

Christian Soutou

UML 2

**pour les
bases de données**

Avec 20 exercices corrigés

EYROLLES

À Louise et Émile-Barthélémy, mes grands-parents,
à Elisabeth, ma mère,
pour Paul-Émile, mon trésor de fils.

Remerciements

Je remercie à nouveau Daniel Vielle, Didier Donsez et Dominique Nanci qui m'ont aidé en 2002 à la finalisation de la première version de cet ouvrage (*De UML à SQL*, Éditions Eyrolles). J'en profite aussi pour remercier les lecteurs qui ont soulevé des errata, Placide Fresnais, Pascal Chemin et Philippe Peuret.

Merci aussi à ceux qui m'ont donné un coup de main pour cette version fortement remaniée, Thierry Millan pour OCL, Matthieu Brucher (alias *Miles*) et Romain Gallais (alias *Nip*).

Merci enfin à mon éditeur en les personnes d'Éric Sulpice et d'Antoine Derouin d'avoir donné un second souffle à mon travail.

Table des matières

Remerciements	VI
Avant-propos	1
À qui s'adresse cet ouvrage ?	2
Ouvrages relatifs à UML et aux bases de données	2
Guide de lecture	4
Conception et normalisation	4
Programmation SQL2 et SQL3	5
Outils du marché	5
Annexes	5
Site Web	5
Conventions typographiques	5
Contact avec l'auteur	6
Introduction	7
Évolution des SGBD relationnels	7
Les niveaux d'abstraction	8
Caractéristiques des SGBD	9
Modèle de données	10
Que sont devenus les SGBD objet ?	12
Les SGBD objet-relationnels	12
Bilan	14
Du modèle entité-association à UML	15
Pourquoi faudra-t-il utiliser UML ?	15
Comment concevoir une base de données avec UML ?	16
1 Le niveau conceptuel : face à face Merise/UML	19
Généralités	20
Face à face Merise/UML	20
Concepts de base	20
Associations <i>un-à-un</i>	29
Associations <i>un-à-plusieurs</i>	31
Associations <i>plusieurs-à-plusieurs</i>	33
Associations <i>n-aires</i>	36
Associations réflexives	41

Associations dérivées et qualifiées	43
Associations navigables	44
Contraintes	45
Affinage des associations <i>n</i> -aires	55
Associations d'agrégation	60
Règles de validation	66
Caractère élémentaire d'un attribut	66
Vérification	66
Première forme normale	67
Deuxième forme normale	69
Troisième forme normale	70
Forme normale de Boyce Codd	71
Décomposition des <i>n</i> -aires	72
Héritage	75
Formalisme	75
Différents cas d'héritage	76
Héritage multiple	82
Héritage simple	83
Bilan	83
Encapsulation	83
Positionnement des méthodes	83
Visibilité des attributs et des méthodes	84
Au niveau de la base de données	85
Attributs dérivés	85
Identification et incidence sur la réification	86
Identification absolue d'une entité	86
Identification relative	87
Identification d'une association	87
Identifiant alternatif	88
Entité faible	88
Exemple récapitulatif	89
Aspects temporels	91
Modélisation d'un moment	91
Modélisation de chronologie	92
Modélisation de l'historisation	93
La démarche	93
Décomposition en propositions élémentaires	93
Propositions incomplètes	94
Propositions redondantes	94
Propositions réductibles	95
Propositions complexes irréductibles	95
Chronologie des étapes	95

Bilan	96
UML 2 ou Merise/2 ?	96
Quelques règles à respecter avec UML	97
Et après ?	98
Exercices	98
 2 Le niveau logique : du relationnel à l'objet	 103
Modèle relationnel	103
Historique, généralités	103
Modèle de données	104
Équivalences avec le modèle de données du SGBD	106
Dépendances fonctionnelles	107
Formes normales	113
Approche par décomposition	119
Approche par synthèse	122
Bilan	124
Modèles objet	124
Notation UML	124
Les concepts objet au niveau logique	125
Du conceptuel au logique	128
D'un schéma entité-association/UML vers un schéma relationnel	128
D'un schéma entité-association/UML vers un schéma objet	135
Associations d'agrégation	143
Raisonnement par rétroconception	152
Deux relations en liaison	153
Trois relations en liaison	153
Du conceptuel à l'objet	156
Transformation des entités/classes	156
Exercices	169
 3 Le niveau physique : de SQL2 à SQL3	 177
Le langage SQL	178
Les normes	178
Définition des données	179
Manipulation des données	181
Interrogation des données	182
Contrôle des données	184
Passage du logique à SQL2	189
Traduction des relations	189
Traduction des associations binaires	190
Traduction des associations d'héritage	198

Traduction des contraintes d'héritage	202
Transformation des agrégations	209
Traduction des contraintes	217
Du modèle objet à SQL3	222
Traduction des classes UML	222
Associations un-à-un	224
Associations un-à-plusieurs	225
Associations plusieurs-à-plusieurs	228
Associations <i>n</i> -aires	229
Associations réflexives	232
Classes-associations UML	234
Transformation des associations d'héritage	240
Exercices	243
 4 Outils du marché : de la théorie à la pratique	 247
Associations binaires	248
Niveau conceptuel	249
Niveau logique	251
Script SQL	252
Bilan intermédiaire	252
Associations n-aires	254
Niveau conceptuel	254
Niveau logique	255
Script SQL	256
Bilan intermédiaire	256
Classes-associations	258
Niveau conceptuel	258
Niveau logique	259
Script SQL	260
Bilan intermédiaire	260
Contraintes	262
Niveau conceptuel	262
Niveau logique	263
Script SQL	263
Bilan intermédiaire	264
Agrégations	265
Niveau conceptuel	265
Niveau logique	266
Script SQL	267
Bilan intermédiaire	267

Héritage	268
Niveau conceptuel	269
Niveau logique	270
Script SQL	272
Bilan intermédiaire	272
La rétroconception	273
Bilan général	278
Quelques mots sur les outils	278
Conclusion	299
A URL utiles	301
Outils	301
UML	301
B Bibliographie	303
Index	309

Avant-propos

Dans cet avant-propos et dans l'introduction, j'expliquerai pourquoi il convient d'utiliser à présent UML (Unified Modeling Language) pour concevoir une base de données relationnelle de type SQL2 ou objet-relationnelle de type SQL3. Dans les chapitres suivants, j'exposerai les moyens à mettre en œuvre, étape par étape, pour arriver à définir un script SQL à partir d'une spécification UML 2 sous la forme d'un diagramme de classes.

Rappel

« Une base de données est un ensemble de données évolutives, organisé pour être utilisé par des programmes multiples, eux-mêmes évolutifs. » (Le Petit Larousse)

Depuis plus de 30 ans, la conception des bases de données est réalisée à l'aide du modèle entité-association. Ce modèle a fait ses preuves et la plupart des outils informatiques de conception (destinés aux concepteurs français) l'utilisent encore aujourd'hui. La notation UML s'est imposée depuis quelques années pour la modélisation et le développement d'applications écrites dans un langage objet (C++ et Java principalement). Cette notation n'a pas été initialement pensée pour les bases de données mais elle permet d'offrir un même formalisme aux concepteurs d'objets métiers et aux concepteurs de bases de données. Le marché a suivi cette tendance car, aujourd'hui, tous les outils utilisent cette notation.

Personnellement je considère, et je l'expliquerai, que le diagramme de classes, avec ses caractéristiques, convient bien à la modélisation d'une base de données (relationnelle ou objet-relationnelle). En effet, on retrouve tous les concepts initiaux tout en découvrant de nouvelles possibilités qui, si elles sont employées à bon escient, n'entravent en rien la normalisation des schémas SQL dérivés. Lors du face à face entre le modèle entité-association et le diagramme de classes UML 2 et du rappel des règles de dérivation du conceptuel vers SQL, il n'y aura qu'un pas à franchir pour passer de UML 2 à SQL.

