Matière : Bases de Données Avancées — Niveau : **3ème Année Ingénieur (GLSI)**

Enseignants : **R. ZAAFRANI** — Octobre 2014

Cours n°4 (suite) : Standards et langages des bases de données à objets

Il est très important de disposer d'un standard pour un système de base de données particulier, parce qu'il assure la portabilité des applications. On définit généralement la **portabilité** comme la possibilité d'exécuter un programme donné sur différents systèmes en n'apportant que des modifications mineures au programme lui-même.

Dans le domaine des bases de données objet, la portabilité permet à un programme écrit pour accéder à un package SGBDO d'accéder à un autre package SGBDO, pourvu que le standard soit appliqué scrupuleusement. Il s'agit d'une caractéristique importante aux yeux des utilisateurs, qui hésitent généralement à investir dans une nouvelle technologie si les différents fournisseurs n'adhèrent pas à un standard commun.

Pour illustrer l'importance de la portabilité, supposons qu'un utilisateur donné ait dépensé des milliers de dinars pour créer une application qui tourne sur le produit d'un fournisseur, puis qu'il soit déçu par ce produit pour une raison quelconque - par exemple parce que les performances ne correspondent pas à ses besoins. Si l'application a été écrite en utilisant les constructions d'un langage standard, il peut la convertir afin d'utiliser le produit d'un autre fournisseur - qui adhère aux mêmes standards mais dont la performance est supérieure pour cette application - sans modifications majeures, ce qui représente une économie en temps et en moyens non négligeable.

L'adhésion à des standards présente un autre avantage: elle contribue à l'**interopérabilité**, qui désigne la capacité d'une application à accéder à plusieurs systèmes distincts. Pour les bases de données, cela signifie que la même application peut accéder à des données stockées dans un package SGBDO et aux données stockées dans un autre type de package. Il existe différents niveaux d'interopérabilité. Le SGBD peut être constitué de deux packages du même type (par exemple deux systèmes de bases de données objet) ou de deux packages de types différents (par exemple un SGBD relationnel et un SGBD objet). Un troisième avantage des standards est qu'ils permettent aux clients de comparer des produits commerciaux plus facilement, en déterminant quelles parties du standard sont prises en charge par chaque produit.

Comme nous le savons, l'une des raisons du succès des SGBD relationnels du commerce est le standard SQL. L'absence de standard pour les SGBDO pendant plusieurs années peut expliquer que certains utilisateurs potentiels aient hésité à se convertir à cette nouvelle technologie. C'est pourquoi un consortium de fournisseurs de SGBDO, l'ODMG (Object Data Management Group), a proposé un standard connu sous le nom de ODMG-93 ou ODMG 1.0. Après révision, il est devenu ODMG 2.0, que nous allons décrire, dans ce cours.

Il est constitué de plusieurs parties: le **modèle objet**, le **langage de définition objet** (ODL ou Object Definition Language), le **langage de requête objet** (OQL ou Object Query Language) et les **liaisons** (bindings) avec les langages de programmation orientés objet. Ces dernières ont été spécifiées pour plusieurs langages, notamment C++, Smalltalk et Java. Certains fournisseurs ne proposent que des liaisons de langages spécifiques sans offrir toutes les capacités d'ODL et d'OQL.

Nous décrirons le modèle objet de l'ODMG à la section 1, ODL à la section 2 et OQL à la section 3. Les exemples reprennent la base de données UNIVERSITE présentée au TD1. Nous suivrons le modèle objet ODMG2 décrit par Cattell et al[1]. Il convient de noter que nombre des idées présentes da le modèle ODMG sont fondées sur deux décennies de travaux sur la modélisation conceptuelle et les bases de données objets effectués par de nombreux chercheurs.

1. Vue d'ensemble du modèle objet de l'ODMG

Le **modèle objet** de l'ODMG est le modèle de données sur lequel s'appuie le langage de définition (ODL) et le langage de requête (OQL). En réalité, ce modèle objet fournit les types de données, les constructeurs de types et

[1] Cattell, R. (dir.), The abject Database Standard: ODMG, Re/ease 2.0, Morgan Kaufmann, 1997. 2. Les termes valeur et état sont ici interchangeables.

d'autres concepts qui servent en ODL à spécifier des schémas des bases de données objet. C'est donc un modèle de données standard pour les bases de données orientées objet, tout comme SQL décrit un modèle de données standard pour les bases de données relationnelles. Il propose également une terminologie standard dans un domaine où les mêmes termes sont parfois employés pour décrire des concepts différents.

1.1 Objets et littéraux

Les objets et les littéraux sont les blocs de base du modèle objet. La principale différence entre les deux est qu'un objet possède un identifiant d'objet et un **état** (sa valeur courante), tandis qu'un littéral n'a qu'une valeur mais pas d'*identifiant d'objet*. Dans les deux cas, la valeur peut avoir une structure complexe. L'état de l'objet peut changer dans le temps si on modifie sa valeur. Un littéral est en substance une valeur constante, peut avoir une structure complexe mais immuable.

Un objet est décrit par quatre caractéristiques:

- un identifiant;
- un nom;
- une durée de vie;
- une structure.

L'**identifiant d'objet** (ou OBJECT_ID[2]) est un identifiant unique à l'échelle du système. Tout objet doit avoir un identifiant.

En outre, les objets peuvent recevoir facultativement un **nom** unique dans une base de données particulière. Ce dernier sert à référencer un objet dans un programme. Le système doit pouvoir localiser l'objet à partir de ce nom[3]. De toute évidence, il est impossible d'attribuer à tous les objets un nom unique. En général, seuls quelques objets sont nommés, principalement ceux qui contiennent des collections d'objets particuliers, tels les extensions. Ces noms sont utilisés comme **points d'entrée** dans la base de données: en localisant ces objets à l'aide de leur nom unique, l'utilisateur peut accéder aux autres objets qu'ils référencent. D'autres objets importants de l'application peuvent également porter un nom unique. Tous les noms doivent être uniques à l'intérieur d'une base de données.

La **durée de vie** spécifie s'il s'agit d'un objet *persistant* (autrement dit un objet de la base) ou d'un objet temporaire (un objet appartenant à un programme en cours d'exécution et qui disparaît lorsque ce programme s'achève).

Enfin, la **structure** d'un objet définit la façon dont l'objet est construit au moyen de constructeurs de types. Elle spécifie s'il s'agit d'un *objet atomique* ou d'un *objet collection*[4]. L'objet atomique diffère du constructeur *atom* décrit au cours 1, et ne doit pas être confondu avec le littéral atomique. Dans le modèle de l'ODMG, un objet est atomique s'il ne constitue pas une collection. Le terme s'applique donc aux objets structurés, construits au moyen du constructeur struct[5].

Dans le modèle objet, un **littéral** est une valeur qui n'a pas d'identifiant d'objet. Toutefois, cette valeur peut avoir une structure simple ou complexe. Il existe trois types de littéraux:

- atomiques,
- collections,
- structurés.

Les **littéraux atomiques**[6] correspondent aux valeurs d'un type de données de base et sont prédéfinis. Les types de données de base du modèle objet sont les entiers longs, courts et non signés (spécifiés en ODL par les mots clés *Long*, *Short*, *Unsigned Long* et *Unsigned Short*), les décimaux ordinaires et en double précision (*Float*, *Double*), les booléens (*Boolean*), les caractères uniques (*Char*), les chaînes de caractères (*String*) et les énumérations (*Enum*).

Les **littéraux structurés** correspondent globalement aux valeurs construites au moyen du constructeur tuple décrit au cours 1. Ce sont les structures intégrées - *Date*, *Time*, *Interval*, *Timestamp* (voir le listing B) - et toutes les structures de types supplémentaires définies par l'utilisateur en fonction des besoins de chaque application. On crée une structure définie par l'utilisateur à l'aide du mot clé ODL **Struct**, comme en C et en C++.

---

[2] Correspond à l'OID du cours 1.

[3] Correspond au mécanisme de nommage décrit dans la section 3 du cours 1.

[4] Dans le modèle de l'ODMG, les objets *atomiques* ne correspondent pas aux objets dont les valeurs sont des types de données de base. Ces derniers (entiers, réels, etc) sont considérés comme des littéraux.

[5] Le constructeur struct correspond au constructeur tuple du cours 1.

[6] L'utilisation du mot atomique dans littéral atomique correspond à notre emploi du constructeur atom dans la section 2.2 du cours 1.

Les **littéraux collections** spécifient une valeur qui est une collection d'objets ou de valeurs, mais la collection elle-même n'a pas d'identifiant d'objet. Les collections du modèle objet sont *SET<T>*, *BAG<T>*, *LIST<T>* et *ARRAY<T>*, où T est le type d'objets ou de valeurs de la collection. Un autre type de collection est *DICTIONARY<K, V>*, une collection d'associations <K, V>, où chaque K est une clé (une valeur de recherche unique) associée à une valeur v. Ce mécanisme peut servir à créer un index sur une collection de valeurs.
Les listings A, B et C proposent une vue simplifiée des composants de base du modèle objet de l'ODMG. La notation ODMG emploie le mot clé interface là où nous avons utilisé les termes type et classe au cours 1. Il est plus approprié, puisqu'il désigne l'interface des **types** d'objets - nommément leurs opérations, leurs relations et leurs attributs visibles. Ces interfaces ne sont généralement pas instanciables (autrement dit, elles ne permettent de créer aucun objet), mais servent à définir des opérations qui peuvent être héritées par les objets définis par l'utilisateur pour une application particulière. Le listing A est une version simplifiée du modèle objet. Pour consulter les spécifications complètes, voir l'ouvrage de Cattell[7]. Nous étudierons les constructions représentées à mesure que nous décrirons le modèle objet.

