Université A/Mira
Département Mathématiques
Module : BDDA, niveau : Master 1 PSA

Le Modèle Objet

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. Certains sont obligatoires, d'autres optionnels. Ce modèle est proche du modèle de classe de C++, qui peut être vu comme une implémentation des types abstraits de données. Il est aussi très proche du modèle objet du langage Java.

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 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.
À 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 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 de 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.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.

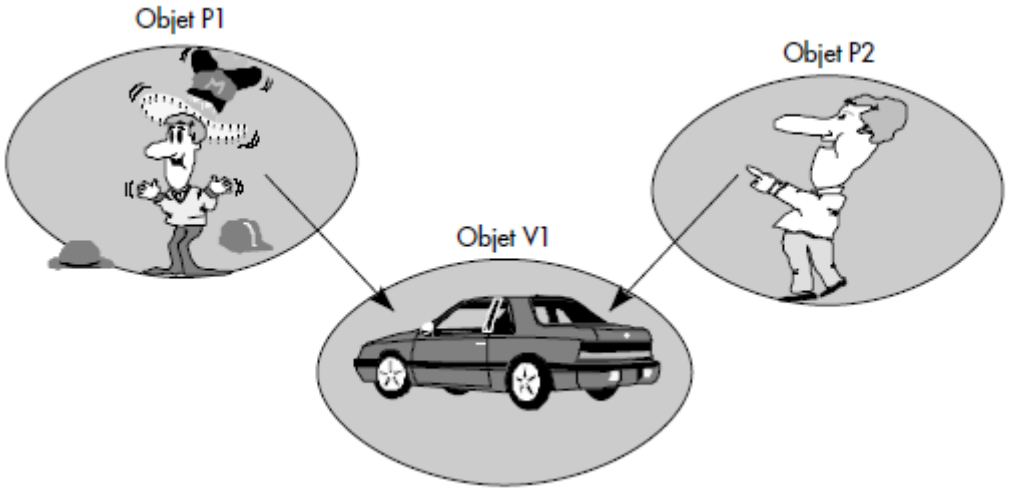


Figure 1 : Partage référentiel d'un objet par deux autres objets.

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 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.

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.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.

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.

Notion 5 : 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.

Exemple :

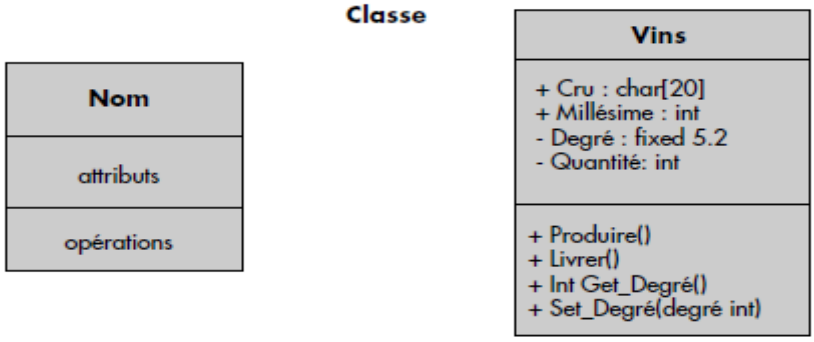


Figure 2 : exemple de classe.

