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.

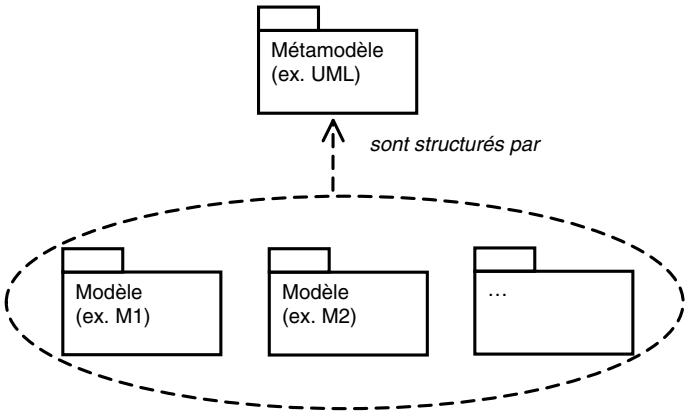
Les métamodèles

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 métamodèle. 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 :

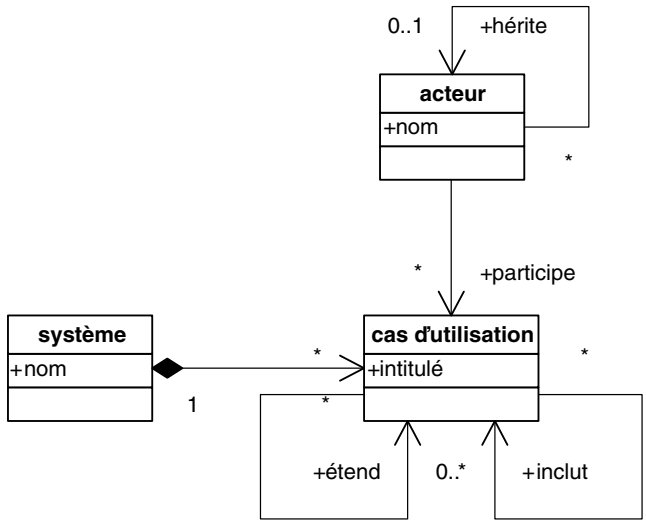
> « Un diagramme de cas d'utilisation contient des **acteurs,** un **système** et des **cas d'utilisation.** Un acteur a un nom et est *relié* aux cas d'utilisation. Un acteur peut *hériter* d'un autre acteur. Un cas d'utilisation a un intitulé et peut *étendre* ou *inclure* un autre cas d'utilisation. 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

