérations sur les modèles mais qu'elles sont plus délicates à utiliser.

Fonctionnalités du framework EMF

EMF est un framework Open Source dont l'objectif dépasse la simple génération des interfaces de manipulation des modèles. Le but de ce framework est de faciliter la manipulation des modèles afin de permettre leur intégration dans la plate-forme Eclipse.

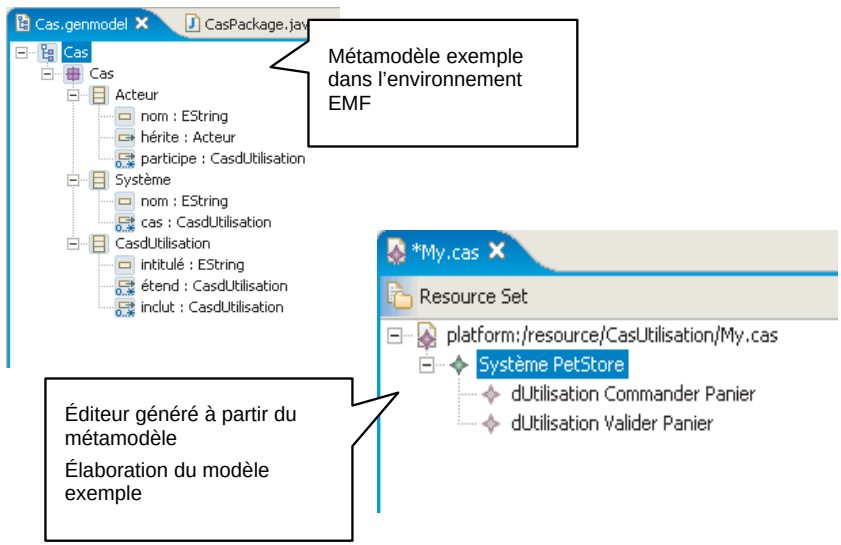
C'est dans cet objectif que plusieurs fonctionnalités ont été développées pour permettre le développement de nouveaux métamodèles et assurer la manipulation des modèles instances de ces métamodèles dans la plate-forme Eclipse. Parmi ces fonctionnalités, celle qui est certainement la plus agréable est la génération automatique d'un simple éditeur graphique permettant l'édition des modèles sous forme arborescente.

L'idée sous-jacente est de générer automatiquement, à partir d'un métamodèle, un éditeur graphique offrant une vue arborescente d'un modèle. Chacun des nœuds de l'éditeur représentera une instance d'une métaclasse.

Cette fonctionnalité s'utilise très simplement dans le framework EMF. Il suffit de demander la génération des classes Java composant l'éditeur graphique correspondant à un métamodèle puis d'exécuter ces classes dans la plate-forme Eclipse afin de visualiser l'éditeur graphique.

Nous avons utilisé cette fonctionnalité sur notre métamodèle exemple et avons pu élaborer notre modèle grâce à cet éditeur graphique, comme l'illustre la figure 6.6.

Figure 6.6
Génération d'éditeur graphique de modèle dans EMF



D'autres fonctionnalités proposées par le framework EMF mériteraient d'être présentées. Compte tenu de notre sujet, qui est la mise en œuvre des aspects de production des modèles, nous avons fait le choix de ne présenter que celle permettant la génération d'une interface graphique arborescente.

Synthèse

Ce chapitre s'est penché sur la manipulation des modèles par les langages de programmation orientée objet. Cette approche consiste à fournir des interfaces très génériques permettant

la manipulation des modèles indépendamment de leur métamodèle (interfaces dites réflectives) et à générer des interfaces spécifiques d'un métamodèle permettant uniquement la manipulation d'un type de modèle (interfaces dites taylored).

Ces approches se fondent sur une structuration forte des modèles. C'est grâce à l'architecture en couches (modèle, métamodèle et métamétamodèle) qu'il a été possible de définir ces interfaces de manipulation de modèles.

Nous avons vu deux mises en œuvre de cette approche aux travers de JMI et d'EMF. JMI est un standard JCP de Sun, qui consiste à permettre la manipulation en Java des modèles instances de métamodèles MOF. EMF est un framework Open Source de la plate-forme Eclipse, qui vise à permettre la manipulation dans Eclipse des modèles instances de métamodèles Ecore.

Ces deux approches sont comparables à tous égards, et il n'est pas possible de dire si l'une est meilleure que l'autre. Ajoutons que JMI est utilisé par l'outil Poseidon pour manipuler les modèles UML1.4, tandis qu'EMF est utilisé par RSA (Rational Software Architect) pour manipuler les modèles UML2.0.

Nous avons souhaité présenter ces deux approches dans cet ouvrage parce que ce sont les plus diffusées et que de nombreux projets, souvent Open Source, les supportent.

7

Transformation de modèles

Rendre les modèles productifs consiste à leur appliquer des transformations de modèles pour obtenir des résultats utiles au développement, généralement sous la forme de modèles plus détaillés et proches de la solution technique ou du code.

Pour ces raisons, les transformations de modèles sont omniprésentes dans MDA. Elles permettent notamment de transformer un modèle CIM en un modèle PIM ou d'obtenir un modèle PSM à partir d'un modèle PIM.

Ce chapitre précise la notion de transformation de modèles telle qu'elle est définie dans MDA et présente les différentes technologies qui permettent de réaliser des transformations de modèles.

Transformation et MDA

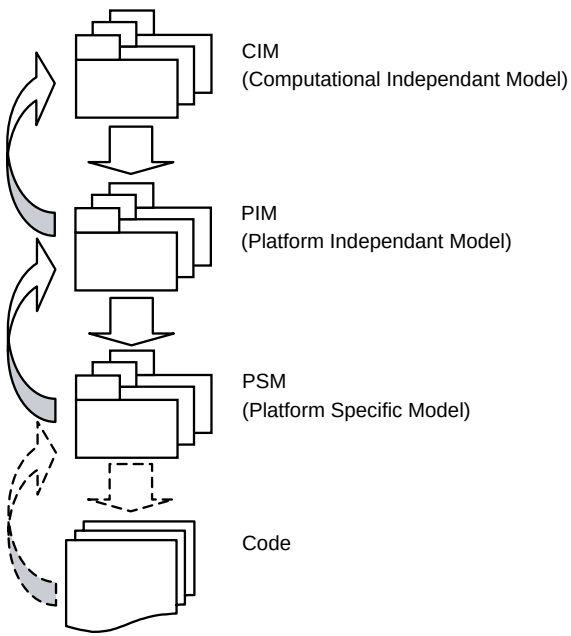
Comme expliqué au chapitre 1, MDA représente à l'aide de modèles toutes les informations permettant la construction d'applications informatiques. La mise en œuvre de MDA induit donc nécessairement la création d'un nombre important de modèles. Chacun de ces modèles contient une partie de l'information utile pour la génération des applications informatiques.

La mise en production de MDA passe par la mise en relation de ces modèles et plus précisément par leurs transformations respectives. C'est grâce aux transformations de modèles qu'il est possible, par exemple, d'obtenir une traçabilité entre des modèles très abstraits, proches des besoins exprimés par les utilisateurs, et des modèles très concrets, proches des plates-formes d'exécution.

La figure 7.1 reprend les phases de MDA et illustre les principales transformations de modèles MDA :

- **Transformations de modèles CIM vers PIM.** Permettent de construire des PIM partiels à partir des informations contenues dans les CIM. L'objectif est de s'assurer que les besoins exprimés dans les CIM sont retranscrits dans les PIM. Ces transformations sont essentielles à la pérennité des modèles. Ce sont elles qui garantissent les liens de traçabilité entre les modèles et les besoins exprimés par le client.
- **Transformations de modèles PIM vers PIM.** Permettent de raffiner les PIM afin d'améliorer la précision des informations qu'ils contiennent. En UML, de telles transformations peuvent être, par exemple, la création de classes UML à partir d'un ensemble de diagrammes de séquences ou la création de diagrammes d'états à partir de diagrammes de classes. Ces transformations sont omniprésentes dans les outils d'élaboration de modèles PIM. Elles permettent d'accélérer la production des PIM et donc de raccourcir le cycle de développement.
- **Transformations de modèles PIM vers PSM.** Permettent de construire une bonne partie des modèles PSM à partir des modèles PIM. Ces transformations sont les plus importantes de MDA car elles garantissent la pérennité des modèles aussi bien que leur productivité et leur lien avec les plates-formes d'exécution. Nous illustrons ces transformations au chapitre 12, dédié à l'étude de cas.
- **Transformations de modèles PSM vers du code.** Permettent de générer la totalité du code. Nous verrons plus loin dans ce chapitre que ces transformations ne sont pas vraiment considérées comme étant des transformations de modèles.

Figure 7.1
Transformations principales de MDA



Transformations inverses

Même si ce n'est pas l'objectif principal de MDA, toutes ces transformations ont bien entendu leur transformation inverse (code vers PSM, PSM vers PIM et PIM vers CIM). Ces transformations inverses seront d'ailleurs fortement mises à contribution dans l'élaboration du futur standard ADM (Architecture Driven Modernization) de l'OMG, qui vise à définir l'approche inverse de MDA, c'est-à-dire à construire des modèles à partir d'applications existantes.

Nous venons de voir à quel point les transformations de modèles sont incontournables dans MDA. Dans la mesure où ce sont elles qui permettent de rendre les modèles productifs, leur définition est stratégique pour la mise en œuvre de MDA.

Métamodèles et règles de correspondance

Quel que soit son type (PIM vers PSM, PIM vers PIM, etc.), une transformation de modèles s'apparente toujours à une fonction qui prend en entrée un ensemble de modèles et qui fournit en sortie un ensemble de modèles.

Les modèles d'entrée et de sortie sont tous structurés par leur métamodèle. C'est d'ailleurs ce qui permet de discerner dans MDA les transformations de modèles des générations de code. Pour pouvoir considérer la génération de code comme une transformation de modèles, il faudrait être capable de construire le métamodèle du code. Plusieurs tentatives ont vu le jour pour construire, par exemple, le métamodèle de Java afin de considérer la génération de code Java comme étant une transformation de modèles. Ces tentatives n'ont pas eu un succès majeur et ont même plutôt démontré que la réalisation d'un tel métamodèle n'était pas très réaliste.

Cela explique en partie pourquoi l'OMG accomplit actuellement un effort de standardisation afin de créer un langage permettant de définir des générations de code à partir de modèles. Dans la suite de l'ouvrage, nous considérons que les générations de code ne sont pas des transformations de modèles.

Une transformation de modèles MDA est donc une fonction dont les paramètres d'entrée et de sortie sont des modèles structurés par des métamodèles. Il est aussi possible, dans le contexte particulier des modèles UML, que les modèles d'entrée et de sortie soient *profilés*, c'est-à-dire qu'ils soient étiquetés par des stéréotypes définis par des profils *(voir le chapitre 3)*. Du point de vue des transformations de modèles, les profils sont considérés comme des métamodèles dans le sens où ils structurent un ensemble de modèles.

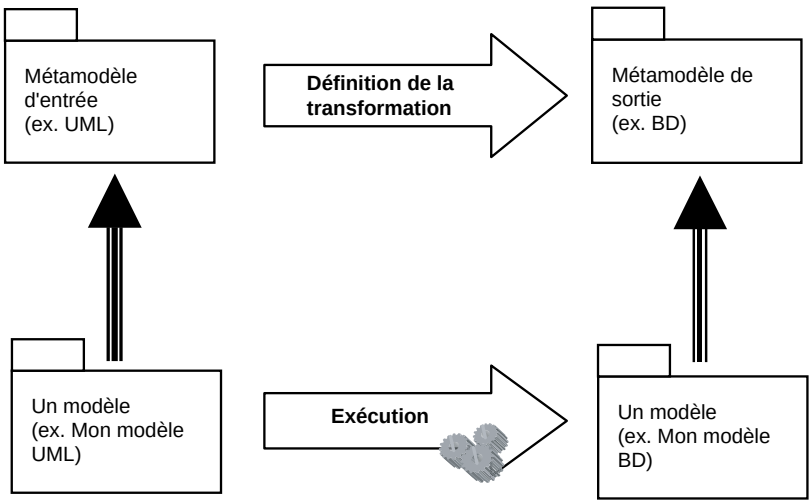
Si une transformation de modèles s'exécute au niveau des modèles, elle se spécifie au niveau des métamodèles. En effet, une transformation exprime des correspondances structurelles entre les modèles source et cible. Ces correspondances structurelles s'appuient sur les métamodèles des modèles source et cible.

Par exemple, une transformation de modèles visant à transformer des modèles UML vers des schémas de bases de données relationnelles (BD) spécifierait la règle suivante : à toute classe UML correspond une table d'un schéma d'une base de données. Cette même

règle, exprimée selon les métamodèles, deviendrait : à tout élément du modèle source instance de la métaclasse `Class` du métamodèle UML doit correspondre un élément du modèle cible instance de la métaclasse `Table` du métamodèle BD (en faisant l'hypothèse que le métamodèle BD définit la métaclasse `Table`). Cette règle exprime bien une correspondance structurelle entre les métamodèles UML et BD et plus précisément une correspondance entre les métaclasses `Class` et `Table`.

La figure 7.2 illustre les relations entre les transformations de modèles et les métamodèles. Si la transformation est ici binaire, avec une unique source et une unique cible, il est important de souligner que les transformations de modèles sont *N*-aires, c'est-à-dire qu'elles peuvent avoir en entrée et en sortie plusieurs modèles.

Figure 7.2
Métamodèles et transformation de modèles



Nous venons de voir que les transformations de modèles étaient des transformations structurelles. Elles sont constituées d'un ensemble de règles exprimant chacune des correspondances structurelles entre les métamodèles source et cible. Le rôle des métamodèles dans les transformations de modèles MDA est de définir les structurations possibles des modèles source et cible et de servir de base à la définition des règles de transformation.

Transformations sémantiques

Les transformations de modèles ne sont pas des transformations sémantiques. Une transformation de modèle ne peut intrinsèquement préserver la sémantique des modèles source et cible. Il n'est en effet pas possible de définir des relations sémantiques à partir de relations structurelles. Dans notre exemple, le fait d'avoir défini la transformation des modèles UML vers des modèles BD ne signifie pas que la sémantique des modèles UML se retrouve transcrite dans les modèles BD résultats de la transformation. Assurer une cohérence sémantique lors d'une transformation de modèle ne peut se faire que par l'utilisation d'autres techniques, telles que les techniques formelles. Ce domaine relève pour l'instant de la recherche.

Spécification des règles de transformation

Ce qui différencie les différentes approches permettant l'élaboration des transformations de modèles est la façon dont sont spécifiées les règles des transformations. Trois approches aujourd'hui répertoriées permettent la spécification de ces règles de correspondances.

- **Approche par programmation.** Consiste à utiliser les langages de programmation orientée objet. L'idée est de programmer une transformation de modèles de la même manière que l'on programme n'importe quelle application informatique. Ces transformations sont donc des applications informatiques qui ont la particularité de manipuler des modèles. Ces applications informatiques utilisent les interfaces de manipulation de modèles que nous avons présentées au chapitre précédent. Cette approche est la plus utilisée car elle est très puissante et fortement outillée.
- **Approche par template.** Consiste à définir les canevas des modèles cibles souhaités en y déclarant des paramètres. Ces paramètres seront substitués par les informations contenues dans les modèles sources. On appelle ces canevas des « modèles cibles paramétrés » ou des « modèles templates ». L'exécution d'une transformation consiste à prendre un modèle template et à remplacer ses paramètres par les valeurs d'un modèle source. Cette approche nécessite un langage particulier permettant la définition des modèles templates. Sa puissance réside principalement dans le langage de définition des modèles templates. De tels langages sont en cours d'élaboration et n'ont pas encore la maturité des langages de programmation orientée objet utilisés dans la première approche.
- **Approche par modélisation.** Consiste à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'objectif est de modéliser les transformations de modèles et de rendre les modèles de transformation pérennes et productifs et d'exprimer leur indépendance vis-à-vis des plates-formes d'exécution. Cette approche est actuellement au stade de la recherche. Le standard MOF2.0 QVT (Query, View, Transformation) en cours de définition à l'OMG a pour objectif de définir le métamodèle permettant l'élaboration des modèles de transformation de modèles. Actuellement, seuls quelques prototypes de recherche supportent cette approche.

Exemple de mise en œuvre

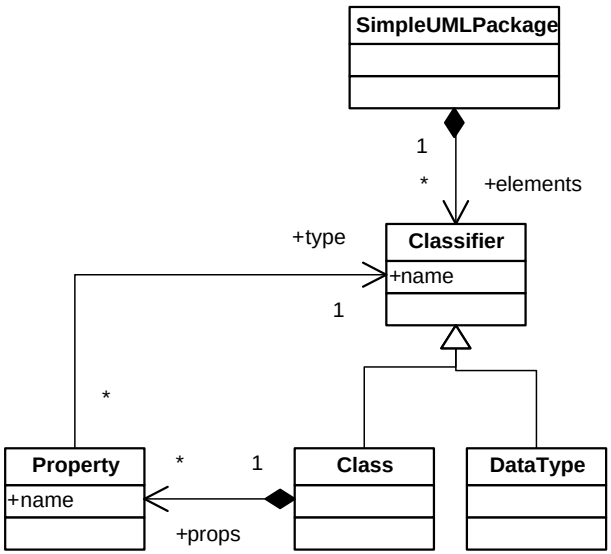
Afin de mieux comprendre les différentes approches de transformation de modèles, nous allons les illustrer par un exemple simple de transformation d'un modèle UML vers un schéma de bases de données (BD).

La figure 7.3 illustre le métamodèle source de cette transformation. Ce métamodèle représente une version très simplifiée d'UML. La métaclasse `SimpleUMLPackage` représente le concept de package UML. Cette métaclasse est reliée à la métaclasse `Classifier`, qui est la métaclasse abstraite représentant aussi bien le concept de classe UML que le concept de type de donnée UML. La métaclasse `DataType` représente le concept de type de

donnée UML. La métaclasse `Class`, qui hérite de la métaclasse `Classifier`, représente le concept de classe UML. Cette métaclasse est reliée à la métaclasse `Property`, qui représente le concept de propriété UML. La métaclasse `Property` est reliée à la métaclasse `Classifier` afin d'exprimer le type d'une propriété.

Ce métamodèle structure des modèles UML simplifiés constitués de packages contenant des types de données et des classes, ces dernières pouvant contenir des propriétés typées.

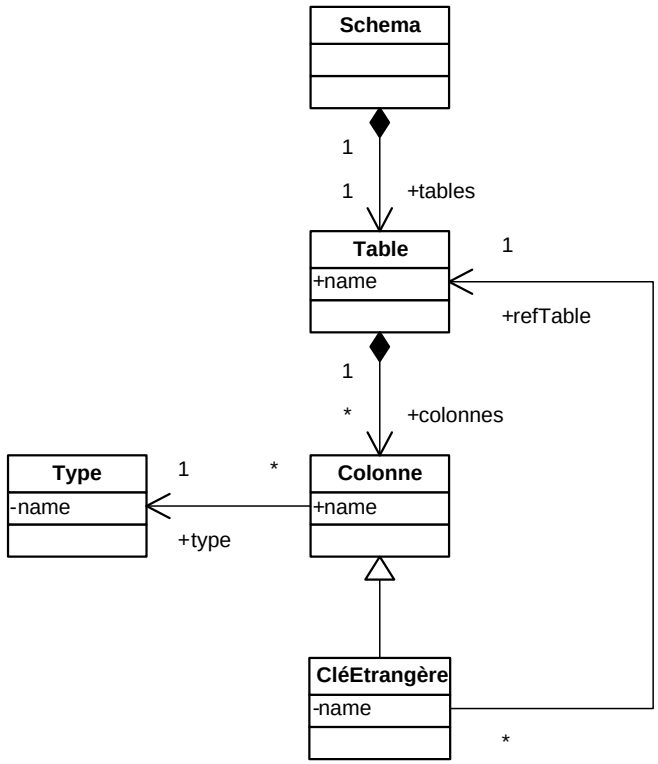
Figure 7.3
Métamodèle UML simplifié (source de la transformation)



La figure 7.4 illustre le métamodèle cible de la transformation. Ce métamodèle représente une version très simplifiée des schémas relationnels de base de données. La métaclasse `Schema` représente le concept de schéma de base de données. Cette métaclasse est reliée à la métaclasse `Table`, qui représente le concept de table dans les bases de données. La métaclasse `Table` est reliée à la métaclasse `Colonne`, qui représente une colonne d'une table de base de données. La métaclasse `Colonne` est reliée à la métaclasse `Type` afin de permettre de spécifier le type de la colonne. La métaclasse `CléEtrangère`, qui hérite de la métaclasse `Colonne`, représente le concept de clé étrangère. Cette métaclasse est reliée à la métaclasse `Table`, ce qui permet de spécifier à quelle table la clé étrangère fait référence.

Ce métamodèle structure des modèles représentant des schémas de base de données. Ces schémas sont constitués de plusieurs tables, elles-mêmes constituées de colonnes typées. Certaines des colonnes peuvent contenir des clés étrangères permettant d'exprimer des relations entre les tables.

Figure 7.4
Métamodèle d'un schéma relationnel simplifié (cible de la transformation)



Ces deux métamodèles exemples ont pour unique objectif de servir de support à la transformation de modèles que nous allons détailler. Ceux-ci sont simplifiés à l'extrême et ne doivent pas être considérés comme des métamodèles établis du domaine UML et de celui des schémas de base de données.

La transformation exemple, qui consiste à transformer les modèles UML vers des schémas de base de données, peut se définir de la façon suivante :

1. À tout package UML correspond un schéma.
2. À toute classe UML du package correspond une table dans le schéma.
3. À toute propriété UML d'une classe correspond une colonne dans la table :
   - Si le type de la propriété est une classe UML, la colonne est une clé étrangère.
   - Si le type de la propriété est un type de donnée UML, la colonne est du type correspondant.
4. À tout type de donnée UML d'un package correspond un type de donnée dans le schéma.

Les sections suivantes montrent comment ces quatre règles sont élaborées selon les approches par programmation, par template et par modélisation.

Élaboration des règles par programmation

Comme expliqué précédemment, l'approche par programmation consiste à programmer les transformations de modèles en utilisant les langages de programmation orientée objet et les interfaces de manipulation de modèles.

Les principes de l'approche par programmation sont donc très simples. Il suffit de disposer d'un langage de programmation et d'un moyen de manipuler les modèles.

Certains outils suivent cette approche en offrant plusieurs assistants ou frameworks facilitant le codage des transformations de modèles. Ces assistants et frameworks sont très agréables à utiliser et apportent des gains significatifs. Le chapitre 8 présente les frameworks proposés dans deux outils MDA du marché, IBM Rational Software Modeler et Softeam MDA Modeler.

Exemple de mise en œuvre

Dans cet exemple, nous avons fait le choix d'utiliser le langage Java avec les interfaces de manipulation de modèles taylored d'EMF *(voir le chapitre 6)*. Ce choix illustre pleinement l'approche par programmation sans nécessiter d'assistant ni de framework supplémentaire au framework EMF. Le framework EMF est très facilement téléchargeable étant donné son caractère Open Source.

Nous convertissons les métamodèles source et cible MOF vers des métamodèles Ecore *(voir le chapitre précédent)* puis appliquons les règles de génération d'interfaces taylored du framework EMF.

Grâce à EMF, le métamodèle UML simplifié génère les interfaces suivantes permettant la manipulation des modèles UML :

- `SimplePackageUML`. Obtenue à partir de l'EClass `SimplePackageUML`, cette interface offre l'opération `getElements()`, qui permet d'obtenir les éléments contenus dans un package.
- `Classifier`. Obtenue à partir de l'EClass `Classifier`, cette interface offre les opérations `getName()` et `setName()`, qui permettent de lire et d'écrire le nom de l'élément.
- `Class`. Obtenue à partir de l'EClass `Class`, cette interface, qui hérite de l'interface `Classifier`, offre l'opération `getProps()`, qui permet d'obtenir la liste des propriétés de la classe.
- `DataType`. Obtenue à partir de l'EClass `DataType`, cette interface hérite de l'interface `Classifier`.
- `Property`. Obtenue à partir de l'EClass `Property`, cette interface offre les opérations `getType()` et `setType()`, qui permettent de spécifier le type de la propriété. Cette interface offre aussi les opérations `getName()` et `setName()`, qui permettent de lire et d'écrire le nom de l'élément.

- `UmlFactory`. Obtenue à partir de l'EPackage UML, cette interface offre toutes les opérations permettant de créer des instances des EClass de l'EPackage UML. Par exemple, l'opération `createClass()` permet de créer une instance de l'EClass `Class`.

Le métamodèle simplifié de schéma de base de données permet quant à lui de générer les interfaces suivantes :

- `Schema`. Obtenue à partir de l'EClass `Schema`, cette interface offre l'opération `getTables()`, qui permet d'obtenir toutes les tables contenues dans le schéma de base de données.
- `Table`. Obtenue à partir de l'EClass `Table`, cette interface offre l'opération `getColonnes()`, qui permet d'obtenir toutes les colonnes contenues dans une table. Cette interface offre aussi les opérations `getName()` et `setName()`, qui permettent de lire et d'écrire le nom de l'élément.
- `Colonne`. Obtenue à partir de l'EClass `Colonne`, cette interface offre les opérations `getType()` et `setType()`, qui permettent de spécifier le type de la colonne. Cette interface offre aussi les opérations `getName()` et `setName()`, qui permettent de lire et d'écrire le nom de l'élément.
- `Type`. Obtenue à partir de l'EClass `Type`, cette interface offre les opérations `getName()` et `setName()`, qui permettent de lire et d'écrire le nom de l'élément.
- `CléEtrangère`. Obtenue à partir de l'EClass `CléEtrangère`, cette interface offre les opérations `getRefTable()` et `setRefTable()`, qui permettent de spécifier la table référencée par la clé étrangère.
- `BdFactory`. Obtenue à partir de l'EPackage BD, cette interface offre toutes les opérations permettant de créer des instances des EClass de l'EPackage BD. Par exemple, l'opération `createTable()` permet de créer une instance de l'EClass `Table`.

Grâce à ces interfaces de manipulation de modèles, il est possible d'écrire sous forme de programme Java la transformation présentée précédemment. Nous avons choisi de structurer ce programme Java en une seule classe comprenant une méthode par règle de transformation. Les sections suivantes présentent les différentes méthodes de cette classe.

Méthodes de transformation de la classe Java

La méthode `simpleUMLPackage2Schema` ci-dessous correspond à la première règle, qui consiste à construire un schéma à partir d'un package. La deuxième ligne de cette méthode permet de créer une instance de l'EClass `Schema`. Les troisième et quatrième lignes permettent d'itérer sur tous les éléments contenus dans le package. Les cinquième, sixième et septième lignes permettent d'appeler l'exécution de la règle de transformation relative aux classes UML afin de créer des tables dans le schéma. Les huitième, neuvième et dixième lignes permettent d'appeler l'exécution de la règle de transformation relative aux types de données UML afin de créer des types dans le schéma.

```
[1] public void simpleUMLPackage2Schema(SimpleUMLPackage pc) {
[2]  Schema sh = BdFactoryImpl.eINSTANCE.createSchema();
[3]  for (Iterator it = pc.getElements().iterator() ; it.hasNext() ; ) {
```

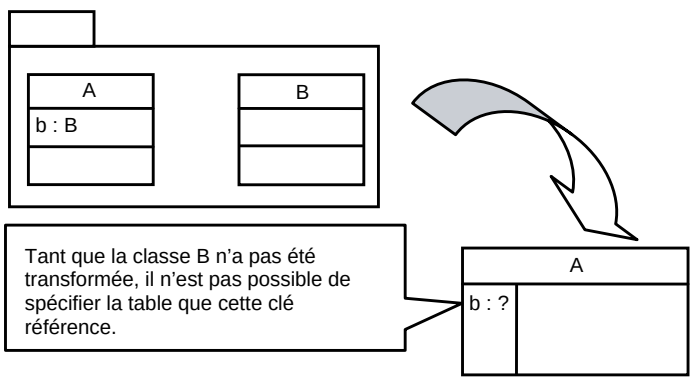
```
[4] Object next = it.next();
[5] if ( next instanceof uml.Class) {
[6]     sh.getTables().add(class2Table((uml.Class) next));
[7] }
[8] else if (next instanceof DataType) {
[9]     dataType2Type((DataType)next);
[10] }
[11]  }
[12] }
```

La méthode `class2Table` ci-dessous correspond à la deuxième règle, qui consiste à créer une table à partir d'une classe. La deuxième ligne de cette méthode permet de créer une instance de l'EClass `Table`. La troisième ligne permet de spécifier que le nom de la table correspond au nom de la classe. Les quatrième, cinquième, sixième et septième lignes permettent d'itérer sur les propriétés de la classe et d'appeler l'exécution de la règle de transformation relative aux propriétés UML.

Les lignes suivantes, assez complexes, permettent d'établir les liens entre les clés étrangères et les tables. En effet, comme notre algorithme est fondé sur une navigation descendante dans le modèle UML, il est possible d'appeler l'exécution de la règle de transformation relative à une propriété alors que le type de celle-ci n'a pas encore donné lieu à la création d'un élément lui correspondant dans le schéma.

La figure 7.5 présente un exemple dans lequel il n'est pas encore possible de lier la clé étrangère avec la table qu'elle référence car la table correspondante n'a pas encore été créée par l'algorithme.

Figure 7.5
Lien entre clé étrangère et table



Pour faire face à ce problème, nous avons fait le choix de stocker les liens restant à établir dans des tables de hachage. Ainsi, ces liens seront réalisés lorsque tous les éléments nécessaires à leur établissement auront été créés.

```
[1] public Table class2Table(uml.Class clazz) {
[2]  Table tab = BdFactoryImpl.eINSTANCE.createTable();
[3]  tab.setName(clazz.getName());
```

```
[4]  for (Iterator it = clazz.getProps().iterator() ; it.hasNext() ; ) {
[5] Property next = (Property) it.next();
[6] tab.getColonnes().add(property2Colonne(next));
[7]  }
[8]  classTable.put(clazz , tab);
[9]  if (cleRef.containsKey(clazz)) {
[10] Vector cles = (Vector) cleRef.get(clazz);
[11] for (Iterator it = cles.iterator() ; it.hasNext() ; ){
[12]     CléEtrangère current = (CléEtrangère) it.next();
[13]     current.setRefTable(tab);
[14] }
[15]  }
[16]  return tab;
[17] }
```

La méthode `property2Colonne` ci-dessous correspond à la troisième règle, qui consiste à créer une colonne ou une clé étrangère à partir d'une propriété d'une classe. La deuxième ligne de cette méthode permet de savoir si le type de la propriété est une classe ou un type de donnée. S'il s'agit d'une classe, il faut créer une clé étrangère tandis que s'il s'agit d'un type de donnée, il faut créer une simple colonne. La troisième et la vingt et unième lignes permettent respectivement de créer des instances de l'EClass `CléEtrangère` et de l'EClass `Colonne`. Les autres lignes permettent d'établir les liens entre les clés étrangères et leur table ou entre les colonnes et leur type.

Nous préférons ne pas nous attarder sur le fonctionnement du mécanisme permettant d'établir les liens car celui-ci n'est pas d'un grand intérêt dans le contexte de cet ouvrage et risque d'obscurcir la présentation des transformations de modèles.

```
[1] public Colonne property2Colonne(Property prop){
[2]  if (prop.getType() instanceof uml.Class){
[3]    CléEtrangère cle = BdFactoryImpl.eINSTANCE.createCléEtrangère();
[4]    cle.setName(prop.getName());
[5]    if (classTable.containsKey(prop.getType())) {
[6]      cle.setRefTable((Table) classTable.get(prop.getType()));
[7]    }
[8]    else {
[9]      if (cleRef.containsKey(prop.getType())){
[10]        ((Vector)cleRef.get(prop.getType())).add(cle);
[11]      }
[12]      else {
[13]        Vector vec = new Vector();
[14]        vec.add(cle);
[15]        cleRef.put(prop.getType() , vec);
[16]      }
[17]    }
[18]    return cle;
[19]  }
[20]  else { //DataType
[21]    Colonne col = BdFactoryImpl.eINSTANCE.createColonne();
[22]    col.setName(prop.getName());
```

```
[23]    if (datatypeType.containsKey(prop.getType())) {
[24]      col.setType((Type)datatypeType.get(prop.getType()));
[25]    }
[26]    else {
[27]      if (colType.containsKey(prop.getType())){
[28]        ((Vector)cleRef.get(prop.getType())).add(col);
[29]      }
[30]      else {
[31]        Vector vec = new Vector();
[32]        vec.add(col);
[33]        colType.put(prop.getType() , vec);
[34]      }
[35]    }
[36]    return col;
[37]  }
[38] }
```

La méthode `dataType2Type` ci-dessous correspond à la dernière règle, qui consiste à créer un type à partir d'un type de donnée. La deuxième ligne de cette méthode permet de créer une instance de l'EClass `Type`. La troisième ligne permet de spécifier que le nom du type correspond au nom du type de donnée. Les lignes suivantes servent à établir les liens entre les colonnes et leur type.

```
[1] public Type dataType2Type(DataType dt) {
[2]  Type t = BdFactoryImpl.eINSTANCE.createType();
[3]  t.setName(dt.getName());
[4]  datatypeType.put(dt , t);
[5]  if (colType.containsKey(dt)) {
[6]    Vector cles = (Vector) colType.get(dt);
[7]    for (Iterator it = cles.iterator() ; it.hasNext() ; ){
[8]      Colonne current = (Colonne) it.next();
[9]      current.setType(t);
[10]   }
[11] }
[12] return t;
[13] }
```

En résumé

Cet exemple montre qu'il est possible de développer n'importe quelle transformation de modèle en utilisant un langage de programmation orientée objet avec des interfaces de manipulation des modèles. Il montre en outre que la programmation de transformation de modèles n'est pas un exercice facile. Bien que très simple, l'exemple nécessite quelques aménagements afin de mettre en place les liens entre les éléments des modèles. Le mécanisme que nous avons utilisé pour mettre en place ces liens n'est certainement pas idéal mais fait ressortir le niveau de complexité du problème.

Précisons que l'approche par programmation est actuellement la plus utilisée car c'est la mieux supportée par les outils de développement.

Élaboration des règles par template

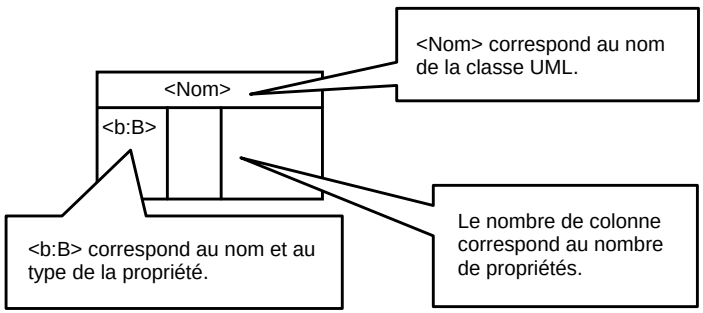
Nous avons déjà précisé que l'approche par template consistait à définir les canevas des modèles cibles souhaités en y déclarant des paramètres. Ces paramètres seront substitués par les informations contenues dans les modèles sources. Les canevas des modèles cibles sont appelés « modèles cibles paramétrés » ou « modèles templates ».

Templates et sites Web

L'approche par template est très utilisée dans les développements de sites Web avec des technologies telles que PHP ou JSP. L'idée est de définir les canevas des pages Web devant être affichées en y déclarant des paramètres. Ces paramètres sont définis à l'aide d'un langage de programmation et seront calculés dynamiquement à partir, par exemple, d'une base de données.

La figure 7.6 présente un exemple de template pour les tables des schémas de base de données. Ce template contient plusieurs paramètres, dont les valeurs seront obtenues à partir des informations contenues dans le modèle source. Le template présenté ici contient deux paramètres, l'un pour le nom de la table et l'autre pour toutes les colonnes de la table. Ces paramètres seront respectivement remplacés par le nom de la classe du modèle source et par toutes ses propriétés.

Figure 7.6
Exemple de template pour les tables de schéma de base de données



L'exemple de la figure 7.6 a pour unique fonction d'illustrer l'importance du langage permettant la définition des templates. Ce langage doit offrir un compromis entre puissance d'expression et facilité d'utilisation. Dans notre exemple, nous avons utilisé un pseudolangage dédié aux schémas de base de données. Ce langage semble facile d'utilisation mais il paraît évident que ses capacités d'expression sont limitées.

Les outils qui supportent l'approche par template proposent soit des langages graphiques spécifiques de métamodèles cibles, soit des langages textuels indépendants des métamodèles :

- Les langages graphiques spécifiques d'un métamodèle sont souvent faciles à utiliser mais très peu expressifs, étant donné leur adhérence à un unique métamodèle. Ils ne permettent la transformation que d'un seul type de modèle.

- Les langages textuels indépendants des métamodèles sont à l'inverse très expressifs mais plus difficiles à utiliser, étant donné leur caractère générique. Ils permettent la transformation de tous les types de modèles.

Le choix d'un langage dépend de l'objectif de l'utilisateur et du type des modèles qu'il souhaite transformer.

Un exemple de langage de définition de template spécifique d'un métamodèle est UML. UML permet en effet la définition de modèles UML paramétrés. Les paramètres de ces modèles seront substitués par des informations contenues dans des modèles sources. Les templates UML sont largement supportés dans les outils UML du marché et permettent la définition facile de transformations de modèles UML.

Un exemple de langage de définition de template indépendant des métamodèles est fourni par les langages fondés sur XMI et sur les langages de programmation classiques en raison des mots-clés permettant l'automatisation des traitements. L'idée est que le template soit un document XMI correspondant au modèle cible. Les paramètres de ce document XMI ainsi que l'automatisation de certains traitements s'écrivent avec un langage de programmation directement dans le document XMI. Des caractères particuliers permettent de bien identifier le langage de programmation des balises XMI.

Exemple de mise en œuvre

Nous proposons d'illustrer l'approche par template avec un langage de définition de template indépendant des métamodèles. Nous avons choisi un langage fondé sur XMI et sur Java pour les mots-clés permettant l'automatisation des traitements. Ce langage utilise une approche semblable à JSP (JavaServer Pages), qui consiste à entourer les mots-clés Java par les caractères `<%` et `%>` afin de les distinguer des balises XMI.

Plutôt que de détailler l'intégralité de la transformation, nous ne présentons qu'un modèle template permettant la génération d'une table à partir d'une classe. Nous considérons que le modèle source qui permettra de fournir les valeurs aux paramètres du template contient une classe. Cette classe est identifiée par le terme `clazz` dans le template. Ce template est présenté ci-dessous.

La première ligne contient l'élément XMI représentant une table *(voir le chapitre 5)*. Cette ligne contient en outre un paramètre qui sera remplacé par la valeur du nom de la classe contenue dans le modèle source. Nous avons utilisé ici le langage Java pour préciser que le paramètre sera remplacé par le nom de la classe (`clazz.getName()`).

