UML et les Bases de Données

1. Diagramme de classes / diagramme d'objets (UML)

1.1. Premier niveau de modélisation des données d'une application

Objectif : définir un premier niveau de modélisation des données de l'application en faisant abstraction des aspects techniques (informatiques) qui se poseront lors de la réalisation de l'application. Les éléments de modélisation d'UML sont suffisamment généraux pour permettre aux utilisateurs non spécialistes en informatique (décideurs en entreprise, par exemple) de comprendre les éléments fondamentaux (essentiels) qui vont être pris en compte lors de la réalisation de l'application.

Eléments de modélisation de ce premier niveau de modélisation :

- classe, attribut, association, multiplicité, rôle : diagramme de classes
- objet, donnée, lien : diagramme d'objets
- spécification formelle à l'aide d'expressions OCL complétant le diagramme de classes

1.2. Les éléments de modélisation

1.2.1. Objet

Objet : entité qui a des propriétés structurelles et comportementales.
Par exemple un compte bancaire a un numéro, un solde qui correspond au montant d'euros que le propriétaire du compte a confié à la banque. Sur ce compte ce dernier pourra déposer, retirer de l'argent ou effectuer des virements.

1.2.2. Classe

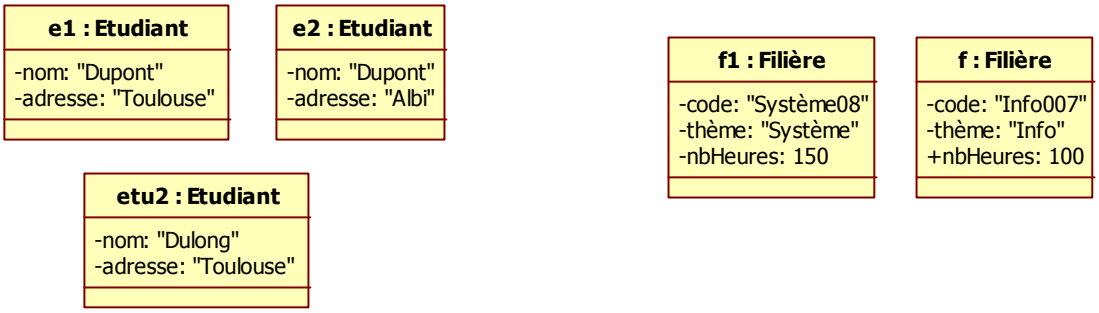
Classe : définition structurelle et comportementale d'un ensemble d'objets ayant les mêmes propriétés.

1.2.3. Diagramme de classes et d'objets

Diagramme de classes, diagramme d'objets : représentation graphique des classes et des objets.

Exemple : A l'Université, on souhaite mettre en place une application permettant d'enregistrer, pour chaque étudiant inscrit à l'Université, son nom et son adresse ainsi que pour chaque formation de l'Université son code, le thème des enseignements et le nombres d'heures d'enseignement. Modéliser les données de cette application à l'aide d'un diagramme de classes et donner des exemples d'instances d'objets.
De ce texte, on en déduit le diagramme de classes suivant :

et un exemple de diagramme d'objets, instance du diagramme ci-dessus :



Remarques :

- e2, e1, etu2, … sont appelés les noms (identifiants) des objets.
- 'Dupont', 'Albi', … sont des données de l'application.
- e2 : Etudiant = ('Dupont', 'Albi') ou e2 sont des représentations simplifiées d'un objet.

1.2.4. Atomicité des attributs, classe normalisée

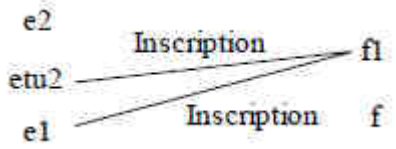
Atomicité des attributs, classe normalisée : lorsque la modélisation des données est faite en vue d'une implantation des données à l'aide d'un SGBD Relationnel, le type d'un attribut ne peut être qu'un type de base (Integer, String, Boolean, …). On dit que les attributs doivent être atomiques et par voie de conséquence les classes sont normalisées.

Autre exemple : Les responsables d'une banque souhaitent mettre en place une application pour gérer les données concernant les clients de la banque et leurs comptes bancaires : pour chaque client, on enregistre son nom et son numéro de téléphone, et pour chaque compte bancaire, son numéro et le montant disponible sur ce compte. Sur chaque compte, on pourra déposer, retirer de l'argent ou effectuer des virements.

Modéliser ces données à l'aide d'un diagramme de classes et donner des exemples d'instances.

1.2.5. Lien entre objets et associations entre classes

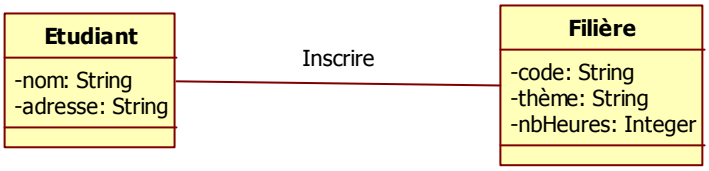
Liens entre objets (diagramme d'objets) : Dans un diagramme d'objets on peut établir des liens entre des objets. Par exemple, les étudiants e1 et etu2 sont *inscrits* en filière f2. Ce qui se représente graphiquement au niveau du diagramme d'objets de la manière suivante :



Associations entre classes (diagramme de classes) : Tout objet d'un diagramme d'objets doit être une instance d'une classe définie dans le diagramme de classes correspondant : de même tout lien entre des objets doit être une instance d'une association définie dans le diagramme de classes correspondant. Une classe a pour objectif de définir les propriétés (attributs et opérations) d'un ensemble d'objets qui pourront être créés et manipulés par les programmes de l'application : de même, une association a pour objectif de définir les propriétés d'un ensemble de liens que l'on pourra établir entre les objets.

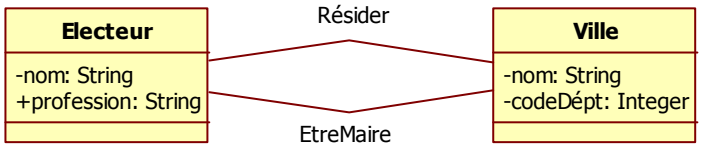
A toute association, définie par un nom, correspond un ensemble de classes, définies dans le diagramme de classes. Tout lien étant une instance d'une association a pour nom, le nom de l'association correspondant et relie un objet de chaque classe caractérisant l'association.

Le diagramme de classes suivant montre que l'association Inscription définit un ensemble de liens, chacun qui a pour nom Inscription, relie un objet de la classe Etudiant et un objet de la classe Filière :



Un diagramme de classes peut avoir plusieurs associations. Une association peut définir des liens entre 2 classes, 3 ou plus. Par exemple un *contrat* est signé par un client, un représentant pour le compte d'une compagnie d'assurance : dans ce cas chaque contrat reliera un client, un représentant et une compagnie d'assurance. On peut avoir plusieurs associations entre deux classes. Une association peut définir des liens entre des objets d'une même classe (association réflexive). Un objet peut appartenir à plusieurs liens.

Exemple : En prévision des élections, le service informatique d'un Ministère souhaite mettre en place une application gérant les données concernant les électeurs et les villes où ils résident : pour chaque électeur, on enregistre son nom et sa profession, et pour chaque ville, on enregistre son nom et le code du département où elle se trouve. Les électeurs résident dans des villes, et les villes ont des maires qui sont des électeurs.



