Bases de Données Avancées, M1 Informatique et Miage

S. Cerrito

Année 2009-2010, Evry

Plan du Cours

1. Introduction et rappels du modèle relationnel
2. Modèles à objet et objet-relationnel
3. XML et le Modèle “Données Semi-structurées”
4. Médiation de donnés.

INTRODUCTION

Historique

– Avant 1970 : BD=fichiers d'enregistrements, "modèles" *réseaux* et *hiérarchique* ; pas de vraie indépendance logique/physique.
– En 1970 : modèle *relationnel* (Codd) : vraie indépendance logique/physique.
– Années 80 et 90 : nouveaux modèles :
  modèle à objets et object-relationnel
  modèle à base de règles (Datalog)
– Fin années 90 : données dites *semi-structurées* (XML).

Centre de ce cours : modèle à objets et object-relationnel, puis modèle semi-structuré.

1 Notions essentielles des BD relationnelles

Mots clés :

– Univers U, Attributs $A_1, \cdots, A_n$
– Domaine $Dom(A)$ d'un attribut A
– Schéma d'une relation dont le nom est R.
– n-uplet sur un ensemble E d'attributs
– Relation (ou "table") sur un schéma de relation
– Schéma d'une BD
– Base de données B sur un schéma de base

Un *univers* U est un ensemble fini et non-vide de noms, dits *attributs*.

Le *domaine* d'un attribut A ($Dom(A)$) est l'ensemble des valeurs possibles associé à A.

Exemple :

$U = \{NomFilm, Realisateur, Acteur, Producteur, NomCinema, Horaire\}$

$Dom(NomFilm) = Dom(Realisateur) = Dom(Acteur) = Dom(Producteur) = Dom(NomCinema)$= chaînes de caractères.

$Dom(Horaire) = \{h.m \mid h \in [1, \cdots, 24],\ m \in [0, \cdots, 60]\}$

Un *schéma* d'une relation dont le nom est R est un sous-ensemble non-vide de l'univers U.

Suite de l'exemple :

– Schéma de la relation $Film$=$\{NomFilm, Realisateur, Acteur, Producteur\}$
– Schéma de la relation $Projection$=$\{NomFilm, NomCinema, Horaire\}$

Intuition : Format de deux tables.

$Film$:

NomFilm	Realisateur	Acteur	Producteur
⋮	⋮	⋮	⋮

$Projection$:

NomFilm	NomCinema	Horaire
⋮	⋮	⋮

Soit $E = \{A_1, \cdots, A_n\}$ le schéma d'une relation. Un n-*uplet* n sur E est une fonction $E \to Dom(A_1) \cup \cdots \cup Dom(A_n)$ telle que, pour tout $A_i \in E$, $n(A_i) \in Dom(A_i)$.

Si $E' \subset E$, la restriction de n à E se note $n(E')$.

Exemple.

Un n-uplet possible sur le schéma de $Projection$:
$\langle$“$Jugez - moi\ coupable''$, “$Gaumont\ Alesia''$, $13.35\rangle$.

Sa restriction à $\{NomCinema, NomFilm\}$:
$\langle$ “Jugez-moi coupable”, “Gaumont Alesia”$\rangle$.

Pourquoi définir un n-uplet comme une *fonction* plutôt que comme un élément de $Dom(A_1) \times \cdots \times Dom(A_n)$?

Une *relation* (table) r sur un schéma de relation S est un ensemble d'n-uplets sur S. On dit aussi : S est le schéma de r.

Exemple.

$Film$:

NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

$Projection$:

NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

Un schéma $\mathcal{S}$ *d'une base* sur un univers U est un ensemble non-vide d'expressions de la forme $N(S)$ où S est un schéma de relation et N un nom de relation.

Exemple(on omet les $\{\}$).

$U =$
$\{NomFilm, Realizateur, Acteur, Producteur, NomCinema, Horaire, Spectateur\}$

$$\mathcal{S} =$$

$\{$
$Film(NomFilm, Realizateur, Acteur, Producteur),$
$Projection(NomFilm, NomCinema, Horaire), Aime(Spectateur, NomFilm)$
$\}$

Schéma de la base = Format des données de la base.

Quel est le format de la base de l'exemple ?

