LE STANDARD DE L'ODMG :

ODL, OQL ET OML

1. INTRODUCTION

Depuis 1988, une dizaine de petites sociétés commercialisent des SGBDO, avec un succès encore limité. Elles se heurtent au problème de la portabilité des applications. Il existe maintenant des langages de programmation objet ayant une bonne portabilité comme C++, ou vraiment portables comme Java, qui a été conçu pour être porté. Mais porter des applications accédant à des bases de données exige la portabilité des interfaces d'accès. Ceci est possible en relationnel, avec SQL et des *middlewares* universels comme ODBC ou JDBC pour Java. Ceci était très difficile en objet devant l'absence de standards.

Ainsi, afin de définir des interfaces portables, s'est constitué l'*Object Database Management Group* (**ODMG**), formé au départ par cinq vendeurs de SGBDO. L'ODMG vise à réaliser pour les bases de données objets l'équivalent de la norme SQL, ou au moins d'un projet de norme. Deux versions du standard proposé ont été publiées assez rapidement : l'une en 1993 [Odmg93], l'autre présentée dans ce chapitre, en 1997 [Odmg97]. Un des buts des SGBDO, et donc de l'ODMG, est d'éviter le problème soulevé par les SGBD classiques où deux systèmes cohabitent lors de l'interfaçage avec un langage : celui du SGBD et celui du langage. Pour permettre une utilisation directe des types des langages objet, l'ODMG a choisi de définir un modèle abstrait de définition de bases de données objet, mis en œuvre par un langage appelé ODL (*Object Definition Language*). Ce modèle est ensuite adapté à un langage objet particulier : l'ODMG propose un standard d'intégration en C++, Smalltalk, et Java. Un langage d'interrogation pour ce modèle est proposé : il s'agit d'OQL (*Object Query Language*), pour beaucoup issu du langage de requête du

système O2 réalisé à l'INRIA [Bancilhon92, Adiba93]. OQL est aussi intégrable dans un langage de programmation objet.

Ce chapitre présente le standard de l'ODMG, en l'illustrant par des exemples. Après cette introduction, la section 2 précise le contexte et l'architecture d'un SGBDO conforme à l'ODMG. La section 3 développe le modèle abstrait et le langage ODL. La section 4 présente un exemple de base et de schéma en ODL. La section 5 aborde le langage OQL à travers des exemples et des syntaxes types de requêtes constituant des exemples génériques, appelés profils. La section 6 se consacre à l'intégration dans un langage de programmation ; le cas de Java est détaillé. La section 7 conclut ce chapitre en montrant les limites du standard de l'ODMG.

2. CONTEXTE

Dans cette section, nous présentons le contexte général du « standard » : les auteurs, le contenu de la proposition et l'architecture d'un système conforme à l'ODMG.

2.1 L' ODMG (*Object Database Management Group*)

Apparus vers 1986, les SGBD objet n'avaient pas connu le succès escompté cinq ans après leur naissance. Une des difficultés majeures provenait de l'absence de standards. Alors que les applications des bases de données relationnelles pouvaient prétendre à une très bonne portabilité assurée par le standard SQL, du poste de travail au calculateur central sur toute machine et tout système d'exploitation, les SGBD objet présentaient chacun une interface spécifique, avec des langages parfois exotiques. Un groupe de travail fut donc fondé en septembre 1991 à l'initiative de SUN par cinq constructeurs de SGBD objet : O2 Technology, Objectivity, Object Design, Ontos et Versant. Ce groupe prit rapidement le nom de ODMG (*Object Database Management Group*) et trouva un président neutre chez SUN en la personne de Rick Cattell, auteur de différent bancs d'essai sur les bases de données objet. Le groupe publia un premier livre intitulé *The Object Database Management Standard*, connu sous le nom ODMG'93. En fait, il ne s'agit pas d'un standard avalisé par les organismes de normalisation, mais plutôt d'une proposition d'un groupe de pression représentant des vendeurs de SGBDO.

Le groupe a continué à travailler et s'est enrichi de représentants de POET Software, UniSQL, IBEX et Gemstone Systems, ainsi que de multiples gourous et observateurs externes. Une nouvelle version du livre a été publiée en 1997, sous le titre **ODMG 2.0** ; elle comporte dix auteurs. Le groupe est maintenant bien établi et collabore avec l'OMG et l'ANSI, notamment sur l'intégration à CORBA et à SQL3. Les constructeurs participants s'engagent à suivre les spécifications de l'ODMG, malheureusement sans dates précises. Un des échecs majeurs du groupe est sans doute l'absence de systèmes conformes aux nouvelles spécifications. O2, qui est le système le plus proche, ne répond pas exactement aux fonctionnalités requises [Chaudhri98].

2.2 Contenu de la proposition

La proposition décrit les interfaces externes d'un SGBDO utilisées pour réaliser des applications. Le SGBDO se fonde sur une adaptation du modèle objet de l'OMG, modèle de référence étudié au

chapitre précédent. Il comporte un langage de définition des interfaces des objets persistants dérivés de l'IDL de l'OMG et appelé **ODL** (*Object Definition Language*).

> **Notion XII.1 : ODL (*Object Definition Language*)**
>
> Langage de définition de schéma des bases de données objet proposé par l'ODMG.

Une définition peut aussi être effectuée directement dans l'un des langages supportés. La partie la plus intéressante de la proposition est le langage **OQL** (*Object Query Language*).

> **Notion XII.2 : OQL (*Object Query Language*)**
>
> Langage d'interrogation de bases de données objets proposé par l'ODMG, basé sur des requêtes SELECT proches de celles de SQL.

Une intégration est proposée avec les langages objets C++, Smalltalk et Java. Celle-ci précise les conversions de types effectuées et permet la manipulation des objets gérés par le SGBD depuis le langage. Elle est appelée **OML** (*Object Manipulation Language*).

> **Notion XII.3 : OML (*Object Manipulation Language*)**
>
> Langage de manipulation intégré à un langage de programmation objet permettant la navigation, l'interrogation et la mise à jour de collections d'objets persistants, dont l'OMG propose trois variantes : OML C++, OML Smalltalk et OML Java.

La figure XII.1 illustre les différentes interfaces proposées par le standard ODMG. Ce sont celles permettant de réaliser des applications autour d'un SGBDO. Sont-elles suffisantes pour assurer la portabilité ? Probablement non, car les interfaces graphiques sont aussi importantes et souvent spécifiques du SGBDO. Quoi qu'il en soit, le respect de ces interfaces par les produits améliorerait de beaucoup la portabilité des applications.

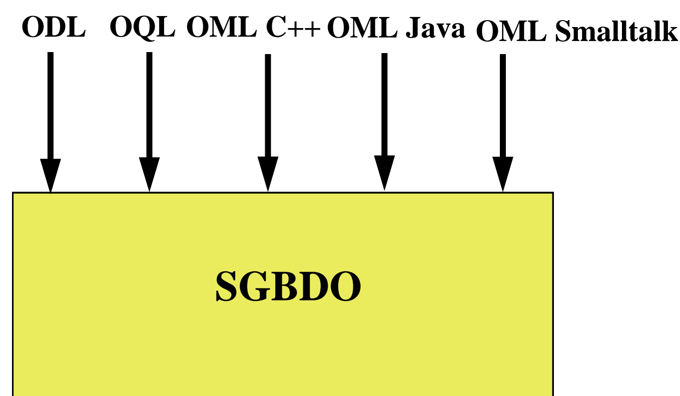

Figure XII.1 — Interfaces d'accès à un SGBDO

2.3 Architecture

La figure XII.2 illustre l'architecture typique d'un SGBDO conforme à l'ODMG. Autour d'un noyau gérant la persistance des objets, l'attribution des identifiants, les méthodes d'accès, et les aspects transactionnels, gravitent trois composants : le préprocesseur ODL permet de compiler les définitions d'objets et de générer les données de la métabase ; le composant langage OML spécifique à chaque langage de programmation permet de manipuler les objets conformes aux définitions depuis un langage de programmation tel C++, Smalltalk ou Java ; le composant OQL comporte un analyseur et un optimiseur du langage OQL capables de générer des plans d'exécution exécutables par le noyau. Au-dessus de ces trois composants, différents outils interactifs permettent une utilisation facile des bases ; ce sont par exemple un éditeur de classes pour éditer les schémas ODL, un manipulateur d'objets pour naviguer dans la base (les deux réunis constituent souvent le *browser*), une bibliothèque d'objets graphiques, des débogueurs et éditeurs pour les langages de programmation, etc.

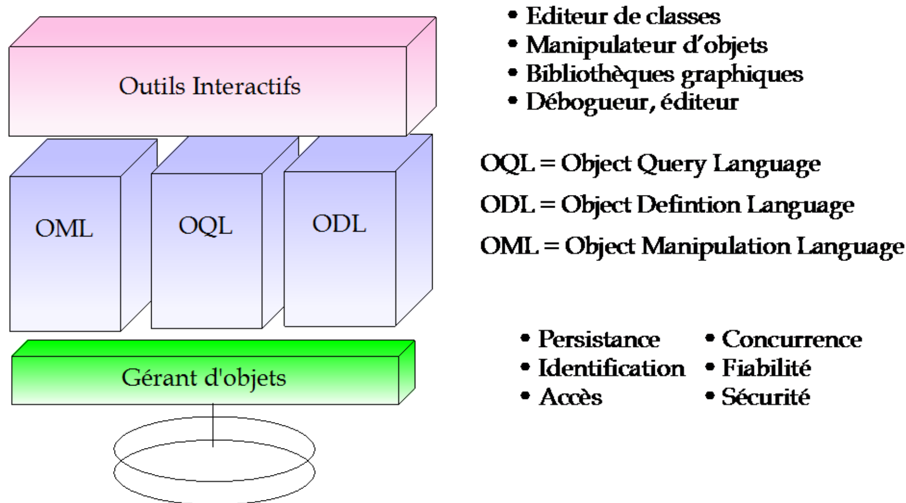

Figure XII.2 — Architecture type d'un SGBDO conforme à l' ODMG

Du côté interface avec les langages de programmation, le schéma préconisé est basé sur un système de type unique entre le langage et le SGBDO, chaque type du modèle objet du SGBDO (en principe celui de l'ODMG) étant traduit directement dans un type correspondant du langage. Un principe de base est aussi de ne nécessiter aucune modification du compilateur du langage. Les déclarations de classes persistantes peuvent être écrites en ODL, ou directement dans le langage de programmation (PL ODL). Un précompilateur permet de charger la métabase du SGBDO et de générer la définition pour le langage de programmation. Les programmes enrichis avec les définitions de classes d'objets persistants sont compilés normalement. Le binaire résultant est lié à la bibliothèque d'accès au SGBD lors de l'édition de liens, ce qui permet la génération d'un exécutable capable d'accéder à la base. La figure XII.3 illustre le processus d'obtention d'un exécutable.

