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

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

Cours n°4 : Le modèle objets

Nous allons dans ce cours étudier les modèles de données et les systèmes de bases de données orientés objet. Les modèles et les systèmes traditionnels (relationnel, en réseau, hiérarchique) se sont révélés très performants dans le développement de la technologie nécessaire à la conception de nombreuses applications de bases de données d'entreprise classiques. Ils présentent néanmoins quelques limitations lorsqu'il s'agit de concevoir et d'implémenter des applications plus complexes, comme des bases de données pour la conception et la fabrication en génie civil (CAO/FAO et FIO[1]), les expériences scientifiques, les télécommunications, les systèmes d'information géographiques et le multimédia[2].

Ces nouvelles applications ont des exigences et des caractéristiques qui diffèrent des applications de gestion traditionnelles : des objets de structure plus complexe, des transactions de plus longue durée, de nouveaux types de données permettant de stocker des images ou des textes volumineux, et la nécessité de définir des opérations non standard spécifiques aux applications.

Les bases de données orientées objet ont vu le jour pour répondre aux besoins de ces applications plus complexes. Elles offrent la souplesse nécessaire pour traiter certains de ces besoins sans souffrir des limitations associées aux types de données et langages de requête disponibles dans les systèmes traditionnels. Elles permettent également au concepteur de spécifier aussi bien la structure d'objets complexes que les opérations applicables à ces objets.

Une autre raison tient à l'utilisation croissante de langages de programmation OO dans le développement d'applications logicielles. Les bases de données sont maintenant devenues des composants fondamentaux de nombreux systèmes logiciels, et les bases de données traditionnelles sont difficilement exploitables par des applications orientées objet développées dans des langages tels que C++, Smalltalk ou Java.

Les bases de données orientées objet sont conçues pour pouvoir être intégrées directement ou de façon homogène, à des logiciels développés dans un langage de programmation orienté objet.

Les fournisseurs de SGBD relationnels ont également reconnu la nécessité d'ajouter d'autres possibilités de modélisation des données, et les nouvelles versions des systèmes relationnels incorporent nombre de fonctionnalités qui étaient l'apanage des bases de données objet. Il en résulte des systèmes qualifiés de SGBD *relationnels-objet* ou *relationnels étendus* (voir cours 3). La dernière version du standard SQL pour les SGBD relationnels intègre certaines de ces fonctionnalités.

Malgré la création de nombreux prototypes expérimentaux et de SGBD orientés objet commerciaux, ceux-ci ne sont pas extrêmement répandus en raison de la popularité des systèmes relationnels et relationnels-objet.

Parmi les prototypes expérimentaux, citons le système ORION développé par MCC (Microelectronics and Computer Technology Corporation, Austin, Texas), OPENOODB par Texas Instruments, le système IRIS par les laboratoires de Hewlett Packard, le Système ODE par AT&T Bell Labs et le projet ENCORE/ObServer par l'université Brown. Les systèmes disponibles dans le commerce comprennent notamment GEMSTONE/OPAL de GemStone Systems, ONTOS de Ontos, Objectivity de Objectivity Inc., Versant de Versant object Technology, ObjectStore de object Design, ARDENT de ARDENT Software3 et POET de POET Software.

À mesure que des SGBD orientés objet devenaient disponibles, la nécessité d'un modèle et d'un langage standard se faisait reconnaître. Comme la procédure formelle d'approbation d'une norme prend généralement un certain nombre d'années, un consortium de fournisseurs et d'utilisateurs de SGBD OO, l'ODMG[3], a proposé un standard connu sous le nom de standard ODMG-93, qui a été révisé depuis. Nous décrirons certaines caractéristiques de ce standard au cours suivant.

[1] Conception Assistée par Ordinateur, Fabrication Assistée par Ordinateur et Fabrication Intégrée par Ordinateur.
[2] Les bases de données multimédias peuvent stocker différents types d'objets: vidéo, audio, images et documents.
[3] Objet Database Management Group.

Les bases de données orientées objet ont adopté de nombreux concepts originellement développés pour les langages de programmation objet. À la section 1, nous examinons les origines de l'approche orientée objet et la façon dont elle s'applique aux bases de données. Nous décrivons ensuite les différents concepts de cette approche:

- l'identité des objets, la structure des objets et les constructeurs de types (section 2);
- les concepts d'encapsulation des opérations et la définition des méthodes dans les déclarations de classe, ainsi que les mécanismes de stockage des objets dans une base de données grâce à la persistance (section 3) ;
- les hiérarchies de types et de classes, et l'héritage dans les bases de données orientées objet (section 4) ;
- les problèmes qui se posent quand il s'agit de représenter et de stocker des objets complexes (section 5) ;
- le polymorphisme, la surcharge des opérateurs, la liaison dynamique, l'héritage multiple et sélectif, le versionnage et la configuration des objets (section 6).

1. Vue d'ensemble des concepts de l'orientation objet

Cette section fournit un aperçu de l'historique et des principaux concepts des bases de données orientées objet (ou BDOO). Le terme « orienté objet» (abrégé en OO ou O-O) trouve son origine dans les langages de programmation OO (ou LPOO). Aujourd'hui, les concepts OO sont appliqués dans le domaine des bases de données, du génie logiciel, des bases de connaissances, de l'intelligence artificielle et des systèmes informatiques en général. Les LPOO ont leurs racines dans le langage Simula, proposé à la fin des années 1960. En Simula, le concept de classe groupe la structure de données interne à un objet dans une déclaration de classe. Par la suite, les chercheurs ont proposé le concept de type de données abstrait, qui masque les structures de données internes et spécifie toutes les opérations externes possibles applicables à un objet, débouchant sur le concept d'encapsulation. Le langage Smalltalk, développé au PARC[4] de Xerox dans les années 1970, a été l'un des premiers langages à incorporer explicitement d'autres concepts OO, tels que la transmission de messages et l'héritage. Il est connu comme un langage de programmation purement OO, ce qui signifie qu'il a été explicitement conçu pour être orienté objet. Il diffère en cela des langages OO hybrides, qui incorporent les concepts OO à un langage existant, par exemple C++, qui ajoute des caractéristiques OO au langage C.
Un objet possède généralement deux composants: un état (une valeur) et un comportement (des opérations). En conséquence, il est comparable à une variable d'un langage de programmation, excepté qu'il a habituellement une *structure de données complexe* et des *opérations spécifiques* définies par le programmeur.
Dans un LPOO les objets n'existent que durant l'exécution du programme et sont donc appelés temporaires. Une base de données OO peut prolonger l'existence des objets pour les stocker de façon permanente : les objets persisteront au-delà de la fin du programme et pourront être plus tard extraits et partagés par d'autres programmes. Autrement dit, les bases de données OO stockent des objets permanents dans un stockage secondaire, et permettent à plusieurs programmes et applications de les partager. Cela nécessite l'incorporation d'autres fonctionnalités bien connues des SGBD, comme les mécanismes d'indexation, le contrôle de la concurrence et la récupération. Un système bases de données OO s'interface avec un ou plusieurs LPOO pour permettre la persistance et le partage des objets.
L'un des buts des bases de données OO est de maintenir une correspondance directe entre les objets de la base et ceux du monde réel, afin qu'ils ne perdent pas leur identité ni leur intégrité, et qu'ils soient faciles à identifier et à utiliser. C'est pourquoi chaque objet d'une base OO possède un identifiant d'objet unique (OID pour *Object Identifier ou Object ID*) généré par le système. Nous pouvons lui trouver une analogie dans le modèle relationnel, dans lequel chaque relation doit avoir un attribut de clé primaire dont la valeur identifie chaque tuple de façon unique. Si l'on modifie la valeur de la clé primaire, le tuple aura une nouvelle identité, même s'il continue à représenter le même objet du monde réel. Par ailleurs, les attributs clé d'un objet du monde réel peuvent porter des noms différents dans différentes relations; il est donc difficile de s'assurer que les clés représentent bien le même objet (par exemple, l'identifiant de l'objet peut être représenté par ID_EMP dans une relation et par NoSS dans une autre).
Une autre caractéristique des bases OO est que les objets peuvent avoir une structure de complexité arbitraire pour pouvoir contenir toutes les informations nécessaires à les décrire. En revanche, dans les systèmes de bases de données traditionnels, les informations concernant un objet complexe sont souvent réparties entre plusieurs relations ou enregistrements, débouchant sur une perte de correspondance directe entre un objet du monde réel et sa représentation dans la base.