– Une *base de données* B sur un schéma de base $\mathcal{S}$ (avec univers U) est un ensemble de relations finies $r_1, \cdots r_n$ où chaque r_i est associée à un nom de relation N_i et est telle que si $N_i(S) \in \mathcal{S}$, alors r_i a S comme schéma.
– On peut aussi imposer des *contraintes* sur les données. Par exemple : les *dépendances fonctionnelles*, qui fixent, entre autres, les *clés* des relations (cours SGBD L3).
– Ces contraintes, dites d'*intégrité*, font aussi partie de la spécification du format des données de la base.

Exemple d'une base.

$Film$			
NomFilm	Réalisateur	Acteur	Producteur
nf1	r1	a1	p1
nf1	r1	a2	p1
nf2	r2	a1	p2
nf3	r2	a1	p2

$Projection$		
NomFilm	NomCinema	Horaire
nf1	nc1	h1
nf1	nc2	h2
nf2	nc1	h3
nf3	nc2	h1

$Aime$	
NomFilm	Spectateur
nf1	s1
nf1	s2
nf2	s1
nf3	s3

2 Fondements des Langages de Requête (qque soit le modèle)

- Informellement : *Requête sur une base* = question que l'on pose à la base.
- *Langage de requête*= langage permettant d'écrire des requêtes
- Importance d'un langage de requête formel et rigoureux :

1. Conception de langages commerciaux
2. Evaluation de la puissance d'expression de chaque langage commercial
3. Possibilité de déterminer ce qu'un langage commercial ne pourra pas exprimer
4. Notion d'équivalence entre deux expressions de requête $\Rightarrow$ Optimisation "logique" de l'évaluation d'une requête

Deux langages formels pour le modèle relationnel : *algèbre relationnelle* et *calcul relationnel* (cours SGBD L3).

Question : Et pour d’autres modèles de données ?

2.1 Les opérateurs de l'algèbre relationnelle

– Opérateurs ensemblistes : union ($\cup$), intersection ($\cap$), différence ($\setminus$), produit cartésien ($\times$)
– projection sur un ensemble d'attributs E (π_E), sélection d'un ensemble de n-uplets selon une condition C (σ_C), jointure "naturelle" ($\bowtie$), division ($\div$), renommage (ρ).

3 Limites du modèle relationnel

1. On ne peut pas imbriquer les informations
2. La structure du schéma est très rigide
3. On ne peut pas exprimer la clôture transitive d'une relation (par ex. vol(départ, arrivée) par rapport à vol_direct(départ, arrivée)

Commençons par (1).

En relationnel (“première forme normale”) :

$OPERAS$:

Auteur	*Titre*	*Langue*
Mozart	La Flûte Enchantée	Allemand
Mozart	Don Juan	Italien
Mozart	Les noces de Figaro	Italien
Bizet	Carmen	Français
Bizet	Djamileh	Français

Redondance.

Si on imbrique :

$OPERAS$:

Auteur	*Opéra*	
Mozart	*Titre*	*Langue*
	La Flûte Enchantée	Allemand
	Don Juan	Italien
	Les noces de Figaro	Italien
Bizet	*Titre*	*Langue*
	Carmen	Français
	Djamileh	Français

4 Modèle à Objets

– Extension de concepts de langages comme C^{++} or Java au cas des BD, où la *persistance* des données est primordiale.
– Concepts clés :
  – types,
  – classes et objets,
  – identité des objets,
  – héritage.

Ici, on choisi *ODL* (Object Definition Language) comme langage de spécification de la structure d'une BD à objets → Ecriture du **schéma**.

En relationnel : en SQL, `create table` permet de spécifier une table.

En objet : une déclaration ODL permet de spécifier une classe.

Types

Types atomiques. Integer, float, char, string, boolean et les *énumérations*.

Syntaxe d'un type enuméré : `enum NomType el1,...,elN`.
Par ex. : `enum CoulDrapeauFrbleau,blanc,rouge`

Une classe aussi est un type atomique (voir après ce que c'est une classe, ici).

Constructeurs de Types :

- `Set<Type>`. *Ex.* : `Set<integer>`. *NB* : ensembles finis.
- `Bag<Type>`. Si T est un type, `Bag<`T`>` est un type T' dont les valeurs sont des *multi-ensembles finis* d'éléments de type T. *Ex.* : $\{1, 2, 1\}$ est de type `Bag<integer>`.
- `List<Type>`. *Ex.* : `List<char>` (les chaînes de caractères), `List<integer>`. *NB* : listes finies.
- `Array<Type, entier>` : type des tableaux de n (entier) éléments de type `Type`. *Ex.* : `Array<char, 10>`.
- `Dictionary<Type1, Type2>` : un type dont les valeurs sont des ensembles finis de couples $\langle$ clé, val $\rangle$, où clé est de type `Type1` et val est de type `Type2`.
- `Struct Nom` { `Type1 Nom1,...,TypeN NomN` }.
  Ex. : `Struct Adresse` { `string rue, string ville` }. (Type record !)

