LE MODELE OBJET

1. INTRODUCTION

Les modèles à objets, encore appelés modèles orientés objets ou simplement modèles objet, sont issus des réseaux sémantiques et des langages de programmation orientés objets. Ils regroupent les concepts essentiels pour modéliser de manière progressive des objets complexes encapsulés par des opérations de manipulation associées. Ils visent à permettre la réutilisation de structures et d'opérations pour construire des entités plus complexes. Ci-dessous, nous définissons les concepts qui nous semblent importants dans les modèles de données à objets. Ces concepts sont ceux retenus par l'OMG — l'organisme de normalisation de l'objet en général —, dans son modèle objet de référence [OMG91]. Certains sont obligatoires, d'autres optionnels. Ce modèle est proche du modèle de classe de C++ [Stroustrup86, Lippman91], qui peut être vu comme une implémentation des types abstraits de données. Il est aussi très proche du modèle objet du langage Java [Arnold96].

Bien que C++ et Java soient aujourd'hui les langages de programmation d'applications sur lesquels s'appuient la plupart des bases de données à objets, il existe d'autres langages orientés objet. Historiquement, Simula a été le premier langage orienté objet ; il est toujours un peu utilisé. Simula a introduit le concept de classe qui regroupe au sein d'une même entité la structure de données et les fonctions de services qui la gèrent. Simula avait été développé pour la simulation et disposait notamment d'outils génériques dont un échéancier intéressants. Smalltalk [Goldeberg83] est né à la fin des années 70, en reprenant des concepts de Simula. C'est un langage orienté objet populaire, souvent interprété et disposant d'environnements interactifs intéressants. Dans le monde des bases de données, il a été utilisé par les concepteurs de GemStone [Maier86] comme langage de base pour définir les structures d'objets et programmer les applications.

Divers dialectes Lisp sont orientés objet, si bien que l'approche bases de données à objets est née à partir de Lisp. Orion [WonKim88] fut historiquement le premier

SGBD à objets construit à partir d'un Lisp objet. Aujourd'hui, et malgré quelques détours vers des langages spécifiques [Lécluse89], la plupart des SGBD à objets sont basés sur C++ et s'orientent de plus en plus vers Java. Ils permettent de gérer des classes d'objets persistants. Ce choix est effectué essentiellement pour des raisons de performance et de popularité du langage C++, qui est une extension du langage C ; C++ est d'ailleurs traduit en C par un pré-compilateur. Quelques SGBDO supportent Smalltalk. Java est le langage objet d'avenir, supporté par la plupart des SGBDO. La très grande portabilité et la sécurité du code intermédiaire généré — le fameux *bytecode* facile a distribuer —, en font un langage de choix pour les applications client-serveur et *web* à plusieurs niveaux de code applicatif.

Ce chapitre se propose d'introduire les concepts de la modélisation objet et les opérations de base pour manipuler des objets persistants. Après cette introduction, la section 2 introduit les principes des modèles à objets en les illustrant par un modèle de référence proche de celui de l'OMG. La section 3 définit plus précisément ce qu'est un SGBDO et discute des principales techniques de gestion de la persistance proposées dans les systèmes. La section 4 propose une algèbre pour objets complexes définie sous forme de classes et permettant de manipuler des collections d'objets à un niveau intermédiaire entre un langage SQL étendu et un langage de programmation navigationnel de type C++ persistant. La conclusion résume les points abordés et introduit les problèmes essentiels à résoudre pour réaliser un SGBDO.

2. LE MODELE OBJET DE REFERENCE

2.1 Modélisation des objets

Les modèles de données à objets ont été créés pour modéliser directement les entités du monde réel avec un comportement et un état. Le concept essentiel est bien sûr celui d'**objet**. Il n'est pas simple à définir car composite, c'est-à-dire intégrant plusieurs aspects. Dans un modèle objet, toute entité du monde réel est un objet, et réciproquement, tout objet représente une entité du monde réel.

> **Notion XI.1 : Objet (*Object*)**
>
> Abstraction informatique d'une entité du monde réel caractérisée par une identité, un état et un comportement.

Un objet est donc une instance d'entité. Il possède une identité qui permet de le repérer. Par exemple, un véhicule particulier V1 est un objet. Un tel véhicule est caractérisé par un état constitué d'un numéro, une marque, un type, un moteur, un nombre de kilomètres parcourus, etc. Il a aussi un comportement composé d'un ensemble d'opérations permettant d'agir dessus, par exemple créer(), démarrer(), rouler(), stopper(), détruire(). Chaque opération a bien sûr des paramètres que nous ne précisons pas pour l'instant.

L'objet de type véhicule d'identité V1 peut être représenté comme un groupe de valeurs nommées avec un comportement associé, par exemple :

V1 {

Numéro: 812 RH 94, Marque: Renault, Type: Clio, Moteur: M1 ;

créer(), démarrer(), rouler(), stopper(), détruire() }.

Une personne est aussi un objet caractérisé par un nom, un prénom, un âge, une voiture habituellement utilisée, etc. Elle a un comportement composé d'un ensemble d'opérations { naître(), vieillir() , conduire(), mourir() }. L'objet de type personne d'identité P1 peut être décrit comme suit :

P1 {

Nom: Dupont, Prénom: Jules, Age: 24, Voiture: V1 ;

naître(), vieillir(), conduire(), mourir() }.

Un objet peut être très simple et composé seulement d'une identité et d'une valeur (par exemple, un entier E1 {Valeur: 212}). Il peut aussi être très complexe et lui-même composé d'autres objets. Par exemple, un avion est composé de deux moteurs, de deux ailes et d'une carlingue, qui sont eux-mêmes des objets complexes.

A travers ces exemples, deux concepts importants apparaissent associés à la notion d'objet. Tout d'abord, un objet possède un **identifiant** qui matérialise son identité. Ainsi, deux objets ayant les mêmes valeurs, mais un identifiant différent, sont considérés comme différents. Un objet peut changer de valeur, mais pas d'identifiant (sinon, on change d'objet).

> **Notion XI.2 : Identifiant d'objet (*Object Identifier*)**
>
> Référence système unique et invariante attribuée à un objet lors de sa création permettant de le désigner et de le retrouver tout au long de sa vie.

L'**identité d'objet** [Khoshafian86] est un concept important : c'est la propriété d'un objet qui le distingue logiquement et physiquement des autres objets. Un identifiant d'objet est en général une adresse logique invariante. L'attribution d'identifiants internes invariants dans les bases de données à objets s'oppose aux bases de données relationnelles dans lesquelles les données (tuples) ne sont identifiés que par des valeurs (clés). Deux objets O1 et O2 sont identiques (on note O1 == O2) s'ils ont le même identifiant ; il n'y a alors en fait qu'un objet désigné par deux pointeurs O1 et O2. L'identité d'objet est donc l'égalité de pointeurs. Au contraire, deux objets sont égaux (on note O1 = O2) s'ils ont même valeur. O1 == O2 implique O1 = O2, l'inverse étant faux.

L'identité d'objet apporte une plus grande facilité pour modéliser des objets complexes ; en particulier, un objet peut référencer un autre objet. Ainsi, le véhicule V1 référence le moteur M1 et la personne P1 référence le véhicule V1. Les graphes peuvent être directement modélisés (voir figure XI.1). Le **partage référentiel** d'un sous-objet commun par deux objets devient possible sans duplication de données. Par exemple, une personne P2 { Nom: Dupont; Prénom:

Juliette; Age: 22; Voiture: V1 } peut référencer le même véhicule que la personne P1.

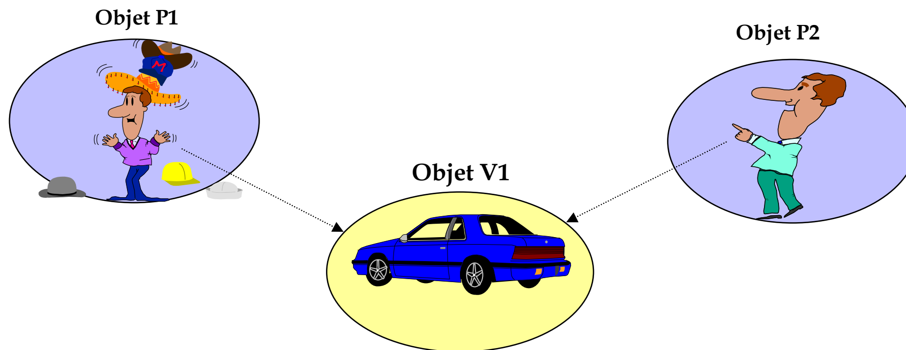


Figure XI.1 — Partage référentiel d'un objet par deux autres objets

Comme le montre ces exemples, en plus d'un identifiant, un objet possède des **attributs** aussi appelés **variables d'instance**. Un attribut mémorise une valeur ou une référence précisant une caractéristique d'un objet. La valeur peut être élémentaire (un entier, un réel ou un texte) ou complexe (une structure à valeurs multiples). La référence correspond à un identifiant d'un autre objet. Elle permet de pointer vers un autre objet avec des pointeurs invariants.

> **Notion XI.3 : Attribut (*Attribute*)**
>
> Caractéristique d'un objet désignée par un nom permettant de mémoriser une ou plusieurs valeurs, ou un ou plusieurs identifiants d'objets.

2.2 Encapsulation des objets

Au-delà d'une structure statique permettant de modéliser des objets et des liens entre objets, les modèles à objets permettent d'encapsuler les structures des objets par des **opérations**, parfois appelées **méthodes** (en Smalltalk) ou **fonctions membres** (en C++).

> **Notion XI.4 : Opération (*Operation*)**
>
> Modélisation d'une action applicable sur un objet, caractérisée par un en-tête appelé signature définissant son nom, ses paramètres d'appel et ses paramètres de retour.

Le terme méthode sera aussi employé pour désigner une opération. Issu de Smalltalk, ce concept a cependant une connotation plus proche de l'implémentation, c'est-à-dire que lorsqu'on dit méthode, on pense aussi au code constituant l'implémentation. Quoi qu'il en soit, un objet est manipulé par les méthodes qui l'encapsulent et accédé via celles-ci : le **principe d' encapsulation** hérité des types abstraits cache les structures de données (les attributs) et le code des méthodes en ne laissant visible que les opérations exportées, appelées opérations **publiques**. Par opposition, les opérations non exportées sont qualifiées de **privées**. Elles ne sont accessibles que par des méthodes associées à l'objet.

L'encapsulation est un concept fondamental qui permet de cacher un groupe de données et un groupe de procédures associées en les fusionnant et en ne laissant visible que **l'interface** composée des attributs et des opérations publics.

> **Notion XI.5 : Interface d'objet (*Object Interface*)**
>
> Ensemble des signatures des opérations, y compris les lectures et écritures d'attributs publics, qui sont applicables depuis l'extérieur sur un objet.

L'interface d'un objet contient donc toutes les opérations publiques que peuvent utiliser les clients de l'objet. Pour éviter de modifier les clients, une interface ne doit pas être changée fréquemment : elle peut être enrichie par de nouvelles opérations, mais il faut éviter de changer les signatures des opérations existantes. En conséquence, un objet exporte une interface qui constitue un véritable contrat avec les utilisateurs. Les attributs privés (non visibles à l'extérieur) et le code des opérations peuvent être modifiés, mais changer des opérations de l'interface nécessite une reprise des clients.

Ce principe facilite la programmation modulaire et l'indépendance des programmes à l'implémentation des objets. Par exemple, il est possible de développer une structure de données sous la forme d'un tableau, permettant de mémoriser le contenu d'un écran. Cette structure peut être encapsulée dans des opérations de signatures fixées permettant d'afficher, de redimensionner, de saisir des caractères. Le changement du tableau en liste nécessitera de changer le code des opérations, mais pas l'interface, et donc pas les clients.

En résumé, les opérations et attributs publics constituent l'interface d'un objet. Ceux-ci sont les seuls accessibles à l'extérieur de l'implémentation de l'objet. Le code qui constitue cette implémentation peut être structuré. Certaines opérations sont alors invisibles à l'extérieur de l'objet : elles sont appelées **opérations privées**. Par exemple, la saisie d'un texte peut s'effectuer par plusieurs appels d'une méthode saisissant une ligne : la méthode SaisirLigne restera invisible au monde extérieur et seule la méthode SaisirTexte pourra être invoquée. Dans la suite et afin de simplifier, nous supposerons que toutes les méthodes d'un objet sont publiques. Si nous ne le précisons pas, les attributs sont publiques. Mettre des attributs publics ne respecte pas très bien le principe d'encapsulation qui consiste à cacher les propriétés servant à l'implémentation. Briser ainsi l'encapsulation conduit à des difficultés si l'on veut changer par exemple le type d'un attribut : il faut alors prévenir les clients qui doivent être modifiés !

Notez qu'un attribut d'un objet peut être lu par une fonction appliquée à l'objet délivrant la valeur de l'attribut. Nous noterons cette fonction GetAttribut, où Attribut est le nom de l'attribut considéré. Ainsi, l'attribut Nom définit une fonction GetNom qui, appliquée à une personne P, délivre un texte (son nom). La propriété Voiture peut aussi être lue par une fonction GetVoiture qui, appliquée à une personne, délivre l'identifiant de l'objet constituant son véhicule habituel. Un attribut peut aussi être écrit par une fonction particulière que nous noterons PutAttribut. En résumé, tout attribut définit implicitement deux méthodes (écriture : Put, et lecture : Get) qui peuvent être privées ou publiques, selon que l'attribut est visible ou non à l'extérieur. Le formalisme unificateur des fonctions

est très puissant, car il permet de raccrocher à une même théorie (les types abstraits) les propriétés mémorisées (attributs) et calculées (fonctions) d'un objet.

2.3 Définition des types d'objets

Le concept de **type de données abstrait** est bien connu dans les langages de programmation [Guttag77]. Un type de données abstrait peut être vu comme un ensemble de fonctions qui cache la représentation d'un objet et contraint les interactions de l'objet avec les autres objets. Ainsi, un type de données abstrait englobe une représentation cachée d'un objet et une collection de fonctions abstraites visibles à l'extérieur. La définition d'un type de données abstrait correspond à la définition d'une ou plusieurs interfaces, comme vu ci-dessus. L'emploi de types abstraits favorise la modularité des programmes car la modification de la structure d'un objet n'affecte pas les objets extérieurs qui le manipulent.

Dans les systèmes à objets, les types abstraits (et donc les interfaces) doivent être implémentés. En général, un type abstrait est implémenté sous la forme d'un moule permettant de définir les attributs et les opérations communs associés aux objets créés selon ce moule. Un tel moule est appelé **classe**.