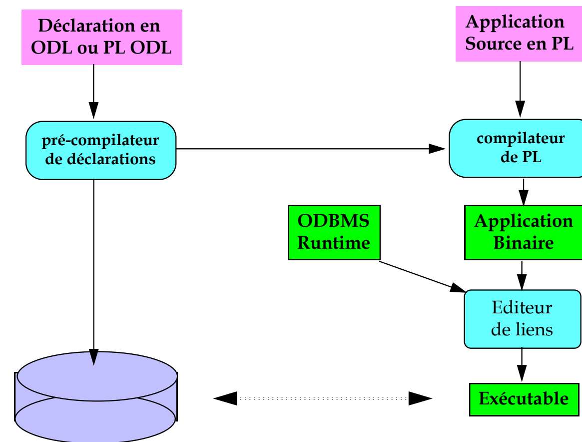

Figure XII.3 — Lien avec les langages de programmation

3. LE MODELE DE L'ODMG

Nous décrivons maintenant le modèle abstrait proposé pour la définition des schémas des bases de données objet.

3.1 Vue générale et concepts de base

L'OMG — organisme de normalisation de l'objet composé de plus de 800 membres, à ne pas confondre avec l'ODMG — a proposé un modèle standard pour les objets permettant de définir les interfaces visibles par les clients. Le modèle de l'ODMG est une extension du modèle de l'OMG et un candidat pour un profil BD de ce dernier. Il est mis en oeuvre à l'aide du langage ODL qui permet de spécifier les schémas de bases de données, alors que le modèle de l'OMG est supporté par le langage IDL (*Interface Definition Language*). Les bases de données objets nécessitent des adaptations ou extensions ; ODL se veut l'adaptation d'IDL aux bases de données.

Une extension principale est tout d'abord la nécessité de considérer un niveau d'abstraction permettant de manipuler des états abstraits pour les objets. Ce sont ces états qui sont mémorisés plus ou moins directement dans la base pour les objets persistants. En outre, les objets peuvent être groupés en collections beaucoup plus variées que les seules séquences de l'OMG. Ils peuvent aussi être associés par des associations. Tout cela conduit à un modèle et à un langage de définition associé (ODL) beaucoup plus complexe que celui de l'OMG. ODL reste cependant un langage de niveau conceptuel, supportant un modèle objet abstrait, qui peut être implémenté dans différents langages, en particulier C++, Smalltalk ou Java. Chaque construction ODL a donc une implémentation correspondante pour chacun de ces langages. Au contraire d'ODL, OML s'appuie sur une implémentation et est donc spécifique à un langage de programmation objet.

Le modèle de l'OMG comporte des types d'objets avec identifiants et des types de valeurs, ou littéraux. Outre les interfaces qui permettent de spécifier des comportements abstraits, ODL permet de spécifier des classes qui définissent, en plus d'un comportement abstrait, un état abstrait pour des objets d'un même type. On aboutit donc à trois types de définitions d'objets ou valeurs,

précisés ci-dessous ; c'est un peu complexe, voire confus, mais nécessaire pour permettre l'implémentation des spécifications de comportement ou d'état en C++, Smalltalk ou Java.

> **Notion XII.4 : Définition d'interface (*Interface definiton*)**
>
> Spécification du comportement observable par les utilisateurs (ou d'une partie de celui-là) pour un type d'objets.

Nous définissons par exemple figure XII.4 une interface `calculateur` modélisant une machine à calculer.

```
INTERFACE CALCULATEUR {
 CLEAR();
 FLOAT ADD(IN FLOAT OPERAND);
 FLOAT SUBSTRACT (IN FLOAT OPERAND);
 FLOAT DIVIDE(IN FLOAT DIVISOR);
 FLOAT MULTIPLY (IN FLOAT MULTIPLIER);
 FLOAT TOTAL();}
```

Figue 4 — Définition de l'interface d'un calculateur

> **Notion XII.5 : Définition de classe (*Class definition*)**
>
> Spécification du comportement et d'un état observables par les utilisateurs pour un type d'objets.

Une classe implémente ainsi une ou plusieurs interfaces. En plus d'un comportement, une classe définit un état abstrait. Pour mémoriser les états abstraits de ses instances, une classe possède aussi une **extension de classe**.

> **Notion XII.6 : Extension de classe (*class extent*)**
>
> Collection caractérisée par un nom contenant les objets créés dans la classe.

Le comportement abstrait pourra être hérité d'une interface. On voit donc qu'une classe donne une spécification d'une ou plusieurs interfaces en précisant quelques éléments d'implémentation, en particulier l'état des objets (abstrait car indépendant de tout langage) et l'extension qui va les contenir. Dans certains cas complexes, une classe peut d’ailleurs avoir plusieurs extensions. Il est aussi possible de préciser une clé au niveau d'une extension de classe : comme en relationnel, il s'agit d'un attribut dont la valeur détermine un objet unique dans l'extension. La figure XII.5 illustre une définition de classe incorporant l'interface calculateur.

```
CLASS ORDINATEUR (EXTENT ORDINATEURS KEY ID) : CALCULATEUR {
 ATTRIBUTE SHORT ID ;
 ATTRIBUTE FLOAT ACCUMULATEUR;
 VOID START();
 VOID STOP(); }.
```

Figure XII.5 — Un exemple de classe

Notez qu'une interface peut aussi comporter des attributs abstraits, mais ceux-ci sont vus comme des raccourcis d'opérations, une pour lire l'attribut, l'autre pour l'écrire. Une interface n'a en principe pas d'extension.

Interfaces et classes sont des spécifications de types. Il est aussi possible de spécifier des types de valeurs : ceux-ci sont appelés des littéraux.

> **Notion XII.7 : Définition de littéral (** ***Literal définition*****)**
>
> Spécification d'un type de valeur correspondant à un état abstrait, sans comportement.

Les littéraux correspondent aux types de base tels entier, réel, chaîne de caractères, mais aussi aux structures. Un exemple de littéral est un nombre complexe : struct complex {float re; float im}. Les littéraux sont directement implémentés comme des types de valeurs en C++. Dans les autres langages purs objets (Smalltalk ou Java), ce seront des objets.

En résumé, l'ODMG propose donc un système de types sophistiqué, capable d'être facilement mappé en C++, Smalltalk ou Java. Il y a donc des types, des interfaces, des classes et des littéraux. L'ensemble forme la hiérarchie de spécialisation représentée figure XII.6.

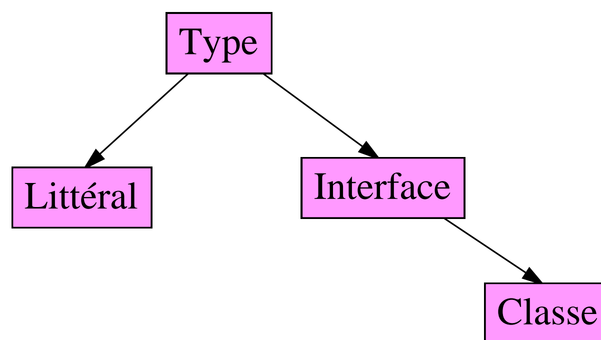

Figure XII.6 — Classification des définitions de type

3.2 Héritage de comportement et de structure

Comme vu ci-dessus, une classe peut hériter d'une interface : il s'agit d'un héritage de comportement. Toutes les opérations de l'interface seront alors définies pour les objets de la classe. Au niveau de l'implémentation, la classe fournira le code des opérations.

En plus de cette relation d'héritage de comportement d'interface vers une classe, l'ODMG propose un héritage de structure abstraite, c'est-à-dire de l'état des objets, cette fois de classe à classe. Cette relation d'héritage d'état est notée **EXTENDS** (à ne pas confondre avec **EXTENT** !). Par exemple, nous étendrons la classe ORDINATEUR comme indiqué figure XII.7.

Soulignons que l'héritage de comportement (noté :) peut être multiple : une classe peut implémenter plusieurs interfaces, mais elle ne peut dériver que d'une seule autre classe. En cas de conflits de noms, c'est à l'utilisateur qu'il incombe de distinguer les noms.

```
CLASS ORDINATEURAREGISTRE EXTENDS ORDINATEUR: CALCULATEUR {
 ATTRIBUTE FLOAT REGISTRE;
 VOID RTOA();
 VOID ATOR (); }.
```

Figure XII.7 — Exemple d'héritage de structure et de comportement

3.3 Les objets instances de classes

Les objets sont donc regroupés selon des types. Comme dans tout modèle objet, ils sont identifiés par des OID. Ceux-ci sont des chaînes binaires de format spécifique de chaque implémentation, gérés par le SGBD OO pour distinguer les objets. Un OID permet de retrouver l'objet qu'il identifie pendant toute sa durée de vie. Il reste donc invariant pendant la vie de l'objet.

Classiquement, les objets sont persistants ou transients : les objets persistants sont les objets de la base, les autres restant en mémoire. Les objets persistants peuvent être nommés : les noms sont donnés par les utilisateurs et sont uniques dans une base de données.

