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

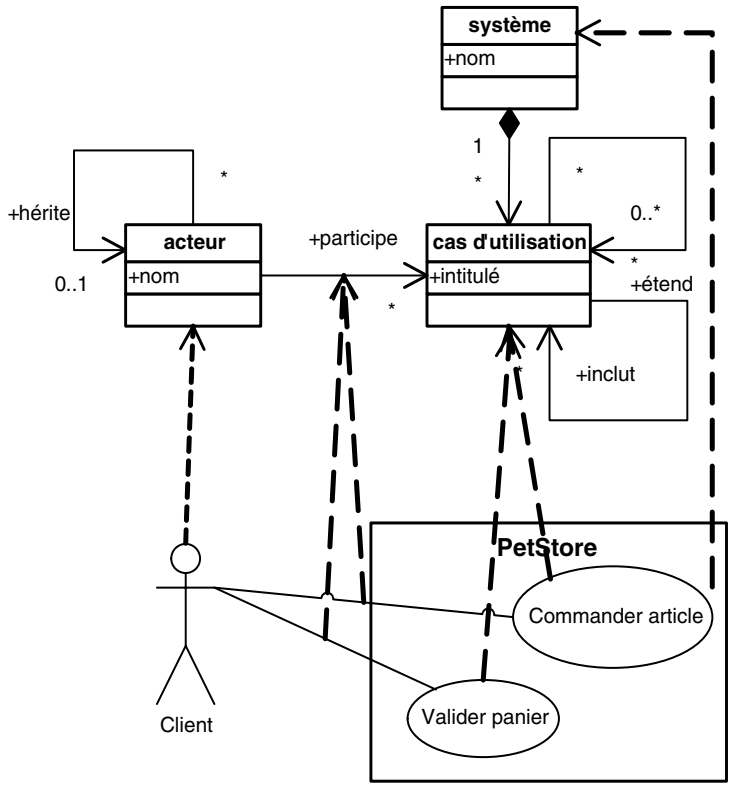
Comme expliqué précédemment, les interfaces taylored sont adaptées à un unique type de modèle. Elles offrent des opérations spécifiques permettant une navigation dans les modèles instances d'un unique métamodèle.

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.

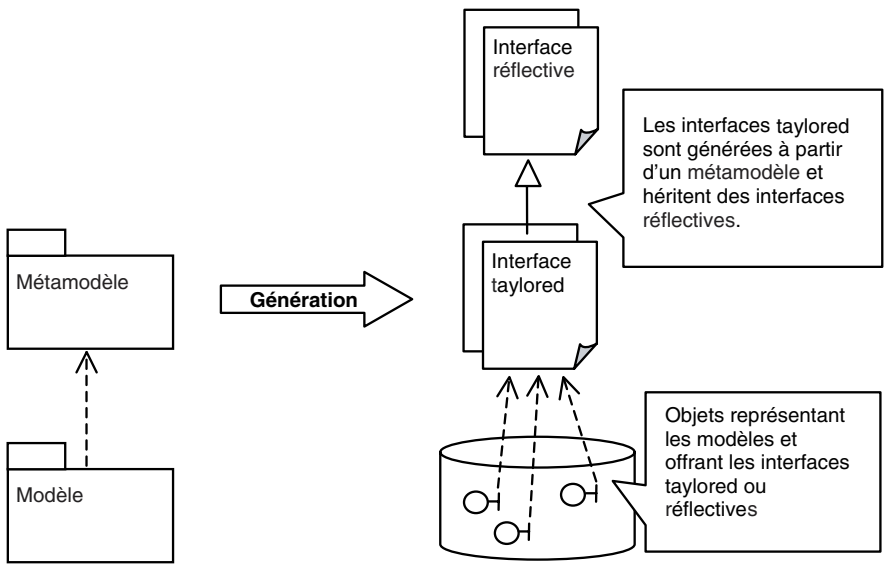
L'intérêt des interfaces réflectives est qu'elles offrent des moyens d'accéder aux métaclasses des métamodèles. À partir d'un élément d'un modèle, il est possible d'accéder à sa métaclasse et ainsi d'obtenir toutes les informations qui le structurent (attributs, références, etc.). Par exemple, sur le 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.