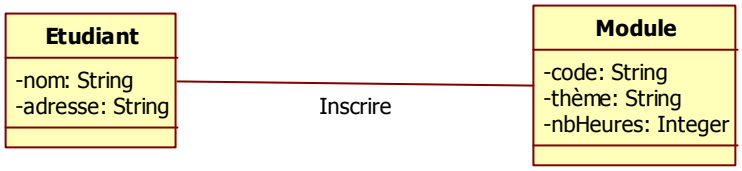
Exemple : L'application de gestion du personnel d'une entreprise a pour objectif de gérer les données de ses employés, de leurs affectations dans les départements de l'entreprise ainsi que leurs salaires :

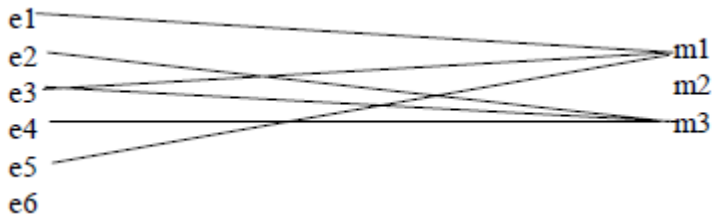
- le nom et le numéro de téléphone de chaque employé sont enregistrés dans la base de données ainsi que le nom et l'activité de chaque département ; une grille de salaire donne pour chaque salaire, un indice et le montant de ce salaire ;
- les employés dont les salaires sont référencés dans la grille des salaires, sont affectés dans les départements de l'entreprise ; tout département a un directeur qui est un employé de l'entreprise ; tout employé a un responsable qui est un employé de l'entreprise.

Question : en déduire du texte, une modélisation des données à l'aide du diagramme de classes, et donner des exemples d'instance.

1.2.6. Multiplicités (cardinalités)

Soit le diagramme de classes et le diagramme d'objets suivants :





A chaque étudiant, on fait correspondre les modules où il est inscrit :

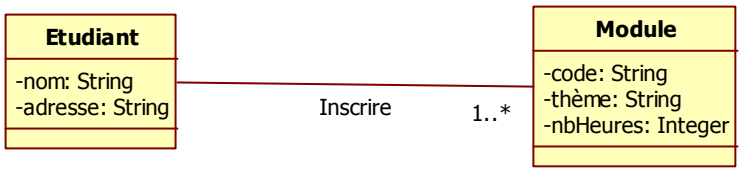
- à e1, on fait correspondre {m1} (dans le cadre de l'association Inscription)
- de même, à e2, on fait correspondre {m3}
- de même, à e3, on fait correspondre {m1, m3}
- …

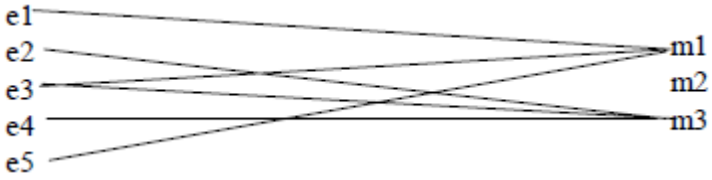
D'une manière générale, à tout objet e ∈ Etudiant, on associe dans le cadre de l'association Inscription un ensemble de modules liés à e, dont le nombre peut varier selon les mises à jour effectuées sur le diagramme d'objets.

Au niveau du diagramme de classes, on doit indiquer le nombre minimum et maximum de modules qui pourront être liés à tout étudiant : un tel couple d'entiers est appelé multiplicités (cardinalités). Ce couple d'entiers exprimant les multiplicités d'une association vis à vis d'une classe :

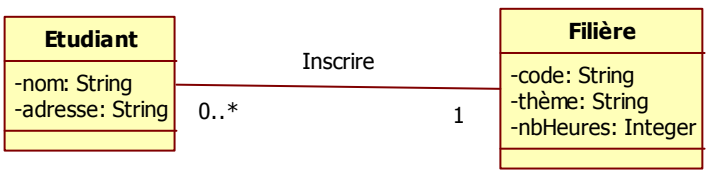
- est noté m..M, avec m = le nombre minimum (généralement 0 ou 1), et M = le nombre maximum de modules (généralement 1 ou *, pour indiquer un entier > 1) auxquels tout étudiant e peut s'inscrire ;
- est situé sur l'une des extrémités de l'association, près de la classe Module.

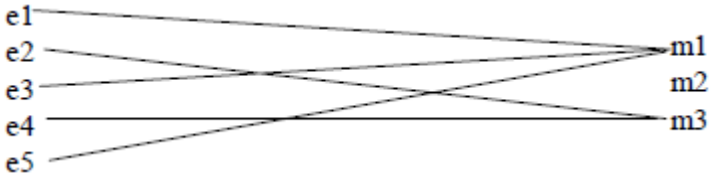
Le diagramme suivant indique que tout étudiant doit être inscrit à au moins un module, et le diagramme d'objets qui suit, est une instance pertinente du diagramme de classes, puisque l'on voit des étudiants qui sont inscrits à un module (e1, e2, e4 et e5) et que e3 est inscrit à plusieurs modules.





Une association binaire, reliant 2 classes, a donc deux extrémités : on doit indiquer ces types de multiplicités sur les 2 extrémités de l'association. Le diagramme suivant indique que tout étudiant doit être inscrit dans une (seule) filière, et que dans une filière il peut y avoir plusieurs étudiants qui y sont inscrits. Le diagramme d'objets qui suit en montre une instance pertinente :

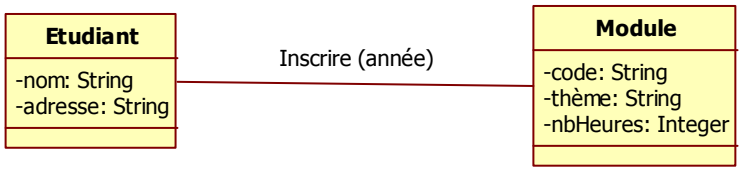


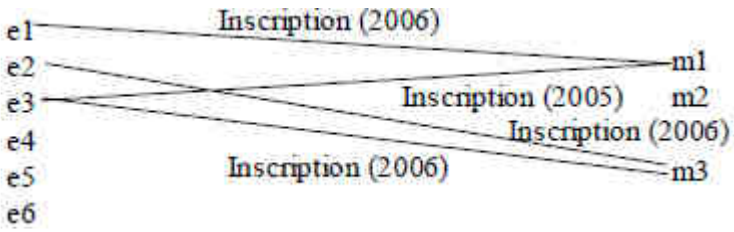


Retour à l'exemple des électeurs et des villes : Tout électeur réside dans une ville, et toute ville a un maire qui est un électeur. Un maire ne peut être maire que d'une ville.
Donner le diagramme de classes, et donner une instance pertinente de votre diagramme de classes.