Les objets peuvent être atomiques, structurés, ou collections, c'est-à-dire composés de collections d'objets ou de littéraux. Les objets atomiques sont spécifiés par l'utilisateur par des définitions de classes comportant des attributs, des associations et des opérations. Les objets structurés sont prédéfinis et obéissent à des interfaces spécifiques.

3.4 Propriétés communes des objets

Les objets sont créés par une opération `new()` définie au niveau d'une interface ObjectFactory implémentée par le SGBDO (voir figure XII.8). Ils héritent d'un ensemble d'opérations implémentées par le SGBDO pour le verrouillage — qui peut être bloquant si l'objet est occupé (opération `LOCK`) ou non (opération `TRY_LOCK`) —, la comparaison d'identifiants, la copie d'objet avec génération d'un nouvel objet et la suppression (voir figure XII.8).

```
INTERFACE OBJECTFACTORY {
 OBJECT NEW(); // CREATION D'UN NOUVEL OBJET
};
INTERFACE OBJECT {
 ENUM LOCK_TYPE{READ, WRITE, UPGRADE}; // TYPES DE VERROUS
 EXCEPTION LOCKNOTGRANTED(); // ERREUR VERROU REFUSE
 VOID LOCK(IN LOCK_TYPE MODE) RAISES (LOCKNOTGRANTED); //VERROUILLAGE BLOQUANT
 BOOLEAN TRY_LOCK(IN LOCK_TYPE MODE);// VERROUILLAGE NON BLOQUANT
 BOOLEAN SAME_AS(IN OBJECT ANOBJECT); // COMPARAISON D'IDENTIFIANTS D'OBJETS
 OBJECT COPY(); // COPIE AVEC GENERATION D'UN NOUVEL OBJET
 VOID DELETE()}; // SUPPRESSION D'UN OBJET
```

Figure XII.8 — Interface commune des objets

3.5 Les objets structurés

Les objets structurés s'inspirent des types SQL2 utilisés pour la gestion du temps. L'intérêt de définir ces types comme des objets est de permettre de spécifier leur comportement sous forme d'interfaces. Des structures correspondantes sont aussi fournies dans les types de base proposés par l'ODMG. Les objets seuls offrent des opérations standard. Voici les types d'objets structurés supportés :

- `Date` représente un objet date par une structure (mois, jour, an) munie de toutes les opérations de manipulation classique des dates.

- `Interval` représente un objet durée par une structure (jour, heure, minute, seconde) munie de toutes les opérations de manipulation nécessaires, telles que l'addition, la soustraction, le produit, la division, les tests d'égalité, etc.

- `Time` représente les heures avec zones de temps, en différentiel par rapport au méridien de Greenwich ; l'unité est la milliseconde.

- `Timestamp` encapsule à la fois une date et un temps. Un objet `timestamp` permet donc une référence temporelle absolue en millisecondes.

3.6 Les collections

L'ODMG préconise le support de collections homogènes classiques de type SET<t>, BAG<t>, LIST <t> et ARRAY<t>. Une collection un peu moins classique est DICTIONARY<t,v>, qui est une collection de doublets <clé-valeur>. Toutes les collections héritent d'une interface commune `COLLECTION`, résumée figure XII.9. Celle-ci permet de créer des collections d'une taille initiale donnée par le biais d'une "ObjectFactory", puis de manipuler directement les collections pour récupérer leurs propriétés (taille, vide ou non, ordre ou non, doubles permis ou non, appartenance d'un élément), pour insérer ou supprimer un élément, pour parcourir la collection par le biais d'un itérateur mono- ou bidirectionnel.

```
INTERFACE COLLECTIONFACTORY : OBJECTFACTORY {
 COLLECTION NEW_OF_SIZE(IN LONG SIZE)
};
INTERFACE COLLECTION : OBJECT {
 EXCEPTION INVALIDCOLLECTIONTYPE(), ELEMENTNOTFOUND(ANY ELEMENT);
 UNSIGNED LONG CARDINALITY();
 BOOLEAN IS_EMPTY(),IS_ORDERED(),ALLOWS_DUPLICATES(),
    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);
 BIDIRECTIONALTERATOR CREATE_BIDIRECTIONAL_ITERATOR()RAISES(INVALIDCOLLECTIONTYPE);
};
```

Figure XII.9 — Interface commune aux collections

Comme son nom l'indique, un itérateur permet d'itérer sur les éléments. Pour cela, il fournit une interface résumée figure XII.10. Un itérateur bidirectionnel permet d'aller en avant, mais aussi en arrière.

```
INTERFACE ITERATOR {
 VOID RESET() ; // INITIALISATION AU DEBUT
 ANY GET_ELEMENT() RAISES(NOMOREELEMENTS); // OBTENTION ELEMENT
 VOID NEXT_POSITION RAISES(NOMOREELEMENTS); // AVANCE POSITION
 REPLACE_ELEMENT (IN ANY ELEMENT) RAISES(INVALIDCOLLECTIONTYPE) ;
...
};
```

Figure XII.10 — Eléments d'interface pour les itérateurs

Chaque collection possède en plus des interfaces spécifiques classiques. La figure XII.11 présente les opérations spécifiques aux dictionnaires. Il s'agit de la gestion de doublets <clé-valeur>, la clé étant en principe unique. Elle correspond à un mot d'entrée dans le dictionnaire, la valeur étant sa définition (ou son synonyme). L'utilisateur peut lier une clé à une valeur (ce qui revient à insérer ce doublet), délier la clé (la supprimer), rechercher la valeur associée à une clé, et tester si une clé figure dans le dictionnaire. Les dictionnaires peuvent permettre par exemple de gérer des répertoires de noms d'objets, etc.

```
INTERFACE DICTIONARY : COLLECTION
 EXCEPTION KEYNOTFOUND(ANY KEY);
 VOID BIND(IN ANY KEY, IN ANY VALUE); // INSERTION
 VOID UNBIND (IN ANY KEY)RAISE(KEYNOTFOUND); // SUPPRESSION
 ANY LOOKUP (IN ANY KEY)RAISE(KEYNOTFOUND); // RECHERCHE
 BOOLEAN CONTAINS_KEY(IN ANY KEY) ;
};
```

Figure XII.11 — Interface spécifique aux dictionnaires

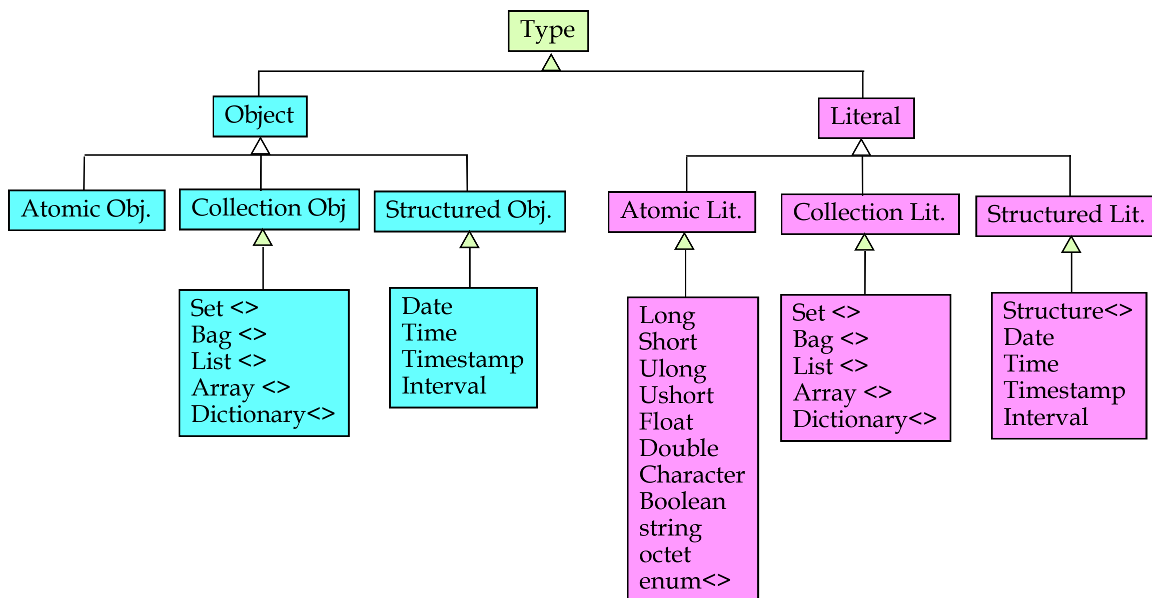

Figure XII.12 — La hiérarchie des types ODMG

3.7 Les attributs

Les attributs permettent de modéliser les états abstraits des objets. Un attribut est une propriété permettant de mémoriser un littéral ou un objet. Il peut être vu comme une définition abrégée de deux opérations : Set_value et Get_value. Un attribut possède un nom et un type qui précise ses valeurs légales. Il n'est pas forcément implémenté, mais peut être calculé.

3.8 Les associations (*Relationships*)

Les associations permettent de compléter la modélisation des états des objets. L'ODMG préconise le support d'associations binaires, bidirectionnelles de cardinalité (1:1), (1:N), ou (N:M), sans données. Une association de A vers B définit deux chemins de traversée inverses, A->B et B->A. Chaque chemin doit être défini en ODL au niveau du type d'objet source par une clause RELATIONSHIP. L'association pointe vers un seul objet cible ou vers une collection. Elle porte un nom et son inverse doit être déclaré. Pour la gestion, le SGBDO doit fournir des opérations sur

associations telles que Add_member, Remove_member, Traverse et Create_iterator_for. De fait, les associations sont simplement des déclarations abstraites d'attributs couplés, valués par une valeur simple ou une collection, contenant des identifiants d'objets réciproques. La figure XII.13 illustre l'association classique entre VINS et BUVEURS, mais sans données.

```
INTERFACE BUVEURS {
  ...
  RELATIONSHIP LIST<VINS> BOIRE INVERSE VINS::EST_BU_PAR;
  ...
};
INTERFACE VINS {
  ...
  RELATIONSHIP SET<BUVEURS> EST_BU_PAR INVERSE BUVEURS::BOIRE;
  ...
};
```

Figure XII.13 — Exemple de définition d'association

Lorsqu'une association est mise à jour, le SGBDO est responsable du maintien de l'intégrité : il doit insérer ou supprimer les références dans les deux sens si l'option inverse a été déclarée. Il ne doit pas y avoir de référence pointant sur des objets inexistants. Ce n'est pas le cas pour un attribut valué par un objet, d'où l'intérêt d'utiliser des associations plutôt que de définir directement les attributs supports. Par exemple, la déclaration au niveau des buveurs d'un attribut par la clause **ATTRIBUT** BOIRE **LIST**<VINS> n'implique pas l'existence d'un chemin inverse ni la gestion de l'intégrité référentielle par le SGBDO. Cependant, les associations de l'ODMG restent assez pauvres puisqu'elles ne peuvent avoir de données associées.