> **Notion XI.6 : Classe (*Class*)**
>
> Implémentation d'une ou plusieurs interfaces sous la forme d'un moule permettant de spécifier un ensemble de propriétés d'objets (attributs et opérations) et de créer des objets possédant ces propriétés.

Notez que certains systèmes à objets séparent la fonction de création des objets de la classe : on obtient alors des classes qui sont simplement des implémentations de types abstraits et l'on ajoute des **usines à objets** (*Object factory*) pour créer les objets. Chaque classe doit alors avoir son usine, ce qui complique le modèle.

Une classe spécifie donc la structure et le comportement communs des objets qu'elle permet de créer. Au-delà du nouveau type de données abstrait ajouté à l'environnement par une définition de classe, une classe supporte une implémentation : c'est le code des opérations. Elle donne aussi naissance à une famille d'objets : on parle alors de l'**extension de la classe**. Cette extension est une collection d'objets ayant mêmes structure et comportement. La classe est donc un peu l'analogue de la table dans les bases de données relationnelles, bien qu'une table ne permette que de modéliser la structure commune des tuples qui la composent.

En résumé, le concept de classe est plutôt complexe. Par abus de langage, le mot classe désigne généralement une intention (le type abstrait), mais aussi parfois une extension (la collection des objets membres de la classe), d'autre fois une implémentation (la structure des objets et le code des opérations). La spécification progressive des classes d'objets composant une base de données à objets permet de modéliser le comportement commun des objets de manière modulaire et extensible. Elle permet aussi de spécifier les collections contenant les objets ou extensions de classes. La plupart des systèmes distinguent l'intention de

l'extension, une classe (ou un type selon le vocabulaire) pouvant avoir plusieurs extensions.

Du point de vue de la représentation d'une définition de classe, nous utiliserons la notation préconisée par UML [Rational97] et illustrée figure XI.2. Nous avons à gauche le cadre générique servant à représenter une classe, à droite le cas de la classe `Vin`. Chaque attribut a un nom, de même que chaque opération. Un attribut est typé. Les opérations peuvent avoir des paramètres, et un type dans le cas de fonctions. Un attribut ou une opération publics sont précédés de + ; de même pour une opération.

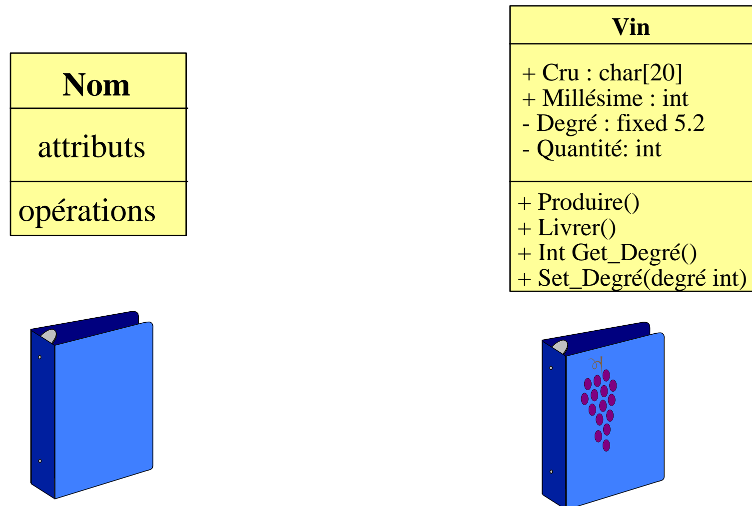


Figure XI.2 — Représentation d'une classe en UML

Nous pouvons maintenant illustrer la notion de classes par quelques exemples. La syntaxe choisie pour les définitions de classe est proche de C++, chaque attribut ou méthode ayant un type suivi d'un nom et d'éventuels paramètres. Notre premier exemple (voir Figure XI.3) vise à modéliser le plan géométrique sous forme de points. Pour créer et rassembler les points créés, nous définissons une classe comportant deux attributs, `X` et `Y`, de type flottant. Une méthode sans paramètre `Distance` permet de calculer la distance du point à l'origine ; cette méthode retourne un flottant. Une méthode `Translater` permet de translater un point d'un vecteur fourni en paramètre et ne retourne aucun paramètre (void en C++). Une autre méthode `Afficher` permet de générer un point lumineux sur un écran ; elle ne retourne aucun paramètre.

```
Class Point {
    Float X;
    Float Y;
    Float Distance();
    Void Translater(Float DX, Float DY);
    Void Afficher(); };
```

Figure XI.3 — La classe point

La figure XI.4 permet de définir les classes `Vin`, `Personne` et `Véhicule` dont quelques objets ont été vus ci-dessus. Les mots clés `Public` et `Private` permettent de préciser si les propriétés sont exportées ou non. Par défaut, il restent cachés dans la classe, donc privés. Notez que comme en C++, nous définissons une référence par le type de l'objet pointé suivi d'une étoile (par exemple `Véhicule*`). Deux méthodes sont attachées à la classe `Personne` : `Vieillir` qui permet d'incrémenter de 1 l'âge d'une personne et rend le nouvel âge, `Conduire` qui permet de changer le véhicule associé et ne retourne aucun paramètre. Notez que la classe `Véhicule` référence d'autres classes (`Constructeur`, `Propulseur`) non définies ici.

```
Class Vin {
        Fixed 5.2 degré ;
        Int quantité ;
        Public :
        Char cru [20] ;
        Int millésime ;
        void Produire()
        void Livrer()
        Int Get_Degré()
        void Set_Degré(Int degré) } ;

Class Personne {
        String Nom;
        String Prénom;
        Int Age;
        Véhicule* Voiture;
        Public :
        Int Vieillir();
        Void Conduire(Véhicule V); };

Class Véhicule {
        Public :
        String Numéro;
        Constructeur* Marque;
        String Type;
        Propulseur* Moteur; };
```

Figure XI.4 — Les classes Vin, Personne et Véhicule

Une opération définie dans le corps d'une classe s'applique à un objet particulier de la classe. Par exemple, pour translater un point référencé par P d'un vecteur unité, on écrira `P→Translater(1,1)`. Pour calculer la distance de P à l'origine dans une variable d, on peut écrire `d = P→Distance()`. Certaines méthodes peuvent s'appliquer à plusieurs objets de classes différentes : une telle méthode est dite **multiclasse**. Elle est en général affectée à une classe particulière, l'autre étant un paramètre. La possibilité de supporter des méthodes multiclasses est essentielle pour modéliser des associations entre classes sans introduire de classes intermédiaires artificielles.

2.4 Liens d'héritage entre classes

Afin d'éviter la répétition de toutes les propriétés pour chaque classe d'objets et de modéliser la relation « est-un » entre objets, il est possible de définir de nouvelles classes par affinage de classes plus générales. Cette possibilité est importante pour faciliter la définition des classes d'objets. Le principe est d'affiner une classe plus générale pour obtenir une classe plus spécifique. On peut aussi procéder par mise en facteur des propriétés communes à différentes classes afin de définir une classe plus générale. La notion de **généralisation** ainsi introduite est empruntée aux modèles sémantiques de données [Hammer81, Bouzeghoub85].

> **Notion XI.7 : Généralisation (*Generalization*)**
>
> Lien hiérarchique entre deux classes spécifiant que les objets de la classe supérieure sont plus généraux que ceux de la classe inférieure.

La classe inférieure est appelée **sous-classe** ou **classe dérivée**. La classe supérieure est appelée **super-classe** ou **classe de base**. Le parcours du lien de la super-classe vers la sous-classe correspond à une **spécialisation**, qui est donc l'inverse de la généralisation. Une sous-classe reprend les propriétés (attributs et méthodes) des classes plus générales. Cette faculté est appelée **héritage**.

> **Notion XI.8 : Héritage (*Inheritance*)**
>
> Transmission automatique des propriétés d'une classe de base vers une sous-classe.

Il existe différentes sémantiques de la généralisation et de l'héritage, qui sont deux concepts associés [Cardelli84, Lécluse89]. La plus courante consiste à dire que tout élément d'une sous-classe est élément de la classe plus générale : il hérite à ce titre des propriétés de la classe supérieure. La relation de généralisation est alors une relation d'inclusion. Bien que reprenant les propriétés de la classe supérieure, la classe inférieure possède en général des propriétés supplémentaires : des méthodes ou attributs sont ajoutés.

Le concept de généralisation permet de définir un **graphe de généralisation** entre classes. Les nœuds du graphe sont les classes et un arc relie une classe C2 à une classe C1 si C1 est une généralisation de C2. Le graphe dont les arcs sont orientés en sens inverse est appelé **graphe d'héritage**. La figure XI.5 illustre un graphe de généralisation entre les classes `Personne`, `Employé`, `Cadre`, `NonCadre` et

`Buveur`. Les définitions de classes correspondantes dans un langage proche de C++ apparaissent figure XI.6. Les super-classes d'une classe sont définies après la déclaration de la classe suivie de « : ». A chaque classe est associé un groupe de propriétés défini au niveau de la classe. Les classes de niveaux inférieurs héritent donc des propriétés des classes de niveaux supérieurs.

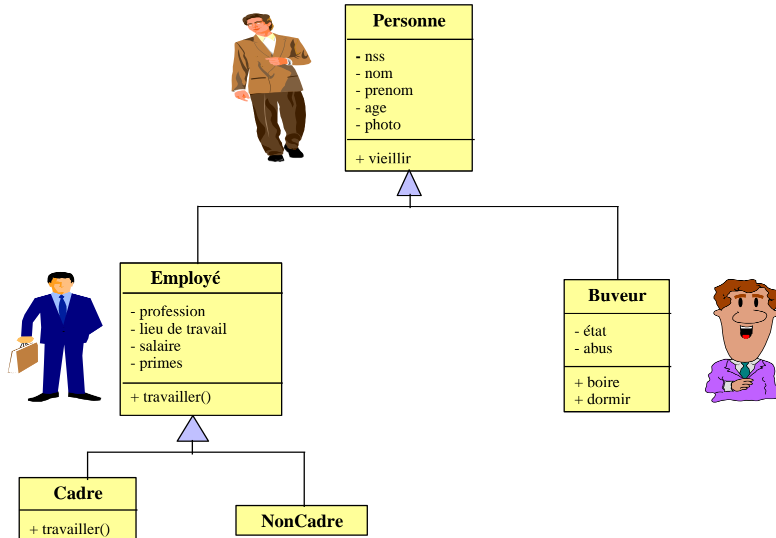


Figure XI.5 — Exemple de graphe de généralisation

```
Class Personne {
        Int nss ;
        String nom ;
        String prenom ;
        Int age ;
        Image* photo ;
 Public :
        Int vieillir() ; }

Class Employé : Personne {
        String profession ;
        String lieudetravail ;
        Double salaire ;
        Double primes ;
 Public :
        void travailler() ; }
```

```
Class Buveur : Personne {
        Enum etat (Normal, Ivre) ;
        List <consommation> abus ;
 Public :
        Int boire() ;
        void dormir() ; }

Class Cadre : Employé {
 Public :
        void travailler() ; }
Class NonCadre : Employé {
        }
```

Figure XI.6 — Définition des classes de la figure précédente

Afin d'augmenter la puissance de modélisation du modèle, il est souhaitable qu'une classe puisse hériter des propriétés de plusieurs autres classes. L'**héritage multiple** permet à une classe de posséder plusieurs super-classes immédiates.

> **Notion XI.9 : Héritage multiple (*Multiple Inheritance*)**
>
> Type d'héritage dans lequel une classe dérivée hérite de plusieurs classes de niveau immédiatement supérieur.

Dans ce cas, la sous-classe hérite des propriétés et opérations de toutes ses super-classes. L'héritage multiple permet de définir des classes intersection comme dans la figure XI.7, où les EmployésBuveurs héritent à la fois des Buveurs et des Employés. Ils héritent certes d'un seul nom provenant de la racine Personne.

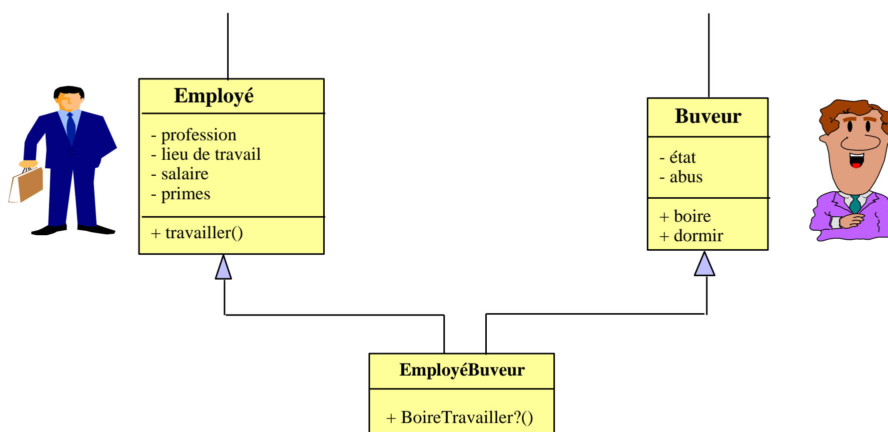


Figure XI.7 — Exemple d'héritage multiple

Des **conflits de noms** d'attributs ou d'opérations peuvent survenir en cas d'héritage multiple. Plusieurs solutions sont possibles. En particulier, un choix peut être effectué par l'utilisateur de manière statique à la définition, ou de manière

dynamique pour chaque objet. Il est aussi possible de préfixer les noms de propriétés ou méthodes héritées par le nom de la classe où elles ont été définies, cela bien sûr dans le seul cas d'ambiguïté de noms. Enfin, un choix automatique peut être fait par le système, par exemple le premier à gauche. Dans l'exemple de la figure XI.8, les triangles rectangles isocèles vont hériter de trois fonctions de calcul de surface. Deux d'entres elles sont identiques et proviennent de l'opération `surface` associée aux triangles. Il faut pouvoir choisir parmi les deux restantes, soit en conservant les deux en les renommant `polygone_surface` et `triangle_surface`, soit en en sélectionnant une, par exemple la première à gauche, donc celle des polygones droits.