Les deuxième et troisième lignes du template font appel à Java pour itérer sur toutes les propriétés de la classe contenue dans le modèle source. La quatrième ligne contient l'élément XMI représentant une colonne. Cette ligne contient aussi un paramètre qui sera remplacé par la valeur du nom de la propriété de la classe contenue dans le modèle source. La cinquième ligne contient l'élément XMI représentant le type de la colonne. Cette ligne comporte un paramètre qui sera remplacé par la valeur du nom du type de la propriété du modèle source.

```
[1] <tables name="<%clazz.getName()%>">
[2]   <%for (Iterator it=clazz.getProps().iterator() ; it.hasNext() ; ) {
[3]     Property next = (Property) it.next(); %>
[4]    <colonnes name="<%next.getName()%>">
[5]     <type name="<%next.getType().getName()%>">
[6]   </colonnes>
[7]   <%}%>
[8] </tables>
```

Cette transformation de modèle réalisée selon l'approche par template n'est pas complète. Il faudrait encore définir tous les autres modèles templates et expliquer les relations d'ordre d'exécution existant entre eux. Cette transformation, même incomplète, est toutefois suffisante pour comprendre les principes de fonctionnement de l'approche par template.

En résumé

Bien outillée, l'approche par template est très utilisée pour réaliser certaines transformations de modèles. Certains langages de définition de template, comme le langage UML, sont très aboutis et permettent de définir facilement des transformations de modèles UML. Pour autant, il est difficile d'affirmer que cette approche apporte un gain significatif par rapport à l'approche par programmation. Tout dépend en fait du langage de définition des templates.

S'il est un domaine pour lequel cette approche est indiscutable, c'est celui de la génération de texte à partir de modèle. En effet, cette approche ne nécessitant pas obligatoirement de définition des métamodèles cibles, elle est parfaitement exploitable pour générer tout document textuel à partir d'un modèle sans avoir à définir le métamodèle du texte. Cette approche est abondamment utilisée pour réaliser tous les générateurs de code et de documentation.

Élaboration des règles par modélisation

Nous avons déjà vu que l'approche par modélisation consistait à appliquer les concepts de l'ingénierie des modèles aux transformations des modèles elles-mêmes. L'idée est donc de modéliser les transformations de modèles.

Pour réaliser cette approche, l'OMG a décidé, début 2001, de lancer des travaux de standardisation afin de définir un langage permettant d'élaborer des modèles de transformation de modèles. Ces travaux de standardisation visent à définir le métamodèle MOF2.0 QVT (Query, View, Transformation), qui structure ces modèles de transformation de modèles.

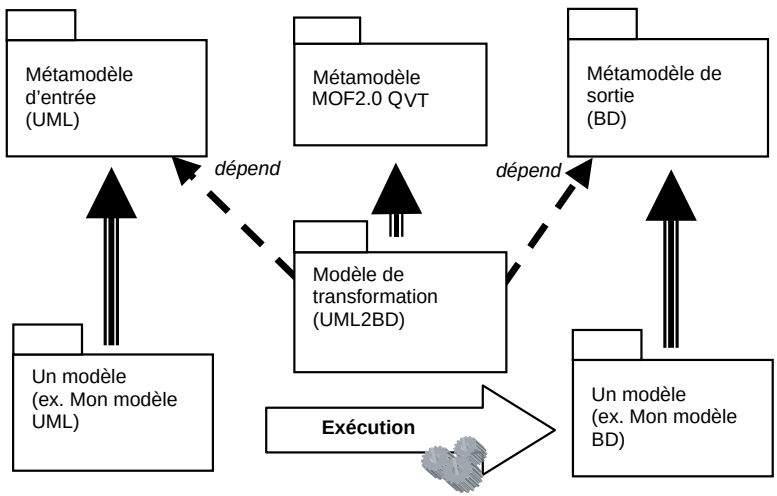
MOF2.0 QVT est toujours en phase d'élaboration, et la première version publique devrait sortir fin 2005. Plusieurs acteurs français membres de l'OMG travaillent à sa définition. Prenons le temps ici de remercier chaleureusement Mariano Belaunde (France Telecom R&D), Sébastien Gérard (CEA-List), Laurent Rioux (Thalès R&D) et Didier

Votjisek (INRIA Triskel) pour leurs contributions et pour les moments que nous avons passés et que nous passerons encore dans ces meetings OMG.

Même si ces travaux relèvent encore de la recherche, il nous paraît intéressant de les présenter car l'approche par modélisation sera sans doute incontournable d'ici quelques mois.

La figure 7.7 illustre cette approche. La définition de la transformation de modèles est un modèle structuré selon le métamodèle MOF2.0 QVT. Les modèles instances du métamodèle MOF2.0 QVT expriment des règles de correspondance structurelles entre les métamodèles source et cible d'une transformation. Ce modèle est un modèle pérenne et productif, qu'il faut transformer pour permettre l'exécution de la transformation sur une plate-forme d'exécution.

Figure 7.7
Approche par modélisation (MOF2.0 QVT)



La figure 7.8 donne une vision simplifiée du futur métamodèle MOF2.0 QVT, dans laquelle nous avons représenté les métaclasses de QVT nécessaires à la compréhension de la philosophie de ce standard.

La métaclasse `Module` représente une transformation de modèles. Cette métaclasse est reliée à la métaclasse `Package` du métamodèle MOF2.0 par deux méta-associations (source et cible). Ces méta-associations permettent de spécifier quels sont les métamodèles source et cible de la transformation. Cette métaclasse est aussi reliée par des méta-associations aux métaclasses `Query`, `Relation` et `Mapping`. Ces liens permettent de spécifier l'ensemble des requêtes, règles de correspondance et règles de construction qui composent une transformation.

La métaclasse `Query` représente les requêtes effectuées dans une transformation de modèles. Les requêtes QVT sont principalement des expressions OCL *(voir le chapitre 4)* permettant de naviguer dans un modèle afin d'obtenir certaines informations. Les requêtes sont sans effet de bord, c'est-à-dire qu'elles ne modifient pas les modèles sur lesquels elles sont exécutées.

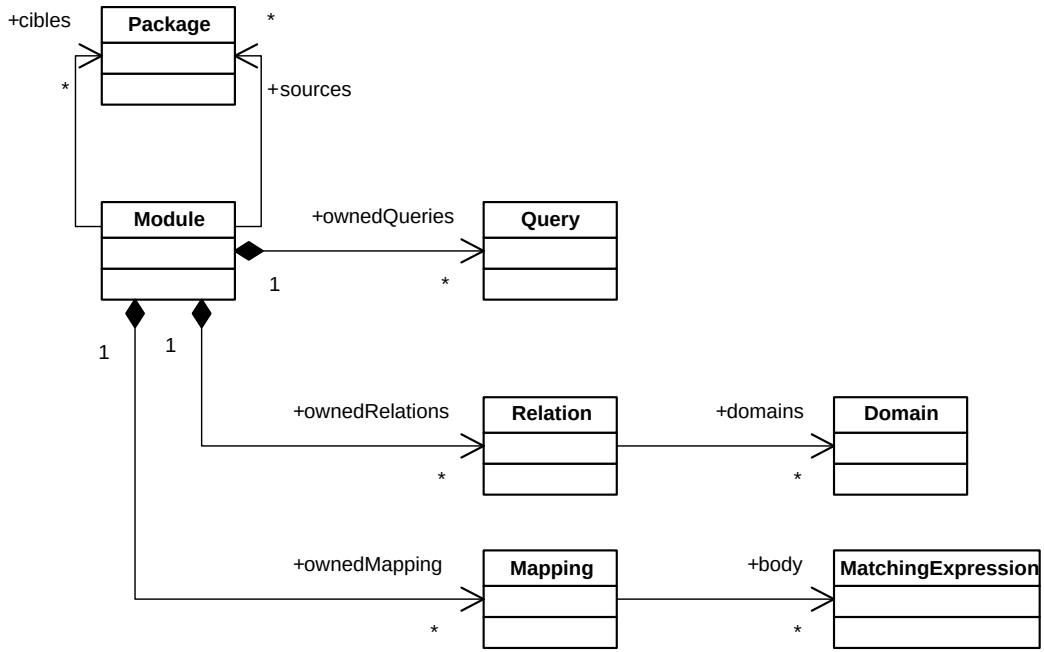


Figure 7.8
Vision simplifiée du futur métamodèle MOF2.0 QVT

La métaclasse `Relation` représente des règles de correspondance entre des parties structurelles des métamodèles source et cible. Cette métaclasse est reliée à la métaclasse `Domain`, qui représente une sous-partie structurelle d'un métamodèle. Plus précisément, un domaine est un sous-ensemble d'un métamodèle. Un domaine doit être composé d'au moins une métaclasse du métamodèle et peut contenir jusqu'à l'intégralité des métaclasses du métamodèle, avec les méta-associations qui les relient.

Les relations ne définissent pas de construction d'éléments de modèle. Elles expriment uniquement des correspondances structurelles entre des métamodèles sans définir la façon dont sont réalisées ces correspondances. On peut comparer ces relations à des règles de programmation déclarative, dans le sens où elles permettent la déclaration de correspondances structurelles et non pas la réalisation de ces correspondances.

La métaclasse `Mapping` représente des règles de construction. Cette métaclasse est reliée à la métaclasse `MatchingExpression`, qui représente le concept d'action de construction. Contrairement aux relations, les mappings spécifient des correspondances structurelles entre métamodèles en expliquant la façon dont doivent être réalisées ces correspondances. Les mappings peuvent être comparés à des instructions de programmation impérative puisqu'ils définissent un ensemble d'actions à réaliser pour établir une correspondance structurelle.

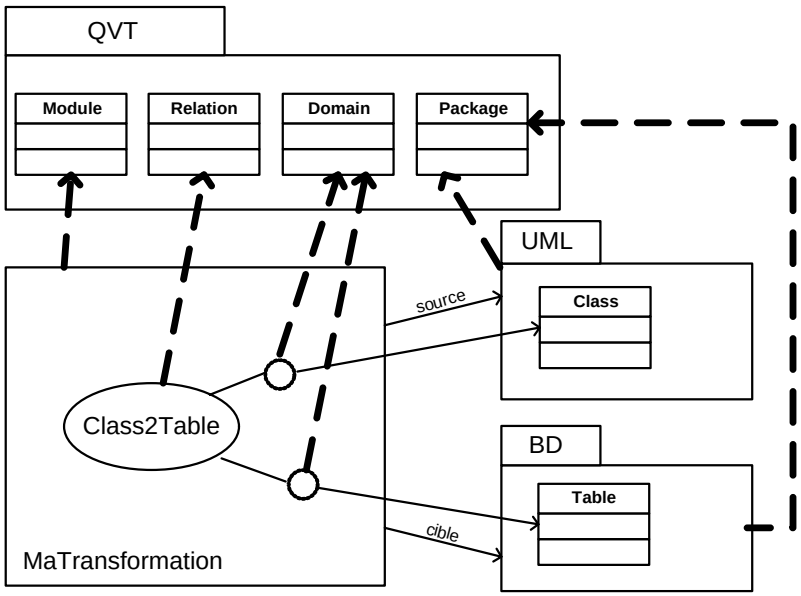
Pour résumer ce qu'exprime ce métamodèle, on peut dire qu'une transformation QVT s'écrit dans un module. Ce module, qui peut contenir des requêtes écrites dans un langage proche d'OCL, peut aussi contenir des règles de transformation. Ces dernières sont soit des règles de correspondance (approche déclarative) soit des règles de construction (approche impérative). Le standard rend possible la construction d'une transformation par une approche hybride, contenant à la fois des règles de correspondance et des règles de construction.

Exemple de mise en œuvre

Cet exemple met partiellement en œuvre l'approche par modélisation. Nous avons choisi d'illustrer uniquement la partie déclarative du métamodèle MOF2.0 QVT et avons modélisé de façon déclarative les règles de transformation. Le modèle de la transformation ne fait qu'exprimer des relations structurelles entre les métamodèles source et cible. Ces relations devront être traduites ou interprétées par un moteur de transformation, qui permettra l'exécution de la transformation.

La figure 7.9 illustre une partie du modèle de la transformation UML vers BD. Nous avons utilisé une notation graphique, qui, sans être aucunement standard, permet de bien représenter les liens entre le modèle de la transformation, ses métamodèles source et cible et le métamodèle MOF2.0 QVT.

Figure 7.9
Une partie de la transformation réalisée par l'approche par modélisation



Les liens entre les éléments du modèle de la transformation et le métamodèle MOF2.0 QVT sont représentés par des flèches en pointillé. Le module de la transformation est

représenté par un carré. Ce carré représente une instance de la métaclasse `Module` du métamodèle MOF2.0 QVT. Ce carré est lié aux packages UML et BD par deux liens nommés `source` et `cible`. Les packages UML et BD sont d'ailleurs des métamodèles instances de la métamétaclasse `Package` de MOF. Ces liens permettent d'exprimer que les modèles sources de la transformation sont conformes au métamodèle UML et que les modèles cibles de la transformation sont conformes au métamodèle BD.

Le module de la transformation ne contient qu'une relation, représentée graphiquement par une ellipse. Cette relation, qui est une instance de la métaclasse `Relation`, représente la règle de la transformation, qui consiste à faire correspondre une table à une classe UML. Cette relation référence donc deux domaines, représentés graphiquement par des cercles. Ces domaines font respectivement référence aux métaclasses `Class` et `Table` des métamodèles UML et BD.

Le modèle de la transformation UML vers BD n'est pas complet. Il faudrait en effet modéliser de la même façon les autres règles de la transformation. Cette partie du modèle permet néanmoins de bien illustrer l'approche par modélisation.

En résumé

L'approche par modélisation a pour ambition d'appliquer les principes MDA aux transformations de modèles. L'objectif est bien évidemment de rendre les transformations de modèles pérennes et productives et de les définir en fonction des plates-formes d'exécution.

Le futur standard MOF2.0 QVT est une première réponse à cet objectif. Ce métamodèle définit la structure des modèles de transformation de modèle. L'approche MOF2.0 QVT est très intéressante en ce qu'elle permet de modéliser les transformations selon deux styles communément utilisés, le style déclaratif et le style impératif.

Étant donné le caractère stratégique des transformations de modèles, plusieurs acteurs de l'OMG se sont livrés et se livrent encore à des débats quant à la définition de ce standard. Cela explique en partie pourquoi, alors que les premiers travaux ont démarré il y a plus de deux ans, il n'y a pas encore de version stable et publique du standard. On peut penser que les premières versions outillées de ce futur standard verront le jour d'ici un an ou deux.

Synthèse

Ce chapitre a insisté sur l'importance des transformations de modèles dans MDA. Celles-ci sont stratégiques, car elles apparaissent dans toutes les étapes de l'approche et permettent l'établissement de liens de traçabilité entre les modèles.

Nous avons ensuite précisé ce qu'étaient les transformations de modèles dans le contexte MDA. Celles-ci sont comparables à des fonctions dont les paramètres d'entrée et de sortie sont des modèles structurés par des métamodèles. C'est pourquoi la génération de code et la génération de documentation ne sont pas considérées comme étant des transformations de modèles.

Pour finir, nous avons présenté les différentes approches permettant de réaliser des transformations de modèles. L'approche par programmation consiste à utiliser les langages de programmation orientée objet avec des interfaces de manipulation de modèles. L'approche par template consiste à utiliser des langages spécifiques permettant la définition de modèles cibles paramétrés en fonction des modèles sources. L'approche par modélisation consiste à modéliser les transformations de modèles.

Plusieurs travaux ont montré que ces trois approches ont à peu près la même puissance d'expression. En principe, n'importe quelle transformation peut être réalisée avec chacune d'elles. Ces approches ne sont donc pas comparables selon ce critère.

L'approche par programmation est sans doute la plus facile d'abord, car elle ne nécessite aucun apprentissage pour qui connaît un langage de programmation orientée objet. L'approche par template n'est pas non plus trop complexe à utiliser pour peu que l'on dispose d'un langage adéquat de définition des templates. Ces deux approches sont de plus exploitables pour exprimer des générations de code et de documentation, parfois appelées transformations de modèles vers textes.

L'approche par modélisation est la plus complexe. De l'avis des experts du domaine, c'est toutefois la plus prometteuse, car elle offre des solutions aux problèmes de pérennité des transformations et à leur réutilisation.

8

Les outils MDA

Ce chapitre présente différents outils du marché afin de bien montrer que l'approche MDA n'est pas seulement théorique et qu'elle deviendra incontournable dans les développements logiciels des prochaines années.

Nous avons sélectionné deux outils, Rational Software Modeler et Softeam MDA Modeler, car ce sont ceux que nous connaissons le mieux et qu'il nous est donc plus facile de montrer leur façon de supporter l'approche MDA. Le choix de ces outils est arbitraire et ne signifie pas que les autres outils MDA ne sont pas de bonne qualité.

Nous souhaitons exprimer ici nos plus vifs remerciements envers IBM et Softeam, qui ont relu et amendé ce chapitre, et dont l'active collaboration nous a permis de le rédiger dans des conditions plus que confortables. Que soient ici plus particulièrement remerciés S. Bonnaud, F. Grelier et J. Desquilbet d'IBM et P. Desfray et S. Ammour de Softeam.

IBM Rational Software Modeler

Après avoir acheté Rational en décembre 2002, IBM a annoncé publiquement, le 18 novembre 2004, la sortie de sa nouvelle suite d'outils de développement logiciel nommée *Atlantic*.

Atlantic représente la nouvelle génération de la plate-forme de développement logiciel d'IBM Software, dont l'objectif est de transformer et simplifier l'ensemble des activités de développement logiciel. Atlantic inclut de nouveaux produits ainsi que des améliorations significatives de produits existants. Cette nouvelle génération de produits de développement logiciel se fonde sur les standards Eclipse 3.0, UML2.0 et Hyades dans le domaine des tests, Eclipse étant le socle technologique permettant l'intégration des rôles, des fonctions, des données et des produits.

Atlantic se décline en plusieurs produits commerciaux imbriqués les uns dans les autres, à la manière des poupées russes :

- **RWD (Rational Web Developer), précédemment WebSphere Studio Site Developer.** Environnement de développement très simple d'emploi permettant de construire visuellement et sans programmation des applications Web Services, Web ou Java. Convient au développeur novice en Java.
- **RAD (Rational Application Developer), précédemment WebSphere Studio Application Developer.** Environnement de développement fondé sur Eclipse 3.0 permettant de développer pour J2EE. Le produit est adapté aux applications transactionnelles distribuées ayant des contraintes de forte montée en charge et de sécurité et pour lesquelles la puissance de J2EE est pleinement justifiée. RAD contient RWD.
- **RSM (Rational Software Modeler).** Contient tous les outils permettant l'élaboration des modèles UML2.0.
- **RSA (Rational Software Architect).** Produit le plus englobant et le plus complet de la suite Atlantic. Contient RSM et RAD.

Tous les outils de la suite Atlantic sont intégrés dans Eclipse, ce qui facilite la collaboration de plusieurs intervenants sur un même projet, quel que soit leur métier.

Dans le cadre de cet ouvrage, nous nous intéressons plus particulièrement à RSM, qui offre un support à la modélisation UML2.0. RSM permet l'élaboration de modèles UML2.0 et offre des facilités de production sur ces modèles. RSM permet, entre autres, la génération de code et de documentation ainsi que l'application de patterns.

RSM permet aux utilisateurs d'enrichir les fonctionnalités existantes et même de définir des fonctionnalités personnalisées. RSM permet ainsi la définition de transformations de modèles et de générations de textes et de patterns. Les sections qui suivent détaillent les mécanismes offerts par RSM pour réaliser ces définitions.

L'outil RSM est fourni dans le CD accompagnant cet ouvrage.

Transformation de modèles

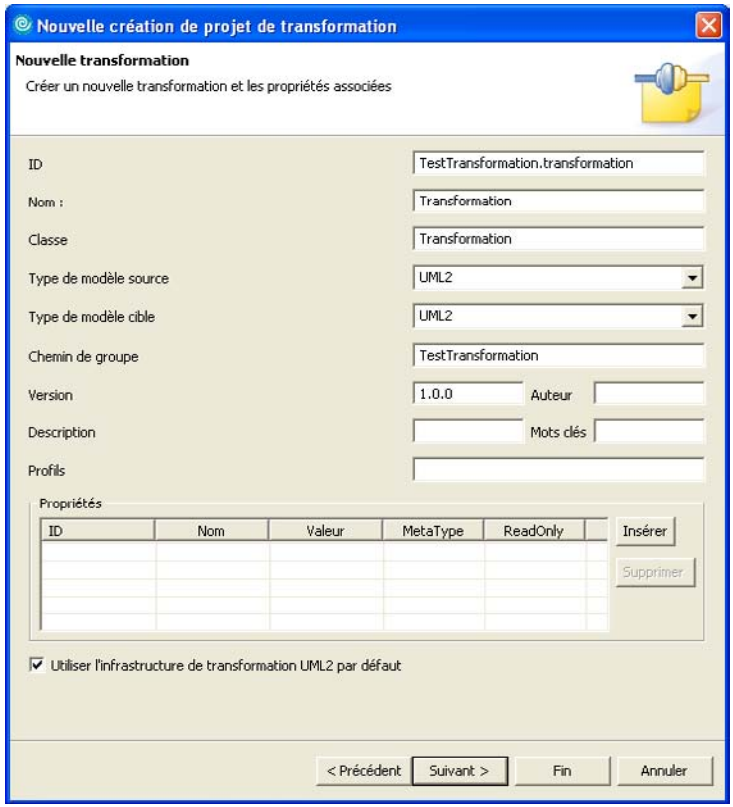
RSM permet à ses utilisateurs de définir des transformations de modèles. L'architecture de RSM lui permet de supporter n'importe quel métamodèle. Il est donc théoriquement possible de définir des transformations de modèles ayant comme sources et comme cibles des métamodèles différents. En pratique, la version de RSM que nous avons utilisée est essentiellement adaptée à la définition des transformations de modèles UML, qu'ils soient profilés ou non.

La création d'une nouvelle transformation de modèle se fait dans RSM par le biais de la construction d'un nouveau projet de transformation (projet de plug-in avec l'option de création de transformations). Un assistant facilitant la création des projets de transformation permet de spécifier les informations obligatoires nécessaires à la définition des

transformations, telles que l'identifiant unique de la transformation, son nom, sa classe d'implémentation et ses métamodèles source et cible.

La figure 8.1 illustre la construction par cet assistant d'un exemple de transformation de modèles que nous avons réalisé. Cet exemple consiste simplement à construire des accesseurs pour chaque attribut d'une classe (opérations d'accès en lecture et en écriture, aussi appelées getters et setters).

Figure 8.1
Création d'un projet de transformation de modèles



Après avoir construit un projet de transformation de modèles, la définition d'une transformation de modèles passe par la spécification de l'ensemble des règles qui compose la transformation. Chaque règle porte sur un unique type d'élément UML, par exemple, classe, package, cas d'utilisation, etc., et dispose de sa propre classe d'implémentation. C'est d'ailleurs dans cette classe que seront codées en Java les transformations effectuées sur les modèles.

La figure 8.2 illustre la seule règle de notre transformation. Cette règle porte sur les classes UML (instances de la métaclasse `Class`). Elle ne peut donc être appliquée que sur les classes UML. La classe d'implémentation de cette règle est la classe `ClassRule`. C'est dans cette classe que seront codées les transformations effectuées par cette règle.

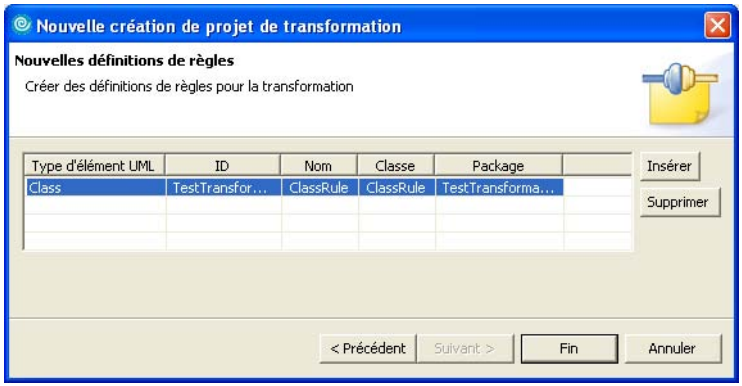


Figure 8.2
Règles de la transformation de modèles

La définition d'une transformation de modèles passe par le codage des règles de transformation. Ce codage se fait dans chaque classe d'implémentation des règles de la transformation. Le code est du Java, et il utilise les interfaces de manipulation de modèles UML proposées par RSM. Ces interfaces sont des interfaces spécifiquement adaptées, générées grâce à EMF sur le métamodèle UML2.0. Elles sont Open Source.

Les lignes de code suivantes correspondent à la règle de notre transformation exemple. Elles appartiennent donc à la classe `ClassRule`. Elles permettent la construction des accesseurs pour chaque attribut de la classe. Nous ne jugeons pas nécessaire d'expliquer ce code, car il n'est pas d'un grand intérêt pour l'objectif de ce chapitre, qui est d'illustrer le niveau de support de MDA dans les outils du marché.

```
public Object createTarget(ITransformContext ruleContext) {
org.eclipse.uml2.Class param = (org.eclipse.uml2.Class)ruleContext.getSource();
for (Iterator it = param.getAttributes().iterator() ; it.hasNext() ;) {
org.eclipse.uml2.Property current = (org.eclipse.uml2.Property) it.next();
org.eclipse.uml2.Operation operaGetter = param.createOwnedOperation
➥(UML2Package.eINSTANCE.getOperation());
operaGetter.setName("get"+current.getName());
org.eclipse.uml2.Operation operaSetter = param.createOwnedOperation
➥(UML2Package.eINSTANCE.getOperation());
operaSetter.setName("set"+current.getName());
}
return param;
}
```

Après avoir défini intégralement la transformation, RSM permet de packager le projet de transformation sous forme de plug-in afin de le rendre utilisable dans n'importe quel projet de modélisation.

Les figures 8.3 et 8.4 illustrent l'utilisation de cette transformation dans un projet de modélisation. La figure 8.3 représente le modèle source auquel sera appliquée la transformation et la figure 8.4 le modèle cible après exécution de la transformation.

L'insertion du menu contextuel permettant l'exécution de la transformation par l'utilisateur est effectuée automatiquement par RSM lors de la création du projet de transformation de modèles.

Figure 8.3
Application de la transformation de modèles

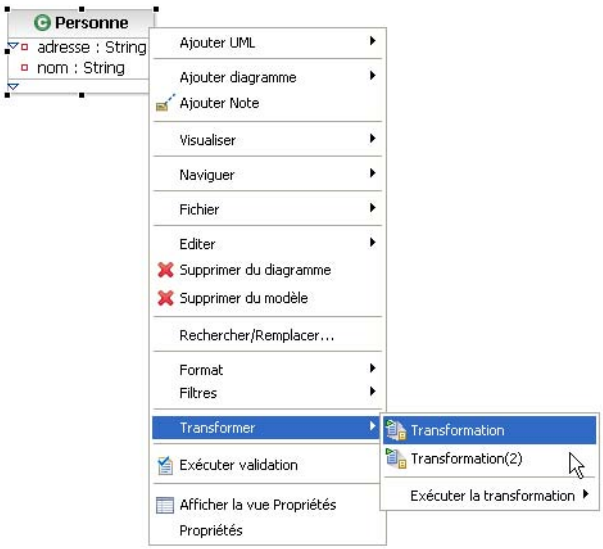
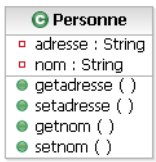


Figure 8.4
Résultat de la transformation de modèles



Nous venons de voir que RSM fournissait des mécanismes pour définir des transformations de modèles. Ces mécanismes consistent en un framework technique Java et en plusieurs assistants graphiques. Le framework technique permet à l'utilisateur souhaitant définir ses propres transformations de modèles de ne coder que les parties effectuant réellement ces transformations. Le code relatif à la mise en œuvre des transformations et à leur présentation dans les outils n'est pas à sa charge. Les assistants graphiques facilitent la prise en main par les utilisateurs de l'outil RSM pour la définition de leurs propres transformations.

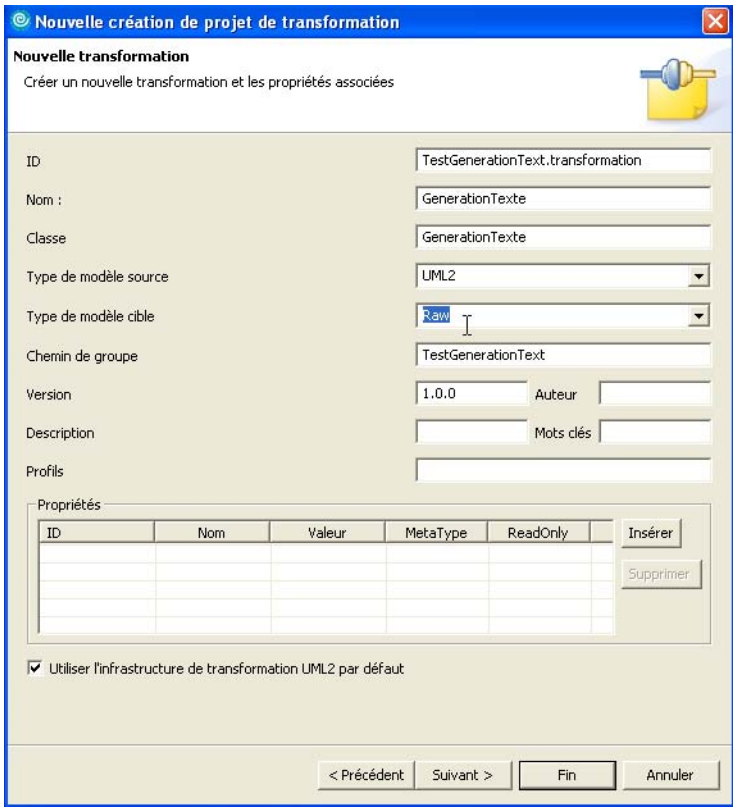
Génération de texte

RSM permet à ses utilisateurs de définir des générations de texte à partir de modèles. Le moyen de définir des générations de texte est similaire à celui employé pour définir des transformations de modèles.

La première étape consiste à construire un nouveau projet de transformation. La différence avec la transformation de modèles consiste à spécifier le fait que le métamodèle cible n'est pas un réel métamodèle mais une simple suite de lignes de texte. Cela se fait en choisissant Raw comme cible dans l'assistant de création de projet de transformation.

La figure 8.5 illustre l'utilisation de l'assistant de création de projet de transformation pour notre propre génération de texte. Celle-ci consiste simplement à lister sous forme textuelle le nom d'une classe ainsi que les noms de chacun de ses attributs. Nous constatons que le métamodèle cible est bien Raw.

Figure 8.5
Création d'une génération de texte



La suite des étapes de définition d'une génération de texte est en tout point similaire à celles de la définition d'une transformation de modèles. La deuxième étape consiste donc à spécifier l'ensemble des règles constituant la génération de texte. Ces règles sont similaires aux règles constituant les transformations de modèles. Elles portent sur un unique type d'élément UML, par exemple classe, package, cas d'utilisation, etc., et disposent de leur propre classe d'implémentation.

La figure 8.6 illustre la seule règle de notre génération de texte. Celle-ci porte sur les classes UML. Tout comme les règles de transformation de modèles, les règles de génération de texte sont codées dans des classes Java. Notre règle est implémentée dans la classe `ClassRule`.

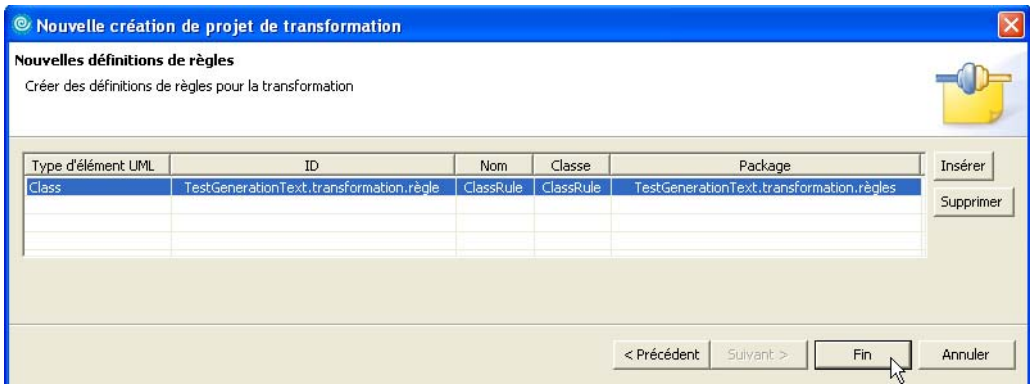


Figure 8.6
Règles de la génération de texte

Les lignes de code suivantes correspondent à notre unique règle de génération exemple. Elles appartiennent à la classe `ClassRule`. Elles permettent simplement d'afficher dans la console textuelle de RSM le nom de la classe ainsi que les noms de ses attributs. Là encore, nous ne jugeons pas nécessaire de les expliquer.

```
public Object createTarget(ITransformContext ruleContext) {
org.eclipse.uml2.Class param = (org.eclipse.uml2.Class)ruleContext.getSource();
System.out.println("la classe UML nommée "+param.getName()+" a les attributs
➥suivants:\n");
for (Iterator it = param.getAttributes().iterator() ; it.hasNext() ;) {
org.eclipse.uml2.Property current = (org.eclipse.uml2.Property) it.next();
System.out.println("\t"+current.getName()+"\n");
}
return param;
}
```

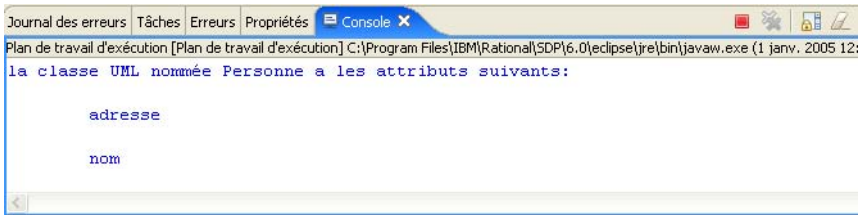
De la même manière que pour les transformations de modèles, RSM permet de packager dans des plug-in les générations de texte afin de les rendre disponibles pour n'importe quel projet de modélisation.

Les figures 8.7 et 8.8 illustrent l'utilisation de cette génération de texte dans un projet de modélisation. La figure 8.7 représente le modèle source sur lequel sera appliquée la génération de texte et la figure 8.8 le texte généré.

Tout comme dans le cas des transformations de modèles, l'insertion du menu contextuel permettant l'exécution de la génération de texte par l'utilisateur est effectuée automatiquement par RSM lors de la création du projet de transformation de modèles.

Figure 8.7
Application de la génération de texte

Figure 8.8
Résultat de la génération de texte



Nous venons de voir que RSM fournissait des mécanismes pour définir des générations de texte. La définition d'une génération de texte se fait comme celle d'une transformation de modèles. RSM a lui-même utilisé ces mécanismes de génération de texte pour offrir à ses utilisateurs les générations de code vers les langages de programmation.

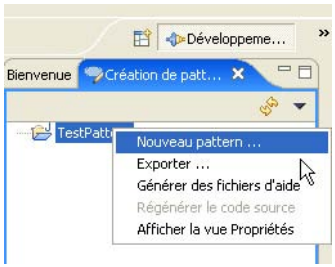
Définition de patterns UML

RSM permet à ses utilisateurs de définir leurs propres patterns UML. Les patterns UML permettent de construire automatiquement n'importe quels modèles UML. La définition d'un nouveau pattern dans un projet quelconque se fait directement grâce à un assistant de création de nouveaux patterns.

La figure 8.9 illustre l'utilisation que nous avons faite de cet assistant pour construire notre pattern exemple. Ce pattern vise à construire, dans une classe existante, les opérations d'accès en lecture et en écriture pour chaque attribut de la classe.

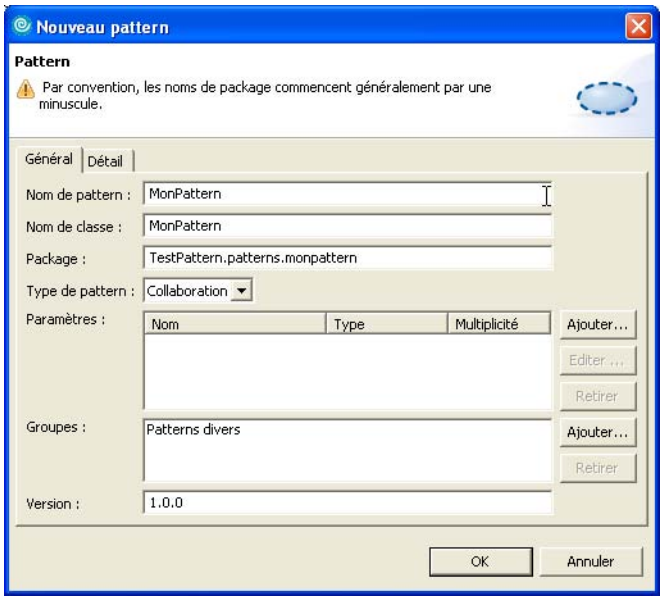
Dans RSM, la définition d'un nouveau pattern nécessite de spécifier des informations obligatoires, telles que le nom du pattern et celui de la classe d'implémentation du pattern. Le nom du pattern permet d'identifier un pattern lors de son utilisation. La classe d'implémentation du pattern contient le code Java réalisant le pattern.

Figure 8.9
Création d'un nouveau pattern



La figure 8.10 illustre l'utilisation que nous avons faite de l'assistant permettant de renseigner ces informations pour la construction de notre pattern exemple. On voit que le nom du pattern est `MonPattern` et que sa classe d'implémentation se nomme `MonPattern`.

Figure 8.10
Définition du nouveau pattern



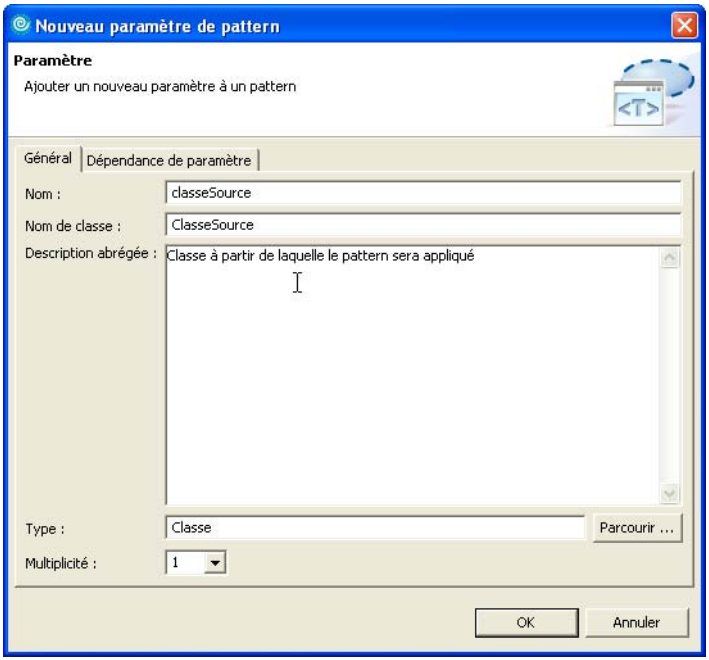
Après avoir renseigné toutes les informations nécessaires à la définition d'un nouveau pattern, RSM demande la spécification des paramètres du pattern. C'est grâce aux paramètres qu'il est possible de transmettre les valeurs nécessaires lors de l'application du pattern.