Listing A. Vue d'ensemble des définitions d'interfaces d'une partie du modèle objet de l'ODMG. L'interface de base Object, dont tous les autres objets héritent.

```
interface Object {
…
        boolean   same_as(in Object other_object);
        Object            copy();
        void              delete();
```

Listing B. Vue d'ensemble des définitions d'interfaces d'une partie du modèle objet de l'ODMG. lnterfaces standard des littéraux structurés.

```
interface Date: Object {
        enum              Weekday
{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
        enum              Month
{January, February, March, April, May, June, July, August, September, October, November, December};
        unsigned short    year();
        unsigned short    month();
        unsigned short    day();
…
        boolean           is_equal(in Date other_Date);
        boolean           is_greater(in Date other_Date);
…
};
interface Time : Object {
…
        unsigned short    hour();
        unsigned short    minute();
        unsigned short    second();
        unsigned short    millisecond();
…
        boolean           is_equal(in Time other_Time);
        boolean           is_greater(in Time other_Time);
…
        Time              add_interval(in Interval some_Interval);
        Time              subtract_interval(in Interval some_Interval);
        Interval          subtract_time(in Time other_Time);
…
};
interface Timestamp : Object {
        unsigned short    year() ;
        unsigned short    month();
        unsigned short    day() ;
        unsigned short    hour();
        unsigned short    minute();
        unsigned short    second();
        unsigned short    millisecond();
…
        Timestamp         plus(in Interval some_Interval);
        Timestamp         minus(in Interval some_Interval);
```

[7] Cattell, R. (dir.), *The abject Database Standard: ODMG, Re/ease 2.0*, Morgan Kaufmann, 1997.

```
        boolean         is_equal(in Timestamp other_Timestamp);
        boolean         is_greater(in Timestamp other_Timestamp);
…
}
interface Interval : Object {
        unsigned short  day() ;
        unsigned short  hour();
        unsigned short  minute();
        unsigned short  second();
        unsigned short  millisecond();
…
        Interval        plus(in Interval some_Interval);
        Interval        minus(in Interval some_Interval);
        Interval        product(in long some_value);
        Interval        quotient(in long some_value);
        boolean         is_equal(in Interval other_Interval);
        boolean         is_greater(in Interval other_Interval);
…
} ;
```

Listing C. Vue d'ensemble des définitions d'interfaces d'une partie du modèle objet de l'ODMG. Définitions d'interfaces pour les objets collections.

```
interface Collection: Object {
…
        exception       ElementNotFound{any element; };
        unsigned long   cardinality() ;
        Boolean                 is_empty() ;
…
        boolean         contains_element(in any element);
        void            insert_element(in any element);
        void            remove_element(in any element) raises(ElementNotFound);
        Iterator        create_iterator(in boolean stable);
} ;
interface Iterator {
        exception       NoMoreElements();
…
        boolean         is_stable() ;
        boolean         at_end() ;
        void            reset() ;
        any             get_element() raises(NoMoreElements);
        void            next_position() raises(NoMoreElements);
…
} ;
interface Set: Collection {
        Set             create_union(in Set other_set);
…
        boolean         is_subset_of(in Set other_set);
…
} ;
interface Bag : Collection {
        unsigned long   occurrences_of(in any element);
        Bag             create_union(in Bag other_bag);
} ;
interface List : Collection {
        exception       Invalid_Index{unsigned_long index; };
        any             remove_element_at(in unsigned long position) raises(InvalidIndex);
        any             retrieve_element_at(in unsigned long position) raises(InvalidIndex);
        void            replace_element_at(in any element, in unsigned long position) raises(InvalidIndex);
        void            insert_element_after(in any element, in unsigned long position) raises(InvalidIndex);
        void            insert_element_first(in any element);
        any             remove_first_element() raises(InvalidIndex);
…
        any             retrieve_first_element() raises(Invalidlndex);
…
        List            concat(in List other_list);
        Void            append(in List other_list);
};
```

```
interface Array : Collection {
        exception       Invalid_Index{unsigned_long index; };
        any             remove_element_at(in unsigned long index) raises(InvalidIndex);
        any             retrieve_element_at(in unsigned long index) raises(InvalidIndex);
        void            replace_element_at(in unsigned long index, in any element) raises(InvalidIndex);
        void            resize(in unsigned long new_size);
} ;
struct Association {any key; any value; };
interface Dictionary : Collection {
        exception       KeyNotFound{any key; };
        void            bind(in any key, in any value);
        void            unbind(in any key) raises(KeyNotFound);
        any             lookup(in any key) raises(KeyNotFound);
        boolean         contains_key(in any key);
};
```

Dans le modèle objet, tous les objets héritent de l'interface de base *Object*, représentée dans le listing A. En conséquence, les opérations dont héritent tous les objets sont *copy* (qui crée une nouvelle copie de l'objet), *delete* (qui supprime l'objet) et *same_as* (qui compare l'identité de l'objet à celle d'un autre objet)[8]. En général, on applique les opérations aux objets en utilisant la **notation pointée**. Pour comparer par exemple un objet **o** à un autre objet **p**, on écrira : **o.same_as(p).** Le résultat retourné par cette expression est booléen, et vrai si l'identité de **p** est la même que celle de **o**, faux sinon.

De même, pour créer une copie **p** d'un objet **o**, on écrira: **p = o.copy()**

Une solution de rechange à la notation pointée est la **notation fléchée**: **o->same_as(p)**.

L'héritage de type, qui sert à définir des relations type/sous-type, est spécifié dans le modèle ODMG au moyen de la notation « : » (deux points), comme en C++. Nous constatons ainsi dans les listings que toutes les interfaces, notamment Collection, Date et Time, héritent de l'interface de base d'Object. Le modèle objet reconnaît deux grands types d'objets: les objets collections et les objets atomiques qui seront décrits dans les deux sections suivantes.

1.2 Interfaces intégrées pour les objets collections

Tout **objet collection** hérite de l'interface de base Collection (voir le listing C). Étant donné un objet l'opération **o.cardinality()** retourne le nombre d'éléments de la collection. L'Opération **o.is_empty()** retourne vrai si la collection o est vide, faux sinon. Les opérations **o.insert_element(e)** et **o.remove_element(e)** insèrent ou suppriment un élément de la collection o. Enfin, l'opération **o.contains_element(e)** retourne vrai si collection o contient l'élément e, faux sinon. L'opération i = **o.create_iterator** crée un **objet itérateur** i pour l'objet collection o, qui opère une itération sur chaque élément de la collection. L'interface pour les objets itérateurs est également représentée dans le listing C. L'opération **i.reset()** positionne l'itérateur sur premier élément de la collection (pour une collection non ordonnée, ce serait un élément arbitraire), et **i.next_position()** le positionne sur l'élément suivant. **i.get_element()** extrait l'élément courant, celui sur lequel l'itérateur est actuellement positionné.

Le modèle objet ODMG utilise des **exceptions** pour signaler les erreurs ou les conditions particulières. Par exemple, l'exception **ElementNotFound** de l'interface Collection sera levée par l'opération **o.remove_element(e)** si e n'est pas un élément de la collection o. L'exception **NoMoreElements** de l'interface **Iterator** sera levée par l'opération **i.next_position()** si l'itérateur est positionné sur le dernier élément de collection, et qu'il ne reste plus d'élément sur lequel pointer.

Les objets collections sont ensuite spécialisés en **Set**, **List**, **Bag**, **Array** et **Dictionary** qui héritent des opérations de l'interface **Collection**.

Un type d'objet **Set<t>** peut servir à créer des objets tels que la valeur de l'objet o soit un *ensemble d'éléments de type t*. L'interface **Set** comprend également l'opération **p = o.create_union(s)** (voir listing C), qui retourne un nouvel objet p de type **Set<t>** qui est l'union de deux ensembles o et s. D'autres opérations apparentées à **create_union** (non représentées dans le listing C) sont **create_intersection(s)** et **create_difference(s)**. Les opérations de comparaison des ensembles comprennent l'opération **o.is_subset_of(s)**, si l'objet o est un sous-ensemble d'un autre objet s, faux sinon. Les opérations apparentées (non représentées dans le listing C) sont **is_proper_subset_of(s)**, **is_superset_of(s)**, et **is_proper_superset_of(s)**.

Le type d'objet **Bag<t>** autorise des éléments dupliqués dans la collection et hérite également de l'interface **Collection**. Elle possède trois opérations - **create_union(b)**, **create_intersection(b)**, **create_difference (b)** -

[8] D'autres opérations, non représentées dans le listing A, sont définies pour des besoins de verrouillage.

qui retournent toutes trois un nouvel objet de type **Bag<t>**. Par exemple, **p = o.create_union(b)** retourne un objet p de type **Bag** qui est l'union de o et b (en conservant les doublons). L'opération **o.occurrences_of(e)** retourne le nombre d'occurrences dupliquées de l'élément e dans la collection o.