3.9 Les opérations

Classiquement, les opérations permettent de définir le comportement abstrait d'un type d'objets. Elles possèdent les propriétés habituelles des opérations sur objet : le nom de l'opération, le nom des conditions d'erreurs (après le mot clé **RAISES**), le nom et le type des arguments (après le mot clé **IN**), le nom et le type des paramètres retournés (après le mot clé **OUT**). L'ensemble constitue la signature de l'opération. La figure XII.14 illustre la définition de l'opération boire dans l'interface buveurs.

```
INTERFACE BUVEURS {
  ...
  INT BOIRE(IN VINS V, IN INT QTE) RAISES(NOVINS); //SIGNATURE DE L'OPERATION
  ...
};
```

Figure XII.14 — Un exemple de définition d'opération

3.10 Méta-modèle du modèle ODMG

Dans sa nouvelle version 2.0, l'ODMG spécifie un méta-modèle objet, c'est-à-dire un modèle objet décrivant son modèle. Ce modèle, plutôt complexe, permet de définir le schéma d'un référentiel objet capable de gérer des schémas de bases de données objet conformes à l'ODMG. Nous en donnons une vue très simplifiée figure XII.15. Il faudrait tout d'abord ajouter les concepts de littéraux, type et interface dont classe hérite. D'autres concepts sont nécessaires, comme MetaObject, Scope, Module, Exceptions. Tout cela est défini dans [ODMG97].

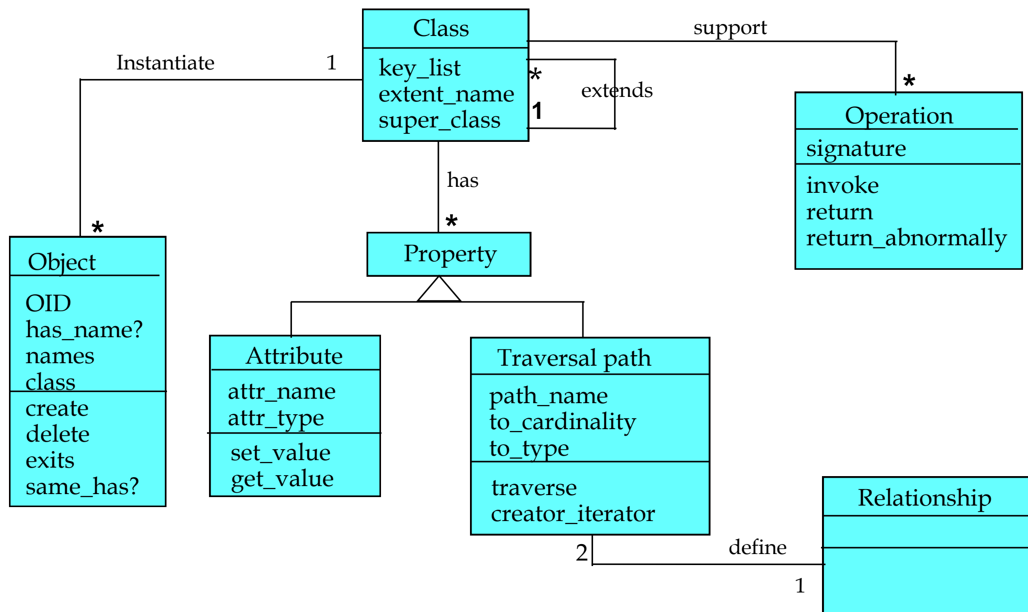

Figure XII.15 — Vue simplifiée du méta-modèle de l'ODMG

4. EXEMPLE DE SCHEMA ODL

A titre d'illustration, la figure XII.17 définit le schéma d'une base de données objet dont le modèle est représenté figure XII.16. La base décrit simplement des situations du monde réel : des personnes possèdent des voitures, et habitent dans des appartements. Parmi elles, certaines sont employées, d'autres sont buveurs. Parmi les employés il y a des buveurs. Les buveurs boivent des vins.

Une décision à prendre lors de la définition est de choisir entre interfaces et classes. Certaines classes peuvent avoir des extensions. Nous avons choisi d'implémenter les extensions de `Voiture`, `Vin`, `Appartement`, `Buveur` et `Employé`. `Personne` devient une interface. L'héritage multiple n'étant pas possible au niveau des classes, les employés buveurs sont des employés avec les propriétés de buveurs répétées. On aurait pu éviter cette répétition en définissant une interface `Buveurs`, puis une classe `CBuveurs` implémentant cette interface. Nous ne l'avons pas fait pour ne pas perturber le lecteur. Le bon choix est probablement de définir des interfaces pour tout ce qui est visible à l'extérieur, puis de réaliser ces interfaces par des classes.

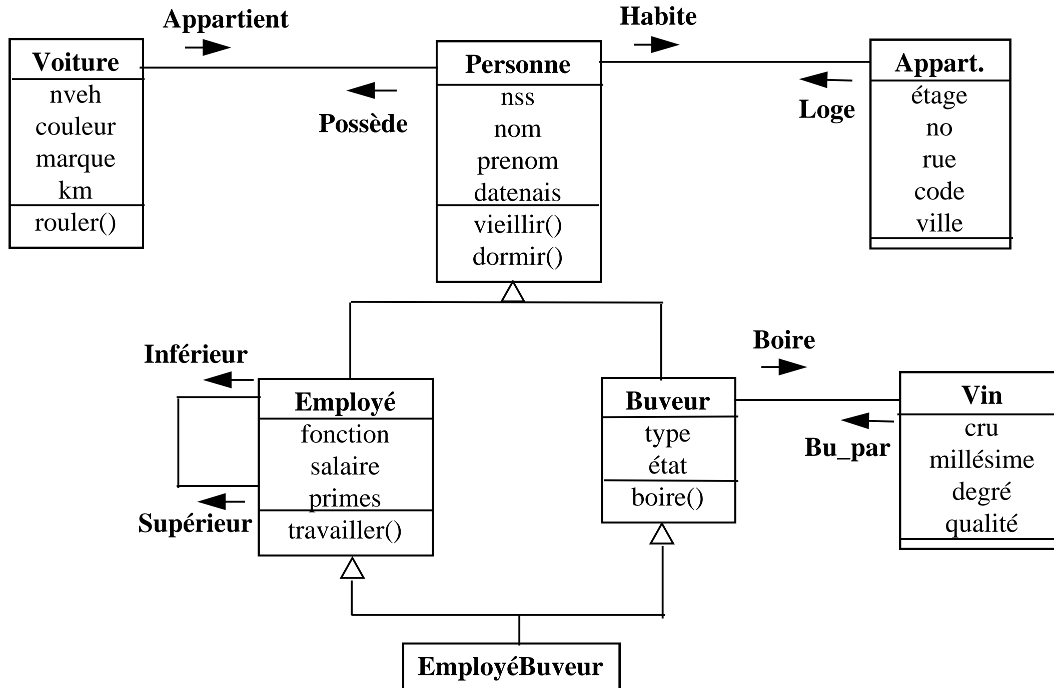

Figure XII.16 — Schéma graphique de la base exemple (en UML)

Class Voiture (**extent** voitures **key** nveh) **{** // classe avec extension

attribute string nveh;

attribute string couleur;

attribute string marque;

attribute short km;

relationship Personne Appartient **inverse** Personne::Possede;

short rouler(**in short** distance); **};**

```
Interface Personne { // interface abstraite pour implémentation dans classe
    attribute string nss ;
    attribute string nom ;
    attribute string prenom ;
    attribute date datenaissance;
    relationship Appart habite inverse Appart::loge; // relationship avec inverse
    relationship Voiture Possede inverse Voiture::Appartient;
    short vieillir();
    void dormir()
    short age(); };

class Employé : Personne(extent Employés key nss) { //classe avec extension
    attribute enum fonct{ingénieur, secrétaire, analyste, programmeur} fonction;
    attribute float salaire ;
    attribute list<float> primes ; //attribut multi-valué
    relationship Employé inferieur inverse supérieur;
    relationship Employé supérieur inverse inférieur;
    void travailler(); };

class Buveur : Personne(extent buveurs key nss) { // classe avec extension
    attribute typebuv{petit,moyen,gros} type;
    attribute etabuv{normal,ivre} etat;
    relationship list<Vin> boire inverse vin::bu_par;
    void boire(in Vin v); // paramètre d'entrée v };

class Appart (extent Apparts) { // classe avec extension
    attribute struct adresse (short etage, unsigned short numero, string rue,
    unsigned short code, string ville);
    relationship Set<personne> loge inverse Personne::habite; };
```

```
class Vin (extent Vins) { // classe avec extension
        attribute string cru;
        attribute string millesime;
        attribute string degre;
        attribute string qualite;
        relationship list<Buveur> bu_par inverse Vin::boire; };

class EmployéBuveur extends Employé { // classe sans extension hérite de employé
        attribute typebuv{petit,moyen,gros} type;
        attribute etabuv{normal,ivre} etat;
        relationship list<Vin> boire inverse vin::bu_par;
        void boire(in Vin v); // paramètre d'entrée v
};
```

Figure XII.17 — Schéma en ODL de la base exemple

5. LE LANGAGE OQL

Cette section détaille le langage de requêtes OQL et introduit des profils de requêtes typiques.