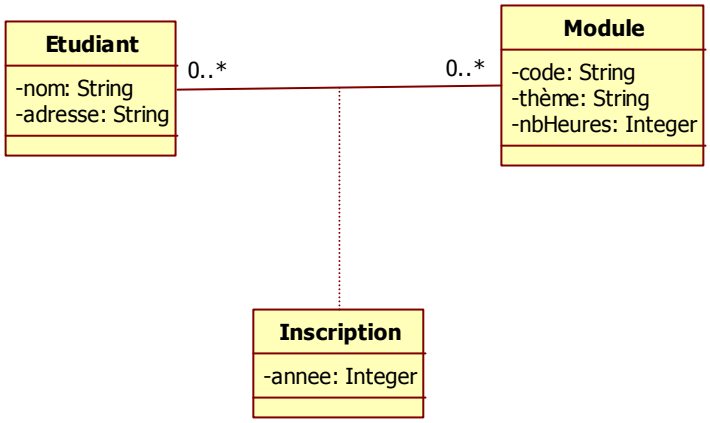
Exemple : On suppose que l'on a une base de données qui centralise les données de l'état civil concernant les français et les françaises. On suppose que l'on a enregistré pour tout français son nom, sa date de naissance et le nom de la ville et du pays où il est né. Il en est de même pour toute française. On enregistre dans la base de données les mariages, en supposant qu'un mariage permet d'unir, par les liens du mariage, un français à une française.
Donner le diagramme de classes, et donner un exemple pertinent d'instance de votre diagramme de classes.

Association porteuse d'informations (classe d'association) :
Une association peut être porteuse d'information : une telle association est appelée classe d'association. Ce qui signifie qu'à tout lien, instance d'une telle classe d'association devra correspondre des données conformes à ce qui est déclaré dans la classe d'association.
Par exemple, le diagramme suivant montre que l'on souhaite enregistrer l'année où tout étudiant s'inscrit à un module :

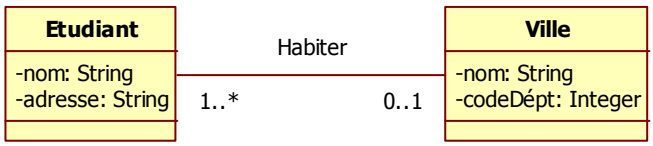


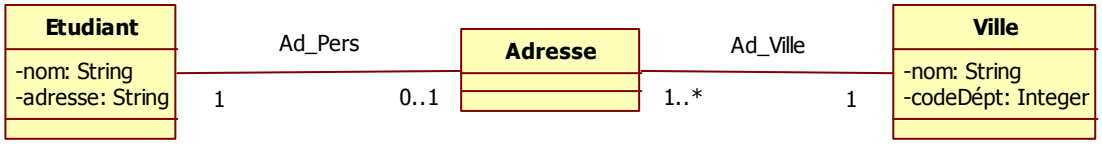


De cette instance, on en déduit, par exemple, que e3 est inscrit au module 1 en 2005 et au module 3 en 2006. La représentation en UML d'une classe d'association est la suivante :



Remarque : association ou classe ?
Une association (ou classe d'association) peut être vue comme une classe. Les 2 diagrammes suivants définissent des structures de données équivalentes :

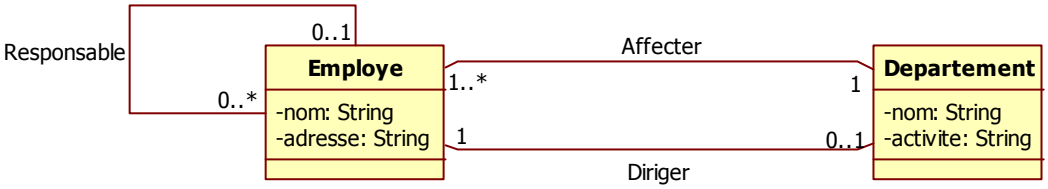




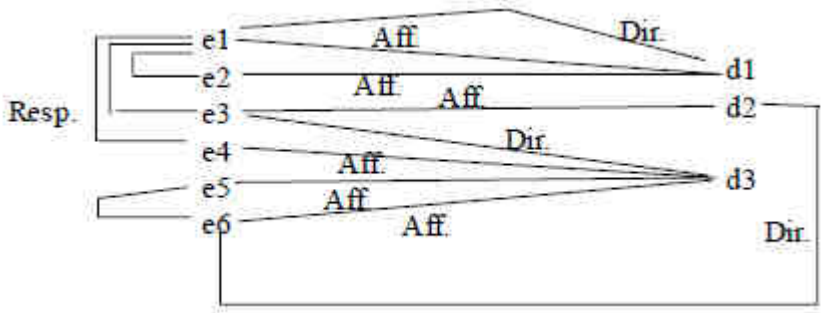
Question : donner, pour chaque diagramme, l'instance modélisant le même ensemble de données. Justifier les passages des multiplicités du premier diagramme, par rapport au deuxième diagramme.

1.2.7. Les rôles

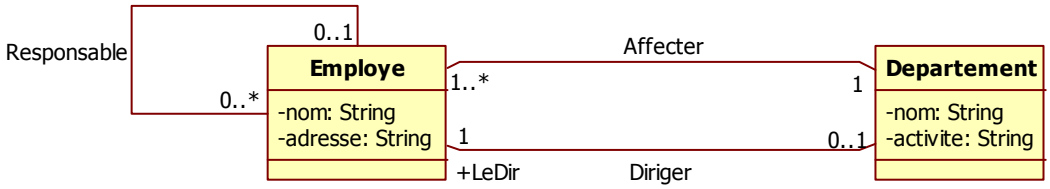
Objectifs des rôles : les rôles permettent d'apporter plus de lisibilité, plus de compréhension aussi bien au niveau du diagramme de classes qu'au niveau du diagramme d'objets. Par exemple, le diagramme de classes suivant ne permet pas de savoir comment s'interprète l'association Responsable, par rapport aux multiplicités (1..1 et 0..*) :



Cette imprécision se répercute au niveau du diagramme d'objets, où, dans certains cas, on ne peut pas savoir qui est responsable de qui :

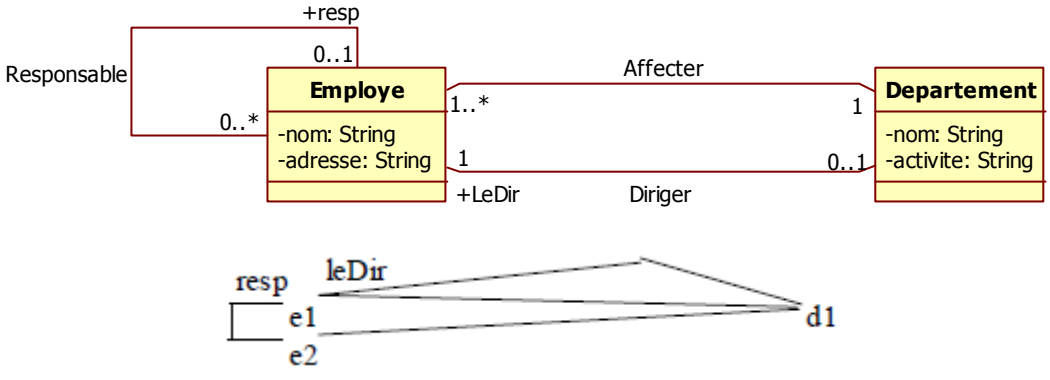


Un rôle est un nom que l'on met au niveau de l'extrémité d'une association (au même niveau que les multiplicités) tel que le montre le diagramme de classe suivant :



