

Éléments de cours de BD du CNAM (2005/2006)

1	Diagramme de classes / diagramme d'objets (UML)	4
1.1	Premier niveau de modélisation des données d'une application	4
1.2	Les éléments de modélisation	4
1.2.1	Objet	4
1.2.2	Classe	4
1.2.3	Diagramme de classes et d'objets	4
1.2.4	Atomicité des attributs, classe normalisée	5
1.2.5	Lien entre objets et associations entre classes	5
1.2.6	Multiplicités (cardinalités)	6
1.2.7	Les rôles	8
1.2.8	Composition, lien de composition	10
1.3	Exemples de modélisation des données d'une application	12
1.3.1	Exemple 1 (facturation de produits à des clients)	12
1.3.2	Exemple 2 (Gestion de bons de commandes, des livraisons et des factures)	13
1.3.3	Exemple 3 (gestion des prêts de livres dans un club)	19
2	Introduction au langage OCL (Object Constraint Language)	19
2.1	Aspect structurel des éléments de modélisation du diagramme de classes	19
2.2	Concepts de base du langage OCL	20
2.2.1	OCL est un langage sans effet de bord	21
2.2.2	OCL est un langage objet	21
2.2.3	OCL est un langage ensembliste	22
2.2.4	OCL est un langage basé sur la logique des prédicats du 1 ^{er} ordre	23
2.3	L'approche fonctionnelle du langage OCL	24
2.3.1	OCL est un langage fonctionnel	24
2.3.2	Enchaînement des opérations sur les collections	24
2.3.3	Navigation des expressions OCL	25
2.3.4	Remarque sur la visibilité d'un objet	25
2.3.5	Quelques expressions OCL	26
2.4	Modélisation : diagramme de classes + contraintes d'intégrité OCL	26
3	Transformation d'un diagramme de classes en un schéma relationnel	28
3.1	Conception, Génération de code	28
3.2	Transformation d'une classe dans le modèle relationnel	28
3.2.1	Principe général de transformation	28
3.2.2	Identifiant, clé primaire d'une relation	28
3.2.3	Génération du code du LDD SQL	29
3.2.4	Attribut clé primaire de la relation	29
3.3	Principe de transformation des associations et des liens en relationnel	30
3.3.1	Association par valeur de clé (primaire)	30
3.3.2	Dépendance référentielle, clé étrangère	31
3.4	Transformation d'une association en relationnel (cas général)	31
3.4.1	Relation de base, relation d'association	32
3.4.2	Exemple	32

3.5	Transformation d'une association en relationnel (en tenant compte des contraintes de multiplicités)	33
3.5.1	Multiplicités 0..* / m..M	33
3.5.2	Multiplicités 0..1 / m..M	36
3.5.3	Multiplicités 1..* / m..M	37
3.5.4	Multiplicités 1..1 / m..M	38
3.5.5	Circuits dans les dépendances référentielles	39
4	Résumé, Bilan	40

Introduction sur le cours

Intervenants dans les enseignements : Thierry Millan (millan@irit.fr), Pierre Bazex (bazex@irit.fr)

Application bases de données : application où l'aspect donnée est primordial :

- volume de données important
- très grande variété des données
- persistance des données
- échange de données entre applications
- aspect multi-utilisateur
- application à de très nombreux domaines d'activité
- très grande variété d'utilisateurs autour d'une application BD
- ...

Les réseaux informatiques (WEB), les interfaces graphiques font des bases de données un sujet d'actualité très important, ... : manipulation de très gros volumes de données, accès aux bases de données 24 h / 24, très grand nombre d'utilisateurs demandant des accès en même temps, Les applications bases de données ne sont plus limitées à des applications de gestion : on trouve des bases de données dans pratiquement tous les secteurs d'activités (transport, aéronautique, espace, ... , médecine, pharmacie, ...).

Processus de développement (différentes étapes de développement) d'une application base de données :

- à partir de l'expression des besoins (rédigée sous forme informelle) :
 - → premier niveau de modélisation des données : conception/spécification.
 - → réalisation de l'application.
- (faire le parallèle avec la programmation : algorithme → programmation)
- (faire le parallèle avec le plan d'une maison déduit des besoins du client → construction)

Conception/Spécification :

- elle est généralement faite avec la méthode Merise préconisant un premier niveau de modélisation des données à l'aide du modèle Entité/Association.
- dans ce cours on remplace le modèle Entité/Association par le diagramme de classes et le diagramme d'objets d'UML (Unified Modeling Language) qui, en plus de la modélisation des données, permet de modéliser d'autres aspects des applications. UML est une norme.

Réalisation à l'aide d'un Système de Gestion de Base de Données (SGBD) :

- les SGBD sont classés en fonction des modèles de données qu'ils proposent :
 - . actuellement les SGBD Relationnels (Oracle, DB2, ...) sont les plus courants.

Plan du cours :

- **I** (P. Bazex) :
 - modélisation, spécification des données (diagramme de classes et d'objets (UML))
 - introduction au langage OCL permettant de compléter la modélisation des données
 - passage au niveau relationnel :
 - . transformation d'un diagramme de classes en un schéma relationnel
- **II** (T. Millan) :
 - le modèle relationnels et les SGBD Relationnels

Remarques :

- un éditeur UML, et l'évaluateur d'expressions OCL USE peuvent être téléchargés sur PC,
- les étudiants qui auraient un compte CICT peuvent accéder à ces logiciels : les éditeurs UML et le logiciel USE sur la machine Marine, et le logiciel Oracle (SGBD Relationnel) sur la machine Telline).

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 de 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.

Éléments de modélisation de ce premier niveau de modélisation :

(voir livre PA. Muller : *modéliser avec UML (?) 2^{ème} édition* ; des cours sur UML sont accessibles sur le WEB : se limiter au diagramme de classes et d'objets et OCL)

- 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 : - une entité atomique (?) 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

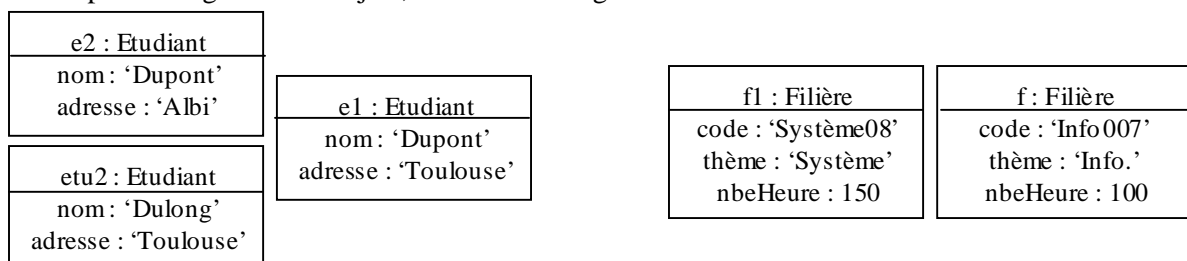
Diagramme de classes, diagramme d'objets : représentation graphique des classes et des objets.

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 nombre 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.

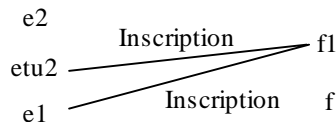
1.2.4 Atomicité des attributs, classe normalisée

Atomicité des attributs, classe normalisée : lorsque la modélisation des données est faite en vue d'une implémentation 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

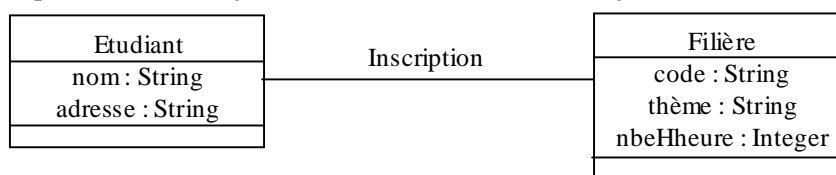
Liens entre objets (diagramme d'objets) : Dans un diagramme d'objets on peut établir des liens entre des objets. Par exemple, les étudiants e1 et etu2 sont *inscrits* en filière f2. Ce qui se représente graphiquement au niveau du diagramme d'objets de la manière suivante :



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.

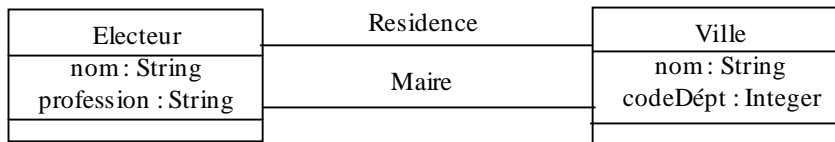
A toute association, définie par un nom, correspond un ensemble de classes, définies dans le diagramme de classes. Tout lien étant une instance d'une association a pour nom, le nom de l'association correspondant et relie un objet de chaque classe caractérisant l'association.

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 :

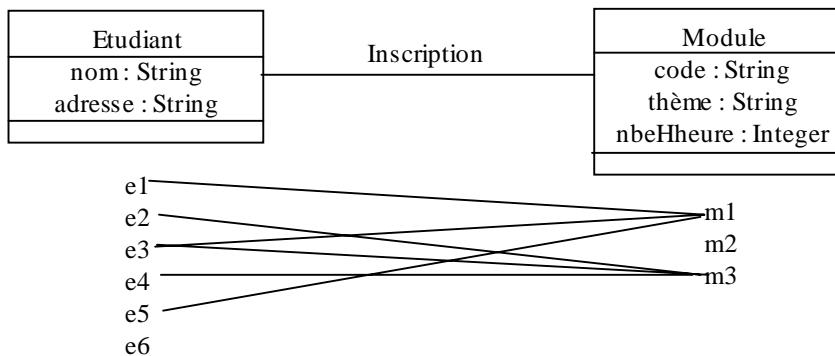
- le nom et le numéro de téléphone de chaque employé sont enregistrés dans la base de données ainsi que le nom et l'activité de chaque département ; une grille de salaire donne pour chaque salaire, un indice et le montant de ce salaire ;

- 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)

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 }
- ...

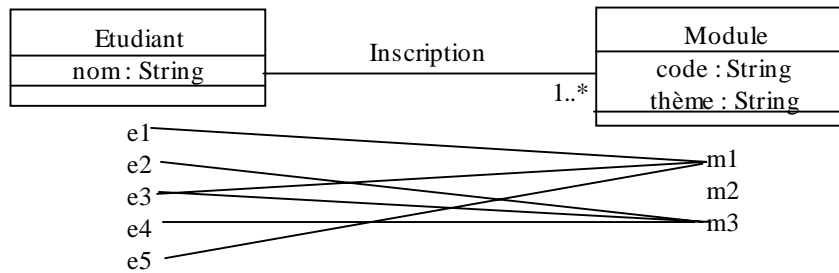
D'une manière générale, à tout objet $e \in \text{Etudiant}$, on associe dans le cadre de l'association Inscription un ensemble de modules liés à e, dont le nombre peut varier selon les mises à jour effectuées sur le diagramme d'objets.

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).

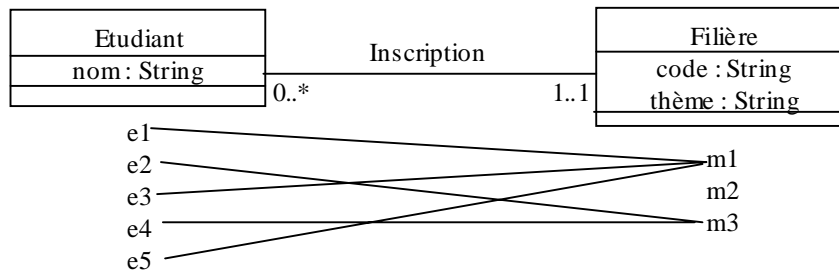
Ce couple d'entiers exprimant les multiplicités d'une association vis à vis d'une classe :

- 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.

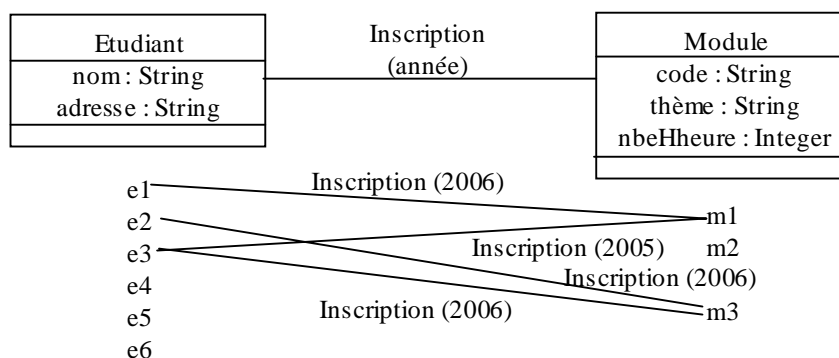
Exemple : On suppose que l'on a une base de données qui centralise les données de l'état civil concernant les français et les françaises. On suppose que l'on a enregistré pour tout français son nom, sa date de naissance et le nom de la ville et du pays où il est né. Il en est de même pour toute française. 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) :

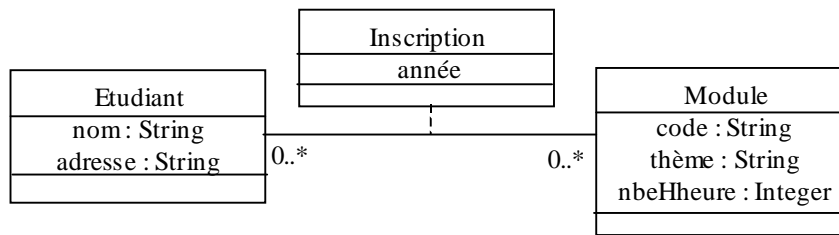
Une association peut être porteuse d'information : une telle association est appelée classe d'association. Ce qui signifie qu'à tout lien, instance d'une telle classe d'association devra correspondre des données conformes à ce qui est déclaré dans la classe d'association.