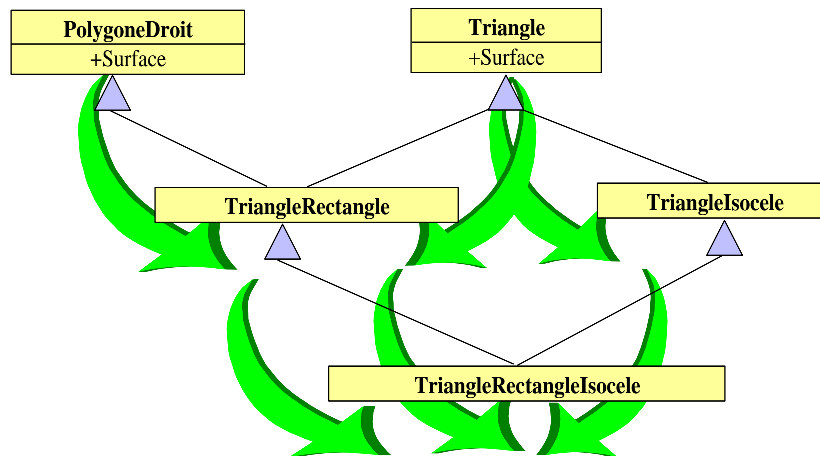


Figure XI.8 — Exemple de conflit de noms

Dire qu'une classe C1 est plus générale qu'une classe C2 permet de définir une relation d'ordre C2 $\subseteq$ C1, qui représente le fait que les instances de C2 sont incluses dans celles de C1. Cette relation d'ordre correspond donc à une inclusion au niveau des ensembles d'objets. La relation d'ordre ainsi définie permet de considérer le treillis des classes. Ce treillis possède en général un plus grand élément qui est la classe dénommée `Objet` (*Object*) ; cette classe apparaît alors comme la racine de la hiérarchie d'héritage qui fournit toutes les méthodes de base de manipulation des objets, méthodes héritées par toutes les autres classes. La hiérarchie des types et l'héritage de propriétés ont été formellement étudiés dans le contexte des types abstraits sous forme de sigma-algèbres [Guogen78], et plus récemment dans le contexte de la programmation objet [Castagna96].

2.5 Polymorphisme, redéfinition et surcharge

Lors de la définition d'une hiérarchie de classes, certaines propriétés peuvent être spécifiées différemment pour chaque sous-classe. L'idée est de modéliser dans une sous-classe une fonctionnalité similaire de même signature d'opération, mais avec un code différent. On parle alors de **redéfinition**.

> **Notion XI.10 : Redéfinition (*Overriding*)**
>
> Spécification d'une méthode existante dans une super-classe au niveau d'une sous-classe, avec une implémentation différente.

Par exemple, une méthode globale telle que `Travailler` peut être spécifiée au niveau de la classe `Employé`, puis redéfinie de manière spécifique au niveau de la classe `Cadre`. En effet, la méthode `Travailler` de la classe `Employé` pourra être redéfinie par un code correspondant à un travail de direction au niveau de la classe `Cadre`. Il est même possible de ne pas définir le code de l'opération au niveau d'une classe et d'imposer de le définir au niveau des sous-classes : on parle alors de **méthode virtuelle** au niveau de la super-classe.

Pour une même classe, il est aussi possible de définir plusieurs implémentations pour une même opération. Chacune d'elle sera alors distinguée par des paramètres différents. Ceci correspond à la notion de **surcharge**.

> **Notion XI.11 : Surchage (*Overloading*)**
>
> Possibilité de définir plusieurs codes pour une même opération d'une classe, le code approprié étant sélectionné selon le type des paramètres fournis lors d'un appel.

La possibilité de surcharge rend nécessaire le choix du code d'une méthode en fonction de ses arguments. Cette possibilité est d'ailleurs généralisée : une opération d'un nom donné peut avoir différentes signatures; un code est alors attaché à chaque signature. Par exemple, au niveau de la classe `Personne`, il est possible de définir plusieurs méthodes `Vieillir`, l'une sans paramètre ajoutant un à l'âge, l'autre avec un paramètre indiquant le nombre d'années à ajouter. Chaque signature aura alors un code spécifique. Cette possibilité est souvent utilisée pour gérer des valeurs par défaut de paramètres.

Redéfinition et surcharge sont deux formes de **polymorphisme**. Cette fonctionnalité importante peut être définie comme suit :

> **Notion XI.12 : Polymorphisme (*Polymorphism*)**
>
> Faculté pour une opération d'avoir différentes signatures avec un code spécifique attaché à chaque signature.

Le polymorphisme permet donc à une même opération de s'appliquer à des objets de différentes classes ou à des objets d'une même classe. Dans ce dernier cas, les paramètres de l'opération doivent être de types différents. En particulier, certains peuvent ne pas exister.

Par exemple, la méthode `Travailler()` peut être appliquée à un objet de type `Cadre` ou `NonCadre` sans argument. Une méthode `Travailler(Int Durée)` pourra être définie au niveau des `NonCadre`. Lors d'un appel de la méthode `Travailler` du type `P→Travailler()` ou `P→Travailler(10)` (P référence une personne particulière), un code différent

sera exécuté, selon que P est un cadre ou non, et selon qu'un paramètre est passé ou non pour un non cadre.

Le polymorphisme (étymologiquement, la faculté de posséder plusieurs formes) est une fonctionnalité évoluée qui implique en général le choix du code de la méthode à l'exécution (lorsque le type des paramètres est connu). Ce mécanisme est appelé **liaison dynamique** (en anglais, *dynamic binding*).

2.6 Définition des collections d'objets

Certains attributs peuvent être multivalués. Par exemple, un livre peut avoir plusieurs auteurs. Un texte est composé de plusieurs paragraphes. Il est donc souhaitable de pouvoir regrouper plusieurs objets pour former un seul attribut. Ceci s'effectue généralement au moyen de classes génériques permettant de supporter des **collections** d'objets.

> **Notion XI.13 : Collection (*Collection*)**
>
> Container typé désigné par un nom, contenant des éléments multiples organisés selon une structure particulière, auxquels on accède par des opérations spécifiques au type du container.

Les principales collections sont :

- l'**ensemble** (Set) qui permet de définir des collections non ordonnées sans double ;
- le **sac** (Bag) qui permet de définir des collections non ordonnées avec doubles;
- la **liste** (List) qui permet de définir des collections ordonnées avec doubles ;
- le **tableau** (Array) qui permet de définir des collections ordonnées et indexées.

D'autres sont utilisées, par exemple la pile, le tableau insérable où il est possible d'insérer un élément à une position donnée, etc. Les éléments rangés dans les collections sont en général des objets ou des valeurs. Par exemple {10, 20, 30, 75} est un ensemble d'entiers ; <O1, O5, O9> où les Oi représentent des objets est une liste d'objets.

Dans les systèmes objet, la notion de collection est souvent réalisée par des **classes paramétrées**, encore appelées **classes génériques** ou **patterns de classes**.

> **Notion XI.14 : Classe paramétrée (*Template*)**
>
> Classe avec paramètres typés et ayant différentes implémentations selon le type de ces paramètres.

La génération des implémentations selon le type des paramètres est généralement à la charge du compilateur du langage objet de définition de classe paramétrée. L'intérêt d'utiliser des classes paramétrées est la facilité de réutilisation, dans des contextes différents selon les paramètres. L'inconvénient est souvent l'accroissement de la taille du code généré.

Une classe paramétrée est donc une classe possédant un ou plusieurs paramètres ; dans le cas des collections, le paramètre sera souvent le type des objets de la collection. La collection doit donc être homogène. Elle apparaît à son tour comme une classe. Par exemple List<X> et Set<X> sont des collections, respectivement des listes et des ensembles d'objets de type X. Set<Int> est un ensemble d'entiers alors que List<Vin> est une liste de vins. List<X> et Set<X> sont des classes paramétrées. List<Vin> étant une classe, Set<List<Vin>> est un type de collection valide. Les collections permettent donc de construire des objets complexes par imbrications successives de classes. Par exemple, { <V1, V5, V7>, <V3, V2, V1>, <V4, V7>, <>} est une collection de type Set<List<Vin>>.

Dans la plupart des modèles à objets, les collections sont des classes d'objets génériques qui offrent des méthodes d'accès spécifiques (par exemple, le parcours d'une liste ou d'un ensemble). Elles peuvent être organisées en hiérarchie de généralisation. La figure XI.9 illustre une bibliothèque de classes génériques de type collections avec, pour chacune, des méthodes caractéristiques [Gardarin94]. Outre les méthodes d'insertion, suppression et obtention d'un élément, la classe Collection offre des méthodes de second ordre travaillant sur plusieurs objets d'une collection ; Count compte le nombre d'objets, Apply applique une fonction passée en paramètre à chaque objet et Aggregate calcule une fonction d'agrégat passée en paramètre (par exemple, somme ou moyenne). Search recherche un élément donné, par exemple par une clé. Les opérations des sous-classes sont classiques. Par exemple, First et Next permettent le parcours de liste, Tail extrait la sous-liste après l'élément courant et Append ajoute un élément. D'autres fonctions peuvent être envisagées.

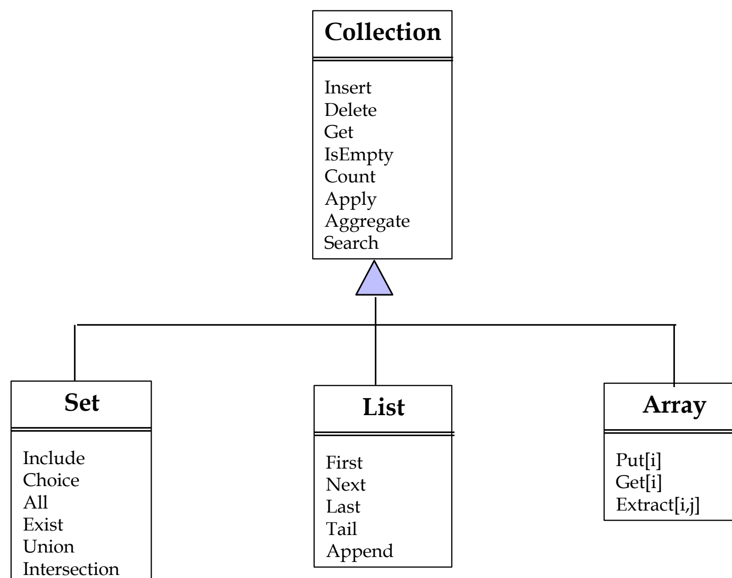


Figure XI.9 — Exemple de bibliothèque de collections avec opérations

On écrira par exemple Array<Véhicule> pour spécifier un tableau de Véhicules de dimension variable. Notez que la plupart des langages orientés objets

supportent des tableaux de dimension fixe par la notation C[N], où C est une classe et N un entier. Par exemple, `Véhicule[4]` définit un tableau de 4 véhicules. Comme cela a déjà été dit, il est possible d'imbriquer des collections. L'exemple de la figure XI.10 montre comment on peut gérer des listes de tableau afin de structurer intelligemment un texte. Celui-ci est composé d'une liste de paragraphes, chacun ayant un thème, des détails structurés sous forme d'une liste de phrases, et une conclusion. Thème et conclusion sont des phrases. Une phrase est une liste de mots modélisés par des tableaux de caractères.

```
class Phrase {
        List <Array <char>> Séquence; };
class Paragraphe {
        Phrase Thème;
        List <Phrase> Détails;
        Phrase Conclusion; };
class Texte {
        List <Paragraphe*> Contenu; };
```

Figure XI.10 — Utilisation de collections imbriquées

L'existence de collections imbriquées traduit le fait que certains objets sont inclus dans d'autres. Il s'agit en fait d'une représentation de la relation d'**agrégation** entre classes.

> **Notion XI.15 : Agrégation (*Aggregation*)**
>
> Association entre deux classes exprimant que les objets de la classe cible sont des composants de ceux de la classe source.

L'agrégation traduit la relation « fait partie de » ; par exemple, les caractères font partie de la phrase. Les collections permettent d'implémenter les agrégations multivaluées. L'agrégation peut être implémentée par une référence (par exemple `List<Paragraphe*>`) ou par une imbrication, selon que l'on désire ou non partager les objets inclus. La représentation des agrégations par un graphe permet de visualiser le processus de construction des objets complexes. La figure XI.11 représente le graphe d'agrégation, encore appelé graphe de composition, de la classe Texte. Les cardinalités multiples sont mentionnées par des étoiles (*), conformément à la notation UML [Rational97].

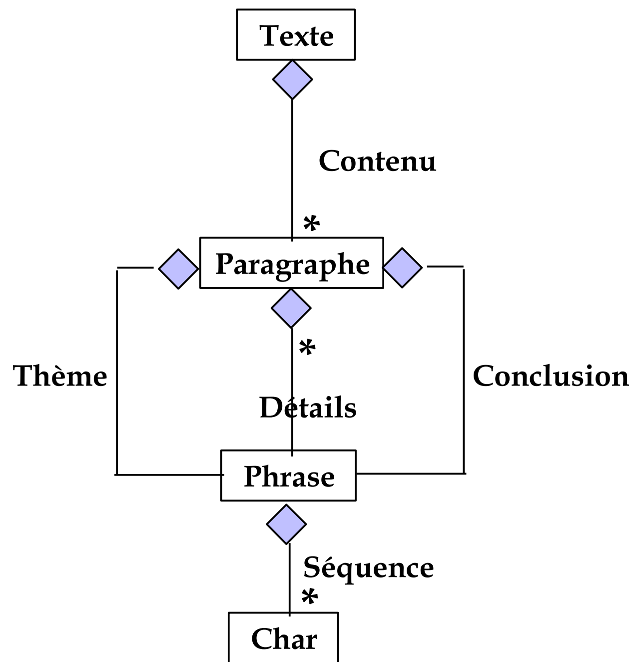


Figure XI.11 — Graphe d'agrégation de la classe Texte

2.7 Prise en compte de la dynamique

Dans les environnements à objets, les objets communiquent entre eux par des **messages** comportant le nom d'une méthode et ses paramètres. Un objet reçoit un message et réagit à un message. L'envoi de messages est en fait une implémentation flexible et contrôlée de l'appel de procédure par valeur classique des langages de programmation. C'est un élément important des langages objet qui permet de réaliser le polymorphisme lors de l'interprétation du message.

> **Notion XI.16 : Message (*Message*)**
>
> Bloc de paramètres composé d'un objet récepteur, d'un nom d'opérations et de paramètres, permettant par envoi l'invocation de l'opération publique de l'objet récepteur.

Ainsi, un objet réagit à un message en sélectionnant le code de la méthode associée selon le nom de la méthode et le type des paramètres, réalisant ainsi le polymorphisme. Il retourne les paramètres résultats de l'opération. Une base de données à objets ou un programme structuré en objets apparaît donc comme un ensemble d'objets vivants qui communiquent par des messages. Un dispatcher de messages fait transiter les messages entre les objets et assure leur bonne délivrance. Un nouveau graphe peut être construit dynamiquement, le **graphe des appels de méthodes**, qui permet de visualiser quelle méthode a appelé quelle autre méthode.