[4] Palo Alto Research Center, Palo Alto, Californie.

Dans un LPOO, la structure interne d'un objet comprend la spécification des **variables d'instance**, qui contiennent les valeurs définissant son état interne. Une variable d'instance est donc comparable à la notion d'*attribut* du modèle relationnel, excepté qu'elle peut être encapsulée dans l'objet, et qu'elle n'est donc pas nécessairement visible pour les utilisateurs externes. Les variables d'instance peuvent également être d'un type de données arbitrairement complexe. Les systèmes orientés objet permettent de définir des opérations ou des fonctions (comportements) qui peuvent s'appliquer à tous les objets d'un type particulier. En fait, certains modèles OO préconisent de prédéfinir toutes les opérations qu'un utilisateur peut appliquer sur un objet, ce qui implique une *encapsulation totale*. Cette approche rigide a été assouplie dans la plupart des modèles, et ce pour deux raisons. Tout d'abord, l'utilisateur d'une base a souvent besoin de connaître les noms des attributs pour pouvoir spécifier les conditions de sélection nécessaires à l'extraction des objets spécifiques. Deuxièmement, l'encapsulation totale implique que toute extraction, si simple soit-elle, demande qu'une opération ait été prédéfinie, ce qui rend les requêtes *ponctuelles* difficiles à spécifier à la volée.
Pour encourager l'encapsulation, une opération est définie en deux parties. La première, nommée *signature* ou *interface* de l'opération, spécifie le nom de l'opération et ses arguments (ou paramètre). La seconde, nommée *méthode* ou *corps* définit l'implémentation de l'opération. On invoque une opération en transmettant à un objet un message qui inclut le nom de l'opération et ses paramètres. L'objet exécute alors la méthode pour cette opération. Cette encapsulation permet de modifier la structure d'un objet et l'implémentation de ses opérations, sans devoir toucher aux programmes externes qui invoquent ces opérations. En conséquence, l'encapsulation assure une forme d'indépendance des données et des opérations.
Un autre concept clé des systèmes OO est celui de hiérarchie de types et de classes, autrement dit d'*héritage*. Celui-ci permet de spécifier de nouveaux types ou classes qui héritent la plus grande partie de leur structure et/ou de leurs opérations de types ou classes définis auparavant. En conséquence, la définition de types d'objets peut effectuer de façon systématique, ce qui facilite le développement incrémental des types de données d'un système et la réutilisation de définitions de types existantes lors de la création de nouveaux types d'objets.
L'un des problèmes des premiers systèmes de bases de données OO concernait les *relations* entre les objets. Le fait d'insister sur l'encapsulation totale dans les premiers modèles de données OO a conduit à vouloir décrire les relations en définissant des méthodes appropriées qui localiseraient les objets apparentés, au lieu de les représenter explicitement. Cependant, cette approche ne fonctionnait pas très bien pour les bases de données complexes comprenant de nombreuses relations, qui devaient être clairement identifiées et visibles aux utilisateurs. Le standard de l'ODMG a reconnu cette nécessité, et représente explicitement les relations binaires au moyen d'une paire de références inverses - autrement dit, en plaçant les OID es objets apparentés dans les objets eux-mêmes, et en maintenant l'intégrité référentielle (chapitre suivant).
Certains systèmes OO permettent de gérer plusieurs versions du même objet - une caractéristique essentielle dans les applications d'ingénierie. Par exemple, une ancienne version d'un objet qui a été vérifiée et testée doit être conservée jusqu'à ce que la nouvelle version soit elle-même vérifiée et testée. Une nouvelle version d'un objet complexe peut n'inclure que quelques nouvelles versions de ses composants, tandis que d'autres composants demeureront inchangés. En plus d'autoriser le versionnage, les bases de données OO doivent également permettre l'évolution du schéma, qui a lieu quand les déclarations de types sont modifiées ou quand de nouveaux types ou de nouvelles relations sont créés. Ces deux caractéristiques ne sont pas spécifiques aux SGBDO, et devraient dans l'idéal faire partie de tous les SGBD[5].
Le concept de *surcharge des opérateurs* désigne quant à lui la possibilité d'appliquer une opération à différents types d'objets. Dans une telle situation, le nom d'une opération peut référencer plusieurs implémentations distinctes, selon le type d'objets auxquels elle est appliquée. On nomme également cette caractéristique *polymorphisme des opérateurs*. Par exemple, une opération destinée à calculer la surface d'un objet géométrique peut différer dans sa méthode (implémentation), selon que l'objet est de type triangle, cercle ou rectangle. Cela peut nécessiter la liaison dynamique du nom de l'opération à la méthode appropriée au moment de l'exécution, lorsque le type de l'objet auquel s'applique l'opération est connu.

2. Identité des objets, structure des objets et constructeurs de types

Cette section commence par étudier le concept d'identité des objets, puis présente les opérations types qui permettent de définir la structure de l'état d'un objet, souvent appelées constructeurs de types. Il s'agit des opérations basiques de structuration des données que l'on peut combiner pour former des structures d'objets complexes.

[5] Plusieurs opérations d'évolution du schéma, par exemple ALTER TABLE, sont déjà définies dans le standard SQL relationnel.

2.1 Identité des objets

Un système de bases de données OO attribue une identité unique à chaque objet indépendant stocké dans la base. Cette identité unique est généralement implémentée via un identifiant d'objet unique généré par le système, ou OID. La valeur d'un OID n'est pas visible pour l'utilisateur externe, mais elle est utilisée en interne par le système pour identifier sans ambiguïté chaque objet, et pour créer et gérer les références entre objets. L'OID peut être affecté à des variables du programme de type approprié lorsque c'est nécessaire.
La principale propriété requise d'un OID est l'**immuabilité**: autrement dit, la valeur de l'OID d'un objet donné ne doit jamais changer. Cette caractéristique préserve l'identité des objets du monde réel représentés. En conséquence, un système de bases de données OO doit disposer d'un mécanisme pour générer des OID et préserver leur immuabilité. Il est également souhaitable que chaque OID ne soit utilisé qu'une fois: lorsqu'un objet est supprimé de la base, son OID ne doit pas être réaffecté à un autre objet. Ces deux propriétés impliquent que l'OID ne dépende d'aucune valeur d'attribut de l'objet, puisque la valeur d'un attribut peut être modifiée ou corrigée. Il est aussi généralement inapproprié de baser l'OID sur l'adresse physique de l'objet dans le stockage, puisque cette adresse peut varier après une réorganisation physique de la base. Certains systèmes emploient toutefois l'adresse physique pour augmenter l'efficacité de l'extraction des objets. Si l'adresse physique de l'objet change, il est possible de placer à l'ancienne adresse un pointeur indirect qui indiquera le nouvel emplacement de l'objet. Il est plus courant d'utiliser des entiers longs comme OID, puis d'employer une table de hachage pour faire correspondre la valeur de l'OID à l'adresse physique courante de l'objet dans le stockage.
Certains anciens modèles de données OO exigeaient que tout élément (de la valeur simple à l'objet complexe) soit représenté comme un objet. Dans cette optique, chaque valeur de base (entier, chaîne de caractères ou valeur booléenne par exemple) possède un OID. Cela permet à deux valeurs de base d'avoir des OID différents, ce qui peut être utile dans certains cas. Par exemple, la valeur entière 50 peut servir à un moment donné à représenter un poids en kilogrammes et à un autre l'âge d'une personne. Il est alors possible de créer deux objets ayant des OID distincts, mais représentant la valeur entière 50. Si ce modèle est utile sur le plan théorique, il l'est moins dans la pratique car il peut entraîner la génération d'OID trop nombreux. C'est pourquoi la plupart des systèmes de bases de données OO autorisent à représenter à la fois des objets et des valeurs. Chaque objet doit avoir un OID immuable, tandis qu'une valeur est autonome et n'a pas d'OID. En conséquence, une valeur est généralement mémorisée dans un objet et ne peut pas être référencée depuis d'autres objets. Certains systèmes permettent également de créer des valeurs de structure complexe sans OID correspondant si nécessaire.