Par exemple, le diagramme suivant montre que l'on souhaite enregistrer l'année où tout étudiant s'inscrit à un module :



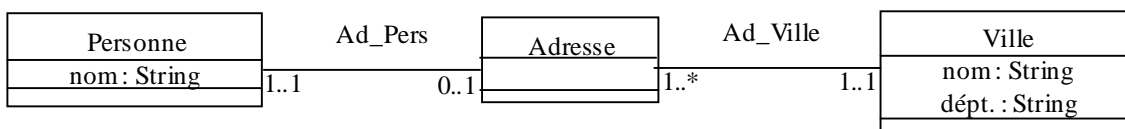
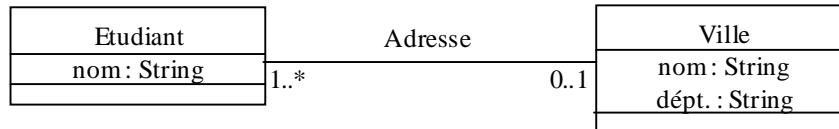
De cette instance, on en déduit, par exemple, que e3 est inscrit au module 1 en 2005 et au module 3 en 2006.

La représentation en UML d'une classe d'association est la suivante :



Remarque : association ou classe ?

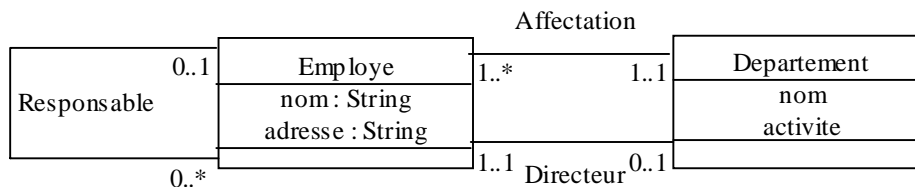
Une association (ou classe d'association) peut être vue comme une classe. Les 2 diagrammes suivants définissent des structures de données équivalentes :



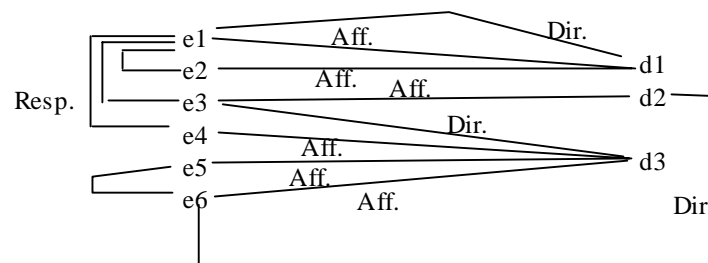
question : donner, pour chaque diagramme, l'instance modélisant le même ensemble de données. Justifier les passages des multiplicités du premier diagramme, par rapport au deuxième diagramme.

1.2.7 Les rôles

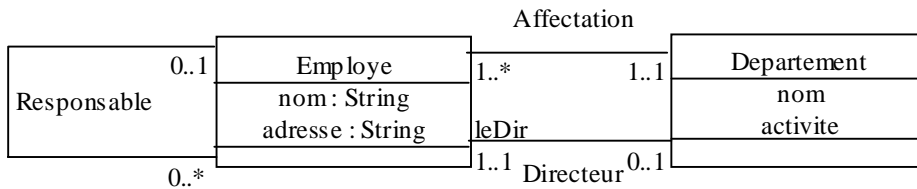
Objectifs des rôles : Les rôles permettent d'apporter plus de lisibilité, plus de compréhension aussi bien au niveau du diagramme de classes qu'au niveau du diagramme d'objets. Par exemple, le diagramme de classes suivant ne permet pas de savoir comment s'interprète l'association Responsable, par rapport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



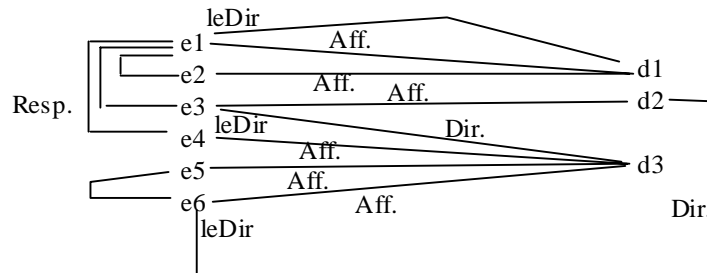
Cette imprécision se répercute au niveau du diagramme d'objets, où, dans certains cas, on ne peut pas savoir qui est responsable de qui :



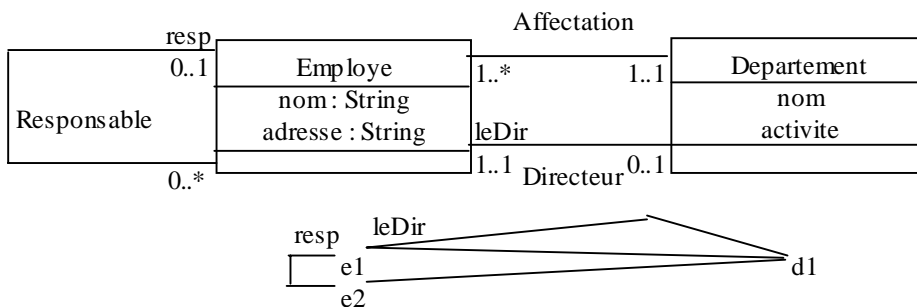
Un rôle est un nom que l'on met au niveau de l'extrémité d'une association (au même niveau que les multiplicités) tel que le montre le diagramme de classe suivant :



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 :

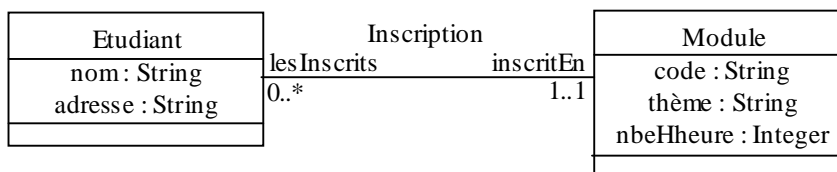


Un nom de rôles peut être déclaré à chaque extrémité d'une association. D'une manière générale un nom de rôle facilite la lecture et l'interprétation d'un diagramme de classes, mais, pour ne pas surcharger les diagrammes de classes et leurs instances on ne met que le nom des associations et des rôles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

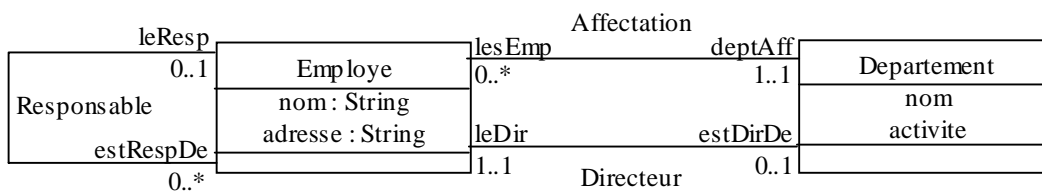


Cette notation permet de voir que, d'après cette instance précédente, le responsable de l'employé e2 est e1.

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.

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

Entreprise E		
<u>Facture</u>		
nom du Client : <i>Dupont</i>	numéroF : 123	
	date de facturation : 10/03/2006	
	date de paiement :	
produits facturés :	nom du produit	quantité facturée
	<i>vis</i>	35
	<i>clou</i>	30
	<i>pointe</i>	50

Ce qui peut se modéliser de la manière suivante :

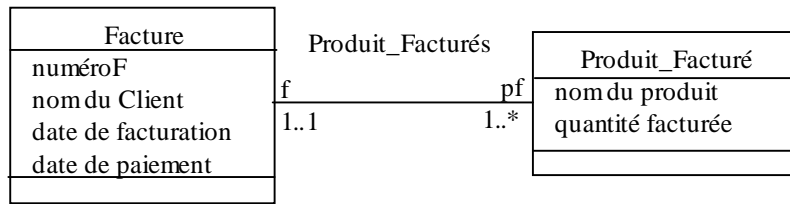
Facture
numéroF
nom du Client
date de facturation
date de paiement
produits_facturés(nom du produit, quantité facturée)

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 'produits facturés' 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).

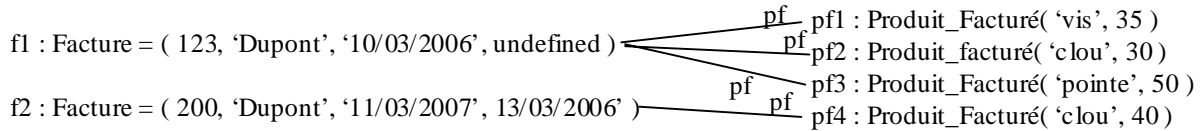
On rappelle que tout diagramme de classes modélisant des données en vue d'une gestion relationnelle des données, doit être normalisé : ce qui implique que tout attribut de la classe doit être atomique. La modélisation précédente de Facture ne peut donc pas être considérée comme une classe normalisée. 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 : Produits_Facturés ;
- 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 :

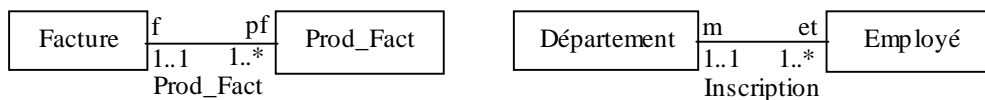


Ce diagramme d'objets, instance du diagramme de classes ci-dessus, montre que 2 factures ont été enregistrées : sur la première on trouve 3 produits facturés et sur la deuxième, 1 seul produit a été facturé : le client de nom Dupont se voit facturé des clous en quantité 40.

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.

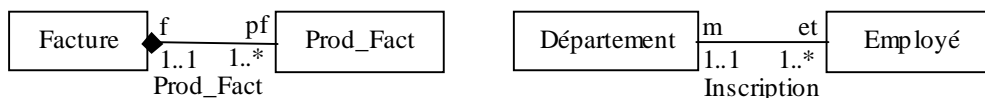
Lorsque d'un diagramme de classes est conçu en vue d'une implantation Java, par exemple, cette étape préliminaire de décomposition d'une classe n'est pas nécessaire, puisque un attribut Java peut être implanté à l'aide d'un tableau, ce qui n'est pas le cas en relationnel.

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 :



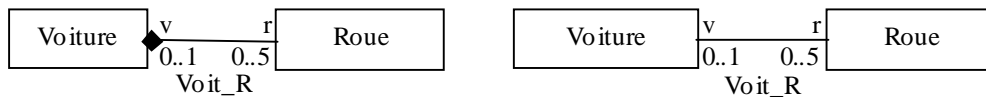
Si ces deux modélisations des données ci-dessus sont identiques structurellement parlant, en fait au niveau applicatif, la destruction d'une facture entraîne la destruction des produits facturés dans cette facture ; par contre, la destruction d'un département ne peut pas se faire si des employés sont 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 être transformée en ce que l'on appelle une Composition et se note habituellement à l'aide d'un losange noir tel que le montre la figure suivante :



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 manipulation. 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 que 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 toutes façons, 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,
- ...

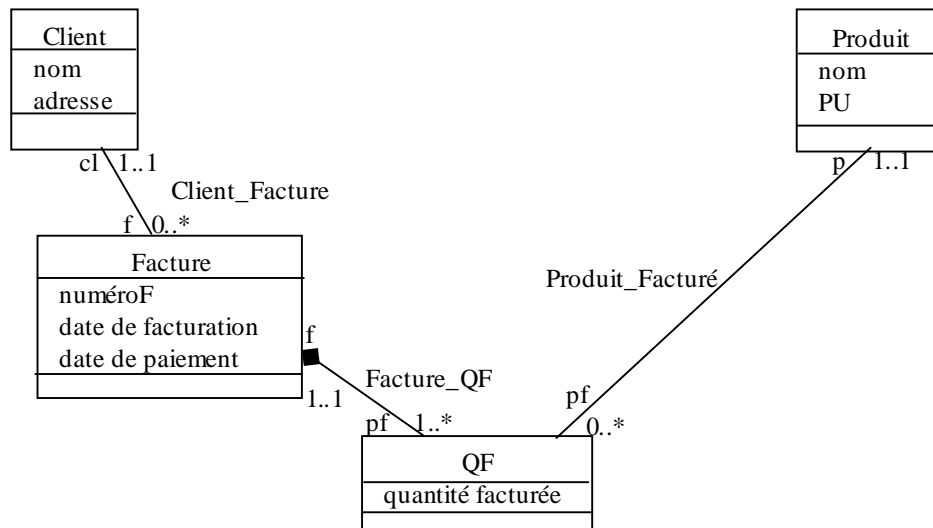
1.3 Exemples de modélisation des données d’une application

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.