Dans notre exemple, nous avons défini un seul paramètre. Celui-ci permet de transmettre la classe sur laquelle le pattern sera appliqué afin d'y construire les opérations d'accès pour chacun de ses attributs.

La figure 8.11 illustre l'utilisation que nous avons faite de l'assistant permettant de spécifier les paramètres dans le cas de notre pattern exemple. Nous constatons la spécification

d'un unique paramètre, nommé `classSource`. Dans RSM, chaque paramètre dispose d'une classe d'implémentation. Dans notre cas, il s'agit de la classe `ClassSource`.

Figure 8.11
Ajout d'un paramètre au nouveau pattern



Pour terminer la définition d'un pattern, il suffit de fournir le code Java réalisant le pattern. Les lignes de code suivantes permettent la réalisation de notre pattern exemple. Ces lignes appartiennent à la classe d'implémentation du pattern, la classe `MonPattern`. Là encore, nous ne jugeons pas nécessaire d'expliquer le détail de ce code.

```
public boolean expand(PatternParameterValue value) {
org.eclipse.uml2.Class param = (org.eclipse.uml2.Class)value.getValue();
for (Iterator it = param.getAttributes().iterator() ; it.hasNext() ;) {
org.eclipse.uml2.Property current = (org.eclipse.uml2.Property) it.next();
org.eclipse.uml2.Operation operaGetter = param.createOwnedOperation
➥(UML2Package.eINSTANCE.getOperation());
operaGetter.setName("get"+current.getName());
org.eclipse.uml2.Operation operaSetter = param.createOwnedOperation
➥(UML2Package.eINSTANCE.getOperation());
operaSetter.setName("set"+current.getName());
}
return true;
}
```

Une fois le pattern entièrement défini, RSM permet son application dans n'importe quel projet de modélisation. Pour appliquer un pattern, RSM offre une représentation graphique

du pattern. La figure 8.12 illustre un modèle contenant une représentation graphique de notre pattern exemple. Grâce à cette représentation graphique, il est ensuite possible d'attribuer une valeur aux paramètres du pattern. Dès que des valeurs sont attribuées aux paramètres du pattern, celui est appliqué par RSM.

La figure 8.13 illustre le même modèle après avoir précisé que la classe `Personne` devait prendre la valeur du paramètre du pattern. Nous constatons que la classe contient les opérations d'accès aux attributs.

Figure 8.12
Application du nouveau pattern

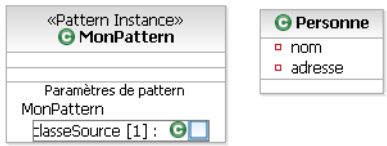
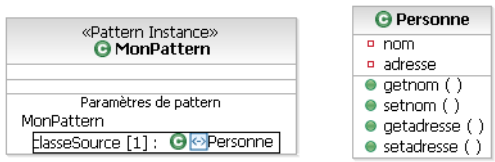


Figure 8.13
Résultat de l'application du nouveau pattern



Nous venons de voir que RSM fournissait des mécanismes pour définir des patterns UML. La définition de ces patterns se fait grâce à des assistants et en codant en Java la réalisation du pattern. RSM a d'ailleurs lui-même utilisé ces mécanismes pour offrir à ses utilisateurs un support des patterns bien connus tels que ceux du GOF (Gang of Four).

En résumé

RSM propose plusieurs mécanismes pour définir des opérations sur les modèles et ainsi rendre les modèles productifs. En nous appuyant sur la présentation de quelques-uns de ces mécanismes, nous pouvons dire que RSM adhère aux principes de l'approche MDA. Il permet notamment de définir des opérations sur les modèles et de les rendre exploitables pour les utilisateurs. Les modèles sont au centre de cet outil.

IBM met d'ailleurs fortement en avant sa volonté de suivre l'approche MDA. Ce message est rappelé dans toutes les présentations des produits IBM Rational.

Actuellement, toutes les définitions des opérations sur les modèles proposées par RSM suivent l'approche par programmation que nous avons présentée au chapitre précédent. S'il faut toujours coder dans un langage de programmation les traitements réalisés par ces opérations, IBM améliore cette situation dans ses laboratoires et travaille fortement sur l'approche par template mais aussi et surtout sur l'approche par modélisation. IBM participe de fait activement aux travaux de standardisation MOF2.0 QVT *(voir le chapitre 7).*

Softeam MDA Modeler

Depuis 1992, Softeam n'a de cesse de faire évoluer son atelier de génie logiciel, appelé Objecteering. Fondé initialement sur la méthode Classe/Relation, celui-ci a intégré UML en 1996 et a été le premier outil à supporter le concept de profil UML en 2000.

Objecteering est composé de deux outils, Objecteering Modeler et Objecteering Profile Builder. Le premier est un outil de support à la modélisation permettant à ses utilisateurs d'élaborer des modèles UML et d'appliquer des opérations telles que la génération de documentation ou de code. Le second est un outil permettant à ses utilisateurs d'élaborer des opérations sur les modèles *via* les concepts de profil UML et de module Objecteering.

UML Modeler et MDA Modeler sont les nouvelles versions de ces outils réalisées dans l'optique d'un support de l'approche MDA :

- **UML Modeler.** Permet à ses utilisateurs d'élaborer des modèles UML et d'appliquer des opérations de production sur ceux-ci (génération de documentation, génération de code, etc.). Les utilisateurs sont idéalement les développeurs ou les chefs de projet. Leur objectif est de pouvoir utiliser dans leurs projets des savoir-faire existants et de tirer rapidement et facilement profit des techniques de modélisation.
- **MDA Modeler.** Permet à ses utilisateurs de définir des opérations de production sur les modèles. Les utilisateurs sont idéalement les ingénieurs qualité, architectes, ingénieurs méthodes ou certains chefs de projet. Leur objectif est de capitaliser leurs savoir-faire ou ceux de leur entreprise et de les institutionnaliser afin que ceux-ci puissent être facilement utilisés et rapidement exploités.

La construction de composants MDA s'effectue elle-même par modélisation, en limitant au minimum les travaux de programmation.

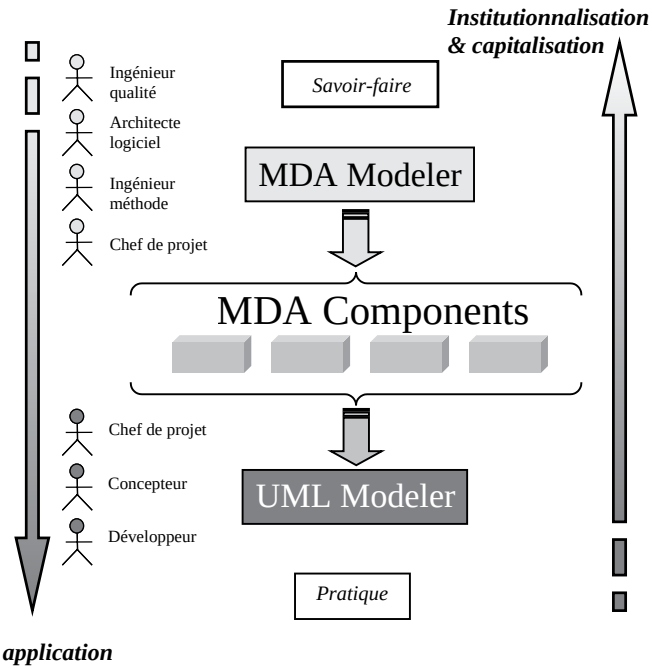
L'intégration de ces deux outils est réalisée grâce au concept de composant MDA. Un composant MDA contient des savoir-faire et automatise leur exploitation. Dans cette optique, MDA Modeler permet la création de composants MDA tandis qu'UML Modeler permet l'utilisation et la mise en exploitation de ceux-ci.

La figure 8.14 illustre cette vision avec les outils MDA Modeler et UML Modeler et leur intégration grâce au concept de composant MDA.

Dans le cadre de cet ouvrage, il est évident que nous sommes plus particulièrement intéressés par l'outil MDA Modeler et par les mécanismes qu'il propose à ses utilisateurs pour définir des opérations de production sur les modèles. Les sections suivantes présentent quelques-uns de ces mécanismes.

L'outil MDA Modeler ainsi qu'une démonstration automatique de ses fonctionnalités sont fournis sur le CD d'accompagnement de cet ouvrage.

Figure 8.14
Principes de MDA Modeler et UML Modeler



Transformation de modèles par programmation

MDA Modeler permet à ses utilisateurs de définir des transformations de modèles. Bien que son architecture lui permette de supporter plusieurs métamodèles, MDA Modeler ne supporte actuellement que le métamodèle UML. Il est donc possible de définir des transformations de modèles UML vers UML (profilés ou non).

La création d'une transformation de modèle se fait dans MDA Modeler par le biais de la création d'un nouveau profil. Un assistant facilitant la création de nouveau profil permet de renseigner les informations obligatoires du profil, telles que son nom et son métamodèle de référence (Objecteering 5.2.2 pour le métamodèle UML1.4).

La figure 8.15 illustre l'utilisation que nous avons faite de cet assistant pour créer notre propre transformation exemple. Cette transformation consiste à créer, dans une classe UML, les opérations d'accès en lecture et en écriture pour chacun des attributs de la classe. MDA Modeler offre un support graphique de modélisation des profils, représentant, par exemple, les métaclasses référencées, ainsi que les stéréotypes utilisés et leurs propriétés et associations.

Après la création d'un profil, la définition d'une transformation de modèles nécessite de référencer toutes les métaclasses UML concernées par la transformation. Ces références contiendront le code de la transformation de modèles. Ce code est écrit en J, un langage propriétaire de l'atelier, qui permet la manipulation des modèles UML.

La figure 8.16 illustre le profil de notre transformation exemple référençant la métaclasse `Class`. En effet, étant donné que notre transformation exemple cible les classes UML, seule la référence vers la métaclasse `Class` est nécessaire.

Figure 8.15
Création d'un nouveau profil

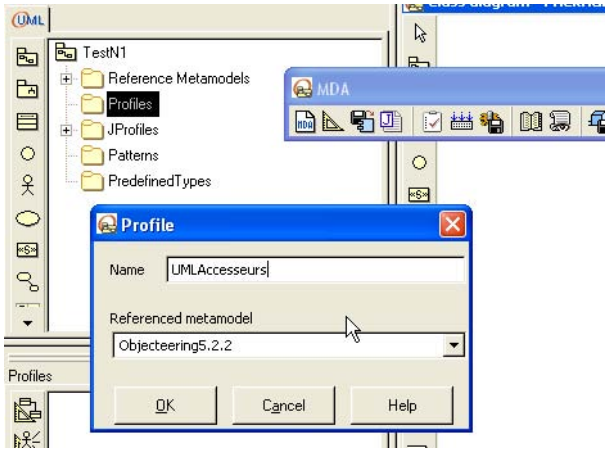


Figure 8.16
Référence de la métaclasse Class

Pour terminer la définition d'une transformation de modèles, il faut écrire le code réalisant la transformation. Ce code, écrit en J, est partagé dans plusieurs méthodes, chacune rattachée à une métaclasse référencée par le profil.

Pour notre transformation exemple, nous avons fait le choix de la coder dans une unique méthode rattachée à la métaclasse `Class`. Le code suivant présente cette méthode. Nous préférons ne pas détailler le code J car cela ne serait pas d'un grand intérêt dans le contexte de cet ouvrage.

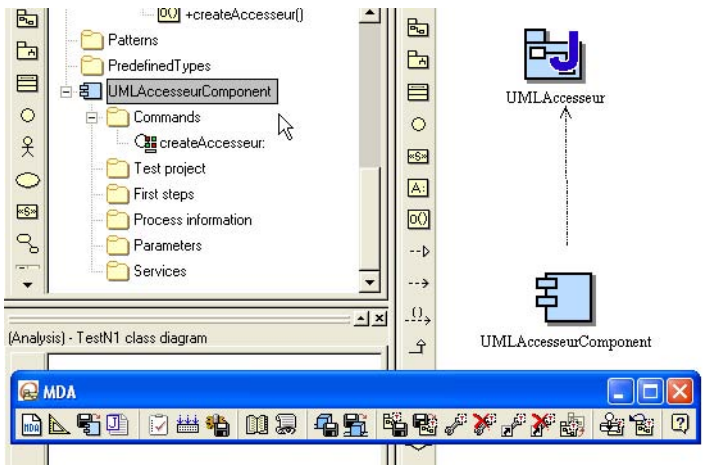
```
Operation M;
Class C = this;
sessionBegin ("addAccessor", true);
PartAttribute {
        M = Operation.new;
        M.setName("set_" + Name);
        C.appendPart(M);
        M = Operation.new;
        M.setName("get_" + Name);
        C.appendPart(M);
}
sessionEnd();
```

Une fois la transformation réalisée, MDA Modeler permet son intégration dans un composant MDA. Il est possible d'intégrer plusieurs transformations dans un unique composant MDA. L'objectif de cette étape est de permettre l'exploitation du composant MDA, et donc de la transformation de modèles, dans l'outil UML Modeler.

Un assistant facilite la création d'un composant MDA ainsi que l'intégration des transformations dans celui-ci. Par ailleurs, MDA Modeler permet de créer des entrées de menu graphique et de les lier aux transformations définies. Ces entrées de menu permettent de rendre les transformations accessibles aux utilisateurs.

La figure 8.17 illustre l'utilisation que nous avons faite de cet assistant pour créer un composant MDA et y intégrer notre transformation.

Figure 8.17
Création d'un composant MDA



Pour utiliser ce composant MDA, l'utilisateur de l'outil UML Modeler doit demander son chargement dans son propre projet de modélisation. Après avoir effectué le chargement du composant MDA, il peut utiliser toutes les opérations fournies par celui-ci.

La figure 8.18 illustre l'assistant permettant de charger un composant MDA dans l'outil UML Modeler et la figure 8.19 la façon d'utiliser ce composant.

Nous venons de voir que MDA Modeler fournissait des mécanismes pour définir des transformations de modèles et les intégrer dans des composants MDA. Nous avons vu que la définition d'une transformation passait par la création d'un profil UML et par le codage en J des transformations.

Figure 8.18
Chargement d'un composant MDA dans UML Modeler

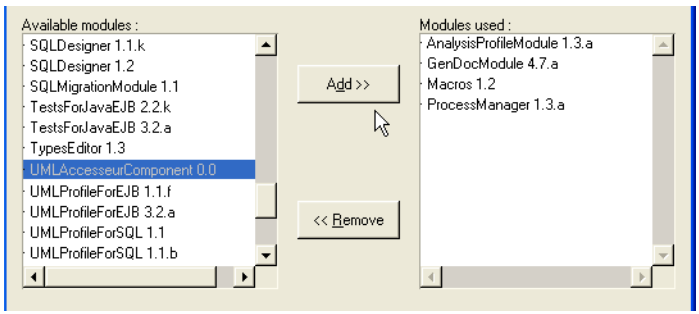
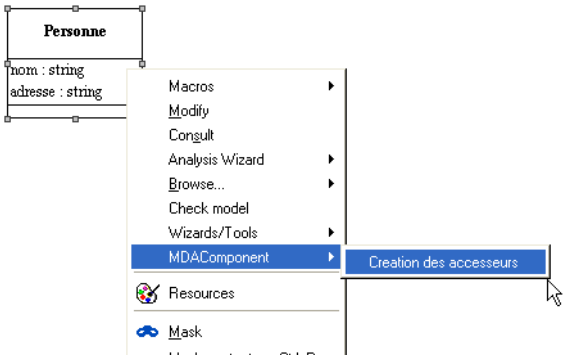


Figure 8.19
Utilisation d'un composant MDA



Génération de texte et de code

MDA Modeler permet à ses utilisateurs de définir des générations de texte à l'aide d'un mécanisme spécifique appelé template. Ce principe a pour objet de remplacer la programmation de manipulation du métamodèle par un éditeur graphique. MDA Modeler offre différents types de templates selon la cible visée (génération de document, de code ou de diagramme). Nous présentons ici les facilités offertes pour la génération de code.

De la même façon que la définition d'une transformation de modèles, la définition d'une génération de code nécessite la création d'un profil. Une fois le profil créé, la deuxième étape consiste à créer un template de code et à l'ajouter au profil. Un assistant permet de faire cela facilement.

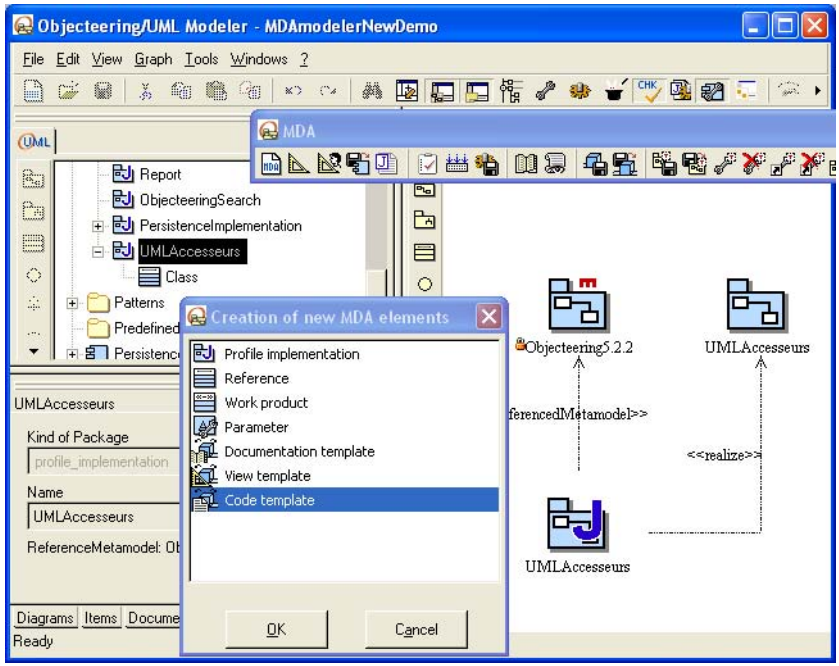
La figure 8.20 illustre l'utilisation de cet assistant pour créer un exemple de template de code permettant de lister dans un fichier le nom de la classe sur laquelle sera appliquée la génération ainsi que les noms de chacun de ses attributs avec leur type.

Dans MDA Modeler, un template de code contient la définition d'un parcours dans un modèle défini graphiquement (hiérarchiquement) par des nœuds de navigation, avec pour chacun de ces nœuds un squelette de code pouvant être appliqué sur le type d'élément UML correspondant.

Les squelettes de code sont les définitions du code à générer. Un template de code référence donc une métaclasse UML racine, ou de départ, à partir de laquelle il permet de naviguer vers chacun des éléments accessibles pour appliquer des traitements de génération spécifiques. Un assistant permet de spécifier la métaclasse du template de code, ainsi que chacun des nœuds accessibles depuis leur nœud parent.

La figure 8.21 illustre l'utilisation de cet assistant pour spécifier la métaclasse de notre template de code exemple, ici la métaclasse `Class`. Notre template de code pourra donc contenir des squelettes de code applicables à des classes UML et à chacun des éléments de modèles accessibles à partir de ces classes, comme les attributs, classes parentes, classes associées, etc.

Figure 8.20
Création d'un template de code dans un profil



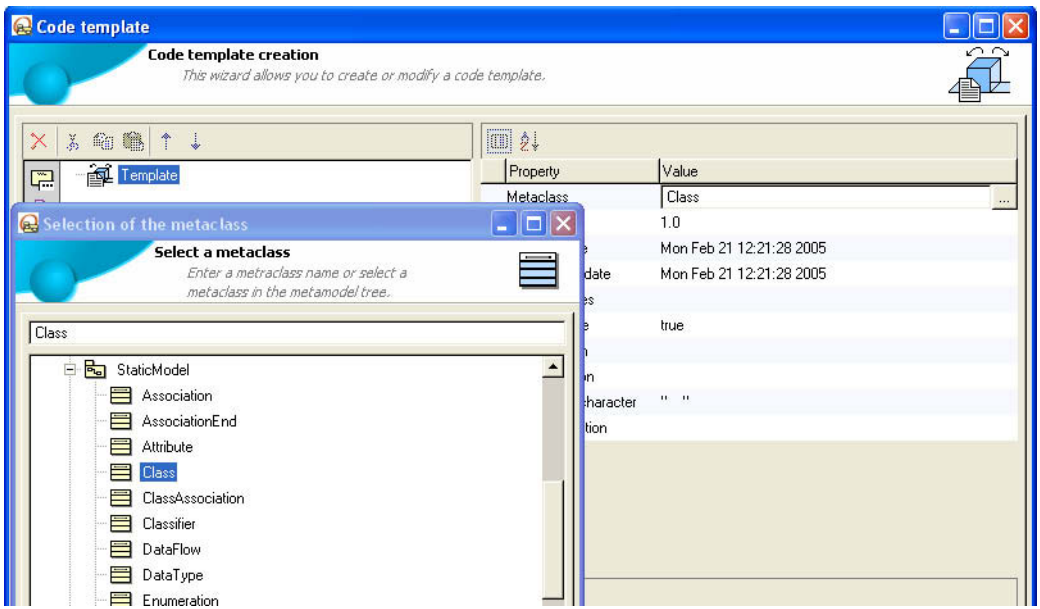
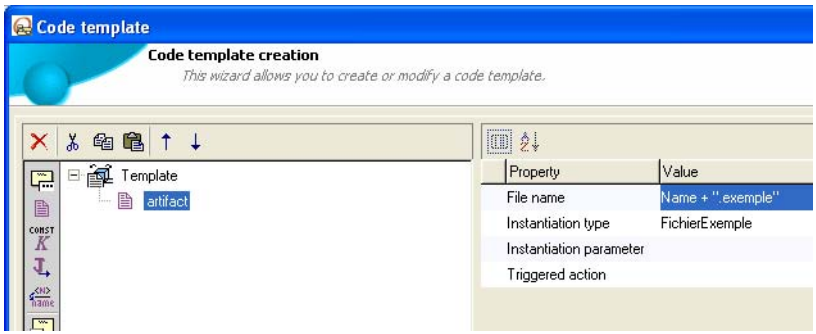


Figure 8.21
Spécification du template de code

Après avoir spécifié le template de code et avant de définir les squelettes de code, MDA Modeler permet d'associer un ou plusieurs artefacts à chaque template de code. Un artefact identifie l'élément dans lequel sera généré le code. Un artefact est très souvent un fichier, par exemple.

La figure 8.22 illustre la création d'un artefact pour notre template de code exemple. L'artefact créé ici est un fichier dont le nom correspond au nom de la classe UML à laquelle sera appliqué le template de code (`Name` d'après le métamodèle) suivi de la chaîne de caractères `".exemple"` (spécifiée grâce à l'instruction `Name + ".exemple"`).

Figure 8.22
Définition d'un artefact du template de code

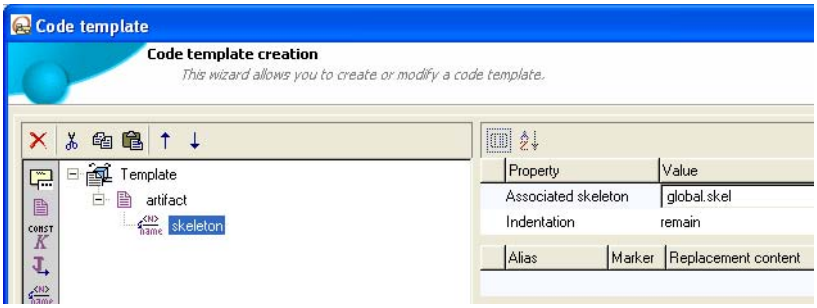


Après avoir associé un ou plusieurs artefacts à un template de code, MDA Modeler permet de spécifier graphiquement les navigations opérées sur les éléments liés à l'élément de départ (racine) puis d'attacher un squelette de code relatif à chacun de ces nœuds de navigation. Par exemple, nous pouvons attacher un squelette de code au niveau de la classe, qui référera les squelettes de code attachés aux attributs, classes parentes, etc.

Les squelettes de code contiennent les définitions du code à générer. Ils reprennent un principe connu, similaire, par exemple, aux JSP Java, qui permet d'insérer des variables ou expressions au sein d'un texte prédéfini.

La figure 8.23 illustre l'assistant permettant d'associer un squelette de code à notre exemple.

Figure 8.23
Définition d'un squelette de code

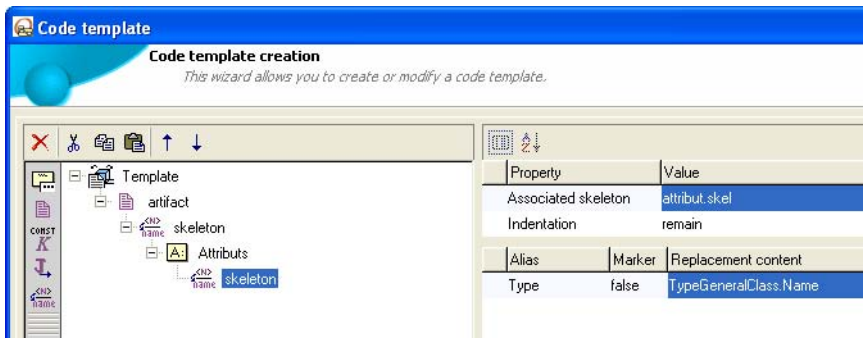


Le code suivant présente le squelette de code de notre exemple au niveau de la classe. Ce squelette permet de lister le nom de la classe UML (`Name` dans le métamodèle) suivi des

noms de ses attributs et de leur type. Dans ce squelette, `Attributs` fait référence au résultat du nœud de navigation suivant appliquant un squelette dédié aux attributs de la classe *(voir figure 8.24).*

```
La classe nommée <% Name %> a les attributs suivants :
<% Attributs %>
```

Figure 8.24
Ajout du nœud de navigation Attributs *et d'un squelette*



Le squelette attaché au nœud de navigation `Attributs` a le code ci-dessous, où `Name` est le nom de l'attribut et `Type` est associé à l'expression J `TypeGeneralClass.Name`, qui, à partir d'un attribut, va rechercher le nom du type associé (navigation dans le métamodèle) :

```
<% Name %> : <% Type %>
```

Après avoir défini intégralement le template de code, il faut l'intégrer dans un composant MDA afin que son utilisation soit possible dans l'outil UML Modeler. La création du composant MDA ainsi que l'intégration du template de code dans celui-ci se font de la même manière que pour les transformations de modèles.

Nous venons de voir que MDA Modeler permettait à ses utilisateurs de définir des générations de texte. Les mécanismes proposés par MDA Modeler sont différents selon les cibles visées. Nous avons présenté le mécanisme de génération de code qui permet de définir les squelettes des textes à générer. Un template documentaire permet de définir les documentations (Word, HTML, PDF), tandis qu'un template de diagramme permet de définir la construction de diagrammes. Dans beaucoup de cas, ces templates ne requièrent aucune programmation annexe en J.

Définition de patterns UML

MDA Modeler permet à ses utilisateurs de définir leurs propres patterns UML. Les patterns UML servent à construire automatiquement n'importe quel modèle UML.

La définition d'un nouveau pattern se fait directement dans MDA Modeler, sans nécessiter de profil. La figure 8.25 illustre l'assistant de création de nouveau pattern. Nous avons utilisé cet assistant pour créer notre pattern exemple. Celui-ci permet de créer les opérations d'accès aux attributs d'une classe UML.

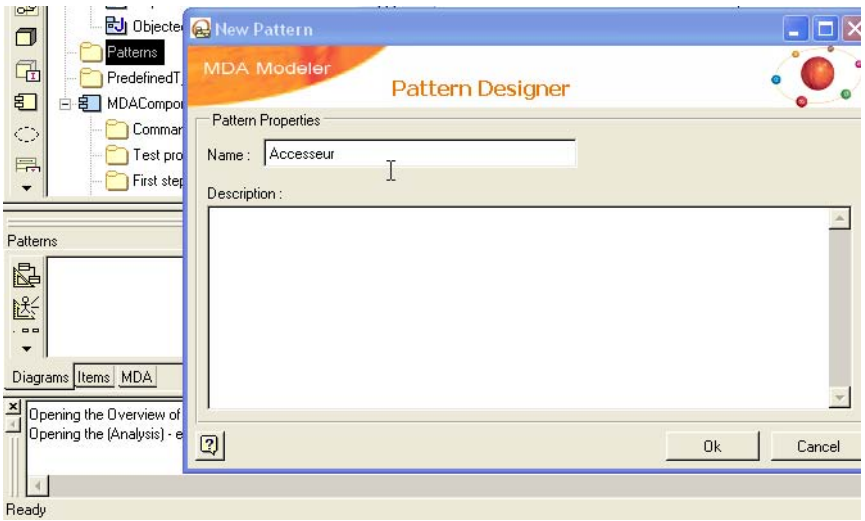


Figure 8.25
Création d'un nouveau pattern

Dans MDA Modeler, la définition d'un nouveau pattern se fait en élaborant un modèle UML et en y définissant des paramètres. Les paramètres du pattern peuvent porter sur n'importe quel élément du modèle UML (classe, attribut, opération, etc.). En plus de la définition des paramètres, MDA Modeler permet de définir des contraintes quant à l'application du pattern et des actions permettant de modifier le modèle sur lequel sera appliqué le pattern. Ces contraintes et ces actions sont spécifiées à l'aide du langage J.

La figure 8.26 illustre la définition de notre pattern exemple avec ses paramètres. Nous ne jugeons pas nécessaire dans le contexte de cet ouvrage de présenter ces paramètres ni les langages utilisés pour définir les contraintes et les actions de ce pattern.

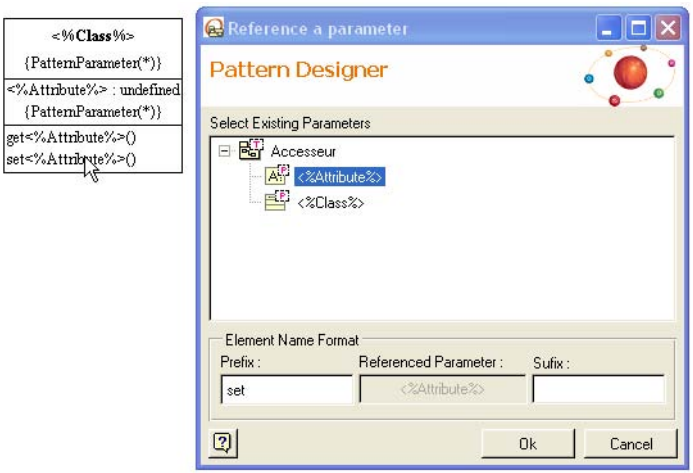
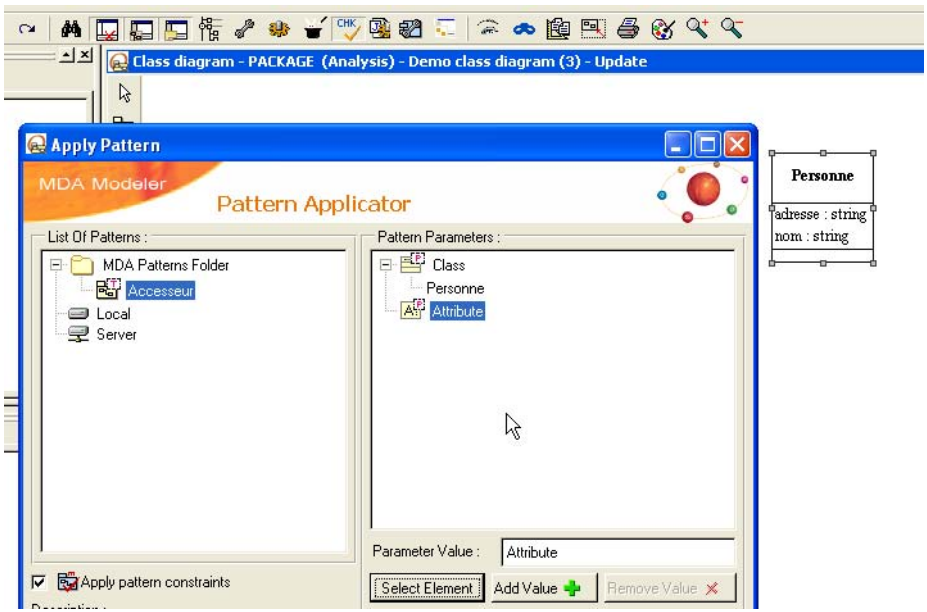


Figure 8.26
Définition des paramètres du pattern

Une fois le nouveau pattern entièrement défini, il est possible de l'intégrer dans des composants MDA et de l'utiliser dans l'outil UML Modeler.

La figure 8.27 illustre l'application de notre pattern exemple dans UML Modeler. Nous constatons que l'outil présente à l'utilisateur un assistant lui permettant de donner des valeurs à chacun des paramètres du pattern.

Figure 8.27
Application d'un pattern



Nous avons vu que MDA Modeler permettait à ses utilisateurs de définir leurs propres patterns. La définition d'un nouveau pattern se fait principalement à l'aide d'un modèle UML. Il est ainsi possible de définir des patterns sur tous les modèles UML.

Ajoutons que ce mécanisme de définition de pattern a été utilisé pour définir tous les patterns du GOF afin qu'il soit possible de les appliquer dans UML Modeler.

En résumé

MDA Modeler propose plusieurs autres mécanismes permettant de définir des opérations sur les modèles et ainsi de rendre les modèles productifs. Il propose notamment un mécanisme permettant de construire rapidement des générations de documentation.

Grâce à la présentation de ces quelques mécanismes, nous pouvons affirmer que MDA Modeler suit les principes de l'approche MDA. Il permet de définir des opérations sur les modèles et rend celles-ci exploitables par les utilisateurs. Soulignons l'importance du concept de composant MDA, qui permet de capitaliser les opérations de production sur les modèles.

Actuellement, les définitions des opérations sur les modèles proposées par MDA Modeler suivent les approches par programmation et par template, que nous avons présentées au chapitre précédent. Il est possible de coder les traitements réalisés par ces opérations dans un langage de programmation ou de les définir à l'aide de templates (squelette de code et pattern). Cela ne signifie pas que Softeam se désintéresse de l'approche par modélisation. Tout au contraire, Softeam participe activement à l'OMG à la définition du standard MOF2.0 QVT.

Synthèse

Nous venons de voir comment deux outils MDA du marché permettaient à leurs utilisateurs de construire leurs propres opérations de production sur les modèles. Rational Software Modeler et Softeam MDA Modeler permettent tous deux de définir des transformations de modèles, des générations de documents et des templates UML. Même s'ils offrent des mécanismes différents, ces deux outils suivent donc les principes de MDA en mettant les modèles au centre du cycle de développement et en faisant en sorte que ces modèles soient productifs.

Remarquons que ces outils suivent principalement l'approche par programmation et un peu moins l'approche par template. L'approche par modélisation, qui est encore essentiellement du domaine de la recherche, fait ses débuts dans les outils du marché. En tant qu'acteurs de MDA, IBM et Softeam sont pleinement impliqués dans ces recherches.

En conclusion, nous voyons bien que MDA est entré dans une phase d'industrialisation. Il ne s'agit donc en aucun cas d'une approche purement théorique ni d'un standard sans avenir. Le fait que Microsoft ait publié son approche orientée modèle, dite Software Factory, et qu'il soit en train de l'intégrer à Visual Studio renforce encore, s'il en était besoin, cette conviction. Nous remercions d'ailleurs Microsoft, avec qui nous avons eu beaucoup de contacts pour en savoir plus sur l'approche de Software Factory. Malheureusement, un problème de calendrier de rédaction nous a empêché d'en parler dans ce chapitre. Nous invitons le lecteur intéressé à se rendre sur le site *http://www.microsoft.fr/* pour en savoir plus.

Partie III

Prise en compte des plates-formes d'exécution

Les plates-formes d'exécution telles que Java, PHP et .Net ont été élaborées pour faire face à la complexité sans cesse croissante des applications informatiques. Elles fournissent des services indispensables à la construction et à l'évolution des applications, comme la gestion de la répartition, de la sécurité et des transactions.

Malgré tous les avantages qu'offrent ces plates-formes, le problème est que les applications qui les utilisent en deviennent dépendantes. Il s'ensuit de nouveaux problèmes de maintenance et d'interopérabilité, causés non par les applications mais par les plates-formes.

La prise en compte des plates-formes d'exécution dans le cycle de vie des applications a pour objectif de gérer la dépendance des applications vis-à-vis de leur plate-forme d'exécution. L'objectif est de bénéficier des services offerts par ces plates-formes, qui sont absolument nécessaires aux applications, tout en disposant de mécanismes permettant de réduire considérablement l'impact de leur dépendance.

MDA considère les plates-formes d'exécution comme des entités de première importance. Leur prise en compte s'effectue grâce à l'élaboration de modèles de plate-forme. La dépendance des applications à l'égard des plates-formes est gérée comme une transformation de modèles.

Cette partie présente tous les aspects de la prise en compte des plates-formes dans MDA. Nous expliquons tout d'abord de manière abstraite comment sont considérées les plates-formes dans MDA puis illustrons ces principes avec les plates-formes d'exécution J2EE et PHP.

9

Les plates-formes d'exécution

Ce chapitre détaille la notion de plate-forme telle que définie dans MDA et explique la place des plates-formes dans les transformations PIM vers PSM.

Nous présentons ici les définitions abstraites des concepts de plates-formes fournies par l'OMG et montrons les deux approches, par profil et approche par métamodèle, qui permettent de prendre en compte les plates-formes dans MDA.

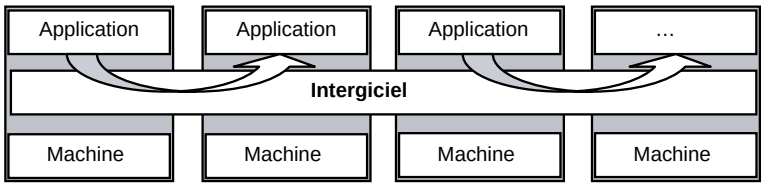
MDA et la séparation des préoccupations

L'interopérabilité est un des problèmes récurrents des systèmes informatiques. Elle vise à faire en sorte que plusieurs applications interagissent alors qu'elles sont écrites dans des langages différents, qu'elles s'exécutent sur des machines différentes et qu'elles ne partagent pas les mêmes principes architecturaux.

Pour faire face à ce problème majeur, plusieurs solutions ont été élaborées. La plus importante d'entre elles a sans doute été la définition des intergiciels, ou middlewares. Un intergiciel est une couche d'abstraction entre les applications et les machines sur lesquelles elles s'exécutent, qui permet de résoudre les problèmes techniques d'interopérabilité.

La figure 9.1 présente différentes applications qui interagissent par le biais d'un intergiciel. Les applications communiquent toujours *via* l'intergiciel, lequel résout, au minimum, les problèmes de représentation et de transport des données. Les intergiciels ont permis une avancée considérable et sont aujourd'hui totalement intégrés dans les environnements de développement. Les plus connus d'entre eux sont CORBA, EJB, .Net et les Web Services.