Un type d'objet **List<t>** hérite des opérations de l'interface **Collection** et sert à créer des collections dans lesquelles l'ordre des éléments est important. La valeur de chaque objet o est une liste *ordonnée dont les éléments sont de type t*, ce qui permet de référencer le premier, le dernier ou le i^{e} élément de la liste. Lorsqu'on ajoute un élément à la liste, il faut spécifier la position à laquelle il est inséré. Certaines des opérations de **List** sont représentées dans le listing C. Si o est un objet de type **List<t>**, l'opération **o.insert_element_first(e)** insère l'élément e avant le premier élément de la liste o, de sorte que e devient le premier élément. L'opération inverse (non représentée dans le listing) est **o.insert_element_last(e)**. L'opération **o.insert_element_after(e,i)** insère l'élément e après le i^{e} élément de la liste o et lève l'exception **Invalidlndex** si aucun i^{e} élément n'existe dans o. L'opération inverse (non représentée dans le listing) est **o.insert_element_before(e,i)**. Pour supprimer des éléments de la liste, les opérations sont **e = o.remove_first_element()**, **e = o.remove_last_element ()**, et **e=o.remove_element_at(i)**. Ces opérations suppriment l'élément spécifié de la liste et retournent l'élément résultant. D'autres opérations extraient un élément sans le supprimer de la liste. Ce sont **e=o.retrieve_first_element()**, **e = o.retrieve_last_element()** et **e = o.retrieve_element_at(i)**. Enfin, deux opérations permettent de manipuler des listes. Il s'agit de **p = o.concat(l)**, qui crée une nouvelle liste p issue de la concaténation des listes o et l (les éléments de la liste o suivis de ceux de la liste l), et **o.append(l)**, qui ajoute les éléments de la liste l à la fin de la liste o (sans créer de nouvelle liste).

Le type **Array<t>** hérite également des opérations de **Collection**. Il permet de créer des tableaux, comparables à des listes ayant un nombre fixe d'éléments. Les opérations spécifiques d'un objet o de type Array sont **o.replace_element_at(i,e)**, qui remplace l'élément occupant la position i par l'élément e, **e=o.remove_element_at(i)**, qui extrait le i^{e} élément et le remplace par la valeur null, et **e=o.retrieve_element_at(i)**, qui extrait simplement le i^{e} élément du tableau. Toutes ces opérations peuvent lancer l'exception **InvalidIndex** si i est supérieur à la taille du tableau. L'opération **o.resize(n)** change le nombre d'éléments du tableau en n.

Le dernier type d'objets collections est le type **Dictionary<k, v>**. Il permet de créer des paires associatives **<k,v>** où toutes les valeurs de k (la clé) sont uniques, et d'extraire une paire donnée en donnant la valeur de sa clé (semblable à un index). Si o est un objet collection de type **Dictionary<k, v>**, alors **o.bind (k, v)** lie la valeur v à la clé k sous forme d'association <k, v> dans la collection, tandis que **o.unbind(k)** supprime de o l'association dont la clé est k, et **v = o.lookup(k)** retourne la valeur v associée à la clé k. Les deux dernières opérations peuvent lever l'exception **KeyNotFound**. Enfin, **o.contains_key(k)** retourne vrai si la clé k existe dans o, faux sinon.

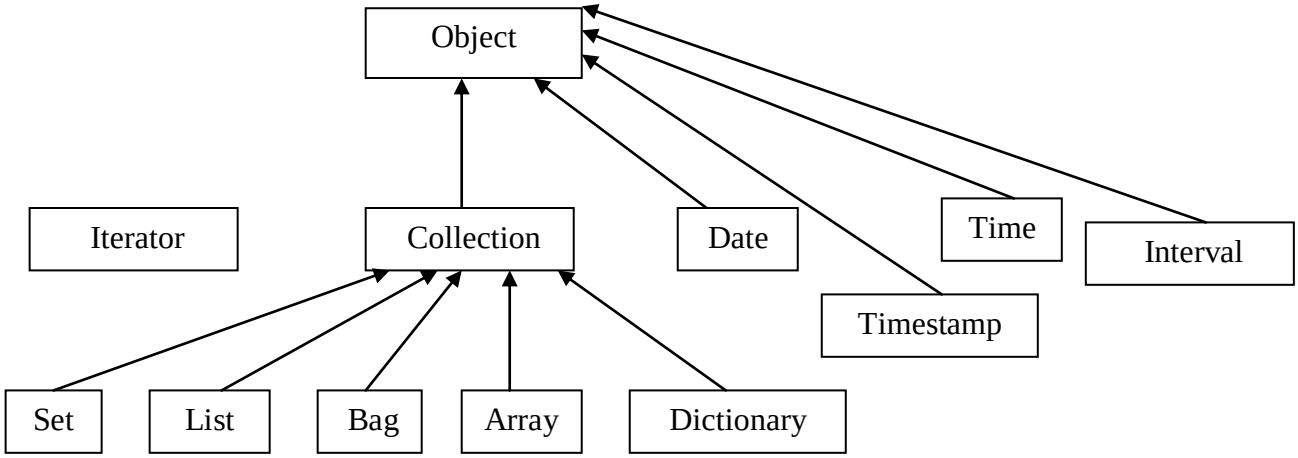

Figure 1 : Hiérarchie d'héritage pour les interfaces intégrées du modèle objet

La figure 1 représente la hiérarchie d'héritage des constructions intégrées du modèle objet. Les sous-types héritent des opérations du supertype. Les interfaces d'objets collections ne sont pas directement instanciables, ce qui signifie qu'elles ne permettent pas de créer directement des objets. En revanche, les interfaces peuvent servir à définir des objets collections définis par l'utilisateur - de type **Set**, **Bag**, **List**, **Array** ou **Dictionary** - pour une application particulière. Quand un utilisateur conçoit le schéma d'une base de données, il déclare les classes et les interfaces d'objets appropriées à son application. Si une interface ou une classe est l'un des objets collections, par exemple un Set, ce dernier héritera des opérations de l'interface Set. Par exemple, dans une base

de données destinée à une université, l'utilisateur peut, spécifier une classe pour Set <Etudiant>, dont les objets seront des ensembles d'objets Etudiant. Le programmeur peut alors utiliser les opérations de Set<t> pour manipuler un objet de type Set<Etudiant>.
Tous les objets d'une collection doivent être du même type. En conséquence, même si le mot clé **any** apparaît dans les spécifications des interfaces collections, cela ne signifie pas que des objets d'un type quelconque peuvent être mélangés dans une même collection, mais que tous les types sont éligibles lorsqu'on spécifie le type des éléments d'une collection donnée.

1.3 Objets atomiques (définis par l'utilisateur)

Nous allons maintenant étudier la façon dont on peut construire des types d'objets pour les objets atomiques. Ceux-ci sont spécifiés à l'aide du mot clé **class** en ODL. Dans le modèle objet, tout objet défini par l'utilisateur qui n'est pas une collection est qualifié d'**objet atomique**[9]. Par exemple, dans une application UNIVERSITE, l'utilisateur peut spécifier un type d'objet (une classe) pour les objets Etudiant. La plupart de ces objets seront des **objets structurés**; un objet Etudiant aura une structure complexe, avec de nombreux attributs, relations et opérations, mais il est toujours considéré comme atomique, parce qu'il ne s'agit pas d'une collection. Un tel type d'objet est défini sous forme de classe en spécifiant ses **propriétés** et ses **opérations**. Les propriétés définissent l'état de l'objet, et sont ensuite décomposées en **attributs** et **relations**.
Dans cette sous-section, nous étudions les trois types de composants - attributs, relations et opérations - qu'un type d'objet défini par l'utilisateur pour les objets atomiques (structurés) peut posséder. Cette analyse sera illustrée par les deux classes EMPLOYE et SERVICE représentées dans le listing D.

Listing D. Attributs, opérations et relations dans une définition de classe.

```
class Employe
( extent        tous_employes
  key           noss )
{
  attribute     string          nom;
  attribute     string          noss;
  attribute     date            datenaissance;
  attribute     enum Genre(M,F)       sexe;
        attribute       short   age;
        relationship    Service travaille_pour inverse      Service::a_emps;
        void            reaffecter_emp(in string, nouveau_snom) raises (snom_not_valid);
} ;
class Service
( extent        tous_services
  key           snom, snumero )
{
  attribute     string          snom;
        attribute               short           snumero;
        attribute               struct          Dir_SCE {Employe directeur, date datedebut} dir;
        attribute               set<string>     sites;
        attribute               struct          Projs {string nomprojet, time heures_hebdo} projs;
        relationship    set<Employe>    a_emps inverse   Employe :: travaille_pour;
        void            ajouter_emp(in string, nouveau_enom) raises (enom_not_valid);
        void            changer_dir(in string, nouveau_nom_dir; in date datedebut);
} ;
```

Un **attribut** est une propriété qui décrit un aspect d'un objet. Sa valeur, généralement un littéral doté d'une structure simple ou complexe, est mémorisée dans l'objet. Mais la valeur d'un attribut peut également être l'identifiant d'un autre objet. On peut même spécifier la valeur d'un attribut via des méthodes qui permettent de la calculer.
Dans le listing D, les attributs de Employe sont nom, noss, datenaissance, sexe et age, et ceux de Service sont snom, snumero, dir, sites et projs. Les attributs dir et projs de Service ont une structure complexe et sont définis via le constructeur struct, qui correspond au constructeur tuple du cours 1. En conséquence, la valeur dir de chaque objet Service aura deux composants: Directeur, dont la Valeur est un OBJECT_ID qui référence l'objet Employe qui dirige le service et datedebut dont la valeur est une date. L'attribut sites de Service est défini via le constructeur set, puisque chaque objet Service peut avoir un ensemble de sites.