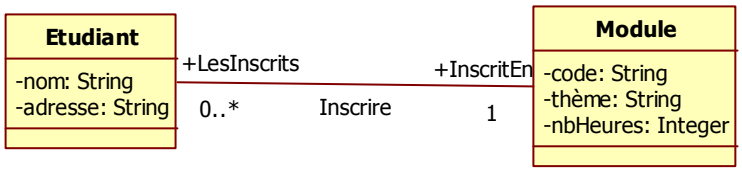
Ce diagramme montre que de la classe Departement, la classe Employe est vue sous le nom leDir. Le nom du rôle peut être reporté au niveau du diagramme d'objets, où on peut mieux voir, dans l'instance suivante que le directeur du département d3 est e3, celui du département d2 e6 et celui du département d1 qui est l'employé e1 :

Un nom de rôles peut être déclaré à chaque extrémité d'une association. D'une manière générale un nom de rôle facilite la lecture et l'interprétation d'un diagramme de classes, mais, pour ne pas surcharger les diagrammes de classes et leurs instances on ne met que le nom des associations et des rôles qui permettent de lever les ambiguïtés, tel que le montre l'exemple suivant :

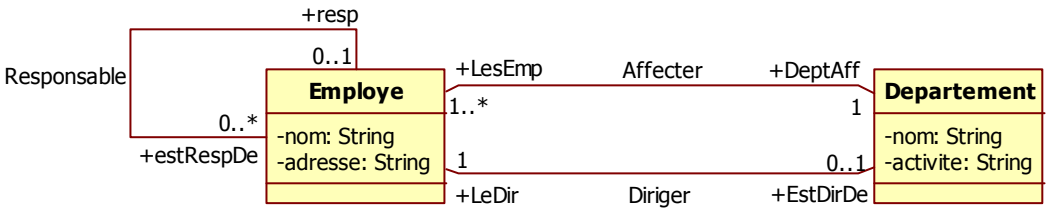


Cette notation permet de voir que, d'après cette instance précédente, le responsable de l'employé e2 est e1.

Retour à l'exemple des étudiants inscrits dans des modules : quelle interprétation donnez-vous à ce diagramme en prenant en compte les noms des rôles :



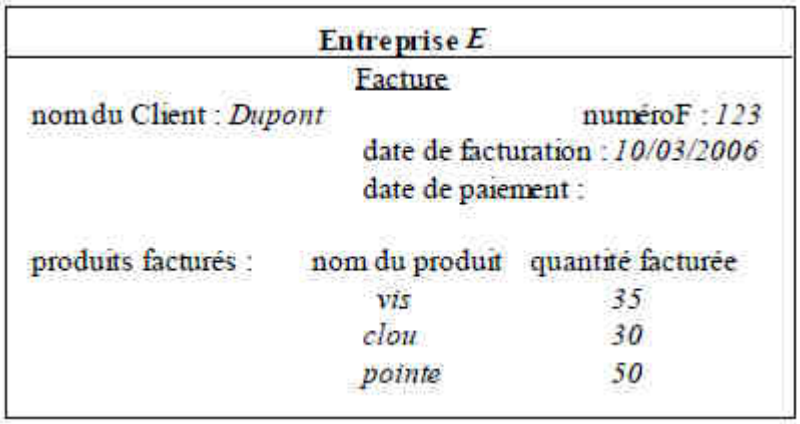
Même question pour ce diagramme de classes :



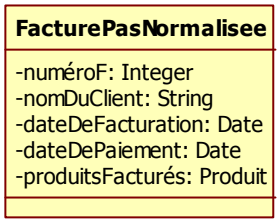
1.2.8. Composition, lien de composition

Exemple de normalisation d'une classe : Les responsables d'une entreprise souhaitent faire gérer, par un système de gestion de bases de données relationnelles, les données contenues dans les factures qui sont envoyées aux clients. Sur chaque facture, on trouve le numéro de la facture, le nom du client à qui est envoyée la facture, la date de facturation, la date de paiement de la facture, et pour chaque type de produit acheté par le client, le nom du produit et la quantité achetée.
Modéliser les données de cette application à l'aide du diagramme de classes et donner une instance du diagramme.

Eléments de solution de cet exemple : on pourrait imaginer qu'une facture (simplifiée) se représente de la manière suivante :



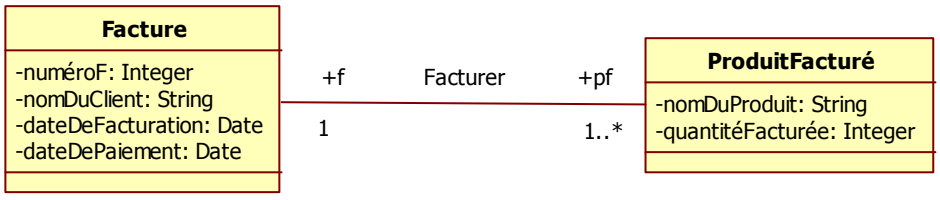
Ce qui peut se modéliser de la manière suivante :



Les 4 premiers attributs de la facture sont atomiques, puisque à toute facture correspond un numéro, un nom de client, et une date de facturation et de paiement. Par contre, l'attribut 'produitsFacturés' n'est pas atomique car cet attribut se décompose en deux attributs (pour chaque type de produit facturé, on enregistre son nom et la quantité facturée). D'autre part, le nombre de produits facturés peut varier d'une facture à une autre (dans l'exemple ci-dessus, le nombre de type de produits facturés est 3).
On rappelle que tout diagramme de classes modélisant des données en vue d'une gestion relationnelle des données, doit être normalisé : ce qui implique que tout attribut de la classe doit être atomique. La modélisation précédente de Facture ne peut donc pas être considérée comme une classe normalisée.
Une solution consiste à éclater Facture en deux classes normalisées reliées par une association permettant de rattacher une facture à tous ses produits facturés :

- la première reprend les attributs atomiques de la facture, ce que l'on appelle généralement « l'entête » de la facture qui contient les données des attributs atomiques de la facture ;
- la deuxième modélise les données de chaque type de produit facturé, cette classe est donc définie sur les attributs nom du produit et quantité commandée de ce produit : cette deuxième classe modélise donc un ensemble de produits facturés dont chacun (chaque instance) doit être lié à une facture : la facture où se produit est facturé ;
- l'association définissant les liens entre les factures et leurs produits facturés reprendra, comme nom d'association, l'attribut non atomique de la classe Facture : produitsFacturés ;
- compte tenu du fait que toute facture fait référence à, au moins, un produit facturé et que tout produit facturé doit être lié à une facture, on en déduit les multiplicités respectives : 1..* et 1..1.

Le diagramme de classes modélisant les données, en vue d'une implantation relationnelle, pourrait donc être le suivant :



Exemple d'instance (diagramme d'objets) de ce diagramme de classes :

Ce diagramme d'objets, instance du diagramme de classes ci-dessus, montre que 2 factures ont été enregistrées : sur la première on trouve 3 produits facturés et sur la deuxième, 1 seul produit a été facturé : le client de nom Dupont se voit facturé des clous en quantité 40.