2.2 Structure des objets

Dans une base de données OO, l'état (la valeur courante) d'un objet complexe peut être construit à partir d'autres objets (ou d'autres valeurs) grâce à des **constructeurs de types**[6]. Un moyen formel de représenter de tels objets consiste à considérer chaque objet comme un triplet (i, c, v), où i est un identifiant d'objet unique (l'OID), c un constructeur de type (autrement dit, une indication de la façon dont l'état de l'objet est construit) et v l'état de l'objet (sa valeur courante). Le modèle de données comprendra généralement plusieurs constructeurs de types. Les trois constructeurs les plus élémentaires sont *atom*, *tuple* et *set*. D'autres constructeurs couramment utilisés sont *list*, *bag* et *array*. Le constructeur *atom* sert à représenter toutes les valeurs atomiques de base, comme les entiers, les nombres réels, les chaînes de caractères, les valeurs booléennes et autres types primitifs que le système prend en charge directement.
L'état v d'un objet (i, c, v) est interprété en fonction du constructeur c. Si *c = atom*, l'état (la valeur) v est une valeur atomique appartenant à l'ensemble des valeurs de base prises en charge par le système. Si *c = set*, l'état v est un ensemble d'identifiants d'objets $\{i_1, i_2, \ldots, i_n\}$, autrement dit les OID d'un ensemble d'objets généralement du même type. Si *c = tuple*, l'état v est un tuple de la forme $<a_1:i_1, a_2:i_2, \ldots, a_n:i_n>$, où chaque a_j est un nom d'attribut[7] et chaque i_j est un OID. Si *c = list*, la valeur est une liste ordonnée $[i_1, i_2, \ldots, i_n]$ d'OID d'objets du même type. Une liste est similaire à un ensemble, excepté que ses éléments sont ordonnés, ce qui permet de référencer le premier, le deuxième ou le j^e objet. Si *c = array*, l'état de l'objet est un tableau unidimensionnel d'OID. La principale différence entre un tableau et une liste est qu'une liste peut avoir un nombre arbitraire d'éléments tandis qu'un tableau a généralement une taille maximale. La différence entre set et *bag*[8] est que tous les éléments d'un *set* doivent être distincts alors qu'un *bag* peut contenir des éléments dupliqués.

[6] Cette notion diffère de celle du constructeur utilisé les LPOO pour créer de nouveaux objets.
[7] Également nommé variable d'instance dans la terminologie OO.
[8] Également nommé multiset.

Ce modèle d'objets permet d'imbriquer arbitrairement des ensembles, listes, tuples et autres constructeurs. L'état d'un objet qui n'est pas de type atomique référencera les autres objets par leur OID. En conséquence, le seul cas dans lequel une valeur réelle apparaît est celui de l'état d'un objet de type atom.
Les constructeurs de types **set**, **list**, **array** et **bag** sont appelés **types collection**, pour les distinguer des types de base et des types tuple. La principale caractéristique d'un type collection est que l'état de l'objet sera une *collection d'objets* qui peut être non ordonnée (comme *set* ou *bag*) ou ordonnée (comme *list* ou *array*). Le constructeur de type **tuple** est souvent appelé **type structuré**, puisqu'il correspond à la construction **struct** en C et en C++.

*** Exemple d'objet complexe :**

Nous représentons maintenant certains objets de la base de données relationnelle Entreprise (figure 1), en appliquant le modèle précédent dans lequel un objet est défini par un triplet (OID, constructeur de type, état) et où les constructeurs de types disponibles sont *atom*, *set* et *tuple*. Nous utilisons i_1, i_2, i_3… pour représenter les OID uniques générés par le système.

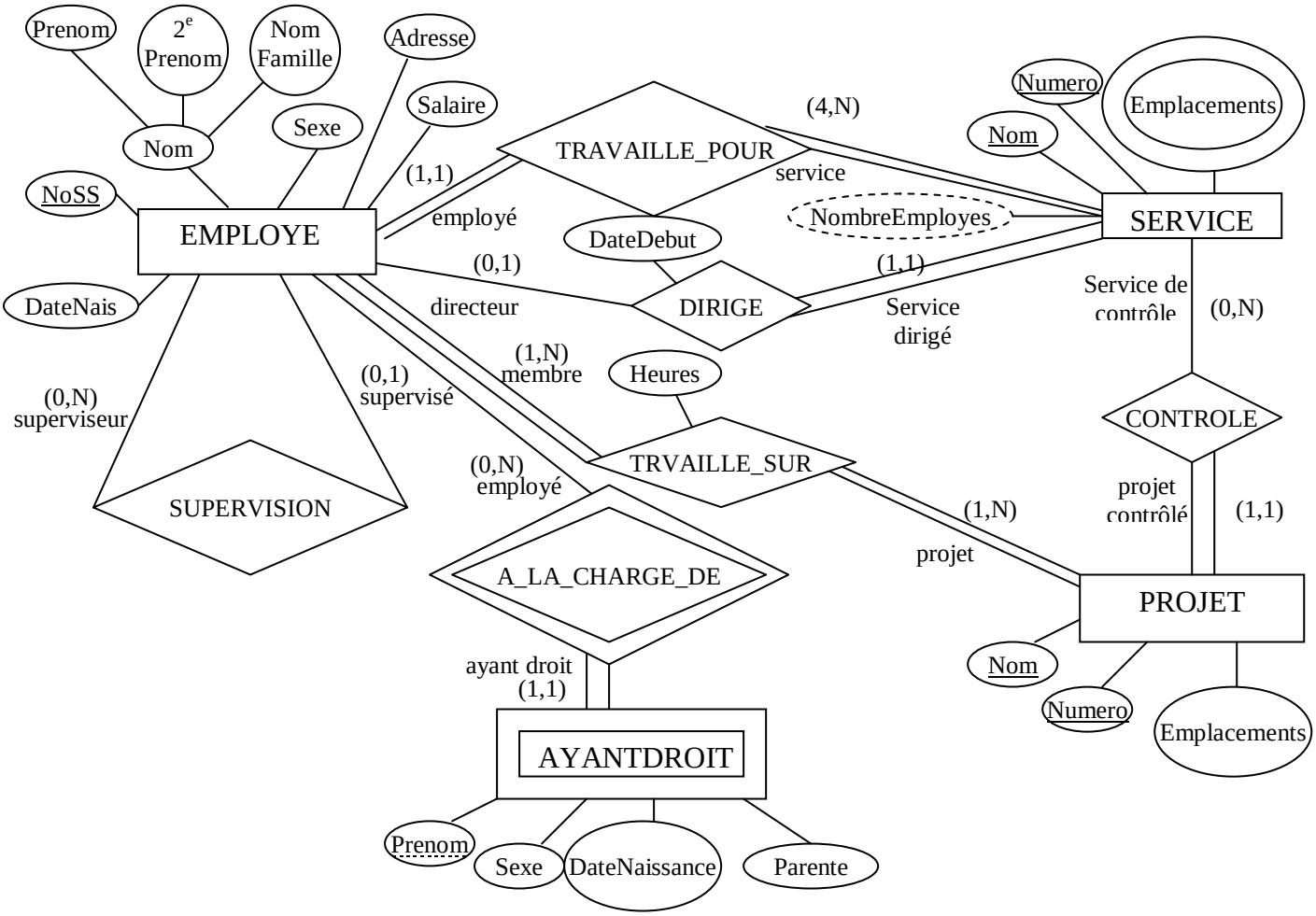

Figure 1 : Diagramme ER de BD entreprise

La base de données ENTREPRISE contient des données sur les employés, les services et les projets d'une société. Les concepteurs de base de données fournissent la définition suivante du « mini-monde » (la partie de la société devant être représentée dans la base) :

1. La société est organisée en services. Chaque service a un nom et un numéro uniques et est dirigé par un employé unique. La date à laquelle l'employé a commencé à diriger le service est comptabilisée. Un service peut avoir plusieurs emplacements.
2. Un service contrôle un certain nombre de projets, chacun d'entre eux ayant un nom, un numéro et un emplacement uniques.
3. Le nom de chaque employé, son numéro de Sécurité sociale, son adresse, son salaire, son sexe et sa date de naissance sont mémorisés. Un employé est affecté à un service, mais peut travailler sur plusieurs projets qui ne sont pas forcément contrôlés par le même service. Le nombre d'heures hebdomadaires travaillées par chaque employé est comptabilisé. Le supérieur immédiat de chaque employé est lui aussi mémorisé.