Un message peut être envoyé à un objet désigné par son identifiant en utilisant le **sélecteur de propriété** dénoté →, déjà vu ci-dessus. Par exemple, un point défini comme figure XI.3, référencé par P, pourra recevoir les messages :

P→Distance();

P→Translater(10,10);

P→Afficher().

Le dispatcher du message activera le code de l'opération sélectionnée en effectuant le passage de paramètre par valeur.

2.8 Schéma de bases de données à objets

A partir des outils et concepts introduits ci-dessus, une base de données à objets peut être décrite. La description s'effectue par spécification d'un **schéma de classes** composé de définitions de classes, chacune d'elles comportant des attributs (éventuellement organisés en collections) et des opérations. Parmi les classes, certaines sont plus générales que d'autres, comme spécifié par le graphe de généralisation. La figure XI.12 représente le schéma partiel d'une base de données à objets dont d'autres éléments ont été introduits ci-dessus. Le graphe de référence entre classes correspondant est représenté figure XI.14

> **Notion XI.17 : Schéma de BD objet (*Object DB Schema*)**
>
> Description d'une base de données à objets particulière incluant les définitions de classes, d'attributs et d'opérations ainsi que les liens entre classes.

Les agrégations, cas particuliers d'associations, sont bien sûr représentées. Un schéma de BD objet est donc un schéma objet général. Il pourra en plus inclure les directions de traversée des associations, afin de faciliter le passage à des références entre objets au niveau de l'implémentation. La figure XI.12 représente un schéma de BD objet sous une forme proche de la notation UML ; la figure XI.13 en donne une vue selon une syntaxe proche de C++.

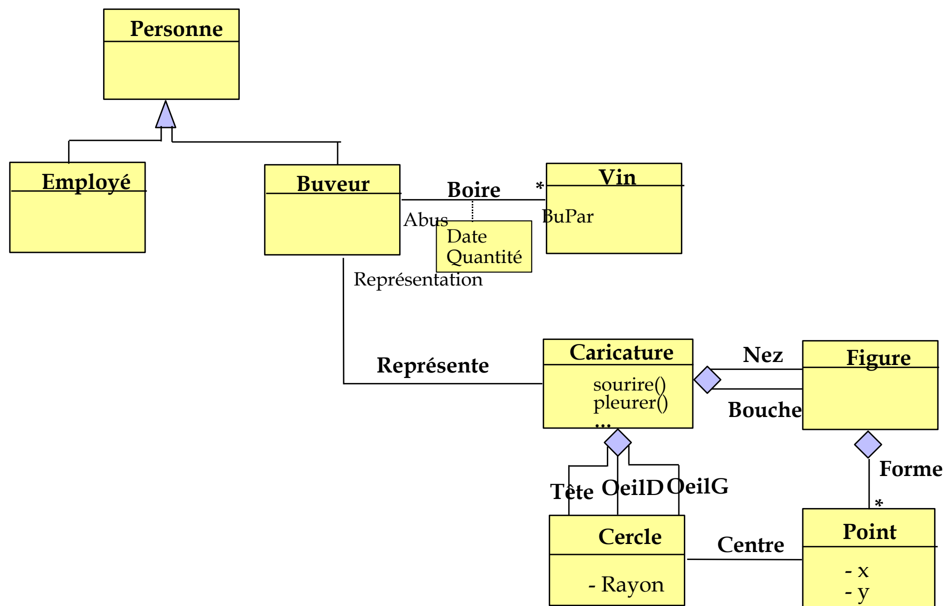


Figure XI.12 — Schéma graphique partiel de la base exemple

```
Class Vin {
public:
        Char Cru[20] ;
        Int Mill ;
        Double Degré ; }

Class Buveur {
private:
        Enum Etat (Normal, Ivre) ;
public:
        Char Nom[15] ;
        Char Prénom[15] ;
        List <Date Jour, Vin* Boisson, Int Quantité> Abus ;
        Caricature* Représentation ;
        // opération sur buveur
        void Boire (Date d, Vin* v, Int q) ;
        void Dormir (Int durée); }
```

```
Class Caricature {
        Cercle* Tête ;
        Cercle* OeilD ;
        Cercle* OeillG ;
        Figure* Nez ;
        Figure* Bouche ;
        // operations sur personnage
        void Sourire () ;
        void Pleurer () ;
        Graphic Dessiner (Int Echelle) ; }

Class Cercle {
        Point Centre ;
        Double Rayon ; }

Class Figure {
        List <Point> Forme ; }
```

Figure XI.13 — Définition C++ d'une partie de la base exemple

Soulignons qu'un schéma de classes modélise à la fois la structure (les attributs des classes) et le comportement (les méthodes) des objets. La définition complète d'une base de données à objets nécessite donc un langage de programmation afin de spécifier le code des programmes. Ce langage est un langage orienté objet, du type Smalltalk, ou de plus en plus souvent C++ ou Java.

3. LA PERSISTANCE DES OBJETS

Après avoir précisé le concept de base de données à objets, cette section étudie les diverses techniques utilisées pour réaliser la persistance des objets nécessaire à l'approche base de données. La dernière partie montre comment on navigue dans une base de données à objets depuis un langage orienté objet tel C++.

3.1 Qu'est-ce-qu'une BD à objets ?

La notion de bases de données à objets s'est précisée au début des années 90. Nous avons introduit ci-dessus les concepts essentiels de la modélisation orientée objet, tels qu'ils sont implémentés dans des langages objet Smalltalk, C++ ou Java. A partir de là, le concept de base de données à objets s'élabore en ajoutant la persistance. La notion de système de base de données à objets étant au départ très confuse, Atkinson, Bancilhon, Dewitt, Ditrich, Maier et Zdonick ont proposé de la clarifier dans une publication à la conférence DOOD (*Deductive and Object-*

Oriented Databases) en 1989, intitulée « *The Object-Oriented Manifesto* » [Atkinson89].

Pour mériter le nom de SGBD objet (SGBDO), un système doit d'abord supporter les fonctionnalités d'un SGBD. Pour ce faire, il doit obligatoirement assurer :

- La **persistance des objets**. Tout objet doit pouvoir persister sur disque au-delà du programme qui le crée. Un objet peut être **persistant** ou **transient**. Dans le deuxième cas, sa durée de vie est au plus égale à celle du programme qui le crée ; il s'agit d'un objet temporaire qui reste en mémoire.

> **Notion XI.18 : Objet persistant (*Persistent object*)**
>
> Objet stocké dans la base de données dont la durée de vie est supérieure au programme qui le crée.

> **Notion XI.19 : Objet transient (*Transient object*)**
>
> Objet restant en mémoire, dont la durée de vie ne dépasse pas celle du programme qui le crée.

- La **concurrence d'accès**. La base d'objets doit pouvoir être partagée simultanément par les transactions qui la consultent et la modifient ; les blocages doivent être minimaux afin d'assurer la cohérence de la base.
- La **fiabilité des objets**. Les objets doivent être restaurés en cas de panne d'un programme dans l'état où ils étaient avant la panne. Les transactions doivent être atomiques, c'est-à-dire totalement exécutées ou pas du tout.
- La **facilité d'interrogation**. Il doit être possible de retrouver un objet à partir de valeurs de ses propriétés, qu'il s'agisse de valeurs d'attributs, de résultats de méthodes appliquées à l'objet ou de liens avec les objets référencés ou référençants.

Le manifesto propose en outre des fonctionnalités bases de données optionnelles :

- La **distribution des objets**. Cette facilité permet de gérer des objets sur différents sites, en particulier sur un serveur et des clients.
- Les **modèles de transaction évolués**. Il s'agit de supporter des transactions imbriquées, c'est-à-dire elles-mêmes décomposées en sous-transactions qui peuvent être totalement reprises.
- Les **versions d'objets**. La gestion de versions permet de revenir à un état antérieur de l'objet avant modification. A partir d'un objet, plusieurs versions peuvent être créées par des modifications successives ou parallèles. On aboutit ainsi à un graphe des versions d'un objet qui peut être géré par le système. Il est alors possible de remonter aux versions précédentes à partir des dernières versions et vice versa. Si plusieurs versions sont créées en parallèle, une fusion avec possibilité de choisir certaines modifications est ultérieurement nécessaire. Un objet pouvant posséder des versions est appelé **objet versionnable**.

> **Notion XI.20 : Objet à versions (*Versionnable object*)**
>
> Objet dont l'historique des instances créées (successivement ou simultanément) est gardé dans la base sous forme de versions consultables et modifiables.

Outre les fonctionnalités orientées bases de données définies ci-dessus, le manifesto prescrit les fonctionnalités orientées objets que doit supporter un SGBDO. Sont obligatoires :

- Le **support d'objets atomiques et complexes**. Il s'agit de supporter des objets avec des attributs références et des collections imbriquées.
- L'**identité d'objets**. Tout objet doit avoir un identifiant système invariant, permettant de le retrouver sur disque et en mémoire.
- L'**héritage simple**. Une classe doit pouvoir être une spécialisation d'une autre classe et hériter de celle-ci.
- Le **polymorphisme**. Le code d'une méthode doit être choisi en fonction de ses paramètres instanciés.

Deux fonctionnalités sont optionnelles :

- L'**héritage multiple**. Il permet qu'une sous-classe soit la spécialisation directe de deux sur-classes ou plus. Elle hérite alors de toutes ses sur-classes.
- Les **messages d'exception**. Il s'agit d'un mécanisme de traitement d'erreur analogue à celui introduit en C++ ou en Java. Lorsqu'une erreur survient dans une méthode, un message d'exception est levé. Il peut être repris par un traitement d'erreur inséré par exemple dans un bloc de reprise, par une syntaxe du type : `try <traitement normal> catch <exception : traitement d'erreur>`.

En résumé, le manifesto essaie de définir précisément ce qu'est une base de données à objets. S'il a apporté en son temps une clarification, il manque aujourd'hui de précision, si bien qu'un SGBD objet-relationnel, c'est-à-dire un SGBD relationnel étendu avec des types abstraits, peut souvent être classé comme un SGBDO. Il présente aussi un peu d'arbitraire dans la sélection de fonctionnalités, notamment au niveau des options.

3.2 Gestion de la persistance

Un modèle de données orienté objet permet de définir les types des objets. Dans les environnements de programmation, les objets doivent être construits et détruits en mémoire par deux fonctions spécifiques, appelées **constructeur** et **destructeur**.

> **Notion XI.21 : Constructeur d'objet (*Object constructor*)**
>
> Fonction associée à une classe permettant la création et l'initialisation d'un objet en mémoire.

Notion XI.22 : Destructeur d'objet (*Object destructor*)

Fonction associée à une classe permettant la destruction d'un objet en mémoire.

En C++ ou en Java, le constructeur d'un objet est une fonction membre de la classe. Il fait en général appel à une fonction de réservation de mémoire et à des fonctions d'initialisation. Les constructeurs sont normalement définis par le programmeur, mais C++ et Java insèrent un constructeur minimal dans les classes qui n'en possèdent pas. Le nom du constructeur est le nom de la classe. Par exemple, un point origine pourra être défini comme suit :

```
Point Origine(0,0).
```

Le compilateur génère alors automatiquement l'appel au constructeur `Point(0,0)` lors de la rencontre de cette déclaration.

En C++, le destructeur est une fonction membre notée par le nom de la classe précédé de ~ ou appelé explicitement par `delete`. Par exemple, la destruction de l'origine s'effectue par:

```
~Point(0,0).
```

Le destructeur libère la place mémoire associée à l'objet. Il peut être fourni par le programmeur. Certains langages orientés objet tels Java et Smalltalk sont munis d'un ramasse-miettes détruisant automatiquement les objets non référencés, si bien que le programmeur n'a pas à se soucier d'appeler le destructeur d'objets.

Le problème qui se pose dans les SGBDO est d'assurer la persistance des objets sur disques pour pouvoir les retrouver ultérieurement. En effet, constructeur et destructeur d'objets ne font que construire et détruire les objets en mémoire. Une solution couramment retenue pour sauvegarder les objets sur disques consiste à donner un nom à chaque objet persistant et à fournir une fonction permettant de faire persister un objet préalablement construit en mémoire. Cette fonction peut être intégrée ou non au constructeur d'objet. Sa signature pourra être la suivante :

```
// Rendre persistant et nommer un objet pointé.
Oid = Persist(<Nom>, <Ref>);
```

Nom est le nom attribué à l'objet qui permettra ultérieurement de le retrouver (dans le même programme ou dans un autre programme). Ref est la référence en mémoire de l'objet. Oid est l'identifiant attribué à l'objet dans la base par le SGBDO.

Un objet persistant pourra ensuite être retrouvé à partir de son nom, puis monté en mémoire à partir de son identifiant d'objet (adresse invariante sur disque). On trouvera donc deux fonctions plus ou moins cachées au programmeur dans les SGBD à objets :

```
// Retrouver l'oid d'un objet persistant à partir du nom.
```

```
Oid = Lookup(<Nom>);
// Activer un objet persistant désigné par son oid.
Ref = Activate(<Oid>);
```

Un objet actif en mémoire pourra être désactivé (réécriture sur disque et libération de la mémoire) par une fonction du style:

```
// Désactiver un objet persistant actif.
Oid = DesActivate(<Ref>);
```

Finalement, il sera aussi possible de rendre non persistant (détruire) dans la base un objet persistant à partir de son identifiant d'objet ou de son nom, en utilisant l'une des fonctions suivantes :

```
// Supprimer un objet persistant désigné par son nom
Void UnPersist(<Nom>);
// Supprimer un objet persistant désigné par son identifiant
Void UnPersist(<Oid>);
```

Une telle bibliothèque de fonction peut être utilisée directement par le programmeur dans un langage orienté objet comme C++ ou Java pour gérer manuellement la persistance des objets. De plus, des fonctions de gestion de transactions devront être intégrées afin d'assurer les écritures disques, la gestion de concurrence et de fiabilité. Les écritures peuvent être explicites par une fonction `Put` ou implicites lors de la validation, en fin de transaction. Un tel système est voisin de celui offert par Objective-C pour la persistance des objets dans des fichiers. Il constitue un niveau minimal de fonctionnalités nécessaire à la gestion de la persistance que nous qualifierons de **persistance manuelle**. Ce niveau de fonctions peut être caché au programmeur du SGBDO par l'une des techniques étudiées ci-dessous.

Dans tous les cas, un problème qui se pose lors de l'écriture d'un objet dans la base est la sauvegarde des pointeurs vers les autres objets persistants. Comme lors de l'activation d'un objet persistant, il faut restaurer les pointeurs en mémoire vers les autres objets actifs. Des solutions sont proposées par chacune des techniques de persistance décrites ci-dessous. Notez que les systèmes implémentent parfois des techniques de persistance mixtes, qui empruntent un peu aux deux décrites ci-dessous.