5.1 Vue générale

Le langage OQL a été défini à partir d'une première proposition issue du système O2 développé à l'INRIA. Les objectifs des auteurs étaient les suivants :

- permettre un accès facile à une base objet via un langage interactif autonome, mais aussi depuis un langage objet par intégration dans C++, Smalltalk, et plus tard Java ;
- offrir un accès non procédural pour permetttre des optimisations automatiques (ordonnancement, index,…) ;
- garder une syntaxe proche de SQL au moins pour les questions exprimables en SQL ;
- rester conforme au modèle de l'ODMG, en permettant l'interrogation de toutes les collections d'objets — extensions de classes ou autres collections imbriquées ou non —, le parcours d'association, l'invocation d'opérations, la manipulation d'identifiants, le support de l'héritage et du polymorphisme, etc.
- permettre de créer des résultats littéraux, objets, collections, …

- supporter des mises à jour limitées via les opérations sur objets, ce qui garantit le respect de l'encapsulation.

Ces objectifs sont-ils atteints ? Beaucoup le sont, mais il n'est pas sûr que le langage soit simple, au moins pour les habitués de SQL. La compatibilité avec SQL reste faible, la sémantique étant généralement différente, par exemple pour le traitement des valeurs nulles. Le langage est aussi faiblement assertionnel, en ce sens que les parcours de chemins doivent être explicitement exprimés. Pour le reste, le langage est très riche, plutôt élégant, et il existe une définition formelle, certes assez difficile à lire. La syntaxe est compacte.

Le langage est construit de manière fonctionnelle. Une question est une expression fonctionnelle qui s'applique sur un littéral, un objet ou une collection d'objets, et retourne un littéral, un objet ou une collection d'objets. Les expressions fonctionnelles peuvent être composées afin de constituer des requêtes plus complexes. Le langage est aussi fortement typé, chaque expression ayant un type qui peut être dérivé de la structure de l'expression et du schéma de la base. La correction de la composition des types doit être vérifiée par le compilateur OQL. Du point de vue du typage, OQL est peu permissif, et ne fait guère de conversions automatiques de types, à la différence de SQL. Syntaxiquement, beaucoup de questions sont correctes, mais erronées du point de vue typage. OQL n'est pas seulement un langage d'interrogation, mais bien un langage de requêtes avec mises à jour. En effet, il est possible de créer des objets et d'invoquer des méthodes mettant à jour la base.

OQL permet aussi la navigation via les objets liés de la base, mais seulement avec des expressions de chemins monovaluées.

Il est possible de naviguer par des **expressions de chemins** un peu particulière, réduites à des chemins simples, que l'on définira comme suit :

> **Notion XII.8 : Expression de chemin monovalué (*Monovalued path expression*)**
>
> Séquence d'attributs ou d’associations monovaluées de la forme X1.X2…Xn telle que chaque Xi à l'exception du dernier contient une référence à un objet ou un littéral unique sur lequel le suivant s'applique.

Par exemple, voiture.appartient.personne.habite.adresse.ville est une expression de chemin valide sur la base de données exemple.

Pour parcourir les associations multivaluées, OQL utilise la notion de **collection dépendante**.

> **Notion XII.9 : Collection dépendante (*Depending collection*)**
>
> Collection obtenue à partir d'un objet, soit parce qu'elle est imbriquée dans l'objet, soit parce qu’elle est pointée par l'objet.

Par exemple, chaque buveur référence par l'association Boire une liste de Vins. Si B est un buveur, B.Boire est une collection dépendante, ici une collection de Vins. Par des variables ainsi liées du

style B in Buveurs, V in B.Boire, OQL va permettre de parcourir les associations multivaluées. Ceci est puissant pour naviguer, mais plutôt procédural.

Pour présenter un peu plus précisément ce langage somme toute remarquable, nous procédons par des exemples un peu généralisés. Avant de les lire, il est bon de se rappeler que dans toute syntaxe une collection peut être remplacée par une requête produisant une collection. Réciproquement, il est possible de créer des collections intermédiaires et de remplacer chaque sous-requête par une collection intermédiaire. Cela peut permettre de simplifier des requêtes à nombreuses sous-requêtes imbriquées.

5.2 Exemples et syntaxes de requêtes

Nous présentons OQL à travers des exemples sur la base définie en ODL ci-dessus. Pour chaque exemple ou groupe d'exemples, nous abstrayons une syntaxe type dans un langage intuitif proche de la spécification de syntaxe en SQL : [a] signifie que a est optionnel, [a]* signifie une liste de 0 à N a séparés par des virgules, [a]+ de 1 à N a séparés par des virgules, {a|b} signifie un choix entre a ou b, et ... indique une syntaxe libre compatible avec celles déjà introduites. Cette syntaxe peut servir de profil (les *patterns* sont à la mode) pour formuler un type de requête. Nous appelons aussi l'expression syntaxique un profil, ou plus simplement un format. Chaque profil est précédé d'un nom de trois lettres suivi de ":" servant de référence pour un usage ultérieur éventuel. Nous donnons aussi le type du résultat inféré par le compilateur (après la requête, sous forme ===> type). Le langage étant complexe, nous ne donnons pas une définition, celle-ci pouvant être trouvée dans [ODMG97]. Il y en a d'ailleurs deux, une formelle, une autre en BNF, et elles ne semblent pas totalement cohérentes !

5.2.1 Calcul d'expressions

OQL permet de calculer des expressions arithmétiques de base, qui sont donc comme des questions.

```
(Q0) ((STRING) 10*5/2) || "TOTO"
```

===> string

Ceci donne en principe la chaîne "25TOTO". Cette requête montre à la fois le calcul d'expressions et les conversions de type. Son profil syntaxique est :

```
PEX: (<TYPE>) <EXPRESSION>
```

où l'expression peut être calculée avec tous les opérateurs classiques (+, *, -, /, mod, abs, concaténation). Plus généralement, l'expression peut aussi être une collection d'objets, et donc une requête produisant une telle collection, comme nous en verrons beaucoup ci-dessous.

5.2.2 Accès à partir d'objets nommés

OQL permet de formuler des questions sous forme d'expressions simples, en particulier construites à partir du nom d'un objet. Si MAVOITURE est le nom de l'objet désignant ma voiture, il est possible de demander :

```
(Q1) MAVOITURE.COULEUR
===> LITTERAL STRING
```

Plus généralement, OQL permet de naviguer dans la base en parcourant des chemins monovalués, comme vu ci-dessus. Par exemple, la question (Q2) retourne le nom du propriétaire de ma voiture (en principe, Gardarin !) :

```
(Q2) MAVOITURE.APPARTIENT.NOM
===> LITTERAL STRING
```

Nous appelerons de telles requêtes des **extractions d'objets**. Un profil correspondant à de telles questions est :

```
PEO: <NOM OBJET>.<CHEMIN>
```

où <CHEMIN> désigne une expression de chemins.

5.2.3 Sélection avec qualification

Nous retrouvons là les expressions de sélection du SQL de base. La notation de SQL pour la définition de variable derrière le FROM (du style BUVEURS AS B, ou BUVEURS B) peut d'ailleurs être utilisée. La requête suivante retrouve les noms et prénoms des gros buveurs :

```
(Q3) SELECT B.NOM, B.PRENOM
     FROM B IN BUVEURS
     WHERE B.TYPE = "GROS"
==> LITTERAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>>
```

La syntaxe générale de sélections simples est:

```
PSE: SELECT [<VARIABLE>.<ATTRIBUT>]+
     FROM <VARIABLE> IN <COLLECTION>
     WHERE <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <CONSTANTE>
```

5.2.4 Expression de jointures

OQL permet les jointures, tout comme SQL, avec une syntaxe similaire. Par exemple, la requête suivante liste les noms et prénoms des employés gros buveurs :

```
(Q4) SELECT B.NOM, B.PRENOM
     FROM B IN BUVEURS, E IN EMPLOYES
     WHERE B.NSS = E.NSS AND B.TYPE = "GROS"
==> LITTERAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>>
```

Le profil syntaxique d'une requête de sélection avec un ou plusieurs critères de sélection et une ou plusieurs jointures est :

```
(PSJ) SELECT [<VARIABLE>.<ATTRIBUT>]+
      FROM <VARIABLE> IN <COLLECTION>,[<VARIABLE> IN <COLLECTION>]+
      [WHERE <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <VARIABLE>.<ATTRIBUT>
      [AND <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <VARIABLE>.<ATTRIBUT>]*
      [{AND|OR} <VARIABLE>.<ATTRIBUT> <COMPARATEUR> <CONSTANTE>]]*
```

De manière générale, les qualifications peuvent faire intervenir, comme en SQL, des AND et des OR, et des expressions parenthésées contenant ces connecteurs logiques.

Une question équivalente peut utiliser le type EMPLOYEBUVEUR comme suit :

```
(Q5)  SELECT ((EMPLOYEBUVEUR)B).NOM, ((EMPLOYEBUVEUR)B).PRENOM
      FROM B IN BUVEURS
      WHERE B.TYPE = "GROS"
==> LITTERAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>
```

L'évaluateur doit alors vérifier à l'exécution l'appartenance des gros buveurs trouvés à la classe des EMPLOYEBUVEUR.

Le profil de telles questions avec indicateurs de classe est :

```
PIC:  SELECT [((<CLASSE>)<VARIABLE>).<ATTRIBUT>]+
      FROM ...
      [WHERE ...]
```

5.2.5 Parcours d'associations multivaluées par des collections dépendantes

Les collections dépendantes se parcourent par des variables imbriquées derrière le FROM. Ceci permet de traverser des chemins multivalués correspondant à des associations [1:N] ou [M:N]. Par exemple, la requête suivante retrouve le nom et le prénom des buveurs qui ont bu du Volnay :

```
(Q6)  SELECT B.NOM, B.PRENOM
      FROM B IN BUVEURS, V IN B.BOIRE
      WHERE V.CRU = "VOLNAY"
      ==> LITTERAL BAG<STRUCT<NOM:STRING,PRENOM:STRING>
```

Le profil syntaxique général d'une requête de parcours de collections dépendantes est :

```
PCD:  SELECT [<VARIABLE>.<ATTRIBUT>]+
      FROM ...,[<VARIABLE> IN <VARIABLE>.<ATTRIBUT>]+
      [WHERE ... ]
```

Bien sûr, la clause WHERE peut être composée de sélections, de jointures, etc.