4. Les ayants droit de chaque employé doivent être indiqués pour des raisons d'assurance. Leur prénom, leur sexe, leur date de naissance et leur lien de parenté avec l'employé sont mémorisés.

La figure 1 ci-dessus illustre comment le schéma de cette application de base de données peut être transcrit grâce à la notation graphique connue sous le nom de diagrammes ER et la figure 2 illustre comment le même schéma peut être transcrit avec le diagramme de classes UML.

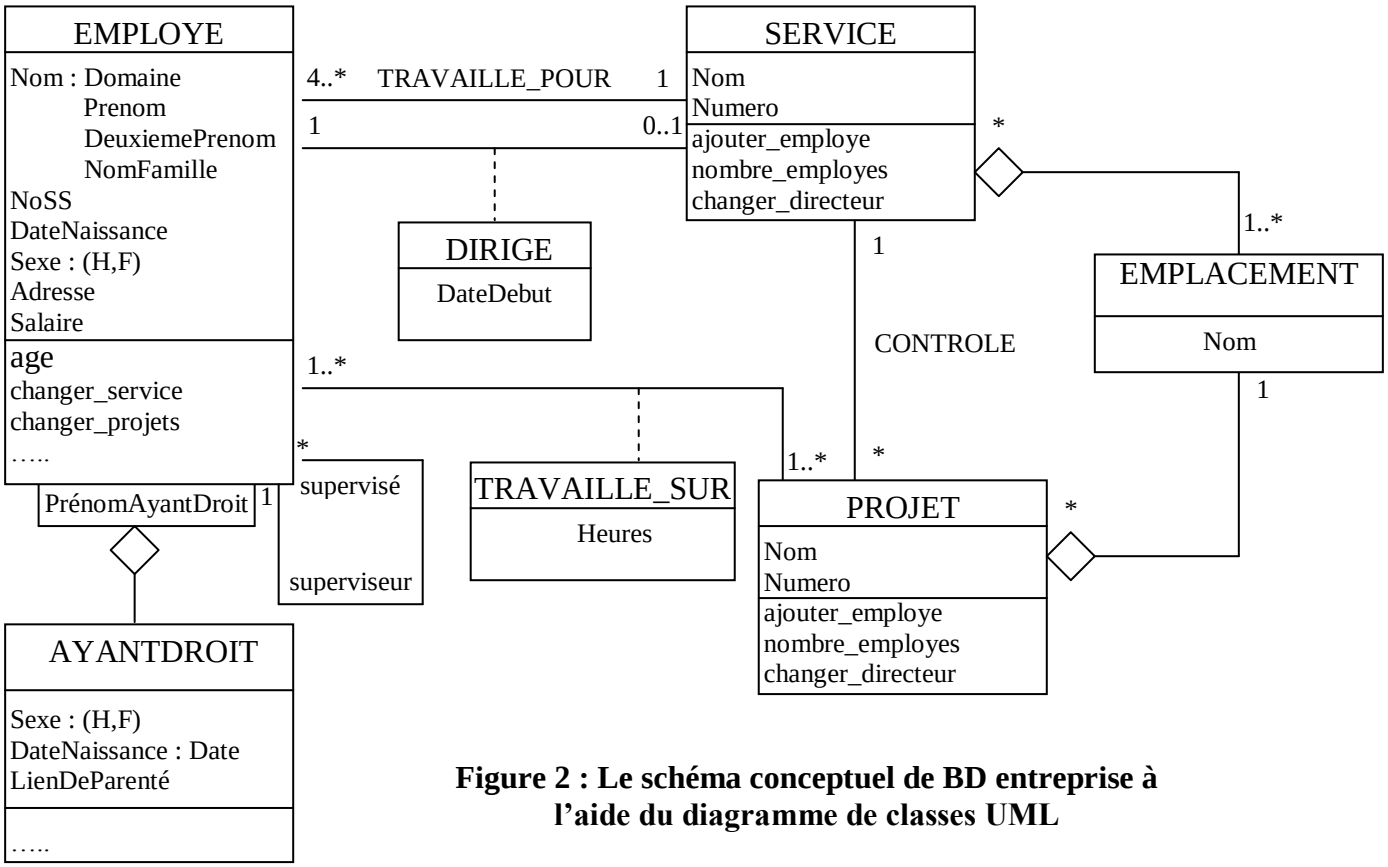

Figure 2 : Le schéma conceptuel de BD entreprise à l'aide du diagramme de classes UML

Considérons les objets suivants:

O_1 = (i_1, atom, 'Valbonne')
O_2 = (i_2, atom, 'Les Ulis')
O_3 = (i_3, atom, 'Lanester')
O_4 = (i_4, atom, 5)
O_5 = (i_5, atom, 'Recherche')
O_6 = (i_6, atom, '1988-05-22')
O_7 = (i_7, set, {i_1, i_2, i_3})
O_8 = (i_8, tuple, <NOMS:i_5, NoS:i_4, DIR:i_9, EMPLACEMENTS:i_7, EMPLOYES:i_{10}, PROJETS:i_{11} >)
O_9 = (i_9, tuple, <DIRECTEUR:i_{12}, DATE_DEBUT_DIRECTEUR:i_6>)
O_{10} = (i_{10}, set, {i_{12}, i_{13}, i_{14}})
O_{11} = (i_{11}, set, {i_{15}, i_{16}, i_{17}})
O_{12} = (i_{12}, tuple, <PRENOM:i_{18}, DEUXIEMEPRENOM:i_{19}, NOM:i_{20}, NoSS:i_{21}, . . . , SALAIRE:i_{26}, SUPERVISEUR:i_{27}, SERV:i_8>)
….

Les six premiers objets (O_1-O_6) représentent les valeurs atomiques. Il y aura de nombreux objets semblables, un pour chaque valeur atomique constante distincte dans la base de données[9]. L'objet O_7 est un objet typé set représentant l'ensemble des EMPLACEMENTS pour SERVICE 5; l'ensemble { i_1, i_2, i_3} désigne les objets atomiques ayant les valeurs {'Valbonne', 'Les Ulis', 'Lanester'}. L'objet O_8 est un objet de type tuple qui représente SERVICE 5 lui-même, et possède les attributs SNOM, SNUMERO, DIR, SITES, etc. Les deux premiers attributs, SNOM et SNUMERO ont pour valeur les objets atomiques O_5 et O_4. L'attribut DIR a pour valeur l'objet tuple O_9, qui possède à son tour deux attributs. La valeur de l'attribut DIRECTEUR est l'objet

[9] Ce sont ces objets atomiques qui posent problème : le nombre d'OID généré est trop élevé lorsque le modèle est implémenté directement.

dont l'OID est i_{12}, qui représente l'employé 'Jean Ferrant' qui dirige le SERVICE, tandis que la valeur de DATE_DEBUT_DIRECTEUR est un autre objet atomique dont la valeur est une date. La valeur de l'attribut EMPLOYES de O_8 est un objet de type set dont l'OID est i_{10} et dont la valeur est l'ensemble des OID des employés qui travaillent pour le SERVICE (objets i_{12}, i_{13} et i_{14}, qui ne sont pas représentés ici). De même, la valeur de l'attribut PROJETS de O_8 est un objet de type set dont l'OID est i_{11}, et dont la valeur est l'ensemble des OID des projets gérés par le SERVICE 5 (objets i_{15}, i_{16} et i_{17}, qui ne sont pas représentés ici). L'objet dont l'OID est i_{12} représente l'employé 'Jean Ferrant' avec tous ses attributs atomiques (PRENOM, DEUXIEMEPRENOM, NOMFAMILLE, NOSS, . . ., SALAIRE, qui référencent respectivement les objets atomiques i_{19}, i_{20}, i_{21}, …, i_{26} non représentés), SUPERVISEUR qui référence l'objet EMPLOYE dont l'OID est i_{27} (il représente 'Jacques Bernard' qui supervise 'Jean Ferrant' mais il n'est pas représenté) et SERV qui référence l'objet SERVICE dont l'OID est i_8 (il représente le SERVICE numéro 5 dans lequel 'Jean Ferrant' travaille).