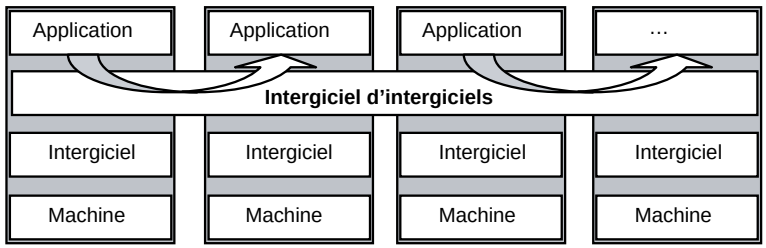
Figure 9.1
Intergiciels et interopérabilité des applications



Malgré tous les avantages qu'ils offrent, les intergiciels présentent l'inconvénient que les applications qui en bénéficient en sont aussi dépendantes. Du fait d'incompatibilités entre intergiciels, elles ne peuvent interagir avec des applications qui utilisent d'autres intergiciels. Il s'ensuit un problème d'interopérabilité entre intergiciels. Le paradoxe bien connu des intergiciels est de faire naître un problème d'interopérabilité en essayant de résoudre un problème d'interopérabilité.

Ce paradoxe des intergiciels est accentué par le nombre sans cesse croissant d'intergiciels (RPC, DCE, CORBA, DCOM, EJB, CCM, Web Services, etc.). Résoudre le problème de l'interopérabilité entre intergiciels en essayant de définir une couche d'abstraction supplémentaire (intergiciel d'intergiciels) ne semble guère prometteur *(voir figure 9.2)*. Il faudrait définir un unique intergiciel permettant de résoudre tous les problèmes d'interopérabilité entre les intergiciels existants tout en prenant en considération la création inévitable de nouveaux intergiciels. Ce serait irréalisable puisque, à peine sorti, cet intergiciel d'intergiciels serait rendu obsolète du fait de la sortie d'un nouvel intergiciel.

Figure 9.2
Le paradoxe des intergiciels



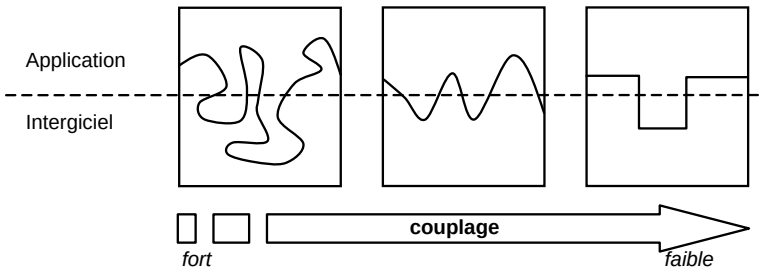
Outre ce paradoxe, l'utilisation des intergiciels a soulevé le problème de leur intégration dans le code des applications. Ce sont les parties techniques des intergiciels (composants, bibliothèques ou frameworks) qui fournissent les solutions d'interopérabilité (communications, sécurité, transactions, etc.). Pour utiliser les intergiciels, il faut intégrer ces parties techniques dans le code des applications.

Certains intergiciels nécessitent une forte intégration de leurs parties techniques dans le code de l'application. Il est dès lors quasiment impossible de reprendre une application et de changer d'intergiciel puisque le code de l'application est trop imbriqué avec l'intergiciel. D'autres intergiciels proposent un couplage plus faible, ce qui favorise une réutilisation du code des applications mais au prix de performances moins intéressantes.

Quel que soit le degré d'intégration de l'intergiciel dans l'application, il est toujours plus ou moins difficile de distinguer le code de l'application, aussi appelé code métier, du code technique assurant l'interopérabilité. De ce fait, la pérennité du code se trouve irrémédiablement diminuée. De plus, cela augmente considérablement le coût d'un changement d'intergiciel, pourtant nécessaire pour obtenir des gains de qualité.

La figure 9.3 illustre différents degrés d'intégration de l'intergiciel dans l'application. Le but est de schématiser l'impact de l'intégration de l'intergiciel sur la pérennité du code métier. La séparation du code de l'application et des parties techniques de l'intergiciel est un problème majeur pour l'interopérabilité des applications et leur pérennité relative face à l'évolution des intergiciels.

Figure 9.3
Intégration de l'intergiciel dans l'application



Pour affronter le paradoxe des intergiciels et le problème de l'intégration de l'intergiciel dans l'application, les concepteurs de MDA ont naturellement appliqué le fameux principe de séparation des préoccupations.

Le principe de séparation des préoccupations consiste à séparer la spécification d'une application en plusieurs spécifications indépendantes et cohérentes. Une spécification doit être réalisée pour chacune des préoccupations, et ce, indépendamment des spécifications correspondant aux autres préoccupations. Il faut ensuite spécifier les relations de cohérence entre toutes ces spécifications.

Séparation des préoccupations

Le principe de la séparation des préoccupations est pratiqué de longue date en informatique. Les niveaux d'abstraction « conceptuel », « logique » et « physique » recommandés par l'approche Merise en sont une illustration. Le standard ISO RM-ODP (Reference Model of Open Distributed Processing), qui propose de découper les spécifications des systèmes informatiques en cinq points de vue relatifs à cinq préoccupations différentes, en est un exemple très abouti. Le framework de Zachman constitue pour sa part une référence complète en terme de définition d'aspects à considérer dans un système d'information.

Concernant les intergiciels, MDA applique la séparation des préoccupations en préconisant l'élaboration de deux spécifications, ou modèles dans la terminologie MDA : un modèle pour la partie métier de l'application, le PIM (Platform Independent Model), et un modèle pour la partie technique relative à l'utilisation d'un intergiciel donné, le PSM

(Platform Specific Model). Le lien entre le modèle métier et le modèle technique est assuré par une transformation de modèles.

Grâce à cette approche, le problème de l'intégration de l'intergiciel dans l'application est résolu de manière définitive puisque le métier de l'application se retrouve dans le PIM sans aucune dépendance envers l'intergiciel. Quant au paradoxe des intergiciels, il n'est pas réglé mais se retrouve projeté dans les transformations PIM vers PSM. En effet, celles-ci ont la responsabilité de résoudre les problèmes d'interopérabilité entre applications mais aussi entre intergiciels.

Prise en compte des plates-formes d'exécution par MDA

La dernière des trois caractéristiques de MDA, après la pérennité et la productivité, est donc la considération des intergiciels, appelés plates-formes d'exécution dans la terminologie MDA, comme une préoccupation à part entière.

Cette préoccupation est différente de celle relative au métier de l'application. Elle dispose de ses propres modèles, liés aux autres modèles MDA par des transformations de modèles.

Les concepts MDA traitant des plates-formes

Avant d'entrer dans le vif du sujet, il est nécessaire de poser les concepts nécessaires à l'étude des plates-formes dans MDA.

Les définitions des concepts que nous donnons ici sont dérivées des définitions fournies dans le guide MDA de l'OMG.

Le système

L'objectif de MDA est de générer des applications informatiques. Ces applications informatiques existent au sein d'un système.

La définition d'un système en terminologie MDA se comprend au sens le plus large. Cela peut être un programme informatique, une machine, une organisation humaine, une entreprise, un ensemble de systèmes et même une fédération de systèmes. C'est pourquoi les applications informatiques des systèmes MDA sont complexes, avec des besoins d'interopérabilité intra- ou intersystèmes.

La plate-forme

Une plate-forme MDA est une entité technique qui fournit un ensemble cohérent de fonctionnalités grâce à des interfaces. Une application qui s'exécute sur une plate-forme peut bénéficier des fonctionnalités de la plate-forme en utilisant uniquement les interfaces, sans avoir à connaître les détails d'implémentation de ces fonctionnalités.

Une plate-forme peut être générique, telles les plates-formes suivantes :

- Plate-forme objet, dont les fonctionnalités sont, par exemple, l'instanciation d'objets et l'appel de méthodes.
- Plate-forme batch, dont les fonctionnalités sont la possibilité d'exécuter en série un ensemble d'applications.
- Plate-forme flot de données, dont les fonctionnalités ont pour rôle de permettre l'exécution d'applications en même temps que les échanges de données entre celles-ci.

Une plate-forme peut être spécifique d'une technologie, telles les plates-formes Java, PHP ou .Net.

Dans la suite de ce chapitre, nous utilisons le symbole pi (Π) pour représenter une plate-forme quelle qu'elle soit.

L'application

L'application est ce que l'on développe. Une application existe dans un système et s'exécute sur une plate-forme.

Un système peut contenir une ou plusieurs applications.

Indépendance à l'égard des plates-formes d'exécution

L'indépendance à l'égard de la plate-forme est une qualité dont peut disposer un modèle. Un modèle est dit indépendant d'une plate-forme si aucun de ses éléments n'utilise les fonctionnalités offertes par la plate-forme.

Étant donné qu'il existe plusieurs types de plates-formes, il est possible qu'un modèle soit indépendant de certaines plates-formes et pas d'autres. De ce fait, l'indépendance à l'égard de la plate-forme est sujette à un degré d'appréciation. Nous dirons, par exemple, que les modèles dépendants de la plate-forme objet sont plus indépendants que les modèles dépendants de la plate-forme Java.

Le modèle de plate-forme

Un modèle de plate-forme, ou PM (Platform Model), modélise une plate-forme. Plus précisément, il définit les différentes fonctionnalités de la plate-forme et précise comment les utiliser.

Nous pouvons considérer qu'un modèle de plate-forme possède deux facettes : une facette interne, définissant les fonctionnalités de la plate-forme et la façon dont elles sont implémentées, et une facette externe, définissant comment utiliser ces fonctionnalités et plus précisément comment un modèle qui souhaite utiliser ces fonctionnalités doit se connecter au modèle de la plate-forme.

L'implémentation

Une implémentation désigne un ensemble de modèles contenant suffisamment d'information pour permettre la génération intégrale d'une application et son exécution.

PIM, PSM et plates-formes

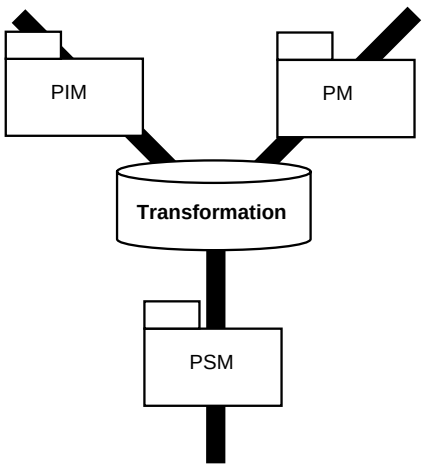
Les définitions que nous venons de donner permettent de différencier les concepts MDA traitant des plates-formes. Elles nous permettent de saisir que les notions de PIM, PSM et plate-forme sont différentes mais fortement liées.

Au cours des chapitres précédents, nous avons présenté les transformations PIM vers PSM comme de simples transformations prenant en entrée un modèle indépendant des plates-formes et fournissant en sortie un modèle dépendant d'une plate-forme. Ces transformations sont en réalité beaucoup plus complexes.

Les transformations PIM vers PSM doivent en effet prendre en compte la notion de plate-forme et plus précisément de modèle de plate-forme. Les transformations PIM vers PSM ont donc un troisième paramètre, les PM (Platform Model).

La représentation communément admise des transformations PIM vers PSM est une représentation en Y *(voir figure 9.4).* La branche de gauche représente le modèle PIM, la branche de droite le PM et le tronc le modèle PSM.

Figure 9.4
Transformation PIM vers PSM



Cette représentation illustre bien l'importance du PM et montre la complexité des règles de correspondance de la transformation. Nous allons voir qu'elle est cependant difficile à concrétiser.

Nous avons vu au chapitre 7, consacré à la transformation des modèles, qu'une transformation de modèles était une transformation structurelle fondée sur des métamodèles.

Pour concrétiser la représentation en Y avec des transformations de modèles, il faut disposer des métamodèles des PIM, PSM et PM. Grâce à ces métamodèles, il est possible de définir les règles de correspondance structurelle qui composent la transformation.

Disposer du métamodèle du PIM ne pose pas de problème. Nous avons d'ailleurs décrit dans les chapitres précédents plusieurs métamodèles pouvant jouer ce rôle, tels les métamodèles UML et OCL.

Disposer du métamodèle de plate-forme pose en revanche de sérieux problèmes. Un tel métamodèle doit permettre, en théorie, de construire les modèles de toutes les plates-formes. Or, comme nous l'avons vu, les plates-formes peuvent être très différentes (plates-formes génériques, plates-formes spécifiques). Malheureusement, il n'existe pas actuellement de métamodèle permettant de modéliser une telle variété de plates-formes. Un travail de recherche est en cours sur ce sujet, mais ses résultats ne sont pas encore utilisables.

Il va donc falloir se passer de ce métamodèle pendant encore quelques années. De ce fait, les règles de correspondance des transformations PIM vers PSM ne peuvent s'appuyer sur le métamodèle des PM.

Pour résoudre ce problème, c'est naturellement dans les métamodèles des PSM que l'on intègre toutes les informations relatives à la plate-forme. Un métamodèle de PSM est défini pour une plate-forme particulière. Celui-ci définit la structuration des modèles qui utilisent les fonctionnalités de la plate-forme. En d'autres termes, ce métamodèle définit les interfaces permettant d'utiliser les fonctionnalités de la plate-forme et précise la façon de se connecter à ces interfaces.

Bien qu'elle soit présentée pédagogiquement sous forme de Y, la transformation des modèles PIM vers les modèles PSM est une transformation structurelle fondée sur les métamodèles des PIM et des PSM. La plate-forme, ou plus précisément les interfaces d'utilisation de la plate-forme apparaissent dans le ou les métamodèles des PSM.

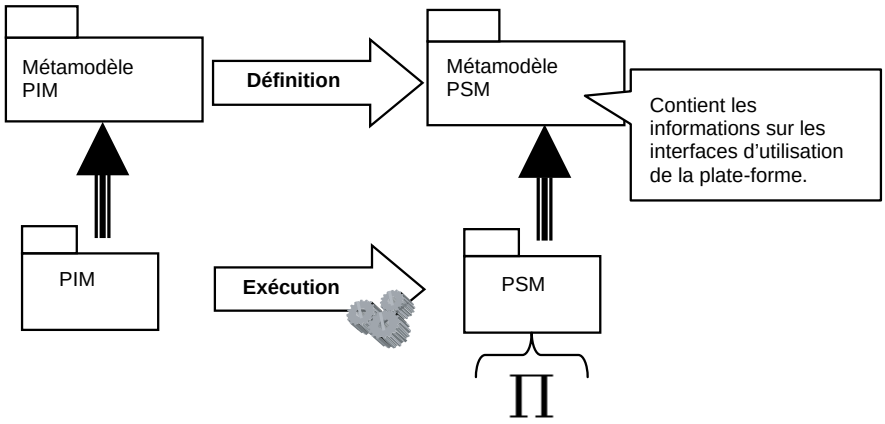
La figure 9.5 illustre la concrétisation de la transformation PIM vers PSM. Nous constatons que les modèles de plates-formes (PM) ne sont nécessaires ni pour la définition de la transformation ni pour son exécution. Soulignons que cela ne signifie pas que les modèles de plates-formes sont inintéressants pour MDA. Il existe d'ailleurs des travaux de recherche qui exploitent ces modèles de plates-formes pour automatiser et modéliser les activités de supervision ou de test d'applications. Cela signifie simplement qu'il est possible de s'en passer pour élaborer les transformations PIM vers PSM.

La figure 9.5 illustre aussi l'adhérence du PSM à une plate-forme particulière, contrairement au PIM, qui n'utilise aucune fonctionnalité d'aucune plate-forme.

Cette concrétisation de la transformation des PIM vers des PSM préserve néanmoins la séparation des préoccupations recherchée dans l'approche sous forme de Y. L'intégration de l'intergiciel et de l'application se trouve séparée dans les modèles PIM et PSM, et l'interopérabilité réside toujours dans les règles de la transformation.

Figure 9.5

Concrétisation de la transformation PIM vers PSM



Modèles intermédiaires

Les transformations PIM vers PSM définissent des correspondances structurelles entre les métamodèles PIM et PSM. Or, ce qui caractérise ces transformations, c'est qu'il apparaît bien souvent qu'un concept PIM correspond à différents concepts PSM.

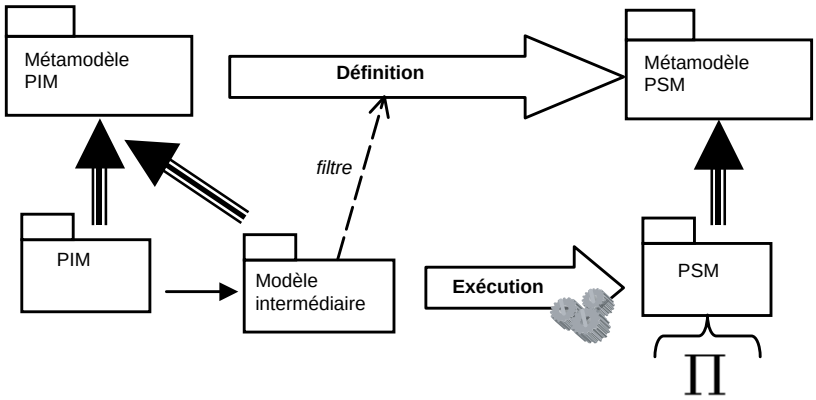
Si nous considérons la transformation UML vers EJB, une transformation PIM vers PSM que nous détaillons plus précisément au chapitre 12, il est parfaitement envisageable de transformer une classe UML en un composant EJB de type Session ou Entity. Le choix du type de composant (Session ou Entity) se fait en fonction du rôle joué par la classe dans l'application et de la façon dont l'utilisateur souhaite exploiter la plate-forme EJB. Ce choix ne dépend donc pas de la structuration du modèle UML. La transformation UML vers EJB ne peut s'écrire comme une simple transformation de modèles car, à une classe UML, peut correspondre structurellement plusieurs concepts EJB, ici des composants Session ou Entity.

Ce problème est caractéristique des transformations PIM vers PSM. Il fait apparaître le besoin d'une étape supplémentaire permettant de préciser les choix des règles de correspondance à appliquer. Pour marquer cette étape, un modèle intermédiaire est souvent généré à partir du PIM. Ce modèle est une sorte de copie du modèle PIM, si ce n'est qu'il contient les informations permettant de sélectionner les règles de correspondance à appliquer. Ce modèle intermédiaire, parfois appelé modèle de choix, permettra la génération concrète du PSM. Il n'est ni un PIM ni un PSM.

La figure 9.6 illustre la transformation PIM vers PSM utilisant un modèle intermédiaire. Le métamodèle de ce modèle est le métamodèle PIM, mais cela pourrait aussi bien être un autre métamodèle. L'important est que ce modèle contienne les informations permettant de filtrer l'application des règles de la transformation. Bien souvent, un modèle intermédiaire est généré automatiquement à partir du modèle PIM et contient des choix par défaut concernant le filtrage des règles de transformation. Le modèle intermédiaire et les

choix par défaut qu'il contient peuvent être modifiés manuellement par l'utilisateur avant d'exécuter la transformation de modèles qui permettra d'obtenir le PSM.

Figure 9.6
Transformation PIM vers PSM et modèle intermédiaire



Si nous reprenons notre exemple de transformation UML vers EJB, le modèle intermédiaire pourrait contenir des choix par défaut précisant que toutes les classes UML doivent être transformées en des composants EJB Session. Ainsi, l'utilisateur pourrait préciser pour certaines classes UML un choix différent et permettre une transformation vers des composants Entity.

Superposition de plates-formes

Pour finir avec les transformations PIM vers PSM, il est important de rappeler que la notion d'indépendance à l'égard de la plate-forme est sujette à un degré d'appréciation *(voir les définitions de concepts précédentes).*

Ainsi, une transformation PIM vers PSM peut permettre la création de modèles qui devront eux-mêmes être transformés pour pouvoir bénéficier d'une plate-forme plus concrète.

Par exemple, la transformation UML vers EJB permet la création de modèles qui définissent comment les applications utilisent les fonctionnalités de la plate-forme EJB. Ces modèles devront eux-mêmes être transformés vers, par exemple, des modèles Java afin de définir comment sont utilisées les fonctionnalités de la plate-forme Java. Nous considérons ici qu'il est possible de métamodéliser la plate-forme Java *(voir encadré).*

Métamodèle Java

Le métamodèle Java que nous considérons définit la structure des applications Java. Ses métaclasses ne représentent donc que les concepts de classe, d'interface, de package, d'exception, d'opération et d'attribut. Ce métamodèle ne permet toutefois pas de représenter sous forme de modèle n'importe quelle application Java. Il manque en effet toutes les métaclasses permettant de représenter les instructions et les différents blocs d'instructions possibles (`for`, `while`, `try`, etc.). En d'autres termes, le code Java n'est pas représenté sous forme de modèle. La transformation de modèles EJB vers Java n'est donc pas une génération de code.

Ces différents degrés d'appréciation font qu'il est possible de superposer les transformations PIM vers PSM. La figure 9.7 illustre une superposition des plates-formes EJB et Java sans y faire apparaître les modèles intermédiaires. Nous voyons que la transformation d'un PIM a donné un PSM, lequel sera à son tour transformé en un autre PSM. Cette figure montre bien la subjectivité du concept de plate-forme dans MDA.

Cette subjectivité, qui pourrait être considérée comme une aberration de MDA, est en fait un avantage important car elle permet de ne pas figer l'état des plates-formes. Des technologies qui paraissent abstraites aujourd'hui pourront être considérées comme des plates-formes demain sans pour autant bouleverser les fondements de l'approche MDA.

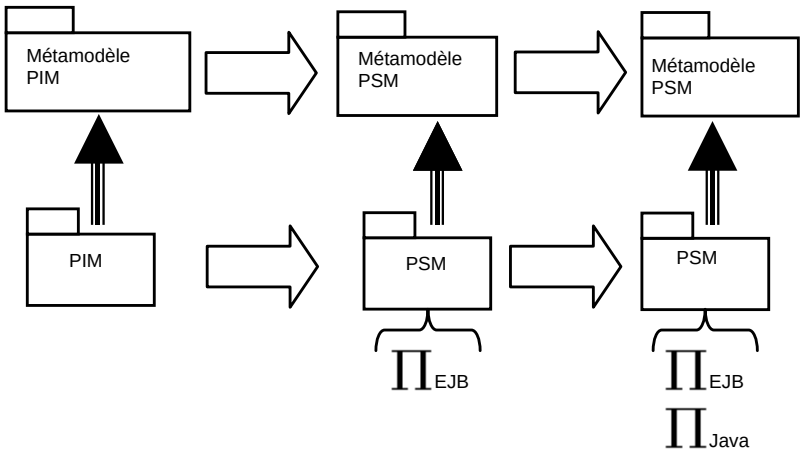


Figure 9.7
Superposition des plates-formes EJB et Java

En résumé

Nous venons de présenter les causes qui ont rendu nécessaire la prise en compte des plates-formes dans MDA. Ces causes sont principalement le paradoxe des intergiciels et l'intégration des intergiciels dans les applications. Les plates-formes sont une préoccupation à part entière de MDA, qui suit le principe de la séparation des préoccupations.

Nous avons ensuite présenté la complexité des transformations PIM vers PSM. Celles-ci, présentées sous forme de Y, prennent en entrée des PIM mais aussi des modèles de plate-forme (PM) pour fournir en sortie les PSM.

Nous avons vu que les plates-formes ne pouvaient malheureusement pas être modélisées, car il n'existe pas à l'heure actuelle de métamodèle couvrant la diversité des plates-formes. Cela explique pourquoi les transformations PIM vers PSM ne peuvent être concrétisées selon l'approche en Y.

Ces transformations sont des transformations relativement classiques de modèles. Elles sont constituées de règles de correspondance structurelle entre métamodèles PIM et PSM. Afin de permettre une certaine flexibilité dans l'utilisation des plates-formes, elles

disposent de modèles intermédiaires permettant d'effectuer des choix quant aux règles de correspondance à appliquer.

Les transformations PIM vers PSM et la prise en compte des plates-formes dépendent donc fortement des métamodèles des PSM. Ceux-ci contiennent toutes les informations d'utilisation des plates-formes. Étant donné qu'il est possible de superposer les plates-formes, leur rôle s'en trouve d'autant plus renforcé.

Initialement, MDA demandait aux fournisseurs de plates-formes d'élaborer les transformations de modèles correspondant à leurs plates-formes. À l'heure actuelle, la tendance est inversée, et ce sont les utilisateurs de plates-formes qui développent leurs propres transformations et y intègrent leur façon d'utiliser les plates-formes, fruit de leur expérience et de leur métier.

Les sections suivantes de ce chapitre présentent les différentes possibilités de définir des métamodèles des PSM, soit en passant par le mécanisme de profils UML, soit en élaborant directement des métamodèles MOF .

Définition d'un métamodèle de PSM par profil UML

Une première approche pour définir des métamodèles de PSM consiste à utiliser le mécanisme de profil UML. MDA recommande cette approche car le support industriel d'UML la rend facile à mettre en œuvre.

Rappelons qu'un profil permet d'adapter UML à un domaine particulier. Dans le contexte des métamodèles de PSM, l'objectif est d'adapter UML à une plate-forme particulière. Il est de la sorte possible de construire des modèles UML modélisant des applications qui utilisent les fonctionnalités d'une plate-forme.

Étant donné qu'UML est déjà fortement orienté objet, l'approche par profil ne permet de cibler que des plates-formes ayant un caractère relativement orienté objet. En effet, un profil UML ne permet que d'adapter UML à une plate-forme particulière. Un modèle UML profilé, c'est-à-dire qui utilise un profil, est avant tout un modèle UML. Il a donc un caractère objet profondément ancré.

La souplesse d'UML offre toutefois la possibilité de cibler des plates-formes non-objet, telles que SQL, VB ou Fortran.

Les profils de PSM

Plusieurs profils de PSM sont déjà définis. C'est le cas, entre autres, des profils EJB pour UML, CORBA et CCM.

Nous détaillons au chapitre 10 le profil de PSM de la plate-forme EJB.

La définition d'un profil de PSM consiste à définir les stéréotypes, tagged-values et contraintes nécessaires pour exprimer l'utilisation d'une plate-forme par une application.

Les stéréotypes

Les stéréotypes sont applicables à un type d'élément UML, tel que classe, package, cas d'utilisation, etc. La définition d'un stéréotype passe donc par l'identification de la métaclasse UML correspondant au type de l'élément. Ce choix est très important compte tenu de l'adhérence d'UML au modèle objet.

Bien souvent, les stéréotypes d'un profil de PSM ciblent les métaclasses `Class`, `Package`, `Interface`, `Association`, `Operation` et `Attribute`. Ces métaclasses correspondent aux métaclasses formant les diagrammes de classes, qui sont généralement les diagrammes UML les plus proches des plates-formes.

Très souvent, un stéréotype est défini sur la métaclasse `Class` pour chaque concept clé de la plate-forme. Par exemple, pour la plate-forme EJB, les stéréotypes d'`EJBImplementation`, `EJBRemoteInterface` et `EJBHomeInterface` sont définis sur la métaclasse `Class`. Ces stéréotypes sont utilisés pour marquer dans le modèle UML les éléments ayant une correspondance directe avec les concepts de la plate-forme.

Les stéréotypes sur la métaclasse `Association` permettent quant à eux la définition de nouveaux liens propres à la plate-forme. Grâce à ces stéréotypes, il est possible de préciser la signification des liens UML au regard de la plate-forme et ainsi de mieux établir son utilisation dans les modèles UML. Par exemple, pour la plate-forme EJB, le stéréotype d'`EJBRealizeRemote` défini sur la métaclasse `Association` permet d'établir un lien entre un composant EJB et son interface remote.

Les définitions des stéréotypes sur les autres métaclasses, telles que `Package`, `Operation` et `Attribute`, sont aussi utilisées mais ne jouent pas de rôles particuliers dans le profil.

Les tagged-values

Les tagged-values sont très utiles dans les profils de PSM car elles portent des valeurs. Il est ainsi possible d'ajouter certaines données propres aux plates-formes dans les modèles UML.

Depuis UML2.0, les tagged-values sont exclusivement utilisées sur des éléments stéréotypés. Cela permet de préciser en détail la façon dont est utilisée la plate-forme. Bien souvent, les tagged-values permettent d'exprimer des informations de configuration de la plate-forme. Elles sont généralement employées pour préciser les valeurs des propriétés non fonctionnelles de la plate-forme.

Par exemple, dans EJB, la tagged-value `EJBSessionType` appliquée aux éléments stéréotypés `EJBSessionHomeInterface` permet de préciser le type du composant Session (Stateless ou Stateful).

Les tagged-values sont utilisées pour spécifier une valeur parmi un ensemble de valeurs possibles ou une quantité. Les tagged-values ne sont pas utilisées pour exprimer des références entre les éléments.

Les contraintes

Définies au sein des profils de PSM, les contraintes permettent de restreindre l'utilisation des stéréotypes et des tagged-values. Grâce aux contraintes, il est possible de spécifier des

structures obligatoires entre des éléments stéréotypés. Il est aussi possible de spécifier le fait que tel élément stéréotypé possède tel ensemble de tagged-values.

Par exemple, pour le profil EJB, il est possible de définir que tout élément stéréotypé `EJBImplementation` soit obligatoirement relié à une interface stéréotypée `EJBHomeInterface` et à une interface stéréotypée `EJBRemoteInterface`. Il est aussi possible de définir que les éléments stéréotypés `EJBSessionHomeInterface` aient une tagged-value `EJBSessionType` portant le type du composant Session (Stateful ou Stateless).

Les contraintes permettent aussi d'interdire certaines structurations tolérées par le méta-modèle UML. Par exemple, il est possible de spécifier qu'un élément stéréotypé `EJBImplementation` ne puisse posséder d'opération de construction avec des paramètres.

Définition du modèle intermédiaire

Dans le contexte de l'utilisation des profils pour les PSM, il est important de noter que tous les modèles (PIM et PSM) sont des modèles UML. Les modèles intermédiaires sont donc inévitablement des modèles UML, ce qui facilite énormément la définition du modèle intermédiaire.

Rappelons que le modèle intermédiaire porte en lui les informations permettant de filtrer l'application des règles de transformation PIM vers PSM. Un modèle intermédiaire n'est donc nécessaire que s'il existe différents choix de transformations.

Pour définir un modèle intermédiaire, il faut inventorier les différents choix possibles de transformation. Il existe deux transformations possibles lors d'une transformation PIM vers PSM lorsque le PSM est un profil UML :

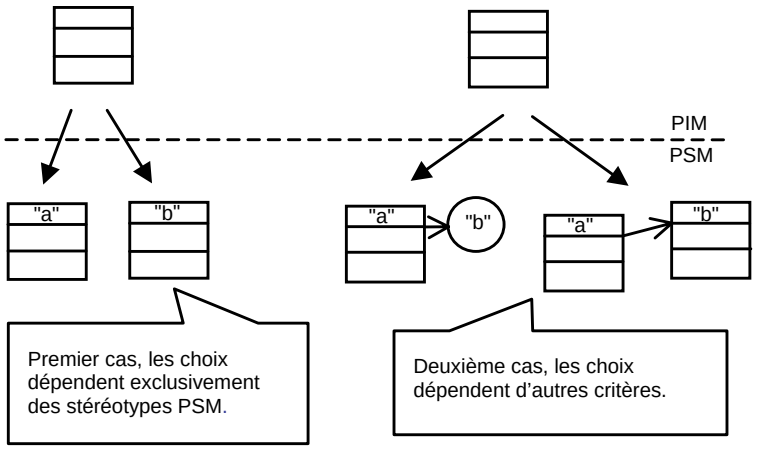
- Un élément du PIM donne lieu à la création d'un élément du PSM qui porte différents stéréotypes et tagged-values.
- Un élément du PIM donne lieu à la création d'un ensemble d'éléments du PSM. Ces éléments sont stéréotypés, portent des tagged-values et respectent les contraintes du profil.

Dans le premier cas, les choix de transformation correspondent aux stéréotypes et tagged-values que peut porter l'élément du PSM. Préciser ces choix revient à déclarer explicitement les stéréotypes et tagged-values à utiliser. Cela peut se faire avec les stéréotypes et tagged-values du profil de PSM. Le modèle intermédiaire est donc un modèle PSM.

Dans le deuxième cas, les choix de transformation correspondent à différents critères, qui ne sont pas exprimables avec les stéréotypes et tagged-values du profil du PSM. Le modèle intermédiaire doit donc inclure des informations pour opérer un choix.

La figure 9.8 illustre les différentes possibilités de transformations. Dans le premier cas (à gauche sur la figure) les choix sont exprimables avec les stéréotypes et tagged-values du profil de PSM, tandis que dans le second les choix ne sont pas exprimables avec le profil de PSM.

Figure 9.8
Choix des règles de transformation à appliquer et profil de PSM



Seul le deuxième cas nécessite la définition d'une nouvelle structure pour le modèle intermédiaire. Une solution consiste à définir un nouveau profil contenant un stéréotype par choix possible de règles de transformation à appliquer. Cette solution n'est pas très esthétique et nécessite la définition d'un nouveau profil. La définition d'assistant permettant à l'utilisateur de préciser son choix pendant l'application de la transformation est parfois préférée.

Utilisation d'un profil de PSM

Grâce aux profils de PSM, les transformations PIM vers PSM permettent de préciser l'utilisation des plates-formes par les applications. Une fois générés, les PSM servent de base à la génération de code. Leur caractéristique principale est donc d'être productifs. Ils facilitent la génération de code car ils contiennent les informations d'utilisation des plates-formes.

Cependant, même si les profils de PSM permettent de préciser l'utilisation de toutes les fonctionnalités d'une plate-forme, les transformations PIM vers PSM sont relativement générales. Elles n'exploitent donc pas toutes les possibilités des profils de PSM.

C'est la raison pour laquelle les PSM sont bien souvent raffinés après exécution des transformations PIM vers PSM. C'est d'ailleurs à ce moment que les patrons de conception, propres aux plates-formes, sont appliqués. Ces PSM retravaillés contiennent l'information permettant une exploitation optimale des fonctionnalités des plates-formes. Cette information mérite parfois d'être capitalisée, ce qui fait que certains PSM ont aussi un caractère de pérennité.

Ajoutons qu'un modèle UML peut contenir les stéréotypes et tagged-values de plusieurs profils. On dit alors qu'il est profilé par plusieurs profils. Cela n'est possible que si tous les profils partagent les mêmes contraintes de structuration des modèles UML. C'est grâce

à ce mécanisme que sont réalisées certaines des superpositions de plates-formes. Par exemple, la superposition de la plate-forme EJB sur la plate-forme Java se fait avec les mêmes modèles (la structure ne change pas) qui contiennent les stéréotypes et les tagged-values des deux profils.

Définition directe d'un métamodèle MOF de PSM

L'autre approche possible pour définir des métamodèles de PSM consiste à élaborer directement des métamodèles MOF. MDA propose cette approche, qui a l'avantage de ne pas dépendre du métamodèle UML et de son caractère orienté objet. Il est de la sorte possible de prendre en compte toutes les plates-formes d'exécution, qu'elles soient orientées objet ou non.

Définition du métamodèle

La définition d'un métamodèle de PSM est plus complexe que celle d'un profil de PSM car il n'existe pas de similitude entre le métamodèle de PSM et le métamodèle du PIM, alors que, pour les profils, les métamodèles sont tous fondés sur UML.

Il faut donc définir *ex nihilo* l'intégralité du métamodèle de PSM. De plus, l'OMG ne définit aucun métamodèle de PSM sous forme de métamodèle MOF, contrairement à l'approche par profil.

Les métaclasses

Bien souvent, les métamodèles de PSM contiennent une métaclasse par concept de la plate-forme. L'idée est de réifier, autrement dit de représenter sous forme d'instance, chaque concept afin de permettre une meilleure identification dans les modèles instances. Par exemple, le métamodèle de PSM pour la plate-forme PHP contient les métaclasses correspondant aux concepts de classes, attributs, méthodes, etc.

Chacune de ces métaclasses contient les méta-attributs nécessaires pour préciser les propriétés du concept. Par exemple, toujours pour la plate-forme PHP, la métaclasse correspondant au concept de classe contient un méta-attribut pour le nom de la classe.

Les méta-associations

Les méta-associations sont souvent utilisées dans les métamodèles de PSM pour exprimer des relations entre concepts. Par exemple, une méta-association est définie dans la plate-forme PHP pour exprimer la relation d'héritage entre les classes PHP. Cette méta-association a pour source et pour cible la métaclasse correspondant au concept de classe PHP.

Les contraintes OCL

Tout comme dans les profils, les contraintes OCL peuvent être utilisées dans les métamodèles de PSM pour contraindre la structuration des modèles instances. Par exemple pour la plate-forme PHP, une contrainte peut être définie pour préciser que la source d'une relation d'héritage ne peut être équivalente à la cible.

Structure ou comportement ?

Lors de l'élaboration d'un métamodèle de PSM, il est toujours difficile de déterminer quel degré de couverture cibler. Les deux degrés extrêmes de couverture sont les suivants :

- Degré de couverture structurelle : le métamodèle de PSM représente tous les concepts structurels de la plate-forme. Par exemple, pour la plate-forme PHP, le métamodèle contiendra les métaclasses représentant les concepts de classe, de module, de méthode, etc.
- Degré de couverture de comportementale : le métamodèle de PSM représente les concepts structurels mais aussi tous les concepts relatifs aux comportements dans la plate-forme. Par exemple, pour la plate-forme PHP, le métamodèle contiendra les concepts d'instruction, de boucle, etc.

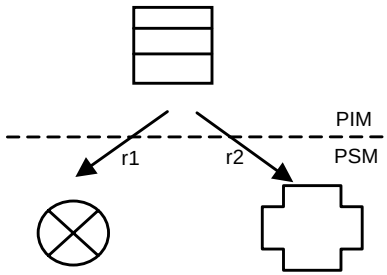
Soulignons qu'actuellement, seuls des métamodèles de PSM à couverture structurelle ont été élaborés compte tenu de la difficulté d'élaboration des métamodèles à couverture comportementale.

Définition du modèle intermédiaire

Contrairement à l'approche par profil, il n'existe pas plusieurs possibilités pour réaliser la définition des modèles intermédiaires. En effet, les transformations PIM vers PSM réalisées avec cette approche sont inévitablement des transformations qui changent la structuration des modèles (les modèles source et cible ne partagent pas le même métamodèle).

La figure 9.9 illustre différents choix d'application des règles de correspondance (`r1` ou `r2`) pour une transformation PIM vers PSM avec une approche par métamodèle de PSM. Les symboles utilisés montrent que la structure des PIM et des PSM est totalement différente.

Figure 9.9
Choix des règles de correspondance et métamodèle de PSM



Préciser les choix des règles de correspondance à appliquer ne peut donc se faire qu'en ajoutant de l'information au modèle PIM, information qui ne peut être structurée par le métamodèle de PSM. À la figure 9.9, cela se ferait par l'ajout d'une information dans le modèle PIM pour spécifier le choix, par exemple, de la règle `r1`.