5.2.6 Sélection d' une structure en résultat

Il est possible de sélectionner une structure en résultat ; c'est d'ailleurs l'option par défaut que nous avions utilisée jusque là, les noms des attributs étant directement ceux des attributs sources. La requête suivante permet de retrouver des doublets (Name, City) pour chaque gros buveur :

```
(Q7)  SELECT STRUCT (NAME: B.NOM, CITY: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.TYPE = 'GROS'
 ===> LITTERAL BAG <STRUCT(NAME,CITY)>
```

Par similarité avec SQL, la collection résultat est un BAG. Il est aussi possible d'obtenir une collection de type SET en résultat ; on utilise alors le mot clé DISTINCT, comme en SQL :

```
(Q8) SELECT DISTINCT (NAME: B.NOM, CITY: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.NOM = 'DUPONT'
 ===> LITTERAL SET <STRUCT(NAME, CITY)>
```

Le mot clé `STRUCT` étant implicite, le profil correspondant à ce type de requêtes est :

```
PST:  SELECT [DISTINCT][STRUCT]([<ATTRIBUT>: <VARIABLE>.<CHEMIN>]+)
      FROM ...
      [WHERE ...]
```

5.2.7 Calcul de collections en résultat

Plus généralement, le résultat d'une requête peut être une collection quelconque de littéraux ou d'objets. Par exemple, la requête suivante fournit une liste de structures :

```
(Q9) LIST(SELECT STRUCT(NOM: B.NOM, VILLE: B.HABITE.ADRESSE.VILLE)
      FROM B IN BUVEURS
      WHERE B.NOM = "DUPONT")
===> LITTERAL LIST <STRUCT(NOM, VILLE)>
```

Le profil syntaxique plus général est :

```
PCS:  <COLLECTION> (SELECT [DISTINCT]
                    [STRUCT] ([<ATTRIBUT>: <VARIABLE>.<CHEMIN>]+)
                    FROM ...
                    [WHERE ...])
```

5.2.8 Manipulation des identifiants d'objets

Les identifiants sont accessibles à l'utilisateur. Il est par exemple possible de retrouver des collections d'identifiants d'objets par des requêtes du type :

```
(Q10) ARRAY(SELECT V
      FROM V IN VOITURES
      WHERE B.MARQUE = "RENAULT")
===> LITTERAL ARRAY(<OID>)
```

dont le profil est :

```
POI: <COLLECTION> (SELECT <VARIABLE>
                   FROM ...
                   [WHERE ...])
```

5.2.9 Application de méthodes en qualification et en résultat

La question suivante sélectionne les employés ainsi que leur ville d'habitation et leur âge, dont le salaire est supérieur à 10000 et l'âge inférieur à 30 ans (age() est une méthode) :

```
(Q11) SELECT DISTINCT E.NOM, E.HABITE.ADRESSE.VILLE, E.AGE()
      FROM E IN EMPLOYES
      WHERE E.SALAIRE > 10000 AND E.AGE() < 30
===> LITTERAL DE TYPE SET <STRUCT>
```

Elle peut être généralisée au profil suivant :

```
PME: SELECT [DISTINCT]...,
            [<VARIABLE>.<CHEMIN>.<OPERATION>([<ARGUMENT>]*)]*
     FROM ...
     [WHERE ....
     [AND<VARIABLE>.<CHEMIN>.<OPERATION>([<ARGUMENT>]*) <COMPARATEUR>
     <CONSTANTE>]*]
```

5.2.10 Imbrication de SELECT en résultat

Afin de construire des structures imbriquées, OQL permet d'utiliser des SELECT imbriqués en résultat de SELECT. Par exemple, la question suivante calcule pour chaque employé une structure comportant son nom et la liste de ses inférieurs mieux payés :

```
(Q12) SELECT DISTINCT STRUCT (NOM : E.NOM, INF_MIEUX_PAYES :
             LIST (SELECT I
             FROM I IN E.INFERIEUR
             WHERE I.SALAIRE > E.SALAIRE))
      FROM E IN EMPLOYES
===>  LITTERAL DE TYPE SET <STRUCT (NOM: STRING, INF_MIEUX_PAYES : LIST
      <EMPLOYES>)>
```

Voici un profil possible pour une telle question :

```
PQI: SELECT [DISTINCT][STRUCT](...,[<ATTRIBUT>:<QUESTION>, ...]*)
     FROM ...
     [WHERE ...]
```

5.2.11 Création d'objets en résultat

Il est possible de créer des objets dans une classe par le biais de constructeurs ayant pour arguments des requêtes. Bien sûr, des constructeurs simples peuvent être appliqués pour insérer des objets dans des extensions de classes, par exemple la requête :

```
(Q13) EMPLOYE (NSS:15603300036029, NOM:"JEAN", SALAIRE:100000).
```

Plus généralement, il est possible d'employer tous les buveurs sans emploi par la requête suivante :

```
(Q14)  EMPLOYE( SELECT STRUCT(NSS : B.NSS, NOM: B.NOM, SALAIRE : 4000)
       FROM B IN BUVEURS
       WHERE NOT EXIST E IN EMPLOYES : E.NSS=B.NSS )
  ==> BAG<EMPLOYES> INSERE DANS EMPLOYES
```

Notez que cette requête utilise une quantification que nous allons expliciter plus loin. Le profil général de ces requêtes de création d'objets est donc :

```
PCO : <CLASSE>(<QUESTION>).
```

5.2.12 Quantification de variables

OQL propose des opérateurs de quantification universelle et existentielle directement utilisables pour composer des requêtes, donc des résultats ou des qualifications.

5.2.12.1 Quantificateur universel

La question suivante retourne vrai si tous les employés ont moins de 100 ans :

```
(Q15) FOR ALL P IN EMPLOYES : P.AGE < 100
```

Son profil est :

```
PQU: FOR ALL <VARIABLE> IN <QUESTION> : <QUESTION>
```

5.2.12.2 Quantificateur existentiel

La requête suivante retourne vrai s'il existe une voiture de marque Renault possédée par un buveur de plus de 60 ans :

```
(Q16) EXISTS V IN SELECT V
      FROM V IN VOITURES, B IN V.POSSEDE
      WHERE V.MARQUE = "RENAULT": B.AGE() > 60.
```

Un profil généralisé possible est :

```
PQE: EXISTS <VARIABLE> IN <QUESTION> : <QUESTION>
```

5.2.13 Calcul d'agrégats et opérateur GROUP BY

Dans la première version, les agrégats étaient calculables par utilisation d'un opérateur fonctionnel GROUP applicable sur toute requête. La version 2 est revenue à une syntaxe proche de celle de SQL pour des raisons de compatibilité. Les attributs à grouper sont généralement définis par une liste d'attributs, alors que le critère de groupement peut être explicitement définies par des prédicats, ou implicitement par une liste d'attributs. Dans le dernier cas, le mot clé PARTITION peut être utilisé pour désigner chaque collection résultant du partitionnement. Les deux requêtes suivantes illustrent ces possibilités :

```
(Q17) SELECT E
      FROM E IN EMPLOYES
      GROUP BY (BAS : E.SALAIRE < 7000,
                MOYEN : E.SALAIRE ≤ 7000 AND E.SALAIRE < 21000,
                HAUT : E.SALAIRE ≥ 21000)
===> STRUCT<BAS: SET(EMP.), MOYEN:SET(EMP.),HAUT:SET(EMP.)>


(Q18) SELECT STRUCT(VILLE: E.HABITE.ADRESSE.VILLE, MOY : AVG(SALAIRE))
      FROM E IN EMPLOYES
      GROUP BY E.HABITE.ADRESSE.VILLE
     HAVING AVG (SELECT E.SALAIRE FROM E IN PARTITION)> 5000
 ===> BAG <STRUCT (VILLE: STRING, MOY: FLOAT)>


PAG:  SELECT ...[<AGGREGAT>(<ATTRIBUT>)]*
      FROM ...
      [WHERE ...]
      GROUP BY {<ATTRIBUT>+ | <PREDICAT>+}
      HAVING <PREDICAT>
```

AGGREGAT désigne une fonction d'agrégats choisie parmi COUNT, SUM, MIN, MAX et AVG. PREDICAT désigne un prédicat expression logique de prédicats simples de la forme

<ATTRIBUT> <COMPARATEUR> <CONSTANTE> ou <AGGREGAT>(PARTITION) <COMPARATEUR> <CONSTANTE>. On voit donc que OQL généralise les agrégats de SQL2 en permettant de grouper par prédicats explicites. C'est aussi possible en SQL3 avec d'autres constructions.

5.2.14 Expressions de collections

Il est possible de manipuler directement les collections en appliquant des opérateurs spécifiques. Les résultats de requêtes sont souvent des collections, si bien que les opérateurs peuvent être appliqués aux requêtes. Voici quelques exemples d'opérateurs.

5.2.14.1 Tris

La première syntaxe était fonctionnelle. La nouvelle est copiée sur SQL, comme suit :

```
(Q19) SELECT E.NOM, E.SALAIRE
      FROM E IN EMPLOYES
      WHERE E.SALAIRE > 21000
      ORDER  BY DESC E.SALAIRE
```

Le profil est donc analogue à celui de SQL :

```
PTR: <COLLECTION>
     ORDER  BY {DESC|ASC} <ATTRIBUT>+
```

5.2.14.2 Union, intersection, différence

Comme en SQL, union, intersection et différence de questions sont possibles. Par exemple :

```
(Q20) EMPLOYES UNION BUVEURS
```

est une requête valide.