Remarques : cette étape de décomposition d'une classe non normalisée en 2 classes normalisées, reliées par une association n'est pas très naturelle : mais elle est nécessaire avant de passer à une étape de modélisation relationnelle. En fait, cette étape de décomposition prépare la définition des fichiers de l'application, chacun étant un ensemble d'articles de même structure (logique et physique) : intuitivement, on pourrait admettre qu'à chaque classe correspondrait un fichier.
Lorsque d'un diagramme de classes est conçu en vue d'une implantation Java, par exemple, cette étape préliminaire de décomposition d'une classe n'est pas nécessaire, puisque un attribut Java peut être implanté à l'aide d'un tableau, ce qui n'est pas le cas en relationnel.

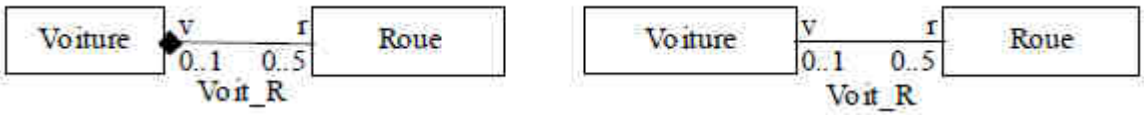
Association/Composition : Cette décomposition d'une classe non normalisée en deux classes normalisées, réunies par une association, fait que des données de deux applications différentes peuvent se modéliser structurellement de la même manière, alors qu'en fait les objets se manipuleront selon des approches différentes. Par exemple, soient les 2 diagrammes, chacun correspondant aux besoins d'une application :

Si ces deux modélisations des données ci-dessus sont identiques structurellement parlant, en fait au niveau applicatif, la destruction d'une facture entraîne la destruction des produits facturés dans cette facture ; par contre, la destruction d'un département ne peut pas se faire si des employés sont affectés à ce département.

Pour marquer cette différence montrant un rattachement fort (physique) d'objets d'une classe entrant dans la composition des objets d'une autre classe, l'association doit transformée en ce que l'on appelle une Composition et se note habituellement à l'aide d'un losange noire tel que le montre la figure suivant :

Dans ce cours, on admettra que toute composition vient de la décomposition d'un attribut d'une classe non atomique. En reprenant l'exemple précédent, Facture est appelée classe englobante de la classe prod_Fact ; inversement, la classe Prod_Fact est appelée classe composite de la classe Facture.

La nuance entre une association et une composition est difficile à cerner au niveau de la modélisation des données et lors de la modélisation. En fait, elle apparaît plus claire lors de la manipulation des données. C'est pourquoi modéliser des données peut s'avérer être une étape très délicate. Souvent, le choix entre une association ou une composition dépend des applications. Les deux modèles suivants montrent qu'à toute voiture peut être liée au maximum 5 roues :



Le diagramme de gauche pourrait modéliser les données d'une application d'un ferrailleur par exemple, qui quand il élimine une voiture (la compresse pour la réduire en un cube prenant le moins de place possible) il élimine aussi tous les éléments de la voiture qui la composent. Par contre, le diagramme de droite montre qu'avant d'éliminer une voiture, il faut préalablement éliminer les liens avec ses roues : ce modèle de données irait bien, par exemple, à une application destinée à un garagiste qui voudrait récupérer toutes les pièces d'une voiture avant de l'éliminer.
Le langage SQL permet de prendre en compte la différence entre une association et une composition.

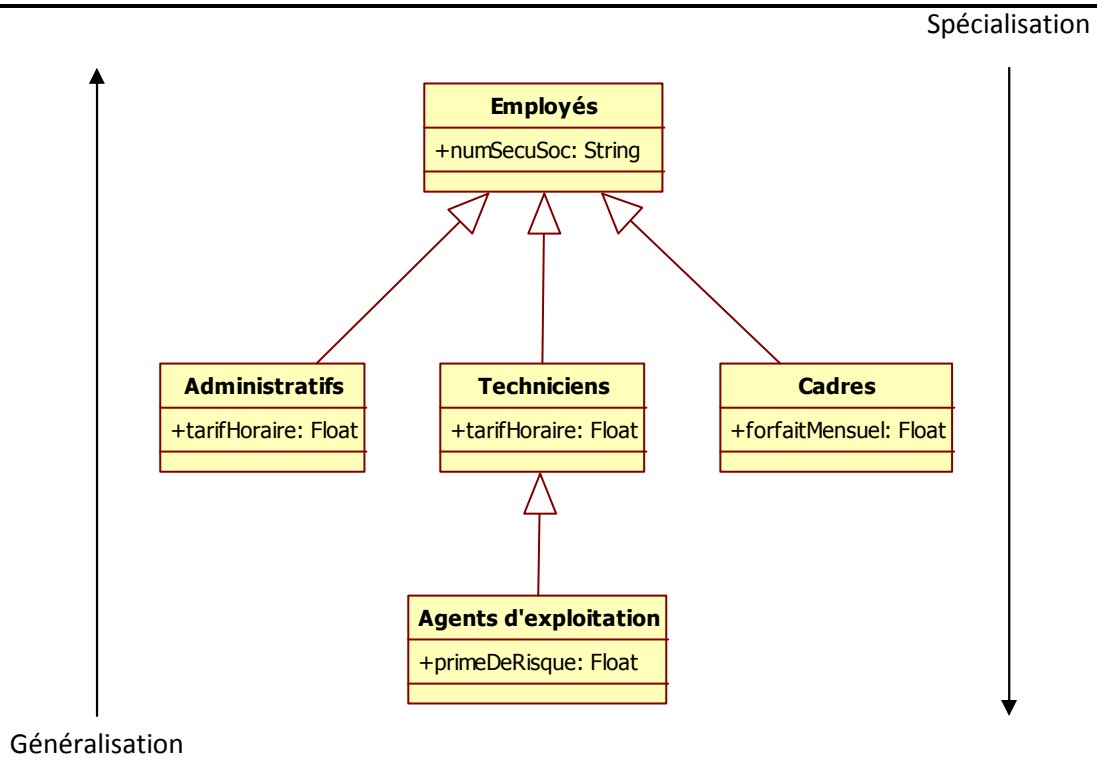
Cette décomposition de classes montre que quand on modélise, il ne faut pas chercher à représenter la structure des données d'une application telle que l'on pourrait se les imaginer. Il faut les modéliser de manière à ce que l'on puisse les retrouver intégralement et sans ambiguïté. De toute façon, même au niveau implantation, on aura du mal à conserver exactement une structure de données physique reprenant exactement la modélisation qui en a été faite lors de la conception. Par contre, au niveau manipulation des données, il faudra absolument avoir les moyens de retrouver les données telles que souhaitent les voir les utilisateurs finals.

Ce problème de décomposition de classes en classes normalisées est très courant, et se rencontre dans un très grand nombre d'applications, en particulier toutes les applications de gestion où les données sont naturellement structurées hiérarchiquement à plusieurs niveaux :

- gestion d'appartements dans des immeubles qui ont des propriétaires,
- dossier médical,
- bordereau de notes d'étudiants,
- emploi du temps,
- …

1.2.9. Héritage

L'héritage est un mécanisme de transmission des propriétés d'une classe (ses attributs et méthodes) vers une sous-classe. Une classe peut être spécialisée en d'autres classes, afin d'y ajouter des caractéristiques spécifiques ou d'en adapter certaines. Plusieurs classes peuvent être généralisées en une classe qui les factorise, afin de regrouper les caractéristiques communes d'un ensemble de classes. La spécialisation et la généralisation permettent de construire des hiérarchies de classes. L'héritage peut être simple ou multiple. L'héritage évite la duplication et encourage la réutilisation.