[9] Cette définition d'un objet atomique dans le modèle de l'ODMG diffère de celle du constructeur atom donnée au cours 1, qui est la définition employée dans la plus grande partie de la littérature sur les bases de données orientées objet.

Une **relation** est une propriété qui spécifie que deux objets de la base sont liés. Dans le modèle objet de l'ODMG, seules les relations binaires sont explicitement représentées par une paire de références inverses spécifiées via le mot clé **relationship**. Dans le listing D, une relation unit chaque Employe au Service dans lequel il travaille -la relation *travaille_pour* de Employe. Dans la direction opposée, chaque Service est lié à l'ensemble d'Employes qui travaillent dans le Service – par la relation a_emps de Service. Le mot clé inverse spécifie que ces deux propriétés définissent une seule relation conceptuelle dans les deux directions. Grâce à la spécification de relations inverses, le système de base de données maintient automatiquement l'intégrité référentielle de la relation. Autrement dit, si la valeur travaille_pour d'un Employe e pointe Sur Service s, alors la valeur de a_emps de Service d doit inclure une référence à e dans son ensemble de références. Si le concepteur de la base désire représenter une relation unidirectionnelle, il doit la modéliser sous forme d'attribut (ou d'opération); exemple: le composant Directeur de l'attribut dir de Service.
Outre les attributs et les relations, le concepteur peut faire figurer des **opérations** dans les spécifications de types d'objets (ou classes). Chaque type d'objet peut avoir un certain nombre de **signatures d'opérations**, qui spécifient le nom de l'opération, les types de ses arguments et éventuellement sa valeur de retour. Les noms d'opérations sont uniques pour chaque type d'objet, mais on peut les surcharger en faisant apparaître le même nom d'opération pour deux types d'objet distincts. La signature peut également spécifier les noms des **exceptions** qui peuvent se produire au moment de l'exécution. L'implémentation de l'opération contiendra le code destiné à lever ces exceptions. Dans le listing D, la classe Employe a une seule opération (reaffecter_emp), et la classe Service en a deux (ajouter_emp et changer_dir).

1.4 Interfaces, classes et héritage

Le modèle objet de l'ODMG possède deux concepts pour spécifier des types d'objets : les interfaces et les classes. De plus, il existe deux types de relations d'héritage. Dans cette section, nous étudierons les différences et les similitudes entre ces deux concepts. Pour suivre la terminologie de l'ODMG, nous employons le mot **comportement** pour désigner les opérations, et nous parlons d'**état** pour les propriétés (attributs et relations).
Une **interface** est la spécification du comportement abstrait d'un type d'objets, qui définit les signatures des opérations. Bien que les spécifications d'une interface puissent comprendre des propriétés d'état (attributs et relations), celles-ci ne peuvent pas être héritées. Une interface est également non instanciable ; autrement dit, il est impossible de créer des objets qui correspondent à une définition d'interface.
Une **classe** spécifie à la fois le comportement et l'état abstraits d'un type d'objet, mais elle est instanciable, ce qui signifie que l'on peut créer des instances d'objets correspondant à la définition de la classe. Comme les interfaces ne sont pas instanciables, on les utilise principalement pour spécifier des opérations abstraites qui peuvent être héritées par des classes ou d'autres interfaces. C'est ce qu'on appelle l'**héritage de comportement**, et il est spécifié par le symbole « : ». En conséquence, dans le modèle de l'ODMG, l'héritage de comportement nécessite que le super-type soit une interface, tandis que le sous-type peut être soit une classe, soit une autre interface.
Une autre relation d'héritage, nommée EXTENDS et spécifiée par le mot clé **extends**, sert pour l'héritage d'états et de comportements strictement entre classes. Dans un héritage EXTENDS, le supertype et le sous-type peuvent tous deux être des classes. L'héritage multiple via EXTENDS n'est pas permis. Mais l'héritage multiple est autorisé pour l'héritage de comportement via « : ». Une interface peut donc hériter des comportements de plusieurs autres interfaces et hériter de l'état et du comportement d'au plus une autre classe via EXTENDS. Nous donnerons ci-dessous des exemples de la façon dont ces deux relations d'héritage, « : » et EXTENDS, peuvent être employées.

1.5 Extensions, clés et objets fabriques

Dans le modèle de l'ODMG, le concepteur d'une base de données peut déclarer une extension pour tout type d'objet défini via une déclaration de classe. Dans le listing D, les classes Employe et Service ont des extensions nommées respectivement tous_employes et tous_services. Cela revient à créer deux objets -l'un de type Set <Employe> et le second de type Set<Service> - et à les rendre persistants en les nommant tous_employes et tous_services.
On emploie également des extensions pour appliquer automatiquement la relation ensemble/sous-ensemble entre les extensions d'un supertype et de son sous-type. **Si deux classes A et B possèdent des extensions tous_A et tous_B, et si la classe B est un sous-type de la classe A (autrement dit, si la classe B étend la classe A), alors la collection d'objets contenue dans tous_B doit être un sous-ensemble de ceux de tous_A à n'importe quel temps t.** Cette contrainte est appliquée automatiquement par le système.

Une classe possédant une extension peut avoir une ou plusieurs clés. Une clé regroupe une ou plusieurs propriétés (attributs ou relations) dont les valeurs doivent obligatoirement être uniques pour chaque objet de l'extension. Par exemple, la classe Employe a pour clé l'attribut noss (la valeur de noss de chaque objet Employe de l'extension doit être unique), et la classe Service a deux clés distinctes: snom et snumero (chaque Service doit avoir un snom et un snumero uniques). Dans le cas d'une clé composite[10] constituée de plusieurs propriétés, les propriétés qui forment la clé apparaissent entre parenthèses. Par exemple, si une classe Vehicule possédant une extension tous_les_vehicules a une clé composée d'une combinaison de deux attributs departement et numero_de_permis, ces derniers seront placés entre parenthèses (departement, numero_de_permis) dans la déclaration de la clé.

Un **objet fabrique** (Factory) peut servir à générer ou à créer des objets individuels via ses opérations. Certaines des interfaces des objets fabriques qui font partie du modèle de l'ODMG sont représentées dans le listing E. L'interface ObjectFactory n'a qu'une seule opération, new(), qui retourne un nouvel objet avec un OBJECT _ID. Hériter de cette interface permet aux utilisateurs de créer leurs propres interfaces fabriques pour chaque type d'objet défini par l'utilisateur (objet atomique). Le programmeur peut implémenter l'opération new() différemment pour chaque type d'objet. Le listing E représente également une interface DateFactory, qui dispose d'opérations supplémentaires permettant notamment de créer une nouvelle calendar_date et de générer un objet dont la valeur est current_date (ces opérations ne sont pas représentées dans le listing E).

Un objet fabrique sert essentiellement à fournir les **opérations constructeurs** des nouveaux objets.

Voyons enfin le concept de **base de données**. Puisqu'un SGBDO peut créer de nombreuses bases de données différentes, chacune ayant son propre schéma, le modèle objet de l'ODMG possède des interfaces pour les objets DatabaseFactory et Database, comme le montre le listing E. Chaque base possède son propre nom de base de données. L'opération **bind** permet d'affecter des noms individuels uniques aux objets persistants. L'opération **lookup** retourne l'objet portant l'object_name spécifié. L'opération **unbind** supprime le nom d'un objet persistant nommé de la base de données.

Listing E. Interfaces pour illustrer les objets Factory et les objets Database.

```
interface ObjectFactory {
        Object          new();
} ;
interface DateFactory : ObjectFactory {
        exception InvalidDate{};
…
        Date    calendar_date    ( in unsigned short year,  in unsigned short month, in unsigned short day) raises(InvalidDate);
…
        Date    current();
} ;
interface DatabaseFactory {
        Database new();
} ;
interface Database {
        void            open    (in string database_name);
        void            close () ;
        void bind               (in any some_object, in string object_name);
        object unbind   (in string name);
        object lookup   (in string object_name)    raises(ElementNotFound);
…
}
```

2. Le langage de définition d'objets: ODL

Après cette vue d'ensemble du modèle objet de l'ODMG, nous allons voir comment ces concepts peuvent servir à créer un schéma de base de données objets avec le langage de définition d'objets ODL (Object Definition Language)[11]. ODL est conçu pour prendre en charge les constructions sémantiques du modèle objet de l'ODMG. Il est indépendant de tout langage de programmation. Il sert principalement à créer des spécifications d'objets - autrement dit des classes et des interfaces. En conséquence, ODL n'est pas un langage de programmation à part entière. Un utilisateur peut définir un schéma de base de données en ODL indépendamment de tout langage de programmation, puis utiliser une liaison de langage (language binding)

[10] Une clé composite se nomme clé composée dans le rapport de l'ODMG.