Possibilité d’imbriquer les constructeurs de types (la déf. des types est récursive !)

Par ex. :

```
Struct famille {
Set(char) enfants,
Struct père { char nom, char prenom} LePere,
Struct mère { char nom, char prenom } LaMère
}
```

Classes et Objets

– Une *classe* est un *type abstrait* : on définit les propriétés d'un objet et ce que l'objet peur faire (*méthodes*).
– Un objet o est une instance d'une classe C et a un et un seul idéntificateur (*OID*).
– Déclaration d'une classe en ODL :
`class` Nom = { liste de propriétés et méthodes }

– La sorte la + simple de propriété : un *attribut*.
– Dćlaration d'un attribut : on indique son type et sa valeur.
– *Exemple* :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
};
```

Attribut `SorteFilm` : de type énumération.

`couleurs` est le nom de ce type énuméré, tandis que le nom de l'attribut est `SorteFilm`.

– Un objet de cette classe : ("Autant en emporte le vent", 1939, 231, couleur)

Un autre exemple de classe :

```
class Star {
attribute string nom;
attribute Struct Adr { string rue, string ville } adresse
};
```

Ici l’attribut adresse est de type “structure” (record), ce type s’appelle “Adr” et le type des 2 champs est string.

Autre sorte de propriétés : Associations entre objets

– Mot clé dans la déclaration d'une classe : `relationship`
– Définition + riche de la classe `Film` :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs {couleur, noir&blanc} SorteFilm;
relationship Set<Star> acteurs;
};
```

Possibilité d'indiquer les acteurs d'un film donné (objet de la classe `Film`).

Une relationship : comme une association en EA.
Le type d'une relationship (ensemble "d'arrivée", ici : `Set<Star>`) : une classe, ou bien construit à partir d'une classe avec **un** constructeur de collection = tout constructeur sauf `Struct`.

Donc :

– Type d'un attribut : construit à partir des types atomiques avec autant de constructeurs qu'on veut, peu importe lesquels.
– Type (ensemble "d'arrivée") d'une relationship : une classe, ou construit à partir d'une classe avec un seul constructeur c, $c \neq$ `Struct`.

Associations INVERSES entre objets

– Mot clé : `inverse`
– Définition de classes encore + riches :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs {couleur, noir&blanc} SorteFilm;
relationship Set<Star> acteurs
inverse Star : :JoueDans};

class Star {
attribute string nom;
attribute Struct Adr = {string rue, string ville} adresse;
relationship Set<Film> JoueDans
inverse Film : :acteurs;
};
```

Notation C : :R signifie : la relationship R est une relation de la classe C.

Toute relationship ODL est binaire.

Relationship R *many-many* : le type de R et celui de son inverse sont un `Set` (ou un autre type collection). C'est le cas d'`acteurs` (et `JoueDans`) l'exemple précédent.

Autre example de relation many-many :

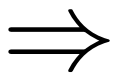
```
class Bière {
...;
relationship Set<Buveur> Fans
inverse Buveur : :Aime;}


class Buveur {
...;
relationship Set<Bière> Aime
inverse Bière : :Fans;
}
```

Une bière peut avoir plusieurs fans, et un buveur peut aimer plusieurs bières.

N.B `Fans` a comme type (= type du cible) un `Set` d'objets de `Buveur` et `Aime` a comme type un `Set` d'objets de `Bière`. La notation `: :` dit où sont définies leur respectives relations inverses :
`Buveur : :Aime` et `Bière : :Fans`.

Relationship R *many-one* : d’un coté `Set` (ou un autre type collection), de l’autre la classe.

On enrichie l’exemple précédent des bières et des buveurs en y ajoutant : une bière peut avoir plusieurs `super-fans`, mais un bière a une seule `bière-favorie` :



```
class Bière{
...;
relationship Set<Buveur> Fans
inverse Buveur : :Aime;
relationship Set<Buveur> super-fans
inverse Buveur : :bière-favorie;
}

class Buveur{
...;
relationship Set<Bière> Aime
inverse Bière : :Fans;
relationship Bière bière-favorie
inverse Bière : :super-fans;
}
```

Relationship R *one-one* : le type de la relation est une classe, des 2 cotés. Exemple : une fille a un seul époux et un garçon a une seule épouse.