3.3 Persistance par héritage

La **persistance par héritage** permet de cacher plus ou moins complètement au programmeur les mouvements d'objets entre la base et la mémoire. L'idée est de profiter de l'héritage pour assurer la persistance automatiquement. Le système offre alors une classe racine des objets persistants, nommée par exemple `PObject` (voir figure XI.14). Cette classe intègre au constructeur et destructeur

d'objets des appels aux fonctions `Persist` et `Unpersist` vues ci-dessus. Ainsi, tout objet appartenant à une classe qui hérite de `PObject` est persistant. En effet, il hérite du constructeur d'objet qui le rend persistant. Il sera détruit sur disque par le destructeur, qui fait appel à la fonction `Unpersist`. La qualité d'être persistant ou non dépend alors du type de l'objet : un objet est persistant si et seulement s'il est du type d'une sous-classe de `PObject`. On dit alors que la persistance n'est pas orthogonale au type.

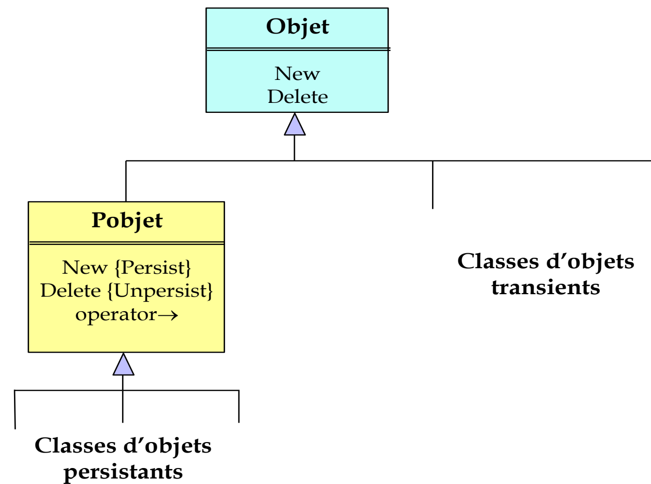


Figure XI.14 —Persistance par héritage

Outre les constructeurs et destructeurs destinés à gérer l'objet sur disque, la technique de persistance par héritage surcharge en général le parcours de référence (opérateur → en C++). Cela permet d'activer automatiquement un objet pointé par un objet déjà actif lors du premier parcours de la référence, en utilisant une technique de mutation de pointeur, comme nous le verrons ci-dessous.

En résumé, nous définirons la persistance par héritage comme suit:

> **Notion XI.23 : Persistance par héritage (*Persistance by inheritance*)**
>
> Technique permettant de définir la qualité d'un objet à être persistant par héritage d'une classe racine de persistance, rendant invisible l'activation et la désactivation des objets.

Cette technique présente l'avantage d'être simple à réaliser. Cependant, elle n'assure pas l'orthogonalité de la persistance aux types de données, si bien que tout type ne peut pas persister. Si l'on veut avoir dans une même classe des objets persistants et transients (par exemple des personnes persistantes et des personnes transientes), on est conduit à dupliquer les classes. Ceci peut être évité en marquant les objets non persistants d'une classe persistante simplement par un booléen. La performance de la technique de persistance par héritage est discutée, la surcharge des opérateurs (constructeurs et parcours de références) pouvant être coûteuse.

3.4 Persistance par référence

Une autre technique possible pour cacher les mouvements d'objets est la **persistance par référence**. L'idée est que tout objet ou variable peut être une racine de persistance à condition d'être déclaré comme tel, puis que tout objet pointé par un objet persistant est persistant (voir figure XI.15). En général, les objets persistants sont les objets nommés, nom et persistance étant déclarés dans la même commande de création d'objet persistant. Cela conduit à ajouter un mot clé « persistant » ou « db » au langage de programmation (par exemple C++) et donc nécessite un précompilateur qui génère les appels aux fonctions `Persist`, `Unpersist`, `Activate`, etc. Par exemple, un employé pourra être créé persistant par la déclaration :

Employe* emp = new persistant Employe(« Toto »);

De même, une simple variable x pourra être rendue persistante par la déclaration :

persistant int x;

Au vu de ces déclarations, le précompilateur générera les commandes de persistance et de recherche nécessaires. Tout objet racine de persistance (donc déclaré persistant) sera répertorié dans un catalogue où l'on retrouvera son nom et son oid.

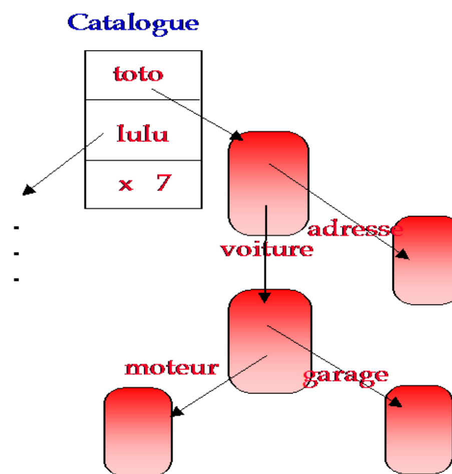


Figure XI.15 — Persistance par référence

Tout objet référencé par un objet persistant sera persistant. Là encore, le pré-compilateur devra assurer la génération des appels aux fonctions de persistance lors des assignations ou des parcours de références. Les références devront aussi être rendues persistantes par l'une des techniques que nous allons étudier ci-dessous.

En résumé, nous définirons la persistance par référence comme suit :

Notion XI.24 : Persistance par référence (*Persistence by reference*)

Technique permettant de définir la qualité d'un objet à être persistant par attribution d'un nom (pour les racines de persistance) ou par le fait qu'il soit référencé par un objet persistant.

Cette technique présente l'avantage de l'orthogonalité de la persistance au type de données, si bien que toute donnée peut être rendue persistante. Elle permet de gérer des graphes d'objets persistants. Ainsi, un arbre d'objet est rendu persistant simplement en nommant la racine. Il est rendu transient en supprimant ce nom.

3.5 Navigation dans une base d'objets

Quelle que soit la technique de persistance, les objets persistants se référencent, et il faut pouvoir retrouver un objet puis naviguer vers les objets référencés.

Notion XI.25 : Navigation entre objets (*Object navigation*)

Parcours dans une base d'objets par suivi de pointeurs entre objets.

Dans les langages objet comme C++, la navigation s'effectue en mémoire par simple décodage de pointeurs. Dans une base d'objets, les choses ne sont pas aussi simples : un objet pointant sur un autre objet est en effet écrit dans la base, puis relu par un autre programme plus tard. L'objet pointé n'est alors plus présent en mémoire (voir figure XI.16). Le problème soulevé est donc de mémoriser de manière persistante les chaînages d'objets sur disques, puis de les décoder en mémoire de manière efficace.

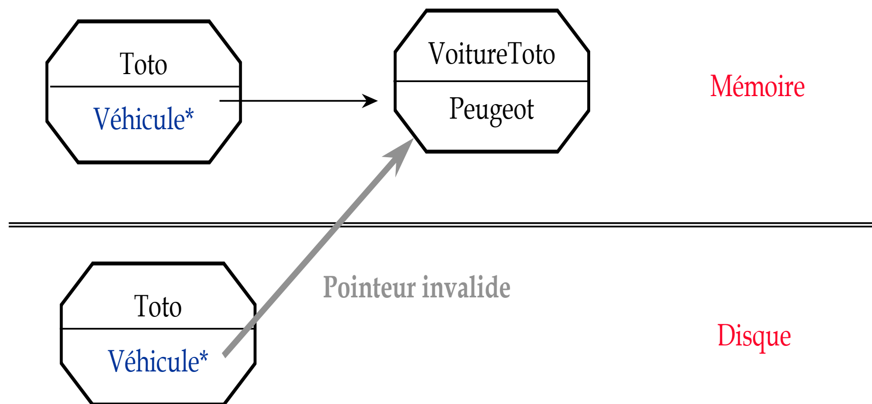


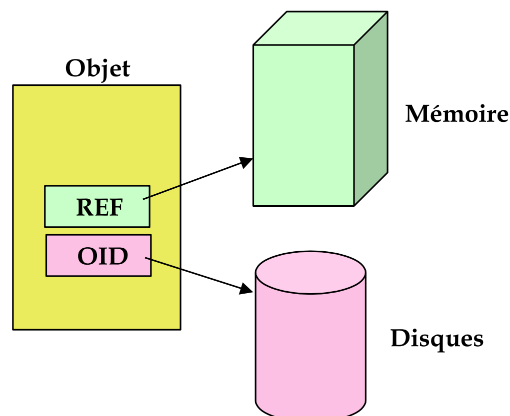
Figure XI.16 — Nécessité de mettre à jour les pointeurs en mémoire

En résumé, il faut pouvoir utiliser des identifiants d'objets comme des pointeurs sur disques et des adresses en mémoire. Le passage d'un type de pointeur à l'autre est appelé **mutation de pointeurs**.

> **Notion XI.26 : Mutation de pointeurs (*Pointer swizzling*)**
>
> Transformation consistant à passer de pointeurs disques à des pointeurs mémoire lors de la première navigation en mémoire via un objet et inversement lors de la dernière.

La mutation de pointeurs disques sous forme d'identifiants à pointeurs en mémoire peut s'effectuer par différentes techniques : utilisation de doubles pointeurs ou utilisation de mémoire virtuelle.

L'**utilisation de doubles pointeurs** consiste à remplacer les références mémoire par des couples `<oid-ref>` (voir figure XI.17). Lors du chargement d'un objet, toutes les parties `ref` des couples sont mises à 0. Lors du parcours d'une référence (opérateur →), si la partie `ref` est à 0, l'objet référencé est activé à partir de son oid et la référence est ensuite chargée avec l'adresse de l'objet en mémoire. Lors des accès suivants, l'adresse mémoire est directement utilisée. Ainsi, un objet référencé par un objet actif (par exemple retrouvé par `lookup`) est activé automatiquement lors de la première traversée du pointeur référençant.



- **En écriture**
  - si OID = NUL ALORS { OID = AllouerBD(Objet) ; Ecrire (REF, OID) } ;
- **En lecture**
  - si REF = NUL ALORS {REF = AllouerMC(Objet); Lire(OID, REF) } ;

Figure XI.17 — L'utilisation de doubles pointeurs

La **technique de mémoire virtuelle** consiste à avoir les mêmes adresses en mémoire centrale et sur disques. Elle a été proposée et brevetée par Object Design, le fabricant du SGBDO ObjectStore. Une référence est une adresse mémoire virtuelle, cette dernière contenant une image exacte de la base de données (ou d'une partition de la base). Lors de la lecture d'un objet, l'adresse mémoire virtuelle composant toute référence est forcée sur une page manquante si l'objet référencé n'est pas en mémoire. La page manquante est réservée en mémoire et marquée inaccessible en lecture et en écriture. Ainsi, lors de la traversée du pointeur, une violation mémoire virtuelle en lecture est déclenchée et récupérée par le SGBDO (voir figure XI.18). Celui-ci retrouve alors la page de l'objet sur disques dans ces tables, lit cette page dans la page manquante, et rend la page

accessible en lecture. Celle-ci peut alors être accédée comme une page normale et l'objet peut être lu. En cas de mise à jour, la violation de page est aussi récupérée pour mémoriser la nécessité d'écrire la page en fin de transaction à la validation, puis la page est rendue accessible en écriture.

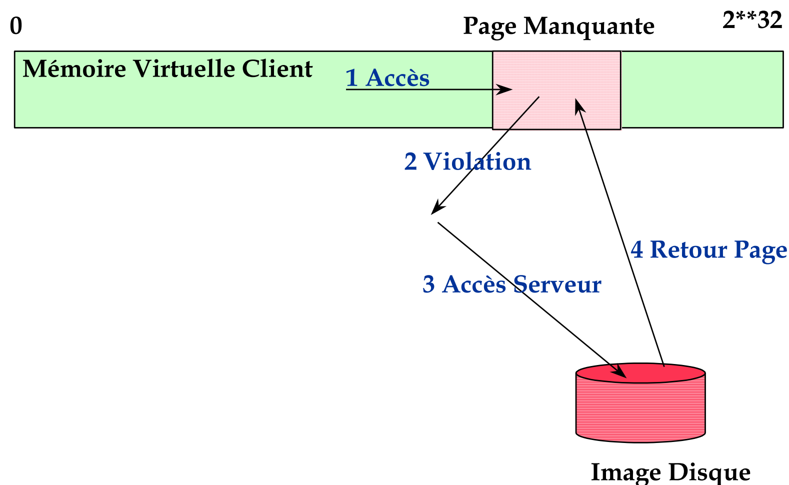


Figure XI.18 — La technique de mémoire virtuelle

Cette technique est parfois appelée **mémoire à un seul niveau** (*Single Level Store*) ; en effet, l'utilisateur travaille directement sur l'image de la base en mémoire. Elle est très efficace lorsqu'un même objet est traversé plusieurs fois et que les objets parcourus sont groupés dans une même page. Cependant, la dépendance du programme à la structure de la mémoire virtuelle peut constituer un inconvénient encore mal mesuré aujourd'hui.

4. ALGEBRE POUR OBJETS COMPLEXES

Les algèbres pour objets complexes résultent d'extensions de l'algèbre relationnelle aux objets complexes. Elles permettent d'exprimer les questions sur une bases de données objet comme des expressions d'opérations élémentaires. Afin d'illustrer la conception orientée objet, nous proposons une structuration de l'algèbre en graphe d'objets, chaque opération étant un objet d'une classe.

4.1 Expressions de chemins et de méthodes

L'algèbre relationnelle permet normalement de référencer les attributs des relations dans les expressions résultats et les expressions de sélection. Ces valeurs de résultats ou de sélections sont extraites des tuples par des **expressions valuables** qui figurent en arguments des opérations de l'algèbre. La notion d'expression valuable recouvre traditionnellement les références aux attributs, les constantes et les fonctions arithmétiques en relationnel. Par exemple, dans une table Produits, il est possible de sélectionner le prix toutes taxes comprises des produits par l'expression valuable PRIX*(1+TVA).

Dans un environnement objet, il est nécessaire de pouvoir appliquer des opérations sur les objets et aussi de pouvoir référencer les identifiants, par exemple afin de

parcourir les associations. On arrive ainsi naturellement à généraliser la notion d'expression valuable. Une première généralisation telle que proposée ici a été effectuée par [Zaniolo85]. Deux types d'expressions valuables nouvelles sont nécessaires : les **expressions de chemins** et les **expressions de méthodes**.