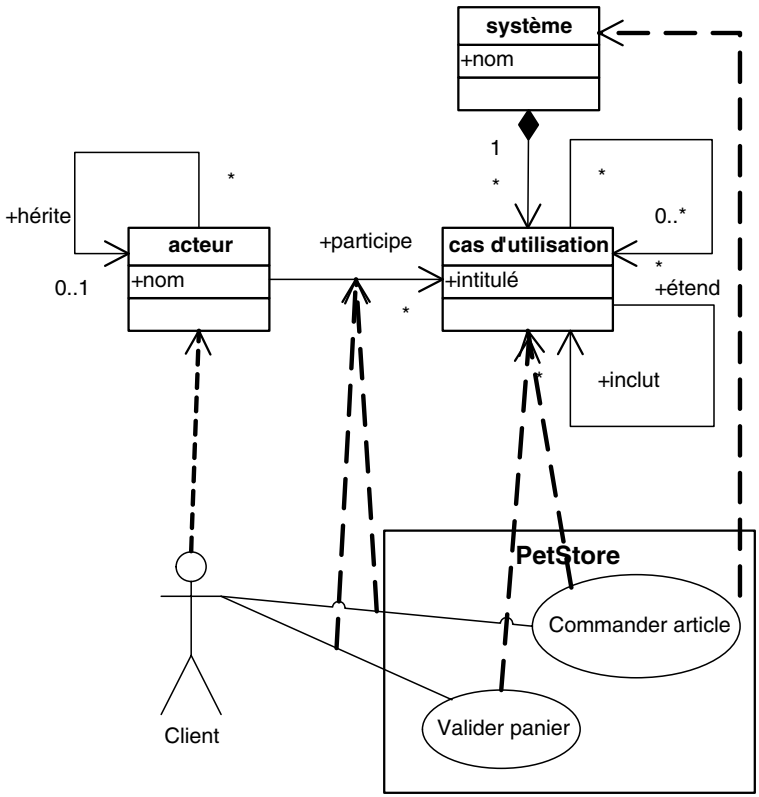


Ce métamodèle définit que les modèles en conformité avec lui ne peuvent contenir que des acteurs, des systèmes et des cas d'utilisation. Un acteur a un nom, un cas d'utilisation a un intitulé et un système a un nom. Ce métamodèle définit aussi que les modèles conformes peuvent faire apparaître des relations 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.

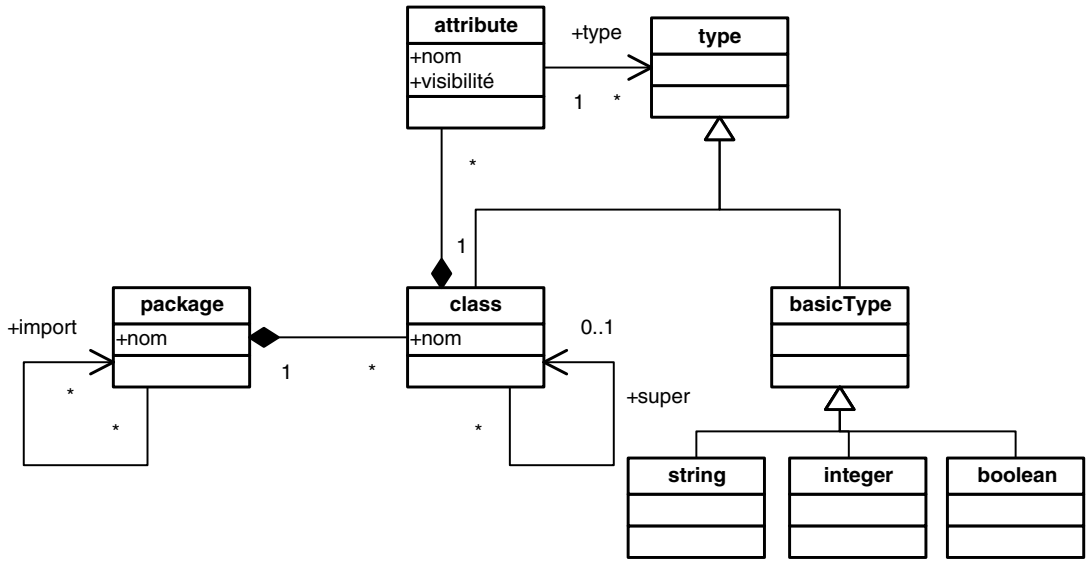


Figure 2.4
Métamodèle des diagrammes de classes

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.