```
class Fille{
...;
relationship Garçon Epoux
inverse Garçon : :Epouse;}


class Garçon{
...;
relationship Fille Epouse
inverse Fille : :Epoux;
}
```

N.B : l'expression `Garçon` joue 2 rôles différentes, dans la déf. de la classe `Fille`.

Question : est-il vrai ou faux que la définition donnée permet le célibat ?

Une classe C peut être telle que l'inverse d'une de ses relationships est défini dans C elle même.

```
class Personne {
...;
relationship Personne Epoux
inverse Epouse;
relationship Personne Epouse
inverse Epoux;}
```

N.B : Puisque on écrit : `relationship Personne Epoux inverse Epouse`, sans le `: :`, c'est implicite que `Epouse` est définie dans la même classe `Personne`.

Associations liant + que 2 classes

En ODL on peut spécifier seulement des relations binaires. Comment faire si l'on veut modéliser une relation R entre n classes, où $n > 2$? On introduit une nouvelle classe C qui joue le rôle de R et on définit n relations binaires entre C et chaque C_i.

Exemple On veut modéliser une relation *contrats* qui lie une star, un film et un studio.

On crée une nouvelle classe `Contrat` :

```
class Contrat {
attribute integer salary;
relationship Film leFilm
inverse ...;
relationship Studio leStudio
inverse ...;
relationship Star LaStar
inverse ...;
};
```

Ici, `leFilm` est le nom de la relation qui lie un objet de la classe `Contrat` au seul objet de la classe `Film` auquel **ce** contrat fait référence et Film est le “type d’arrivée” de cette relation (un contrat est lié à un objet de la classe `Film`).
Pour pouvoir remplir la partie ... qui suit `inverse` après la déclaration de la relation `leFilm` de la classe `Contrat`, en y écrivant, par ex. :

```
Film : :ContratsPour
```

il faut modifier la définition de la classe `Film`, en y ajoutant :

```
relationship Set<Contrat> ContratsPour inverse Contrat : :leFilm;
```

Méthodes

– *Méthode* = code exécutable qui peut être appliqué à un objet.
– En ODL, on n'écrit pas le code d'une méthode : on indique juste son nom et sa *signature* : les types des entrées/sorties.
– Comme dans les langages de programmation orientés objet, l'objet est un argument "caché" d'une méthode m.
– Une méthode m peut soulever des *exceptions*. Mot clé : `raises`.
– Les paramètres d'une méthode m sont spécifiés par :
(a) `in` (entrée)
(b) `out` (sortie).
Une méthode peut aussi (c) renvoyer une valeur.
Différence entre (b) et (c) : selon (b) la méthode m se comporte comme une procédure d'un langage impératif (C, par ex.), selon (c) m est comme une fonction. Si (b), alors et le type du résultat est `void` (vide), car on n'a pas calculé une fonction, mais effectué une action.

Exemple

Définition encore + riche de la classe `Film`, avec 3 méthodes :

```
class Film {
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Stars> acteurs
inverse Star : :JoueDans;
float longeurHeures() raises(ErreurHoraire);
void NomsActeurs(out Set<String>);
void AutresFilms(in Star, out Set<Film>)
raises(PasValable);;
};
```

Lire :

`float longeurHeures()` raises(ErreurHoraire)
ainsi : `longeurHeures` est une fonction, dont le type d'arrivée est `float`, et peut soulever des exceptions.

`NomActeurs` et `AutresFilms` sont des procédures.

`AutresFilms` prend en entrée un paramètre de type `Star`. Etant donné un objet f de la classe `Film` et une star s qui joue dans f, cette procedure affecte à une variable de type `Set<Film>` l'ensemble des autres fils dans les quels s joue (si possible).

4.1 Sous-classes et Héritage

Avec ODL on peut déclarer qu'une classe C est une *sous-classe* d'une autre classe D.

Mot-clé : `extends`

Par ex. :