Le profil général de ce type de requête est :

```
PEN: <COLLECTION> {INTERSECT|UNION|EXCEPT} <COLLECTION>
```

Les priorités sont selon l'ordre indiqué, les parenthèses étant aussi possibles.

5.2.14.3 Accesseurs aux collections

Il est possible d'extraire un élément d'une collection en utilisant une fonction d'accès générale, comme `first` ou `last`, ou plus généralement des fonctions d'accès spécifiques au type de collection. Les constructeurs de collections `struct`, `set`, `list`, `bag`, `array` sont aussi utilisables pour construire des collections comme nous l'avons déjà vu. Ainsi :

```
(Q21)  SELECT B.NOM, FIRST(B.BOIRE)
       FROM B IN BUVEURS
```

sélectionne le premier vin bu par un buveur.

Plus généralement :

```
PFC: FIRST(<COLLECTION>) |
     LAST(<COLLECTION>) |
     <FONCTION>(<COLLECTION>)
```

sont des profils admissibles.

5.2.15 Conversions de types appliquées aux collections

Comme nous l'avons vu, le résultat d'une question est souvent une collection. Réciproquement, une collection correctement spécifiée est une question dont la réponse est la liste de ses éléments. Tout opérateur applicable à une ou plusieurs collections l'est aussi à une ou plusieurs questions. C'est particulièrement le cas des opérateurs ci-dessus.

Comme il existe différents types de collections, il est possible d'appliquer des opérateurs de conversion, par exemple LISTOSET qui traduit une liste en un ensemble et DISTINCT qui traduit toute collection en un ensemble. De plus, il est possible de transformer une collection ayant un seul élément en un singleton par l'opérateur ELEMENT. Par exemple, la question suivante retrouve la marque de la voiture de numéro 120 ABC 75 :

```
(Q22) ELEMENT (SELECT V.MARQUE
      FROM V IN VOITURES
      WHERE V.NUMERO = "120 ABC 75")
 ===> STRING
```

Les collections apparaissant en résultat de requêtes sont parfois imbriquées. Les collections peuvent être aplaties à l'aide de l'opérateur FLATTEN. Par exemple :

```
(Q23) FLATTEN (SELECT B.NOM, SELECT V.MILLESIME
                             FROM V IN B.BOIRE
                             WHERE V.CRU = "VOLNAY"
      FROM B IN BUVEURS)
```

sélectionne simplement des doublets nom de buveurs et millésime de Volnay pour les buveurs ayant bu du Volnay.

En général, les profils permis sont donc :

```
PCO: LISTTOSET(<COLLECTION>) |
     ELEMENT(<COLLECTION>) |
     DISTINCT(<COLLECTION>) |
     FLATTEN(<COLLECTION>)
```

5.2.16 Définition d'objets via requêtes

Finalement, OQL permet aussi de nommer le résultat d'une requête. Celui-ci apparaît alors comme un littéral ou un objet nommé (selon le type du résultat). Par exemple, la requête suivante obtient un ensemble d'objets buveurs nommé IVROGNE :

```
(Q24) DEFINE IVROGNES AS
      SELECT DISTINCT B
      FROM B IN BUVEURS
      WHERE B.TYPE = "GROS"
```

Le profil général est simplement :

```
PDE:  DEFINE <IDENTIFIANT> AS <QUESTION>.
```

5.3 Bilan sur OQL

OQL est un langage de requête d'essence fonctionnelle très puissant. Nous n'avons donné que quelques exemples de possibilités avec une syntaxe approximative. En fait, OQL est un langage hybride issu d'un mélange d'un langage fonctionnel et d'un langage assertionnel comme SQL. D'ailleurs, SQL est lui-même un mélange d'un langage d'opérateurs (ceux de l'algèbre relationnelle) et d'un langage logique purement assertionnel (QUEL était proche d'un tel langage). Tout cela devient très complexe et la sémantique est parfois peu claire, quoi qu'en disent les auteurs. Pour être juste, disons que SQL3, décrit au chapitre suivant, souffre des mêmes défauts.

6. OML : L'INTEGRATION A C++, SMALLTALK ET JAVA

6.1 Principes de base

Le modèle de l'ODMG est abstrait, c'est-à-dire indépendant de toute implémentation. Pour l'implémenter dans un langage objet spécifique, il est nécessaire d'adapter le modèle. Par exemple, il faut préciser ce que devient un littéral dans un langage pur objet tel Smalltalk, ce que devient une interface en C++, ou une association en Java. Les types de données ODMG doivent être implémentés avec ceux du langage pour conserver un système de type unique. Il faut aussi pouvoir invoquer les requêtes OQL depuis le langage et récupérer les résultats dans des types intégrés au langage, par exemple comme des objets du langage.

L'ODMG propose trois intégrations de son modèle et de son langage de requêtes, une pour C++, une autre pour Smalltalk, une enfin pour Java. Autrement dit, la vision de l'usager d'une BD conforme aux spécifications de l'ODMG est précisée pour ces trois langages objet. Les objectifs essentiels de ces propositions sont :

1. La **fourniture d'un système de types unique à l'utilisateur**. En principe, le système de types utilisé est celui du langage de programmation parfois étendu, par exemple avec des collections. En effet, les vendeurs de SGBDO ayant longtemps décrié les SGBD relationnels pour l'hétérogénéité des types des langages de programmation et de SQL, ils se devaient d'éviter cet écueil.

2. Le **respect du langage de programmation**. Cela signifie que ni la syntaxe ni la sémantique du langage ne peuvent être modifiées pour accommoder la manipulation des données. Des classes supplémentaires seront généralement fournies, mais le langage de base ne sera pas modifié.

3. Le **respect du standard abstrait ODMG.** Ceci signifie que l'intégration proposée doit donner accès à toutes les facilités abstraites du modèle ODMG et de son langage de requêtes OQL. Ce principe n'est que partiellement suivi dans l'état actuel du standard, certaines facilités n'étant pas supportées, par exemple les associations en Java. Mais ceci devrait être le cas dans une future version.

Pour illustrer ces principes, nous traitons ci-dessous de l'intégration à Java, celles avec les autres étant similaires, mais un peu plus complexes. Auparavant, nous allons discuter des interfaces générales de gestion des bases de données et des transactions qui doivent être fournies.

6.2 Contexte transactionnel

L'accès aux bases de données s'effectue dans un contexte transactionnel. Le langage doit donc fournir des transactions implémentées par le SGBDO. Plus précisément, une transaction est un objet créé par une `TransactionFactory` qui implémente l'interface comportant les opérations suivantes :

- `begin()` pour ouvrir une transaction ;
- `commit()` pour valider les mises à jour de la transaction, relâcher les verrous obtenus et terminer la transaction avec succès ;
- `abort()` pour défaire les mises à jour de la transaction, relâcher les verrous obtenus et terminer la transaction avec échec ;
- `checkpoint()` qui a le même effet que `commit()` suivi de `begin()`, mais ne relâche pas les verrous ;
- `join()` pour récupérer l'objet transaction récepteur sous contrôle du contexte d'exécution (*thread* courante) ;
- `leave()` pour dissocier un objet transaction du contexte d'exécution courant ;
- `isOpen()` pour tester si la transaction est ouverte.

Ce type d'opérations sur transactions est commun, à l'exception de `join()` et `leave()` prévus pour permettre à un contexte d'exécution de gérer plusieurs transactions, ou à l'inverse pour permettre à une transaction de traverser plusieurs contextes d'exécution.

Avant d'exécuter des opérations au sein de transactions, un utilisateur doit ouvrir un objet bases de données. Plus généralement, l'implémentation doit fournir des objets `Database` (créés à partir d'une `DatabaseFactory`) qui supportent les opérations suivantes :

- `open(in string nom-de-base)` pour ouvrir une base de nom donné afin d'y accéder via le langage en transactionnel ;

- `close()` pour fermer une base ;
- `bind(in any unObjet, in string nom)` pour donner un nom à l'objet passé en paramètre; c'est l'opération de base pour nommer un objet dans une base de données, le nom ne devant pas déjà exister dans la base; un même objet peut avoir plusieurs noms ;
- `Object unbind(in string nom)` pour enlever un nom à un objet précédemment nommé ;
- `Object lookup(in string nom)` pour retrouver et charger comme un objet du langage l'objet de la base de nom donné.

L'accès depuis un langage de programmation objet à une base ODMG nécessite donc l'implémentation de ces interfaces dans le langage objet. Ceci est simple puisque le SGBDO doit supporter ces opérations, les interfaces *`Transaction`* et *`Database`* étant partie intégrante du standard ODMG.

6.3 L'exemple de Java

Java est un langage objet à la mode, plus simple et plus sûr que C++, idéal pour réaliser des applications distribuées robustes. Java comporte des littéraux de base (entier, réel, string, etc.) et des objets instances de classes. Il ne supporte pas les associations autrement que par des références inter-classes. Tout ceci en fait un langage de choix pour développer des applications bases de données. Nous examinons brièvement les différents aspects de Java OML ci-dessous.

6.3.1 La persistance des objets

Pour pouvoir être persistant, un objet doit être instance d'une classe connue du SGBDO. Plusieurs moyens sont possibles pour faire connaître une classe Java au SGBDO : définir la classe en ODL et écrire un compilateur ODL générant des définitions de classes Java, avoir un préprocesseur reconnaissant les classes persistantes ou ajouter une interface spécifique de persistance à ces classes, etc. Le standard actuel ne se prononce pas sur le moyen de déclarer une classe Java persistante. De telles classes sont supposées existantes et connues du SGBDO. Elles peuvent alors contenir des objets persistants, mais aussi toujours des objets transients.

Alors, comment rendre un objet (d'une classe persistante) persistant ? Le mécanisme retenu est la **persistance par atteignabilité**. Les racines de persistance sont les objets nommés au moyen de l'opération `bind` de l'interface base de données vue ci-dessus. Tout objet référencé par un objet persistant est persistant. Tout objet non référencé par un objet persistant doit être détruit automatiquement de la base, ce qui sous-entend l'existence d'un ramasse-miettes sur disques. Au-delà, le standard propose que les objets persistants puissent conserver des attributs non persistants, ce qui n'est sans doute pas simple à déclarer sans modifier Java.