Bien qu'il existe depuis quelques années des outils informatiques permettant de générer des scripts SQL à partir d'un schéma conceptuel graphique, il est courant de constater que ces mêmes scripts (ou les modèles logiques de données), doivent être modifiés manuellement par la suite, soit pour des raisons d'optimisation, soit parce que l'outil ne permet pas de générer une caractéristique particulière du SGBD (index, vues, type de données...), soit tout simplement parce que le concepteur préfère utiliser une autre possibilité d'implémentation pour traduire telle ou telle autre association.

Il semble donc préférable de maîtriser des concepts et une démarche plutôt que de connaître les caractéristiques d'un outil en particulier. Cela n'empêchera pas, bien au contraire, d'utiliser l'outil de manière optimale. C'est pour cela que cet ouvrage détaille d'une part comment construire un modèle conceptuel sous la forme d'un diagramme de classes, et d'autre part énonce des règles précises de transformation entre les différents niveaux d'abstraction qui interviennent dans la conception d'une base de données. Cette démarche pourra ainsi servir de base théorique à l'utilisation des différents outils du marché.

À qui s'adresse cet ouvrage ?

Cet ouvrage s'adresse aux personnes qui s'intéressent à la modélisation et à la conception des bases de données.

- Les concepteurs habitués au modèle entité-association (que ce soit la notation américaine ou celle de type Merise/2) y trouveront les moyens de migrer vers le diagramme de classes de UML 2.
- Les concepteurs UML repéreront des règles de passage afin de traduire un diagramme de classes dans un modèle de données d'une base de données relationnelle ou objet-relationnelle.
- Les programmeurs connaissant le modèle relationnel et SQL2 découvriront l'influence de l'approche objet sur les bases de données, et les mécanismes de programmation mettant en œuvre les types abstraits de données avec SQL3.
- Les étudiants dénicheront des définitions pragmatiques et de nombreux exercices mettant en jeu tous les niveaux du processus de conception d'une base de données.

Ouvrages relatifs à UML et aux bases de données

Lors de la sortie de la première version de cet ouvrage (juin 2002), il n'existait que peu d'ouvrages relatifs à l'utilisation d'UML pour la conception de bases de données.

- *Oracle8 Design Using UML Object Modeling* [DOR 99], axé sur une implémentation sous Oracle, couvre la traduction de toutes les catégories d'associations UML en SQL2 et avec les caractéristiques objet-relationnelles d'Oracle pour certaines. Il n'est pas organisé autour des niveaux d'abstraction comme ce présent ouvrage. La connaissance d'UML est un prérequis pour cet ouvrage qui est assez austère (*Oracle Press* oblige...) et pas très pédagogique. De plus, pour chaque type d'association, une seule solution d'implémentation est donnée.
- *Database Design for Smarties* [MUL 99] comporte un chapitre sur le diagramme de classes et des règles de passage aux modèles relationnel et objet-relationnel. Très verbeux,

manquant d'exemples et de précisions à propos des associations n -aires, il était toutefois assez complet.

- *UML for database design* [NAI 01], écrit par deux pointures de la société Rational Software (société rachetée par IBM en 2001), est basé sur une étude de cas fictive de la gestion d'une clinique de rééducation (bonjour la joie...). Il s'agit de l'analyse de l'existant à la définition des classes UML et des tables représentées à l'aide du profil UML pour les bases de données créées par Rational (un profil est une proposition d'une communauté et regroupe un ensemble d'éléments UML tels que des composants, stéréotypes, icônes, propriétés, etc. qui s'appliquent à un contexte particulier et qui conservent le métamodèle d'UML intact). Nous détaillerons au chapitre 5 le profil UML pour la conception de bases de données. Il n'était pas question de SQL dans *UML for database design* qui s'axe plutôt vers la représentation de la dynamique d'un système avec de nombreux diagrammes d'activités et de cas d'utilisation. En effet, seules quatre pages sur près de trois cents sont consacrées au passage d'un modèle de classes à un modèle de données relationnel. La connaissance d'UML et des principes des conceptions d'une base de données sont un prérequis pour cet ouvrage.
- *Information Modeling and Relational Database* [HAL 01] consacre un chapitre entier (50 pages sur 760) au comparatif de la notation UML et du modèle qu'il préconise dans son ouvrage : ORM (Object Role Model).

Citons aussi ceux qui passaient sous silence cette notation : *Conception de bases de données* [STE 01], *The Data Model Resource Book* [SIL 01] et *Conception des bases de données relationnelles* [AKO 01] basé sur le modèle entité-relation étendu.

Cinq ans après la sortie de la première version de cet ouvrage, pas grand-chose de vraiment nouveau dans la littérature actuelle, si ce n'est le fait de citer le diagramme de classes de UML soit en quelques lignes, soit en quelques pages, soit, exceptionnellement, faisant l'objet seul d'un chapitre :

- *Bases de données et modèles de calculs* [HAI 04] présente en 6 pages (sur 435) le diagramme de classes sans préconiser de solution d'implémentation.
- *Systèmes de bases de données* [CON 05] se sert de la notation UML (16 pages sur 1412) uniquement pour présenter le concept de spécialisation/généralisation dans le cadre de la modélisation entité-association étendue.
- *Conception et architecture de bases de données* [ELM 04] ne consacre que 7 pages (sur 728) à l'outil de Rational Rose en ne le présentant qu'au niveau logique (notation relationnelle).
- *Data Modeling Essentials* [SIM 05] traite de UML en seulement 7 pages également (sur 532), on y apprend principalement à éviter les associations qualifiées (je n'en parle pas non plus dans cet ouvrage).
- *Création de bases de données* [LAR 05] ne consacre que 3 pages (sur 190) au diagramme de classes sans décrire précisément ses possibilités.

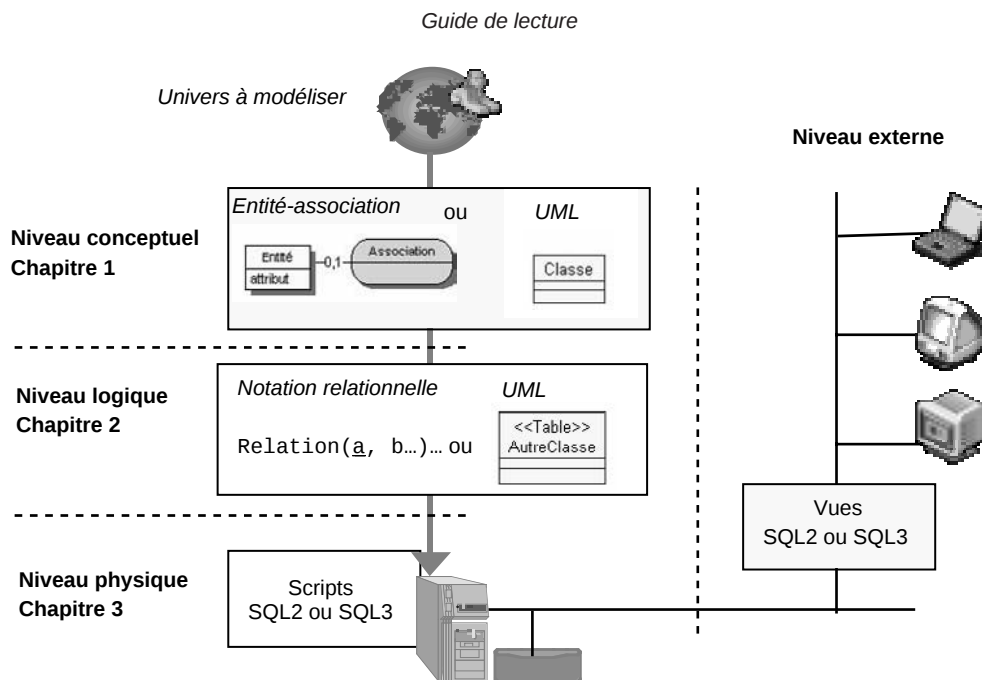
Aucun de ces ouvrages n'étudie en détail les possibilités conceptuelles offertes par UML ni l'adéquation de UML avec les outils de modélisation du marché.