Le concept d'héritage, essentiel dans les approches objet, ne sera plus abordé dans ce cours parce que le diagramme de classes est ici utilisé comme une étape préliminaire à une modélisation relationnelle des données, modèle n'intégrant le concept d'héritage. Cependant, l'héritage étant une valeur importante de l'objet, il est important de savoir qu'il existe et à quoi il correspond.

1.3. Exemples de modélisation des données d'une application

1.3.1. Exemple 1 (facturation de produits à des clients)

Une entreprise souhaite développer une application gérant les informations qui sont contenues dans les factures envoyées aux clients. Les responsables de l'application demandent que soient enregistrées dans la base, les données suivantes :

- pour tout client de l'entreprise, son nom et son adresse,
- pour tout produit mis au catalogue de l'entreprise, son nom et son prix unitaire,
- pour toute facture envoyée à un client, un numéro, la date de facturation, la date de paiement qui est mise à jour quand elle est acquittée par le client, et pour chaque produit facturé sa quantité facturée.

1.3.2. Exemple 2 (gestion des prêts de livres dans un club)

Un club de livre souhaite informatiser la gestion des prêts des livres aux abonnés du club :

- pour chaque livre, on enregistre le titre, l'auteur, et pour chaque exemplaire de livre dont dispose le club, un numéro et le nom de l'éditeur ;
- pour chaque abonné, on enregistre son nom et le nom de la ville où il habite ;
- chaque exemplaire de livres est la propriété d'un abonné du club ;
- à chaque prêt d'un exemplaire de livre à un abonné, on enregistre la date de prêt de l'exemplaire ;
- on souhaite archiver les prêts de livres que les abonnés ont faits, de manière à savoir, en fin d'année, qui a lu quoi pour en tenir compte lors de l'achat de nouveaux livres.

Question : effectuer une analyse du domaine, et en déduire une modélisation les données de l'application à l'aide d'un diagramme de classes. Tous les éléments du texte ont-ils pu être exprimés dans le diagramme de classes ?

2. Transformation d'un diagramme de classes en un schéma relationnel

2.1. Conception, génération de code

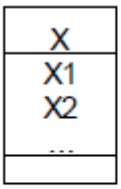
Généralement, on ne génère pas le code du LDD SQL directement à partir d'un diagramme de classes. On préfère transformer le diagramme de classes en un schéma représentant les données au travers de leurs éléments de modélisation relationnelle. Le code du LDD SQL est alors déduit du schéma relationnel. Un des intérêts est de pouvoir représenter la structure relationnelle des données sous une forme graphique donnant une vision générale des données, dégagée de toute contrainte syntaxique du langage du LDD SQL.

2.2. Transformation d'une classe dans le modèle relationnel

2.2.1. Principe général de transformation

Intuitivement, tel que le suggère la figure suivante, à chaque classe on peut faire correspondre une relation telle que :

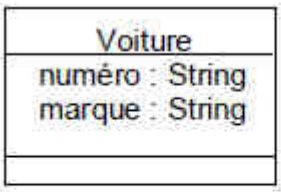
- le nom de la relation reprend le nom de la classe,
- le nom des attributs de la relation sont issus des noms des attributs de la classe.



X(X1, X2, ...)
x11, x12, ...
x21, x22, ...
...

x1 : X(x11, x12, ...)
x2 : X(x21, x22, ...)
...

Exemple :



Voiture(numéro, marque)
'1 AA 31, 'renault'
'3 BB 75', 'renault'

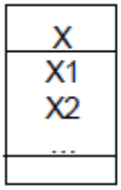
v1 : Voiture('1 AA 31, 'renault')
v2 : Voiture('3 BB 75', 'renault')

Cette correspondance n'est directement possible que si les attributs de la classe sont atomiques (les types des attributs sont des types de base : Integer, String, … , qui devront être adaptés, selon le cas, en type SQL : varchar, number ou date).

2.2.2. Identifiant, clé primaire d'une relation

En fait, il est important de pouvoir avoir les moyens d'identifier de manière unique tout tuple de la relation. Une solution simple et systématique consiste à rajouter arbitrairement à la relation un

attribut qui jouera le rôle de clé primaire de la relation : à cet attribut clé primaire de la relation correspond la notion d'identifiant d'objets. Ce nouvel attribut, clé primaire de la relation, pourra être appelé identifiant de la relation. Lorsqu'un objet est créé, normalement on donne un nom (identifiant) à cet objet, et on affecte une valeur à chacun des attributs de la classe dont l'objet est une instance. Au niveau relation, il faut aussi donner, lors de la création d'un tuple, sa valeur de clé primaire (l'identifiant du tuple), et pour chaque attribut de la relation une valeur. Les identifiants d'objets ne sont pas modifiables ; en principe, il devrait en être de même pour les valeurs de clé primaire.
On a donc la correspondance diagramme de classes/schéma relationnel suivante :



X(**X_ID**, X1, X2, ...)
x1, ...
x2, ...
...

x1 = X(, ...)
x2 = X(...)
...

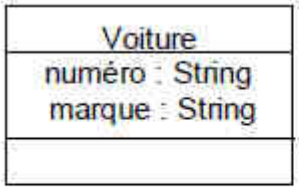
- remarque :
  - X est une relation, les attributs de cette relation sont les attributs de la classe,
  - l'attribut **X_ID** joue le rôle de clé primaire de la relation X.
- pré-conditions :
  - les noms des attributs de la classe X sont différents 2 à 2,
  - le nom des attributs est différent de X_ID,
  - le type des attributs sont des types de base (Integer, Boolean, String)

2.2.3. Génération du code du LDD SQL

A partir du schéma relationnel, on peut en déduire le code du LDD SQL qui devra être transmis au SGBD Relationnel pour qu'il puisse créer le schéma interne des données :

```
SQL>  create table X(  X_ID varchar( 10 ) primary key,
                       X1 …,
                       …                              );
```

Exemple :



Voiture(**Voiture_ID**, numéro, marque)
'v1', '1 AA 31, 'renault'
'v2' '3 BB 75', 'renault'

v1 : Voiture('1 AA 31, 'renault')
v2 : Voiture('3 BB 75', 'renault')

```
SQL>  create table Voiture(   Voiture_ID varchar( 10 ) primary key,
                              numero number,
                              marque varchar( 10 )                );
```

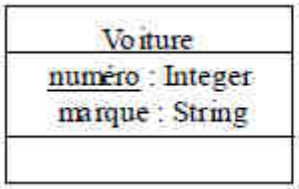
2.2.4. Attribut clé primaire de la relation

Il se peut que sur certains attributs de la classe, un invariant de type clé soit défini : dans ce cas, l'un de ces attributs peut se substituer à la clé primaire créée arbitrairement lors du passage au niveau relationnel, les autres seront des attributs qui, au niveau relationnel, auront la propriété : unique not null.