> **Notion XI.27 : Expression de chemin (*Path Expression*)**
>
> Séquence d'attributs de la forme $A_1.A_2…A_n$ telle que chaque attribut A_i de la classe Ci référence un objet de la classe C_{i+1} dont le suivant est membre, à l'exception du dernier.

Une expression de chemin permet d'effectuer un parcours dans le graphe des associations de classes. Sur la base de la figure XI.12, Représente.Tête.Centre.x est une expression de chemins : partant d'un buveur B, elle permet d'atteindre un réel représentant l'abscisse du cente de la tête, en traversant les classes Caricature Cercle et Point. De telles expressions sont à la fois utilisables en algèbre d'objets complexes et en SQL étendu, comme nous le verrons dans les chapitres suivants. Dès que l'on rencontre un chemin multivalué, la traversée devient ambiguë ; par suite, la plupart des langages interdisent les **expressions de chemins multivaluées** du style Abus.cru sur la base de la figure XI.12, car bien sûr un buveur boit plusieurs vins.

> **Notion XI.28 : Expression de méthodes (*Method Expression*)**
>
> Séquence d'appels de méthodes de la forme $M_1.M_2…M_n$, avec d'éventuels paramètres pour certaines méthodes M_i de la forme $M_i(P_1, P_2, …P_j)$.

Une expression de méthodes permet en principe d'appliquer des méthodes à un objet, l'objet sélectionné étant l'argument distingué permettant l'envoi du message correspondant à la méthode. Le polymorphisme doit être mis en œuvre pour sélectionner le bon code de la méthode. Par exemple, `Travailler(10)` est une expression de méthode dont le code sera différent selon que l'employé est un cadre ou non (voir figure XI.12).

Il est possible de généraliser les expressions de méthodes afin de les appliquer aussi à des valeurs : on obtient alors des **expressions fonctionnelles** de la forme $F_i(P_1, P_2, …P_j)$ s'appliquant sur des éléments (objets ou valeurs) de type Ti et retournant des éléments de type Ti+1 sur lesquels on peut à nouveau appliquer des opérations définies sur le type Ti+1. Une expression de chemins est alors un cas particulier où la fonction consiste à traverser le pointeur. Il est aussi possible de généraliser aux expressions fonctionnelles multivaluées, en tolérant en résultat d'une fonction une collection. En résumé, la figure XI.19 présente les différents types d'expressions valuables dans un environnement objet.

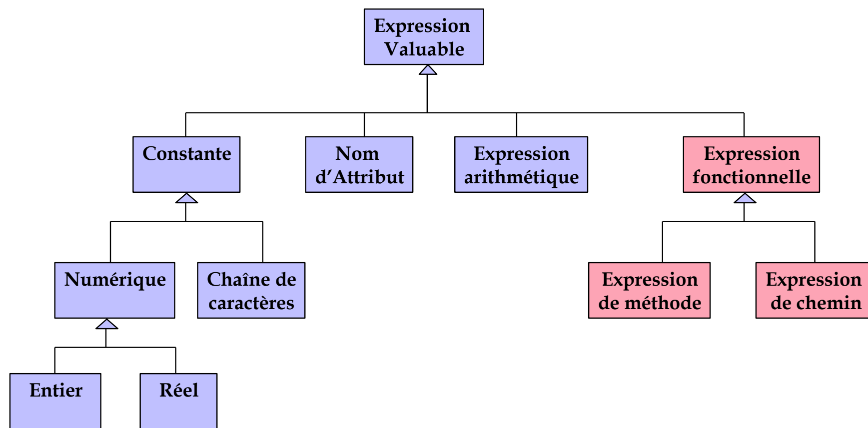


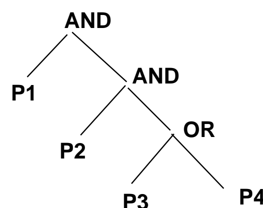
Figure XI.19 — Différents types d'expressions valuables

A partir des expressions valuables généralisées, il est possible de spécifier une qualification de restriction ou jointure généralisée, capable de traiter des objets complexes. Une telle qualification peut être vue comme un arbre ET-OU de prédicats élémentaires (voir figure XI.20). Un prédicat élémentaire est de la forme :

```
<Prédicat élémentaire> ::= <Expression valuable>

[<Comparateur> <Expression valuable>].
```

Un comparateur est choisi parmi $\{=, <, >, \geq, \leq, \neq\}$. Le comparateur et la deuxième expression valuable ne sont pas nécessaires si la première expression valuable est de type booléen (par exemple, une méthode booléenne telle Contains).



((P3 OU P4) AND P2) AND P1

Figure XI.20 — Graphe ET-OU de prédicats élémentaires

4.2 Groupage et dégroupage de relations

Outre les opérations classiques de l'algèbre relationnelle, les algèbres d'objets complexes généralisent les opérations issues des agrégats et introduites dans les modèles non en première forme normale, c'est-à-dire avec des attributs multivalués ensemblistes. Les opérations de groupement comportent le **groupage** (*nest* en anglais) et le **dégroupage** (*unnest*). Ces deux opérations ont à l'origine été définies pour des relations comme suit (nous les étendrons aux objets plus loin) :

Notion XI.29 : Groupage (*Nest*)

Opération transformant une relation en créant pour chaque valeur des attributs de groupement un ensemble de valeurs des attributs groupés.

Cette opération, notée ν dans certaines extensions de l'algèbre relationnelle, fait donc apparaître des attributs à valeur dans des ensembles. Elle est illustrée figure XI.21. Une définition plus générale pourrait consister à grouper la relation selon un schéma hiérarchique des attributs : on pourrait ainsi faire plusieurs groupages en une seule opération. Dans le monde objet, cette opération peut être appliquée à une collection pour générer une nouvelle collection, avec pour chaque objet obtenu par groupage, attribution d'un nouvel identifiant.

L'opération de dégroupage est l'opération inverse (voir figure XI.21). Attention il n'est pas toujours vrai qu'un dégroupage après un groupage donne la relation initiale, en particulier si cette dernière a des doubles.

Notion XI.30 : Dégroupage (*Unnest*)

Opération transformant une relation à attributs groupés en relation plate, créant pour cela un tuple pour chaque valeur du groupe en dupliquant les valeurs des autres attributs.

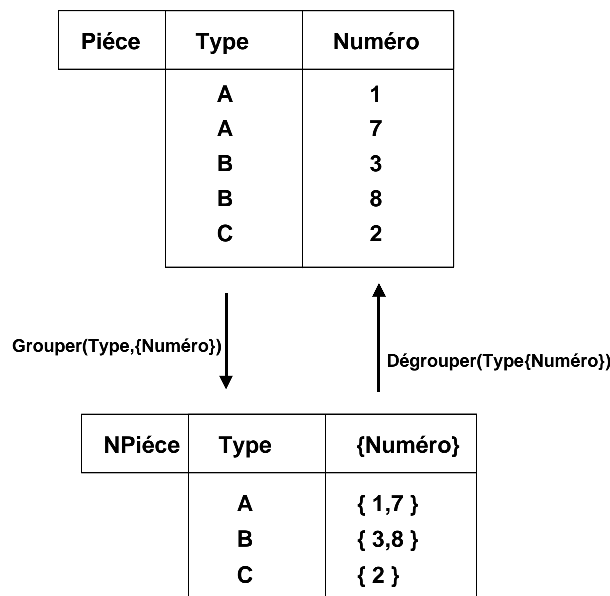


Figure XI.21 — Exemple d'opérations de groupage et dégroupage

4.3 L'algèbre d'Encore

Cette algèbre a été introduite dans le cadre du projet de SGBDO Encore par Shaw et Zdonik [Shaw90]. Elle apporte les concepts objets au sein d'une algèbre proche de l'algèbre relationnelle. Cette algèbre supporte les types abstraits et les identifiants d'objets. Les opérations accèdent des collections typées d'objets en

invoquant l'interface publique du type. Pour cela, l'algèbre utilise des expressions d'opérations notées f_i et des prédicats construits à partir de ces expressions notés p.

Les opérations sont les suivantes :

- La **sélection d'objets** dans une collection d'entrée par un prédicat p est définie comme suit : $Select(Collection, p) = \{s \mid (s \in Collection) \wedge p(s)\}$. Le résultat est donc un ensemble d'identifiants d'objets indiquant ceux qui satisfont au prédicat p.
- L'**image d'une collection** par une expression fonctionnelle f de type T est définie par : $Image(Collection, f : T) = \{f(s) \mid s \in Collection\}$. Il s'agit donc de l'ensemble des objets résultants de l'application de f à ceux de la collection.
- La **projection d'une collection** par une famille d'expressions fonctionnelles f1, f2, …fn sur un tuple $\langle A1, A2, \ldots An \rangle$ est définie par : $Project(Collection, \langle (A1, f1), ...,(An, fn) \rangle) = \{\langle A1 : f1(s), ...,An : fn(s) \rangle \mid (s \in Collection)\}$. Chaque objet donne donc naissance à un tuple, alors que chaque objet donnait naissance à élément simple dans le cas de l'image.
- Le **groupage d'une collection** de tuples sur un attribut Ai est défini par $Nest(Collection,Ai) = \{\langle A1 : s.A1, ..., Ai : t, ...,An : s.An \rangle \mid \forall r \exists s (r \in t \wedge s \in Collection \wedge s.Ai = r)$. L'opération remplace la collection d'entrée par un ensemble de tuples similaires pour les attributs autres que Ai, mais groupant dans un ensemble les valeurs de Ai correspondant aux tuples identiques pour les autres attributs. Cette opération introduit dans le monde objet le groupage relationnel.
- Le **dégroupage d'une collection** de tuples sur un attribut Ai est défini par : $UnNest(Collection,Ai) = \{\langle A1 : s.A1, ...,Ai : t, ...,An : s.An \rangle \mid s \in InputCollection \wedge t \in Ai\}$. C'est l'opération inverse du groupage.
- L'**aplatissage d'une collection** est utilisé pour restructurer des collections de collections. Il est défini par : $Flatten(Collection) = \{r \mid \exists\, t \in Collection \wedge r \in t\}$.
- La **jointure de collections** sur un prédicat p est une transposition simple de la jointure par valeur du relationnel : $OJoin(Collection1, Collection2,A1,A2,p) = \{\langle A1 : s, A2 : r \rangle \mid s \in Collection1 \wedge r \in Collection2 \wedge p(s,r)\}$.

Afin d'éliminer les doubles, une opération d'**élimination de duplicata** est introduite. Elle est notée DupEliminate (Collection, e). e est un paramètre permettant de définir l'égalité d'objets dans la collection. Ce peut être par exemple l'égalité d'identifiants ou de valeurs. Coallesce (Collection, A_i, e) est une variante permettant d'éliminer les doubles dans un attribut A_i d'un tuple avec attribut multivalué.

Les opérations ensemblistes classiques d'**union**, **intersection** et **différence** de collections sont à considérer. Elles nécessitent la définition du type d'égalité d'objets considéré, sur identifiant ou sur valeur.

4.4 Une algèbre sous forme de classes

Examinons maintenant l'algèbre LORA dérivée de celle utilisée dans un SGBD objet-relationnel construit à l'INRIA au début des années 90 [Gardarin89]. Cette algèbre s'appuie aussi sur des expressions valuables permettant de construire des qualifications et prédicats spécifiés figure XI.22.

```
Class Qualification { // Connexion logique de prédicats ET de OU
        Connecteur Node; // Connecteur logique AND, OR, NIL
        Predicat* Primaire; // Prédicat élémentaire
        Qualification* QualifDroite; // Reste de la qualification
};

Class Predicat { // Critère de comparaison élémentaire
        Expression* Droite; // Expression droite
        Comparateur Comp; // Comparateur =, <, ≤, >, ≥, ≠
        Expression* Gauche; // Expression gauche
};
```

Figure XI.22 — Spécifications en C++ des qualifications

L'algèbre distingue les opérations par valeur et par référence, dont l'argument est en général un ou plusieurs identifiants. Les opérations sont classées en opérations de recherche, opérations ensemblistes, opérations de groupement et enfin opérations de mise à jour. La figure XI.23 représente la hiérarchie de généralisation des opérations de l'algèbre proposée.

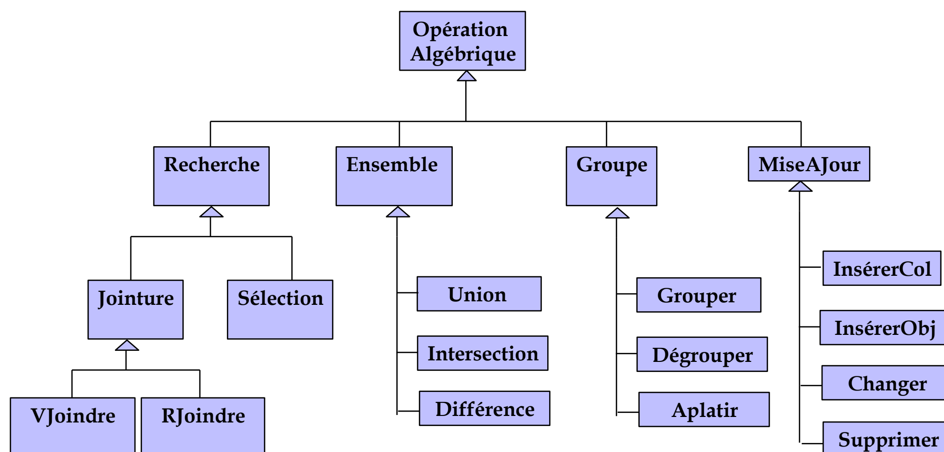


Figure XI.23 — Les différentes opérations de l'algèbre

4.4.1 Les opérations de recherche

Nous distinguons les opérations élémentaires de recherche suivantes :

- **Sélection** (`Filter`) correspond à l'application d'une qualification aux objets d'une collection et à la projection à l'aide d'expressions construites à partir des attributs en résultat ; elle équivaut à `Project` ou `Image` de l'algèbre Encore, qui sont ici regroupées en une seule opération.
- **Jointure par valeur** (`VJoindre`) permet de filtrer le produit cartésien de deux collections à l'aide d'une expression de qualification classique. Elle est réalisée par l'`Ojoin` de l'algèbre Encore.
- **Jointure par référence** (`RJoindre`), où le critère est une expression de chemin mono- ou multivaluée, aussi réalisable par `Ojoin` avec Encore.