Guide de lecture

Cet ouvrage s'organise en quatre chapitres. L'introduction développe cet avant-propos. Les chapitres 1 et 2 traitent de modélisation et de normalisation avec UML 2. Le chapitre 3 est consacré à l'implémentation sous SQL2 et SQL3. Le chapitre 4 valide ma démarche en la comparant aux solutions des outils du marché.

La figure suivante illustre les différents chapitres par rapport aux niveaux d'abstraction d'un processus de conception. Elle peut ainsi servir de guide schématique à la lecture de ce livre (merci à <http://www.iconarchive.com>).

Le niveau externe qui correspond à la définition de vues sort quelque peu du cadre de cet ouvrage, le lecteur trouvera aisément des ressources à propos de ce sujet (*SQL pour Oracle* aux éditions Eyrolles pour les vues SQL2, *Programmer objet avec Oracle* aux éditions Vuibert pour les vues SQL3).



Conception et normalisation

Le chapitre 1 décrit la première étape du processus de conception d'une base de données, à savoir la construction d'un schéma conceptuel normalisé. Nous réalisons un face à face entre le modèle entité-association et le diagramme de classes UML.

Le chapitre 2 expose les règles de dérivation d'un modèle conceptuel dans le modèle de données relationnel (ou objet-relationnel). Nous rappelons aussi les principes de normalisation, qui permettent de préparer correctement un diagramme de classes à sa traduction en langage SQL.

Programmation SQL2 et SQL3

Le chapitre 3 détaille la traduction d'un modèle de données relationnel en script SQL2 et celle d'un modèle objet-relationnel en langage SQL3. Nous décrivons également les moyens de programmer les différentes contraintes d'un schéma conceptuel.

Outils du marché

Le chapitre 4 valide la démarche théorique de l'ouvrage en la comparant aux offres des principaux outils informatiques du marché. L'étude comparative confronte 14 logiciels (Enterprise Architect, MagicDraw, MEGA Designer, ModelSphere, MyEclipse, Objecteering, Poseidon, PowerAMC, Rational Rose Data Modeler, Together, Visio, Visual Paradigm, Visual UML et Win'Design). Chaque outil est évalué selon la qualité dont il dispose à implémenter différents critères de UML 2 (associations binaires, *n*-aires, classes-associations, agrégations, contraintes inter-associations, héritage multiple avec contrainte et rétroconception d'une base de données).

Annexes

Les annexes contiennent une webographie et une bibliographie. L'index recense les termes utilisés dans la définition des concepts et instructions SQL, ce qui permettra au lecteur de faire rapidement le parallèle entre les concepts et la programmation.

Site Web

Les corrigés détaillés des exercices et les errata sont mis en ligne sur <http://www.soutou.net/christian/livres/UML2BD/Complements.html>.

Conventions typographiques

La police *courrier* est utilisée pour souligner les instructions SQL, noms de variables, tables, contraintes, etc. (ex : *SELECT nom FROM Pilote*). De plus, nous employons les majuscules pour les directives SQL et les minuscules pour les autres éléments. Le nom des tables, comme celui des classes, est précédé d'une majuscule. Nous utilisons aussi cette convention pour les noms et composants des classes UML (ex : la classe *Compagnie* dispose de la méthode *affreter*).



Ce sigle introduit une définition, un concept ou une remarque importante. Il apparaît dans la partie théorique de l'ouvrage, mais aussi dans la partie technique pour souligner soit des instructions essentielles, soit la marche à suivre avec SQL.



Ce sigle annonce soit une impossibilité de mise en œuvre d'un concept, soit une mise en garde. Il est principalement utilisé dans la partie consacrée à SQL.



Ce sigle indique une astuce ou un conseil personnel.

Contact avec l'auteur

Si vous avez des remarques à formuler sur le contenu de cet ouvrage, n'hésitez pas à m'écrire à l'adresse soutou@iut-blagnac.fr. Je vous souhaite une bonne lecture.

Introduction

Bougez pas !... Les mains sur la table ! Je vous préviens qu'on a la puissance de feu d'un croiseur et des flingues de concours.

*Les Tontons flingueurs, B. Blier
G. Lautner, dialogues M. Audiard, 1963*

Dans cette introduction, nous verrons pourquoi il est encore nécessaire de travailler avec des systèmes de gestion de bases de données relationnels et comment ces systèmes ont évolué pour inclure peu à peu certains concepts de l'approche objet. Nous expliquerons aussi pourquoi le diagramme de classes d'UML s'est imposé en tant que notation, comme support du modèle conceptuel, en remplacement du modèle entité-association.

Évolution des SGBD relationnels

On pourrait croire que le modèle relationnel a maintenant vécu tant il semble inadapté à la gestion d'informations de plus en plus complexes. Élaboré en 1969 par E.F. Codd [COD 69], chercheur chez IBM, ce modèle est à l'origine des premiers SGBD relationnels, devenus des systèmes incontournables.

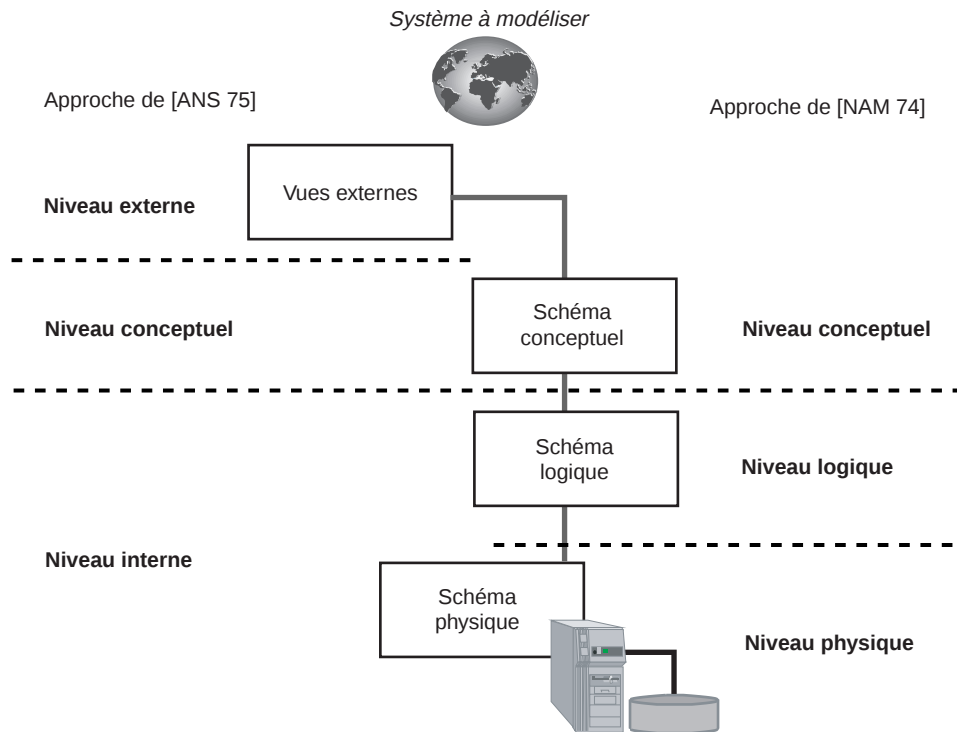
Certains éditeurs comme IBM ou Oracle ayant près de 30 ans d'expérience, leurs produits sont assurément d'une grande fiabilité. Standardisé, SQL, le langage utilisé, peut s'interfacer avec des langages de troisième génération (C, Cobol...), mais aussi avec des langages plus évolués (comme C++ ou Java).

Ces systèmes intègrent des outils de développement comme les précompilateurs, les générateurs de code, d'états ou de formulaires. Enfin, ces systèmes répondent bien à des architectures de type client-serveur et Intranet ou Internet. Ces architectures présentent une interface utilisateur (le plus souvent graphique), fonctionnent grâce à des applications dont une partie s'exécute sur le client et l'autre sur le serveur, et manipulent des données. L'adoption généralisée du client-serveur repose sur le besoin croissant de services réclamés par la partie cliente. Les grands éditeurs de progiciels profitent ainsi de l'occasion pour se concentrer, côté serveur, sur le cœur des applicatifs et sur l'optimisation des moteurs de bases de données (aspects transactionnels, montée en charge, équilibrage). Il reste donc à programmer, du côté client, les aspects de présentation des données et d'interfaces graphiques.