[11] La syntaxe et les types de données d'ODL sont prévus pour être compatibles avec l'IDL (Interface Definition Language) de CORBA (Common Object Request Broker Architecture), avec des extensions pour les relations et autres concepts des bases de données.

pour spécifier comment les constructions ODL correspondront aux constructions d'un langage spécifique comme C++, Smalltalk ou Java.
La figure 2 montre un schéma objet possible pour une partie de la base de données UNIVERSITE présentée au TD1. Nous décrirons les concepts d'ODL en nous appuyant sur cet exemple et sur celui de la figure 3. La notation graphique est présentée en haut de la figure 2. On peut la considérer comme une variante des diagrammes EER, avec la notion supplémentaire d'héritage d'interface mais à laquelle manquent plusieurs concepts EER, tels les catégories (types union) et les attributs de relations.
Le listing F présente un ensemble de définitions de classes possibles en ODL pour la base de données UNIVERSITE. Il existe généralement plusieurs façons de traduire un diagramme de schéma objet (ou un diagramme de schéma EER) en classes ODL. Nous avons choisi la représentation la plus simple. Les types d'entité sont traduits en classes ODL, et l'héritage est obtenu au moyen d'extensions. Mais il n'y a aucun moyen direct de traduire les catégories (types union) ou de réaliser l'héritage multiple. Les classes Personne, Professeur, Etudiant, et EtudDip ont respectivement les extensions personnes, professeurs, etudiants et etudiants_dip. Professeur et Etudiant dérivent toutes deux de Personne, et EtudDip étend Etudiant. De ce fait, on peut forcer la collection etudiants (et la collection professeurs) à être à tout moment un sous-ensemble de la collection personnes. De même, la collection etudiants_dip sera un sous-ensemble de etudiants. En même temps, les objets individuels de type Etudiant et Professeur hériteront des propriétés (attributs et relations) et des opérations de Personne, et les objets de type EtudDip hériteront de celles de Etudiant.
Les classes Departement, Cours, Session et SessionCourante sont de simples transpositions des types d'entités correspondants. Mais la classe Note nécessite quelques explications. Elle correspond en effet à la relation M:N qui existe entre les classe Etudiant et Session. La raison pour laquelle on a créé une classe séparée (au lieu d'une paire de relations inverses) est qu'elle comprend un attribut de relation. En conséquence, la relation M:N est associée à la classe Note, et il existe une paire de relations 1:N, l'une entre Etudiant et Note et l'autre entre Session et Note. Ces deux relations sont représentées par les propriétés suivantes: sessions_terminees de Etudiant, session et etudiant de Note, et etudiants de Session (voir listing F). Enfin, la classe Diplome sert à représenter l'attribut composite multivalué diplomes de EtudDip.

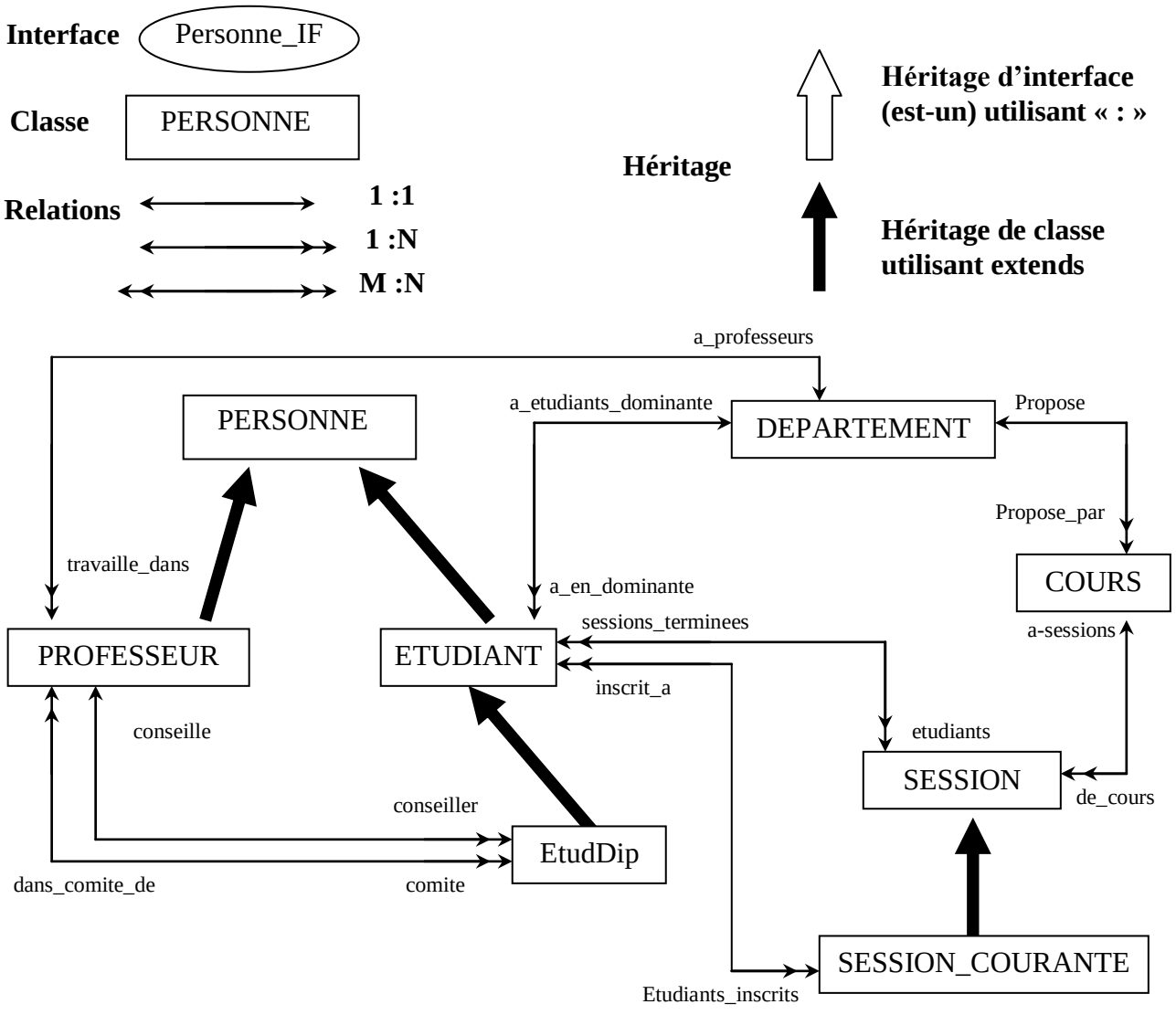

Figure 2 : Schéma d'un partie de la BD UNIVERSITE avec la notation graphique des schémas en ODL

Listing F. Schéma ODL possible pour la base de données UNIVERSITE de la figure 2.

```
class Personne
(       extent  personnes
        key     noss )
{ attribute     struct  PNom {string prenom, string deuxiemeprenom, string nomFamille}  nom;
  attribute     string  noss;
  attribute     date    datenaissance;
  attribute     enum    Genre {M,F}     sexe ;
  attribute     struct  Adresse {short no, string rue, short noappt, string ville, string dep, short codepost} adresse;
  short         age() ;
} ;

class Professeur extends Personne   (       extent  professeurs)
{ attribute     string  rang ;
  attribute     float   salaire;
  attribute     string  bureau;
  attribute     string  tel;
  relationship  Departement     travaille_dans  inverse  Departement :: a_professeurs;
  relationship  set<EtudDip>    conseille       inverse  EtudDip :: conseiller;
  relationship  set<EtudDip>    dans_comite_de  inverse  EtudDip :: comite;
  void          augmenter(in float augmentation);
  void          promouvoir(in string nouveau_rang) ;
} ;
class Note      (       extent  Notes)
{ attribute     enum ValeursNotes {A, B, C, D, F, I, P} note;
  relationship  Session session  inverse  Session :: etudiants;
  relationship  Etudiant        etudiant inverse Etudiant  :: sessions_terminees;
} ;

class Etudiant extends Personne     (       extent  etudiants );
{  attribute    string          annee;
   attribute    Departement     en_sous_dominante;
   relationship Departement     a_en_dominante  inverse  Departement :: a_etudiants_dominante;
   relationship set<Note>       sessions_terminees inverse      Note :: etudiants;
   relationship set<SessionCourante> inscrits_a   inverse  SessionCourante :: etudiants_inscrits;
   void         changer_dominante(in string dnom) raises (dnom_not_valid);
   float        MP() ;
   void         sinscrire(in short nosession) raises (session_not_valid);
   void         note_attribuee(in short nosession ; in ValeursNotes note) raises (Session_not_valid, note_not_valid);
} ;

class Diplome
{  attribute    string  fac;
   attribute    string  diplome;
   attribute    string  an;
} ;

class EtudiantDip extends Etudiant              ( extent            etudiants_dip     )
{  attribute    set<Diplome> diplomes;
   relationship Professeur      conseiller      inverse  Professeur:: conseille;
   relationship set<Professeur> comite          inverse  Professeur :: dans_comite_de;
   void         attribuer_conseiller(in string nom; in string prenom) raises (professeur_not_valid);
   void         attribuer_comite(in string nom; in string prenom) raises (professeur_not_valid);
} ;

class Departement       (  extent  departements   key      dnom)
{  attribute    string          dnom;
   attribute    string          dtel;
   attribute    string          dbureau;
   attribute    string          fac;
```

```
    attribute     Professeur        pdt;
    relationship  set<Professeur>   a_professeurs            inverse  Professeur :: travaille_dans;
    relationship  set<Etudiant>     a_etudiants_dominante    inverse  Etudiant :: a_en_dominante;
    relationship  set<Cours>        propose                  inverse  Cours :: propose_par;
} ;

class Cours          (  extent cours   key      nocours   )
{  attribute     string          dnom;
    attribute     string          nocours;
    attribute     string          description;
    relationship   set<Session>   a_sessions               inverse  Session :: de_cours;
    relationship  Departement     propose_par              inverse          Departement:: propose;
} ;

class Session      (  extent       sessions)
{   attribute     short     nosession;
     attribute     string    an;
    attribute     enum      trimestre {printemps, ete, automne, hiver}trim;
    relationship  set<Note>         etudiants                       inverse  Note:: session;
    relationship  Cours             de_cours                        inverse  Cours:: a_sessions;
} ;
class SessionCourante extends Session  (extent sessions_courantes )
{ relationship    set<Etudiant>     etudiants inscrits        inverse  Etudiant:: inscrit_a;
  void            inscrire_etudiant (in string noss) raises (etudiant_not-valid, session_pleine);
} ;
```