Une modélisation à l’aide du diagramme de classes pourrait être la suivante :



1.3.2 Exemple 2 (Gestion de bons de commandes, des livraisons et des factures)

Exercice de réflexion (à préparer durant les congés de printemps): Une entreprise souhaite développer une application gérant les bons de commandes émis par ses clients. On enregistre :

- pour tout client de l'entreprise, son nom et son adresse, et pour tout produit mis au catalogue de l'entreprise, son nom et son prix unitaire,
- pour toute commande envoyée par un client, un numéro, la date de réception du bon de commande, et pour chaque produit commandé sa quantité commandée.

D'autre part, à chaque livraison de produits commandés par un client, on crée un bon de livraison contenant un numéro et la date de livraison des produits.

question 1 : donner une modélisation des données de cette application à l'aide d'un diagramme de classes, et en déduire une instance de votre diagramme de classes ;

question 2 : en fait la première question n'est pas suffisamment précise pour que l'on puisse donner une modélisation correcte (?) des données. Pour chaque sous-question suivante, donner une modélisation ainsi qu'une instance de votre modèle :

2-1 : l'entreprise effectue une livraison pour chaque bon de commande, dès qu'elle a en stock tous les produits commandés dans le bon de commande ;

2-2 : souvent les livraisons ne peuvent pas se faire dès réception des bons de commandes ; ainsi il pourrait y avoir plusieurs bons de commandes d'un même client, en attente de livraison : l'entreprise peut donc être amenée à livrer, en fonction de son stock, en une livraison tous les produits commandés par un client dans plusieurs bons de commandes (dans ce cas toute la description des données est-elle prise en compte ?) ;

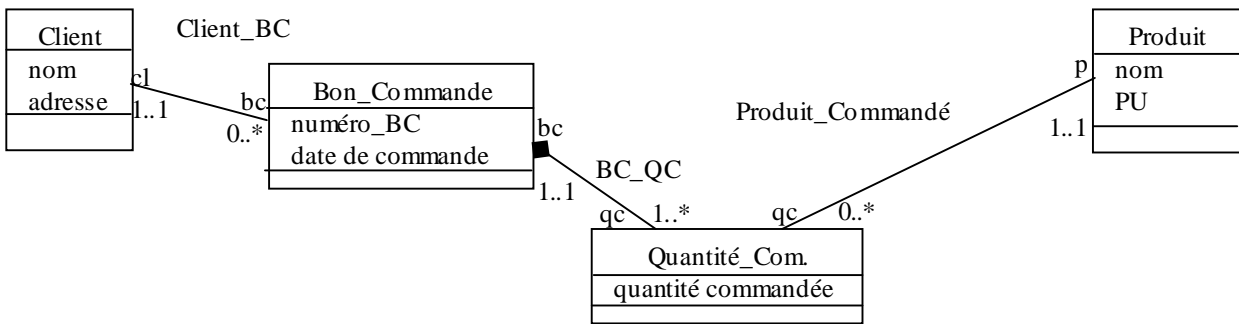
2-3 : pour tenter de satisfaire la clientèle autant que l'entreprise peut le faire, elle peut être amenée, en fonction de son stock, à effectuer des livraisons partielles de produits commandés dans un bon de commande. Dans ce cas, dans chaque bon de livraison on doit retrouver la quantité livrée de chaque produit commandé ; lorsque tous les produits d'un bon de commande sont livrés, une facture est envoyée au client : elle contient un numéro de livraison, bien sûr, la date de facturation et la date d'acquittement de la facture (dans ce cas toute la description des données est-elle prise en compte ?) ;

2-4 : que devient votre modélisation si l'entreprise serait amenée à effectuer, en une livraison, des livraisons partielles de produits issus de plusieurs bons de commande d'un même client ?

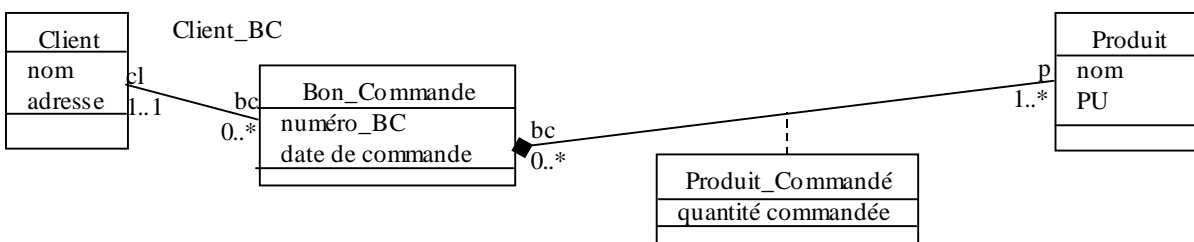
2-5 : ...

éléments de solution de cet exercice de réflexion

question 1 : tout d'abord, il suffit de partir de l'exercice précédent sur les factures et de l'adapter aux bons de commandes :



remarque : puisque une quantité commandée doit être liée à **un** bon de commande et à **un** produit (multiplicités 1..1 aux deux extrémités) on en déduit que la classe Quantité_Com. pourrait être vue comme une association (ou plus exactement comme une classe d'association puisqu'elle est porteuse d'information). On aurait pu avoir la modélisation suivante :



On s'éloigne de la vision que l'on peut avoir d'un bon de commande pouvant contenir plusieurs lignes de commandes de produits, pour chacun d'entre eux, la quantité commandée. Le résultat de la conception ne doit pas forcément aboutir à une photographie de la réalité : un modèle doit être le plus simple possible, au départ, mais il doit surtout permettre de 'restituer' la réalité, sans ambiguïté. A l'usage, ou après réflexion, ou après estimation des performances de l'application, le modèle pourrait évoluer.

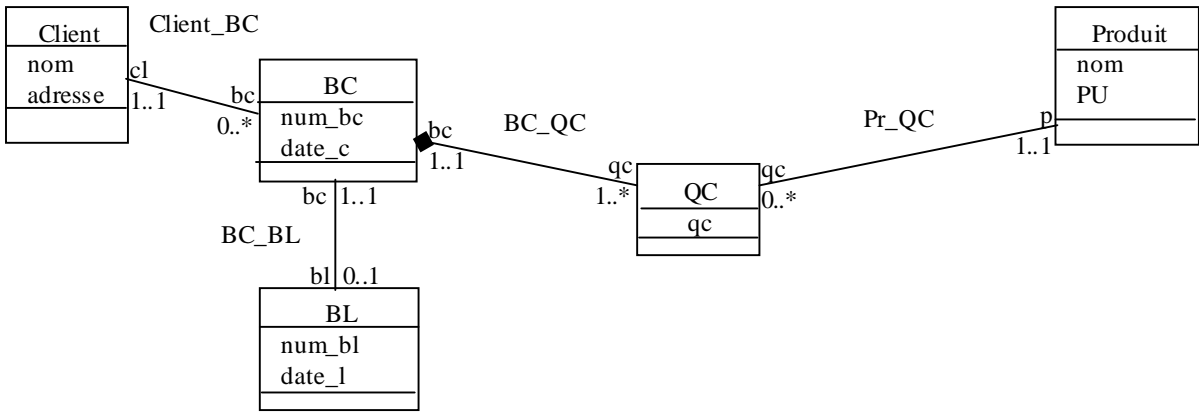
En ce qui concerne les bons de livraisons, il n'y a pas assez d'informations dans la question 1 du texte pour compléter le modèle ci-dessus. L'objectif de la question 2 est

- de modéliser les informations concernant les livraisons des produits commandés par les clients,
- d'intégrer la modélisation des bons de livraison dans le modèle des bons de commandes : cette intégration dépend de la manière dont les livraisons sont gérées.

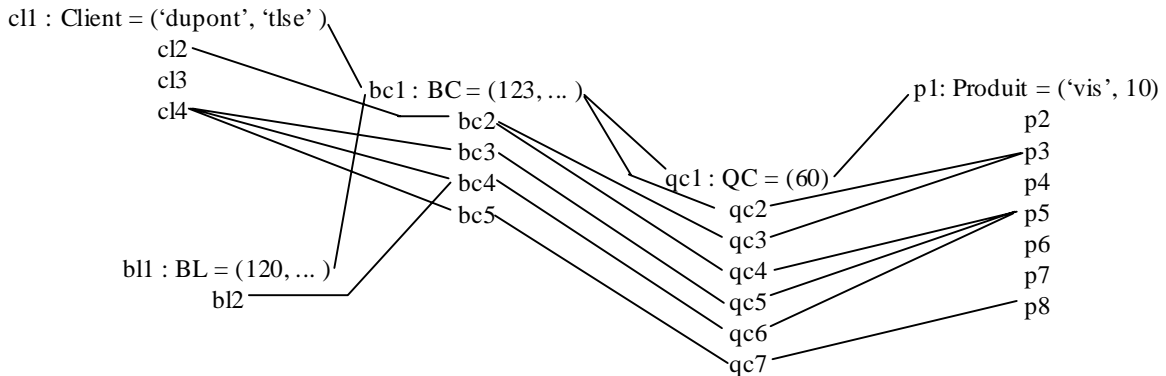
La question 2 propose différentes manières de gérer les bons de commandes dans l'entreprise, ce qui permettra, en fait, d'étudier différents modèles de données.

question 2 :

2-1 : soit, il suffit de rajouter dans le bon de commande (classe BC) la date de livraison, qui sera mise à jour lorsque la livraison sera faite, soit on crée une classe Bon de Livraison (classe BL) permettant ainsi de créer un objet bon de livraison lorsque la livraison sera effectuée, et qui devra être liée au bon de commande correspondant :

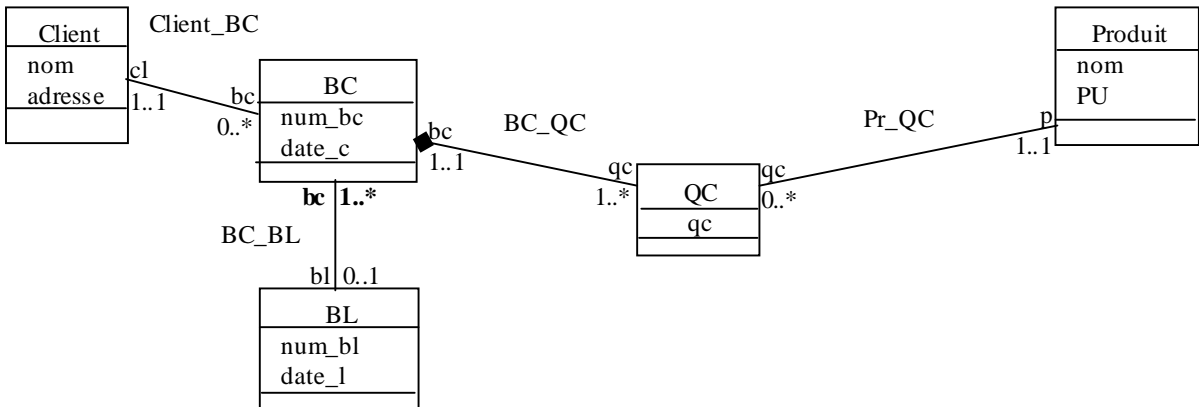


exemple d'instance de ce diagramme de classes :



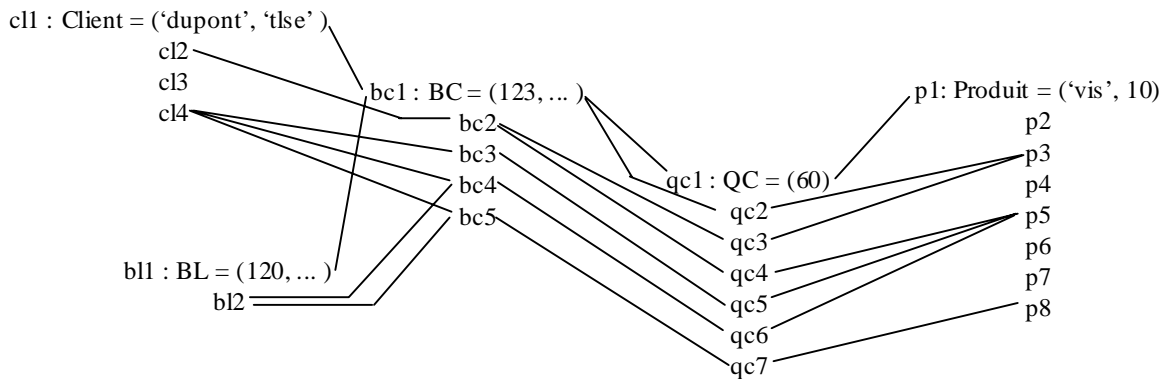
De ces diagramme de classes et d'objets, on en déduit, en particulier, que le client de nom 'Dupont' a passé une seule commande pour 60 vis. Un autre client, dont c14 est l'identifiant de l'objet a émis 3 bons de commandes, chacun ne portant que sur une quantité de produits commandés. ... Seuls, les produits commandés dans les bons de commandes d'identifiants bc1 et bc4 ont été livrés.