Les niveaux d'abstraction

Trois niveaux d'abstraction (conceptuel, logique et physique) ont été définis en 1974 pour la conception d'une base de données [NAM 74]. Un découpage légèrement différent a ensuite été proposé par l'ANSI, en 1975 [ANS 75], qui fait désormais référence en la matière. Ce dernier décrit un niveau externe, un niveau conceptuel et un niveau interne. Nombre de méthodes de conception ont vu le jour et ont associé une forme de représentation appelée « schéma » à chacun de ces niveaux. Un niveau logique est présent dans certaines de ces méthodes.

Figure I-1 Niveaux d'abstraction



L'approche du rapport de Namur

Le niveau conceptuel spécifie les règles de gestion en faisant abstraction de toute contrainte de nature organisationnelle [MOR 92]. Le niveau logique spécifie des choix de type organisationnel (contraintes liées aux acteurs et types de matériels qui seront utilisés pour les traitements). Le niveau physique spécifie des choix techniques (moyens mis en œuvre pour gérer les données ou activer les traitements), les optimisations étant également prises en compte.

L'approche du rapport de l'ANSI

Le niveau interne est le niveau relatif à la mémoire physique. Il s'agit du niveau où les données sont réellement enregistrées. Le niveau externe est le niveau relatif aux utilisateurs. Il s'agit du niveau dans lequel les utilisateurs voient les données. Le niveau conceptuel, aussi appelé « niveau logique communautaire », est le niveau intermédiaire entre les deux précédents [DAT 00].

L'approche du livre

L'approche décrite dans ce livre est plus pragmatique. Elle correspond à la réalité que côtoient tous les jours concepteurs, développeurs, administrateurs et utilisateurs. Chaque niveau fait l'objet d'un chapitre.

Le niveau conceptuel décrit une représentation abstraite de la base de données à l'aide de diagrammes entité-association ou UML. Le niveau logique détaille une représentation intermédiaire entre le niveau conceptuel et le niveau physique. Les diagrammes logiques (qu'on appelle aussi « diagrammes du modèle de données ») peuvent être exprimés soit à l'aide d'une notation mathématique, soit à l'aide d'un diagramme de classes UML (les classes auront le stéréotype particulier <<Table>> et le diagramme sera quelque peu différent de celui du niveau conceptuel). Quant au niveau physique, il concerne les structures de données qui seront à mettre en œuvre dans la base de données relationnelle ou objet-relationnelle. Le schéma physique traduit, à l'aide du langage SQL2 ou SQL3, le schéma logique. Le niveau externe inclut des sous-schémas qui assurent la sécurité et la confidentialité de la base de données. Un schéma externe sera écrit à l'aide du langage SQL2 ou SQL3 selon qu'il s'agit d'une base de données relationnelle ou objet-relationnelle.

Notons que même pour les bases de données objet ou objet-relationnelles, les niveaux de conception sont utiles, car ils permettent de séparer les spécifications formelles de l'implémentation et rendent les schémas indépendants des matériels et des logiciels, dans la mesure du possible.

Caractéristiques des SGBD

Les objectifs et fonctions des SGBD relationnels sont les suivants :

- Centraliser l'information pour éviter les redondances, garantir l'unicité des saisies et la centralisation des contrôles.
- Faciliter l'utilisation à des utilisateurs pas forcément informaticiens. La base de données doit pouvoir être accessible par des interfaces intuitives et par des langages déclaratifs (un langage est dit « déclaratif » lorsqu'il permet de décrire ce que l'on souhaite obtenir sans décrire les moyens de l'obtenir), tel SQL qu'on appelle « langage de requêtes », et non pas procéduraux (un langage est dit « procédural » lorsqu'il impose de décrire toutes les actions nécessaires, que l'on regroupe dans des fonctions ou des procédures).

- Assurer l'indépendance données/traitements de manière à avoir le moins possible à modifier les programmes si la structure de la base évolue.
- Description de l'information, ce qui inclut la gestion de l'espace disque, la structure des données stockées et le dictionnaire des données.
- Partage de l'information entre différents utilisateurs. Assurer cette fonction entraîne, pour le système, la mise en place d'un grand nombre de techniques qui découlent de la gestion des accès concurrents (mise en place de verrous, programmation de transactions et résolution des éventuels interblocages) et de la confidentialité (tous les utilisateurs n'auront pas les mêmes prérogatives).
- Préservation de la cohérence des données dans le temps. Cet aspect des choses inclut notamment la programmation des règles de gestion du système d'information du côté de la base et non pas dans les programmes d'application (par exemple, il faudra vérifier, avant chaque vol, que les pilotes sont à jour de leurs licences pour le type d'avion utilisé). La fonction de cohérence inclut aussi tous les aspects relatifs à la sécurité en cas de panne matérielle ou logicielle. Enfin, l'archivage est assuré par le SGBD (*backup*) de même que la technique de restauration (*recovery*).

Modèle de données

Bien que le modèle de données relationnel repose sur des concepts simples (relations, clés et dépendances fonctionnelles pour les principales), il permet néanmoins de modéliser artificiellement des données complexes. Le modèle relationnel est fondé sur de solides bases théoriques, car il propose des opérateurs issus de la théorie des ensembles. De plus, on peut appliquer des techniques de normalisation. La programmation de ces concepts est assurée par le langage SQL normalisé par l'ISO depuis 1986.

Les liens entre les relations, qui sont des tables au niveau de la base de données, ne font plus intervenir de chaînages physiques comme le faisaient les SGBD précédents (hiérarchiques et réseaux), mais des pointeurs logiques fondés sur des valeurs contenues dans les colonnes. Nous verrons que les liens sont réalisés par les clés primaires (*primary keys*) et par les clés étrangères (*foreign keys*). Pour cette raison, le modèle relationnel est dit « modèle à valeurs ».

Structure des données

La figure suivante décrit le contenu d'une base de données relationnelle, qui représente le fait que des pilotes puissent travailler pour le compte de différentes compagnies. Nous détaillerons ce modèle de données au chapitre 2.

La contrainte référentielle est vérifiée si chaque valeur contenue dans une clé étrangère se retrouve en tant que clé primaire d'une autre table. La majorité des SGBD du marché prennent en charge automatiquement cette contrainte, très utile pour gérer la cohérence entre tables.

Figure I-2 Structure et contenu des tables relationnelles

Compagnie[ncomp, nomcomp]Remuneration[brevet#, ncomp#, paye]Pilote[brevet, nom, adresse]

Clé primaire en gras
Clé étrangère suivie d'un #

Compagnie

ncomp	nomcomp
1	Air-France
2	Quantas

Remuneration

brevet#	ncomp#	paye
3MPY93	1	12830
16LDG01	1	12000
30MPY01	1	18500
25MPY00	2	14700
16LDG01	2	8000

Pilote

brevet	nom	adresse
3MPY93	Soutou	Castanet
16LDG01	Bidal	Paris
30MPY01	Lamothe	Toulouse
25MPY00	Albaric	Ramonville

Limitations

La simplicité du modèle de données induit les limitations suivantes :

- La faiblesse du langage SQL au niveau de la programmation entraîne l'interfaçage d'instructions SQL avec un langage procédural plus évolué (C, COBOL...) ou objet (C++, Java...) pour répondre à des spécifications complexes. On parle de défaut d'impédance (*impedance mismatch* : terme désignant les problèmes de cohabitation entre un langage de programmation avec sa syntaxe et ses règles et une base de données SQL. Il faut en effet introduire dans le programme des directives qui n'appartiennent pas à sa syntaxe initiale).
- La normalisation induit un accroissement du nombre de tables. Ainsi, si deux objets doivent être liés en mémoire, il faut simuler ce lien au niveau de la base par un mécanisme de clés étrangères ou de tables de corrélations. Parcourir un lien implique souvent une jointure dans la base (mise en relation de plusieurs tables deux par deux, fondée sur la comparaison de valeurs des colonnes comparées). Il peut en résulter un problème de performance dès qu'on manipule des données volumineuses.
- Puisque seules les structures de données tabulaires sont permises, il est difficile de représenter directement des objets complexes. En revanche, les SGBD relationnels prennent maintenant en compte la gestion de données multimédias (fichiers binaires stockés hors de la base ou dans la base).