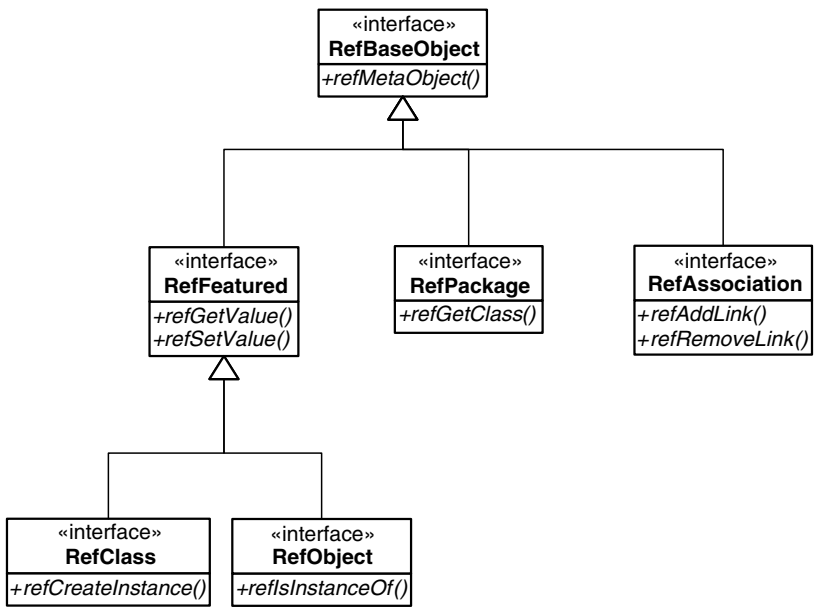
L'approche JMI suit celle que nous avons présentée à la section précédente. L'idée est de générer des interfaces taylored à partir d'un métamodèle et de faire en sorte que ces interfaces héritent des interfaces réflectives.

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.

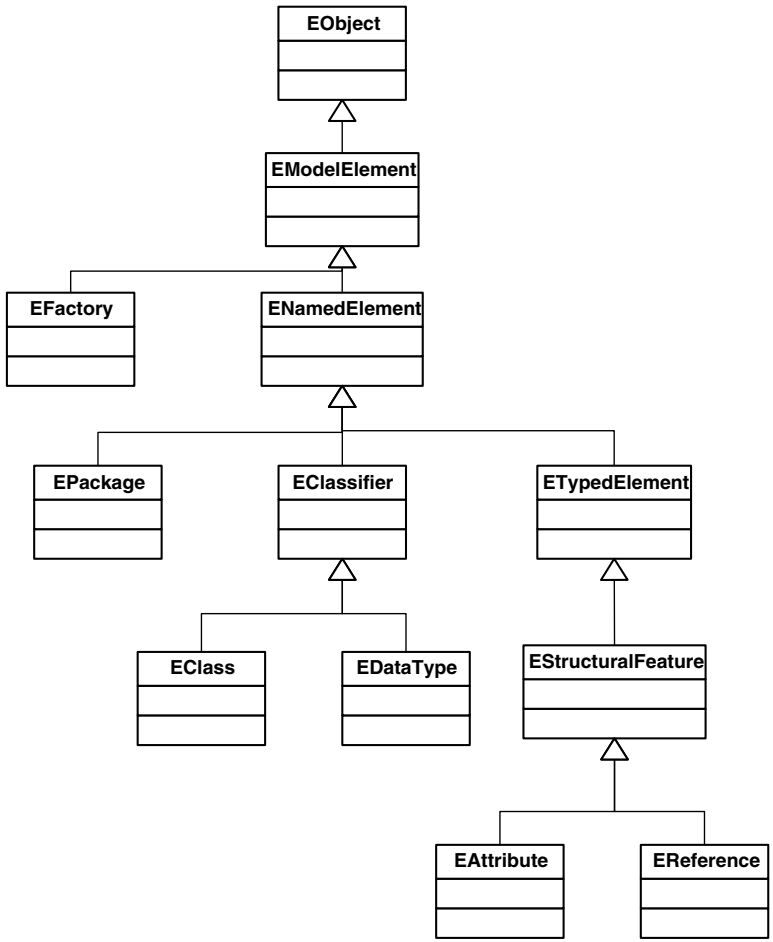
Nous commencerons par présenter le métamétamodèle d'EMF, lequel est différent du métamétamodèle MOF, puis nous présenterons les interfaces réflectives et taylored d'EMF avant de les illustrer par un même exemple. Nous finirons cette section en présentant les fonctionnalités supplémentaires offertes par 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.