2-2 : l'entreprise peut effectuer en une livraison à un client, les produits qu'il a commandés dans plusieurs bons de commandes :



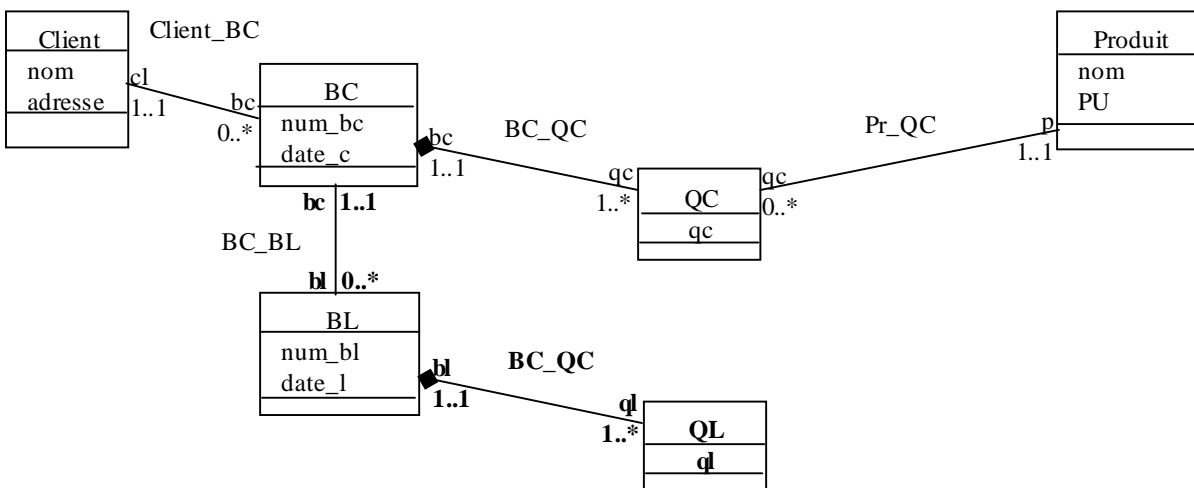
remarque : ce modèle (structurel) ne vérifie pas que tout bon de livraison fait référence à des bons de commande venant du même client. On pourrait rajouter une association permettant de relier tout bon de livraison au client concerné. Mais, ça n'assure pas que tout bon de livraison fait référence à des bons de commandes venant du même client : en fait une contrainte supplémentaire serait rajoutée !!

Exemple d'instance du diagramme de classes précédent :

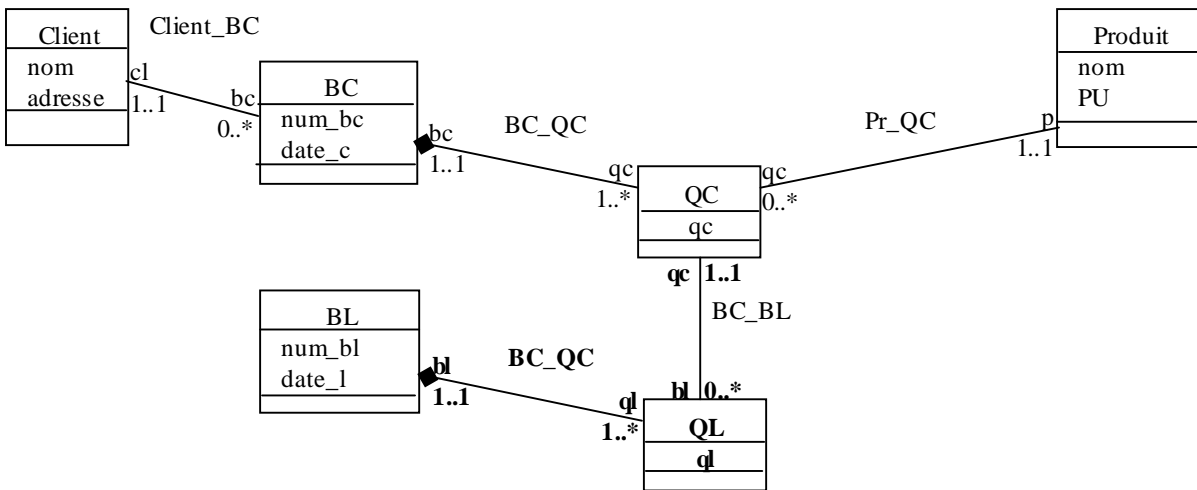


commentaire : le bon de livraison bl2 montre que tous les produits commandés dans les bons de commande bc4 et bc5 ont été livrés, d'une part et que d'autre part ces 2 bons de commande sont issus de même client (c14), dont la commande bc3 n'a pas encore été honorée.

2-3 : les produits commandés dans un bon de commandes pourraient être livrés en plusieurs fois : dans ce cas, il faut que dans le bon de livraison, on puisse enregistrer la quantité de chaque type de produits livrés :



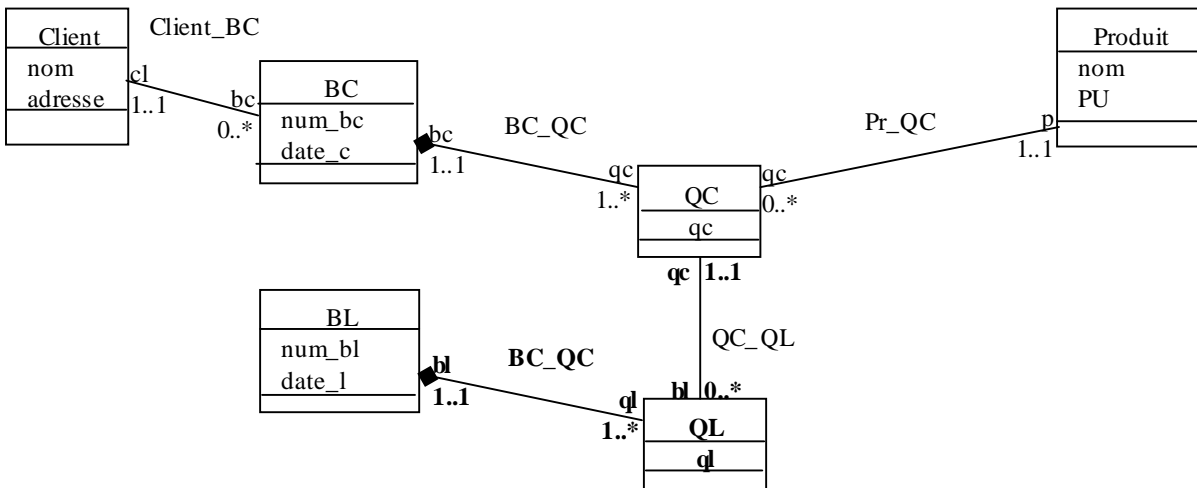
Mais, il faudrait vérifier que les quantités livrées correspondent bien à des produits commandés dans le bon de commande correspondant : il faudrait donc associer les classes QL et QC. Il est toujours nécessaire de vérifier que les produits livrés référencés dans un bon de livraison correspondent bien à des commandes venant du même bon de commande : l'association entre BC et BL devient donc inutile. On pourrait donc avoir le modèle :



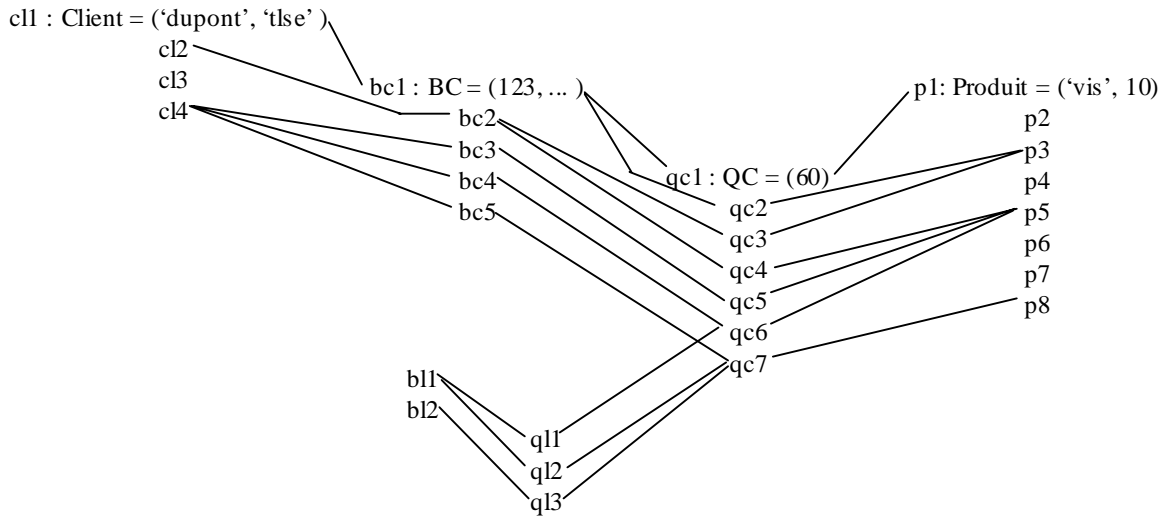
avec les deux contraintes :

- c1- : les produits livrés référencés dans un bon de livraison correspondent à des produits commandés dans un seul bon de commandes.
- c2- : les produits livrés référencés dans un bon de livraison correspondent à des bons de commandes issues d'un même client.

2.4 : le modèle (structurel) est le même :

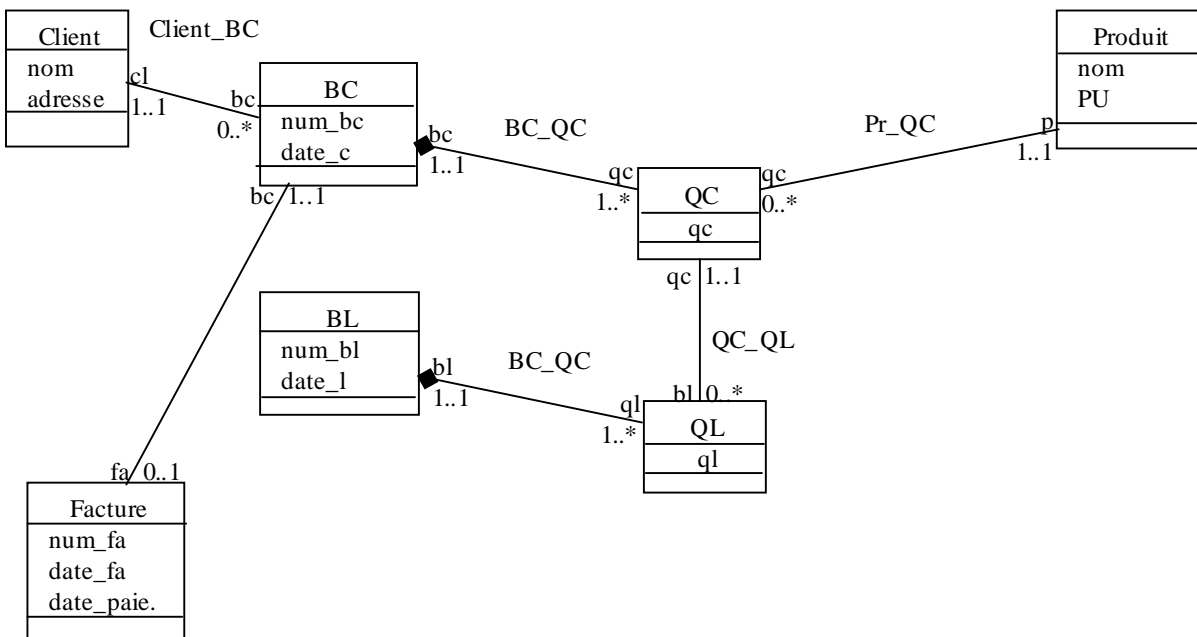


mais la contrainte c1 n'est plus d'actualité.



Il existe une contrainte importante non mentionnée jusqu'ici :

- on ne doit pas livrer plus que ce qui a été commandé : c'est-à-dire que pour chaque quantité de produits commandés, la quantité de produits livrés pour cette commande doit rester inférieure ou égale.



autre contrainte : une facture existe que si les produits du bons de commandes correspondant ont tous été livrés.

remarque : organisation (chronologique) du diagramme de classes, et en conséquence des diagrammes d'objets.

1.3.3 Exemple 3 (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 fait, 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 Introduction au langage OCL (Object Constraint Language)

2.1 Aspect structurel des éléments de modélisation du diagramme de classes

Un diagramme de classes est le résultat d'une modélisation faite à partir d'une description informelle rédigée en texte libre par les responsables de l'application. Cette description est intégrée dans un document appelé cahier des charges de l'application, ou expression des besoins de l'application, ou encore description des exigences, ...

Cependant, tout ce qui est décrit dans le texte dérivant les données de l'application peut ne pas forcément se modéliser à l'aide d'un diagramme de classes qui ne définit qu'une structure de données. Le fait, par exemple, qu'il ne peut exister deux bons de commandes ayant le même numéro ne peut pas s'exprimer à l'aide des éléments de modélisation définis dans le diagramme de classes.

Le langage OCL (Object Constraint Language), partie intégrante de la norme UML, a été créé pour exprimer les éléments du cahier des charges non exprimables dans un diagramme de classes.

Une modélisation (UML) des données d'une application est donc constituée de 2 parties :

- un diagramme de classes définissant la structure des données de l'application,
- des expressions booléennes OCL, appelées spécifications OCL, exprimant des contraintes (propriétés) sur les données de l'application : ces spécifications sont un moyen de préciser d'une manière formelle (à l'aide d'expressions logiques) des contraintes écrites en texte libre.