Ces limitations ont amené le développement de SGBD objet et objet-relationnels.

Que sont devenus les SGBD objet ?

Pendant un temps, les défenseurs de l'objet ont cru que les SGBD du même nom pourraient supplanter, voire remplacer, les systèmes relationnels. Le marché leur a donné tort. Les systèmes purement objet semblent cantonnés à des applications manipulant des données non structurées avec des programmes écrits dans des langages objet. Ils concernent ainsi un segment très limité du marché des SGBD.

Historique

Le premier SGBD objet a été Gemstone [COP 84], extension du langage objet Smalltalk. Certains produits commerciaux existent, citons Java Data Objects de GemStone, FastObjects de Poet Software, Ontos, Objectstore d'Object Design, Objectivity, Versant.

Les SGBD objet n'ont pas bénéficié de l'environnement d'exploitation performant auquel les SGBD relationnels ont habitué les responsables informatique : requêtes efficaces, volume important d'informations à stocker, fiabilité, sauvegarde et restauration, performances transactionnelles OLTP (On Line Transaction Processing), outils complémentaires, etc.

Les éditeurs de SGBD objet n'ont pas eu le succès qu'ils attendaient pour la bonne et simple raison que l'existant des données des entreprises est toujours sous la forme relationnelle et qu'aucun principe formel de migration n'a été et ne sera probablement jamais établi.

Structure des données

Alors que le modèle relationnel manipule des informations sous forme tabulaire, l'une des principales extensions du modèle de données objet (reprise par le modèle objet-relationnel) consiste à manipuler des structures de données complexes incluant des pointeurs et des tables imbriquées (collections).

Les pointeurs facilitent la fonction de navigation dans le langage de requêtes en réduisant considérablement le nombre de jointures. Les tables imbriquées permettent de s'affranchir de la règle de la première forme normale, à savoir qu'un attribut peut être composé d'une liste de valeurs. Le modèle de données est dit « NF2 » (Non First Normal Form) [MAK 77]. Les liens entre objets se réalisent à l'aide d'OID (Object Identifier), qui sont des pointeurs physiques. Certains détracteurs prétendent qu'avec ce modèle de données, on est revenu près de trente ans en arrière au bon vieux temps des SGBD hiérarchiques.

Les SGBD objet-relationnels

Afin de rester en compétition avec les solutions voisines, les éditeurs de SGBD relationnels ont, dans un premier temps, offert la possibilité de stocker des informations non structurées dans des champs appelés « BLOB » (Binary Large Object). Dans un second temps, ils ont étendu le modèle relationnel à un certain nombre de concepts objet. Ce nouveau modèle hybride a été appelé « objet-relationnel » (plus rarement relationnel-objet).

Historique

La technologie objet-relationnelle est apparue en 1992 avec le SGBD UNISQL, combinant un moteur relationnel et un système objet. La sortie de ce SGBD a été rapidement suivie par celle du SGBD Open ODB, devenu Oadapter. En 1993, la société Montage Systems (devenue Illustra) achète la première version commerciale du système Postgres.

Informix a été le premier des grands éditeurs à relever le défi de l'objet-relationnel en intégrant Illustra à son moteur en 1996. Conscients de la nécessité d'enrichir le modèle relationnel, IBM et Oracle ont suivi Informix dans cette voie, l'année suivante. Tout en préservant la totalité des fonctions qu'ils assuraient auparavant, les grands éditeurs de SGBD relationnels s'orientent vers deux grands mouvements technologiques :

- l'évolution interne du moteur de leur SGBD ;
- la promotion des couches réseau (*middleware*), lesquelles permettent d'interconnecter des applications à des SGBD hétérogènes.

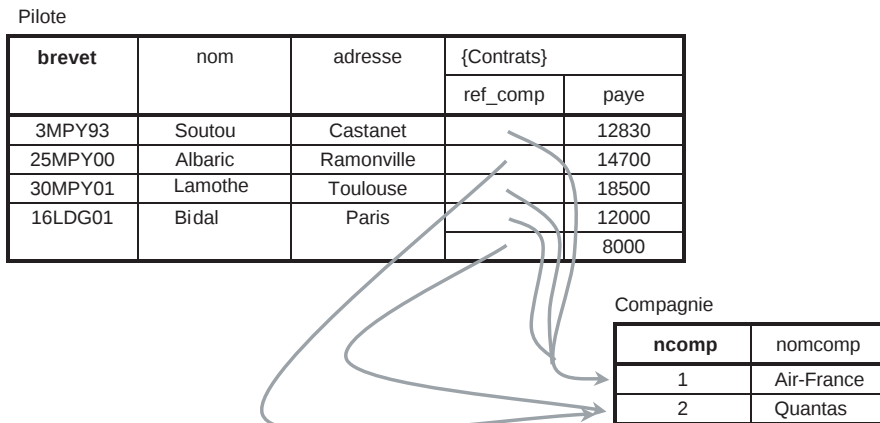
Selon M. Stonebraker [STO 96], un SGBD objet-relationnel doit prendre en compte les mécanismes d'extension de types de données, l'héritage et les systèmes de règles programmées par des déclencheurs (*triggers*). Il faudrait ajouter, à mon sens, la possibilité de programmer des méthodes s'appliquant sur des données persistantes.

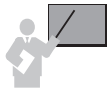
Structure des données

Le modèle de données objet-relationnel étend principalement le modèle relationnel à l'utilisation de pointeurs, de collections et de méthodes au niveau des tables et des types. La règle de la première forme normale n'a plus lieu d'exister.

Dans l'exemple suivant, il apparaît que la table **Pilote** contient la colonne **Contrats** de type collection composée d'un pointeur **ref_comp** et d'un nombre (**paye**). Nous détaillerons dans les chapitres suivants les moyens de concevoir une telle base tout en préservant la normalisation.

Figure I-3 Structure et contenu des tables objet-relationnelles





Chaque enregistrement d'une table objet-relationnelle sera considéré comme un objet sur lequel on pourra exécuter des méthodes définies au même niveau que sa structure.

Dans cette conception, l'accès aux données a été privilégié *via* les pilotes. Cette solution évite toute redondance dans la base, mais pénalise l'accès aux données par les compagnies (la question « Quels sont les pilotes de la compagnie Air-France ? » entraînera le parcours de tous les pilotes).



Les vues SQL3 des tables SQL2 peuvent simuler cette conception [SOU 04]. Une vue pourra privilégier l'accès tantôt par une table, tantôt par une autre.

Bilan

Les SGBD objet et objet-relationnels trouvent leur origine dans les langages de programmation objet. Il s'agit pour eux de ne permettre la manipulation des données qu'en utilisant une méthode. L'objectif principal de l'approche objet est d'augmenter le niveau d'abstraction. La technologie objet vise à réduire les différences entre le langage de programmation et la base de données d'une part, et entre le monde à modéliser et le modèle de données d'autre part. Le concept de l'objet induit ainsi une certaine idée de complémentarité entre les applications qui manipulent des objets différents (métier, d'interface, de connexion réseau, etc.) et les données stockées dans des SGBD. L'objet intervient dans la spécification, la programmation et l'accès aux données.

Néanmoins, bien que SGBD et langages de programmation aient des points communs, ils diffèrent par un aspect fondamental. En effet, un programme est censé résoudre un problème donné, alors qu'une base de données a pour objectif de répondre à un ensemble de problèmes qui sont en partie inconnus au moment de la création de la base. Ainsi, l'intégration de nombreux services dans les objets d'une base de données nécessite au préalable l'identification exhaustive de ces mêmes services, au moins à un niveau de généralité suffisant pour pouvoir les dériver ultérieurement.