La figure 3 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, ils 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; };
```

Figure3. 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 modélise des associations entre classes sans introduire de classes intermédiaires artificielles.

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.

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 6 : 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 noeuds 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 4 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 5. Les super-classes d'une classe sont définies après la déclaration de la classe suivie de « : ». À 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.

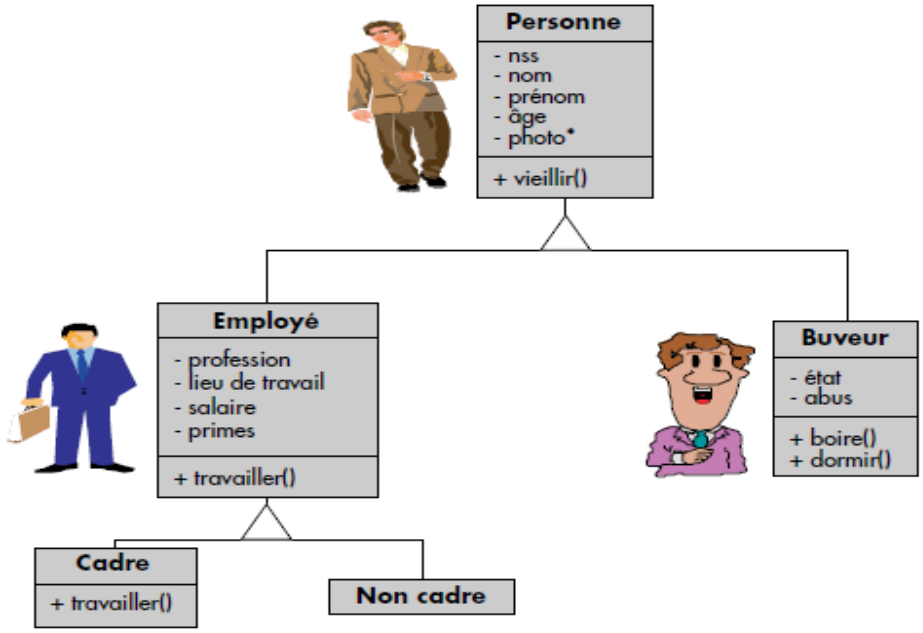


Figure 4 : *Exemple de graphe de généralisation.*

```
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 5 : *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.

Notion7 : 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 superclasses. L'héritage multiple permet de définir des classes intersection comme dans la figure 6, où les Employés Buveurs héritent à la fois des Buveurs et des Employés. Ils héritent certes d'un seul nom provenant de la racine Personne.

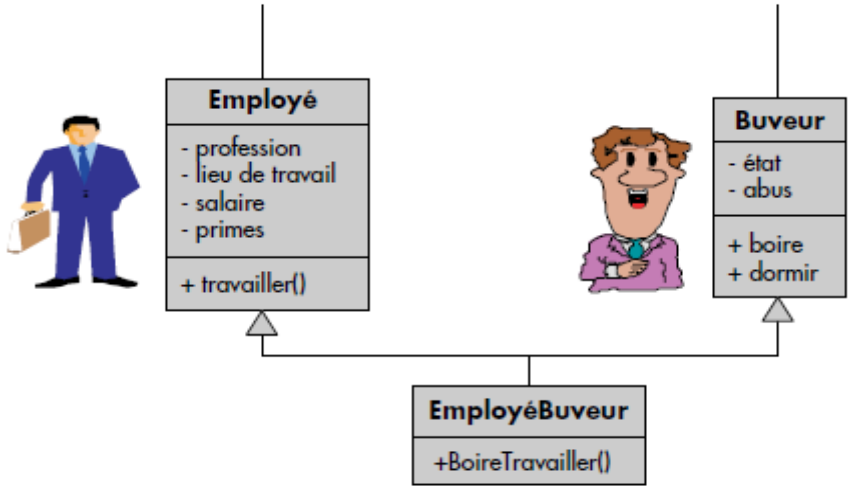
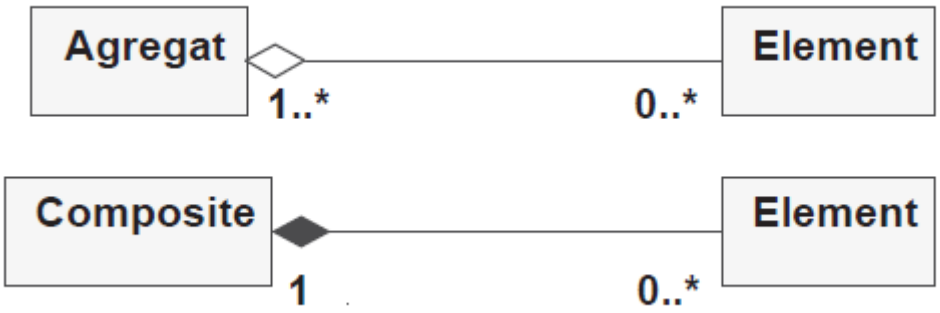


Figure 6 : Exemple *d'héritage multiple*

AGRÉGATION ET COMPOSITION
Une agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance. Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « contient », « est composé de ».
Une composition est une agrégation plus forte impliquant que :
• un élément ne peut appartenir qu'à un seul agrégat composite (agrégation non partagée) ;
• la destruction de l'agrégat composite entraîne la destruction de tous ses éléments (le composite est responsable du cycle de vie des parties).



Exemple : Agrégation et composition