Pour être cohérent par rapport à une modélisation UML, les objets et les liens d'un diagramme d'objets doivent donc vérifier :

- la structure définie par le diagramme de classes (on parle de cohérence structurelle),
- les spécifications OCL accompagnant le diagramme de classes (on parle de cohérence sémantique)

La cohérence sémantique ne peut se vérifier que si les objets et les liens sont cohérents par rapport à la structure définie dans le diagramme de classes.

Exemple : chez un grand garagiste, on pourrait imaginer une base dont les données concerneraient les voitures dont les maintenances sont effectuées dans le garage et les clients du garage. La description des données pourrait en être la suivante :

- pour tout client, on enregistre son nom, son année de naissance, son numéro de téléphone ainsi que le nom de la rue et de la ville où il habite,
- pour toute voiture, on enregistre son numéro, sa marque, son modèle,
- toute voiture a un propriétaire qui est un client,
- tout propriétaire de voiture doit avoir plus de 18 ans.

question 1 :

- 1.1 : modéliser les données de cette application à l'aide d'un diagramme de classes,
 - 1.2 : tous les éléments de la description des données ont-ils pu être pris en compte dans le diagramme de classes ? dans le cas négatif indiquer ces éléments,
- question 2 :
 en supposant que les expressions OCL complétant le diagramme de classes ont pu être écrites, donner :
- 2.1 : un exemple pertinent des données, par rapport à votre modèle de données,
 - 2.2 : un exemple de données, cohérente structurellement et sémantiquement,
 - 2.3 : des exemples de données, ne vérifiant pas la cohérence sémantique,
 - 2.4 : des exemples de données ne vérifiant pas la cohérence structurelle.

La réalisation d'une application base de données demande généralement un lourd investissement, et nécessite un temps plus ou moins long avant que la base de données ait pu acquérir suffisamment d'informations pour répondre pleinement à ce que l'on en attend d'elle. Si à ce moment-là, on s'aperçoit qu'elle ne répond pas aux exigences des utilisateurs finals (performances insuffisantes, informations incomplètes, ...) il faut alors revoir le modèle de la conception, re-adapter les données déjà rentrées à cette nouvelle modélisation et re-écrire les programmes d'applications.

C'est pourquoi, généralement dès que le modèle de données a été défini, on a l'habitude :

- de re-écrire le document décrivant les données de l'application en re-formulant le texte à partir du modèle des données pour montrer aux responsables de l'application comment toute la description des données a pu être prise en compte au niveau modélisation (c'est un moyen de s'assurer que les concepteurs et les responsables de l'application sont bien en phase),
- d'écrire les expressions OCL correspondant aux manipulations attendues les plus courantes pour s'assurer que la structure des données se prête bien à ces manipulations et que les estimations des performances répondront aux exigences attendues par les utilisateurs finals.

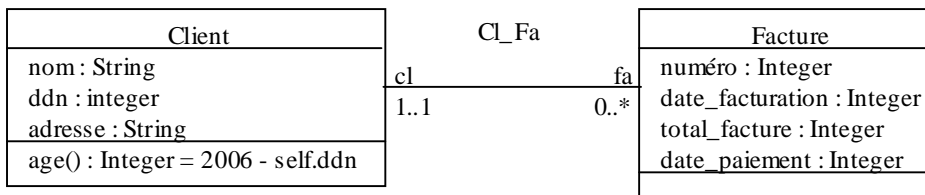
Dans ce cours, on présente les quelques éléments du langage OCL permettant de mettre en pratique ces différents points de vue très importants. Il ne s'agit pas de faire une étude détaillée de OCL comme pour le langage SQL. Pour ceux qui seraient intéressés par le langage OCL, il existe de très nombreux sites sur le WEB où l'on trouve des cours sur OCL, preuve que le langage a une importance de plus en plus grande.

Nous ne verrons dans OCL les aspects les plus proches de SQL : OCL peut être utilisé comme une étape intermédiaire avant d'écrire une requête SQL répondant à une question plus ou moins complexe.

2.2 Concepts de base du langage OCL

OCL est un langage récent qui intègre de nombreux concepts. OCL est un langage sans effet de bord, typé s'appliquant sur les objets et les liens d'un diagramme de classes objet. C'est un langage fonctionnel, basé sur la logique ensembliste des prédicats du premier degré, navigationnel et déclaratif.

Pour écrire les expressions OCL, on s'appuiera sur la modélisation suivante :



```

c11 : ( 'dupont', 1980, 'Toulouse' )
c12 : ( 'dulong', 1970, 'Paris' )
c13 : ( 'ducourt', 1975, 'Albi' )
c110 : ( 'durand', 1985, 'Toulouse' )

f1 : ( 1, 050102, 50, 050201 )
f2 : ( 2, 050102, 100, 050105 )
f3 : ( 3, 051230, 20, 060105 )
f4 : ( 4, 060310, 30, oclUndefined )
f5 : ( 5, 060310, 80, 060401 )

```

ce qui signifie, par exemple que l'objet dont l'identifiant est f4, est une facture liée à l'objet client c110. Cette facture dont le total est 80 a pour numéro 5, a été élaborée le 10 mars 2006, et a été acquittée le 1^{er} avril 2006. Cette facture a été envoyée au client de nom durant, né en 1985 et habitant Toulouse.

2.2.1 OCL est un langage sans effet de bord

OCL est un langage 'sans effet de bord' : OCL n'a aucune action de modification sur un diagramme d'objets, contrairement aux ordres du LMD SQL (*insert*, *update* ou *delete*) qui permettent la mise à jour des données d'une base relationnelle. De même, OCL n'a aucune action sur un diagramme de classes, contrairement aux ordres du LDD SQL (*create table*, *alter table*, ...) qui permettent de créer et de mettre à jour un schéma de base relationnelle.

Ce qui signifie que toute expression OCL a pour seul objectif de chercher des données ou des objets vérifiant des expressions logiques : le langage OCL est au diagramme de classes et au diagramme d'objets ce que le langage de requêtes *select* du langage SQL est à une base de données relationnelle.

Par exemple, pour rechercher les clients qui habitent à Toulouse, on écrira l'expression OCL :

```
Client.allInstances → select( adresse = 'Toulouse' )
```

2.2.2 OCL est un langage objet

Les expressions OCL s'écrivent par rapport à un diagramme de classes et s'appliquent à un diagramme d'objets, instance du diagramme de classes¹ : OCL manipule donc des éléments qui peuvent être :

- des objets pouvant être reliés entre eux (sans les modifier),
- des données de type de base :
 - Boolean (true, false),
 - String ("Toulouse", par exemple),
 - Integer (... , -10, ... , 0, ... 10, ...),
 - ...

L'appel à une opération se fait par rapport à un élément (objet ou donnée) selon le principe classique des langages objet. Par exemple :

- c12.age() donne 36²,
- "Toulouse".size() donne 8,
- "Toulouse".concat("Paris") donne "ToulouseParis",
- c12.ddn donne 1970³.

¹ Le concept d'héritage, essentiel dans les approches objet, n'est pas traité 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.

² On applique l'opération age() à l'objet c12 (même notation pour l'application de toute opération définie dans la classe Personne)

On retrouve les opérateurs classiques sur les données de base :

```

+, -, ...
=, <>, >, ...
and, or, ... , implies
if ... then ... else ... endif

```

```

exemple :   if cl2.age()    > 18   then "majeur"   -- les expressions introduites par
                                     else "mineur"   -- les mots clés then et else
                                     endif           -- doivent être de même type
exemple :   cl2.ddn = oclUndefined( Integer )   -- OCL 1.3

```

les opérations sur les objets : =, <>

2.2.3 OCL est un langage ensembliste

OCL est un langage ensembliste :

- une expression permet de référencer un ensemble d'objets :

Personne.allInstances est une expression OCL qui réfère toutes les instances de la classe *Personne*

- il est possible de définir des collections d'éléments qui peuvent être des ensembles (Set) contenant des éléments sans doublons, des ensembles avec doublons (Bag) ou des séquences ordonnées d'éléments (Sequence) :

```

Set{ 10, 20, 0, 5 }           donne l'ensemble sans doublons : { 0, 5, 10, 20 }
Set{ "toto", "titi", "tata" } donne l'ensemble sans doublons : {"tata", "titi", "toto" }
Bag{ 10, 20, 0, 5, 10 }      donne l'ensemble avec doublons : { 0, 5, 10, 10, 20 }
Set{ p1, p4, p2 }           donne l'ensemble sans doublons : { p1, p2, p4 }

```

La bibliothèque OCL offre un très grand nombre d'opérations applicables sur les collections. Contrairement à la notation pointée concernant l'application d'une opération sur un objet, l'application d'une opération sur une collection se fait à l'aide d'une flèche :

```

- size :      Set{ 10, 20, 0, 5 }->size()           donne 4
              Client.allInstances->size()          donne 4
              Client.allInstances->select( c | c.adresse = "Toulouse" ) donne 2
- includes :  Set{ 10, 20, 0, 5 }->includes( 0 )     donne true
- includesAll : Set{ 10, 20, 0, 5 }->includesAll( Set{ 0, 10 } ) donne true
- ...
- including : Set{ 10, 20, 0, 5 }->including( 15 )   donne Set{ 0, 5, 10, 15, 20 }
- ...
- union :     Set{ 10, 20, 0, 5 }->union( Set{ 5, 0, 30 } ) donne Set{ 0, 5, 10, 30 }
              Bag{ 10, 20, 0, 5 }->union( Bag{ 5, 0, 0, 30 } )
                                                         donne Bag{ 0, 0, 0, 10, 5, 20, 30 }
- ...
- iterate :   Set{ 10, 20, 0, 5 }->iterate( e ; somme : Integer = 0 | somme + e ) donne 35
              ce qui signifie : - somme : Integer = 0
                                - pour chaque élément e ∈ Set{ 10, 20, 0, 5 }
                                - somme = somme + e
                                - résultat : somme
- collect :   Set{ 10, 20, 0, 5 }->collect( e | e+1 ) donne Bag{ 1, 6, 11, 21 }
              ce qui signifie :
              Set{ 10, 20, 0, 5 }->iterate( e ;
                                                ens : Bag( Integer ) = oclEmpty( Set( Integer ) ) |
                                                ens->including( e+1 )
                                              )

```

³ même notation pour l'application de tout attribut défini dans la classe *Personne*

- ...

2.2.4 OCL est un langage basé sur la logique des prédicats du 1^{er} ordre

En logique des prédicats du premier ordre, on a deux types de formules :

- les formules ouvertes qui font référence à des données ou des objets :
`Client.allInstances→select(x | x.adresse = "Toulouse")`
- les formules fermées qui font appel aux quantificateurs
 existentiel (\exists) (opération exists, en OCL),
 universel (\forall) (opérateur forAll, en OCL)
 et qui retourne donc une valeur booléenne.

exemple :

- question : existe-t-il un client qui s'appelle "dupont" ?
- expression OCL : `Client.allInstances→exists(c | c.nom = "dupont")`
- le résultat est booléen : true

exemple :

- question : existe-t-il deux clients qui ont le même nom ?

littéralement : existe-t-il un client x et un client y, tel que :

- x et y sont différents
- x et y ont le même nom
- expression OCL : `Client.allInstances→exists(x, y | x <> y and x.nom = y.nom)`
- le résultat est booléen : true

exemple :

- question : est-ce que toutes les factures ont des numéros différents 2 à 2 ?

littéralement : il ne doit pas exister deux factures qui ont le même numéro :

- expression OCL : `not (Facture.allInstances→exists(x, y | x <> y and x.numero = y.numero))`

ou littéralement : pour tout couple x et y de facture,

si x est différent de y alors, le numéro de x doit être différent du numéro de y
 sinon, l'expression est vrai pour ce couple x, y :

- expression OCL : `Facture.allInstances→forall(x, y | if x <> y then x.numero <> y.numero else true endif)`

ce qui s'écrit généralement avec l'opérateur d'implication :

`Facture.allInstances→forall(x, y | x <> y implies x.numero <> y.numero)`

2.3 L'approche fonctionnelle du langage OCL

L'aspect fonctionnel du langage OCL permet d'intégrer logiquement les concepts OCL présentés précédemment.

2.3.1 OCL est un langage fonctionnel

Ce qui signifie que toute expression OCL retourne un résultat qui peut être :

- une donnée de base (Boolean, String, Integer, ...)
- un objet de l'une des classes du diagramme de classes (Client, Facture, ...)
- une collection de données de base (Set(Integer), Bag(Integer), Set(String), ...)
- une collection d'objets (Set(Client), Bag(facture), ...)

Tout interprète retourne deux informations après évaluation d'une expression OCL : le **résultat** et le **type du résultat**.

Par exemple :

Client.allInstances→select(x | x.adresse = "Toulouse") retourne Set{ @c11, @c110 } : Set(Client)

Client.allInstances→exists(x | x.adresse = "Toulouse") retourne true : Boolean

Remarque : étant donné que le langage OCL est sans effet de bord (c'est-à-dire que toute expression ne modifie en aucune sorte le diagramme d'objets sur lequel porte l'expression) la notation @c11 est un pointeur vers l'objet c11.