Par exemple, on pourrait imaginer qu'il ne peut pas exister deux voitures qui ont le même numéro. L'attribut numéro doit donc vérifier l'invariant suivant, écrit sous la forme d'une expression OCL fermée :

```
Voiture.allInstances→forAll( v1, v2 |   if v1 <> v2      then v1.numero <> v2.numero
                                                          else true
                                        endif )
```

Dans ce cas, le schéma relationnel et le code du LDD SQL peuvent être simplifiés, et le passage au niveau relationnel peut être représenté par la figure suivante :



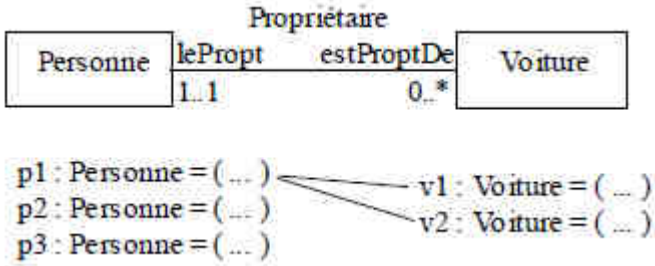
Voiture(**numéro**, marque)

```
SQL>  create table Voiture(   numero number primary key,
                              marque varchar( 10 )          );
```

2.3. Principe de transformation des associations et des liens en relationnel

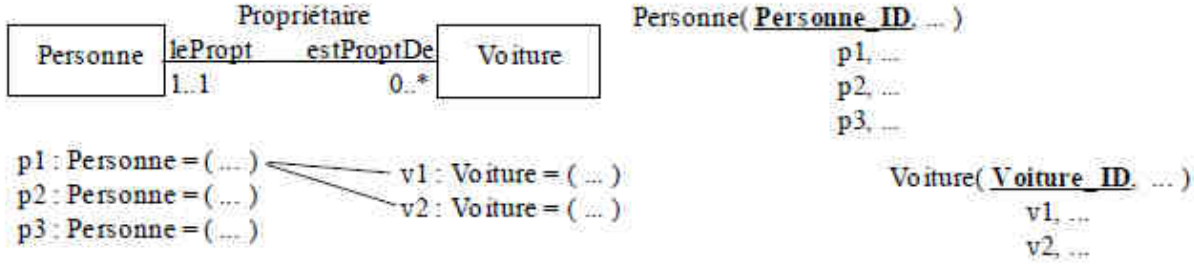
2.3.1. Association par valeur de clé (primaire)

En relationnel, c'est au travers des valeurs de clé (primaire) que l'on peut lier (associer) des tuples entre eux. Supposons le diagramme classique suivant :



La transformation en relationnel d'un tel diagramme se fait, intuitivement, selon les étapes suivantes :

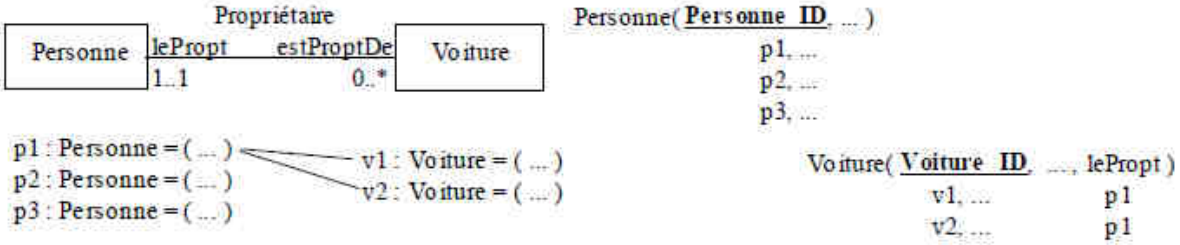
1. à chaque classe correspond une relation :



Personne(**Personne_ID**, ...)
p1, ...
p2, ...
p3, ...

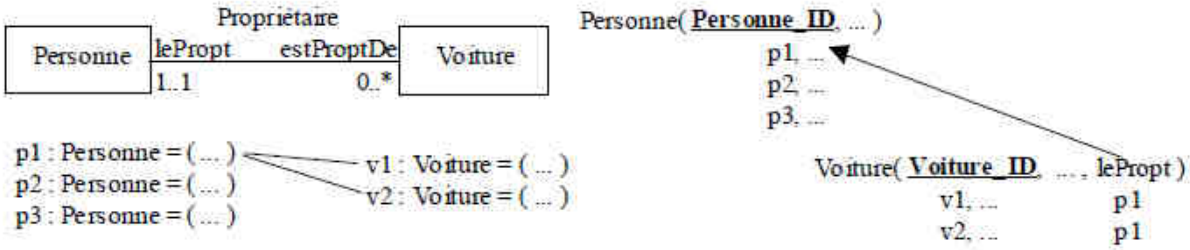
Voiture(**Voiture_ID**, ...)
v1, ...
v2, ...

2. puisque chaque voiture a un propriétaire qui est une personne, il suffit dans la relation Voiture de rajouter une colonne permettant d'indiquer, pour chaque voiture, quel est son propriétaire ; ce propriétaire pourra être représenté par son identifiant, c'est-à-dire sa valeur de clé (primaire) ; il suffit donc de donner comme nom à cette colonne le nom du rôle :



3. puisqu'un propriétaire ne peut être qu'une personne qui existe, c'est-à-dire une personne dont le tuple est dans la relation Personne, toute valeur de la colonne Voiture(lePropt) doit être une valeur de clé primaire de la relation Personne, c'est-à-dire que toute valeur de la colonne(lePropt) doit se retrouver dans la colonne Personne(**Personne_ID**). On dit que l'attribut Voiture(lePropt) a pour domaine référentiel l'attribut Personne(**Personne_ID**). On écrit : Voiture(lePropt) $\subseteq$ Personne(**Personne_ID**).

La figure suivante montre comment on représente cette dépendance dans le schéma relationnel, et au niveau du code du LDD SQL :



Le code du LDD SQL, correspondant au schéma relationnel, est le suivant :

```
SQL>  create table Personne(  Personne_ID varchar( 10 ) primary key,
                              …
                                                        );
SQL>  create table Voiture(   Voiture_ID varchar( 10 ) primary key,
                              …
                              lePropt varchar( 10 ) references Personne(Personne_ID));
```

2.3.2. Dépendance référentielle, clé étrangère

Au niveau syntaxe du code LDD SQL :

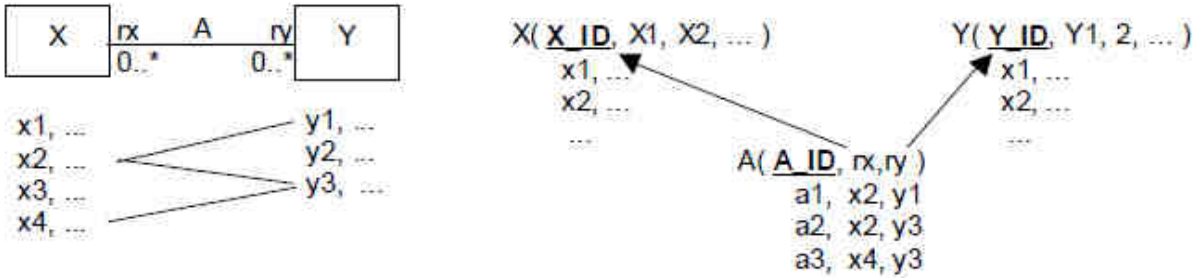
- il existe différentes manières de déclarer la contrainte liée à la dépendance référentielle, mais à ce niveau du cours du CNAM, on ne rentrera pas dans ces détails,
- pour des raisons logiques (au niveau des associations), le domaine de référence (ici Personne(Personne_ID)) doit être une clé ; et pour des raisons d'optimisation, le domaine de référence doit être une clé primaire : c'est pourquoi, l'attribut lePropt est appelé clé étrangère. (l'attribut lePropt n'est pas un attribut clé, mais il a pour domaine de référence un attribut clé étrangère).