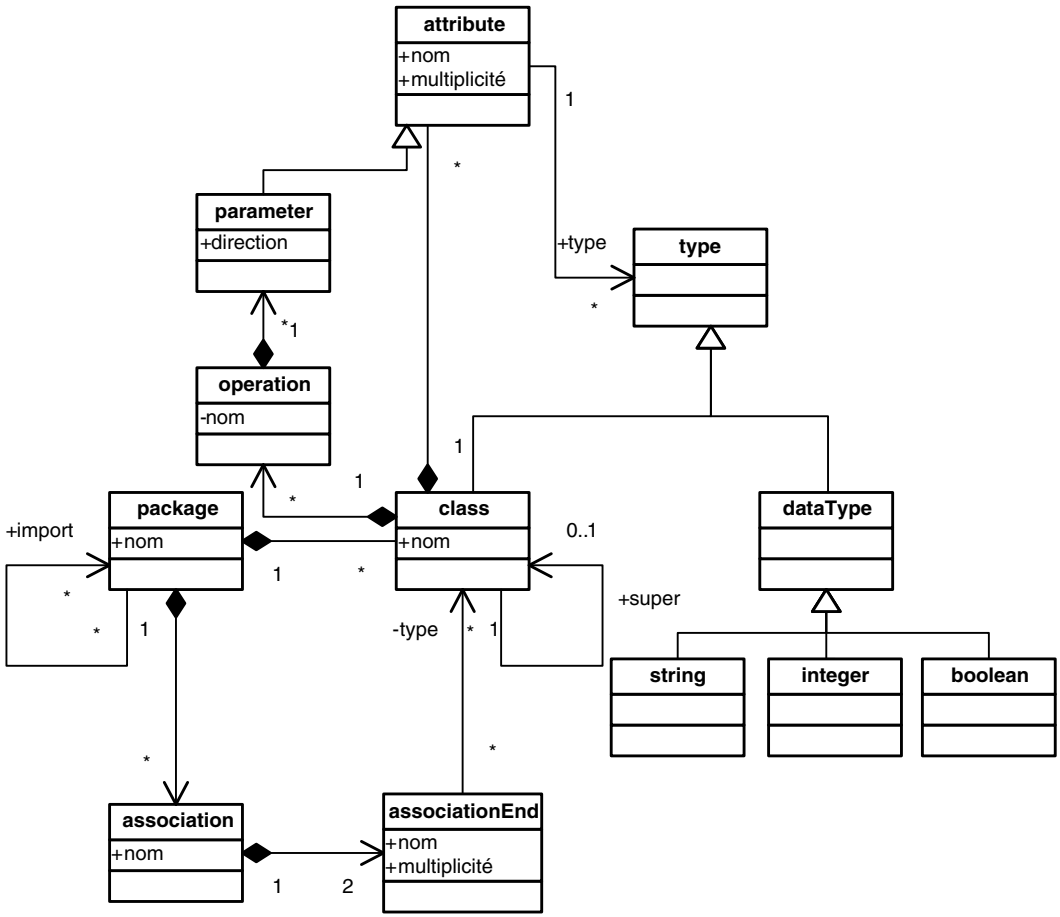


Figure 2.5

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.

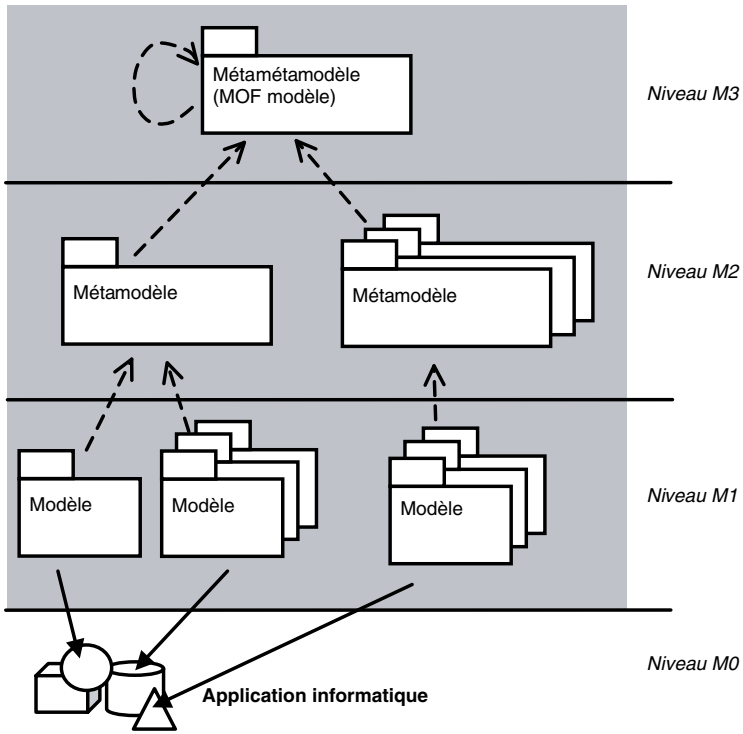
La réponse à ces questions est assez simple. Si nous devions faire le diagramme de classes décrivant la structure du métamétamodèle, nous aboutirions au diagramme de la figure 2.5. La raison à cela est que le métamétamodèle s'autodéfinit et est à lui-même son propre métamodèle. De ce fait, il n'existe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