2.3.2 Enchaînement des opérations sur les collections

Toute expression retourne un résultat typé, donc on peut appliquer une opération à ce résultat : c'est ce que l'on appelle l'enchaînement des opérations. Exemple :

- Client.allInstances→select(x | x.adresse = "Toulouse") retourne Set{ @c11, @c110 } : Set(Client)

Toute opération définie sur les ensembles de clients peut être appliquée à ce résultat. On pourrait alors écrire :

- Client.allInstances→select(x | x.adresse = "Toulouse")→size() retourne 2 : Integer

- Client.allInstances→select(x | x.adresse="Toulouse")→select(Client.allInstances→select(x | x.ddn > 1980))
retourne Set(@c110) : Set(Client)

L'interprète OCL évalue l'expression de gauche à droite :

- tout d'abord : Client.allInstances qui retourne Set{ @c11, @c12, @c13, @c110 } : Set(Client)
- ensuite : Set{ @c11, @c12, @c13, @c110 }→select(x | x.adresse = "Toulouse")
qui retourne : Set{ @c11, @c110 } : Set(Client)
- enfin : Set{ @c11, @c110 }→select(x | x.ddn > 1980) qui retourne Set{ @c110 } : Set(Client)

- Client.allInstances→select(x | x.adresse = "Toulouse").nom

L'interprète évalue la première partie de l'expression :

Client.allInstances→select(x | x.adresse = "Toulouse")
qui retourne Set{ @c11, @c110 } : Set(Client)

l'interprète évalue ensuite :

Set{ @c11, @c110 }.nom qui a pour effet de retourner le nom des clients c11 et c110. Cette dernière expression retourne donc : Bag{ "dupont", "durand" } : Bag(String)

2.3.3 Navigation des expressions OCL

Les expressions OCL de navigation permettent de retourner des objets qui sont liés entre eux.

Par exemple, l'expression OCL : f2 retourne @f2 : Facture, c'est-à-dire qu'elle fait référence à l'objet Facture dont l'identifiant est f2.

Il existe une association entre la classe Client et la classe Facture, dont les rôles sont :

- cl au niveau de l'extrémité de l'association vers la classe Client
- fa au niveau de l'extrémité de l'association vers la classe Facture.

Ce sont ces noms de rôles qui sont utilisés dans l'expression pour retrouver les objets liés entre eux. Par exemple pour retrouver le client à qui la facture f2 a été adressée, c'est-à-dire le client qui est lié à f2 dans le cadre des liens d'associations Cl_Fa, on écrira l'expression OCL : f2.cl

L'interprète évalue

- d'abord f2 qui retourne @f2 : Facture
- le rôle cl permet, à l'interprète, d'accéder à l'objet client d'identifiant c11, à partir de l'objet f2 : on dit que l'interprète navigue dans le diagramme d'objets.

Exemples :

- | | |
|-------------------|---|
| . question : | à qui est envoyée la facture f2 ? |
| expression OCL : | f2.cl |
| résultat retourné | @c11 : Client |
| | |
| . question : | nom du client à qui est envoyée la facture f2 ? |
| expression OCL : | f2.cl.nom |
| résultat retourné | "dupont" : String |
| | |
| . question : | factures adressées au client c11 ? |
| expression OCL : | c11.fa |
| résultat retourné | Set{ @f1, @f2, @f3 } : Set(Client) |
| | |
| . question : | numéros des factures adressées au client c11 ? |
| expression OCL : | c11.fa.numero |
| résultat retourné | Bag{ 1, 2, 3 } : Bag(Integer) |
| | |
| . question : | somme totale des factures adressées au client c11 ? |
| expression OCL : | c11.fa.total_facture→sum() |
| résultat retourné | 170 : Integer |

2.3.4 Remarque sur la visibilité d'un objet

Dans une expression OCL, après un identifiant d'objet, on peut rajouter :

- | | | |
|--|-----------|---|
| . rien : | c11 | retourne @c11 : Client |
| . un attribut de la classe | c11.nom | retourne "dupont" : String |
| . une opération de la classe | c11.age() | retourne 26 : Integer |
| . un rôle d'une association partant de la classe | c11.fa | retourne Set{ @f1, @f2, @f3 } : Facture |

Il en est de même pour une collection d'une classe d'objets :

- | | |
|-----------------------|---------------------------------|
| . Client.allInstances | retourne l'ensemble des clients |
|-----------------------|---------------------------------|

. Client.allInstances.nom	retourne les noms des clients
. Client.allInstances.age()	retourne les âges des clients
. Client.allInstances.fa	retourne les factures adressées à tous les clients

Rappel : une opération s'appliquant à une collection est introduite par la flèche :

```
. Client.allInstances→select( c | c.nom = "dupont" )
. Client.allInstances→select( c | c.nom = "dupont" ).ddn
. Client.allInstances→select( c | c.nom = "dupont" ).age()
. Client.allInstances→select( c | c.nom = "dupont" ).fa
. Client.allInstances→select( c | c.nom = "dupont" ).numero
.      ...
```

L'ensemble des attributs : nom, ddn, adresse, de l'opération age(), du rôle fa constitue ce que l'on appelle la visibilité de tout objet Client. Les noms de rôle servent à naviguer.

2.3.5 Quelques expressions OCL

Toute expression OCL est donc construite de gauche à droite, en alternant les éléments de visibilité et les opérations sur les collections. Un filtre peut apparaître dans une expression booléenne : il se construit selon le même principe.

Exercice : à quoi correspond chacune des expressions OCL suivante :

```
. f1
. f1.cl
. f1.cl.fa
. c11.fa→select( f | f.total_facture > 30 )
. c11.fa→select( f | t.total_facture > 30 )→size()
.      ...

. Client.allInstances→select( c | c.nom = "dupont" )
. Client.allInstances→exists( c | c.nom = "dupont" )
. Client.allInstances→exists( c | c.fa→exists( f | f.total_facture > 30 ) )
. Client.allInstances→select( c | c.fa→exists( f | f.total_facture > 30 ) )
```

2.4 Modélisation : diagramme de classes + contraintes d'intégrité OCL

Les expressions OCL retournant une valeur booléenne peuvent être utilisées pour spécifier les éléments de modélisation qui ne peuvent pas être exprimés à l'aide d'un diagramme de classes.

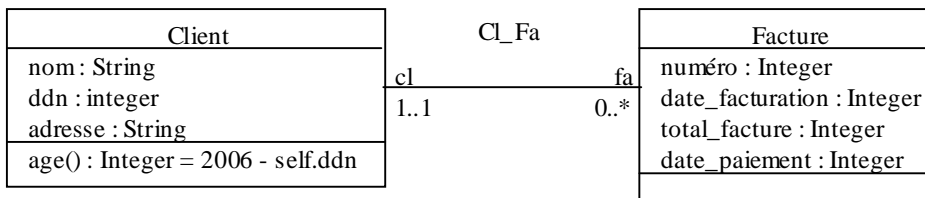
Ces expressions sont appelées des invariants qui ont une syntaxe particulière.

A notre niveau d'enseignement, il sera suffisant de les écrire comme une formule fermée.

Exemple : On reprend l'exercice sur les clients et les factures. On rajoute les contraintes suivantes :

- tout client a plus de 21 ans
- le total de toute facture est positif
- 2 factures ne peuvent pas avoir le même numéro
- tout client ne peut pas avoir plusieurs factures non payées (ce qui pourrait signifier que l'on refuserait de livrer des produits à un client qui n'aurait pas encore acquitté sa dernière facture.

La modélisation de l'application des donc la suivante :



Client.allInstances → forAll(c | c.age() > 21)

Facture.allInstances → forAll(f | f.total_facture > 0)

Facture.allInstances → forAll(f1, f2 | f1 <> f2 implies f1.numero <> f2.numero)

Client.allInstances → forAll(c | c.fa → select(f | f.date_paiement) = oclUndefined(Integer) < 2))

3 Transformation d'un diagramme de classes en un schéma relationnel

3.1 Conception, Génération de code

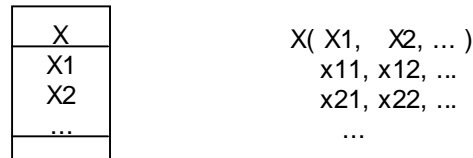
Généralement, on ne génère pas le code du LDD SQL directement à partir d'un diagramme de classes : on préfère transformer le diagramme de classes en un schéma représentant les données au travers de leurs éléments de modélisation relationnelle. Le code du LDD SQL est alors déduit du schéma relationnel. 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.

3.2 Transformation d'une classe dans le modèle relationnel

3.2.1 Principe général de transformation

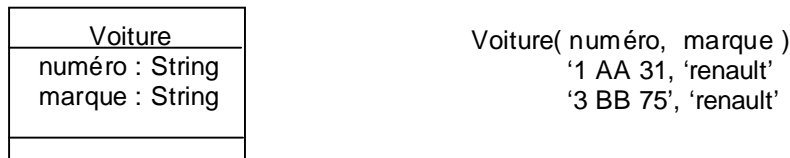
Intuitivement, tel que le suggère la figure suivante, à chaque classe on peut faire correspondre une relation telle que :

- 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.



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

Exemple :



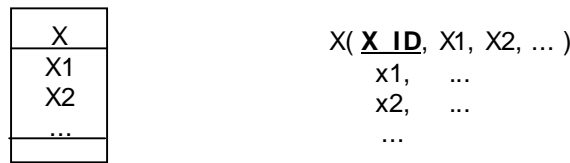
v1 : Voiture('1 AA 31', 'renault')
 v2 : Voiture('3 BB 75', 'renault')

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).

3.2.2 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 :



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

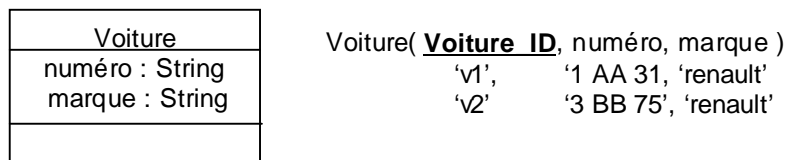
- remarque : . X est une relation, les attributs de cette relation sont les attributs de la classe,
 . l'attribut X_ID joue le rôle de clé primaire de la relation X.
- pré-conditions : . les noms des attributs de la classe X sont différents 2 à 2,
 . le nom des attributs est différent de X_ID,
 . le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

3.2.3 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 :