L'approche objet-relationnelle est innovante, mais n'est pas encore utilisée couramment dans les nouvelles applications, car elle est peut-être trop récente et les concepts objet sont plus intéressants dans la programmation elle-même qu'au niveau des structures de données à stocker. Le modèle relationnel a mis plus de 15 ans à s'imposer, laissons donc du temps à cette approche prometteuse pour qu'elle s'affirme. En effet, l'approche objet-relationnelle ne repose pas sur de nouveaux concepts, mais préserve tout ce qui a fait le succès des systèmes relationnels en y ajoutant des extensions.

Par défaut, la tendance s'orienterait ces dernières années vers des techniques de *mapping* (transformations automatiques de structures de données objet côté client en enregistrements de tables relationnelles côté serveur). Bien qu'il existe des tentatives de normalisation venant

de la part de communautés comme SDO (Service Data Object), JDO (Java Data Object) et JPA (Java Persistence API), il y a fort à penser que beaucoup d'approches seront « hors norme ». En effet, chaque éditeur ou communauté voudra surtout mettre en avant son environnement (.Net avec Microsoft, Java pour Sun, etc.).

Concernant la conception, le bon sens me fait penser qu'il est nécessaire de préserver certains aspects qui ont fait la force des systèmes relationnels (normalisation et indépendance données/traitements) tout en mettant en œuvre des avantages indéniables qu'offre l'approche objet (modularité, réutilisation et encapsulation). C'est l'idée générale dans cet ouvrage que nous développerons.

Du modèle entité-association à UML

Comme nous l'avons évoqué dans l'avant-propos, le modèle entité-association a près de 30 ans d'existence. Il a fait ses preuves puisque tous les outils de conception l'ont employé jusqu'à récemment, certains produits continuent à l'utiliser en parallèle à UML. L'adoption généralisée de la notation UML dépasse le simple effet de mode.

Pourquoi faudra-t-il utiliser UML ?

Tout simplement parce que la majorité des nouveaux projets industriels utilisent la notation UML. Pour convaincre les récalcitrants, je pourrais citer le nom des entreprises appartenant au consortium ayant mis en place UML : DEC, HP, IBM, Microsoft, Oracle et Unisys pour parler des plus connues. Tous les cursus universitaires informatique, qu'ils soient théoriques ou plus techniques, incluent l'étude d'UML. Je pourrais enfin évoquer le nombre d'ouvrages et d'articles parus sur le sujet... Cela ne signifie pas qu'UML soit la panacée, mais que cette notation est devenue incontournable. La dernière version de la spécification UML, sortie en 2006, est la 2.1 (la précédente était la 2.0 en 2003).

Ce succès s'explique par la réussite de la normalisation des concepts objet, qui ont des avantages indéniables au niveau des applications informatiques. Ses principaux avantages sont la réutilisabilité de composants logiciels, la facilité de maintenance, de prototypage et d'extension des applications. Il aura fallu ainsi près de 30 ans (depuis 1966 avec les deux Norvégiens Dahl et Nygaard et leur langage objet SIMULA) pour que l'approche objet devienne incontournable.

Alors que UML 1.x définit neuf diagrammes : cinq pour les aspects statiques (classes, objets, cas d'utilisation, composants et déploiement) et quatre pour les aspects dynamiques (séquence, collaboration, états-transition, activités), UML 2 ajoute ceux d'interaction, de structure composite et le *timing diagram*. Ce livre ne s'intéressera qu'à celui convenant à la conception d'une base de données, à savoir le diagramme de classes, qui fait partie de l'aspect statique d'UML.

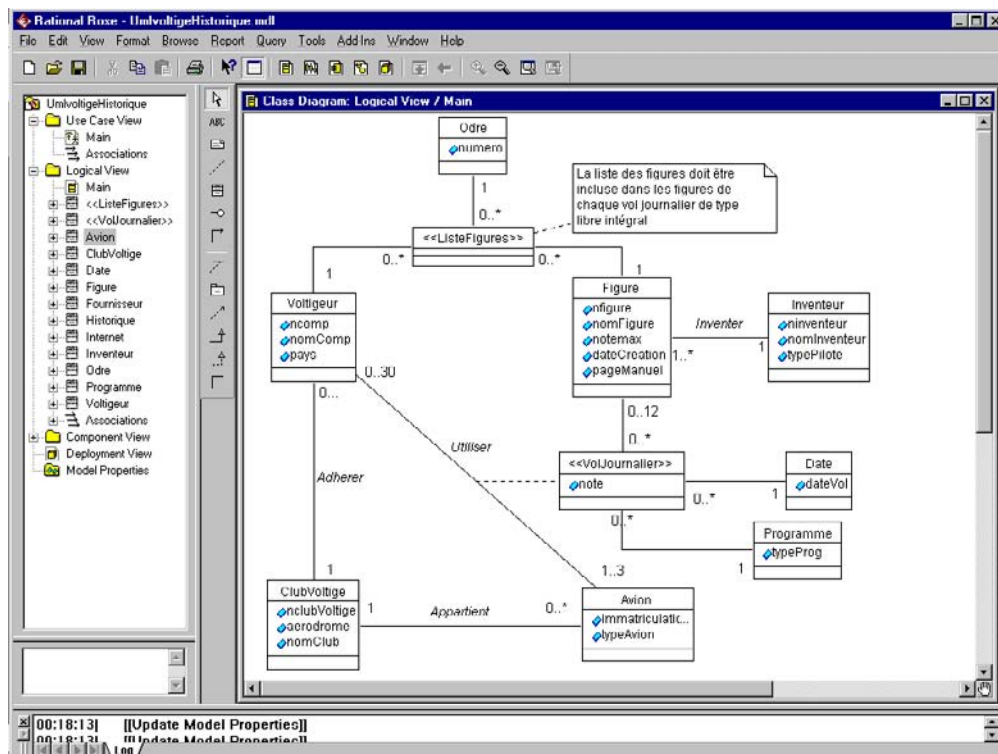
Comment concevoir une base de données avec UML ?

Bien que la notation UML ait été proposée tout d'abord pour la spécification d'applications, il n'en reste pas moins que les concepts relatifs au diagramme de classes peuvent s'adapter à la modélisation d'une base de données relationnelle ou objet-relationnelle.

UML 2 reprend les concepts du modèle entité-association et propose en plus des artifices pour améliorer la sémantique d'un schéma conceptuel (contraintes, classe-association, stéréotype...). De ce fait, cette notation est très complète et puissante et peut s'adapter parfaitement à la description statique d'une base de données.

Le problème qui se pose aux concepteurs d'applications sous UML est le pourquoi et le comment de la transcription d'un diagramme de classes, comme celui qui figure ci-dessous, dans le langage d'une base de données SQL2 ou SQL3. Ce livre devrait permettre de répondre à ces questions.

Figure I-4 Copie d'écran de Rational Rose



Nous verrons dans le chapitre 4 que tous les outils permettent de générer un script SQL2 à partir d'un diagramme de classes. Savoir se servir d'un outil, c'est bien. Mais il est primordial de bien maîtriser le cheminement ayant abouti à un script pour pouvoir modifier soit le script, soit le schéma en amont.

Ce livre permettra, je l'espère, de bien appréhender le cheminement en question, en donnant des règles précises à suivre dans l'élaboration des différents modèles pour éviter de graves erreurs au niveau de la base. Seul le script SQL fera alors office de révélateur.

Chapitre 1

Le niveau conceptuel : face à face Merise/UML

You cannot design databases without a familiarity with the techniques of the ER diagramming.

*Database Design for Smarties,
Using UML for Data Modeling,
R.J. Muller, Morgan & Kaufman, 1999*

Ce chapitre détaille la première étape de conception d'une base de données : il s'agit d'élaborer un schéma conceptuel exprimé soit dans un formalisme de type entité-association avec ses extensions issues de Merise/2 soit à l'aide de la notation UML 2.

Il existe d'autres formalismes mais ils sont bien moins employés par la communauté francophone, citons le modèle entité-relation américain [CHE 76], NIAM (Nijssen Information Analysis Method) du nom du chercheur hollandais, ORM (Object Role Model) [HAL 01] qui étend et a pour but de remplacer NIAM, le langage Z [ABR 74], IDEF1X, etc.