```
class Cartoon extends Film {
relationship Set<Star> voix;
}
```

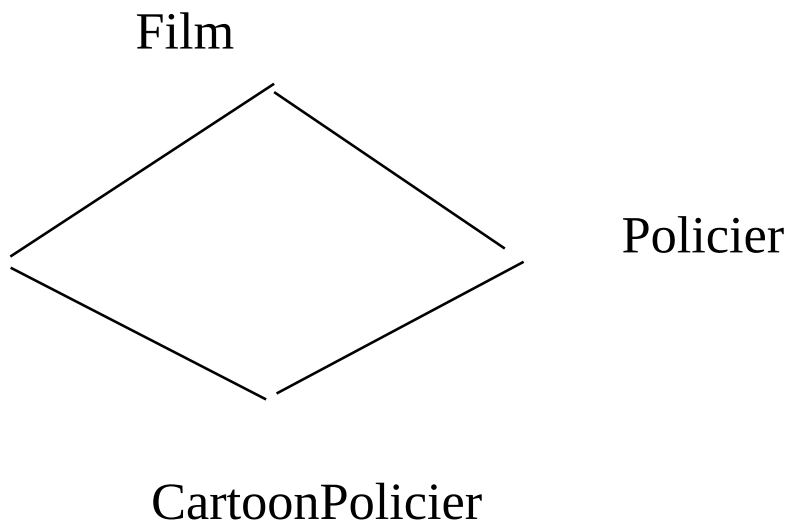
Ici, les objets de `Cartoon` ont, par rapport à `Film`, la relation nouvelle `voix` avec l'ensemble de stars qui donnent leur voix aux personnages.

De plus, une sous-classe C de D *hérite* toutes les propriétés de D. Dans l'exemple : tout objet de `Cartoon` a les attributs `titre, année, longueur, SorteFilm` et les relations `Acteurs`, etc.

Héritage Multiple ?

Supposons que nous avons défini aussi une classe `Policier`, qui est une autre sous-classe de `Film`. Le film *Roger Rabbit* appartient au même temps à `Cartoon` et `Policier`.

Comment faire en ODL ? Déclarer une nouvelle classe : `CartoonPolicier`



`extends` va être suivi par plusieurs noms de classes, séparées par “ :”

```
class CartoonPolicier extends Policier : Cartoon;
```

Héritage multiple ? Problème des conflits de noms

Exemple. Une sous-classe de `Film` qui s'appelle `FilmAmour` a un attribut `fin`, de type énuméré { `happy, triste` }. Une autre sous-classe de `Film` qui s'appelle `FilmTribunal` a aussi un attribut `fin`, de type énuméré { `coupable, innocent` }.
Ambiguïté pour la classe `FilmTribunalEtAmour`, sous-classe des 2.

Le standard ODL ne dicte pas quoi faire.

Possibilités :

– Interdire l'héritage multiple :-(
– Préciser la classe dont il faut hériter la signification de l'attribut `fin`
– Renommer un attribut. Par ex., attribut `fin` de `FilmTribunal` $\leadsto$ `verdict`

Un objet d'une classe C qui est une sous-classe de la classe D hérite aussi les méthodes de D.

C'est exactement comme dans le langages de programmation orientés objet.

Ensemble des définitions ODL décrivant les propriétés des classes et leur relations hiérachiques

=

Définition d’un **schéma** de base de donnés **à objets**

Extension d'une classe

Une définition d'une classe C en ODL spécifie le format (schéma) de la classe. Pour faire référence à l'ensemble des instances de la classe (l'"extension" de C)) et pouvoir interroger la base : mot clé `extent`.

```
class Film (extent Films)
attribute string titre;
⋮
```

N.B. : Expression `Film` $\neq$ Expression `Films`.
(On aurait pu utiliser `Totos`, à la place de `Films`, certes, mais pas `Film`).

Même si chaque objet a un'identité unique on PEUT vouloir traiter plusieurs objets comme "non-distinguables" par rapport aux propriétés observables, même si chacun garde sa propre identité.

C'est alors sensé de déclarer une *clé* : mot clé = `key`

```
class Film
(extent Films key (titre,année))
{
attribute string titre;
⋮
}
```

4.2 Un langage de Requête pour les BD Objet : OQL. Quelques Notions

– *Object Query Language*
– Notation à la SQL.
– Utilisé comme extension d'un langage de programmation hôte, comme C^{++} ou Java.

Format Général d'une requête OQL