2.4. Transformation d'une association en relationnel (cas général)

L'exemple de transformation présenté précédemment permettait de montrer intuitivement comment les valeurs de clé (primaires) peuvent être utilisées pour exprimer des liens entre tuples de relations, ce qui justifie les concepts de dépendance référentielle et de clé étrangère pour exprimer au niveau relationnel le concept d'associations et de liens définis au niveau de diagramme de classes. Cet exemple intuitif était basé sur une association dont les multiplicités étaient 1..1 / 0..*.

2.4.1. Relation de base, relation d'association

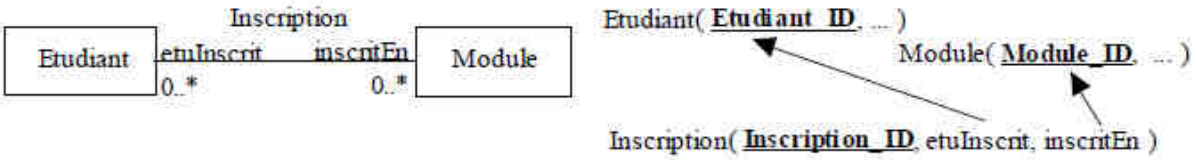
D'une manière générale, une association dont les multiplicités sont (0..* / 0..*) se traduit par une relation (souvent appelée relation d'association), tel que le suggère à la figure suivante, le diagramme de classes et le schéma relationnel qui s'en déduit :



- remarque :
  - A(rx) ⊆ X(**X_ID**), A(ry) ⊆ Y(**Y_ID**)
  - X et Y sont des relations de base ; A est une relation d'association exprimant des liens entre les tuples des relations X et Y
- pré-conditions :
  - les nom de rôles rx et ry sont différents, et sont différents du nom de l'association.
- génération du code du LDD SQL :
  - SQL> create table X(X_ID varchar(10) primary key,
    X1 …,
    …);
  - SQL> create table Y(Y_ID varchar(10) primary key,
    Y1 … ,
    …);
  - SQL> create table A(A_ID varchar(10) primary key,
    rx varchar(10) references X(X_ID),
    ry varchar(10) references Y(Y_ID)) ;

Remarque : les références en avant doivent être satisfaites au niveau du code du LDD SQL. C'est pourquoi, les tables X et Y doivent être créées avant la table A. Il est en de même pour les tuples des relations X et Y qui doivent être créés avant de retrouver leur valeur de clé primaire dans la relation A.
En ce qui concerne la destruction des tuples et des relations, il ne sera possible de détruire la table A que si elle est vide.

2.4.2. Exemple



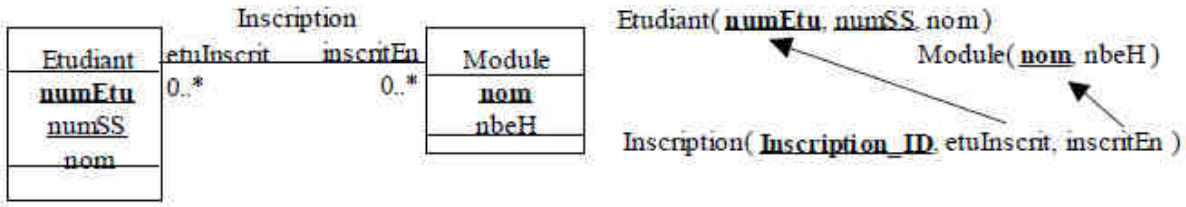
SQL> create table etudiant(Etudiant_ID varchar(10) primary key,
… ,
…);

SQL> create table Module(Module_ID varchar(10) primary key,
… ,
…);

SQL> create table Inscription(Inscription_ID varchar(10) primary key,
etuInscrit varchar(10) references Etudiant(Etudiant_ID),
inscritEn varchar(10) references Module(Module_ID)) ;

Comme pour l'exemple précédent, les attributs des classes, peuvent, au niveau relationnel, jouer le rôle de clé primaire, et dans ce cas, les dépendances référentielles doivent être reportées sur ces attributs.

Exemple :



SQL> create table etudiant(numEtu varchar(10) primary key,
numSS number unique not null,
nom varchar(10));

SQL> create table Module(nom varchar(10) primary key,
nbeH number);

SQL> create table Inscription(Inscription_ID varchar(10) primary key,
etuInscrit varchar(10) references Etudiant(numEtu),
inscritEn varchar(10) references Module(nom)) ;

2.4.3. Algorithme de transformation d'une association en relationnel

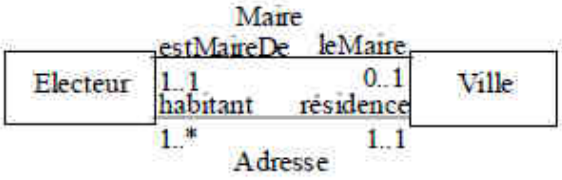
1. Chaque classe du diagramme de classes devient une table en relationnel.
2. A chaque classe du diagramme de classe est associé un identifiant : celui-ci est soit un identifiant à générer « classe_id » soit une propriété de la classe. Cet identifiant devient la clé primaire de la table créée correspondante.
3. Toute relation dont une des cardinalités est de type (…,1) provoque la génération d'une clé étrangère. Cela a pour conséquence la création d'une colonne de nom « le_rôle » dans la table, chaque valeur de cette colonne doit faire référence à la clé primaire de l'autre table issue de l'association.
4. Pour toutes les autres cardinalités, la relation devient une nouvelle table ayant pour clé primaire le couple d'identifiant des tables issues de l'association.

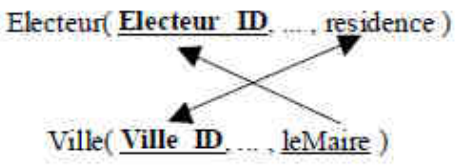
Attention : en fonction des cardinalités, la création des attributs ne sera pas la même. Par exemple, une cardinalité (1,1) donnera un attribut dont la valeur ne devra jamais être nulle et devra référencer une clé primaire d'une autre table alors qu'une cardinalité (0,1) donnera un attribut dont la valeur pourra être nulle ou devra référencer une clé primaire d'une autre table.

2.4.4. Circuits dans les dépendances référentielles

La génération du code du LDD SQL se fait à partir du schéma relationnel, ce dernier étant obtenu en appliquant pour chaque classe et pour chaque association les règles définies ci-dessus. Pour effectuer cette génération de code, on ne tient pas compte du double-sens des dépendances référentielles.
L'ordre de création des relations se fait en respectant les références en avant des dépendances référentielles. Il peut exister un (ou plusieurs circuits) dans les dépendances référentielles. Dans ce cas, il faut que le concepteur 'coupe' ce circuit, et reporte la contrainte correspondante au niveau applicatif.

Exemple :





Dans ce cas, pour pouvoir écrire le code du LDD SQL, il faut supprimer une dépendance, par exemple :