D'un point de vue international, la troisième partie de la norme, l'ISO/IEC 11179 (Technologies de l'information - Coordination de la normalisation des éléments de données) décrit la façon dont doivent être organisées les données de façon sémantique. Cependant, le modèle conceptuel ne décrit en aucune façon une méthode logique pour représenter les données dans un ordinateur.

Dans ce chapitre, nous expliquons comment représenter :

- les faits et les événements qui décrivent le système à modéliser ;
- les différentes contraintes à prendre en compte (exemples : une compagnie aérienne n'affrète pas ses propres avions, un pilote ne doit voler que s'il détient une licence en cours de validité et une qualification valide pour le type d'avion en question, etc.) ;
- l'héritage.

Le schéma conceptuel exprime une vue abstraite de la base de données. Cette vue est représentée de manière graphique – on parle de diagramme, de schéma, de modèle, même si ce dernier mot est employé à toutes les sauces.

Il existe une différence entre un modèle (par exemple le modèle conceptuel de données) et un formalisme (dans lequel est décrit un modèle et qui n'exprime que l'aspect représentation). Ainsi, on parle de la modélisation conceptuelle des données suivant le formalisme entité-association ou suivant la notation UML.



La modélisation est un processus très important car il conditionne la structure de la base de données qui sera déduite des différents éléments du schéma conceptuel : entités (ou classes), associations et contraintes.

Généralités

Afin de préserver l'indépendance entre les données et les traitements, le schéma conceptuel ne doit pas comporter d'indications physiques. Pas question donc d'indiquer sur un diagramme une quelconque information sur l'indexage, l'adressage ou tout autre détail concernant l'accès à la mémoire.

Le schéma conceptuel doit contenir plus d'informations qu'on pouvait en trouver au début des fichiers COBOL, lorsqu'il s'agissait de déclarer les structures de données manipulées par les programmes eux-mêmes (la comparaison est un peu osée, mais les développeurs mûrs feront le rapprochement). Le concepteur devra ajouter au schéma les règles de gestion (aussi appelées « règles de sécurité », « d'intégrité » ou « de fonctionnement »).

L'objectif d'un schéma conceptuel ne peut pas être de décrire complètement un système, il modélise seulement l'aspect statique des données. Un schéma va aussi servir à communiquer et échanger des points de vue afin d'avoir, entre différents acteurs, une compréhension commune et précise d'un système à modéliser. Dans le monde de l'industrie, ces schémas ne sont plus manuscrits mais manipulés à l'aide d'outils graphiques (étudiés au chapitre 4).

Face à face Merise/UML

Nous réalisons ici un face à face entre le modèle conceptuel des données de Merise et le diagramme de classes de la notation UML. Pour chaque caractéristique, nous présenterons les différences entre les deux notations avant de tirer un bilan.

Concepts de base

Modèles entité-association

Le modèle entité-association, appelé « entité-relation » (*entity-relationship*) chez les Anglo-Saxons, a été proposé en 1976 des deux côtés de l'Atlantique. Si la paternité de ce modèle est

attribuée à P. Chen [CHE 76], [MOU 76], d'autres études proposent au même moment un modèle avec des concepts similaires. Il est d'ailleurs amusant de lire le titre de l'article de H. Tardieu [TAR 79] *A Method, A Formalism and Tools for Database Design (Three Years of Experimental Practice)*.

Le formalisme Merise a d'abord été nommé entité-relation. L'appellation entité-association, dans le cadre de Merise, est apparue au cours d'un congrès de l'AFCET en 1991. S'il est incontestable que P. Chen a fait la première publication en langue anglaise, l'équipe animée par H. Tardieu travaillait depuis 1974 sur le projet I(N)RIA « Méthode, Modèles et Outils pour la conception d'une base de données », jalonné par plusieurs publications dans l'environnement français. Le formalisme s'appelait alors « formalisme individuel », terme utilisé à la place d'entité. Au congrès IFIP de Namur en 1975, les deux approches ont été confrontées. Les publications respectives s'entremêlent au point qu'il est très difficile d'attribuer une paternité mais plutôt une simultanéité de publication.

Le modèle conceptuel des données (MCD) de Merise [TAR 83], [TAR 91] issu des travaux de [TAR 79b], a été étendu par les travaux du groupe 135 de l'AFCET avec la version 2 de Merise [PAN 94]. La notation Merise/2 a été introduite par la société SEMA. À cette période, les travaux de l'AFCET étaient plus complets et consensuels que ceux de SEMA [NAN 01], mais l'appellation est passée dans l'usage courant.

Les ouvrages et articles de recherche américains ignorent royalement depuis près de vingt ans le modèle européen, qui a pourtant fait son chemin. Peut-être est-ce le fruit de vieilles rancœurs ? Toujours est-il que le modèle Merise/2 est un modèle riche, qui propose de nombreuses contraintes sémantiques. Certaines contraintes ont été reprises par la notation UML, d'autres seront à redéfinir.

Décrivons à présent les concepts de base des modèles entité-association.



Attribut (attribute) : donnée élémentaire, également appelée « propriété », qui sert à caractériser les entités et les associations.

Entité (entity) : concept concret ou abstrait (fait, moment etc.) du monde à modéliser. Elle se représente par un cadre contenant le nom de l'entité et de ses attributs.

Identifiant (identify) : attribut particulier permettant d'identifier chaque occurrence d'une entité. L'identifiant est souligné.

L'entité permet de modéliser un ensemble d'objets de même nature, dignes d'intérêt dans le discours. La figure 1-1 décrit trois objets (des livres en l'occurrence). Si deux d'entre eux semblent identiques, il s'agit en fait de deux objets distincts pour la bibliothèque (livres numéros 1 et 3).

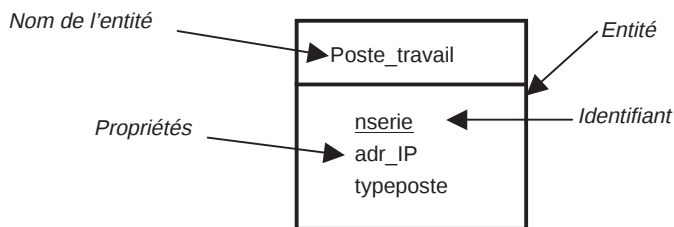
Si le concepteur doit les considérer comme semblables, il n'utilisera pas l'attribut *numero* en tant qu'identifiant mais *ISBN*. Les autres attributs (*titre*, *auteur*, *éditeur*...) seront inchangés.

Figure 1-1 Entité, attributs et identifiant

Chaque propriété doit figurer une seule fois sur le modèle conceptuel (principe de non-redondance).

Il faut éviter l'emploi de synonymes et de polysémies (mot présentant plusieurs sens) pour les attributs. Dans le même ordre d'idée, les mots réservés sont à proscrire.

L'exemple 1-2 décrit une entité. Un poste de travail a trois attributs : un numéro de série (chaîne de caractères, ex : p1), une adresse IP (chaîne de caractères, ex : 130.20.53.60) et un type (chaîne de caractères, ex : Unix, Windows).

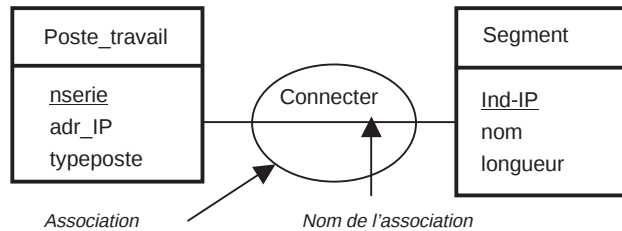
Figure 1-2 Entité

Association (relationship) : l'association permet de relier plusieurs entités entre elles. Une association se représente à l'aide d'un ovale (ou losange) contenant son nom et ses éventuels attributs et connectant plusieurs entités.

Les associations se déduisent en général des verbes du discours.