La solution passe irrémédiablement par la définition d'un profil identifiant tous les choix d'application des règles de correspondance. Ce profil est, bien entendu, à appliquer sur le modèle PIM. À la figure 9.9, cela se ferait, par exemple, par l'ajout d'un stéréotype `r1` sur le PIM pour choisir la règle `r1`.

Il serait aussi possible de définir un métamodèle plutôt qu'un profil pour structurer le modèle intermédiaire, mais celui-ci devrait définir la même structure que la structure des PIM. Il serait donc structurellement équivalent au métamodèle PIM. L'utilisation des profils semble donc mieux adaptée.

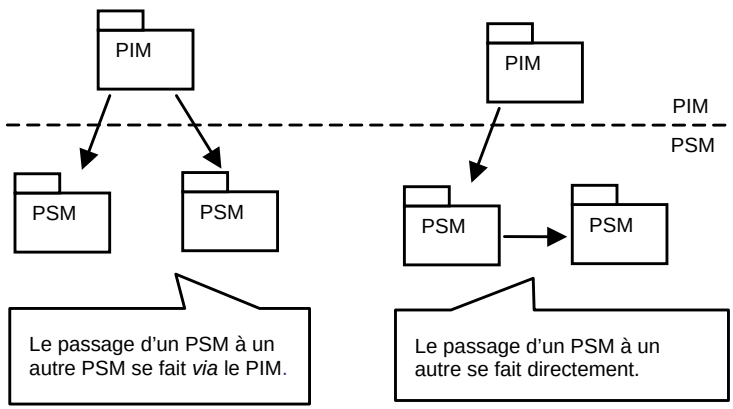
L'intérêt de définir un tel profil doit tenir compte des différents choix d'application des règles de transformation. Il est évident que l'intérêt d'un tel profil croît en fonction du nombre de choix possibles de règles de correspondance à appliquer. Plus il y a de choix, plus il est nécessaire de préciser ces choix dans le PIM, et donc plus l'intérêt du profil intermédiaire est important. Là encore, la définition d'assistant permettant à l'utilisateur de préciser son choix pendant l'application de la transformation est parfois préférée.

Utilisation d'un métamodèle de PSM

Tout comme les profils de PSM, les métamodèles de PSM permettent de préciser l'utilisation des plates-formes par les applications. Une fois générés, les PSM structurés selon les métamodèles de PSM facilitent la génération de code. L'intérêt de ces modèles est donc principalement d'être productifs.

Les PSM structurés selon les métamodèles de PSM ont aussi l'avantage de ne pas partager de structuration commune avec les modèles UML, ce qui n'est pas le cas avec les PSM générés selon l'approche par profil. Ils n'ont donc aucun point commun structurel avec les PIM. Ils contiennent uniquement des informations de structuration propres à la plate-forme d'exécution. Cela fait que leur pérennité est plus importante que celle des PSM générés selon l'approche par profil.

Figure 9.10
Transformation PSM vers PSM



Pour finir, étant donné leur indépendance à l'égard du modèle objet et leur structuration propre, il est possible d'envisager la transformation de ces PSM vers d'autre PSM, quels que soient les métamodèles de PSM considérés. Cette transformation de PSM, qui est une migration de plate-forme, se fait sans avoir à passer par le PIM *(voir la partie droite de la figure 9.10).* Cette approche serait plus difficilement réalisable avec les profils de PSM, étant donné leur adhérence au métamodèle UML.

Précisons que, contrairement à l'approche par profil, il n'est pas possible d'élaborer un PSM structuré selon plusieurs métamodèles de PSM. La superposition de plates-formes ne peut donc se faire que par la définition de nouvelles transformations de modèles.

Synthèse

Ce chapitre a précisé les différentes façons de prendre en compte les plates-formes d'exécution dans MDA. Considérer les plates-formes comme une préoccupation à part entière permet de répondre aux problèmes soulevés par les intergiciels.

Nous avons vu qu'il serait intéressant de pouvoir modéliser les plates-formes et de disposer pour cela d'un métamodèle de plates-formes (PM). Cependant, devant la diversité et la complexité des plates-formes, un tel métamodèle n'existe pas encore.

Les approches actuelles permettant la prise en compte des plates-formes dans MDA visent à intégrer toutes les informations d'exploitation des plates-formes dans les PSM et dans leur métamodèle. Les transformations PIM vers PSM précisent la façon dont les applications utilisent les plates-formes.

Concrètement, deux approches cohabitent :

- L'approche dite « profil de PSM », qui consiste à définir un profil UML contenant toutes les informations d'exploitation des plates-formes. Cette approche a l'avantage de faciliter les transformations PIM vers PSM étant donné son adhérence au métamodèle UML.
- L'approche dite « métamodèle de PSM », qui consiste à définir un métamodèle MOF des PSM. Cette approche a l'avantage de ne pas dépendre d'UML, ce qui permet la prise en compte de plates-formes non-objet.

Nous avons vu l'intérêt des modèles intermédiaires dans l'utilisation de chacune de ces approches. Les modèles intermédiaires permettent, pour chaque transformation, de préciser les règles de transformation à appliquer et ainsi d'affiner la façon dont sont exploitées les plates-formes.

10

La plate-forme J2EE

Ce chapitre est dédié à la prise en compte de la plate-forme d'exécution J2EE (Java 2 Enterprise Edition) par MDA. L'étendue et la complexité de J2EE nous ont conduit à ne considérer que la partie relative aux EJB, dont la prise en compte s'appuie sur le profil standard UML pour EJB élaboré par le JCP (Java Community Process).

Après un rappel des concepts élémentaires de la plate-forme J2EE, permettant de comprendre sa prise en compte par MDA, nous détaillons le profil UML pour EJB, qui permet de modéliser en UML des applications utilisant les services de la plate-forme J2EE.

Nous exposons ensuite la transformation PIM vers PSM, qui permet de transformer un modèle de composants UML2.0 en un modèle EJB. Nous utiliserons cette transformation pour l'étude de cas PetStore au chapitre 12.

Rappels sur la plate-forme J2EE

La plate-forme J2EE couvre l'ensemble des technologies Java permettant la mise en place d'applications d'entreprise, c'est-à-dire d'applications résidant du côté de l'entreprise et réalisant toute la logique métier de cette dernière. Elle permet un développement orienté objet, voire composant, des applications d'entreprise. Son objectif est d'offrir tous les avantages des paradigmes objet et composant aux applications d'entreprise, lesquelles étaient souvent développées à l'aide de langages non-objet, tels que Cobol.

J2EE permet un découpage multitiers. Elle supporte par défaut la séparation selon les trois tiers présentation, traitement et données. Pour chacun de ces tiers, J2EE offre des technologies différentes. Nous verrons en particulier la technologie JSP pour le tiers présentation et la technologie EJB pour les tiers traitement et données.

Les applications d'entreprise construites avec J2EE sont principalement des applications réparties, la répartition étant une composante de base de J2EE. Tous les éléments d'une application J2EE peuvent donc s'exécuter sur des machines différentes. Les clients peuvent être des navigateurs Web ou des clients lourds, et les serveurs être des clusters de machines.

J2EE est un standard entièrement Java. Une application d'entreprise réalisée avec J2EE bénéficie de la sorte du fameux WORA (Write Once Execute Anywhere) et peut s'exécuter sur n'importe quelle plate-forme disposant d'une machine virtuelle Java, ou JVM. Le fait d'être un standard JCP est un gage de qualité. J2EE est en effet le fruit du travail collaboratif des membres du JCP. Il s'agit d'un standard consensuel, élaboré par tous les acteurs industriels du monde Java pour l'entreprise.

J2EE est fortement utilisé par les entreprises, et de nombreux serveurs d'applications J2EE répondent à ce marché, qu'ils soient commerciaux, comme WebLogic de BEA, WebSphere d'IBM ou même Oracle, ou Open Source, comme JBoss.

L'architecture J2EE

J2EE est une plate-forme aussi complète que complexe. Nous en donnons ci-après une vue synthétique. Nous n'en détaillons que les concepts élémentaires qui nous permettront d'expliquer comment cette plate-forme est prise en compte dans MDA.

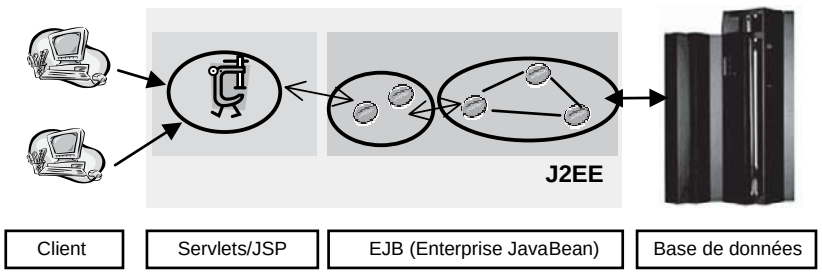
Par souci de clarté et de simplicité, nous ne considérons pas l'intégralité de la plate-forme. Nous ne visons qu'à faire la preuve de la réalité de la prise en compte de J2EE par MDA.

Vue macroscopique

La figure 10.1 illustre une mise en œuvre classique de J2EE pour une application d'entreprise. La partie J2EE de l'application est composée des deux blocs suivants (clients et bases de données ne sont pas couverts par J2EE) :

- **Servlets/JSP.** Ce bloc gère l'interface avec le client. Les servlets et les JSP sont des technologies J2EE qui permettent les communications avec les clients selon, entre autres, le protocole HTTP. Les JSP sont notamment utilisées pour générer les pages HTML qui seront visualisées par le client. Dans une vision trois tiers, ce bloc représente le tiers présentation.

Figure 10.1
Mise en œuvre de J2EE



- **EJB.** Permet de représenter sous forme de composants les traitements et les données des entreprises. Nous verrons à la section suivante que les EJB se déclinent en deux types, les EJB Session et les EJB Entity. Ils permettent respectivement de représenter les traitements et les données. Cela explique pourquoi nous avons représenté deux sous-parties (ellipses) sur la figure. Les EJB sont connectés aux bases de données de l'entreprise. Dans une vision trois tiers, ce bloc représente donc les tiers données et traitement.

Le scénario classique, pour une application J2EE, est le suivant :

1. Le client demande à visionner une page JSP.
2. La page JSP demande aux EJB (Session Beans) de réaliser les traitements correspondants.
3. Les EJB Session questionnent les EJB Entity pour obtenir les données de l'entreprise.
4. Les EJB Entity connectés à la base de données retournent les données aux EJB Session.
5. Les EJB Session effectuent leur traitement et retournent le résultat du traitement à la page JSP.
6. La page JSP présente au client le résultat sous forme de page HTML.

Les servlets/JSP

Le bloc servlets/JSP gère toutes les interactions avec les clients. Il doit donc supporter :

- L'hétérogénéité des clients, afin de pouvoir interagir avec les clients qui utilisent leur navigateur Web comme avec ceux qui utilisent leur téléphone portable.
- La notion de session, afin d'établir un lien personnel avec chaque client et de suivre ainsi sa propre logique d'exécution. Par exemple, la notion de session appliquée à une application de commerce électronique permet d'offrir à chaque client son propre panier virtuel et son propre environnement de commande de produits.
- La sécurité, afin d'authentifier les clients et de ne leur autoriser l'accès qu'aux ressources pour lesquelles ils possèdent des droits.
- La montée en charge, afin de permettre l'accès à l'application quel que soit le nombre de clients connectés.

Concrètement, les servlets sont des objets Java qui fonctionnent en mode requête/réponse. Les servlets les plus utilisées sont les servlets HTTP, qui acceptent les requêtes HTTP des clients et leur renvoient les réponses. Bien souvent les servlets HTTP renvoient des pages HTML comme réponses. Ces pages HTML sont alors visionnées par les clients.

Les pages JSP sont une façon syntaxique d'écrire les servlets. Une page JSP est d'ailleurs automatiquement traduite en une servlet par le serveur d'applications lors de l'exécution de l'application. Les pages JSP permettent d'exprimer plus facilement le résultat rendu par une servlet si celui-ci est au format textuel. Elles sont donc largement utilisées pour élaborer des servlets HTTP qui retournent des pages HTML.

Nous avons fait le choix de ne pas considérer le bloc servlets/JSP dans notre aperçu de la prise en compte de la plate-forme J2EE. En effet, dans la vision théorique de J2EE, ce bloc ne représente que le tiers présentation. La logique métier et les données sont représentées à l'aide d'EJB. De ce fait, les modèles UML sont plus naturellement transformés en EJB qu'en servlets ou en pages JSP. De plus, plusieurs outils de développement, tels que IBM WSAD (WebSphere Studio Application Developer), permettent de construire automatiquement des pages JSP à partir d'EJB.

Les EJB

Le bloc EJB contient les traitements et les données de l'entreprise. Il doit donc supporter les fonctionnalités suivantes :

- **Gestion de la mémoire.** Pour permettre l'accès à toutes les données de l'entreprise et assurer aux traitements un temps de réponse court, il est nécessaire de mettre en place des mécanismes de gestion de la mémoire.
- **Synchronisation avec la base de données.** Les bases de données offrant leurs propres mécanismes de cohérence, il est important de les synchroniser avec les données des EJB.
- **Transactions.** Étant donné que plusieurs données peuvent être manipulées lors d'une seule requête, il est important de mettre en place un mécanisme transactionnel permettant d'assurer des propriétés de cohérence.
- **Sécurité.** Il faut réaliser un contrôle strict des accès aux données de l'entreprise. L'authentification des clients étant faite au niveau du bloc servlets/JSP, il est nécessaire de pouvoir remonter à la source d'une requête pour vérifier ses droits d'accès.

Nous avons choisi de restreindre la prise en compte de la plate-forme J2EE aux EJB. Ces derniers sont des classes Java qui suivent certaines contraintes syntaxiques afin de pouvoir être exécutées sur les serveurs d'applications. Ils contiennent toute la logique et les données d'une application d'entreprise. La transformation d'un modèle UML se fait donc naturellement vers des EJB.

Les EJB (Enterprise JavaBean)

Le JCP a standardisé le profil UML pour EJB, qui permet une prise en compte des EJB version 1.1 dans MDA. Nous nous appuyons sur ce standard dans la suite du chapitre.

La présente section décrit les différents concepts EJB 1.1. Le JCP travaille actuellement à la mise au point d'un profil UML pour EJB 2.1.

Les EJB sont des composants Java qui peuvent être assemblés entre eux afin de construire des applications d'entreprise.

Du point de vue de l'utilisateur, un EJB se caractérise par les éléments suivants :

- **Identifiant unique.** L'identifiant d'un EJB est fourni par le serveur d'applications pendant l'exécution de l'application.
- **Interface home.** Contient toutes les opérations permettant de gérer le cycle de vie de l'EJB (création, suppression, recherche, etc.).
- **Interface remote.** Contient toutes les opérations réalisées par l'EJB. Cette interface est dite remote parce qu'elle est accessible à distance.

Du point de vue du constructeur de serveur d'applications, les EJB sont des ensembles de classes Java respectant les contraintes syntaxiques suivantes :

- Un EJB est au moins défini par trois classes Java, l'interface home, l'interface remote et la classe d'implémentation. Pour chacune de ces classes, plusieurs règles syntaxiques doivent être respectées. Par exemple, toutes les méthodes de l'interface remote doivent jeter l'exception `java.rmi.RemoteException`.
- Un EJB doit être packagé dans une archive spéciale, appelée `ejb-jar`. Cette archive contient les classes définissant le composant ainsi que différents fichiers XML de configuration, que nous ne détaillerons pas.

Selon le rôle qu'ils jouent dans une application d'entreprise, les EJB peuvent être de différents types. Les EJB contenant les données de l'entreprise sont des Entity Beans tandis que les EJB réalisant les traitements sont des Session Beans. Les sections suivantes précisent ces notions.

Les Entity Beans

Comme expliqué précédemment, les Entity Beans sont des EJB qui contiennent les données de l'entreprise. Leur caractéristique principale est d'être fortement persistants. La durée de vie d'un Entity Bean est en principe infinie.

Un Entity Bean est un EJB qui dispose de ses interfaces home et remote et de sa classe d'implémentation.

L'interface home d'un Entity Bean contient une ou plusieurs opérations d'instanciation et une ou plusieurs opérations de recherche d'Entity Bean existant. Les opérations de recherche d'Entity Bean sont appelées des *finders*. Il est possible d'associer une clé primaire à un Entity Bean afin d'utiliser cette clé dans les finders.

Puisqu'un Entity Bean contient les données de l'entreprise, les opérations qu'il propose sont des opérations d'accès aux données (getters et setters). Son interface remote ne contient donc que des méthodes d'accès aux données.

La classe d'implémentation d'un Entity Bean réalise les méthodes de l'interface remote. Elle contient essentiellement la réalisation (le code) des méthodes d'accès aux données.

La persistance d'un Entity Bean peut être soit à la charge du Bean (Bean Managed Persistency), soit à la charge du serveur d'applications (Container Managed Persistency). Si elle est à la charge du Bean, il faut ajouter dans la classe d'implémentation le code

nécessaire à la prise en compte de la persistance. Si elle est à la charge du conteneur d'EJB, il suffit de le déclarer dans un fichier de configuration afin que le serveur d'applications sauvegarde le Bean sur un support persistant.

Dans une application classique de commerce électronique, les Entity Beans sont typiquement utilisés pour représenter le stock et les produits manipulés par l'application. Les Entity Bean seraient alors sauvegardés par le serveur d'applications dans une base de données.

Les Session Beans

Les Session Beans sont des EJB qui contiennent les traitements des applications d'entreprise. Contrairement aux Entity Beans, ils ne sont pas persistants. Leur durée de vie est variable et dépend du type de traitement qu'ils effectuent.

Comme un Entity Bean, un Session Bean dispose de ses interfaces home et remote et de sa classe d'implémentation.

L'interface home d'un Session Bean contient une ou plusieurs opérations d'instanciation. Dans le cas général, chaque utilisation d'un Session Bean nécessite une création préalable. Il faut passer pour cela par l'interface home et demander la création d'une nouvelle instance pour chaque utilisation.

L'interface remote d'un Session Bean contient toutes les opérations correspondant aux traitements d'entreprise réalisés par le Bean.

La classe d'implémentation d'un Session Bean réalise les méthodes de l'interface remote. C'est donc dans la classe d'implémentation que l'on trouve tout le code métier des traitements d'entreprise.

Un Session Bean peut être de type Stateless ou Stateful :

- **Session Bean Stateless.** N'a pas connaissance des différentes opérations qu'il a déjà exécutées (il n'a pas d'historique). Chaque exécution d'une opération doit donc être totalement indépendante des autres opérations. Un Session Bean Stateless n'a pas d'état. Il n'est donc pas possible de suivre les différentes requêtes faites par un même client.
- **Session Bean Stateful.** A connaissance des différentes opérations qu'il a déjà exécutées. On dit qu'il a un état. Il est donc possible de suivre les différentes requêtes faites par un même client.

Dans une application classique de commerce électronique, les Session Beans sont typiquement utilisés pour représenter les traitements de recherche de produits et les traitements de gestion de panier. Un panier est généralement représenté par un Session Bean Stateful, car il doit garder en mémoire les ajouts de produits effectués.

Le profil UML pour EJB

Le profil UML pour EJB a été standardisé par le JCP en 2001. Il a été réalisé pour les versions EJB 1.1 et UML1.3. L'objectif principal de ce profil est de permettre de modéliser

les applications EJB en utilisant UML et de faciliter la génération du code des composants EJB.

Pour les concepteurs du standard, le choix d'un profil UML plutôt que d'un métamodèle MOF pour prendre en compte les EJB est dû au rapprochement sémantique d'UML et des EJB. Selon ces concepteurs, UML est très souvent utilisé pour modéliser des applications qui s'exécutent sur des plates-formes telles que J2EE/EJB. La sémantique d'UML recouvre donc en partie celle des EJB. Le profil permet de préciser ces rapprochements sémantiques.

Le second argument en faveur de la définition d'un profil UML est la disponibilité des outils. À l'époque de la création de ce standard, en 2001, plusieurs outils supportaient la définition des profils UML. Il était donc envisageable de profiter de ces outils pour mettre en œuvre le profil. À cette même époque, aucun outil commercial ne supportait les métamodèles MOF, si bien que la définition d'un métamodèle MOF pour EJB n'a jamais été réellement envisagée.

Le profil UML pour EJB définit un ensemble de stéréotypes, tagged-values et contraintes permettant d'adapter UML à la technologie EJB. Les modèles UML profilés selon ce profil sont donc adaptés à cette technologie. Dans la terminologie MDA, ces modèles sont dits des PSM pour EJB.

Le profil UML pour EJB a été initialement élaboré pour les concepteurs habitués à UML et voulant modéliser la façon dont leurs applications s'exécutaient sur la plate-forme J2EE/EJB. Ces utilisateurs utilisaient directement les stéréotypes et tagged-values du profil dans leurs modèles UML. Leur objectif était avant tout de générer une partie du code des EJB.

Avec le développement de l'approche MDA, ce profil est de plus en plus utilisé comme un profil de PSM. Les modèles UML profilés selon ce profil sont des PSM générés automatiquement à partir de PIM. Cette génération est actuellement partielle, et il faut toujours raffiner ces PSM pour exploiter pleinement les capacités de la technologie EJB.

Nous avons déjà indiqué que ce profil ne supportait que la version EJB 1.1 et qu'il a été réalisé pour UML1.3. Les travaux d'actualisation de ce profil pour les versions EJB 2.1 et UML2.0 sont toujours en cours de réalisation au JCP. Pour utiliser ce profil avant la sortie de son actualisation, l'approche que nous avons suivie consiste à transformer les modèles UML2.0 en modèles UML1.3 puis à transformer ces derniers en modèles profilés selon le profil UML pour EJB. Pour faire cela, nous ne pouvons utiliser que les concepts de la plate-forme EJB 1.1, ce qui n'est pas gênant pour notre propos.

Dans la suite du chapitre nous présentons la partie structurelle du profil UML pour EJB, l'autre partie concernant le déploiement des applications EJB. Cette partie est suffisante pour illustrer la façon dont la plate-forme EJB est prise en compte par MDA et permettre la réalisation de l'étude de cas du chapitre 12.

Structure du profil

Cette section détaille les extensions UML (stéréotypes, tagged-values et contraintes) apportées par le profil UML pour EJB en expliquant leur signification.

Rappelons que par souci de simplicité, nous n'avons pas considéré les stéréotypes et tagged-values concernant le déploiement des composants EJB.

Les stéréotypes

Les stéréotypes suivants sont applicables aux classes UML (instances de la métaclasse `Class`) :

- `EJBRemoteInterface`. Une classe UML qui porte ce stéréotype correspond à une interface remote d'un EJB.
- `EJBHomeInterface`. Une classe UML qui porte ce stéréotype correspond à une interface home d'un EJB.
- `EJBSessionHomeInterface`. Une classe UML qui porte ce stéréotype correspond à une interface home d'un EJB de type Session. Ce stéréotype raffine le stéréotype `EJBHome Interface`.
- `EJBEntityHomeInterface`. Une classe UML qui porte ce stéréotype correspond à une interface home d'un EJB de type Entity. Ce stéréotype raffine le stéréotype `EJBHome Interface`.
- `EJBImplementation`. Une classe UML qui porte ce stéréotype correspond à une classe d'implémentation d'un EJB.

Les stéréotypes suivants sont applicables sur les opérations UML (instance de la métaclasse `Operation`) :

- `EJBCreateMethod`. Une opération UML qui porte ce stéréotype correspond à une opération de création d'un EJB.
- `EJBFinderMethod`. Une opération UML qui porte ce stéréotype correspond à une opération finder (opération de recherche pour les Entity Beans existants).
- `EJBRemoteMethod`. Une opération UML qui porte ce stéréotype correspond à une opération accessible à distance.

Les stéréotypes suivants sont applicables aux attributs UML (instance de la métaclasse `Attribute`) :

- `EJBCmpField`. Un attribut UML qui porte ce stéréotype correspond à un attribut persistant d'un EJB de type Entity Bean.
- `EJBPrimaryKeyField`. Un attribut UML qui porte ce stéréotype correspond à une clé primaire d'un EJB Entity Bean.

Les stéréotypes suivants sont applicables aux associations UML (instance de la méta-classe `Association`) :

- `EJBReference`. Une association UML qui porte ce stéréotype permet de définir une référence entre deux EJB.
- `EJBRealizeHome`. Une association UML qui porte ce stéréotype permet d'établir un lien entre la classe d'implémentation d'un EJB et son interface home.
- `EJBRealizeRemote`. Une association UML qui porte ce stéréotype permet d'établir un lien entre la classe d'implémentation d'un EJB et son interface remote.

Les tagged-values

Les tagged-values suivantes sont applicables aux classes stéréotypées `EJBSessionHome Interface` et `EJBEntityHomeInterface`. Elles permettent de préciser les caractéristiques des EJB Session et Entity.

- `EJBSessionType`. Cette tagged-value, portée par les classes stéréotypées `EJBSessionHome Interface`, peut prendre comme valeur `Stateful` ou `Stateless`. Cela permet de spécifier le type de Session Bean.
- `EJBPersistentType`. Cette tagged-value, portée par les classes stéréotypées `EJBEntity HomeInterface`, peut prendre comme valeur `Bean` ou `Container`. Cela permet de spécifier la façon dont est gérée la persistance de l'Entity Bean.

Les contraintes

Les stéréotypes et tagged-values que nous venons de présenter ont une certaine cohérence entre eux. Cette cohérence est exprimée dans le profil grâce à des contraintes. Nous présentons ci-dessous quelques contraintes de ce profil :

- Toute classe stéréotypée `EJBImplementation` doit être liée par une association stéréotypée `EJBRealizeHome` avec une classe stéréotypée `EJBHomeInterface` et doit être liée par une association stéréotypée `EJBRealizeSession` avec une classe stéréotypée `EJBRemoteInterface`. Cette contrainte permet de s'assurer qu'un EJB est bien constitué d'une interface home, d'une interface remote et d'une classe d'implémentation.
- Toute classe stéréotypée `EJBSessionHomeInterface` doit porter la tagged-value `EJBSession Type`. Cette contrainte permet de s'assurer qu'un Session Bean est soit Stateful soit Stateless.
- Toute classe stéréotypée `EJBEntityHomeInterface` doit porter la tagged-value `EJBPersistent Type`. Cette contrainte permet de s'assurer que la persistance d'un Entity Bean est gérée soit par le serveur d'applications, soit par le Bean lui-même.
- Toute classe stéréotypée `EJBHomeInterface` doit avoir au moins une méthode stéréotypée `EJBCreateMethode`. Cette contrainte permet de s'assurer que l'interface home d'un EJB contient une méthode de création.

- Toute classe stéréotypée `EJBEntityHomeInterface` peut avoir des opérations stéréotypées `EJBFinderMethod`. Cette contrainte permet de s'assurer que l'interface home d'un Entity Bean peut contenir des opérations de recherche (finders) de Beans existants.
- Toute classe stéréotypée `EJBRemoteInterface` doit avoir au moins une méthode stéréotypée `EJBRemoteMethode`. Cette contrainte permet de s'assurer que l'interface remote d'un EJB contient bien une ou plusieurs méthodes accessibles à distance.
- Toute classe stéréotypée `EJBImplementation` liée à une classe stéréotypée `EJBEntity HomeInterface` peut posséder des attributs stéréotypés `EJBCMPField` et au plus un attribut stéréotypé `EJBPrimaryKeyField`. Cette contrainte permet de s'assurer que la classe d'implémentation d'un Entity Bean peut avoir des attributs persistants et au plus une clé primaire.

Génération du code

Les stéréotypes, tagged-values et contraintes que nous venons de présenter permettent de construire des modèles UML modélisant des applications EJB. Cette section présente le code qui peut être généré à partir de tels modèles.

L'interface home

Toute classe stéréotypée `EJBHomeInterface` donne lieu à la création d'une interface Java représentant l'interface home d'un EJB. Cette interface Java respecte les contraintes syntaxiques du standard EJB. Par exemple, elle hérite de l'interface `javax.ejb.EJBHome`, et son nom correspond au nom de l'interface.

Les opérations stéréotypées `EJBCreateMethod` de cette classe donnent lieu à la création de méthodes dans l'interface Java. Ces méthodes respectent les contraintes syntaxiques du standard EJB. Par exemple, leur nom doit être `Create`.

Si la classe est stéréotypée `EJBEntityHomeInterface` et qu'elle contienne des opérations stéréotypées `EJBFinderMethod`, ces opérations donnent lieu à la création de méthodes finder dans l'interface Java. Ces méthodes respectent les contraintes syntaxiques du standard EJB. Par exemple, leur nom commence par `Find`.

L'interface remote

Toute classe stéréotypée par `EJBRemoteInterface` donne lieu à la création d'une interface Java représentant l'interface remote d'un EJB. Cette interface Java respecte les contraintes syntaxiques du standard EJB. Par exemple, elle hérite de l'interface `javax.ejb.EJBObject`.

Les opérations stéréotypées `EJBRemoteMethod` de cette classe donnent lieu à la création de méthodes dans l'interface Java. Ces méthodes respectent les contraintes syntaxiques du standard EJB. Par exemple, elles doivent toutes déclarer qu'elles jettent l'exception `javax.rmi.RemoteException`.

La classe d'implémentation

Toute classe stéréotypée par `EJBImplementation` donne lieu à la création d'une classe Java représentant la classe d'implémentation d'un EJB. Cette classe respecte les contraintes syntaxiques du standard EJB. Par exemple, elle doit posséder les méthodes `ejbActivate`, `ejbPassivate` et `ejbRemove`, qui seront appelées par le serveur d'applications lorsqu'il gérera la mémoire.

Une classe stéréotypée par `EJBImplementation` doit être reliée à une interface `EJBRemote Interface`. La classe Java correspondant à la classe d'implémentation doit contenir toutes les méthodes de l'interface correspondant à l'interface remote.

De même, une classe stéréotypée par `EJBImplementation` doit être reliée à une classe `EJBHomeInterface`. Pour chaque opération stéréotypée `EJBCreateMethod` contenue dans cette classe, la classe Java correspondant à la classe d'implémentation doit posséder une méthode Java correspondante.

Si la classe stéréotypée par `EJBImplementation` est reliée à une classe stéréotypée `EJBSession HomeInterface`, la classe d'implémentation Java doit implémenter l'interface `javax.ejb.Session Bean`. Si la classe stéréotypée par `EJBImplementation` est reliée par `EJBEntityHomeInterface`, la classe d'implémentation Java doit implémenter l'interface `javax.ejb.EntityBean`.

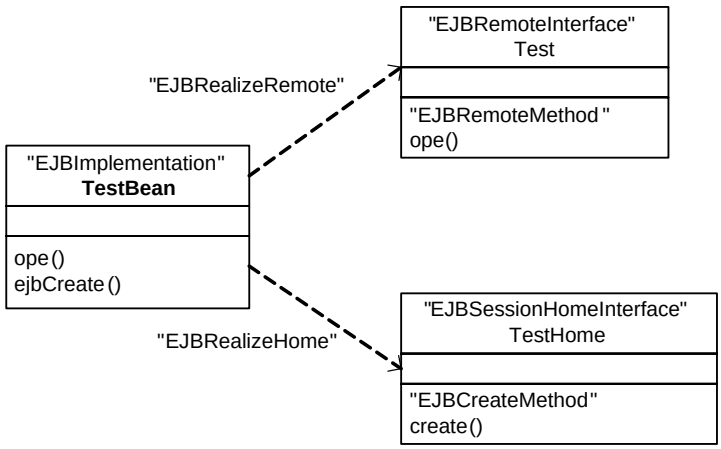
Exemple de mise en œuvre

Afin d'illustrer le profil UML pour Java, nous allons l'employer dans un exemple simple ne contenant qu'un seul EJB.

La figure 10.2 illustre le modèle UML de cet exemple. Suivant les principes du profil UML pour EJB, ce Session Bean est modélisé par trois classes UML :

- Une classe UML correspondant à l'interface remote du Bean. Cette classe, stéréotypée par `EJBRemoteInterface`, est nommée `Test`. Elle contient une unique opération accessible à distance, nommée `ope`, qui est stéréotypée par `EJBRemoteMethod`.

Figure 10.2
Illustration du profil UML pour EJB sur un exemple



- Une classe UML correspondant à l'interface home du Bean. Cette classe, stéréotypée par `EJBSessionHomeInterface`, est nommée `TestHome`. Elle contient une unique opération de création, nommée `create`, qui est stéréotypée par `EJBCreateMethod`.
- Une classe UML correspondant à la classe d'implémentation du Bean. Cette classe, stéréotypée par `EJBImplementation`, est nommée `TestBean`. Elle est reliée aux deux autres classes par des associations stéréotypées `EJBRealizeRemote` et `EJBRealizeHome`.

Ce modèle UML profilé selon le profil UML pour EJB facilite la génération du code correspondant au Bean.

L'interface remote du Bean permet la génération automatique du code suivant :

```
import javax.ejb.*;

public interface Test extends EJBObject
{
void ope() throws java.rmi.RemoteException;
}
```

L'interface home du Bean permet la génération automatique du code suivant :

```
import javax.ejb.*;

public interface TestHome extends EJBHome
{
Test create()
        throws java.rmi.RemoteException,javax.ejb.CreateException;
}
```

La classe d'implémentation permet la génération automatique du code suivant :

```
import javax.ejb.*;

public class TestBean implements SessionBean
{
public SessionContext context;


    public void ope() throws java.rmi.RemoteException
{
}

public void ejbActivate() {}

    public void ejbPassivate(){}

    public void ejbRemove(){}

    public void setSessionContext(SessionContext ctx)
{
context = ctx;
    }
```

```
    public SessionContext getSessionContext()
    {
        return context;
    }

    public void ejbCreate()
        throws javax.ejb.CreateException,java.rmi.RemoteException
{
}
}
```

Le code généré respecte intégralement les contraintes syntaxiques de la plate-forme J2EE. Le concepteur peut donc l'utiliser et y intégrer son code métier afin de finir la création de son application.

Transformations PIM vers PSM

Afin de bénéficier du profil UM pour EJB dans l'étude de cas du chapitre 12, nous avons élaboré une transformation PIM vers PSM permettant de transformer un modèle UML2.0 en un modèle UML1.3 profilé selon le profil UML pour EJB.

Nous avons défini cette transformation en deux étapes. La première correspond à la transformation des modèles UML2.0 en modèles UML1.3, et la seconde à la transformation des modèles UML1.3 vers des modèles UML profilés selon le profil UML pour EJB.

UML2.0 vers UML1.3

UML2.0 apporte de nombreuses nouveautés par rapport à UML1.3. Réaliser une transformation UML2.0 vers UML1.3 n'a de sens que parce que nous voulons bénéficier du profil UML pour EJB. Cette transformation est donc une transformation contextuelle à l'objectif bien précis.

Les éléments UML1.3 qui nous intéressent sont les classes, leurs attributs, leurs opérations et leurs associations. En ce qui concerne UML2.0, nous comptons réaliser nos PIM avec des composants UML. Notre transformation UML2.0 vers UML1.3 est donc une transformation des composants UML2.0 vers des classes UML1.3.

La principale différence entre un composant UML2.0 et une classe UML1.3 est qu'un composant a une partie interne et une partie externe. Un composant UML2.0 a donc des *parts* (éléments de sa partie interne) et des *ports* (éléments de la partie externe). Cet aspect est clairement défini dans le métamodèle UML2.0 *(voir le chapitre 3)* et n'apparaît pas dans le métamodèle UML1.3.

En UML1.3, il n'est pas possible de distinguer la partie interne et la partie externe d'une classe. Il est uniquement possible de préciser les visibilités publiques et privés des attributs et des opérations et de définir les navigabilités des associations entre les classes. La solution consiste donc à générer, à partir d'un composant UML2.0, un ensemble de classes UML

avec différents attributs, opérations et associations, tout en précisant les visibilités permettant de respecter la distinction entre la partie interne et la partie externe du composant.

Règles de la transformation

Nous définissons ici en langage naturel les règles de transformation permettant de transformer des composants UML2.0 vers des classes UML1.3.

Composant

Règle 1. *À un composant UML2.0 correspond une classe UML1.3. Cette classe porte le même nom que le composant.*

Ports

Un composant UML2.0 peut posséder plusieurs ports. Chaque port peut offrir ou requérir une ou plusieurs interfaces. Les ports d'un composant représentent la partie externe du composant. Les règles suivantes offrent différentes solutions pour la transformation des ports vers UML1.3.

Règle 2.a. *À chaque port d'un composant correspond une classe UML (le nom de la classe correspond au nom du port). Cette classe UML réalise les interfaces offertes du port. Cette classe dépend des interfaces requises par le port. La classe du composant est reliée à des associations d'agrégation vers les classes des ports.*

Règle 2.b. *La classe correspondant au composant réalise toutes les interfaces offertes par les ports du composant et dépend de toutes les interfaces requises par les ports du composant.*

La figure 10.3 illustre ces deux solutions appliquées à un exemple de composant contenant deux ports (`P1` et `P2`). `P1` offre l'interface `I1` et requiert l'interface `I3`. `P2` offre l'interface `I2`.

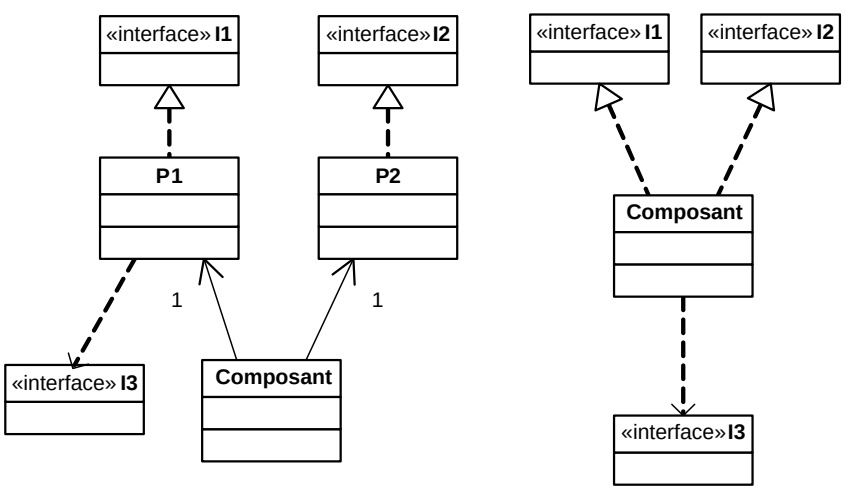


Figure 10.3
Les deux façons de traduire les ports en UML1.3

La règle 2.a génère le résultat de la partie gauche de la figure et la règle 2.b celui de la partie droite.

☛ Parts

Un composant UML2.0 peut posséder plusieurs parts. Chaque part est typée par un composant ou une classe et dispose d'une multiplicité (nombre de parts dans la partie interne du composant). Les règles suivantes offrent différentes solutions pour la transformation des parts vers UML1.3

Règle 3.a. *À chaque part d'un composant correspond un attribut dans la classe UML du composant. Cet attribut a le même type que le type de la part (ou le type transformé s'il s'agit d'un composant). La multiplicité de l'attribut correspond à la multiplicité de la part.*

Règle 3.b. *À chaque part d'un composant correspond une association navigable allant de la classe UML du composant vers la classe correspondant au type de la part. La multiplicité de l'association correspond à la multiplicité de la part.*

La figure 10.4 illustre ces deux solutions appliquées à un exemple de composant contenant deux parts (`partA` et `partB`).