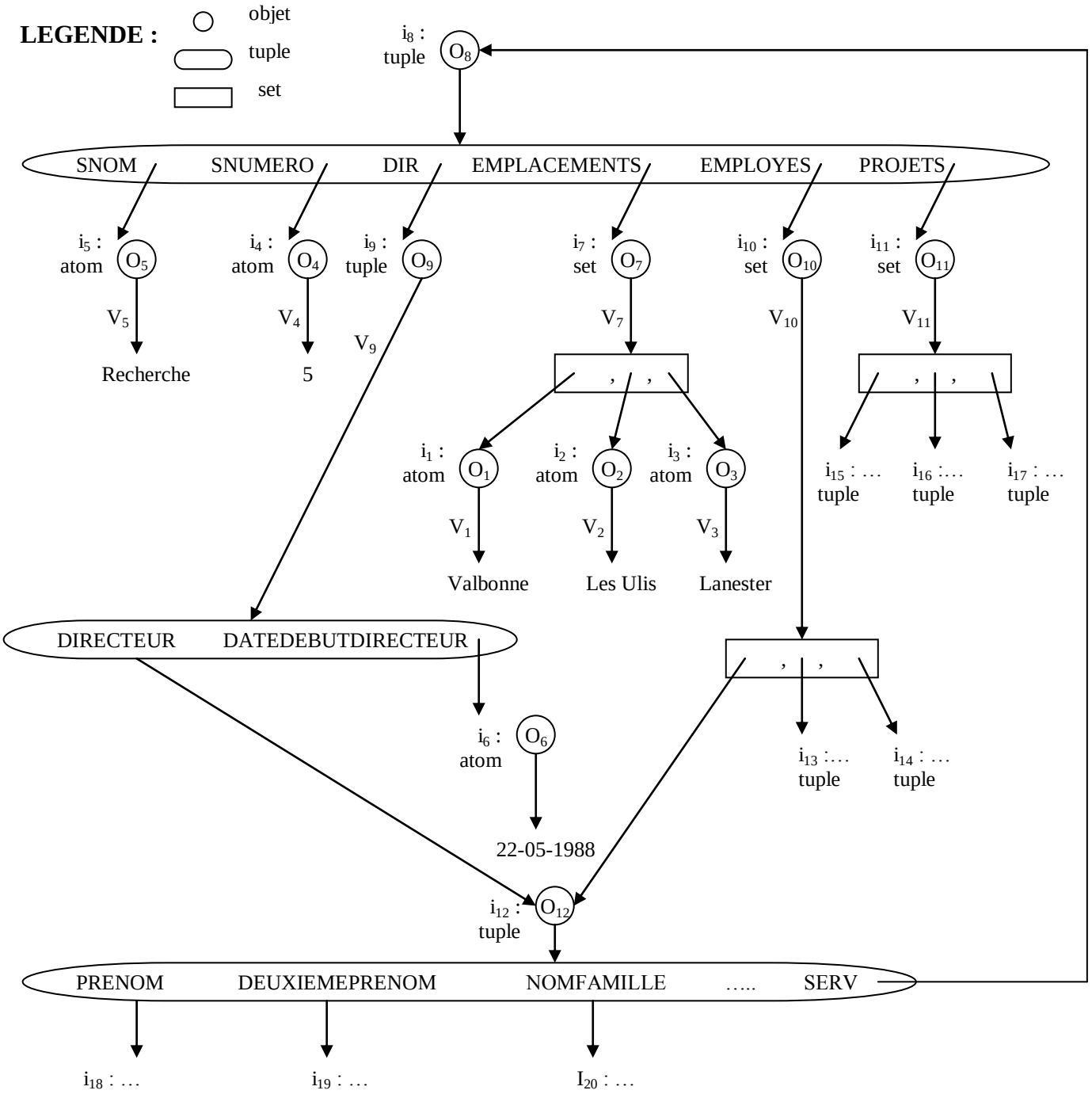

Figure 3 : Représentation de l'objet complexe Service sous forme de graphe

Dans ce modèle, on peut représenter un objet sous forme de graphe, construit en appliquant récursivement les constructeurs de types. Pour construire le graphe représentant un objet O_i, on crée d'abord un nœud pour l'objet

O_i lui-même. Le nœud de O_i reçoit une étiquette portant l'OID et le constructeur de l'objet, c. On crée également un nœud pour chaque valeur atomique de base. Si un objet O_i a une valeur atomique, on trace un arc partant du nœud représentant O_i et allant vers le nœud qui représente sa valeur. Si la valeur de l'objet est construite, on trace à partir du nœud de l'objet un arc orienté dirigé vers celui qui représente la valeur construite. La figure 3 est le graphe de l'objet SERVICE O_8 présenté plus haut.

*** Objets identiques et objets égaux :**

Le modèle précédent permet deux types de définitions lors d'un test d'égalité de *l'état de deux objets*. On dit que deux objets ont des **états identiques** (égalité stricte) si les graphes représentant leur état sont identiques à tous points de vue, y compris les OID à chaque niveau. En revanche, on dit que les états de deux objets sont égaux (égalité superficielle) si la structure des deux graphes est la même et si toutes les valeurs atomiques qui se correspondent dans les deux graphes sont également les mêmes. Cependant, certains nœuds internes qui se correspondent dans les deux graphes peuvent représenter des objets ayant des OID différents.

Cet exemple illustre la différence entre les deux définitions possibles de l'égalité.

Considérons les objets O_1, O_2, O_3, O_4, O_5 et O_6:

$O_1 = (i_1, \text{tuple}, <a_1:i_4, a_2:i_6>)$

$O_2 = (i_2, \text{tuple}, <a_1:i_5, a_2:i_6>)$

$O_3 = (i_3, \text{tuple}, <a_1:i_4, a_2:i_6>)$

$O_4 = (i_4, \text{atom}, 10)$

$O_5 = (i_5, \text{atom}, 10)$

$O_6 = (i_6, \text{atom}, 20)$

Les états des objets O_1 et O_2 sont égaux, puisque leurs états au niveau atomique sont les mêmes mais que les valeurs sont atteintes à travers des objets distincts O_4 et O_5. En revanche, les états des objets O_1 et O_3 sont identiques, même si les objets eux-mêmes ne le sont pas parce qu'ils ont des OID différents. De même, bien que les états de O_4 et O_5 soient identiques, les objets réels O_4 et O_5 sont égaux mais non identiques, parce que leurs OID sont distincts.

2.3 Constructeurs de types

On n'utilisera pas ici un **langage de définitions d'objets** ou **ODL**(***Object Definition Language***), qui incorpore les constructeurs de types précédents, pour définir les types d'objets d'une base de données particulière. Nous présenterons le standard ODL de l'ODMG au cours suivant, mais nous allons d'abord en introduire les concepts dans cette section, en employant une notation plus simple.