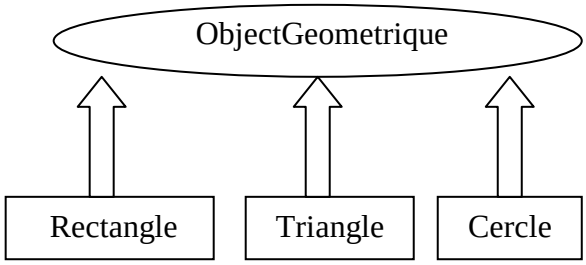

Figure 3 : Illustration de l'héritage d'interface via «: ». Représentation graphique du schéma

L'exemple précédent (figure 2) ne contenait aucune interface mais seulement des classes. Prenons un autre exemple pour illustrer les interfaces et l'héritage d'interface (ou de comportement). La figure 3 fait partie du schéma d'une base de données qui stocke des objets géométriques. On spécifie une interface *ObjetGeometrique*, avec des opérations pour calculer le périmètre et la surface d'un objet, ainsi que des opérations pour translater (déplacer) et faireTourner un objet. Plusieurs classes (*Rectangle*, *Triangle*, *Cercle*, etc.) héritent de l'interface ObjetGeometrique. Comme ObjetGeometrique est une interface, elle n'est pas instanciable - autrement dit, elle ne permet pas de créer directement des objets. Mais on peut créer des objets de type *Rectangle*, *Triangle*, *Cercle*... Ces objets héritent de toutes les opérations de l'interface *ObjetGeometrique*. Notez que, dans l'héritage d'interface, les opérations sont héritées, mais pas les propriétés (attributs, relations). En conséquence, si une propriété est nécessaire dans une classe qui hérite, elle doit être répétée dans la définition de cette classe, comme pour l'attribut point_de_reference dans le listing G. Remarquez que les opérations héritées peuvent avoir des implémentations différentes dans chaque classe. Par exemple, les implémentations des opérations surface et perimetre seront différentes pour les objets Rectangle, Triangle et Cercle.

Une classe peut hériter de plusieurs interfaces, de même qu'une interface peut hériter de plusieurs interfaces. Mais avec EXTENDS (héritage de classe), l'héritage multiple est interdit. Une classe ne peut donc hériter via EXTENDS que d'une seule autre classe (et hériter en plus de zéro ou plusieurs interfaces).

Listing G -Illustration de l'héritage d'interface via « : » - Définition des interfaces et des classes en ODL.

```
interface ObjetGeometrique
{  attribute     enum    Forme{Rectangle,Triangle,Cercle,...} forme;
   attribute     struct  Point {short x, short y} point_de_reference;
   float         perimetre();
   float         aire();
   void          translater(in short x_translation; in short y_translation);
   void          faire_tourner(in float angle_de_rotation);
```

```
} ;
class Rectangle: ObjetGeometrique ( extent rectangles)
{  attribute     struct   Point {short x, short y} point_de_reference;
   attribute     short    longueur;
   attribute     short    hauteur;
   attribute     float    angle_orientation;
} ;
class Triangle: ObjetGeometrique ( extent triangles)
{ attribute     struct   Point {short x, short y} point_de_reference;
   attribute     short    cote_1 ;
   attribute     short    cote_2 ;
   attribute     float    cote1_cote2_angle;
   attribute     float    cte1_angle_orientation;
} ;
class Circle : ObjetGeometrique ( extent circles )
{  attribute     struct   Point {short x, short y} point_de_reference;
   attribute     short    rayon;
} ;
```

3. Le langage de requête objet: OQL

OQL (Object Query Language) est le langage de requête proposé pour le modèle objet de l'ODMG. Il est conçu pour collaborer étroitement avec les langages de programmation pour lesquels une liaison ODMG est définie, par exemple C++, Smalltalk et Java. Une requête OQL encapsulée dans un de ces langages peut retourner des objets qui correspondent au langage utilisé. En outre, il est possible d'écrire dans le langage en question les implémentations des opérations des classes d'un schéma ODMG. La syntaxe des requêtes OQL est semblable à celle de SQL, le langage de requête standard des bases de données relationnelles, avec des fonctionnalités supplémentaires pour les concepts ODMG (identité des objets, objets complexes, opérations, héritage, polymorphisme et relations). Les exemples destinés à illustrer les requêtes OQL sont tirés du schéma de la base de données UNIVERSITE du listing F.

3.1 Requêtes OQL simples, points d'entrées dans une base de données et variables itérateurs

La syntaxe **OQL** de base est une structure **select** … **from** … **where** … comme en **SQL**. Par exemple, la requête qui extrait les noms de tous les départements de la faculté « Ingénierie »peut s'écrire comme suit:

```
Q0:   SELECT     d.dnom
      FROM       d in departements
      WHERE      d.fac = 'Ingénierie';
```

En général, chaque requête nécessite un point d'entrée dans la base de données, qui peut être un objet persistant nommé quelconque. Pour de nombreuses requêtes, ce point d'entrée est le nom de l'extension d'une classe. Souvenez-vous que l'on considère que le nom de l'extension est le nom d'un objet persistant dont le type est une collection (dans la plupart des cas un set) des objets de la classe. Si l'on regarde les noms des extensions du listing F, l'objet nommé départements est de type set<Departement> ; personnes est de type set<Personne> ; professeurs est de type set<Professeur> et ainsi de suite.

L'emploi d'un nom d'extension - departements dans la requête **Q0** - comme point d'entrée référence une collection persistante d'objets. Chaque fois qu'une collection est référencée dans une requête **OQL**, nous devons définir une **variable itérateur** -d dans la requête **Q0** - qui parcourt chaque objet de la collection. Dans de nombreux cas, comme dans cet exemple, la requête sélectionnera dans la collection, en fonction des conditions spécifiées dans la clause where. Dans la requête **Q0**, seuls les objets persistants d de la collection de departements qui satisfont à la condition **d.fac = 'Ingénierie'** sont récupérés dans le résultat de la requête. Pour chaque objet d sélectionné, la valeur de **d.dnom** est retournée. En conséquence, le type du résultat de **Q0** est **bag<string>**, parce que le type de chaque valeur de dnom est string (même si le résultat réel de la requête est un set parce que dnom est un attribut de clé). En général, le résultat d'une requête est du **type bag** pour **select** … **from** … et de type **set** pour **select** … **distinct** … **from** …, comme en **SQL** (l'ajout du mot clé distinct élimine les doublons) .

Pour reprendre l'exemple **Q0**, il existe trois syntaxes possibles pour spécifier un itérateur :

- **d in departments**
- **departements d**
- **departements as d**

Nous utiliserons la première syntaxe dans nos exemples. Notez que les deux dernières options sont comparables à la spécification de variables de tuples dans une requête SQL.
Les objets nommés qui servent de point d'entrée dans les requêtes OQL ne sont pas limités aux noms d'extensions. Tout objet persistant nommé, qu'il référence un objet atomique (isolé) ou un objet collection, peut servir de point d'entrée.

3.2 Résultats des requêtes et expressions de chemin

Le résultat d'une requête peut généralement être de tout type susceptible d'être exprimé dans le modèle objet de l'ODMG. Une requête n'applique pas obligatoirement la structure **select** … **from** … **where.** Dans le cas le plus simple, tout nom persistant peut constituer une requête. Le résultat de cette requête est une référence à cet objet persistant. Par exemple, la requête:

```
Q1:  departments;
```

retourne une référence à la collection de tous les objets persistants departements, collection dont le type est **set<Departement>**. De même, si nous avons attribué (via l'opération **bind**, voir listing E) le nom persistant csdepartement à un seul objet departement (le département informatique), la requête:

```
Q1a: csdepartment;
```

retourne une référence à cet unique objet de type Departement. Une fois un point d'entrée spécifié, l'expression de chemin permet de spécifier un chemin vers les attributs et les objets associés. Une expression de chemin commence habituellement par un nom d'objet persistant, ou par la variable itérateur qui parcourt les objets d'une collection. Ce nom est suivi de zéro ou de plusieurs noms de relations ou de noms d'attributs connectés par des points. Par exemple, dans la base de données du listing F, les lignes suivantes sont des exemples d'expressions de chemin, qui sont également des requêtes valides en **OQL** :

```
Q2:  csdepartment.pdt;
Q2a: csdepartment.pdt.rang;
Q2b: csdepartment.a_professeurs;
```