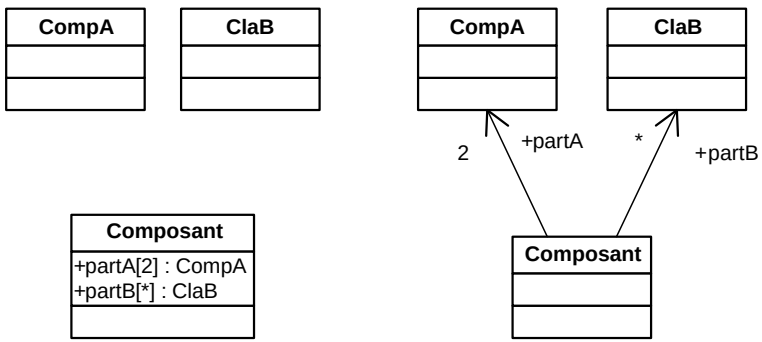


Figure 10.4
Les deux façons de traduire les parts en UML1.3

☛ Connecteur

Un connecteur UML2.0 permet de lier deux éléments (part et port). Selon les types des éléments liés, des règles différentes s'appliquent lors de la transformation vers UML1.3.

Si un connecteur lie deux ports, deux possibilités se présentent. Si la règle 2.a est utilisée, il est possible de construire en UML1.3 une association entre les classes correspondant aux ports.

Règle 4. *À un connecteur entre deux ports correspond une association entre les classes correspondant aux ports.*

Si la règle 2.b est utilisée, il n'est pas possible de construire une association sur les classes car cela modifierait la sémantique des composants en plus du port. Le connecteur ne peut en ce cas être représenté en UML1.3.

Si un connecteur lie une part avec un port ou une part, le connecteur ne peut être représenté en UML1.3. En effet, il faudrait ajouter une association sur les classes et ainsi modifier la sémantique des composants. Par exemple, s'il existait un connecteur entre les parts (`partA` et `partB`) de l'exemple présenté à la figure 10.4, le fait d'ajouter une association entre les classes `CompA` et `ClaB` modifierait la sémantique de l'application.

Le fait de ne pas pouvoir représenter les connecteurs UML1.3 n'est pas bloquant. En UML1.3, tout élément est accessible (en fonction de sa visibilité). Les connecteurs ne faisant que contraindre les accès, les modèles UML1.3 correspondants sont donc plus permissifs.

Le modèle intermédiaire

La transformation que nous venons de présenter comporte deux alternatives (règles 2.a/2.b et règles 3.a/3.b). Ces alternatives engendrent des structurations totalement différentes dans les modèles cibles, lesquelles doivent être considérées dans les modèles intermédiaires *(voir le chapitre 9).*

En conformité avec les principes énoncés au chapitre précédent, la structuration des modèles intermédiaires peut s'exprimer par l'élaboration d'un nouveau profil. Ce profil contient la définition d'une tagged-value par solution de l'alternative considérée, à savoir :

- Une tagged-value pour le choix entre les règles 2.a et 2.b. Cette tagged-value, que l'on peut nommer `portAlternative`, est attachée aux composants UML2.0 et peut prendre, par exemple, les valeurs `classe` ou `fusion` permettant d'identifier respectivement l'application de la règle 2.a ou de la règle 2.b.
- Une tagged-value pour le choix entre les règles 3.a et 3.b. Cette tagged-value, que l'on peut nommer `partAlternative`, est attachée aux composants UML2.0 et peut prendre les valeurs `attributs` ou `association` permettant d'identifier respectivement l'application de la règle 3.a ou de la règle 3.b.

Grâce à ce profil, il sera possible de transformer automatiquement un modèle UML2.0 vers un modèle UML1.3. Pour ce faire, il faudra que, pour chaque composant, les tagged-values `portAlternative` et `partAlternative` soient spécifiées.

UML1.3 vers UML EJB

La transformation UML1.3 vers UML EJB que nous envisageons consiste à générer automatiquement un modèle UML profilé selon le profil UML pour EJB. Le point important à considérer est que le modèle UML source de la transformation a été généré à partir d'un modèle de composants UML2.0.

Puisque les composants EJB n'ont pas de partie interne, la transformation que nous considérons ne se soucie pas des parts. De plus, comme les composants EJB n'ont qu'une unique interface ne contenant que des opérations offertes (interface remote), cette transformation ne se soucie que des ports et de leurs interfaces offertes.

Nous avons vu à la section précédente que la transformation UML2.0 vers UML1.3 générait des classes pour les composants mais aussi pour les ports (selon l'application de la règle 2.a ou de la règle 2.b). Il est important de pouvoir distinguer ces deux types de classes, car, conceptuellement, elles ne représentent pas les mêmes entités et seront donc traitées différemment.

Pour prendre en compte cette distinction, nous avons élaboré les règles suivantes :

Règle A. *Si une classe UML représente un composant UML2.0 et qu'elle ne soit pas liée à des classes représentant des ports, signifiant que la règle 2.b a été appliquée, cette classe donne lieu à la création d'un EJB et donc à la création des trois classes UML interface remote, interface home et classe d'implémentation.*

Règle B. *Si une classe UML représente un composant UML2.0 et qu'elle soit liée à des classes représentant des ports, signifiant que la règle 2.a a été appliquée, chaque classe représentant un port donne lieu à la création d'un EJB. Les classes d'implémentation des EJB doivent en ce cas être reliées à la classe UML représentant le composant UML.*

Ces deux règles sont suffisantes pour générer un modèle profilé UML pour EJB à partir d'un modèle UML1.3 représentant des composants UML2.0.

Le modèle intermédiaire

Les deux règles ci-dessus nécessitent de connaître les classes qui représentent des composants et celles qui représentent des ports. Ces informations doivent apparaître dans les modèles intermédiaires.

Pour ce faire, le profil des modèles intermédiaires doit inclure deux stéréotypes pour représenter ces informations : un stéréotype nommé, par exemple, `composant`, qui sera porté par les classes représentant un composant, et un stéréotype nommé, par exemple, `port`, qui sera porté par les classes représentant des ports. Ces stéréotypes devraient être ajoutés automatiquement par la transformation UML2.0 vers UML1.3.

Nous avons vu que ces deux règles généraient les classes UML nécessaires à la modélisation de composants EJB (classe `remote`, classe `home` et classe d'implémentation). Le choix des différents stéréotypes et tagged-values à appliquer à ces classes pour définir les notions de Session Bean et d'Entity Bean dépend uniquement de la façon dont le concepteur du modèle UML souhaite exploiter la plate-forme J2EE.

Ces choix ne sont aucunement liés à la structure du modèle source et n'entraînent pas de modifications structurelles. Même s'il n'est pas nécessaire de les faire apparaître dans le modèle intermédiaire, il peut être intéressant d'indiquer le type d'EJB à construire (Session ou Entity). Nous avons donc intégré dans le profil une tagged-value portée sur les classes et nommée `Bean`. Les valeurs de cette tagged-value peuvent être `session` ou `entity` de façon à savoir quel type d'EJB générer.

Ces règles nécessitent de connaître les opérations et attributs des classes UML qui devront être considérés comme des opérations et des attributs des EJB. En effet, toutes les opérations et tous les attributs d'un composant UML2.0 ne doivent pas nécessairement apparaître dans le composant EJB. Ces informations doivent en revanche apparaître dans les modèles intermédiaires car leur utilisation modifie la structure des modèles cibles générés.

Pour ce faire, le profil des modèles intermédiaires doit inclure deux stéréotypes : un stéréotype nommé, par exemple, `opérationEJB`, qui sera porté par les opérations devant apparaître dans l'EJB, et un stéréotype nommé, par exemple, `attributEJB`, qui sera porté par les attributs devant apparaître dans l'EJB.

Exemple de mise en œuvre de la transformation UML2.0 vers EJB

Afin de bien faire comprendre la transformation UML vers EJB que nous venons de présenter, nous proposons de l'illustrer par un exemple simple.

La figure 10.5 représente un modèle de composant UML2.0. Ce composant est un distributeur de boissons. Il offre des opérations d'utilisation (sélectionner, payer et retirer la boisson) et des opérations d'administration (faire l'inventaire, ajouter du stock, enlever du stock) *via* ses deux ports.

La figure ne permet pas de visualiser la tagged-value `portAlternative` attachée au composant. Celle-ci permet de spécifier la solution « classe » pour la transformation vers un modèle UML1.3. Grâce à cette information, ce composant UML2.0 peut être transformé automatiquement vers l'ensemble de classes UML1.3 illustrées à la figure 10.6.

Ce modèle UML1.3 contient une classe pour le composant et une classe par port. Il contient aussi les tagged-values permettant d'identifier le type d'EJB à générer. Ces stéréotypes, qui ne sont pas affichés sur la figure, précisent que les ports doivent générer des Session Beans.

La figure 10.7 illustre le modèle UML EJB généré à partir du modèle de classes UML1.3. Nous constatons que chaque port a donné lieu à la création d'un Session Bean. Ce modèle est bien un PSM, et il précise la façon dont le composant UML2.0 `Boisson` utilise la plate-forme J2EE.

À partir de ce modèle UML EJB, il est possible de générer les squelettes de code de l'application. Ces squelettes seront complétés par le concepteur lorsqu'il souhaitera y intégrer son code métier.

Figure 10.5
Exemple de modèle de composant UML2.0

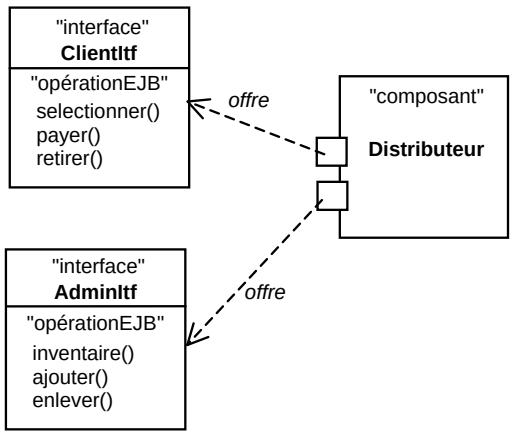
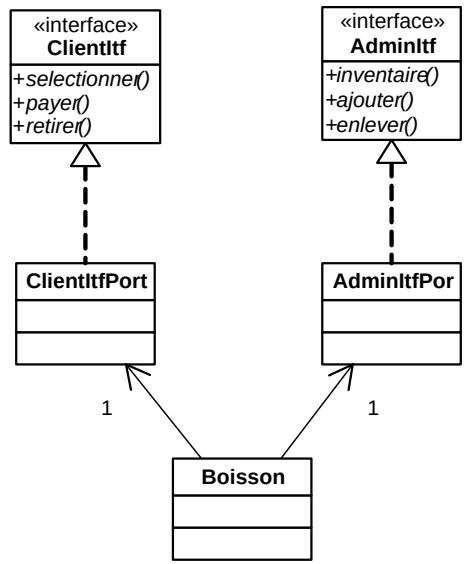


Figure 10.6
Modèle UML1.3 généré à partir du modèle de composant UML2.0



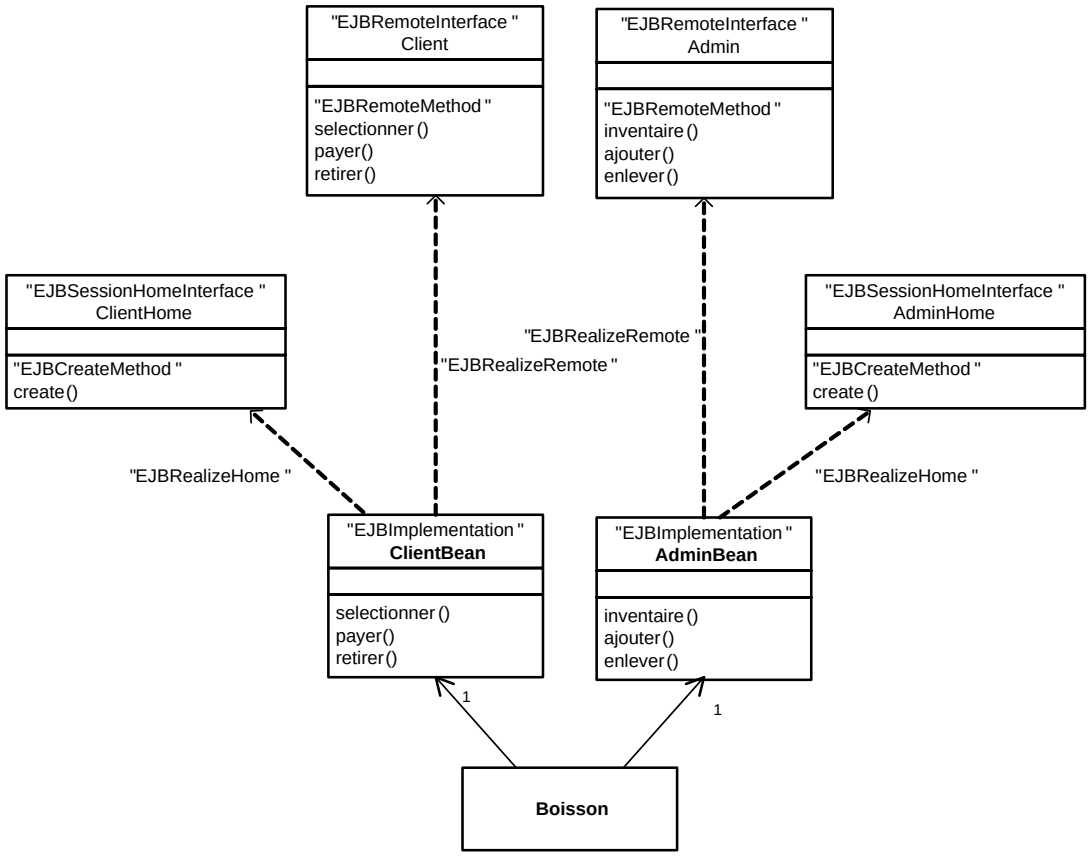


Figure 10.7
Modèle UML EJB généré à partir du modèle UML1.3

Synthèse

Nous venons de présenter brièvement la plate-forme J2EE et avons précisé la façon dont elle est prise en compte par MDA. Cette prise en compte passe par l'utilisation du profil UML pour EJB standardisé par le JCP (Java Community Process).

Nous n'avons pas traité l'intégralité de la plate-forme J2EE et avons volontairement simplifié sa prise en compte. Notre objectif était uniquement de démontrer la faisabilité de l'approche et non pas de couvrir l'étendue de la plate-forme.

En plus du profil UML pour EJB, nous avons présenté une transformation permettant d'exploiter ce profil à partir de modèles UML2.0. Cette transformation se fait en deux temps : une transformation UML2.0 vers UML1.3 puis une transformation UML1.3 vers EJB.

Grâce à ce profil et à cette transformation, il est possible d'appliquer l'approche MDA à des PIM élaborés sous forme de composants UML2.0 afin de générer des PSM pour la plate-forme EJB. Ces PSM facilitent la génération de code.

Nous utiliserons au chapitre 12 le profil UML pour EJB et la transformation que nous venons de présenter en les appliquant à une étude de cas. L'objectif de cette étude de cas est de construire le PIM de l'application PetStore de Sun et de générer automatiquement le PSM pour EJB.

11

La plate-forme PHP

L'objectif de ce chapitre est de présenter la façon dont la plate-forme PHP est prise en compte par MDA. Cette prise en compte est propriétaire en ce sens que nous l'avons définie pour les besoins de notre étude de cas *(voir le chapitre 12)*. Rappelons que cette étude de cas vise à illustrer les principes de MDA en construisant le PIM de l'application PetStore de Sun afin de le transformer en deux PSM, l'un pour la plate-forme J2EE et l'autre pour la plate-forme PHP.

Ce chapitre présente donc la plate-forme PHP. Comme pour J2EE, seuls les concepts élémentaires de cette plate-forme, permettant de comprendre sa prise en compte par MDA, sont exposés.

Après quelques rappels sur PHP, nous introduisons le métamodèle de la plate-forme PHP que nous avons élaboré. Les modèles instances de ce métamodèle représentent les applications qui utilisent les services de la plate-forme PHP. Nous détaillons ensuite la transformation PIM vers PSM, qui permet de transformer des modèles de composant UML2.0 vers des modèles PHP. Cette transformation sera réutilisée dans l'étude de cas au chapitre suivant.

Rappels sur la plate-forme PHP

PHP (Hypertext PreProcessor) est un langage de programmation qui a été initialement élaboré pour permettre le développement de sites Web dynamiques. Sa principale caractéristique est de permettre l'intégration de traitements dans les pages Web HTML (calculs de valeurs, accès à des bases de données, modification de styles d'affichage, etc.). Ces traitements ont pour objectif de modifier dynamiquement les pages HTML et sont exécutés par le serveur Web avant l'envoi des pages vers le client.

Pour tous ses utilisateurs, PHP se distingue par sa simplicité d'utilisation. Langage de script à la syntaxe claire et simple, il est non typé, ce qui facilite grandement la gestion des variables (les conversions de types sont réalisées par le moteur d'exécution de PHP). De plus, ses capacités d'intégration dans les pages HTML permettent d'élaborer des traitements peu complexes aux résultats performants.

Pour autant, PHP n'est pas un langage simpliste. Il dispose notamment de plusieurs bibliothèques permettant de réaliser des traitements complexes. Les plus importantes d'entre elles sont certainement les bibliothèques d'accès aux bases de données. Définies pour quasiment toutes les bases de données du marché (Oracle, db2, MySQL, etc.), elles permettent de s'y connecter et d'exécuter des requêtes écrites, par exemple, en langage SQL (Structured Query Language).

D'autres bibliothèques sont définies, par exemple, pour la manipulation des documents XML, celle des fichiers ou la programmation réseau avec les sockets. La grande diversité de ces bibliothèques explique en grande partie le succès de PHP.

Depuis sa version 4.0, PHP supporte le paradigme objet et permet de définir des classes contenant des attributs, des constantes et des méthodes. Il est ainsi possible de construire des objets et de les utiliser dans les scripts.

PHP est un langage Open Source développé par une communauté de développeurs totalement indépendante. Perçu lors de sa création comme le langage idéal pour le développement de sites Web amateurs, il est devenu un langage incontournable, qui, après plusieurs années d'existence, est utilisé par les grandes entreprises pour le développement de leurs sites Web.

Signalons qu'Apache, le serveur HTTP le plus répandu du marché, intègre un moteur PHP sous forme de module. Le déploiement d'applications PHP s'effectue donc très simplement et très naturellement sur ce type de serveur. Cela explique aussi le succès de PHP.

Architecture de PHP

PHP est un langage qui permet le développement de scripts intégrables dans des pages HTML. On appelle page PHP une page HTML contenant des scripts PHP. Tous les scripts d'une page PHP sont exécutés séquentiellement par le serveur Web lorsqu'un client demande à visionner la page.

Le résultat de l'exécution des scripts donne lieu à la création d'une page HTML, qui est renvoyée au client. Le client n'a donc jamais conscience des traitements qui sont réalisés par les scripts PHP. Il ne voit que le résultat du traitement, c'est-à-dire la page HTML générée.

La page PHP

L'exemple suivant présente une page PHP. Cette page intègre un script PHP identifié par les balises `< ?php` et `?>`. Ce script ne fait qu'afficher la chaîne de caractères « ma première page PHP ».

```
<html>
    <body>
        <?php
echo "ma première page PHP" ;
?>
    <body>
</html>
```

Lorsqu'un client demande à visionner la page PHP présentée ci-dessus, le serveur exécute le script PHP et retourne le résultat suivant :

```
<html>
    <body>
       ma première page PHP
<body>
</html>
```

Variables, fonctions et constantes

Les scripts PHP peuvent définir des variables, des constantes et des fonctions. Ces définitions sont visibles par défaut dans toute la page PHP. Une fois les éléments définis, il est possible de les utiliser n'importe où dans la page.

Les variables sont par défaut non typées. Le type d'une variable est calculé dynamiquement en cous d'exécution, en fonction de la valeur portée. Par exemple, le script PHP suivant déclare une variable `a` et lui affecte successivement les valeurs `12` et `bonjour`. Le type de cette variable passe donc d'entier à chaîne de caractères !

```
<?php
$a ;
$a = 12 ;
    $a = "bonjour" ;
?>
```

Les fonctions peuvent avoir des paramètres non typés. Une fonction peut rendre une valeur après avoir été exécutée, mais cela n'est jamais déclaré dans la définition de la fonction.

Par exemple, le script PHP suivant contient deux fonctions, dont l'une retourne une valeur (`somme`) et l'autre pas (`direBonjour`) :

```
<?php
function somme($a , $b) {
return $a+$B ;
}

function direBonjour() {
        echo "Bonjour" ;
}
?>
```

Les constantes ne sont pas typées. La définition d'une constante se fait par le biais de la fonction `define`. L'exemple suivant permet de définir la constante `TITRE`, dont la valeur est `MDA en action` :

```
<?php
define("TITRE" , "MDA en action") ;
?>
```

Le mécanisme de session

Étant donné que les variables ne sont visibles qu'à l'intérieur d'une page PHP, il n'est pas possible de les transmettre entre deux pages PHP. Autrement dit, si une variable est définie dans une page PHP, il n'est pas possible de l'utiliser dans une autre page.

Pour résoudre ce problème, qui a pour origine une lacune du protocole HTTP, PHP propose le mécanisme de *session*. Une session est un espace mémoire créé chez le serveur pour un client particulier, qui permet de stocker des variables. Ces variables sont accessibles par toutes les pages PHP visionnées par le client.

Par exemple, les scripts suivants appartiennent à deux pages PHP différentes, qui seront visualisées par un même client. La première définit une variable `$couleur` qui sera utilisée par la seconde.

```
<?php
//démarrage de la session
session_start() ;

    //création d'une variable et inscription dans la session
$_SESSION['couleur'] = "rouge" ;
?>
```

```
<?php
echo "la valeur couleur est :" + $_SESSION['couleur'] ;
?>
```

Ce mécanisme de session permet de partager des variables entre plusieurs pages PHP.

L'inclusion

Afin de répartir les scripts PHP dans plusieurs fichiers et de faciliter de la sorte leur réutilisation, PHP définit un mécanisme d'*inclusion*.

Grâce à la fonction `include()`, il est possible d'inclure l'intégralité d'une page PHP dans un script. Si la page incluse contient des définitions de fonctions, il est alors possible d'utiliser ces fonctions dans les scripts.

L'exemple de script suivant inclut la page `mesFonctions.php`. Cette page contient plusieurs définitions de fonctions, telles que la fonction `somme` présentée précédemment, que le script pourra utiliser.

```
<?php
include 'mesFonctions.php' ;
    somme(1,2) ;
?>
```

Connexion aux bases de données

Nous avons déjà souligné que PHP définissait un ensemble de bibliothèques telles que la bibliothèque de manipulation de fichier ou celle de création d'image. Une bibliothèque est un fichier qui contient plusieurs définitions de variables, constantes, fonctions et classes.

Lorsqu'un script a besoin d'utiliser une bibliothèque, il doit inclure le fichier correspondant. PHP dispose pour cela d'un fichier de configuration, qui permet de déclarer toutes les bibliothèques incluses par défaut dans les scripts PHP.

La bibliothèque d'accès aux bases de données est la plus utilisée. Elle permet d'établir des connexions à des bases de données afin de demander l'exécution de requêtes SQL. Comme il s'agit d'une des bibliothèques incluses par défaut dans les scripts PHP, il n'est pas nécessaire de l'inclure dans un script avant de l'utiliser.

L'exemple suivant illustre l'utilisation de cette bibliothèque pour se connecter à une base de données MySQL. La variable `$link` représente la connexion vers la base de données. Cette variable sera utilisée pour demander l'exécution de requêtes SQL.

```
<?php
$link = mysql_connect('sql.lip6.fr' , 'monlogin' , 'password') ;
?>
```

Les objets

Depuis la version 4.0, PHP supporte le paradigme objet. Il est possible de définir des classes contenant des méthodes et des attributs et de construire des objets instances de ces classes.

PHP supporte l'héritage entre classes, mais un héritage non multiple, une classe ne pouvant hériter que d'une seule autre classe.

Les attributs des classes PHP peuvent être considérés comme des variables, si ce n'est qu'ils disposent d'une visibilité interne à la classe (public, private ou protected). Les attributs sont donc non typés.

Les méthodes des classes PHP peuvent être considérées comme des fonctions, mais elles disposent elles aussi d'une visibilité interne à la classe (public, private ou protected).

Le script suivant présente la définition d'une classe PHP :

```
<?php
class JoueurDEchec {
public $nom ;
public $niveau ;

public function joueCoup() {
```

```
            //...
        }
    }
?>
```

Le script suivant présente l'instanciation de la classe précédemment définie et son utilisation :

```
<?php
$fisher = new JoueurDEchec() ;
    $fisher->nom = "fisher" ;
$fisher->niveau = 2700 ;
    $fisher->joueCoup() ;
?>
```

Les classes et les objets sont eux aussi visibles uniquement dans les pages PHP. Pour partager des classes et des objets entre différentes pages, il faut utiliser le mécanisme de session.

L'exemple suivant permet de stocker l'objet `$fisher` dans la session :

```
<?php
//démarrage de la session
session_start() ;

    //création d'une variable et inscription dans la session
$_SESSION['fisher'] = $fisher ;
?>
```

Le métamodèle PHP

Afin de prendre en compte la plate-forme PHP dans MDA, nous avons choisi d'élaborer le métamodèle PHP conformément au standard MOF et aux principes introduits au chapitre 9.

Ce métamodèle définit la structure de modèles représentant des applications PHP. C'est à notre connaissance le seul métamodèle PHP existant à l'heure actuelle.

Lorsque nous avons élaboré ce métamodèle, notre objectif était double. Nous voulions, d'une part, permettre une transformation des modèles de composants UML2.0 vers des modèles PHP et, d'autre part, faciliter la génération de code PHP. Nos choix architecturaux ont donc été guidés par ces objectifs.

Concernant la transformation des modèles de composants UML2.0 vers des modèles PHP, nous voulions illustrer la possibilité qu'offre MDA de prendre en compte des plates-formes non-objet. Le métamodèle que nous avons élaboré permet de modéliser les applications PHP, qu'elles utilisent ou non les objets. Nous avons aussi pris en compte le fait que les applications PHP étaient très souvent couplées à des bases de données relationnelles. Le métamodèle PHP contient donc une partie relative aux bases de données.

Concernant la génération de code, nous voulions uniquement créer les squelettes des classes et des fonctions PHP. Notre objectif était de générer une partie du code et non l'intégralité des scripts. C'est pourquoi le métamodèle que nous avons élaboré se focalise sur les aspects statiques et architecturaux de PHP et ne prend pas en considération les aspects dynamiques du langage (appels de fonction, boucles de contrôle, etc.).

Pour finir, soulignons que notre objectif est avant tout pédagogique. Ce métamodèle est un démonstrateur illustrant la faisabilité de l'approche. L'ambition n'est nullement de construire un métamodèle permettant de modéliser toutes les applications PHP ni de construire une transformation UML2.0 vers PHP prenant en compte toutes les variations possibles dans les modèles UML.

Structuration du métamodèle PHP

Nous avons vu précédemment qu'une page PHP était une page HTML contenant plusieurs blocs de scripts PHP. Pour représenter cela dans notre métamodèle, nous avons choisi de construire les métaclasses `Page`, `Bloc` et `Instruction`. La métaclasse `Page` représente une page PHP, et la métaclasse `Bloc` un bloc d'instructions PHP. Une instruction PHP est représentée par la métaclasse `Instruction`.

Pour représenter les relations de composition entre ces concepts, nous avons choisi de construire des méta-associations d'agrégation. Ainsi, la métaclasse `Page` est reliée par une méta-association d'agrégation à la métaclasse `Bloc`, et la métaclasse `Bloc` est reliée par une méta-association d'agrégation à la métaclasse `Instruction`.

La figure 11.1 illustre cette partie du métamodèle PHP.

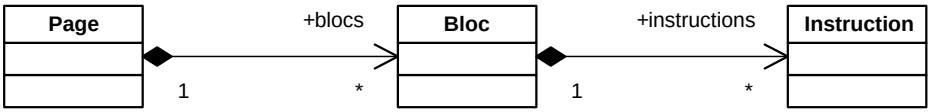


Figure 11.1
Métaclasses représentant la page PHP et les blocs PHP

Nous avons délibérément choisi de ne pas représenter dans ce métamodèle les parties HTML des pages PHP. Ce choix a été guidé par le fait que nous voulons générer uniquement les squelettes des codes PHP et non le code HTML.

Fonctions, variables et constantes

Nous avons déjà vu que PHP permettait de définir des fonctions, des variables et des constantes. Pour faire apparaître cela dans notre métamodèle, nous avons choisi de construire la métaclasse `Définition`, qui représente le concept général de définition. Cette métaclasse contient un méta-attribut nommé `nom`, qui représente le nom de la définition.

Pour bien identifier les définitions présentes dans un bloc PHP, nous avons relié la métaclasse `Bloc` à la métaclasse `Définition` par une méta-association d'agrégation. Cela signifie qu'un bloc peut contenir directement une ou plusieurs définitions.

Nous avons construit les métaclasses `Fonction`, `Variable` et `Constante`, qui héritent de la métaclasse `Définition`. Ces métaclasses représentent respectivement les concepts de fonction, variable et constante. La métaclasse `Constante` contient un méta-attribut nommé `valeur`, qui représente la valeur de la constante.

La métaclasse `Fonction` est reliée à la métaclasse `Variable` par une méta-association d'agrégation. Dans cette méta-association, la métaclasse `Variable` joue le rôle de « paramètre ». Dit autrement, une fonction peut contenir certaines variables qui sont identifiées comme étant les paramètres de la fonction. La métaclasse `Fonction` est reliée à la métaclasse `Bloc` par une méta-association d'agrégation dont la multiplicité est strictement égale à 1. Cela signifie qu'une fonction contient un bloc d'instructions qui correspond au corps de la fonction.

La figure 11.2 illustre cette partie du métamodèle.

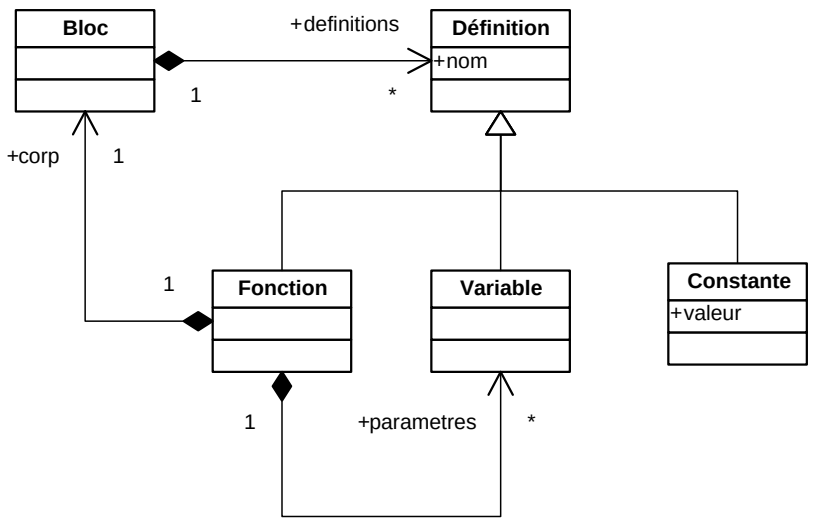


Figure 11.2
Métaclasses représentant les concepts de fonction, variable et constante

Classes, attributs et méthodes

Pour représenter les concepts de classe, d'attribut et de méthode, nous avons choisi de construire les métaclasses `Classe`, `Attribut` et `Méthode` *(voir figure 11.3)*. La métaclasse `Classe` hérite de la métaclasse `Définition`. Pour représenter le concept d'héritage entre classes, nous avons construit une méta-association ayant comme source et comme cible la métaclasse `Classe`. Les multiplicités de cette méta-association ont été définies pour faire en sorte que l'héritage multiple ne soit pas possible.

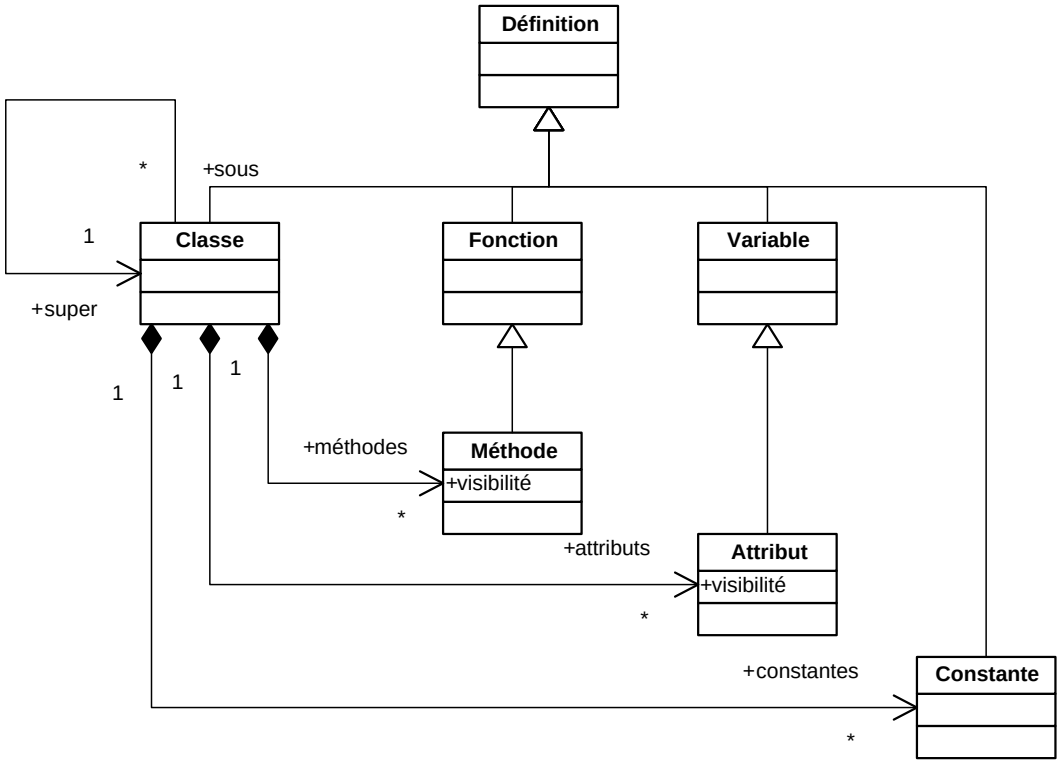


Figure 11.3

Métaclasses représentant les concepts de classe, attribut et méthode

Nous avons considéré qu'une méthode était une sorte de fonction définie à l'intérieur d'une classe. La différence entre une fonction et une méthode est que la méthode a une visibilité propre à la classe (public, private et protected).

Pour représenter cela dans le métamodèle PHP, nous avons choisi d'établir un lien d'héritage entre la métaclasse `Méthode` et la métaclasse `Fonction`. Nous avons ajouté le méta-attribut `visibilité` dans la métaclasse `Méthode`. Ce méta-attribut représente la visibilité de la méthode. Nous avons construit une méta-association d'agrégation entre la métaclasse `Classe` et la métaclasse `Méthode` afin de bien préciser qu'une méthode appartenait à une et une seule classe.

Nous avons aussi considéré qu'un attribut était une sorte de variable définie à l'intérieur d'une classe. De façon similaire à ce que nous avons fait pour la métaclasse `Méthode`, nous avons choisi d'établir un lien d'héritage entre la métaclasse `Attribut` et la métaclasse `Variable` et avons ajouté le méta-attribut `visibilité` à la métaclasse `Attribut`. Nous avons aussi construit une méta-association d'agrégation entre la métaclasse `Classe` et la méta-classe `Attribut` afin de bien préciser qu'un attribut appartenait à une et une seule classe.

Pour finir, nous avons construit un lien d'agrégation entre la métaclasse `Classe` et la métaclasse `Constante` afin de préciser qu'une constante pouvait être définie à l'intérieur d'une classe.

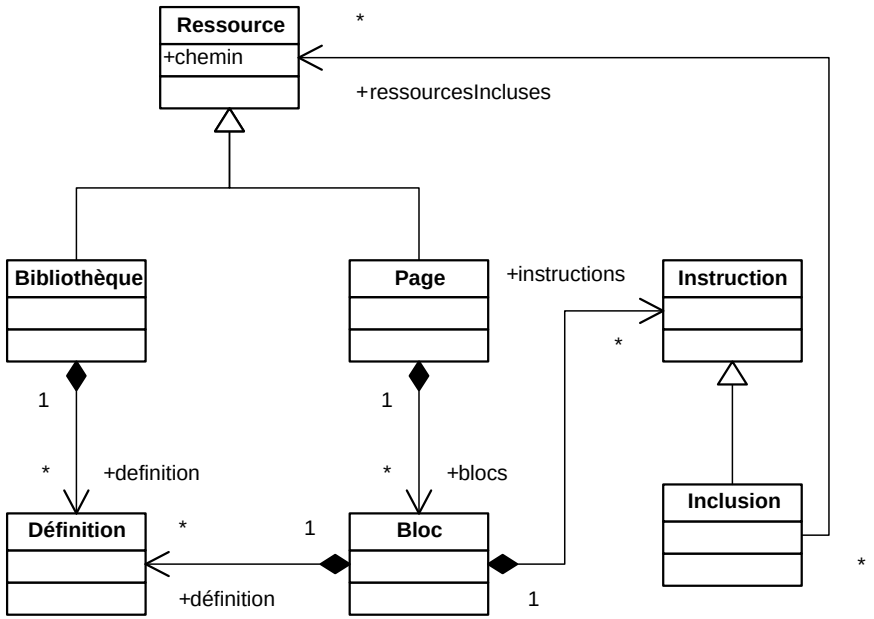
Le mécanisme d'inclusion

Pour le mécanisme d'inclusion de PHP, nous avons introduit le concept de ressource, qui représente n'importe quel fichier accessible *via* un chemin. Pour représenter ce concept dans le métamodèle PHP, nous avons construit la métaclasse `Ressource`, qui dispose d'un méta-attribut nommé `chemin` *(voir figure 11.4).*

Nous avons ensuite considéré deux sortes de ressources, les pages et les bibliothèques. Les bibliothèques ne contiennent que des définitions PHP alors que les pages peuvent contenir des définitions et des scripts (rappelons que nous ne prenons pas en compte le code HTML).

Pour représenter le concept de bibliothèque dans le métamodèle PHP, nous avons construit la métaclasse `Bibliothèque`. Cette métaclasse hérite de la métaclasse `Ressource` et est reliée par une méta-association d'agrégation à la métaclasse `Définition`. Concernant le concept de page PHP, nous avons établi un lien d'héritage entre la métaclasse `Page` et la métaclasse `Ressource` afin de préciser qu'une page était une ressource.