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 )
                           );
```

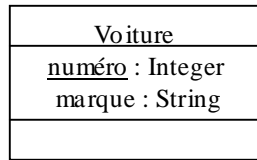
3.2.4 Attribut clé primaire de la relation

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 représentée par la figure suivante :



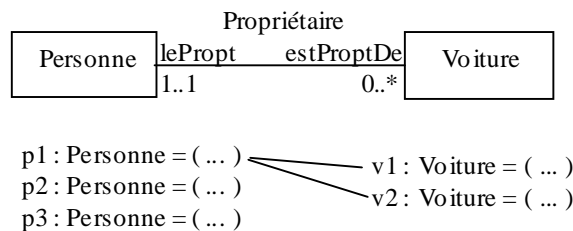
Voiture(numéro, marque)

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

3.3 Principe de transformation des associations et des liens en relationnel

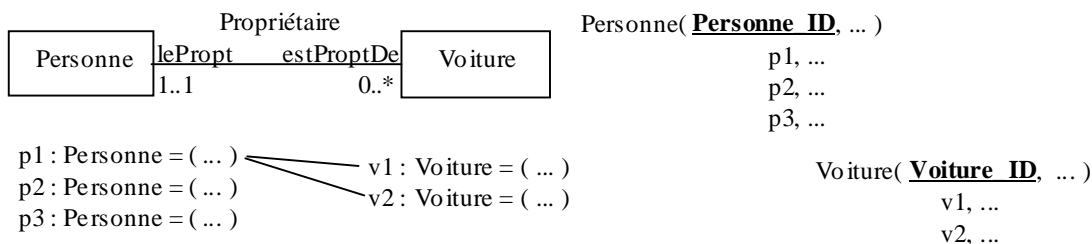
3.3.1 Association par valeur de clé (primaire)

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 :

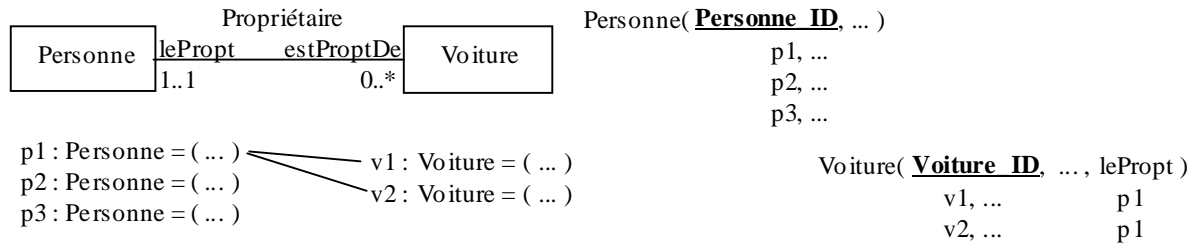


La transformation en relationnel d'un tel diagramme se fait, intuitivement, selon les étapes suivantes :

-1- à chaque classe correspond une relation :

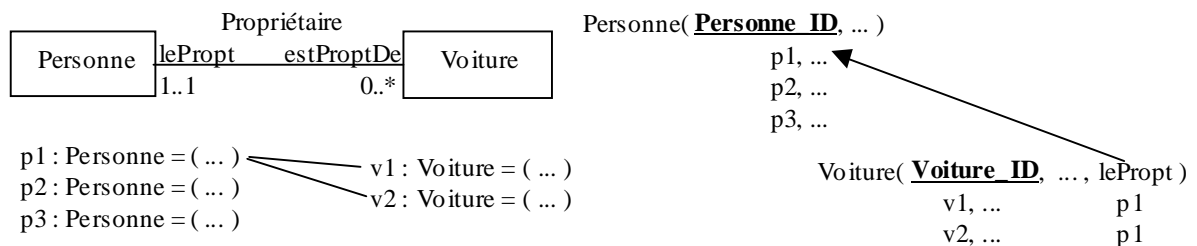


-2- puisque chaque voiture a un propriétaire qui est une personne, il suffit dans la relation Voiture de rajouter une colonne permettant d'indiquer, pour chaque voiture, quel est son propriétaire ; ce propriétaire pourra être représenté par son identifiant, c'est-à-dire sa valeur de clé (primaire) ;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

3.3.2 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).

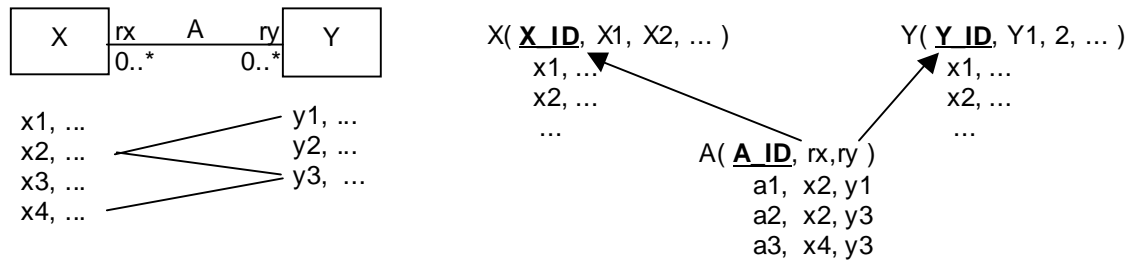
3.4 Transformation d'une association en relationnel (cas général)

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..*.

Dans ce paragraphe, on montre comment, d'une manière générale, les associations se traduisent en relationnel : on étudie d'abord la cas d'une association dont les multiplicités sont $0..*/0..*$, c'est-à-dire sans contraintes particulières. On en déduira ensuite, pour chacune des autres multiplicités ($0..1 / 1..*$, par exemple) comment se traduisent en relationnel les différentes contraintes qu'elles expriment.

3.4.1 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 :

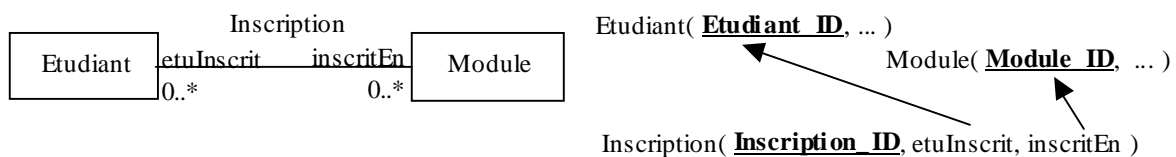


- remarque : . $A(rx) \subseteq X(\underline{X_ID})$, $A(ry) \subseteq Y(\underline{Y_ID})$
 . X et Y sont des relations de base ; A est une relation d'association exprimant des liens entre les tuples des relations X et Y.
- pré-conditions : . les nom de rôles rx et ry sont différents, et sont différents 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.

3.4.2 Exemple

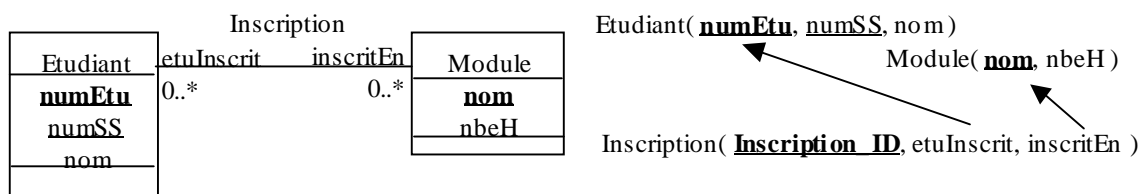


```
SQL> create table etudiant(  Etudiant_ID  varchar( 10 ) primary key,
                           ... ,
                           ... ) ;
```

```
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 ) ) ;
```

3.5 Transformation d'une association en relationnel (en tenant compte des contraintes de multiplicités)

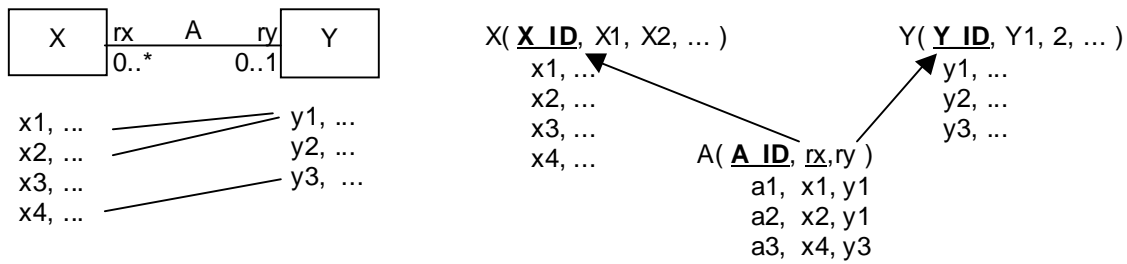
Il suffit d'appliquer les règles de passage d'un diagramme de classes en un schéma relationnel dont l'association correspond au cas général (0..* / 0..*), et de rajouter la (ou les) contrainte(s) de multiplicités qui se traduisent en relationnel sous la forme de clé et/ou de dépendance référentielle éventuellement double-sens.

3.5.1 Multiplicités 0..* / m..M

cas : 0..* / 0..1

Dans ce cas, tout objet x de X ne peut être associé qu'à, au plus, un objet y de Y. Cela signifie que l'on ne peut pas avoir deux tuples de la relation d'association A qui ont la même valeur pour l'attribut rx. A(rx) est donc une clé de la relation A.

- diagramme de classes et schéma relationnel



- propriétés : $A(\underline{rx}) \subseteq X(\underline{X_ID})$, $A(ry) \subseteq Y(\underline{Y_ID})$

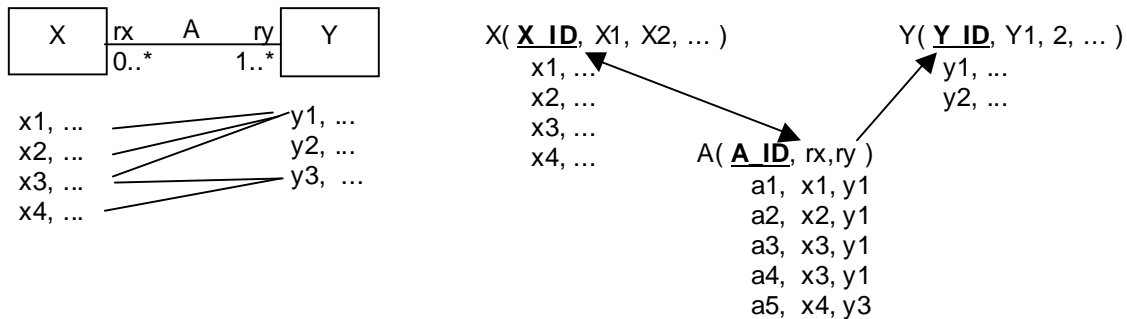
- remarques :
 . la relation A possède 2 clés : une clé primaire et une clé secondaire (unique not null),
 . la clé primaire peut être reporté sur l'attribut rx.

- l'ordre de création de la table A est le suivant :

```
SQL> create table A(  A_ID varchar( 10 ) primary key,
                    rx   varchar( 10 ) references X( X_ID )    unique not null,
                    ry   varchar( 10 ) references Y( Y_ID )    );
```

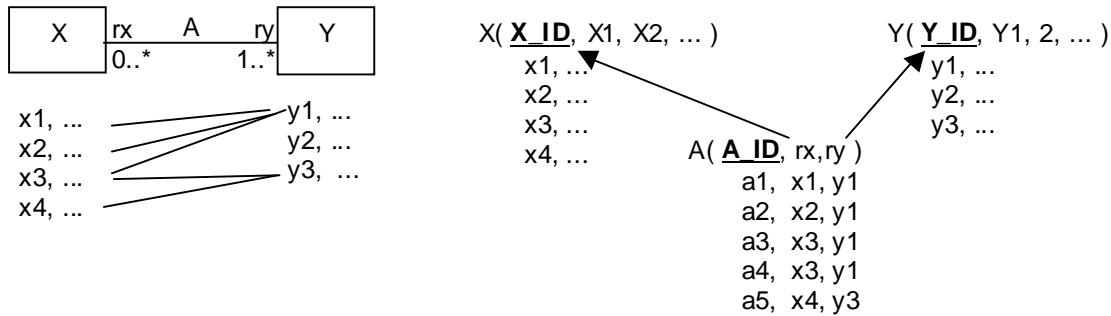
cas : 0..* / 1..*

Dans ce cas, tout objet x de X doit être lié à, au moins, un objet y de Y. Ce qui signifie que l'identifiant de tout tuple x de la relation X doit se retrouver dans la colonne rx de la relation d'association A : on a donc une dépendance référentielle double sens, tel que le montre la figure suivante :



- propriétés : $A(\underline{rx}) \subseteq X(\underline{X_ID})$, $X(\underline{X_ID}) \subseteq A(\underline{rx})$, $A(\underline{ry}) \subseteq Y(\underline{Y_ID})$
- remarques : . la dépendance référentielle est double-sens et ne pourra pas être traduite en LDD SQL : elle devra être intégrée au niveau applicatif (à l'aide de procédures vérifiant cette contrainte).

Le schéma relationnel exprimable en LDD SQL est donc le suivant :



- code du LDD SQL que l'on déduit de ce schéma relationnel :

```

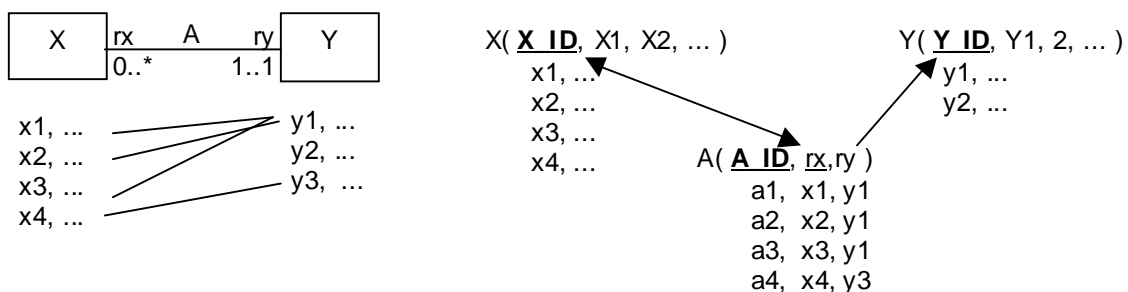
SQL> create table X( X_ID varchar( 10 ) primary key,
                   X1    ... ,
                   ...                                     );
SQL> create table Y( Y_ID varchar( 10 ) primary key,
                   Y1    ... ,
                   ...                                     );
SQL> create table A( A_ID varchar( 10 ) primary key,
                   rx   varchar( 10 ) references X( X_ID ),
                   ry   varchar( 10 ) references Y( Y_ID ) );
  
```

auquel il faut rajouter des triggers ou de sous-programmes pour vérifier lors des mises à jour la dépendance référentielle double-sens qui a été supprimée.

cas : 0..* / 1..1

Dans ce cas, on cumule les deux contraintes : tout objet x doit être lié à un (et un seul) objet y de Y.

- diagramme de classes et schéma relationnel



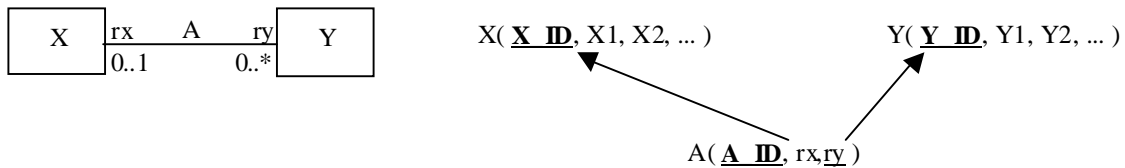
- propriétés : $A(\underline{rx}) \subseteq X(\underline{X_ID})$, $X(\underline{X_ID}) \subseteq A(\underline{rx})$, $A(\underline{ry}) \subseteq Y(\underline{Y_ID})$
- remarques : . du fait que $X(\underline{X_ID})$ et $A(\underline{rx})$ sont clés et sont liés par une dépendance référentielle double-sens, l'attribut ry pourrait être reporté dans la relation X, . en fait, dans ce cas la relation A est inutile, et l'on pourrait avoir e schéma relation, tel qu'il a été défini intuitivement au paragraphe 3.3 : mais le fait de supprimer la

relation d'association A entraîne une perte de modélisation : rx. C'est pourquoi, on préfère garder la relation de lien A, en reportant éventuellement la clé primaire sur l'attribut rx.

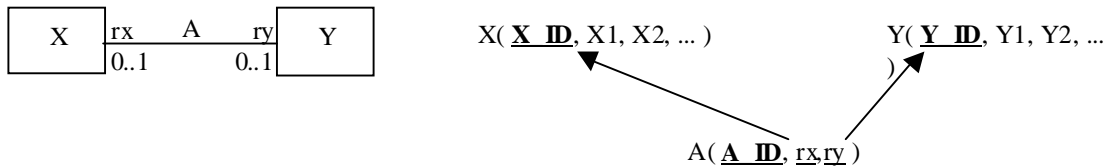
Pour les autres cas de multiplicités, on donne les structures de données équivalentes. Le code du LDD SQL pourra en être déduit en appliquant les règles définies ci-dessus pour chaque type de multiplicités.

3.5.2 Multiplicités 0..1 / m..M

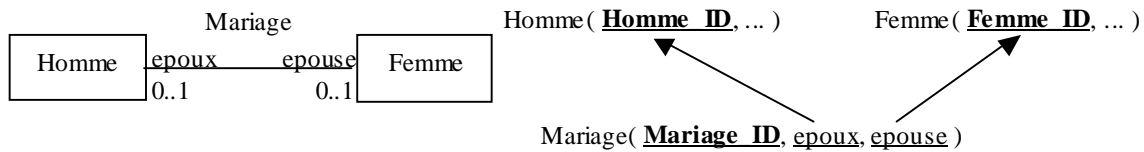
cas : 0..1 / 0..*



cas : 0..1 / 0..1



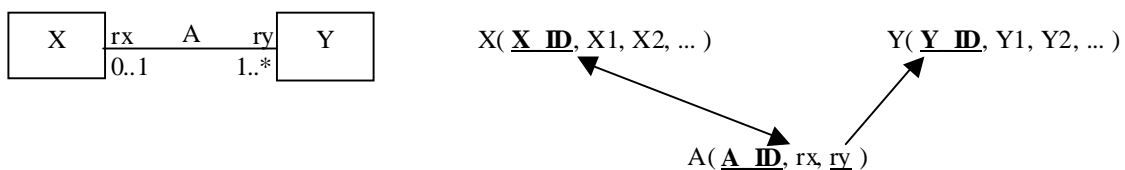
exemple (très classique) :



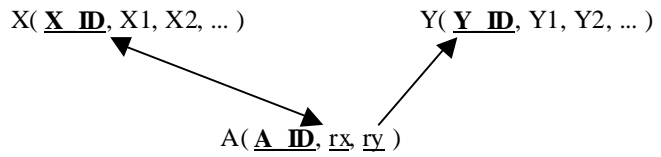
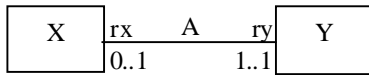
code du LDD SQL :