L'opération générique de recherche permet d'effectuer plusieurs opérations élémentaires à partir d'une qualification générale et d'une expression de résultats (liste d'expressions). La figure XI.24 donne une spécification indicative en C++ de la classe Opération et des sous-classes définissant les opérations de recherche. Chaque sous-classe d'opérations de recherche correspond à un type particulier de qualification.

```
class Opération {
        // Options de tri et dédoublement des résultats:
        LIST(Field*) SortInfo; // Champs de tri.
        Boolean Ascendant; // Ascendant si Vrai.
        Boolean Duplicata; // Garder les doubles si vrai.
};

class Recherche : public Opération {
        // *** output = SEARCH (input-list, qualif, resul [,sort-dup])
        Qualification* Critère; // Qualification générale.
        LIST(Expression) Résultat; // Expressions sélectionnées.
};

class Selection : public Recherche {
        // *** output = FILTER (input-list, qualif, resul [,sort-dup])
        // Le critère est une qualification simple ou de méthodes};

class Jointure : public recherche { } ;
class RJoindre : public Jointure {
 // *** result = RJOINDRE (input-list, qualif-ref, resul [, sort-dup])
 // Le critère est une expression de chemins référence.
};
```

```
class VJoindre : public Jointure {
 // *** result = VJOINDRE(input, input, qualif-join, resul [,sort-dup])
 // Le critère est un prédicat de jointure.
};
```

Figure XI.24 — La classe des opérations de recherche

4.4.2 Les opérations ensemblistes

Les opérations ensemblistes permettent de réaliser l'union, l'intersection et la différence de deux ensembles d'objets (en principe des instances de classes). Les objets peuvent être repérés par leurs identifiants (identité d'objets) ou par leurs valeurs (égalité d'objets). Selon que les opérations comparent les identifiants ou les valeurs, on obtient deux résultats en général différents. Il faut donc spécifier l'égalité d'objets utilisée en paramètre des opérations ensemblistes, ce qui est fait au niveau de la classe Assembler (voir Figure XI.25).

```
class Ensemble : public Operation {
        Egalité Enum {identifiant; valeur}; // Type d'égalité
        Field* Identifiant; // Référence au champ identifiant.
};
```

```
class Union : public Ensemble{
        // *** output = UNION (input-list, [,sort-dup])
};
class Intersect : public Ensemble{
        // *** output = INTERSECT (input-list [,sort-dup])
};
class Difference : public Ensemble{
        // *** output = DIFFERENCE (input-list [,sort-dup])
};
```

Figure XI.25 — Les classes des opérations ensemblistes

4.4.3 Les opérations de mise à jour

Les opérations de mise à jour comportent les insertions, les suppressions et les modifications. Nous distinguons deux types d'insertion : insertion à partir d'une collection temporaire (InsérerCol) et insertion d'une liste d'objets fournis par valeurs (InsérerObj). Ces deux opérations sont définies figure XI.26.

```
class InsertCol : public MiseAJour{
        // *** input1 = INSERT_OBJ (input1, input2 [, sort-dup])
        // La 2e collection contient les objets à insérer dans la première.
        // La 1e collection est une collection de base.
        // La 2e est une collection calculée.
};

class InsertObj : public MiseàJour{
        // *** input = INSERT_VAL (input, Object_list [,sort-dup])
        LIST(Expression) Objet[MaxObj]; // Objets à insérer.
};
```

Figure XI.26 — Les classes des opérations d'insertion

Les suppressions s'effectuent à partir d'une collection calculée qui contient les identifiants des tuples à supprimer dans l'autre collection. La suppression est équivalente à une différence sur identifiant d'objets. Elle est définie figure XI.27.

```
class Supprimer : public MiseàJour{
 // *** input1 = SUPPRIMER (input1, input2, identifiant [, sort-dup])
 // La 2e collection contient les objets à supprimer de la première.
 // La 1e collection est une collection de base.
 // La 2e collection est une collection calculée.
 Field* identifiant; // Référence au champ identifiant.
};
```

Figure XI.27— La classe des opérations de suppression

Les modifications s'effectuent aussi à partir d'une extension de classe calculée qui contient les identifiants de tuples à modifier dans une classe de base et les éléments pour calculer les nouvelles valeurs des champs modifiés. Ces éléments sont exprimés, pour chaque attribut, sous la forme d'une expression de calcul (par exemple A = A *1,1 pour une augmentation de 10% de A). La figure XI.28 définit plus précisément l'opération Changer.

```
class Changer : public MiseAJour{
	// *** input1 = UPDATE (input1, input2 [, sort-dup])
	// La 2e collection contient les objets à changer dans la première.
	// La 1e collection est une collection de base.
	// La 2e collection est une collection calculée.
	Field* identifiant; // Référence au champ identifiant.
	LIST(Field*) Achanger; // Champs à modifier.
	LIST(Expression) Calcul; // Mode de calcul des modifs.
};
```

Figure XI.28 — La classe des opérations de modification

4.4.4 Les opérations de groupe

Les opérations restant à spécifier sont celles de groupage correspondant aux Nest, Unest et Flatten de l'algèbre Encore. La figure XI.29 spécifie plus précisément les opérations de groupage, dégroupage et aplatissage en termes de classes C++.

```
class Grouper : public Groupe {
	// *** output = NEST (input, nest_exp, result-expression)
	// Le groupage est effectué sur un seul groupe d'attributs.
	// Une expression résultat est applicable aux champs groupés.
	// L'expression résultat doit s'appliquer à une collection.
	LIST(Field*) Base; // Attributs pour le partitionnement.
	LIST(Field*) Groupé; // Attributs à grouper.
	LIST(Expression) Résultat; // Résultats à fournir.
};

class Degrouper : public Groupe {
	// *** output = UNNEST (input, unnest_exp, result-expression)
	// Le dégroupage est effectué sur un seul groupe d'attributs.
	// Une expression résultat est applicable aux champs dégroupés.
	LIST(Field*) Base; // Attributs à dupliquer.
	LIST(Field*) Dégroupé; // Attributs à dégrouper.
	LIST(Expression) Résultat; // Résultats à fournir.
};
```

```
class Aplatir : public Groupe {
        // *** output = FLATTEN (input, flatten_exp)
        // La restructuration est faite sur un seul attribut
        Field* Aplatit; // Attributs à désimbriquer de 1 niveau.
} ;
```

Figure XI.29 — Les classes d'opérations de groupage et dégroupage

4.4.5 Arbres d'opérations algébriques

Finalement, comme avec le modèle relationnel, une question peut être représentée par une expression d'opérations de l'algèbre d'objets complexes. L'expression peut être visualisée sous la forme d'un arbre. Pour être plus général, en particulier pour permettre le partage de sous-arbre et le support de boucles, il est souvent permis à chaque opération d'avoir plusieurs flots d'entrées et surtout plusieurs flots de sorties. Un flot correspond à une collection temporaire qui peut être matérialisée ou non, selon la stratégie d'évaluation du système.

Un exemple simple de graphe d'opérations est illustré figure XI.31. Il représente une question portant sur une base de données qui décrit des véhicules référençant des constructeurs automobiles référençant eux-mêmes leurs employés directeurs de divisions (voir figure XI.30). Soit la question « rechercher les numéros des véhicules de couleur rouge dont le fabriquant est de Paris et a un directeur au moins de moins de 50 ans ». Le graphe associé est un graphe possible parmi les nombreux graphes d'opérations permettant d'exécuter cette question. Il est très proche d'un graphe relationnel. Nous étudierons son optimisation au chapitre sur l'optimisation de requêtes objet.

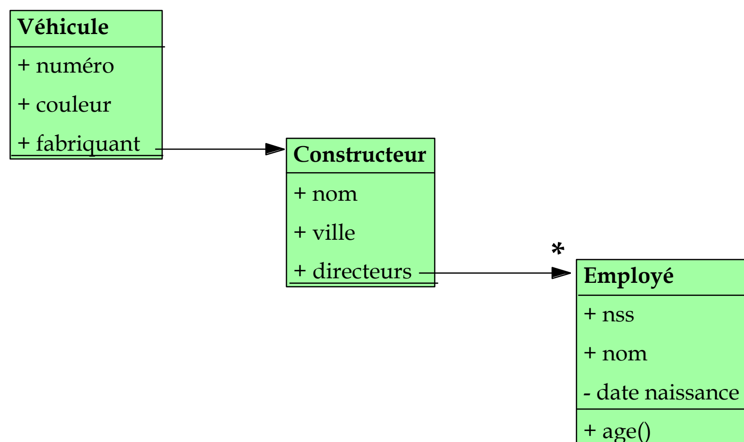


Figure XI.30 — Schéma de la base Véhicule, Constructeur, Employé

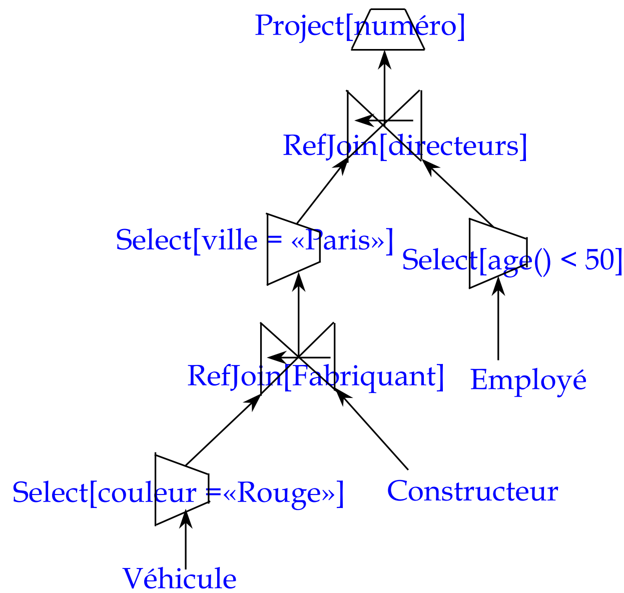


Figure XI.31 — Exemple de graphe d'opérations

Pour terminer la modélisation de l'algèbre en C++, il faut maintenant spécifier la classe dont les graphes d'expressions algébriques sont les instances. La figure XI.32 modélise un graphe d'opérations par deux classes en C++, l'une correspondant aux nœuds (Nœud) et l'autre aux arcs (Flot). Nous avons choisi de représenter chaque arc par un objet de la classe Flot, cela afin de pouvoir étiqueter les arcs, par exemple par un volume de données estimées circulant sur l'arc. Cela permettrait d'approcher les volumes de données traités et les coûts d'opérations, comme nous le verrons au chapitre sur l'optimisation de requêtes.

```
Class Noeud {
        Operation* Oper; // Operation à effectuer en ce nœud.
        Flot* Input[MaxIn]; // collections d'entrée (fils).
        Flot* Output[MaxOut]; // collections de sortie (parents).
};

Class Flot {
        Int TailleEst; // Taille du flux estimée.
} ;
```

Figure XI.32 — Classes modélisant un graphe d'opérations

5. CONCLUSION

Dans ce chapitre, nous avons présenté les concepts de modélisation introduits par l'orientation objet, les techniques de gestion de la persistance des objets et une algèbre d'objets complexes. Ces concepts et techniques constituent l'essentiel des fonctionnalités aujourd'hui offertes par les SGBDO par le biais du langage de

définition d'objets, du langage de requêtes et du langage de manipulation d'objets. Au-delà, il est important de mentionner que les SGBDO offrent pour la plupart des environnements de développements visuels élaborés. Ceux-ci permettent de parcourir les graphes de généralisation et de composition (agrégation et association) de classes, mais aussi de visualiser les objets, voire de créer de nouvelles classes et de programmer des méthodes. Ces éditeurs évolués (en anglais, *browsers*) sont un des attraits importants des SGBDO : en exploitation par exemple, ils autorisent la vision des classes d'objets sous forme d'icônes clicables pour déclencher les méthodes associées. Ces méthodes peuvent d'ailleurs être des opérations de recherche ou de mise à jour proches de celles de l'algèbre d'objets.

Outre ceux étudiés dans ce chapitre, les bases de données à objets soulèvent de nombreux problèmes difficiles. Les plus cruciaux sont sans doute ceux d'architecture et de performance. Quelle architecture client-serveur retenir ? Faut-il plutôt des serveurs d'objets ou de pages [Dewitt90] ? Comment optimiser les performances, en gérant des caches d'objets, en utilisant des techniques de type mémoire virtuelle d'objets, en développant des méthodes de regroupement des objets souvent accédés ensemble, etc. ? Du point de vue du langage de requêtes étendu aux objets, nous allons voir que deux propositions de standards s'opposent quelque peu. Les techniques d'optimisation sont encore mal maîtrisées en pratique. Un autre problème important est celui posé par les modifications de schémas : comment éviter de décharger la base et recompiler les méthodes, par exemple lors d'ajout de super-classes ou de suppression de sous-classes ? Une solution est sans doute la gestion de versions d'objets et de schémas [WonKim90]. Les problèmes de concurrence en présence de transactions longues (une transaction de conception peut durer plusieurs heures) sont eux aussi cruciaux. Tous ces problèmes (notre liste n'est malheureusement pas exhaustive) seront étudiés dans les chapitres qui suivent.

Sous sa forme pure ou sous forme intégrée au relationnel, l'objet constitue sans doute la voie d'avenir pour les bases de données, comme pour bien d'autres domaines. Des standards de représentation se développent au niveau des applications selon les techniques de modélisation orientée objet, en particulier les standards XML, SGML, EXPRESS et CMIS/CMIP respectivement pour le WEB, la gestion de données techniques, la CAO et l'administration de réseaux. Ces langages de modélisation de documents ou de composants semblent bien adaptés aux bases de données à objets. L'approche objet reste cependant limitée, car elle nécessite de bien connaître les objets pour en définir le type. Fondé sur un typage fort, l'objet est peu adapté pour aborder les applications à données faiblement structurées, comme le Web. Des extensions sont nécessaires.

6. BIBLIOGRAPHIE

[Abiteboul95] Abiteboul S., *Foundations of databases*, Addison-Wesley, Reading, Mass., 1995.

Ce livre présente les fondements théoriques des bases de données en faisant le lien avec des domaines de recherche connexes, tels que la logique et la complexité. Il fait le point sur les problèmes avancés des bases de données, notamment sur le couplage des modèles déductifs et objets.

[Abiteboul87] Abiteboul S., Beeri C., « On the Power of Languages for the Manipulation of Complex Objects », *International Workshop on Theory and Applications of Nested Relations and Complex Objects*, 1987, aussi rapport INRIA N° 846, Paris, mai 1988.

Cet article présente une vue d'ensemble théorique mais progressive des algèbres pour objets complexes. Il discute de la puissance des langages résultants.

[Arnold96] K. Arnold, J. Gosling, *Le Langage Java*, International Thomson Publishing France, Traduction de *The Java Programming Language* par S. Chaumette et A. Miniussi, Addison Wesley Pub., 1996.