Figure 11.4
Métaclasses représentant le mécanisme d'inclusion



Pour représenter le mécanisme d'inclusion dans le métamodèle PHP, nous avons construit la métaclasse `Inclusion`. Celle-ci hérite de la métaclasse `Instruction` et est reliée par une

méta-association à la métaclasse `Ressource`. Cette métaclasse permet d'identifier la ressource à inclure.

Le concept de connexion

Nous avons vu que dans PHP une connexion était une instruction permettant l'accès à une base de données. Nous avons souhaité représenter le concept de connexion dans le métamodèle PHP car la plupart des applications PHP sont connectées à des bases de données. Du fait de l'intégration de ce concept dans le métamodèle, la génération de code pourra prendre en compte l'aspect base de données d'une application PHP.

Nous avons donc construit la métaclasse `Connexion`, qui hérite de la métaclasse `Instruction`. Cette métaclasse représente le concept de connexion. Elle est reliée à la métaclasse `Base`, qui représente le concept de base de données. Les multiplicités de cette méta-association précisent qu'une connexion ne peut cibler qu'une seule base de données.

La figure 11.5 illustre les métaclasses représentant le concept de connexion et les métaclasses permettant la représentation des bases de données relationnelles.

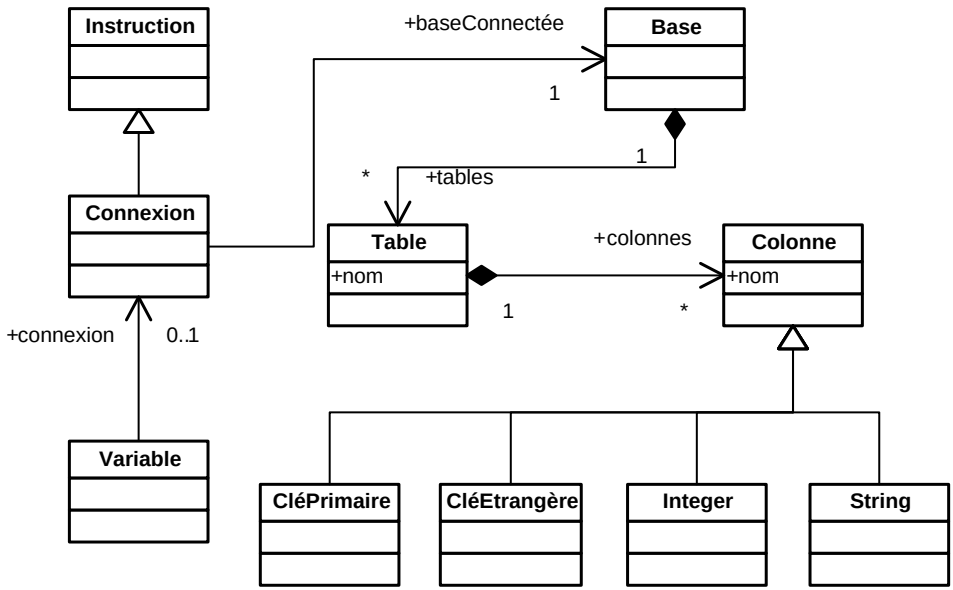


Figure 11.5

Métaclasses représentant les concepts de connexion et de base de données

La métaclasse `Base` est reliée par une méta-association d'agrégation à la métaclasse `Table`. La métaclasse `Table` représente le concept de table dans les bases de données relationnelles. Elle contient un méta-attribut `nom`, qui représente le nom de la table dans la base de données.

La métaclasse `Table` est reliée par une méta-association d'agrégation à la métaclasse `Colonne`. La métaclasse `Colonne` représente le concept de colonne dans les tables de base de données. Afin d'exprimer le fait qu'une colonne peut représenter soit une clé primaire, soit une clé étrangère, soit une chaîne de caractères, soit un nombre, nous avons choisi de construire une métaclasse par type de colonne (nous nous sommes restreint à ces types de base par souci de simplicité).

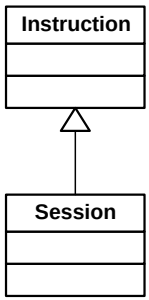
Nous avons de plus construit une méta-association entre la métaclasse `Variable` et la métaclasse `Connexion` afin de représenter le fait qu'une variable pouvait être affectée à une ouverture de connexion.

Nous avons volontairement simplifié la prise en compte des bases de données car notre objectif était de construire un métamodèle simple permettant la transformation des modèles de composants UML2.0 et la génération de code.

Le concept de session

Le concept de session se représente très facilement dans le métamodèle PHP. Il nous suffit de construire la métaclasse `Session`, qui hérite de la métaclasse `Instruction`. Cette instruction correspond au démarrage de la session *(voir figure 11.6)*.

Figure 11.6
Métaclasse représentant le concept de session



Génération du code d'une page

Le métamodèle PHP que nous venons de présenter permet de générer les squelettes de code des applications PHP. Pour ce faire, nous avons défini les règles suivantes :

- Une instance de la métaclasse `Page` permet la génération d'une page PHP (fichier **.php**). Pour chaque bloc de la page, il est possible d'écrire dans la page les balises permettant de délimiter le bloc (balises `<?php` et `?>`).
- Une instance de la métaclasse `Bibliothèque` permet la génération d'une bibliothèque PHP (fichier **.php**).
- Une instance de la métaclasse `Variable` permet la génération du code correspondant à la définition de la variable.

- Une instance de la métaclasse `Fonction` permet la génération du code correspondant à la définition de la fonction.
- Une instance de la métaclasse `Constante` permet la génération du code correspondant à la définition de la constante.
- Une instance de la métaclasse `Classe` permet la génération du code correspondant à la classe :
  - Une instance de la métaclasse `Classe` liée à des instances de la métaclasse `Méthode` permet de générer le code correspondant aux définitions des méthodes.
  - Une instance de la métaclasse `Classe` liée à des instances de la métaclasse `Attribut` permet de générer le code correspondant aux définitions des attributs.
  - Une instance de la métaclasse `Classe` liée à des instances de la métaclasse `Constante` permet de générer le code correspondant aux définitions des constantes.
- Une instance de la métaclasse `Bloc` liée à une instance de la métaclasse `Inclusion` permet la génération du code correspondant à l'inclusion.
- Une instance de la métaclasse `Bloc` liée à une instance de la classe `Session` permet la génération du code correspondant à l'ouverture de session.
- Une instance de la métaclasse `Bloc` liée à une instance de la métaclasse `Connexion` permet la génération du code correspondant à l'établissement d'une connexion.

Nous avons de plus défini une génération de code spécifique des bases de données afin de faciliter la création des applications PHP :

- Une instance de la métaclasse `Base` liée à des instances de la métaclasse `Table` permet la génération du code SQL correspondant à la création du schéma de base de données.

Exemple de mise en œuvre

Afin d'illustrer le métamodèle PHP que nous venons de définir, nous proposons de l'employer dans un exemple simple.

La notation graphique que nous utilisons consiste à représenter toute instance d'une métaclasse par un carré contenant le nom de l'instance suivi du nom de la métaclasse. Les liens entre les carrés représentent des instanciations des méta-associations.

Le modèle PHP de la figure 1.7 représente une application PHP ne contenant qu'une seule page. Cette page contient trois blocs, `B1`, `B2` et `B3`. Le bloc `B1` contient la définition d'une variable (nommée `couleur`) et la définition d'une fonction (nommée `afficherPlateau`). Le bloc `B2` contient une définition d'une classe (nommée `Joueur`). Cette classe contient une méthode nommée `joueCoup` et deux attributs, nommés respectivement `niveau` et `nom`. Le bloc `B3` contient une déclaration de session.

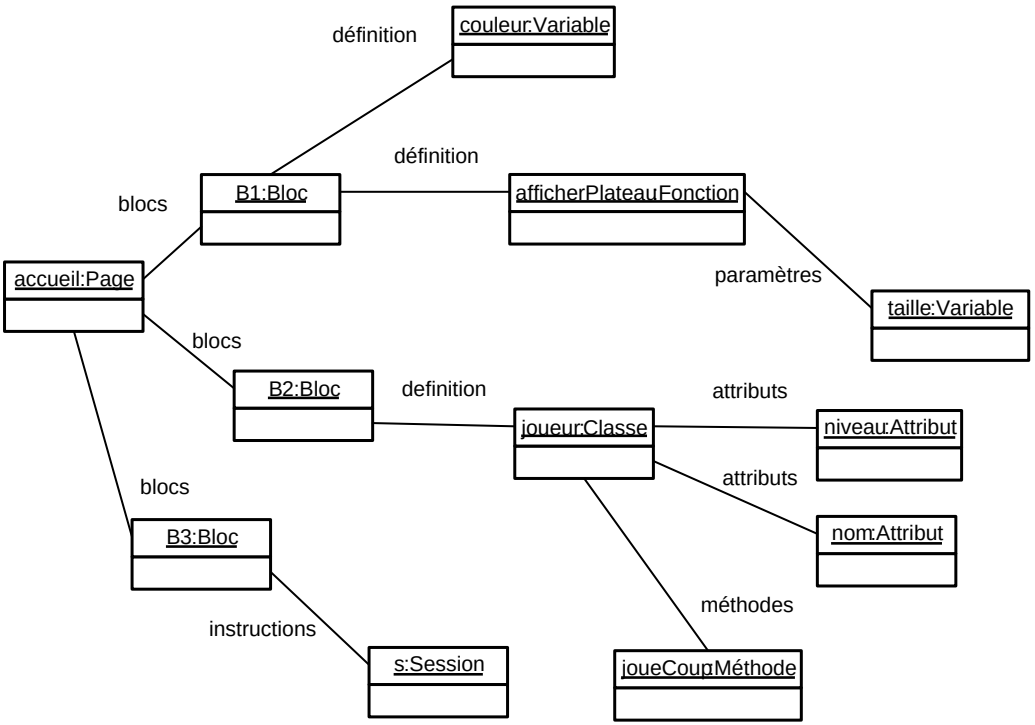


Figure 11.7
Modèle PHP d'une application simple

Grâce aux règles de génération de code que nous avons définies, ce modèle permet la génération du code suivant :

```
//Code HTML non pris en compte

<?php
$couleur ;
function afficherPlateau($taille) {
//Code à saisir
}
?>

//Code HTML non pris en compte
<?php
class Joueur{
public $nom ;
public $niveau ;

public function joueCoup() {
```

```
            //Code à saisir
}
}
?>

<?php
session_start() ;
?>
```

Transformation PIM vers PSM

La transformation des modèles de composants UML2.0 vers des modèles PHP peut paraître surprenante au premier abord tant les domaines d'application ciblés par UML et PHP semblent différents.

Nous pensons cependant qu'une telle transformation s'inscrit naturellement dans la philosophie des transformations PIM vers PSM de MDA. PHP est une plate-forme d'exécution comme les autres, qui doit pouvoir être ciblée par les transformations.

De plus, un petit tour d'horizon des outils de développement d'applications PHP du marché montre que plusieurs d'entre eux proposent d'utiliser UML pour modéliser les applications. L'utilisation d'UML pour modéliser les applications PHP est donc exploitée.

UML2.0 vers PHP

Nous avons déjà indiqué que PHP était essentiellement utilisé pour construire des applications Web. La transformation UML2.0 vers PHP que nous avons élaborée vise donc à transformer un ensemble de composants UML2.0 vers une application Web.

Une application Web réalisée avec PHP est découpée en deux grandes parties, la base de données et les traitements. Les pages PHP correspondent à l'interface graphique avec le client. Transformer une application UML2.0 vers PHP nécessite donc de considérer les composants sous cette perspective (base de données/traitements). Il faut identifier parmi les composants de l'application ceux qui représentent des données et ceux qui représentent les traitements.

Ce rôle que joue un composant est d'une importance capitale pour la transformation vers PHP. Un composant représentant des données est traduit sous forme de table dans une base de données alors qu'un composant représentant des traitements est transformé en script PHP (fonction ou classe).

La construction de pages PHP peut quant à elle être simplifiée en construisant pour chaque composant une page permettant de visualiser le composant sur une page HTML.

Les règles de la transformation

Ce sont sur ces principes de catégorisation des rôles des composants que nous avons bâti notre transformation UML2.0 vers PHP, qui implémente les règles suivantes :

Règles 1 :

- *À un composant UML2.0 qui représente des données (c'est-à-dire qui ne contient que des opérations d'accès à ses attributs) correspond une table dans une base de données. Toutes les tables correspondant aux composants d'une même application appartiennent à une même base de données.*
- *Chaque attribut du composant correspond à une colonne de la table.*
- *Si le composant est lié à un autre composant qui représente aussi des données, une clé étrangère est ajoutée dans la table pour représenter le lien entre les deux tables correspondant aux deux composants.*
- *Pour finir, une page PHP est générée pour ce composant. Cette page contient une connexion à la table correspondant au composant. Cette page contiendra les balises HTML permettant de visualiser les informations contenues dans la table.*

Règles 2 :

- *À un composant UML2.0 qui représente des traitements correspond soit une classe (1), soit un ensemble de fonctions et de variables (2). Le choix entre les deux dépend de l'utilisation faite du composant. Si le composant a un état interne et que plusieurs instances du composant puissent exister dans une même application, il faut faire le choix 1, sinon il faut faire le choix 2.*

  1. À chaque opération appartenant aux interfaces offertes par le composant correspond une méthode de la classe. À chaque attribut ou constante du composant correspond un attribut ou une constante de la classe.

  2. À chaque opération appartenant aux interfaces offertes par le composant correspond une fonction. À chaque attribut ou constante du composant correspond une variable ou une constante.

  Quelle que soit la solution choisie, le code PHP correspondant au composant appartient à une bibliothèque.
- *Si le composant est relié à un composant qui représente des données, le code PHP contient une connexion vers la table correspondant au composant.*
- *Si le composant est relié à un autre composant qui représente des traitements, le code PHP du composant contient une inclusion de la bibliothèque correspondant à l'autre composant.*
- *Si le composant a une durée de vie plus grande que la durée de vie d'une page PHP, le code PHP doit contenir une création de session.*

- *Pour finir, une page PHP est générée pour ce composant. Cette page inclut la bibliothèque du composant. Elle contiendra les balises HTML permettant d'accéder aux opérations du composant.*

Le modèle intermédiaire

La transformation que nous venons de définir contient plusieurs options. Comme nous l'avons vu au chapitre 9, les modèles intermédiaires doivent contenir les informations permettant de faire un choix parmi ces options. Une solution consiste à définir le profil UML des modèles intermédiaires et à exprimer dans ce profil, à l'aide de stéréotypes et de tagged-values, les options possibles de la transformation.

Les règles que nous avons présentées nécessitent de savoir si un composant représente des données ou des traitements. Cette information doit apparaître dans le modèle intermédiaire. Nous proposons de la définir à l'aide d'un stéréotype sur les composants. Le profil contient donc deux stéréotypes possibles, `données` et `traitement`. Un composant qui porte le stéréotype `données` est transformé selon la règle 1 tandis qu'un composant qui porte le stéréotype `traitement` est transformé selon la règle 2.

Si un composant représente des traitements, il y a deux solutions pour le transformer. Cette information doit apparaître dans le modèle intermédiaire. Nous proposons de la définir à l'aide d'une tagged-value nommée `paradigme` portée par les composants stéréotypés `traitement`. Cette tagged-value peut avoir soit la valeur `classe`, soit la valeur `fonction`. Un composant `traitement` dont la tagged-value `paradigme` vaut `classe` est donc transformé en une classe PHP.

La durée de vie des composants doit aussi apparaître dans le modèle intermédiaire. Nous proposons de la définir à l'aide d'une tagged-value nommée `vie`, portée par les composants stéréotypés `traitement`. Cette tagged-value peut avoir soit la valeur `page`, soit la valeur `session`. Un composant `traitement` dont la tagged-value `vie` vaut `session` doit donc définir une session PHP.

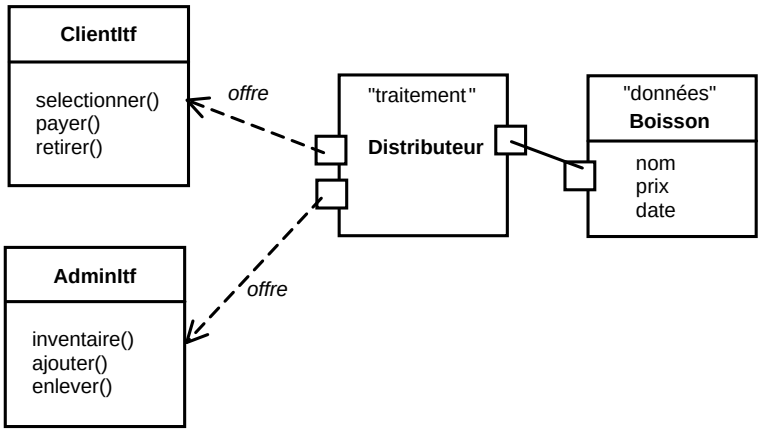
Exemple de mise en œuvre

Afin de bien faire comprendre la transformation UML vers PHP que nous venons de définir, nous proposons de l'illustrer à l'aide de l'exemple du chapitre précédent.

La figure 11.8 donne un exemple de modèle de composants UML2.0. Ce composant est un distributeur de boissons. Il offre des opérations d'utilisation (sélectionner, payer et retirer la boisson) et d'administration (faire l'inventaire, ajouter du stock, enlever du stock) *via* ses deux ports.

La figure représente en fait le modèle intermédiaire. Le composant `Distributeur` est stéréotypé `traitement` alors que le composant `Boisson` est stéréotypé `Boisson`. Cette figure ne fait pas apparaître la tagged-value `paradigme` portée par le composant `Distributeur` qui vaut `fonction`.

Figure 11.8
Exemple de modèle de composants UML2.0



Ce modèle intermédiaire permet la génération automatique du modèle présenté à la figure 11.9.

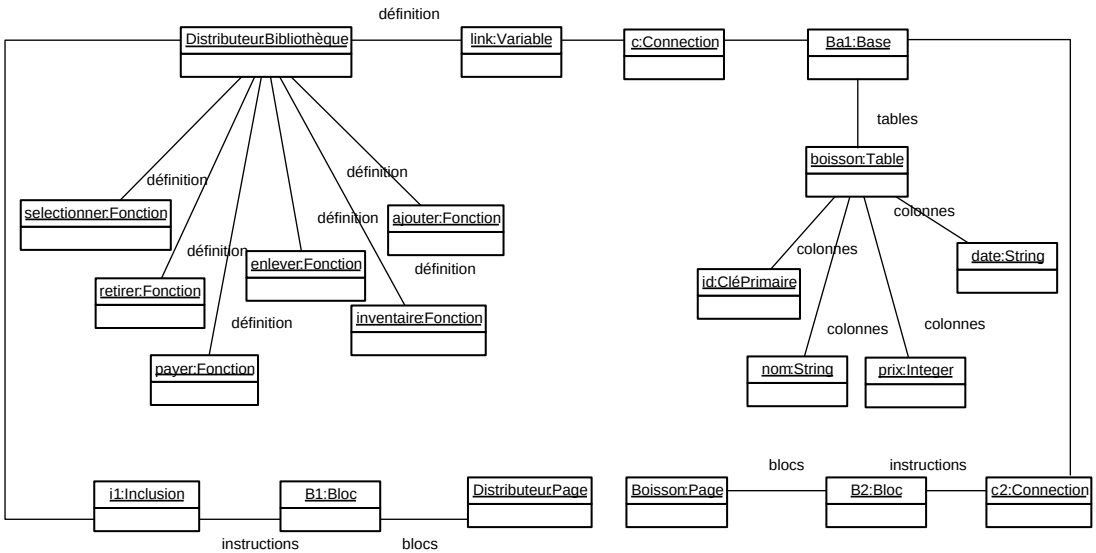


Figure 11.9
Résultat de la transformation de l'exemple vers PHP

Le modèle de la figure 11.9 contient une bibliothèque (pour le composant `Distributeur`), une table de base de données (pour le composant `Boisson`) et une page PHP par composant. Ce modèle facilite la génération des squelettes de code PHP de l'application.

Synthèse

Ce chapitre a présenté la prise en compte de la plate-forme PHP par MDA. L'approche que nous avons suivie a consisté à construire un métamodèle MOF. Ce métamodèle propriétaire représente tous les concepts statiques de la plate-forme. Les modèles instances de ce métamodèle représentent donc les parties statiques des applications PHP. Ils facilitent la génération des squelettes de code PHP, le code dynamique des applications restant à la charge du concepteur de l'application.

Nous avons présenté également la transformation UML2.0 vers PHP. Cette transformation est une transformation PIM vers PSM. Elle permet une projection des applications modélisées avec UML vers la plate-forme PHP. Cette transformation est propriétaire et ne reflète que notre vision d'une projection UML vers PHP.

Nous utiliserons le métamodèle PHP ainsi que la transformation UML vers PHP dans l'étude de cas du chapitre 12 pour construire le PSM de l'application PetStore de Sun. Cette étude de cas nous permettra d'illustrer tous les principes de l'approche MDA.

Partie IV

Étude de cas

Cette dernière partie de l'ouvrage illustre les principes de MDA appliqués à la réalisation d'une application.

Nous avons choisi de reprendre l'application PetStore des BluePrints de Sun en raison de ses vertus pédagogiques et parce qu'il est très facile de trouver de la documentation la concernant sur le Web.

Cette étude de cas nous permettra de revisiter les principes MDA que nous avons présentés tout au long de l'ouvrage afin de bien comprendre la façon de les mettre en œuvre pour la construction d'une application.

Nous en profiterons pour préciser les avantages apportés par MDA dans sa version actuelle ainsi que ceux que nous pouvons entrapercevoir pour les années à venir.

12

MDA en action avec l'application PetStore

PetStore est une application relativement classique de commerce électronique, dans laquelle les clients peuvent acheter des produits sur le Web en les ajoutant à leur panier électronique.

Les produits vendus sur le site PetStore sont des animaux domestiques *(voir figure 12.1)*. Les clients peuvent donc ajouter à leur panier des chiens, des chats ou des animaux plus exotiques, tels que des perruches. Lorsqu'ils souhaitent régler leurs achats, ils valident leur panier. Le prix du panier est alors calculé, et le paiement de la commande peut être effectué.

L'objectif de ce chapitre est d'illustrer l'application des principes de MDA à la construction d'une telle application. Nous revisiterons pour cela les concepts que nous avons introduits tout au long des chapitres précédents.

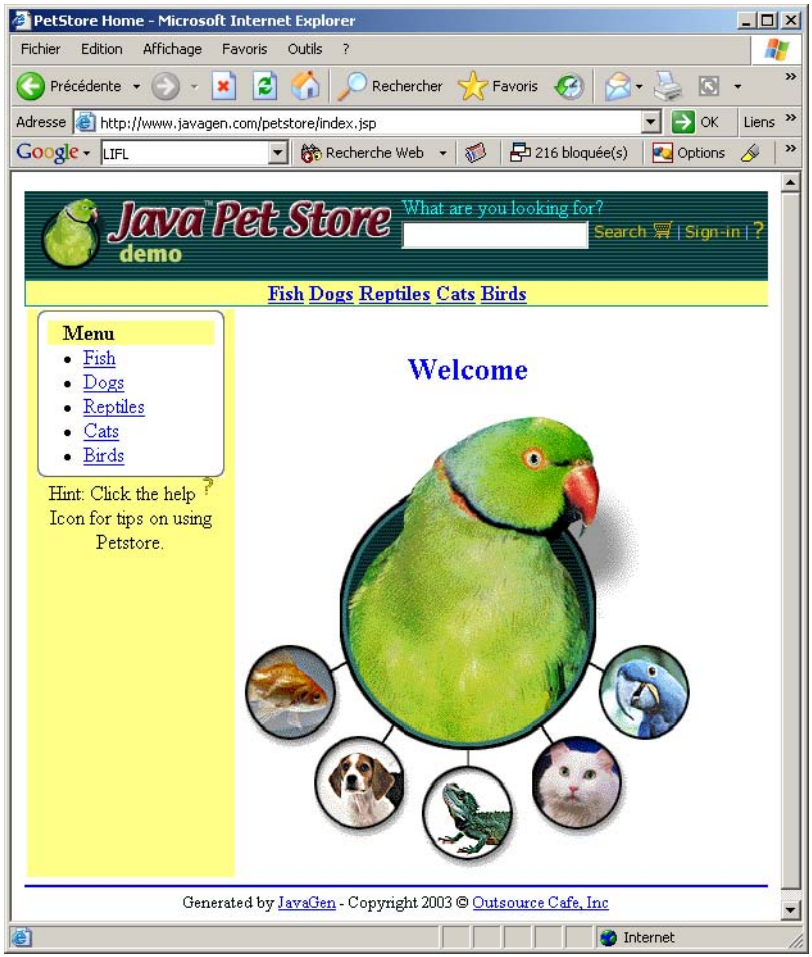


Figure 12.1
L'application PetStore

Modèle UML de l'étude de cas

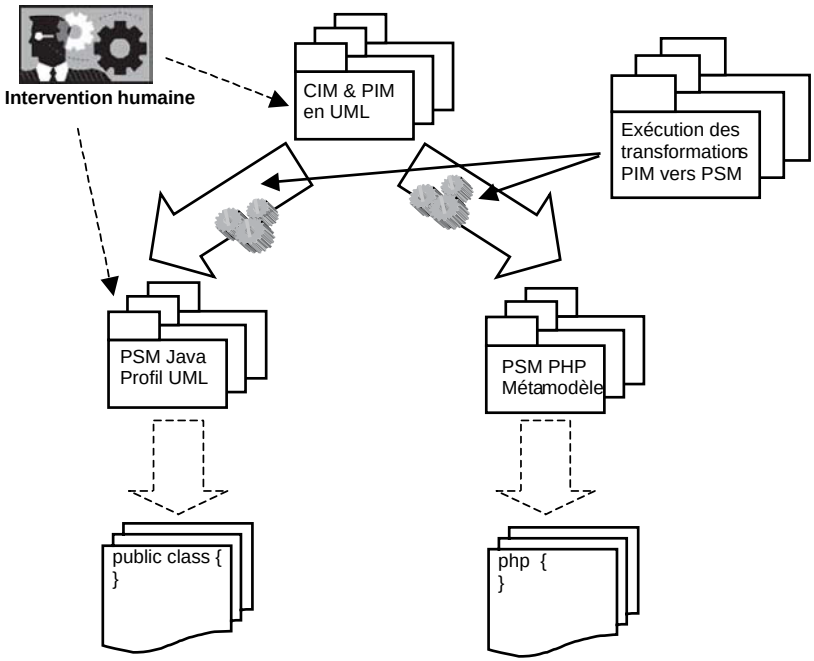
Nous avons réalisé le CIM et le PIM de PetStore dans un seul et même modèle UML2.0. Le CIM est représenté par un diagramme de cas d'utilisation tandis que le PIM est représenté par des diagrammes de composants et d'activités, des contraintes OCL et des expressions AS.

Nous avons ensuite transformé ce modèle UML en deux PSM, un pour la plate-forme J2EE et un pour la plate-forme PHP. Pour faire ces PSM, nous avons respecté les définitions des transformations données aux chapitres 10 et 11.

Nous avons partiellement réalisé ces transformations avec les outils présentés au chapitre 8, Rational Software Modeler et Softeam MDA Modeler. Nous avons choisi de réaliser la transformation vers PHP avec RSM et la transformation vers EJB avec MDA Modeler.

La figure 12.2 illustre notre mise en œuvre de MDA sur l'application PetStore.

Figure 12.2
Mise en œuvre de MDA dans l'application PetStore



Les cas d'utilisation

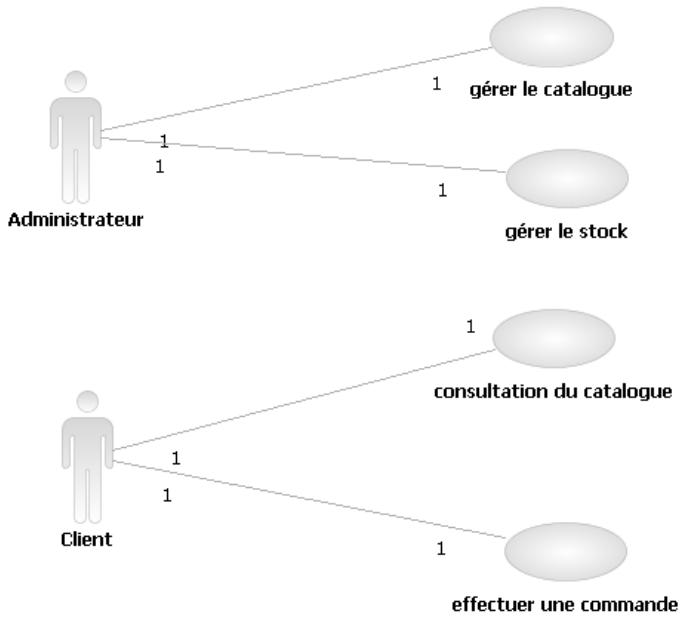
Nous n'avons élaboré qu'une partie de l'application PetStore. Le diagramme de la figure 12.3 illustre les cas d'utilisation que nous avons traités en nous focalisant sur les parties client et administrateur.

Un client peut consulter le catalogue de PetStore et obtenir ainsi toutes les références des produits vendus. Un client peut aussi effectuer ses propres commandes. Il doit alors créer un panier virtuel et y ajouter les produits qu'il souhaite acheter. Lorsqu'il le voudra, il pourra valider sa commande. Le paiement et la livraison de la commande n'entrent pas dans le cadre de notre étude.

Un administrateur peut gérer le catalogue de PetStore (ajouter de nouvelles références ou supprimer des références existantes) ainsi que le stock (acheter de nouveaux produits et faire l'inventaire).

Ce diagramme de cas d'utilisation de PetStore peut être considéré comme un modèle d'exigence, ou CIM (Computational Independent Model), comme expliqué au chapitre 1. Nous pouvons considérer qu'il contient la liste des exigences devant être satisfaites par l'application informatique. Nous n'avons pas approfondi ce modèle d'exigence afin de pas trop complexifier l'étude de cas.

Figure 12.3
Diagramme de cas d'utilisation de l'application PetStore



Le modèle de composants

À partir de ces cas d'utilisation, nous avons réalisé le modèle de composants UML de l'application *(voir le chapitre 3)*. Nous avons construit ce modèle UML à la main, sans utiliser de transformations de modèle.

Le diagramme de composants de la figure 12.4 illustre les composants suivants qui constituent l'application :

- `Produit`. Ce composant représente un produit à vendre. Un produit possède un identifiant et une référence (composant `Reference`). Ce composant offre l'interface `IProduit` *via* son unique port. Cette interface contient les opérations d'accès aux propriétés du composant (`getIdentifiant`, `getReference`).
- `Reference`. Ce composant représente une référence de produit. Une référence possède un nom, une description et un prix. Ce composant offre l'interface `IReference` *via* son unique port. Cette interface contient les opérations d'accès aux propriétés du composant (`getNom`, `getDescription`, `getPrix`).
- `LigneDeCommande`. Ce composant représente une ligne de commande établie par le client. Une ligne de commande possède une référence de produit et une quantité. Ce composant offre l'interface `ILigneDeCommande` *via* son unique port. Cette interface contient les opérations d'accès aux propriétés du composant (`getQuantité`, `getReference`).
- `Panier`. Ce composant représente un panier virtuel grâce auquel un client peut passer une commande. Un panier contient plusieurs lignes de commande, lesquelles sont définies

par l'association entre les composants `Panier` et `LigneDeCommande`. Le composant `Panier` offre l'interface `IPanier` *via* son unique port. Cette interface contient les opérations `ajouterNouvelleLigneDeCommande` et `supprimerLigneDeCommande`, qui permettent respectivement d'ajouter et de supprimer des lignes de commande au panier. Cette interface contient aussi l'opération `calculerPrixTotal`, qui permet de calculer le prix total du panier et l'opération `commander`, qui permet de valider la commande du panier.

- `Catalogue`. Ce composant représente un catalogue de références de produits. Un catalogue contient plusieurs références, lesquelles sont définies par l'association entre les composants `Catalogue` et `Reference`. Le composant `Catalogue` offre l'opération `ICatalogue` *via* son unique port. Cette interface contient les opérations `getReferences` et `getReferencesParPrix`, qui permettent de rechercher des références dans le catalogue en fonction de certains critères (non détaillés ici). Cette interface contient aussi les opérations `ajouterNouvelleReference` et `supprimerReference`, qui permettent respectivement l'ajout et la suppression de références dans le catalogue.
- `Stock`. Ce composant représente un stock de produits. Un stock possède plusieurs produits, lesquels sont définis par l'association entre les composants `Stock` et `Produit`. Le composant

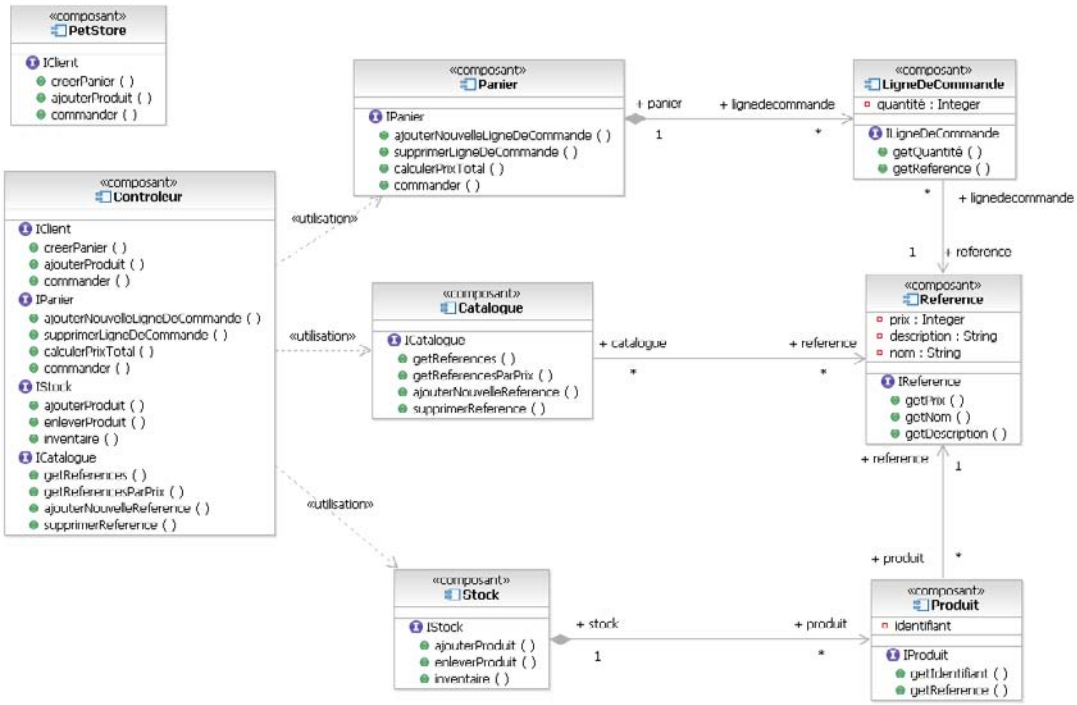


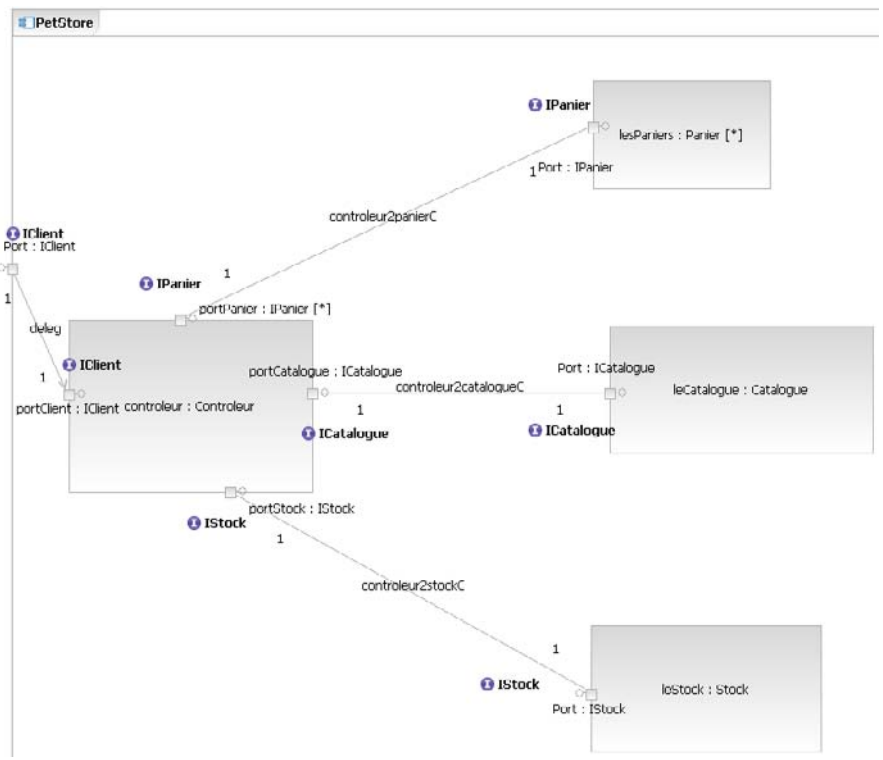
Figure 12.4
Composants de l'application

`Stock` offre l'interface `IStock` *via* son unique port. Cette interface contient les opérations `ajouterProduit` et `enleverProduit`, qui permettent respectivement d'ajouter et d'enlever des produits dans le stock. Cette interface contient aussi l'opération `inventaire`, qui permet de recenser tous les produits d'un stock.

- `Controleur`. Ce composant contient toute la logique métier de l'application. C'est lui qui reçoit les demandes des clients, construit les paniers, leur ajoute des lignes de commande et modifie le stock lors de la commande finale du panier. Ce composant requiert les interfaces `IPanier`, `ICatalogue` et `IStock`. De plus, il offre l'interface `IClient`, au travers de laquelle les clients peuvent demander l'exécution des opérations `creerPanier`, `ajouterPanier` et `commander`.
- `PetStore`. Ce composant représente l'intégralité de l'application. La définition d'un composant représentant l'intégralité de l'application est nécessaire en UML2.0 pour pouvoir définir la structuration interne de l'application.

La figure 12.5 illustre le diagramme de la structuration interne du composant `PetStore`, qui représente l'application. Nous constatons que l'application est composée d'une unique instance du composant `Controleur`, qui est reliée, grâce à des connecteurs, à l'unique instance du composant `Catalogue` et à l'unique instance du composant `Stock`.

Figure 12.5
Structuration interne de l'application



L'instance du composant `Controleur` est aussi reliée aux différentes instances du composant `Panier`. Ces connections entre composants précisent comment seront faites les interactions. En effet, seuls deux composants reliés par un connecteur peuvent interagir.

Par souci de clarté, le diagramme de la figure 12.5 n'illustre pas les connexions vers les composants `Produit`, `Reference` et `LigneDeCommande`.

Les deux diagrammes que nous venons de présenter définissent la structure statique de l'application PetStore à l'aide de composants UML. Ce modèle n'est qu'une partie du PIM de l'application, car il n'utilise pas les fonctionnalités d'une plate-forme.