6.3.2 Les correspondances de types

Les types de l'ODMG sont traduits dans des types Java. Les littéraux long, short, float, double, boolean, octet, char et string sont traduits en type primitif Java correspondant de même nom, sauf long qui devient int et octet qui devient byte. Java propose pour chaque type primitif une représentation valeur et une représentation objet par des instances de la classe de même nom : les deux traductions sont possibles. Les types structurés date, time et timestamp sont traduits comme des objets définis dans le package Java.sql de JDBC, le standard d'accès aux bases relationnelles.

6.3.3 Les collections

L'interface Collection et ses extensions Set, Bag, List et Array sont implémentées en Java avec des opérations similaires, mais harmonisées avec Java. Java supporte le type de collection Array de plusieurs manières : les Array sont des tableaux monodimensionnels de taille fixe intégré au langage, Vector est une classe standard supportant des tableaux dynamiques. Il est possible d'utiliser Array ou Vector pour implémenter une classe Varray conforme au standard ODMG.

6.3.4 La transparence des opérations

Toute opération exécutable en Java sur des objets transients l'est également sur des objets persistants. C'est au SGBDO d'assurer la transparence à la persistance, par exemple de gérer les références entre objets persistants, de lire les objets persistants référencés non présents en mémoire, et de reporter l'état des objets modifiés dans la base à la validation de transaction. Les techniques à mettre en œuvre sont spécifiques à chaque SGBD, mais doivent être transparentes à l'utilisateur.

6.3.5 Java OQL

L'intégration d'OQL s'effectue de deux manières. Tout d'abord l'ajout d'une opération Query(String predicate) permet de filtrer les objets d'une collection avec un prédicat OQL. En supposant que Vins désigne une collection d'objets persistants ou non, on pourra par exemple retrouver les bons vins par la construction suivante :

```
SET<OBJECT> LESBONSVINS;
LESBONSVINS = VINS.QUERY("QUALITE = "BONS"").
```

De manière plus complète, une classe OQLQuery est proposée afin de composer des requêtes paramétrées OQL sous forme d'objets, de demander leur exécution et de récupérer les résultats dans des objets Java, en général des collections. La classe OQLQuery comporte plus précisément les opérations :

- OQLQuery(String question) pour créer des requêtes avec le texte paramétré (les paramètres sont notés $i) en OQL passé en argument ;

- OQLQuery() pour créer des requêtes génériques, le texte étant passé ensuite par l'opération create(String query) ;

- `bind(Object parameter)` pour lier le premier paramètre non encore lié de la requête ;

- `execute()` pour exécuter une requête préalablement construite et récupérer un objet Java de type collection ou simple en résultat.

Voici une illustration simple de cette classe pour retrouver les buveurs d'un cru quelconque, par exemple de Volnay ; Java ne supportant pas les associations, on supposera que l'attribut Boire de buveurs est implémenté comme une collection de vins :

```
SET BUVEURSCRU;
QUERY = OQLQUERY( "SELECT DISTINCT B FROM B IN BUVEURS,
                  V IN B.BOIRE WHERE V.CRU = $1" );
QUERY.BIND("VOLNAY");
BUVEURSCRU = QUERY.EXECUTE();
```

6.3.6 Bilan

Java OML propose une vision simple d'une base de données ODMG en Java. La première version de cette proposition est assez brièvement décrite dans le standard. Nous en avons donné un aperçu ci-dessus. La clé du problème semble être dans le support des collections en Java, qui est loin d'être standard. Le problème de l'ODMG pourrait être à terme de faire coïncider son standard avec le standard Java, notamment pour les collections. Sinon, il y aurait vraiment opposition de phase (*impedance mismatch*), ce qui serait paradoxal ! Ceci ne serait cependant pas étonnant, car peut-on construire deux standards (un pour les bases de données, l'autre pour le langage) indépendamment et conserver un système de type unique ? Une meilleure approche est probablement de développer un langage de programmation basé sur les types du SGBD pour éviter les problèmes de conversion de types. C'est ce que propose SQL3, comme nous le verrons dans le chapitre qui suit.

7. CONCLUSION

Depuis 1991, les vendeurs de SGBDO tentent de créer un standard afin d'assurer la portabilité des programmes utilisateurs, donc des applications. Cette démarche est importante car elle seule peut garantir la pérennité de l'approche bases de données objet pour les utilisateurs. Deux versions de ce « standard » ont été publiées, l'une en 1993, l'autre en 1997. Nous nous sommes basés sur la version 1997 (ODMG 2.0) pour écrire ce chapitre. Bien que remarquable, la version 2 est quelque peu complexe et n'est guère supportée par les systèmes actuels (ni d'ailleurs celle de 1993). Aussi, elle est parfois inégale dans le niveau de détails fourni. Elle se compose d'un modèle abstrait de bases de données objet et d'interfaces concrètes pour chacun des langages supportés.

En outre, le langage de requêtes OQL constitue une contribution importante de cette proposition. OQL offre les fonctionnalités du SQL de base avec en plus des extensions pour manipuler les expressions de chemins, les méthodes et surtout les collections. C'est un langage très puissant,

mais complexe, qui a aussi quelques déficiences, comme l'absence de contrôle de droits et de gestion de vues. Il peut être intégré à C++, Smalltalk et Java de manière standard. Tout ceci est positif.

Alors, peut-on enfin parler de portabilité des applications des SGBDO, par exemple écrites en Java ? Il y a pour l'instant peu de retour d'expériences. De plus, les vendeurs de SGBDO, à l'exception sans doute de O2, n'implémentent pas OQL et ne se soucient finalement guère plus que dans le discours du standard ODMG. Pour garantir une certaine portabilité, il faut donc utiliser du pur Java, sans requête. C'est encore faible. Mais que les adeptes des langages objets, et particulièrement les défenseurs de Java se rassurent : il n'y a pas besoin de SGBDO conforme à l'ODMG pour faire persister et retrouver des objets Java et garantir la portabilité des applications. JDBC, développé par SunSoft, est un package vraiment standard pour faire persister et retrouver via SQL des objets Java dans toute base de données, y compris d'ailleurs certaines bases de données objet. Vous pouvez aussi regarder du côté de Persistant Java et d’interfaces plus spécifiques comme JSQL.

8. BIBLIOGRAPHIE

[Adiba93] M. Adiba, C. Collet, *Objets et Bases de Données : Le SGBD O2*, Hermès, Paris, 1993.

Cet ouvrage de 542 pages décrit la génération des SGBD objet et détaille plus particulièrement le SGBD O2. Il décrit tout le cycle d’utilisation de ce SGBD et les principaux composants du système.

[Bancilhon89] F. Bancilhon, S. Cluet, C. Delobel, « A Query Language for an Object-Oriented Database System », *In 2nd Intl. Workshop on Database Programming Language*, DBPL, pp. 301-322, 1989.

Cet article introduit la première version du langage de requêtes du système O2, précurseur du langage OQL. Cette version souligne les aspects fonctionnels du langage.

[Bancilhon92] F. Bancilhon, C. Delobel, P. Kanellakis, *Building an Object-Oriented Database System : The Story of O2*, Morgan Kaufmann Pub., San Fransisco, CA, 1992.

Ce livre est une collection d’articles présentant l’histoire, la conception et l’implémentation du système O2, un des premiers SGBD pur objet.

[Chaudhri98] A.B. Chaudhri, « Workshop Report on Experiences Using Object Data Management in the Real-World », *SIGMOD Record*, V.27, N°1, pp. 5-10, Mars 1998.

Cet article résume les présentations effectuées à un Worshop portant sur les applications réelles des BD objet. Il discute en particulier de quelques insuffisances d’ObjectStore et des multiples différences entre O2 et le standard ODMG.

[Ferrandina95] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, J. Madec, « Schema and Database Evolution in the O2 Object Database System », *Proc. of 21st Intl. Conf. On Very Large Databases*, Zurich, pp. 170-181, 1995.

Cet article souligne la difficulté d'évolution des schémas ODL dans une base de données objet. Il décrit les algorithmes implémentés dans O2 pour faire évoluer la base en conformité au schéma.

[Fishman88] Fishman *et. al.*, *Overview of the IRIS DBMS*, Hewlett-Packard Internal Report, Palo Alto, 1988.

Cet article présente le système IRIS et son langage OSQL. OSQL est une version fonctionnelle d'un SQL étendu aux objets. OSQL est un langage qui permet de définir des types abstraits comme des ensembles de fonctions, de créer des classes d'objets accessibles par des fonctions, puis d'appliquer ces fonctions en les imbriquant à partir d'arguments éventuellement instanciés. Il s'agit d'un des premiers SQL étendus aux objets basé sur une approche fonctionnelle, donc un précurseur de OQL.

[ODMG93] R. Cattell (Ed.), *Object Databases : The ODMG-93 Standard*, Morgan-Kaufman, San Matéo, CA, 1993.

Ce livre présente la première version du standard ODMG pour les bases de données. Il décrit le modèle objet, le langage de définition ODL, le langage de requêtes OQL et les interfaces avec C++ et Smalltalk.

[ODMG97] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gammerman, D. Jordan, A. Springer, H. Strickland, D. Wade, *Object Database Standard : ODMG 2.0,* Morgan Kaufmann Pub., San Fransisco, CA, 1997.

Ce livre présente la deuxième version du standard ODMG décrit dans ce chapitre. Il inclut tous les aspects étudiés et donne le détail des spécifications de ODL, OQL, OML C++, Smalltalk et Java.

[Won Kim95] Won Kim, *Modern Database Systems*, Addison-Wesley, New-York, 1995.

Cette collection d'articles sur les BD objets aborde tous les aspects d'un SGBD objet, en particulier les interfaces avec les langages objet (OQL-C++) ainsi que les promesses et déceptions des systèmes purs objet.