On peut utiliser les constructeurs de types pour définir les structures de données pour un schéma de base de données OO. À la section suivante, nous verrons comment intégrer la définition des opérations (ou méthodes) dans le schéma OO. Le listing (2.1) ci-dessous montre comment nous pouvons déclarer les types Employe et Service correspondant aux instances d'objets de la figure 2.1. Dans ce listing, le type de données Date est défini comme un tuple et non comme une valeur atomique. Nous utilisons les mots clés tuple, set et list pour les constructeurs de types, et les types de données standard disponibles (integer, string, float, etc.) pour les types atomiques.

```
define type Employe:
        tuple (         prenom:             string;
                        deuxiemeprenom:     char;
                        nom:                string;
                        noss:               string;
                        datenaissance:              Date;
                        adresse:                    string;
                        sexe:               char;
                        salaire:                    float;
                        Superviseur:        Employe;
                        serv:               Service; );
define type Date
        tuple (
                        annee:              integer;
```

```
                    mois:           integer;
                    jour:           integer; );

define type Service
        tuple (     snom:           string;
                    snumero:        integer;
                    dir:            tuple (     directeur: Employe;
                                                datedebut: Date; );
                    sites:          set(string);
                    employes:       set(Employe);
                    projets :           set(Projet); );
```

Listing 2.1 : Spécification des types Employe, Date et Service

Les attributs qui référencent d'autres objets, tels que l'attribut serv de Employe ou l'attribut projets de Service, sont en substance des **références** à d'autres objets, et servent donc à représenter des relations entre des types d'objets. Par exemple, l'attribut serv de Employe est de type Service et sert donc à référencer un objet Service spécifique (dans lequel l'Employe travaille). La valeur d'un tel attribut serait l'OID d'un objet Service spécifique. Une relation binaire peut être représentée dans une seule direction, ou avoir une référence inverse. La dernière représentation facilite la traversée de la relation dans les deux directions. Par exemple, l'attribut employes de Service a pour valeur un ensemble de références (autrement dit, un ensemble d'OID) à des objets de type Employe: ce sont les employés qui travaillent dans le service. L'inverse est l'attribut référence serv de Employe. Nous verrons au cours suivant comment le standard de l'ODMG permet de déclarer explicitement les inverses comme des attributs de relations pour assurer la cohérence des références inverses.

3. Encapsulation des données, méthodes et persistance

Le concept d'encapsulation constitue l'une des principales caractéristiques des langages et des systèmes OO. Il est également lié aux notions de types de données abstraits et de masquage de l'information dans les langages de programmation.

Dans les systèmes et les modèles de bases de données traditionnels, ce concept n'est pas appliqué, puisqu'il est habituel de rendre les objets d'une base visibles aux utilisateurs et aux programmes externes. Dans ces modèles traditionnels, un certain nombre d'opérations standard sont applicables aux objets de tous types. Dans le modèle relationnel par exemple, les opérations de sélection, d'insertion, de modification et de suppression des tuples sont génériques, et elles sont applicables à toute relation de la base de données. La relation et ses attributs sont visibles pour les utilisateurs et les programmes externes qui accèdent à la relation au moyen de ces opérations.

3.1 Spécification du comportement des objets via des opérations de classe

Les concepts de masquage de l'information et d'encapsulation sont applicables aux objets d'une base de données. L'idée principale consiste à définir le **comportement** d'un type d'objets en fonction des **opérations** qu'il est possible d'appeler de l'extérieur sur les objets de ce type. La structure interne de l'objet est masquée, et l'objet n'est accessible qu'à travers un certain nombre d'opérations prédéfinies.

Certaines opérations servent à créer ou détruire des objets; d'autres peuvent mettre à jour l'état de l'objet; d'autres encore peuvent servir à extraire certaines parties de l'état de l'objet ou à effectuer certains calculs. Enfin, d'autres opérations peuvent exécuter une combinaison d'extraction, de calcul et de mise à jour. En général, l'**implémentation** d'une opération peut être spécifiée dans tout *langage de programmation généraliste* qui offre suffisamment de souplesse et de puissance pour les définir.

Les utilisateurs externes de l'objet n'ont connaissance que de l'**interface** du type d'objets, qui définit le nom et les arguments (paramètres) de chaque opération. L'implémentation est masquée aux utilisateurs externes; elle comprend la définition de la structure de données interne de l'objet et la façon dont les opérations vont accéder à ces données. Dans la terminologie OO, la partie interface de chaque opération se nomme la **signature**, tandis que l'implémentation de l'opération est une **méthode**. En général, on appelle une méthode en envoyant un **message** à l'objet pour qu'il exécute la méthode correspondante. L'exécution d'une méthode peut impliquer l'émission d'un message vers un autre objet; ce mécanisme peut être exploité pour retourner des valeurs des objets vers l'environnement externe ou vers d'autres objets.

Pour les applications de bases de données, l'exigence que tous les objets soient complètement encapsulés est trop stricte. Une façon de l'assouplir consiste à diviser la structure des objets en attributs **visibles** et en attributs **masqués** (variables d'instance). Les attributs visibles sont directement accessibles aux opérateurs externes, ou

par un langage de requête de haut niveau. Les attributs masqués sont complètement encapsulés, et ne sont accessibles qu'au moyen d'opérations prédéfinies. La plupart des SGBDOO emploient des langages de requête de haut niveau pour accéder aux attributs visibles. Nous décrirons au cours suivant le langage **OQL**, que l'**ODMG** a proposé comme langage de requête standard.

Dans la plupart des cas, les opérations qui mettent à jour l'état d'un objet sont encapsulées. Il s'agit d'une façon de définir la sémantique de la mise à jour des objets, car, dans de nombreux modèles OO, peu de contraintes d'intégrité sont prédéfinies dans le schéma. Chaque type d'objets a ses contraintes d'intégrité programmées dans les méthodes qui créent, suppriment et modifient les objets, une portion de code étant explicitement écrite pour détecter les violations de contraintes et gérer les exceptions. Dans de tels cas, toutes les mises à jour sont implémentées par des opérations encapsulées. Plus récemment, l'ODL associé au standard ODMG a autorisé la spécification de certaines contraintes communes, comme les clés et les relations inverses (intégrité référentielle), afin que le système puisse les appliquer automatiquement (voir cours suivant).

On utilise souvent le terme de **classe** pour désigner la définition d'un type d'objets ainsi que la spécification des opérations de ce type. Le listing 2.2 montre comment les définitions de types du listing 2.1 peuvent être étendues avec des opérations pour définir des classes. Un certain nombre d'opérations sont déclarées pour chaque classe, et la signature (interface) de chaque opération est incluse dans la définition de la classe. Une méthode (implémentation) pour chaque opération doit être définie ailleurs, dans un langage de programmation. Les opérations classiques comprennent les **constructeurs d'objets**, qui servent à créer de nouveaux objets et les **destructeurs d'objets**, qui servent à les supprimer. On peut déclarer un certain nombre de **mutateurs** pour modifier les états (valeurs) des différents attributs d'un objet et des **accesseurs** qui permettent d'extraire des informations sur les objets.

```
define class Employe:
        type tuple (    prenom:             string;
                        deuxiemeprenom:     char;
                        nom:                string;
                        noss:               string;
                        datenaissance:              Date;
                        adresse:                    string;
                        sexe:               char;
                        salaire:                    float;
                        Superviseur:        Employe;
                        serv:               Service; );
        operations      age:            integer;
                        creer_emp:      Employe;
                        suppr_emp:      boolean;
end Employe;
define class Service:
        type tuple (    snom:               string;
                        snumero:            integer;
                        dir:                tuple (         directeur: Employe;
                                                    datedebut: Date; );
                        sites:              set(string);
                        employes:           set(Employe);
                        projets :                   set(Projet); );
        operations      nombre_employes:    integer;
                        créer_serv:         Service;
                        Suppr_serv:         boolean;
                        affecter_emp(e: Employe):boolean;  (* ajoute un employé au service *)
                        suppr_emp(e: Employe):      boolean;        (* supprime un employé du service *)
end Service;
```

Listing 2.2 : Ajout d'opérations aux définitions d'Employe et de Service

On applique généralement une opération à un objet en employant la **notation pointée**. Par exemple, si d est une référence à un objet Service, nous pouvons invoquer une opération *nombre_employes* en écrivant *d.nombre_employes*. De même, *d.suppr_serv* supprime l'objet référencé par d. La seule exception est le constructeur, qui retourne une référence au nouvel objet de type Service. En conséquence, on attribue

habituellement par défaut au constructeur le même nom que la classe, même si cette convention n'est pas appliquée au listing 2.2. On utilise également la notation pointée pour référencer les attributs d'un objet, en écrivant par exemple *d.snumero* ou *d.dir.datedebut*.

3.2 Spécification de la persistance des objets via le nommage et l'accessibilité

Un SGBDOO est souvent étroitement couplé à un LPOO. Le LPOO sert à spécifier les implémentations des méthodes ainsi que le reste du code de l'application. Un objet est généralement créé par un programme applicatif qui invoque le constructeur de l'objet. Tous les objets ne sont pas destinés à être mémorisés de manière permanente dans la base. Les **objets temporaires** existent dans le programme qui s'exécute et disparaissent une fois que celui-ci s'achève. Les **objets persistants** sont stockés dans la base et continuent d'exister après la fin de l'exécution du programme. Les mécanismes qui permettent de rendre les objets persistants sont le *nommage* et l'*accessibilité*.