```
SELECT liste d'expressions
FROM liste d'une ou plusieurs déclarations de variables.
WHERE condition C de selection
```

Une variable est déclarée en indiquant :

1. Une expression dont la valeur a un type collection, par ex. `set` ou `bag`.
2. Le nom de la variable.

L'expression (1) indique l'extension d'une classe, par exemple `Films`. Analogie avec une relation dans une requête SQL.

Notation “.”

Soit o un objet d’une classe C.

– Si p est un attribut, $o.p$ est la valeur de p pou o.
– Si p est une relation, $o.p$ est l’objet ou la collection d’objets reliés à o par p.
– Si p est une méthode (éventuellement avec paramètres $a_1, \cdots a_k$), $o.p(....)$ est le résultat de l’application de p à a.

N.B : Puisque la déclaration de la méthode `NomsActeurs` de la classe `Film` est :
`void NomsActeurs(out Set<String>);`
cette méthode est une procédure. Donc si `MonFilm` est un objet de la classe `Film`, l’expression `MonFilm.NomsActeurs(mesStars)` ne renvoie pas de valeur, mais, comme effet de bord, fait si que la valeur de la variable output `mesStars` de la méthode, qui est de type `Set<String>`, soit un ensemble de noms d’acteurs.

Exemple d’une BD à objet pur illustrer OQL

```
class Film
(extent Films key (titre, année))
{
attribute string titre;
attribute integer année;
attribute integer longueur;
attribute enum couleurs { couleur, noir&blanc } SorteFilm;
relationship Set<Star> acteurs inverse Star : :JoueDans;
relationship Studio appartient-à inverse Studio : :possède;
float longeurHeures() raises(pasLongueurTrouvée);
void NomsActeurs(out Set<String>);
void autresFilms (in Star, out Set<Film>) raises(PasActeur);
};

class Star
(extent Stars key nom)
{
attribute string nom;
attribute Struct Adr
{string rue, string ville} adresse;
relationship Set<Film> JoueDans inverse Film : :acteurs;
};

class Studio
(extent Studios key nom)
{
attribute string nom;
attribute string adresse;
relationship Set<Film> :possède inverse Film : :appartient-à;
};
```

Exemples de Requêtes OQL

```
SELECT m.année
FROM Films m
WHERE m.titre = ‘‘Autant en emporte le vent’’
```

Ici, `Films m` est une déclaration d’une variable `m` qui a sa valeur dans l’*extension* `Films` de la classe dont le nom est `Film`, i.e. : valeur(`m`) $\in$ `Films`.

Exemples de Requêtes OQL, suite

```
SELECT s.nom
FROM Films m, m.acteurs s
WHERE m.titre = ‘‘Casablanca’’
```

Dans l’évaluation, tous les couples (m,s) tels que m est un film et s est un acteur de m (selon la relation `acteurs` de `Film`) sont considérés :

FOR chaque m dans FILMS DO
FOR chaque s dans m.acteurs DO
IF m.titre = “Casablanca” THEN
ajouter s.nom au multi-ensemble résultat.

La condition après le `WHERE` limite l’espace de recherche pour `m` aux films dont le titre est “Casablanca”.

Exemples de Requêtes OQL, suite

```
SELECT DISTINCT s.nom
FROM Films m, m.acteurs s
WHERE m.appartient-à.nom = ‘‘Disney’’
```

Comme en SQL, le mot clé `DISTINCT` élimine le répétitions dans le multi-ensemble (bag) résultat.

Exemples de Requêtes OQL, suite

Liste des noms des films du studio Disney, ordonnés par leur longueur ; si deux films ont la même longueur, trier selon l'ordre alphabétique des titres.

```
SELECT m
FROM Films m
WHERE m.appartient-à.nom = ‘‘Disney’’
ORDER BY m.longueur, m.titre
```

Exemples de Requêtes OQL, suite

On veut l'ensemble des couples de stars qui vivent à la même adresse :

```
SELECT DISTINCT Struct(star1 :s1, star2 : s2)
FROM Stars s1, Stars s2
WHERE s1.adresse = s2.adresse AND s1.nom < s2.Nom
```

Pour chaque couple qui passe le test on produit un record, avec 2 champs, nommés `star1` et `star2`. Le type de chaque champ est la classe `Star`.

Le type du résultat final de cette requête est :

`Set<Struct{star1 : Star, star2 : Star}>`.

Exemples de Requêtes OQL, suite

Utilisation de “sous-requêtes”

Une autre façon de chercher les noms des acteurs des films du studio Disney :

```
SELECT DISTINT s.nom
FROM (SELECT m
FROM Films m
WHERE m.appartient-à.nom =’’Disney’’) f,
f.acteurs s
```

La variable `m` prend valeur dans la collections des films du studio Disney, qui est le résultat de la sous-requête :

```
SELECT m
FROM Films m
WHERE m.appartient-à.nom =’’Disney’’
```

La variable `s` prend valeur dans la collections des acteurs du film de Disney `f`. **NB** : un autre `WHERE` inutile ici.