Pour compléter ce PIM, il nous faut définir la dynamique des composants. Nous allons pour cela utiliser OCL, AS et les diagrammes de séquences UML.

OCL (Object Constraint Language)

Une façon de définir la dynamique des composants est d'utiliser le langage OCL *(voir le chapitre 4)*, dont nous présentons quelques exemples d'utilisation sur PetStore.

Rappelons que, même si nous utilisons ici la syntaxe textuelle d'OCL, toute expression OCL est un modèle structuré selon le métamodèle OCL.

Les accesseurs

Plusieurs opérations offertes par les composants sont des opérations d'accès en lecture aux propriétés des composants. Ces opérations sont donc sans effet de bord, et elles ne modifient pas les valeurs des propriétés. De ce fait, il est possible de les définir en utilisant OCL.

Par exemple, l'opération `getPrix` du composant `Reference` peut se définir en OCL de la façon suivante :

```
context Reference::getPrix():Integer
post: result=prix
```

Cette expression OCL définit que la valeur de retour de l'opération `getPrix` correspond à la valeur de l'attribut `prix` de la référence.

Les appels

Les composants que nous venons de définir réalisent plusieurs opérations, dont certaines doivent être appelées dans un ordre précis.

Par exemple, l'opération `commander()` du composant `Panier` ne peut être appelée que si au moins une ligne de commande a été ajoutée au panier. Cette contrainte d'ordre peut se spécifier ainsi en OCL :

```
context Panier::commander()
pre: self.lignedecommande.notEmpty()
```

Cette expression définit que l'opération `commander` ne peut être appelée si aucune ligne de commande n'est contenue dans le panier.

AS (Action Semantics)

Nous avons vu au chapitre 4 que le métamodèle AS définissait la structure des modèles représentant des suites d'instructions.

Nous utilisons AS dans PetStore pour modéliser l'activité de la méthode `calculerPrixTotal` du composant `Panier`. En utilisant une syntaxe proche du langage Java, cette activité peut se représenter de la façon suivante :

```
[1] Integer total=0 ;
[2] ForAll ligne in lignedecommande (
[3] total = total + ligne.getReference().getPrix()
[4] )
[5] return total
```

En utilisant la terminologie AS, ces quelques lignes représentent une activité (instance de la métaclasse `Activity`). Cette activité commence par définir la variable `total` (pin), dont la valeur vaut zéro, puis contient une boucle, représentée en AS par un nœud (`Node`), qui définit deux activités possibles, soit réentrer dans la boucle soit en sortir. Le corps de la boucle est une activité (`Activity`) qui contient plusieurs actions d'appel d'opérations (`CallOperationAction`) et dont le résultat est stocké dans la variable `total`. La sortie de la boucle est aussi une activité, qui précise simplement que la valeur de sortie est celle de la variable `total`.

Nous avons vu au chapitre 4 qu'il n'existait pas de syntaxe textuelle ou graphique standard pour représenter les modèles AS. Nous savons cependant que les modèles dynamiques d'UML2.0 s'appuient tous sur AS. C'est le cas notamment des diagrammes de séquences UML2.0. Ceux-ci permettent de représenter graphiquement des séquences d'activités (`Activity`) contenant des actions d'appel d'opérations (`CallOperationAction`) et de création d'instances (`CreateObjectAction`).

La figure 12.6 illustre, par exemple, une commande de produit à l'aide d'un diagramme de séquences.

Cette activité commence par une demande de création de panier par un client. Concrètement, nous voyons que le client appelle l'opération `creerPanier` sur le port client du composant contrôleur. Cet appel entraîne la création d'une instance du composant `Panier` par le contrôleur.

Ensuite, le client effectue plusieurs fois l'ajout d'un produit dans son panier. Concrètement, le client appelle l'opération `ajouterProduit` sur le port client du composant contrôleur. Le contrôleur appelle alors l'opération `ajouterLigne` du panier, ce qui permet l'ajout de la nouvelle ligne dans le panier. Par souci de simplicité nous n'avons pas fait apparaître les paramètres des opérations.

Pour finir, le client valide sa commande. Concrètement, le client appelle l'opération `commander` sur le port client du composant contrôleur. Le contrôleur calcule alors le coût total du panier puis extrait les produits commandés du stock (séquence référencée dans un autre diagramme).

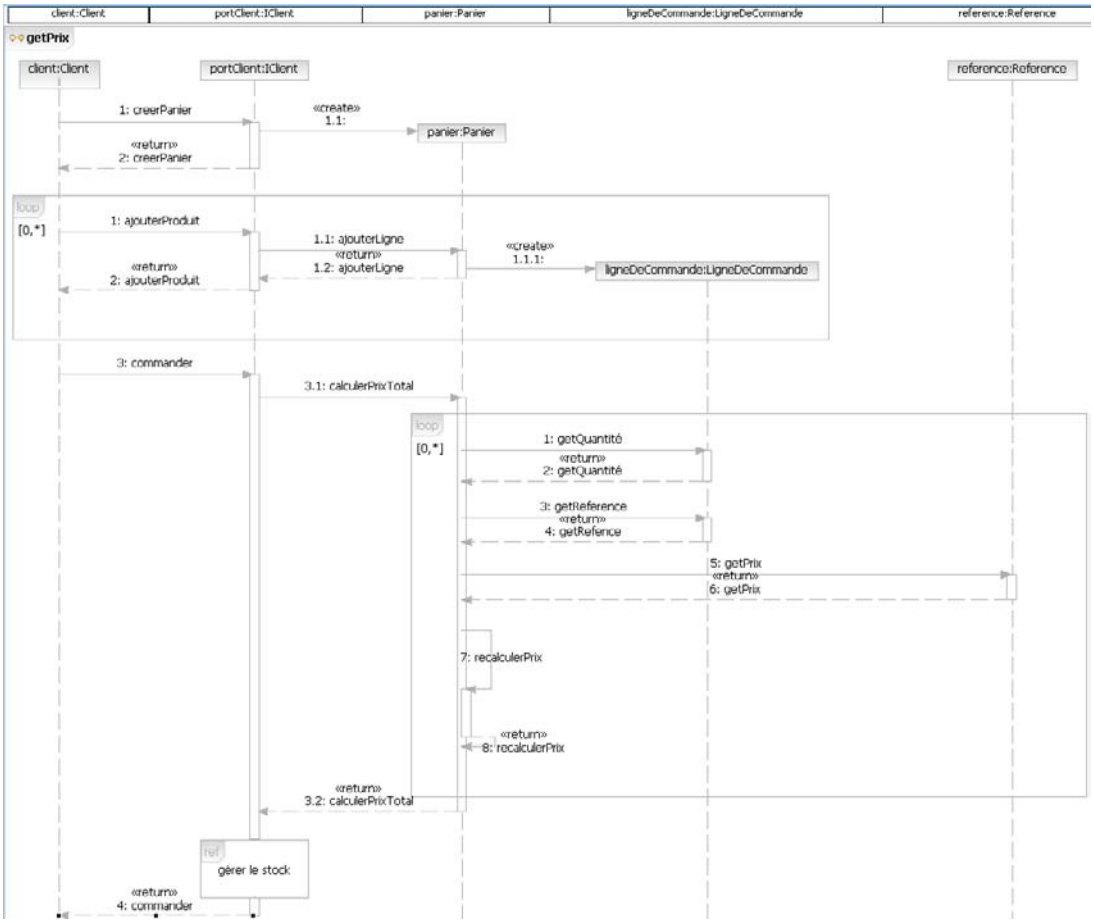


Figure 12.6

Séquence type de commande de produits

En résumé

Nous venons de présenter quelques parties du modèle UML de PetStore. Ce modèle représente à la fois le CIM et le PIM de PetStore. Le diagramme de cas d'utilisation représente le CIM tandis que les diagrammes de composants et de séquences couplés avec les contraintes OCL représentent le PIM.

Le CIM et le PIM de PetStore sont implicitement liés, car ils sont représentés dans le même modèle. De ce fait, nous pouvons considérer qu'il existe un lien de traçabilité entre les exigences du client et l'architecture de l'application.

Le PIM de PetStore est complet et précis. Toute la logique de l'application s'y retrouve spécifiée. La deuxième étape de notre mise en œuvre de MDA consiste à transformer ce PIM vers un PSM. Les sections suivantes détaillent cette transformation vers les plates-formes EJB et PHP.

Transformation UML2.0 vers EJB

Pour réaliser la première partie de la transformation UML2.0 vers EJB que nous avons présentée au chapitre 10, nous avons utilisé un module développé dans le projet de R&D RNTL *(http://www.telecom.gouv.fr/rntl/)* Accord auquel nous avons participé *(http://www.infres.enst.fr/projets/accord/).* Ce module permet de transformer les modèles de composants UML2.0 en modèles UML1.3. Nous n'avons eu à adapter que légèrement ce module car il supportait déjà toutes les solutions de transformation définies au chapitre 10.

Nous avons utilisé MDA Modeler *(voir le chapitre 8)* pour réaliser l'autre partie de la transformation. Grâce à cet outil, nous avons développé le profil du modèle intermédiaire. Rappelons que ce profil contient les stéréotypes et tagged-values `composant`, `port`, `operationEJB` et `attributEJB` nécessaires à la génération des modèles UML profilés selon le profil UML pour EJB. Nous n'avons pas eu à élaborer le profil UML pour EJB car celui-ci est proposé par MDA Modeler dans un module dédié au développement d'EJB.

Nous avons réalisé la transformation UML1.3 vers EJB en nous fondant sur le profil intermédiaire et sur le profil UML pour EJB. Nous avons réalisé cette transformation en utilisant le mécanisme de transformation de modèles par programmation proposé par MDA Modeler. Cette transformation respecte les règles que nous avons définies au chapitre 10. Elle parcourt le modèle UML source et cherche les éléments du modèle profilé selon le profil du modèle intermédiaire. Pour chaque élément identifié du modèle source, la transformation construit les éléments correspondants dans le modèle cible. Ainsi, lorsqu'une classe du modèle source est stéréotypée `composant`, un EJB est construit dans le modèle cible, c'est-à-dire une classe stéréotypée `EJBImplementation`, une classe stéréotypée `EJBHome Interface` et une classe stéréotypée `EJBRemoteInterface`.

Nous n'avons pas eu à construire de générateur de code car celui-ci est proposé par MDA Modeler dans son module EJB. Ce générateur de code permet de construire les codes Java des EJB modélisés avec le profil UML pour EJB. Nous n'avons eu qu'à l'utiliser pour générer les squelettes de code de l'application PetStore.

Exécution de la transformation

Pour exécuter la transformation UML2.0 vers EJB sur PetStore, nous avons commencé par transformer le modèle UML2.0 en un modèle UML1.3, puis nous avons transformé le modèle UML1.3 en un modèle EJB après avoir ajouté les stéréotypes et tagged-values nécessaires à la transformation.

La figure 12.7 illustre le résultat de la transformation UML2.0 vers UML1.3 exécutée sur le modèle PetStore.

Concernant l'alternative `portAlternative` de la transformation, nous avons choisi l'approche dite de « fusion » *(voir le chapitre 10)*. Les ports de chacun des composants sont ainsi fusionnés et intégrés dans la classe représentant le composant. L'autre solution aurait consisté à créer une classe pour chacun des ports des composants. Nous avons fait le choix de la fusion très naturellement car tous les composants de l'application PetStore n'offrent qu'un seul port. Dès lors, il nous semblait naturel de fusionner le port avec le composant.

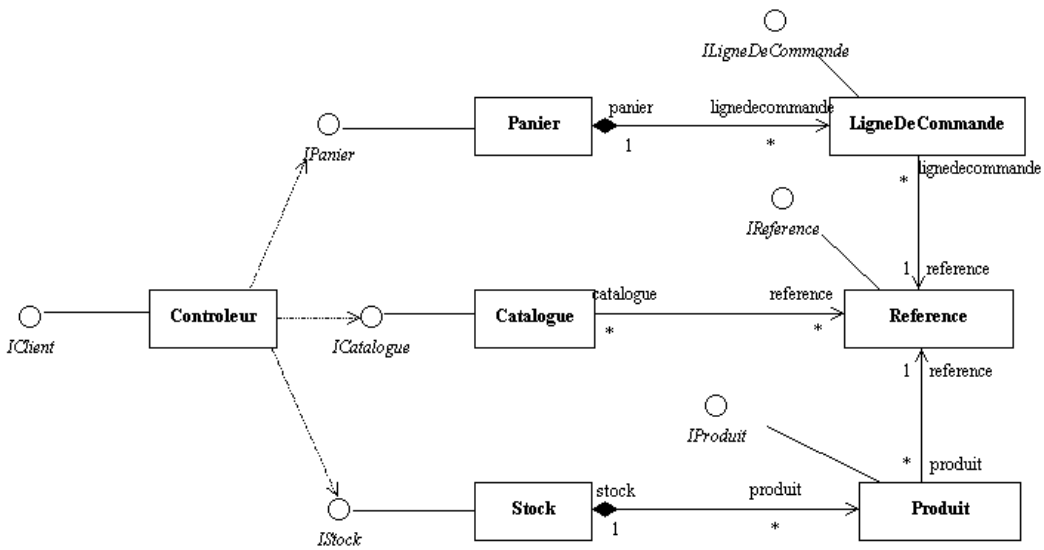


Figure 12.7
Modèle UML1.3 obtenu après transformation du modèle de composants UML2.0

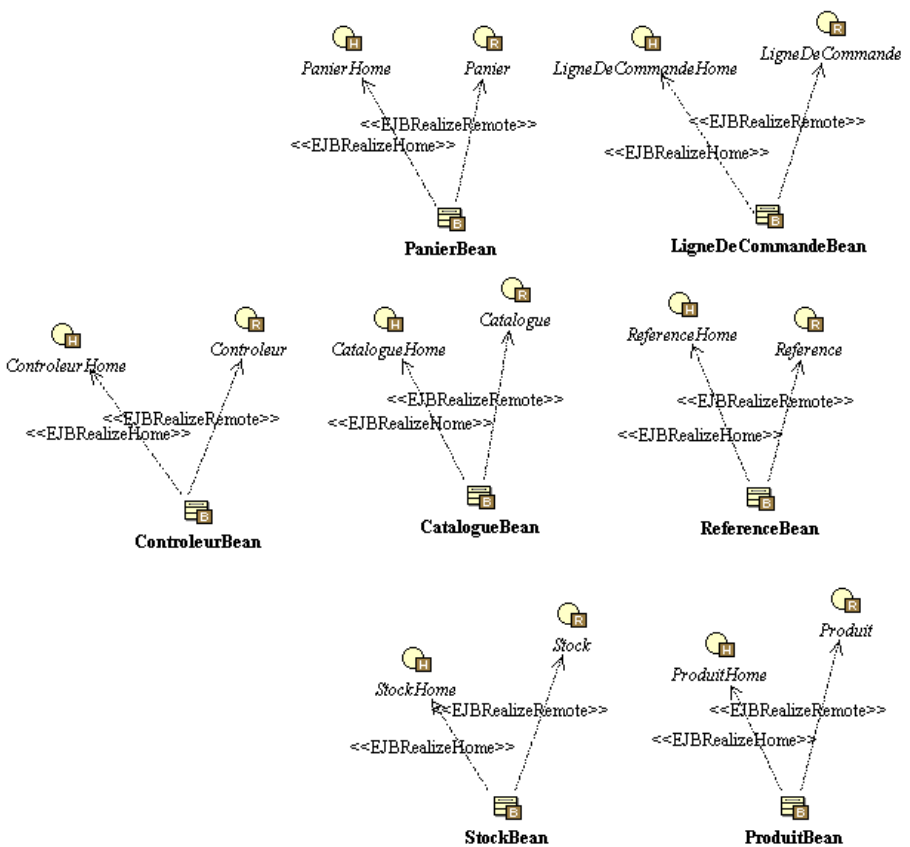
L'alternative `partAlternative` n'a pas d'incidence sur cet exemple, car seul le composant `PetStore`, qui représente l'application dans son intégralité, a des parts. Nous avons remarqué que celui-ci n'avait pas à être pris en considération lors de la transformation vers EJB puisqu'il n'existe pas de composant représentant l'intégralité de l'application en EJB.

Après avoir obtenu le modèle UML1.3, nous avons défini les stéréotypes et tagged-values permettant la génération automatique du modèle EJB. Nous avons donc ajouté le stéréotype `composant` à chacune des classes obtenues. En effet, ces classes représentent toutes des composants, et aucune d'elles ne représente de port. Nous avons ensuite ajouté les stéréotypes `operationEJB` sur toutes les opérations et `attributEJB` sur tous les attributs. Nous avons considéré que toutes les opérations et tous les attributs devaient être accessibles à distance.

L'affectation de la tagged-value `Bean` est plus délicate car elle a une incidence importante sur la génération du modèle EJB. Après mûre réflexion, nous avons considéré que seules les classes `Reference` et `Produit` représentaient les données de l'entreprise. Nous leur avons donc affecté la valeur `entity`. Pour les autres classes, nous avons considéré qu'elles représentaient des traitements de l'entreprise. Ce choix est facilement compréhensible pour les classes `Controleur`, `Catalogue` et `Stock`. Pour les classes `Panier` et `LigneDe Commande`, nous avons considéré qu'elles représentaient l'état de la commande du client. En cela, nous avons considéré qu'il valait mieux les transformer vers des Session Beans de type Stateful, sachant que la définition du caractère Stateful sera faite, après transformation, directement dans le modèle EJB.

La figure 12.8 illustre le modèle UML profilé selon le profil UML pour EJB obtenu après l'exécution de la transformation. Par souci de clarté, cette figure représente les stéréotypes sous forme d'icône. Nous constatons que toutes les classes du modèle UML ont été transformées en EJB. La nature des EJB (Session ou Entity) ne se voit malheureusement pas sur la figure car elle est masquée par les icônes qui sont les mêmes pour les Session Beans et les Entity Beans.

Figure 12.8
Modèle EJB obtenu après transformation



Après avoir généré ce modèle, nous l'avons complété en précisant, par exemple, que les Session Beans `Panier` et `LigneDeCommande` étaient Stateful. Nous avons pour cela spécifié la valeur `Stateful` dans la tagged-value `EJBSessionType`.

Après avoir complété le modèle EJB, nous avons bénéficié de la fonctionnalité de génération de code proposée par MDA Modeler pour obtenir toutes les classes Java de nos EJB. Ces classes Java ne contenant pas de code, il a fallu les compléter pour finir la réalisation de l'application.

Analyse du résultat

Grâce à cette mise en application de MDA, nous avons pu générer de façon semi-automatique les squelettes de code des EJB correspondant aux composants UML2.0 de PetStore. Ces squelettes de code contiennent toute la partie structurelle de l'application PetStore pour la plate-forme J2EE/EJB et respectent les contraintes syntaxiques de cette plate-forme.

Ce résultat est en soi remarquable puisqu'il permet de maintenir un lien fort entre le modèle UML et le code de l'application. Grâce à cette approche, les incohérences traditionnelles entre modèles et code se trouvent considérablement diminuées. Le modèle UML fait partie intégrante des éléments de production de l'application.

De plus, étant donné que le modèle UML contient un lien avec les besoins de l'utilisateur (*via* les cas d'utilisation dans notre exemple), nous pouvons considérer que le code de l'application est transitivement lié aux besoins. Grâce à ce résultat, il est possible d'envisager d'automatiser des opérations telles que l'analyse d'impact dans le code des modifications des besoins.

Il est important de préciser que le résultat que nous obtenons dépend fortement de la qualité les langages de modélisation et des transformations (ici UML vers EJB). En effet, nous avons présenté des modèles et une transformation volontairement simplifiés par souci pédagogique. Il va de soi que la mise en place de cette approche dans des projets industriels mériterait de définir des transformations beaucoup plus complètes.

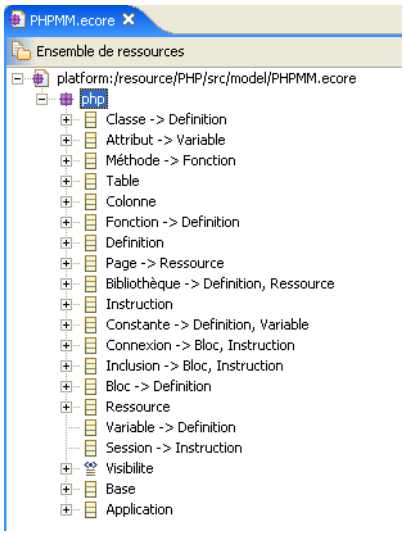
Par exemple, la prise en considération de la partie dynamique des applications, volontairement écartée ici, nécessiterait, d'une part, d'améliorer le profil UML pour EJB et, d'autre part, de compléter les transformations en conséquence. Il serait aussi possible de prendre en considération la partie déploiement de l'application en s'appuyant sur le modèle UML de déploiement présenté au chapitre 3. L'approche pourrait alors théoriquement couvrir tout le cycle de développement de l'application.

Transformation UML2.0 vers PHP

Pour réaliser la transformation UML2.0 vers PHP que nous avons présentée au chapitre 11, nous avons choisi d'utiliser l'outil RSM (Rational Software Modeler) introduit au chapitre 8. Pour ce faire, nous avons dû intégrer le métamodèle PHP dans RSM puis réaliser le profil des modèles intermédiaires et enfin la transformation UML2.0 vers PHP.

Étant donné que le métamodèle PHP n'avait jamais été défini, nous avons dû en premier lieu l'intégrer dans RSM. Pour cela, nous avons élaboré ce métamodèle sous la forme d'un métamodèle Ecore *(voir le chapitre 6)*. La figure 12.9 représente graphiquement le métamodèle PHP intégré dans l'outil RSM.

Figure 12.9
Intégration du métamodèle PHP dans RSM



Grâce à cette intégration du métamodèle PHP dans RSM, nous avons pu élaborer un générateur de code conformément à ce que nous avons défini au chapitre 11. Pour cela, nous avons utilisé le mécanisme de génération de texte de RSM *(voir le chapitre 8)* avec l'API taylored générée à partir du métamodèle PHP *(voir le chapitre 6)*.

Nous avons réalisé le profil intermédiaire en utilisant le mécanisme de création de profil de RSM. Rappelons que ce profil contient les stéréotypes et tagged-values `données`, `traitements`, `paradigme` et `vie` nécessaires à la génération des modèles PHP.

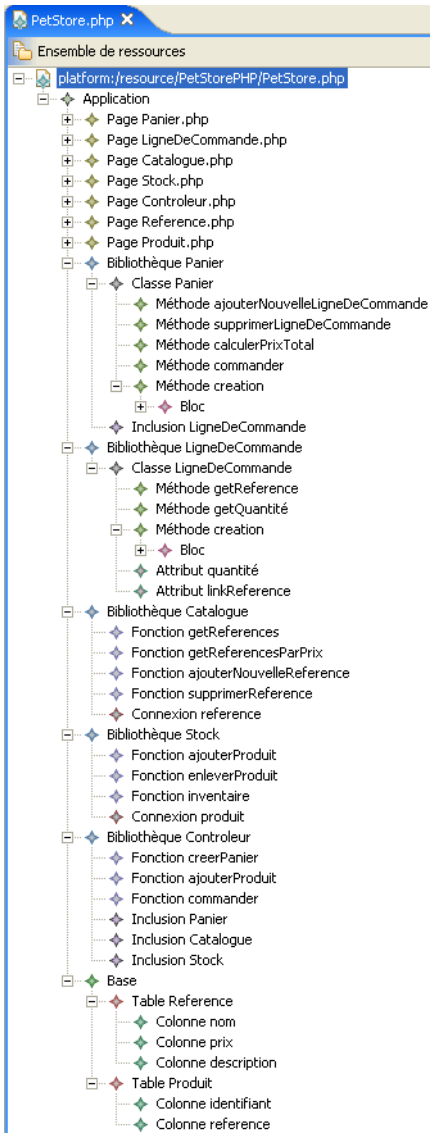
Pour la transformation UML2.0 vers PHP, nous avons utilisé le mécanisme de transformation de modèle de RSM *(voir le chapitre 8)*. Cette transformation respecte les règles définies au chapitre 11. Concrètement, la transformation parcourt le modèle UML2.0 source et recherche les éléments stéréotypés selon le profil des modèles intermédiaires. Pour chaque élément identifié du modèle source, la transformation construit les éléments correspondant dans le modèle cible. Ainsi, lorsqu'un composant du modèle source est stéréotypé `traitements` et que sa tagged-value `paradigme` vaut `classe`, une classe est créée dans le modèle cible avec les méthodes correspondant aux opérations du composant.

Exécution de la transformation

Pour exécuter la transformation UML2.0 vers PHP sur PetStore, nous avons ajouté tous les stéréotypes et tagged-values nécessaires au modèle de composants de PetStore puis appliqué la transformation sur ce modèle.

Pour l'affectation des stéréotypes et tagged-values sur le modèle de composants de PetStore, nous avons suivi la même logique que pour la transformation vers EJB. Nous avons

Figure 12.10
Modèle PHP de PetStore



considéré que seuls les composants `Produit` et `Reference` représentaient des données et que les autres composants représentaient des traitements. Nous avons donc ajouté le stéréotype `données` aux composants `Produit` et `Traitement` ainsi que le stéréotype `traitement` aux composants `Controleur`, `Panier`, `Catalogue`, `Stock` et `LigneDeCommande`. Pour cette transformation, nous n'avons pas non plus considéré le composant `PetStore`, qui représente l'intégralité de l'application et qui n'a pas de correspondance avec PHP.

Concernant les composants de traitements, nous avons considéré que seuls les composants `Panier` et `LigneDeCommande` avaient un état interne et que plusieurs instances existaient en même temps dans l'application. Nous avons donc spécifié la valeur `classe` pour leur tagged-value `paradigme`. Pour les autres composants, nous avons spécifié la valeur `fonction` pour cette tagged-value.

La figure 12.10 illustre le résultat de la transformation UML2.0 vers PHP pour PetStore. Cette représentation est générée automatiquement par l'outil RSM. Chaque nœud de l'arbre représente un élément du modèle instance d'une métaclasse du métamodèle PHP.

Nous constatons que le modèle PHP contient les bibliothèques contenant soit les définitions de fonctions soit les définitions de classes correspondant aux composants représentant des traitements. Le modèle PHP contient aussi la définition de la base de données correspondant aux composants représentant des données. Pour finir, ce modèle contient bien toutes les pages correspondant aux composants.

À partir de ce modèle, nous avons appliqué la génération de code PHP pour obtenir les squelettes de code de l'application PetStore. Nous avons complété ces squelettes par les codes dynamiques afin de finir la réalisation de l'application PetStore dans PHP.

Analyse du résultat

Tout comme pour EJB, grâce à cette mise en application de MDA, nous avons pu générer de façon semi-automatique les squelettes de code PHP correspondant aux composants UML2.0 de PetStore. Ces squelettes de code contiennent toute la partie structurelle de l'application PetStore et sont liés à la plate-forme PHP (ils respectent les contraintes syntaxiques de la plate-forme).

Ce résultat est tout aussi remarquable, sinon plus, que celui obtenu avec EJB. Il montre que les principes de MDA valent aussi sur des plates-formes non-objet. Le code PHP est fortement lié au modèle UML, et l'écart entre modèle et code est quasiment inexistant ou, du moins, se retrouve spécifié dans la transformation UML vers PHP.

Ce résultat montre aussi qu'un support outillé de l'approche MDA est envisageable même pour des plates-formes qui n'ont jamais été traitées. Dans cette étude de cas, nous avons outillé toutes les étapes de l'approche MDA. Nous avons élaboré nous-même le métamodèle PHP ainsi que la transformation UML vers PHP et la génération de code.

Le travail que nous avons effectué n'est évidemment pas complet. Notre objectif est avant tout pédagogique et non pas industriel. Néanmoins, cette étude de cas montre à quel point la mise en production de l'approche MDA est réaliste.

Soulignons pour finir que, tout comme pour EJB, le résultat final obtenu est fonction de la qualité des langages de modélisation et des transformations de modèles. Pour cette étude de cas, nous nous sommes restreint aux parties statiques des applications par souci de simplicité. Pour obtenir de meilleurs résultats il aurait fallu, par exemple, améliorer le métamodèle de PHP ainsi que la transformation pour qu'ils supportent les parties dynamiques. L'approche aurait alors permis de générer une grande partie du code PHP.

Conclusion

Cette étude de cas nous a permis de revisiter tous les sujets abordés dans l'ouvrage concernant l'approche MDA. Cela nous permet de prendre le recul nécessaire pour envisager de tirer des conclusions sur MDA.

Nous présentons ces conclusions selon les trois axes que nous avons suivis tout au long de l'ouvrage et qui sont la pérennité et la productivité des modèles et leur façon de prendre en compte les plates-formes d'exécution.

Pérennité des savoir-faire

Le modèle de l'application PetStore montre à quel point il est possible aujourd'hui, grâce à des standards tels que UML, OCL et AS, de réaliser un modèle d'application extrêmement précis tout en étant indépendant des plates-formes d'exécution (PIM).

La caractéristique majeure de ce modèle est d'être pérenne. En effet, il contient toute la logique de l'application et le fait qu'il soit indépendant des plates-formes d'exécution fait que sa durée de vie est tout aussi grande que celle de la logique de l'application.

Le rôle de MOF et des standards tels que XMI est de faire en sorte que ce modèle soit représenté informatiquement d'une manière standard. Cette représentation standard facilite les échanges informatiques des modèles et leur stockage. De ce fait, la durée de vie des modèles est renforcée car elle ne dépend pas de celle d'un logiciel ou d'une société fournissant une technologie de modélisation.

L'étude de cas PetStore nous permet de mieux nous rendre compte que la pérennité des modèles est aussi présente au niveau des modèles spécifiques des plates-formes d'exécution (PSM). En effet, nous avons vu que les PSM contenaient des informations techniques propres à la mise en œuvre de l'application sur une plate-forme particulière. Ces informations sont importantes pour, par exemple, la maintenance des applications. Elles méritent donc, elles aussi, d'être pérennes.

MDA permet une représentation de ces informations sous forme de modèles grâce notamment aux métamodèles de PSM ou aux profils et aux transformations de modèles. Cela entraîne automatiquement la pérennité de ces informations.

Gains de productivité

La pérennité des modèles n'a réellement d'intérêt que s'il est possible de faire quelque chose des modèles, autrement dit s'ils sont productifs. MDA insiste fortement sur la productivité des modèles grâce notamment à des standards et frameworks tels que MOF, JMI, EMF et QVT *(voir les chapitres 2, 6 et 7).* Cette productivité des modèles est d'ailleurs fortement relayée par les outils de manipulation de modèles que nous avons présentés *(voir le chapitre 8).*

Dans l'étude de cas PetStore, nous avons tiré pleinement parti des capacités de production des modèles. Nous avons en effet transformé les modèles UML2.0 en modèles EJB et en modèles PHP et avons généré du code à partir de ces modèles. Ces capacités de production permettent d'obtenir un retour sur investissement assez important quant à la mise en place de MDA.

Nous avons de plus montré que les outils que nous avons utilisés permettaient une mise en production industrielle des opérations de production sur les modèles. Ces outils sont parfaitement adaptés au développement d'opérations de production sur les modèles et à leur mise en production industrielle. Nous avons en effet réalisé nos propres opérations de production et les avons facilement utilisées sur nos modèles.

Nous pouvons aujourd'hui avancer sans trop de retenue que MDA est entré dans une phase d'industrialisation. Les outils du marché sont des produits industriels ayant une maturité de plus de dix ans. La productivité des modèles est donc une réalité à l'heure actuelle, et les progrès qui sont annoncés laissent entrevoir un essor considérable des activités de développement d'opérations de production sur les modèles, notamment grâce à des standards tels que MOF2.0 QVT.

Prise en compte des plates-formes

Si la pérennité et la productivité des modèles sont aujourd'hui parfaitement au rendez-vous dans MDA, il semble évident que la prise en compte des plates-formes est un domaine pour lequel il reste des progrès à faire.

Nous avons montré grâce à l'étude PetStore qu'il était possible d'automatiser partiellement la prise en compte des plates-formes pour les parties structurelles des applications. La prise en compte de leurs parties dynamiques n'est en revanche pas envisagée pour l'instant.

Même si nous pensons que la génération de 100 % du code de l'application à partir de modèles n'est pas réaliste, il nous semble intéressant de pouvoir générer un sous-ensemble de la partie dynamique. Par exemple, il nous paraît possible de générer automatiquement des requêtes d'accès aux données à partir de modèle OCL de ces accès.

Cependant, pour prendre en compte cette partie dynamique, il faut nécessairement que les métamodèles des PSM gagnent en complétude et donc en complexité. Les transformations supportant ces métamodèles seront donc très complexes, et il faudra des outils capables de gérer cette complexité.

Partie V

Annexe

Contenu du CD-ROM et procédures d'installation

L'ouvrage est livré avec un CD-ROM contenant :

- **IBM Rational Software Modeler (RSM).** Outil de modélisation UML2.0 supportant MDA, RSM permet la définition d'opérations de production sur les modèles. Cette version a une durée d'utilisation limitée à trente jours.
- **Objecteering Software MDA Modeler.** Outil de modélisation UML supportant MDA, MDA Modeler permet la définition d'opérations de production sur les modèles. Cette version n'est pas limitée dans le temps. Elle est bridée en terme de nombre d'opérations autorisées.
- **Standards OMG.** Ensemble des standards OMG présentés dans l'ouvrage. Les versions de ces standards sont les plus récentes au moment de mettre sous presse, fin février 2005.

Installation de RSM

RSM, tel que fourni sur le CD, peut s'exécuter sur l'un des systèmes d'exploitation suivants (RSM fonctionne aussi sur Linux, mais cette version n'est pas livrée sur le CD) :

- Windows XP Professional avec Service Packs 1 et 2
- Windows 2000 Professional avec Service Packs 3 et 4
- Windows 2000 Server avec Service Packs 3 et 4
- Windows 2000 Advanced Server avec Service Packs 3 et 4
- Windows Server 2003 Standard Edition
- Windows Server 2003 Enterprise Edition

La configuration minimale est la suivante :

- Processeur Intel Pentium III à 800 MHz (niveau supérieur recommandé).
- 384 Mo de mémoire vive au minimum (mémoire supérieure recommandée).

- Espace disque :
  - Pour installer la version complète de Rational Software Modeler, vous devez disposer d'un espace disque de 750 Mo. Un espace disque supplémentaire sera requis pour les ressources développées. Cet espace disque peut être réduit si des fonctions et des environnements d'exécution facultatifs ne sont pas installés.

    Vous devez disposer d'un espace disque supplémentaire si vous téléchargez l'image électronique pour installer Rational Software Modeler.

    Si vous utilisez un système de fichiers FAT32 au lieu de NTFS, un espace supplémentaire est requis.
  - Vous devez disposer d'un espace de 500 Mo dans le répertoire **TEMP.**
- Résolution d'écran de 1 024 × 768 au minimum, avec 256 couleurs (résolution supérieure High Color ou True Color recommandée).

Installation

L'installation de RSM se fait en deux étapes :

1. Décompression des fichiers d'installation. Cela se fait en exécutant le fichier **extractor .exe** présent sur le CD dans le répertoire **RSM.** Cet exécutable présente un enchaînement de fenêtres graphiques vous permettant de sélectionner le répertoire cible de l'extraction des fichiers d'installation. À l'issue de cette étape, l'installation du produit peut commencer.

Figure 1
Fenêtre d'installation de RSM

2. L'installation de RSM démarre par la présentation de la fenêtre illustrée à la figure 1. Grâce à cette fenêtre vous pouvez suivre toutes les étapes de l'installation et accéder au guide d'installation.

En suivant toutes les étapes de cette installation, vous aurez alors la possibilité d'exécuter RSM et de tester les fonctionnalités présentées au chapitre 8.

Une documentation de RSM ainsi que plusieurs documentations présentant la gamme des produits IBM Rational sont fournies dans le répertoire **RSM** du CD.

Installation de MDA Modeler

MDA Modeler, tel que fourni sur le CD, peut s'exécuter sur l'un des systèmes d'exploitation suivants) :

- Windows XP Professional avec SP1/SP2
- Windows 2000 avec SP4
- Windows 2003 Server
- Windows NT4 avec SP6

La configuration minimale est la suivante :

- Pentium IV à 1 GHz.
- 512 Mo de RAM.
- 300 Mo d'espace disque.

Limitations

MDA Modeler, tel que fourni sur le CD, a les limitations suivantes :

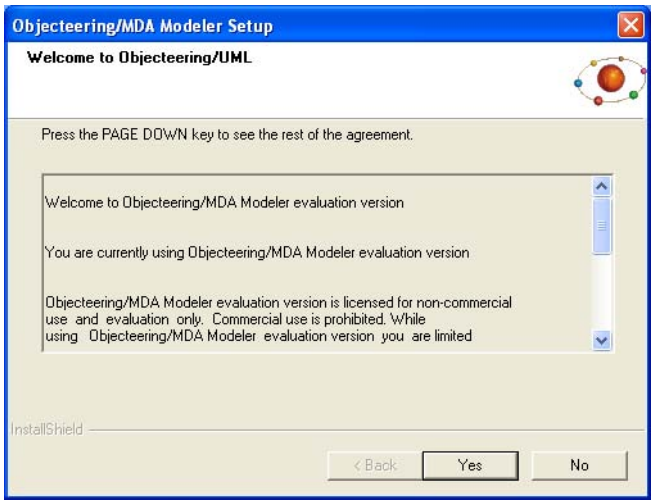
- Les opérations d'import entre les projets ne sont pas autorisées.
- La compilation des composants MDA est limitée à 150 éléments.
- La création de patterns est limitée à 2 éléments.
- La taille d'un template est limitée à 10 Ko.
- La création d'un template de document est limitée à un élément.
- La création d'un template de code est limitée à un élément.
- La création d'un template de diagramme est limitée à un élément.

Installation

L'installation de MDA Modeler consiste à exécuter le fichier **Setup.exe** présent sur le CD dans le répertoire **MDAModeler**.

Cet exécutable lance un enchaînement de fenêtres graphiques vous permettant de sélectionner vos propres choix d'installation. La première fenêtre graphique est illustrée à la figure 2.

Figure 2
Fenêtre d'installation de MDA Modeler



En suivant toutes les étapes de cette installation, vous aurez alors la possibilité d'exécuter MDA Modeler et de tester les fonctionnalités présentées au chapitre 8.

Une documentation de MDA Modeler est fournie sur le CD dans le répertoire **MDAModeler.**

Standards OMG

Le répertoire **OMG** du CD contient les standards OMG qui constituent MDA :

- **03-06-01.pdf**. Guide MDA décrivant l'approche MDA.
- **03-09-01_DI.pdf**. Standard DI présenté au chapitre 5.
- **03-09-15_UMLInfra.pdf.** Standard UML2.0 Infrastructure présenté aux chapitres 2 et 3.
- **04-10-02_UMLSuper.pdf.** Standard UML2.0 Superstructure présenté au chapitre 3.
- **03-10-14_OCL.pdf.** Standard OCL présenté au chapitre 4.
- **03-10-04_MOF2.pdf.** Standard MOF présenté au chapitre 2.
- **03-05-02_XMI20.pdf.** Standard XMI présenté au chapitre 5.

Index