```
SQL> create table Homme(  Homme_ID varchar( 10 )      primary key,
...
);
SQL> create table Femme(  Femme_ID varchar( 10 )      primary key,
...
);
SQL> create table Mariage(  Mariage_ID varchar( 10 )      primary key,
epoux  varchar( 10 )      references Homme( Homme_ID )
unique not null,
epouse varchar( 10 )      references Femme( Femme_ID )
unique not null
);
```

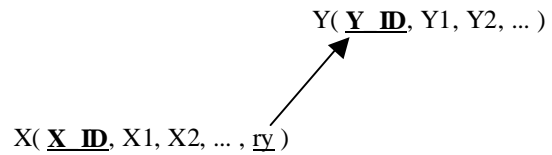
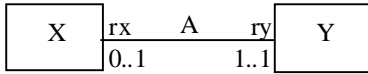
cas : 0..1 / 1..*



cas : 0..1 / 1..1

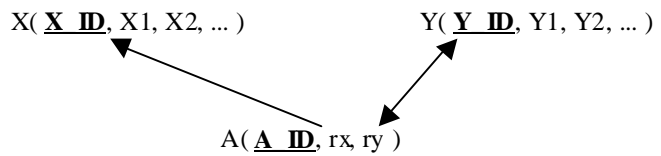
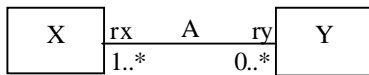


ou plus simplement :

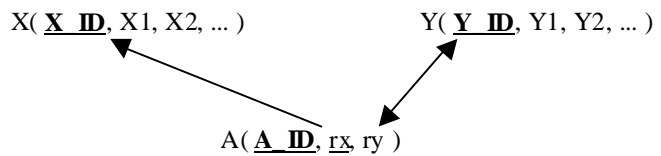
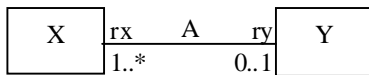


3.5.3 Multiplicités 1..* / m..M

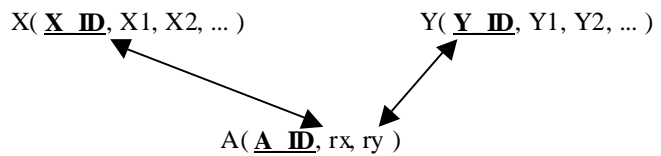
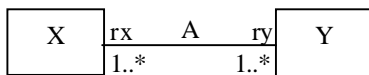
cas : 1..* / 0..*



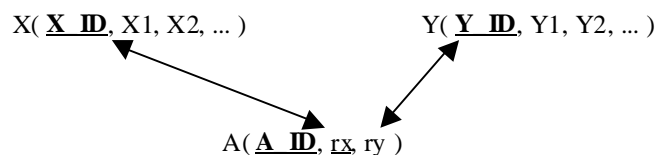
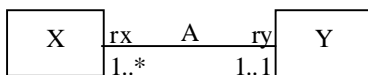
cas : 1..* / 0..1



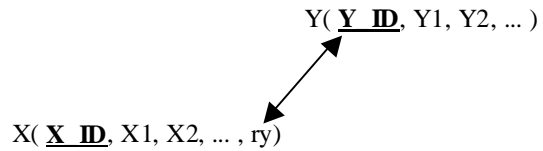
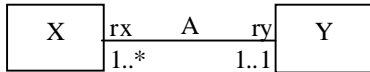
cas : 1..* / 1..*



cas : 1..* / 1..1 ...

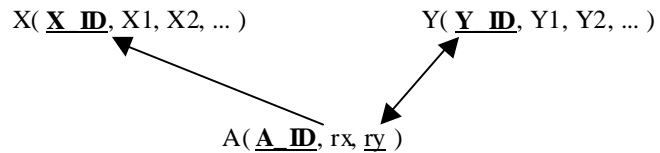
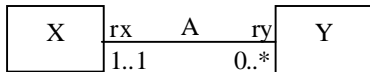


ou plus simplement :

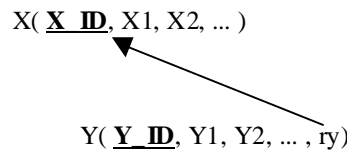
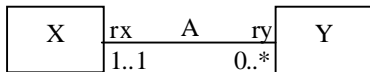


3.5.4 Multiplicités 1..1 / m..M

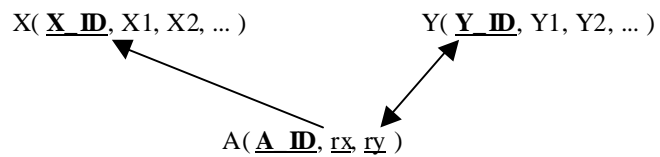
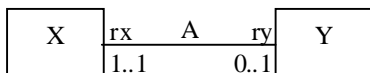
cas : 1..1 / 0..*



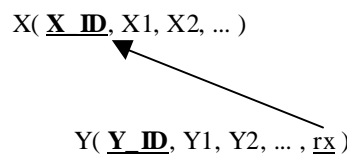
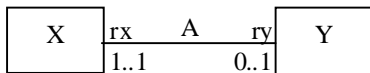
ou plus simplement :



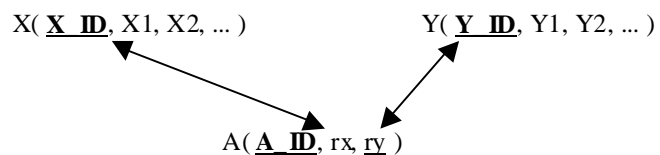
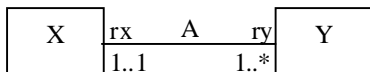
cas : 1..1 / 0..1



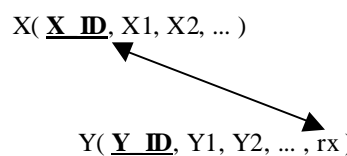
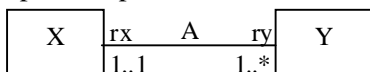
ou plus simplement :



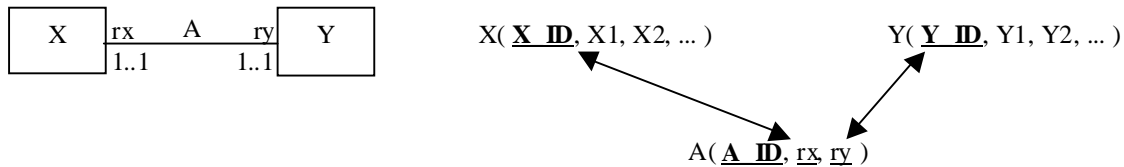
cas : 1..1 / 1..*



ou plus simplement :



cas : 1..1 / 1..1

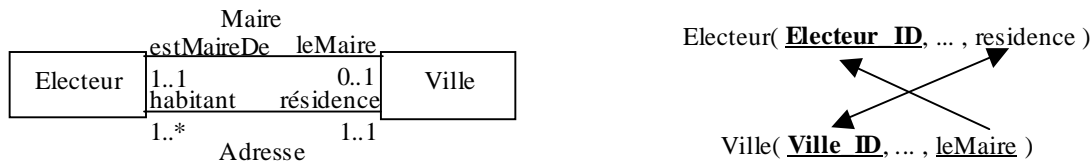


3.5.5 Circuits dans les dépendances référentielles

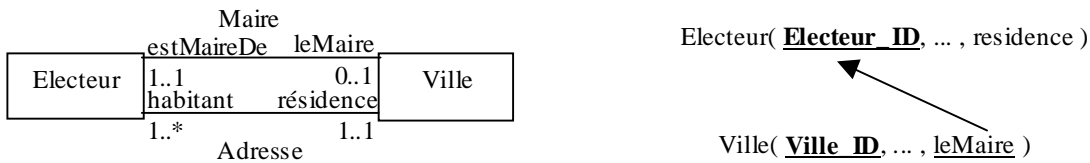
La génération du code du LDD SQL se fait à partir du schéma relationnel, ce dernier étant obtenu en appliquant pour chaque classe et pour chaque association les règles définies ci-dessus. Pour effectuer cette génération de code, on ne tient pas compte du double-sens des dépendances référentielles.

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 :



en déduire le code du LDD SQL correspondant et rajouter des triggers ou des sous-programmes pour vérifier la contrainte supprimée, à savoir : tout électeur réside dans une ville , et que toute ville a au moins un électeur qui y réside.

4 Résumé, Bilan

Dans cet enseignement, on a cherché à montrer que la modélisation des données d'une application se fait selon une démarche en trois étapes :

- le diagramme de classes, complété éventuellement par des contraintes (spécifications OCL),
- le schéma relationnel, avec ses clés primaires et secondaires, et ses dépendances référentielles,
- le code du LDD SQL, avec ses clés primaires, secondaires (unique not null) et étrangères et ses contraintes syntaxiques (référence en avant, circuit, ...) imposées pour des raisons pragmatiques de mises à jour des données et de performances des programmes d'application. A ce code, il est possible de rajouter des index de relations pour améliorer les performances, si nécessaire.

Cette démarche de conception et réalisation d'applications orientée donnée répond à plusieurs objectifs :

- séparation des préoccupations, en faisant apparaître les problèmes au fur et à mesure :
 - . définition des objets et des liens de l'application, à partir de l'expression des besoins,
 - . définition du modèle de données selon la plate-forme qui prendra en charge l'application : ici, c'est le modèle relationnel avec ses concepts définissant la structure des données et leurs implantations,
 - . déduction du code source (LDD SQL) avec ses contraintes de programmation très spécifiques ;
- à chacun, ses préoccupations :
 - . l'analyste/concepteur définit le diagramme de classes avec les responsables de l'application, à partir de l'expression des besoins,
 - . le concepteur/développeur définit le schéma relationnel à partir de diagramme de classes,
 - . l'administrateur de la base de données définit les codes du LDD SQL et les programmes d'application à partir du schéma relationnel, et rajoutant les éventuels les index pour répondre aux exigences des utilisateurs finals.
- à chacun, avoir la possibilité de suivre ce que font les autres pour comprendre les problèmes qui se posent au fur et à mesure :
 - . c'est le rôle du schéma relationnel qui fait le lien la conception et l'implantation proprement dite des données,
 - . c'est la possibilités de définir de façon dynamique les index de relations sans changer la conception des données.

Cet enseignement fait ressortir 2 aspects :

- . l'aspect technique : le modèle relationnel et les outils qui gravitent autour,
- . les aspects méthodologiques (démarches) qui montrent comment on a l'habitude d'aborder les problèmes, de les résoudre (avec les moyens du bord) et comment on peut maîtriser les aspects techniques.

Les aspects techniques sont généralement imposés dans le contexte où l'on travaille, c'est à chacun (ou c'est le rôle du responsable de l'équipe) de définir les aspects méthodologiques les plus appropriés pour mener à bien le projet dans lequel on est impliqué.