La première expression, **Q2**, retourne un objet de type Professeur, parce que c'est le type de l'attribut pdt de la classe Departement. C'est une référence à l'objet Professeur qui est lié à l'objet Departement dont le nom persistant est csdepartment via l'attribut pdt, autrement dit une référence à l'objet Professeur qui est le président du département informatique. La seconde expression, **Q2a**, est similaire, excepté qu'elle retourne le rang de cet objet Professeur (président du département informatique) et non la référence à l'objet. Le type retourné par **Q2a** est donc string, c'est-à-dire le type de données de l'attribut rang de la classe Professeur.
Chaque expression **Q2** et **Q2a** ne retourne qu'une seule valeur, parce que les attributs pdt (de Department) et rang (de Professeur) sont tous deux monovalués et sont appliqués à un seul objet. La troisième expression, **Q2b**, est différente: elle retourne un objet de type **set<Professeur>**, même si elle est appliquée à un unique objet, parce que c'est le type de la relation a_professeurs de la classe Departement. La collection retournée contiendra des références à tous les objets Professeur associés à l'objet departement dont le nom persistant est csdepartement via la relation a_professeurs, autrement dit des références à tous les objets Professeur qui travaillent dans le département informatique. Mais pour obtenir les rangs des enseignants du département informatique, nous ne pouvons pas écrire:

```
Q3': csdepartment.a_professeurs.rang;
```

La raison en est que nous ne savons pas si l'objet retourné serait de type **set<string>** ou **bag<string>** (le dernier étant le plus probable, puisque plusieurs enseignants peuvent partager le même rang). Pour éliminer ce genre d'ambiguïté, **OQL** n'autorise pas d'expressions comme **Q3'**. En revanche, on doit utiliser un itérateur pour parcourir ces collections, comme dans les requêtes **Q3a** ou **Q3b** ci-après:

```
Q3a: select   f.rang
     from     f in csdepartment.a_professeurs;
Q3b: select   distinct f.rang
     from     f in csdepartment.a_professeurs;
```

Ici, **Q3a** retourne **bag<string>** (des doublons apparaissent dans le résultat), alors que **Q3b** retourne **set<string>** (les doublons sont éliminés par le mot clé **distinct**). Les requêtes **Q3a** et **Q3b** illustrent toutes deux la façon dont on peut définir dans la clause **from** une variable qui opérera une itération sur une collection restreinte spécifiée dans la requête. La variable f de **Q3a** et **Q3b** parcourt les éléments de la collection csdepartement.a_professeurs, qui est de type set <Professeur>, et n'en extrait que les enseignants qui sont membres du département informatique.

En général, une requête OQL peut retourner un résultat doté d'une structure complexe spécifiée dans la requête elle-même au moyen du mot clé **struct**. Considérez les deux exemples suivants:

```
Q4:  csdepartement.pdt.conseille;
Q4a: select    struct (  nom:struct(nom_personne: s.nom.nomfamille,
                         prenom_personne: s.nom.prenom),
                         diplomes:(select struct( dip: d.diplome,
                                                  an: d.an,
                                                  fac: d.fac)
                                   from d in s.diplomes)
                from s in csdepartement.pdt.conseille;
```

Ici, **Q4** est une requête simple qui retourne un objet de type **set<EtudDip>** (la collection d'étudiants diplômés dont le directeur de recherches est le président du département informatique). Supposons maintenant que l'on ait besoin d'une requête qui extraie le nom et le prénom de ces étudiants, ainsi que la liste de leurs diplômes. Nous pouvons écrire la requête **Q4a**, dans laquelle la variable s parcourt la collection d'étudiants diplômés supervisés par le président, et la variable d parcourt les diplômes de chaque étudiant s. Le type du résultat de **Q4a** est une collection de structs (premier niveau), dans laquelle chaque structure a deux composants: nom et diplomes. Le composant nom est une structure de deuxième niveau, constituée de nomfamille et prenom, chacun étant formé d'une seule chaîne. Le composant diplomes est défini par une requête imbriquée. C'est une collection de structures de deuxième niveau dont les composants sont trois chaînes: dip, an et fac.

Notez que **OQL** est **orthogonal** en ce qui concerne la spécification des expressions de chemin. Autrement dit, les noms d'attributs, de relations et d'opérations (méthodes) sont interchangeables dans les expressions, tant que le système de types d'OQL n'est pas compromis. Nous pouvons par exemple écrire les requêtes suivantes pour extraire la moyenne des points de tous les étudiants seniors se spécialisant en informatique. Le résultat est trié dans l'ordre décroissant des moyennes, puis dans l'ordre croissant des noms et des prénoms:

```
Q5a: select struct( nom_personne: s.nom.nomfamille, prenom_personne:
                    s.nom.prenom, mp: s.mp)
     from s in csdepartement.a_étudiants_dominante
     where s.anneee = 'senior'
     order by mp desc, nom_personne asc, prenom_personne asc;
Q5b: select struct( prenom_personne: s.nom.nomfamille, prenom_personne:
                    s.nom.prenom, mp: s.mp)
     from s in etudiants
     where s.en_dominante.dnom = 'Informatique'and s.annee = 'senior'
     order by mp desc, nom_personne asc, prenom_personne asc;
```

Q5a utilise directement le point d'entrée nommé csdepartement pour localiser directement la référence au département informatique, puis localise les étudiants via la relation a_étudiants_dominante, alors que **Q5b** recherche dans l'extension etudiants pour localiser tous les étudiants en dominante dans ce département. Remarquez la façon dont les noms d'attributs, les noms de relations et les noms d'opérations (méthodes) sont **interchangeables (de manière orthogonale)** dans les expressions de chemin: mp est une opération, en_dominante et a_étudiants_dominante sont des relations et annee, nom, dnom, nom et prenom sont des attributs. L'implémentation de l'opération mp calcule la moyenne des points et retourne une valeur de type float pour chaque étudiant.

La clause **order by** est similaire à la construction SQL correspondante, et spécifie dans quel ordre le résultat de la requête doit être affiché. C'est pourquoi la collection retournée par une requête contenant une clause order by est de type **list**.

3.3 Autres fonctionnalités d'OQL

a) Spécification de vues avec des requêtes nommées

Le mécanisme des **vues** en **OQL** s'appuie sur le concept de requête nommée. Le mot clé **define** permet de spécifier l'identificateur de la requête, qui doit être un nom unique parmi tous les objets nommés, noms de classes, noms de méthodes ou noms de fonctions du schéma. Si l'identificateur porte le même nom qu'une requête nommée existante, la nouvelle définition remplace la précédente. Une fois définie, une requête nommée est persistante tant qu'elle n'est pas redéfinie ou supprimée. Une définition de vue peut également avoir des

paramètres (arguments). Par exemple, la vue **V1** définit une requête nommée **a_en_sousdominante** pour extraire l'ensemble des étudiants ayant choisi un département donné comme sous-dominante:

```
V1:  define    a_en_sousdominante(deptnom) as
               select s
               from s in etudiants
               where s.en_sous_dominante.dnom = deptnom;
```

Comme le schéma du listing F ne fournit qu'un attribut unidirectionnel **en_sous_dominante** pour un etudiant, nous pouvons utiliser la vue ci-dessus pour représenter son inverse sans définir explicitement une relation. Ce type de vue peut servir à représenter des relations inverses dont on suppose qu'elles ne seront pas utilisées fréquemment. L'utilisateur peut maintenant exploiter la vue ci-dessus pour écrire des requêtes du type:

A_en_sousdominante('Informatique') ;

Celle-ci retournerait une collection de type bag d'étudiants ayant choisi le département informatique comme sous-dominante. Notez que, dans le listing F, nous avons défini une relation explicite **a_en_dominante**, probablement parce que nous nous attendons à l'utiliser plus souvent.

b) Extraire un seul élément d'une collection singleton

Une requête **OQL** retourne généralement une collection, de type bag, set (si distinct est spécifié) ou list (si la clause order by est utilisée). Si l'utilisateur désire qu'une requête ne retourne qu'un seul élément, OQL dispose d'un opérateur **element** qui extraira toujours un seul élément e d'une collection singleton c qui ne contient que ce seul élément. Si c contient plus d'un élément ou si elle est vide, l'opérateur element lève l'exception. Par exemple, **Q6** retourne le seul objet qui référence le département informatique:

```
Q6:  element (select d
     from d in departements
     where d.dnom = 'Informatique');
```

Comme tous les noms de départements sont uniques, le résultat doit être un seul département. Le type du résultat est d:Departement.

c) Opérateurs sur les collections (calculs d'agrégats, quantificateurs)

Comme de nombreuses expressions de requête spécifient une collection comme type de retour, un certain nombre d'opérateurs qui s'appliquent à ces collections ont été définis. Il s'agit notamment des opérateurs qui permettent de calculer des agrégats ou qui testent l'appartenance à une collection et des quantificateurs (universels et existentiels).

Les opérateurs de calcul d'agrégats (min, max, count, sum et avg) opèrent sur une collection. L'opérateur count retourne un entier. Les autres opérateurs (min, max, sum, avg) retournent un résultat du même type que la collection opérande. Voici deux exemples. La requête **Q7** retourne le nombre d'étudiants ayant l'informatique pour sous-dominante, tandis que **Q8** retourne la moyenne mp de tous les étudiants seniors dont l'informatique est la dominante.

```
Q7:  count (s in a_en_sousdominante('Informatique'));
Q8:  avg (select s.mp
     from s in etudiants
     where s.en_dominante.dnom = 'Informatique'and s.annee = 'senior');
```

Notez que ces opérations peuvent être appliquées à toute collection du type approprié et peuvent figurer dans n'importe quelle partie d'une requête. Par exemple, la requête qui extrait les noms de tous les départements ayant plus de cent étudiants en dominante peut s'écrire comme suit:

```
Q9:  select d.dnom
     from d in departements
     where count (d.a_etudiants_dominante) > 100;
```

Les expressions qui testent l'appartenance et les quantificateurs retournent un booléen - autrement dit true ou false. Si v est une variable, c l'expression d'une collection, b une expression de type boolean (une condition booléenne) et e un élément du type des éléments de la collection c. Dans ce cas:

- **(e in c)** retourne **true** si l'élément e est membre de la collection c.
- **(for all v in c: b)** retourne **true** si tous les éléments de la collection c satisfont la condition b.
- **(exists v in c: b)** retourne **true** s'il existe au moins un élément de c qui satisfait la condition b.