Ce livre est la référence sur le langage Java, par les inventeurs. Il inclut une brève introduction au langage et une présentation détaillée des commandes, constructions et bibliothèques. Sont couverts les aspects définition de classes et d'interfaces, traitement des exceptions, multitâche, package, classes systèmes et bibliothèques.

[Atkinson89] Atkinson M., Bancilhon F., DeWitt D., Dittrich K., Maier D., Zdonick S., « The Object-Oriented Database System Manifesto *», Deductive and Object-Oriented Databases Int. Conf.*, Kyoto, Japan, 1989.

Le fameux manifesto pour les bases de données pures objet. Les caractéristiques obligatoires et optionnelles des SGBDO sont précisées comme vu ci-dessus.

[Banerjee87] Banerjee J., W. Kim, H.J. Kim, H.F. Korth., « Semantics and Implementation of Schema Evolution in Object-Oriented Databases », *ACM SIGMOD Int. Conf.*, San Fransisco, Cal., 1987.

Cet article pose les problèmes de modification de schémas dans les bases de données objets: suppression ou addition d'attributs, de méthodes, de sur-classes, etc. Les solutions retenues dans ORION, qui permettent une grande souplesse à condition de respecter des règles précises (par exemple, pas de cycle de généralisation), sont présentées.

[Bouzeghoub85] Bouzeghoub M., Gardarin G., Métais E., « SECSI: An Expert System for Database Design », *11th Very Large Data Base International Conference*, Morgan Kaufman Pub., Stockolm, Suède, 1985.

Cet article décrit le système SECSI basé sur un modèle sémantique appelé MORSE. MORSE supporte l'agrégation, la généralisation, l'association et l'instanciation. SECSI est construit selon une architecture système expert. C'est un outil d'aide à la conception de bases de données relationnelles qui transforme le modèle sémantique en relations normalisées. Le modèle sémantique est élaboré à partir de langages quasi naturels, graphiques ou de commande.

[Bouzeghoub91] Bouzeghoub M., Métais E., « Semantic Modelling of Object Oriented Databases », *17th Very Large Database International Conference*, Morgan Kaufman Pub., Barcelone, Espagne, août 1991.

Cet article propose une méthodologie de conception de bases de données à objets, fondée sur un réseau sémantique. L'application est spécifiée par un langage de haut niveau bâti autour d'un modèle sémantique et permet de définir des contraintes d'intégrité et des règles de comportements. Cette approche est intégrée dans la version objet du système d'aide à la conception SECSI.

[CACM91] Communication of the ACM, « Special Issue on Next-Generation Database Systems », *Communication of the ACM*, Vol. 34, N° 10, octobre 1991.

Ce numéro spécial des CACM présente une synthèse des évolutions des SGBD vers une nouvelle génération. Les produits O2 commercialisé par O2 Technology, ObjectStore commercialisé par Object Design, GemStone commercialisé par Servio, et les prototypes Postgres de l'universté de Californie Berkeley et Starbust du centre de recherche d'IBM à Alamaden sont décrits en détail.

[Cardelli84] Cardelli L., Wegner P., « On Understanding Types, Data, Abstraction, and Polymorphism », *ACM Computing Surveys*, Vol. 17, N° 4, décembre 1985.

Vaste article de synthèse sur le typage, les abstractions de type et le polymorphisme. Les conditions de typage sûr, c'est-à-dire vérifiable à la compilation, sont étudiées.

[Castagna96] Castagna G., *Object-Oriented Programming – A Unified Foundation*, 366p., Birkhaäuser, Boston, 1997.

Ce livre développe une théorie de l'orientation objet, plus spécialement du polymorphisme, qui couvre les méthodes multiclasses. Il apporte un nouvel éclairage au problème du typage des paramètres des méthodes dans les cas de surcharge et redéfinition. En clair, la nouvelle théorie en vogue dans le monde objet.

[Cattell91] Cattell R.G. , « The Engineering Database Benchmark », in [Gray91].

Article présentant les résultats du premier benchmark comparant bases de données à objets et relationnelles. Les résultats démontrent la supériorité des SGBD à objets pour les parcours de graphes.

[Cluet90] Cluet S., Delobel C., Lécluse C., Richard P., « RELOOP, an Algebra Based Query Language for an Object-Oriented Database System », *Data & Knowledge Engineering,* Vol. 5, N° 4, octobre 90.

Une présentation du langage d'interrogation du système O2. La sémantique du langage est basée sur une algèbre étendue. Bien que possédant une syntaxe particulière, le langage est d'un point de vue sémantique proche d'un SQL étendu supportant des objets complexes.

[Delobel91] Delobel C., Lécluse Ch., Richard Ph., *Bases de données : des systèmes relationnels aux systèmes à objets*, 460 pages, InterEditions, Paris, 1991.

Une étude très complète de l'évolution des SGBD, des systèmes relationnels aux systèmes à objets, en passant par les systèmes extensibles. Une attention particulière est portée sur les langages de programmation de bases de données. Le livre décrit également en détail le système O2, son langage CO2 et les techniques d'implémentation sous-jacentes. Un livre en français.

[Dewitt90] DeWitt D., Futtersack P., Maier D., Velez F., « A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems », *16th Very Large Database International Conference*, Morgan Kaufman Pub., Brisbane, Australie, août 1990.

Une étude comparative de trois architectures (serveur d'objets, de pages et de fichiers) client-serveur pour les SGBDO. L'analyse démontre sommairement que l'approche serveur de pages est plus performante sous certaines conditions dans un contexte mono-utilisateur.

[Gardarin89] G. Gardarin, J. Kiernan, J.P. Cheiney, D. Pastre, « Managing Complex Objects in an Extensible Relational DBMS » , *15th Very Large Databases International Conference*, Morgan & Kaufman Ed., Amsterdam, pp. 55-65, Août 1989.

Cet article présente le SGBD Sabrina réalisé à l'INRIA de 1988 à 1990, qui fut un des premiers SGBD relationnels à intégrer l'objet. Les objets étaient définis comme des types abstraits dont les opérations étaient programmées en LeLisp ou en C. Ils étaient intégrés au relationnel comme valeurs de domaines des tables. Ce SGBD fut commercialisé par une start-up qui fut malheureusement plus soutenue après 1988, les pouvoirs publics préférant une démarche objet pure.

[Gardarin94] G. Gardarin, M. Nowak, P. Valduriez, « Flora : A Functional-Style Language for Object and Relational Algebra », *5th DEXA (Database and Expert System Application) Intl. Conf.*, Athens, in LNCS N° 856, pp. 37-46, Sept. 1994.

FLORA est un langage fonctionnel permettant d'écrire des plans d'exécution résultant de la compilation de requêtes objet. Il est du même niveau qu'une algèbre d'objets complexes, mais basé sur une approche fonctionnelle. FLORA manipule une riche bibliothèque de collections.

[Goldberg83] Goldberg A., Robson D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

Le livre de référence de Smalltalk, par les inventeurs du langage.

[Gray91] Gray J. Ed., *The Benchmark Handbook*, Morgan & Kaufman Pub., San Mateo, 1991.

Le livre de base sur les mesures de performances des SGBD. Composé de différents articles, il présente les principaux benchmarks de SGBD, en particulier le fameux benchmark TPC qui permet d'échantillonner les performances des SGBD en transactions par seconde. Les conditions exactes du benchmark définies par le "Transaction Processing Council" sont précisées. Les benchmarks de l'université du Madisson, AS3AP et Catell pour les bases de données à objets, sont aussi présentés.

[Guogen78] Guogen J., « An Intial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types », *Current Trends in Programming Methodology*, Vol. 4, Prentice-Hall 1978, L. Yeh Ed., 1978.

Cet article propose une formalisation des types abstraits de données comme des sigma-algèbres multisortes. Toutes les fonctions sont modélisées comme des transformations de sortes. Cela permet de spécifier chaque fonction par des axiomes. Par exemple, une pile p munie des fonctions PUSH et POP doit obéir à l'axiome PUSH(POP(p)) = p. Un ensemble d'axiomes permet de spécifier complétement un type abstrait. Un des articles de base sur la théorie des types abstraits de données.

[Guttag77] Guttag J., « Abstract Data Types and the Development of Data Structures », *Comm. of ACM*, Vol.20, N° 6, juin 1977.

Cet article montre comment on peut spécifier l'implémentation de types abstraits en fonction d'autres types abstraits et comment ces implémentations peuvent être cachées à l'utilisateur (principe d'encapsulation). Des idées sur le contrôle des spécifications et des implémentations sont aussi proposées. Cet article plutôt pratique développe les principes des types abstraits.

[Hammer81] Hammer M., McLeod D., « Database Description with SDM: A Semantic Database Model », *ACM TODS*, Vol.6, N° 3, septembre 1981.

Un des premiers modèles sémantiques proposés en bases de données ; les notions d'agrégation, généralisation, abstraction sont notamment introduites dans un modèle à base de graphe. Ce modèle a été implémenté sur les systèmes UNISYS, mais le SGBD résultant n'a malheureusement pas eu un grand succès, bien qu'il fût un précurseur des SGBD à objets.

[Hose91] Hose D., Fitch J., « Using C++ and Operator Overloading to Interface with SQL Databases », *The C++ Journal*, Vol. 1, N° 4, 1991.

Cet article présente une intégration de SQL à C++. Des classes Base, Table, Colonne et Curseur sont définies. Les requêtes sont formulées dans une syntaxe proche de C++, mais aussi conforme à SQL. Elles sont traduites en requêtes soumises au SGBD interfacé. Le produit résultant nommé CommonBase s'interface avec SYBASE, ORACLE, INGRES, etc.

[Khoshafian86] Khoshafian S., Copeland G., « Object Identity », *OOPSLA Intl. Conf.*, Portland, Oregon, 1986, also in [Zdonik90].

Une discussion de l'identité d'objets : un identifiant est un repère qui distingue un objet d'un autre objet, indépendamment de son état. Les différents types d'égalité et le partage de sous-objets sont introduits dans cet article de référence.

[Lécluse89] Lécluse C., Richard Ph., « The O2 Database Programming Language », *15th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

La description du langage du système O2 réalisé à l'INRIA au sein du GIP Altaïr. Ce langage est une extension orientée objet de C distinguant objets et des valeurs. Le langage permet d'introduire des classes avec méthodes, des constructeurs d'objets complexes (tuples et ensembles), de la généralisation et des facilités de filtrage itératif de données. Le système O2 est aujourd'hui commercialisé par O2 Technology.

[Lippman91] LippmanB. S., *C++ Primer*, 2e édition, 614 pages, Addison-Wesley, 1991.

Un excellent livre sur C++. Le langage est présenté sous tous ses aspects.

[Maier86] Maier D. et. al., « Development of an object-Oriented DBMS », *1st Int. Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, Oct. 1986.

Un des premiers articles décrivant l'implémentation d'un SGBDO, le SGBD GemStone. Celui-ci est construit à partir d'une interface Smalltalk (Gem) connectée à un serveur d'objets (Stone).

[OMG91] Object Management Group & X Open Group, « The Common Object Request Broker: Architecture and Specification », *OMG Document N° 91, 12.1*, Revision 1.1, octobre 1991.

Présentation de l'architecture CORBA de l'OMG. Cette architecture distribuée permet d'envoyer des requêtes à des objets distants et de recevoir les réponses avec des interfaces et un modèle d'objets en voie de standardisation, en s'abstrayant de l'implémentation des objets. CORBA vise à assurer l'interopérabilité entre les environnements orientés objet. Les plus grandes compagnies industrielles soutiennent CORBA. Aujourd'hui, plus de 800 fournisseurs de logiciels et utilisateurs adhèrent à l'OMG.

[Rational97] Rational Software, *The Unified Modeling Language UML 1.1*, Reference Manual, Release 1.1, Boston, aussi disponible auprès de l'OMG, 1997.

Le document de référence d'UML. Les spécifications d'UML, le langage graphique universel de modélisation de l'objet, sont disponibles à l'adresse Internet www.rational.com/uml/ *sous forme de plusieurs documents HTML ou PDF. Nous en donnons un résumé dans le chapitre sur la conception des bases de données.*

[Shaw90] Shaw G., Zdonik B.S., « A Query Algebra for Object-Oriented Databases », *Proc. of the 6th International Conf. On Data Engineering*, IEEE Ed., pp. 154-162, 1990.

Cet article propose une algèbre d'objet pour interroger les bases de données objet. Cette algèbre a été implémentée dans le projet ENCORE, et est restée connue sous ce nom. Nous l'avons décrite ci-dessus.

[Stoustrup86] Stoustrup B., *The C++ Programming Language*, New York, Addison-Wesley, 1986.

Le livre de référence de C++ par son inventeur. Stoustrup a créé C++ pour modéliser des problèmes de réseaux de télécommunications.

[WonKim88] Won Kim *et. al.*, « Features of the ORION Object-Oriented Database System », dans le livre "*Object-Oriented Concepts, Applications and Databases*", W. Kim et Lochovsjy Ed., Addison-Wesley, 1988.

Une description du système ORION, le SGBDO qui a popularisé l'approche bases de données à objets. Développé à MCC dès 1985, ORION est un SGBDO très complet. Initialement basé sur Lisp, le produit, commercialisé aujourd'hui par Itasca Systems, a évolué vers C et C++.

[WonKim89] Won Kim, « A Model of Queries for Object-Oriented Database », *Very Large Database International Conference*, Morgan Kaufman Pub., Amsterdam, Pays-Bas, août 1989.

Cet article présente les méthodes d'optimisation utilisées dans le système ORION pour le langage d'interrogation. La technique retenue est très proche de la restructuration d'arbre, considérant en plus les jointures par références et les parcours de chemins.

[WonKim90] Won Kim, *Introduction to Object-Oriented Databases*, 235 pages, The MIT Press, 1990.

Ce livre décrit les différentes techniques des SGBD à objets. Il s'inspire fortement du système ORION. Plus particulièrement, les problèmes de modèle orienté objet, de modification de schéma, de langage SQL étendu aux objets, de structures de stockage, de gestion de transactions, d'optimisation de requêtes et d'architecture sont abordés. Une bonne référence sur les bases de données à objets.

[Zaniolo85] Zaniolo C., « The Representation and Deductive Retrieval of Complex Objects », *11th Very Large Data Bases International Conference*, Morgan Kaufman Pub., Stockholm, Suède, août 1985.

Cet article présente une extension de l'algèbre relationnelle aux fonctions permettant de retrouver des objets complexes. Des opérateurs déductifs de type point fixe sont aussi intégrés.

[Zdonik90] Zdonik S., Maier D., *Readings in Object-Oriented Database Systems*, Morgan Kaufman Pub., San Mateo, California, 1990.

Une sélection d'articles sur les bases de données à objets.