Le mécanisme de **nommage** consiste à donner à un objet un nom persistant unique qui permet à un programme d'y accéder. Ce nom persistant peut être affecté par une instruction ou une opération spécifique du programme (voir listing 2.3).

```
define class EnsembleServices:
        type        set (Service);
        operations  ajout_ serv(d: Service):    boolean;    (* ajoute un service à l'objet EnsembleServices *)
                    suppr_serv(d: Service):     boolean;    (* supprime un service de l'objet EnsembleServices *)
                    creer_ens_serv:             EnsembleServices;
                    suppr_ens_serv:             boolean;
end EnsembleServices;
….
persistent name TousServices: EnsembleServices; (* TousServices est un objet persistant nommé de type EnsembleServices *)
….
S:= creer_serv;                         (* créer un nouvel objet Service dans la variable S *)
b:= TousServices.ajout_serv(S);         (* rendre S persistant en l'ajoutant à l'ensemble persistant TousServices*)
```

Listing 2.3 : Création d'objets persistants grâce au nommage et à l'accessibilité

Tous les noms attribués aux objets doivent être uniques pour une base donnée. En conséquence, les objets persistants nommés sont utilisés comme **points d'entrée** par lesquels les utilisateurs et les applications peuvent accéder à la base. De toute évidence, il n'est pas évident de nommer tous les objets d'une grosse base de données, qui peut en contenir des milliers. On rend donc la plupart des objets persistants en utilisant un second mécanisme, nommé **accessibilité**. Ce mécanisme fonctionne en permettant d'atteindre un objet à partir d'un objet persistant. Un objet B est dit **accessible** depuis un objet A si une suite de références dans le graphe de l'objet conduit de l'objet A à l'objet B. Par exemple, tous les objets de la figure 2.1 sont accessibles depuis l'objet O_8. En conséquence, si O_8 est rendu persistant, tous les autres objets de cette figure deviennent également persistants.

Si nous commençons par créer un objet persistant nommé N, dont l'état est un ensemble (set) ou une liste (list) d'objets d'une classe C, nous pouvons rendre les objets de C persistants en les ajoutant à l'ensemble ou à la liste, et en les rendant ainsi accessibles à partir de N. En conséquence, N définit une **collection persistante** d'objets de la classe C. Par exemple, nous pouvons définir une classe EnsembleServices (voir listing 2.3), dont les objets sont de type **set**(Service). Supposons qu'un objet de type EnsembleServices soit créé, et qu'il soit nommé *TousServices*, ce qui le rend persistant. Tout objet Service ajouté à l'ensemble de TousServices via l'opération ajout_serv devient persistant du seul fait qu'il est accessible depuis TousServices. L'objet TousServices est nommé **extension** de la classe Service, puisqu'il va contenir tous les objets persistants de type Service. L'ODL de l'ODMG offre au concepteur du schéma la possibilité de nommer une extension dans le processus de déclaration de la classe (voir cours suivant).

Il existe une différence entre les modèles de bases de données traditionnels et les modèles OO. Dans les modèles traditionnels, comme le modèle relationnel ou le modèle EER, tous les objets sont supposés persistants. En conséquence, quand un type d'entités ou une classe comme EMPLOYE est défini dans le modèle EER, il représente à la fois la *déclaration de type EMPLOYE* et l'*ensemble persistant de tous* les objets

EMPLOYE. Dans l'approche OO, la déclaration d'EMPLOYE ne spécifie que le type et les opérations d'une classe d'objets. L'utilisateur doit définir séparément un objet de type set (EMPLOYE) ou list (EMPLOYE) dont la valeur est la *collection de références* à tous les objets EMPLOYE persistants. Cela permet aux objets temporaires et persistants de suivre les mêmes déclarations de type et de classe de l'ODL et du LPOO. En général, il est possible de définir plusieurs collections persistantes pour la même définition de classe.

4. Hiérarchies de types et de classes et héritage

Une autre caractéristique centrale des systèmes de bases de données OO est qu'ils permettent les hiérarchies de types et l'héritage. Une hiérarchie de types implique habituellement une contrainte sur les extensions correspondant aux types de la hiérarchie. Nous commencerons par aborder les hiérarchies de types à la section 4.1, puis les contraintes sur les extensions à la section 4.2. Nous nous appuierons sur un autre modèle OO, dans lequel les attributs et les opérations sont traités uniformément, puisque les attributs comme les opérations peuvent être hérités.

4.1 Hiérarchies de types et héritage

La plupart des applications de bases de données contiennent de nombreux objets de même type ou classe. En conséquence, les bases de données OO doivent offrir la possibilité de classifier les objets en fonction de leur type, comme dans les autres bases de données. Mais les bases de données OO ont une autre exigence: autoriser la définition de nouveaux types basés sur d'autres types prédéfinis, débouchant sur une **hiérarchie de types** (ou **classes**).

En général, on définit un type en lui affectant un nom, puis en spécifiant un certain nombre d'attributs (variables d'instance) et d'opérations (méthodes). Dans certains cas, les attributs et les opérations sont collectivement qualifiés de fonctions, puisque les attributs ressemblent à des fonctions qui auraient zéro argument. Un nom de fonction peut servir à référencer la valeur d'un attribut ou la valeur résultant d'une opération. Dans cette section, nous employons le terme de **fonction** pour désigner à la fois les attributs et les opérations d'un type d'objets, dans la mesure où ils sont traités de façon semblable dans une introduction de base à l'héritage.

Dans sa forme la plus simple, un type peut être défini par un nom de type suivi de la liste de ses fonctions visibles (publiques). Lorsque nous spécifions un type dans cette section, nous appliquons le format suivant, qui ne spécifie pas les arguments des fonctions pour plus de simplicité:

- NOM_DE_TYPE: fonction, fonction, ..., fonction

Par exemple, un type décrivant les caractéristiques d'une personne peut être défini comme suit:

- PERSONNE: Nom, Adresse, DateNaissance, Age, NoSS

Dans le type PERSONNE, les fonctions Nom, Adresse, NoSS et DateNaissance peuvent être implémentées sous forme d'attributs stockés, tandis que la fonction Age peut être une méthode qui calcule l'âge à partir de la fonction DateNaissance et de la date courante.

Le concept de **sous-type** est utile lorsque le concepteur ou l'utilisateur doit créer un nouveau type similaire mais non identique à un type déjà défini. Le sous-type hérite alors des fonctions du type prédéfini, que nous appellerons **supertype**. Supposons par exemple que nous voulions définir deux nouveaux types EMPLOYE et ETUDIANT comme suit:

- EMPLOYE: Nom, Adresse, DateNaissance, Age, NoSS, Salaire, DateEmbauche, Anciennete
- ETUDIANT: Nom, Adresse, DateNaissance, Age, NoSS, Département, MP

Comme les types ETUDIANT et EMPLOYE comprennent toutes les fonctions définies pour PERSONNE, plus des fonctions spécifiques qui leur sont propres, nous pouvons déclarer que ce sont des **sous-types** de PERSONNE. Chacun hérite des fonctions de PERSONNE précédemment définies, qui sont Nom, Adresse, DateNaissance, Age et NoSS. Pour ETUDIANT, il suffit de définir de nouvelles fonctions (locales), Département et MP, qui ne sont pas héritées. Département sera vraisemblablement un attribut stocké, tandis que MP pourra être implémenté sous forme de méthode qui calcule la moyenne de points de l'étudiant en accédant aux valeurs de Note qui sont mémorisées de façon interne (masquées) dans chaque objet ETUDIANT sous forme d'attributs privés. Pour EMPLOYE, les fonctions Salaire et DateEmbauche peuvent être des attributs stockés, alors que Anciennete peut être une méthode qui calcule l'ancienneté à partir de DateEmbauche.