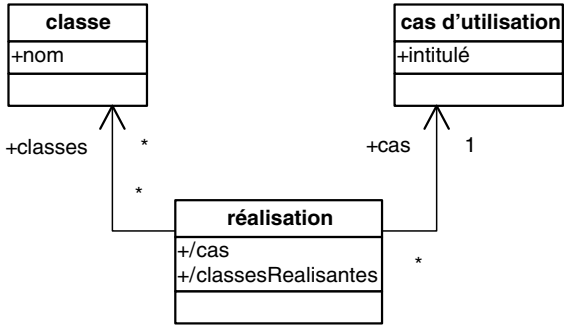
La première solution (lien interne) a l'avantage d'être très simple mais présente l'inconvénient de modifier les métaclasses existantes, en y ajoutant une association et des références si l'on veut naviguer *via* le lien. La seconde solution (lien externe) a l'avantage de ne pas modifier les métaclasses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

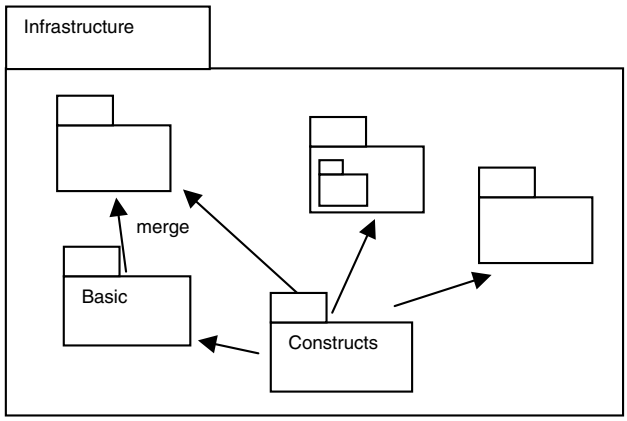
Pour ce faire, l'OMG a construit un métamodèle commun entre UML et MOF. Ce métamodèle contient toutes les métaclasses partagées par UML et MOF au niveau des diagrammes de classes. Il contient donc les métaclasses `package`, `class`, etc. Ce métamodèle commun porte le nom d'*UML2.0 Infrastructure.*

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

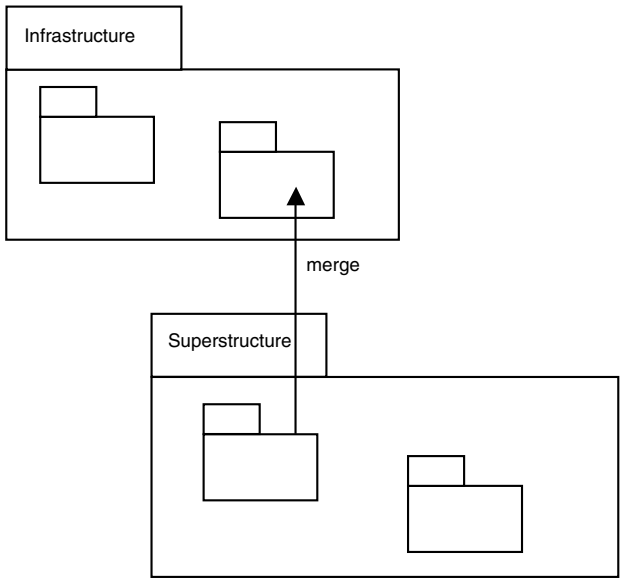
Dans sa version 2.0, le standard UML qui permet de modéliser les applications orientées objet s'appelle UML2.0 Superstructure. Ce nouveau nom permet de le distinguer du métamodèle UML2.0 Infrastructure.

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.

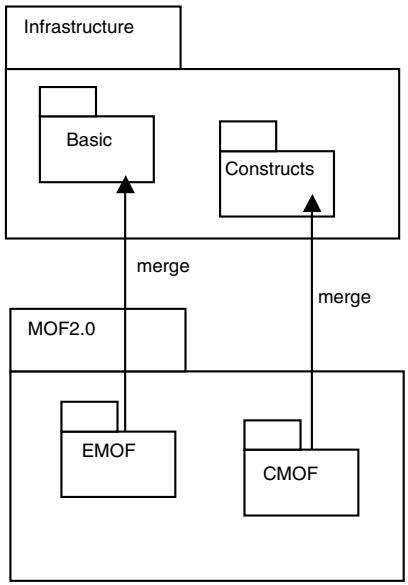
EMOF (Essential MOF)

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.