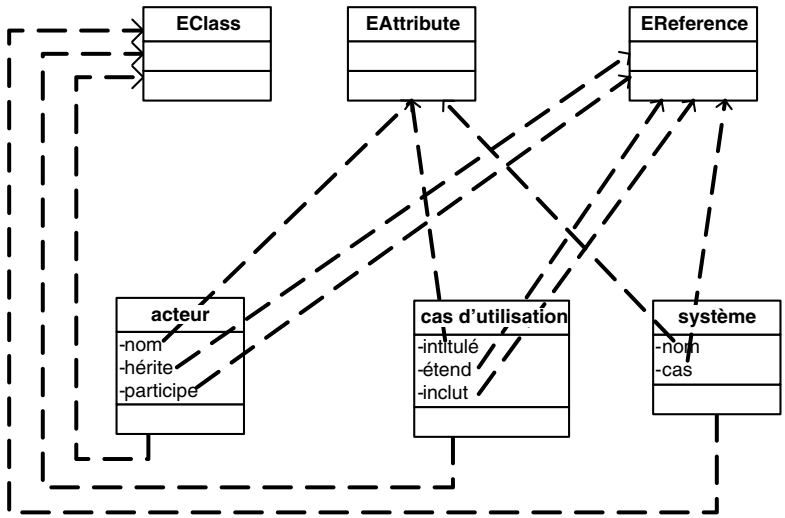
Figure 6.4
Sous-ensemble du métamétamodèle Ecore



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.

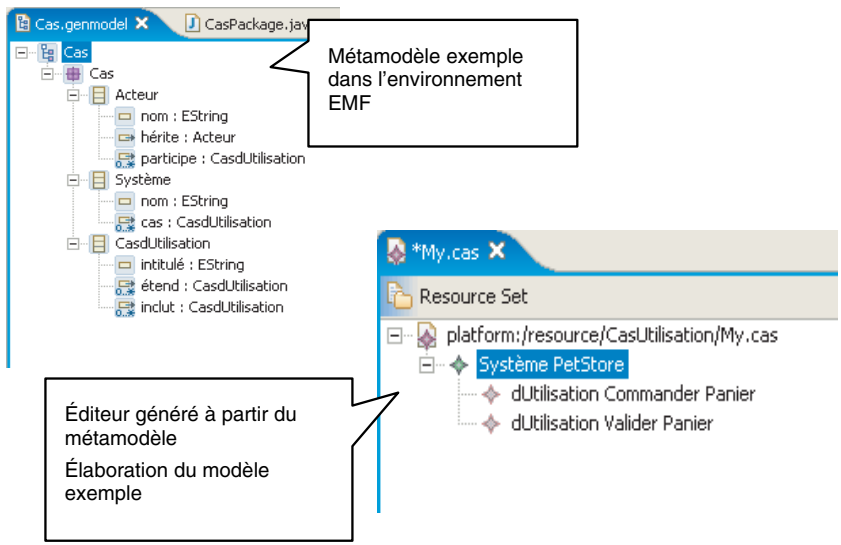
C'est dans cet objectif que plusieurs fonctionnalités ont été développées pour permettre le développement de nouveaux métamodèles et assurer la manipulation des modèles instances de ces métamodèles dans la plate-forme Eclipse. Parmi ces fonctionnalités, celle qui est certainement la plus agréable est 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.