La notion de définition de type implique de spécifier toutes ses fonctions et de les implémenter, soit sous forme d'attributs, soit sous forme de méthodes. Lorsqu'un sous-type est défini, il peut alors hériter de toutes ces fonctions et de leurs implémentations. Seules les fonctions qui sont spécifiques ou locales au sous-type, et qui ne sont donc pas spécifiées dans le supertype, doivent être définies et implémentées.

De ce fait, nous pouvons déclarer EMPLOYE et ETUDIANT comme suit:

- EMPLOYE **subtype-of** PERSONNE: Salaire, DateEmbauche, Anciennete
- ETUDIANT **subtype-of** PERSONNE: Département, MP

En général, un sous-type comprend toutes les fonctions qui sont définies pour son supertype, et certaines fonctions supplémentaires qui ne sont spécifiques qu'au sous-type. En conséquence, il est possible de générer une **hiérarchie de types** pour représenter les relations supertype/sous-type parmi tous les types déclarés dans le système.

Pour prendre un autre exemple, considérons un type qui décrit des objets de la géométrie plane, que l'on peut définir comme suit:

- OBJET_GEOMETRIQUE: Forme, Aire, PointDeReference

Pour le type OBJET_GEOMETRIQUE, Forme est implémenté sous forme d'attribut (son domaine peut être un type énuméré avec les valeurs 'triangle', 'rectangle', 'cercle', etc.), et Aire est une méthode qu'on applique pour calculer une surface. PointDeReference spécifie les coordonnées d'un point qui détermine l'emplacement de l'objet. Supposons maintenant que nous voulions définir un certain nombre de sous-types de l'objet OBJET_GEOMETRIQUE comme suit:

- RECTANGLE **subtype-of** OBJET_GEOMETRIQUE: Largeur, Hauteur
- TRIANGLE **subtype-of** OBJET_GEOMETRIQUE: Cote1, Cote2, Angle
- CERCLE **subtype-of** OBJET_GEOMETRIQUE: Rayon

L'opération Aire peut être implémentée par une méthode différente pour chaque sous-type, puisque la procédure de calcul de l'aire est différente pour les rectangles, les triangles et les cercles. De même, l'attribut PointDeReference peut avoir une signification différente pour chaque sous-type: il peut s'agir du centre pour les objets CERCLE et RECTANGLE, et de l'un des sommets pour un objet TRIANGLE. Certains systèmes de bases de données OO autorisent le **renommage** des fonctions héritées dans différents sous-types pour refléter plus étroitement la signification.

Une autre façon de déclarer ces trois sous-types consiste à spécifier la valeur de l'attribut Forme, avec une condition qui doit être satisfaite pour les objets de chaque sous-type:

- RECTANGLE **subtype-of** OBJET_GEOMETRIQUE: (Forme='rectangle'):Largeur, Hauteur
- TRIANGLE **subtype-of** OBJET_GEOMETRIQUE: (Forme='triangle'):Cote1, Cote2, Angle
- CERCLE **subtype-of** OBJET_GEOMETRIQUE: (Forme='cercle'):Rayon

Seuls les objets OBJET_GEOMETRIQUE pour lesquels Forme='rectangle' appartiennent au sous-type RECTANGLE, et il en va de même pour les deux autres sous-types. Dans ce cas, toutes les fonctions du supertype OBJET_GEOMETRIQUE sont héritées par chacun des trois sous-types, mais la valeur de l'attribut Forme est restreinte à une valeur spécifique pour chacun.

Les définitions de types décrivent des objets, mais n'en génèrent pas. Ce ne sont que des déclarations, au sein desquelles l'implémentation des fonctions de chaque type est spécifiée. Une application de bases de données contient de nombreux objets de chaque type. Lorsqu'un objet est créé, il appartient généralement à un ou plusieurs types déclarés. Par exemple, un objet CERCLE est de type CERCLE et de type OBJET_GEOMETRIQUE (par héritage). Chaque objet devient également membre d'une ou plusieurs collections persistantes d'objets (ou d'extensions), qui servent à grouper des ensembles d'objets qui sont significatifs pour l'application de base de données.

4.2 Contraintes sur les extensions correspondant à une hiérarchie de types

Dans la plupart des bases de données OO, la collection d'objets d'une extension est du même type, mais ce n'est pas une condition nécessaire. Smalltalk, par exemple, qui est un langage OO à *typage faible*, permet à une collection de contenir des objets de types différents. Ce peut être également le cas quand on ajoute des concepts OO à des langages non orientés objet et dépourvus de notion de type, comme Lisp. Mais comme la majorité des bases de données OO prennent en charge les types, nous partirons du principe que les **extensions** sont des collections d'objets de même type dans le reste de cette section.

Dans les applications de bases de données, il est courant que chaque type ou sous-type ait une collection associée, qui contiendra l'ensemble des objets persistants de ce type ou sous-type. Dans ce cas, la contrainte est que chaque objet d'une extension qui correspond à un sous-type soit aussi un membre de l'extension qui correspond à son supertype. Certains systèmes de bases de données OO ont un type système prédéfini (la classe ROOT ou la classe OBJECT), dont l'extension contient tous les objets du système. La classification procède alors en ajoutant des sous-types significatifs pour l'application, créant ainsi une **hiérarchie de types** ou **hiérarchie de classes** du système. Toutes les extensions des classes définies par le système et l'utilisateur sont des sous-ensembles de l'extension correspondant à la classe OBJECT, directement ou indirectement. Dans le

modèle de l'ODMG, l'utilisateur peut ou non spécifier une extension pour chaque classe (type) en fonction de l'application.
Dans la plupart des systèmes OO, on établit une distinction entre collections persistantes et collections temporaires. Une **collection persistante** contient un ensemble d'objets stockés en permanence dans la base de données, et de ce fait accessibles et partageables par plusieurs programmes. Une **collection temporaire** existe provisoirement durant l'exécution d'un programme mais n'est pas conservée lorsque celui-ci se termine. On peut par exemple créer une collection temporaire pour contenir les résultats d'une requête qui extrait des objets d'une collection permanente et les copie dans la collection temporaire. La collection temporaire contient les mêmes types d'objets que la collection permanente. Le programme peut alors manipuler les objets dans la collection temporaire, qui cesse d'exister quand le programme se termine. En général, de nombreuses collections (temporaires ou persistantes) peuvent contenir des objets du même type.
Les constructeurs de types décrits à la section 2 permettent à l'état d'un objet d'être une collection d'objets. Les collections d'objets basées sur le constructeur **set** peuvent ainsi définir plusieurs collections, correspondant chacune à un objet. Les objets d'une collection de type set sont eux-mêmes membres d'une autre collection. Cette caractéristique permet de créer des schémas de classification à plusieurs niveaux, dans lesquels un objet d'une collection donnée a pour état une collection d'objets d'une classe différente.
Le modèle de l'ODMG distingue l'héritage de type, appelé héritage d'interface et dénoté par le symbole « : », de la contrainte d'héritage de l'extension, dénotée par le mot clé EXTEND (voir cours suivant).

5. Résumé

Dans ce cours, nous avons présenté les concepts de l'approche orientée objet des systèmes de bases de données, proposée pour répondre aux besoins des applications complexes et ajouter des fonctionnalités de bases de données aux langages de programmation orientés objet, tels que C++. Ces concepts sont les suivants:

- *Identité des objets.* Les objets ont une identité unique qui ne dépend pas de la valeur de leurs attributs.
- *Constructeurs de types.* On peut construire des structures d'objets complexes en appliquant récursivement un ensemble de constructeurs de base, tels que tuple, set, list et bag.
- *Encapsulation des opérations.* La structure des objets et les opérations qui peuvent être appliquées aux objets sont incluses dans les définitions des classes d'objets.
- *Compatibilité avec les langages de programmation.* Les objets, qu'ils soient persistants ou temporaires, sont gérés de façon homogène. On rend les objets persistants en les rattachant à une collection ou en les nommant explicitement.
- *Hiérarchies de types et héritage.* On peut spécifier les types d'objets en construisant une hiérarchie de types qui permet d'hériter des attributs et des méthodes des types prédéfinis. Certains modèles autorisent l'héritage multiple.
- *Extensions.* Tous les objets persistants d'un type donné peuvent être stockés dans une extension. On applique des contraintes« ensemble/sous-ensemble» aux extensions correspondant à une hiérarchie de types.