Pour illustrer la condition d'appartenance, supposons que nous voulons extraire les noms de tous les étudiants qui ont suivi le cours nommé « Bases de données I ». Nous pouvons écrire la requête **Q10**, dans laquelle la

requête imbriquée retourne la collection de noms des cours que chaque étudiant a suivis, et où la condition d'appartenance retourne true si « Bases de données I » se trouve dans la collection pour un étudiant s donné:

```
Q10: select s.nom.nomfamille, s.nom.prenom
     from s in etudiants
     where 'Bases de données I' in
           (select c.cnom
            from c in s.sessions_terminées.session.de_cours);
```

La requête **Q10** est une façon plus simple de spécifier la clause select dans une requête qui retourne une collection de structs. Le type du résultat retourné par **Q10** est **bag<struct(string, string)>**.

Il est également possible d'écrire des requêtes qui retournent des résultats true/false. Supposons par exemple qu'il existe un objet nommé Mohamed de type Etudiant. La requête **Q11** répond à la question suivante: « La sous-dominante de Mohamed est-elle l'informatique? » De même, la requête **Q12** répond à la question: « Tous les étudiants seniors en informatique sont-ils conseillés par un enseignant du département informatique? ». **Q11** et **Q12** retournent true ou false, que l'on interprète comme des réponses positives ou négatives:

```
Q11: Mohamed in a_en_dominante('Informatique');
Q12: for all g in
     (select s
     from s in etudiants_dip
     where s.en_dominante.dnom= 'Informatique')
     : g.conseiller in csdepartement.a_professeurs ;
```

Notez que la requête **Q12** illustre également la façon dont l'héritage d'attributs, de relations et d'opérations s'applique aux requêtes. Bien que s soit un itérateur qui parcourt l'extension etudiants_dip, nous pouvons écrire s.en_dominante parce que la relation en_dominante est héritée par EtudDip de Etudiant via extends (voir listing F). Enfin, pour illustrer le quantificateur **exists**, la requête **Q13** répond à la question suivante: « Existe-t-il un étudiant en dominante informatique qui a une moyenne de 4? » Ici encore, l'opération mp est héritée par EtudDip de Etudiant via extends.

```
Q13: exists g in
     (select s
     from s in etudiants_dip
     where s.en_dominante.dnom = 'Informatique')
     : g.mp = 4;
```

d) Collections ordonnées (indexées)

Comme nous l'avons vu, les collections sont des listes et des tableaux qui disposent d'opérations supplémentaires, qui permettent par exemple d'obtenir le i^{e}, le premier ou le dernier élément. En outre, il existe des opérations pour extraire des sous-collections et pour concaténer deux listes. En conséquence, les requêtes qui concernent des listes ou des tableaux peuvent invoquer ces opérations. Nous illustrerons quelques-unes de ces opérations par des exemples de requêtes. La requête **Q14** affiche le nom des membres du corps enseignant qui ont le salaire le plus élevé:

```
Q14: first (select struct(professeurs: f.nom.nomfamille,
                          salaire:f.salaire)
            from    f in professeurs
            order by f.salaire desc) ;
```

La requête **Q14** illustre l'emploi de l'opérateur **first** sur une collection de type liste qui contient les salaires des membres du corps enseignant triés par ordre décroissant. Le premier élément de cette liste triée est donc l'enseignant dont le salaire est le plus élevé. Cette requête suppose qu'un seul enseignant gagne le salaire maximal. La requête suivante, **Q15**, extrait les trois meilleurs étudiants en dominante informatique sur la base de leur moyenne (mp).

```
Q15: (select struct(nom_personne: s.nom.nomfamille,
                    prenom_personne:s.nom.prenom, mp: s.mp)
     from s in csdepartement.a_en_dominante order by mp desc) [0:2] ;
```

La requête **select … from … order by** retourne une liste d'étudiants se spécialisant en informatique, triés en fonction de leur moyenne par ordre décroissant. Le premier élément d'une collection ordonnée occupe l'indice 0. L'expression **[0: 2]** retourne donc une liste contenant les trois premiers éléments du résultat de la requête.

e) L'opérateur de groupement

La clause OQL **group by**, bien que similaire à son homologue en SQL, fournit une référence explicite à la collection d'objets contenue dans chaque groupe ou partition. Après un exemple, nous décrirons la forme générale de ces requêtes.

La requête **Q16** affiche le nombre d'étudiants en dominante de chaque département. Dans cette requête, les étudiants sont groupés dans la même partition (groupe) s'ils ont la même dominante, autrement dit si les valeurs de s.en_dominante.dnom sont identiques pour chacun:

```
Q16: (select struct(deptnom, nombre_de_dominantes:count (partition))
      from s in etudiants
      group by deptnom:s.en_dominante.dnom;
```

Le résultat du groupement est de type **set<struct(deptnom: string, partition: bag<struct (s: Etudiant) >) >**. Il contient pour chaque groupe (partition) une structure formée de deux composants: la valeur de l'attribut de groupement (deptnom) et la collection (bag) des objets Etudiants du groupe (partition). La clause select retourne l'attribut de groupement (le nom du département) ainsi que le nombre d'éléments de chaque partition (autrement dit, le nombre d'étudiants de chaque département), où **partition** est le mot clé servant à référencer chaque partition. Le type du résultat de la clause select est **set<struct (deptnom: string, nombre_de_dominantes: integer)** >. La forme générale de la clause group by est:

```
group by f1: e1, f2: e2 . . ., fk: ek
```

où f1: e1, f2: e2, ... , fk: ek est une liste d'attributs de partitionnement (groupement), et où chaque spécification d'attribut fi: ei définit un **nom d'attribut (champ) fi** et une **expression ei**. Le résultat de l'application de ce groupement (spécifié dans la clause group by) est un ensemble de structures:

```
set<struct(f1: t1, f2: t2, . . ., fk: tk, partition: bag<B>)>
```

où ti est le type retourné par l'expression ei, **partition** un nom de champ distingué (un mot clé) et B une structure dont les champs sont les itérateurs du type approprié (s dans Q16) déclarés dans la clause from.

Comme en SQL, on peut utiliser une clause **having** pour filtrer les ensembles partitionnés (autrement dit, ne sélectionner que certains groupes en fonction de conditions de groupement). En **Q17**, la requête précédente est modifiée pour illustrer la clause having (et présenter la syntaxe simplifiée de la clause select). La requête **Q17** extrait pour chaque département ayant plus de 100 étudiants en dominante la moyenne de ces derniers. La clause having ne sélectionne que les partitions (groupes) qui possèdent plus de 100 éléments (autrement dit, les éléments ayant plus de 100 étudiants).

```
Q17: (select deptnom, avg_mp: avg (select p.s.mp from p in partition)
      from s in etudiants
      group by deptnom:s.en_dominante.dnom
      having count (partition) > 100;
```

Notez que cette clause select retourne la moyenne (mp) des étudiants de la partition. L'expression :

```
select p.s.mp from p in partition
```

retourne une collection (bag) de moyennes d'étudiants pour cette partition. La clause from déclare un itérateur p sur la partition, qui est de type **bag<struct (s: Etudiant)** >. Puis l'expression de chemin **p.s.mp** sert à accéder à la moyenne de chaque étudiant de la partition.

4. Résumé

Dans ce cours, nous avons étudié le standard proposé pour les bases de données orientées objet.

Nous avons commencé par décrire les différentes constructions du modèle objet de l'ODMG. Les différents types intégrés, tels **Object**, **Collection**, **Iterator**, **Set**, **List**, etc., ont été décrits par leurs interfaces, qui spécifient les opérations intégrées de chaque type. Ces types intégrés constituent les fondations sur lesquelles sont construits le langage de définition d'objets ODL (Object Definition Language) et le langage de requête objet OQL (Object Query Language).

Nous avons également abordé la différence entre les objets, qui ont un identifiant, et les littéraux, qui n'en ont pas.

Les utilisateurs peuvent déclarer dans leur application des classes qui héritent des opérations interfaces intégrées appropriées. Il est possible de spécifier deux types de propriétés dans une classe définie par l'utilisateur -les attributs et les relations - en plus des opérations applicables aux objets de la classe.

ODL permet aux utilisateurs de définir aussi bien des interfaces que des classes, et autorise deux types d'héritage -l'héritage d'interface via « : » et l'héritage de classe via extends. Une classe peut posséder une extension et des clés.

Nous avons ensuite décrit ODL, en nous appuyant sur le schéma de l'exemple de base de données UNIVERSITE pour illustrer ses constructions. Puis nous avons présenté une vue d'ensemble du langage de requêtes (OQL).
OQL applique le concept d'orthogonalité dans la construction des requêtes; en d'autres termes, une opération peut être appliquée au résultat d'une autre opération tant que le type du résultat est bien celui qui est attendu par l'opération.
La syntaxe OQL ressemble beaucoup à celle de SQL, mais comprend des concepts supplémentaires telles que les expressions de chemin, l'héritage, les méthodes, les relations et les collections.