Dans notre exemple, les postes de travail sont *connectés* à un réseau local. Chaque poste est relié à un segment caractérisé par un indicatif IP, un nom et une longueur de câble. Le verbe « connecter » induit une association entre les entités Poste_travail et Segment. L'association Connecter est décrite dans la figure 1-3.

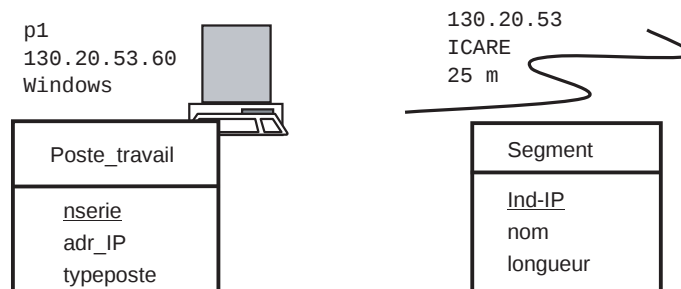
Figure 1-3 Association binaire



Occurrence : élément particulier d'une entité ou d'une association.

Dans l'exemple 1-4, le poste de travail p1 est un élément particulier du réseau local modélisé. Au niveau conceptuel, il s'agit d'une occurrence de l'entité Poste_travail. Pour sa part, le câble 130.20.53 est une occurrence de l'entité Segment.

Figure 1-4 Occurrences d'entités



Un exemple d'occurrence de l'association Connecter est la connexion du poste p1 au segment 130.20.53.

Figure 1-5 Occurrence d'une association





Degré (degree) d'une association : nombre d'entités connectées à cette association. Le degré est aussi appelé « arité » de l'association.

Dans Merise, on appelle « dimension » le nombre d'entités composant la relation ; on appelle « collection » la liste de ces entités. Une association qui relie deux entités est dite binaire, une association qui relie trois entités est dite ternaire, une association qui relie n entités est dite n -aire.

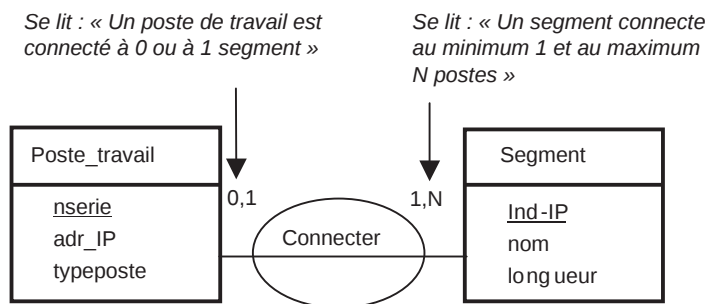


Cardinalités (cardinality) : couple de valeurs (minimum, maximum) indiqué à l'extrémité de chaque lien d'une association. Il caractérise la nature de l'association en fonction des occurrences des entités concernées.

Les cardinalités traduisent les possibilités de participation (mini, maxi) d'une occurrence quelconque d'une entité aux occurrences d'une association (donc des $n-1$ entités de sa collection). C'est pour cela que cette participation se note sur le lien entre l'entité et l'association.

Dans l'exemple 1-6, les cardinalités décrivent la nature de l'association **Connecter** : un poste de travail est relié au plus à un segment et un segment de câble permet de connecter plusieurs postes de travail.

Figure 1-6 Cardinalités Merise



L'interprétation des cardinalités constitue la différence fondamentale entre le formalisme du modèle de P. Chen et le MCD de Merise.



Les cardinalités d'une association binaire dans le modèle de Chen et de Merise sont inversées au niveau de l'axe de représentation de l'association.

Pour la petite histoire, dans les premières versions du formalisme entité-association français, les cardinalités étaient notées selon le formalisme américain, influencé par la majorité des relations binaires. C'est en réfléchissant sur les associations n -aires que le groupe de travail de l'AFCET a choisi (fin 1975) la notation actuelle. Ce choix a l'avantage d'être cohérent quel que soit le degré de l'association car les cardinalités sont indépendantes de la dimension de l'association.

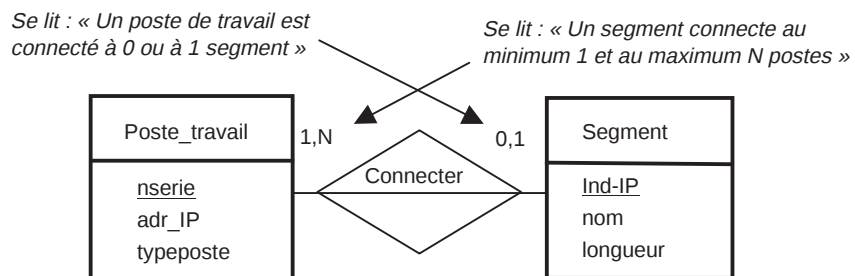
Les cardinalités d'une association n -aire sont interprétées différemment dans les deux formalismes. Dans le modèle de Chen, les contraintes de cardinalités d'une entité donnée sont lues à partir des autres entités de l'association (sens entités connectées $\rightarrow$ entité concernée), alors qu'avec Merise, elles sont lues du sens entité concernée $\rightarrow$ entités connectées. De plus, contrairement aux associations binaires et pour une modélisation donnée, les cardinalités n'auront pas les mêmes valeurs pour les deux formalismes [SOU 98].

Le problème de l'approche de P. Chen réside dans son manque de cohérence entre la représentation des associations binaires et des associations n -aires. La majorité des outils de conception américains n'ont pas suivi cette vision des choses car ils ont été incapables de programmer ce concept (même le produit Designer d'Oracle). Ces outils modélisent les associations n -aires en définissant $n+1$ entité(s), dont n sont reliées à une seule par des associations binaires *un-à-plusieurs*.

Il en va de même pour la notation UML, qui bien qu'adoptant la notation française pour les associations n -aires, voit limitée la programmation d'un tel concept (bon nombre d'outils comme Rational Rose supportent mal le concept d'association n -aire). D'ailleurs, dans son dernier ouvrage G. Booch [BOO 01] reconnaît un « problème » avec les associations n -aires (qu'il passe ensuite rapidement à la trappe...). Nous verrons ultérieurement qu'il n'y a pas de « problème » et reviendrons concrètement sur l'interprétation des cardinalités des associations n -aires dans le cadre de ces deux approches.

L'exemple 1-7 décrit l'association binaire *Connecter* dans le formalisme proposé par P. Chen.

Figure 1-7 Formalisme de P. Chen

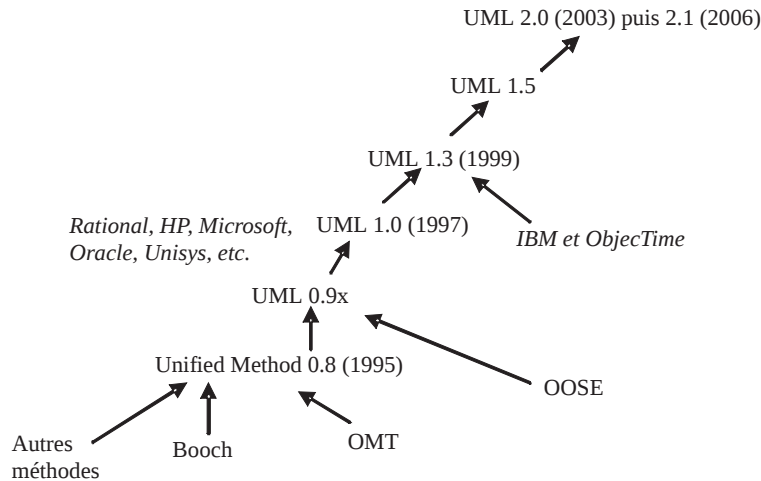


Nous verrons que les cardinalités d'une association binaire dans le modèle de Chen et dans le formalisme UML sont positionnées de façon identique.

Notation UML

En 1994, G. Booch, père de la méthode qui porte son nom [BOO 91], et J. Rumbaugh, principal acteur de la proposition de la méthode OMT [RUM 91], décident de rassembler leurs efforts au sein de la société Rational Software afin d'unifier leurs méthodes. Un an plus tard, I. Jacobson, créateur de la méthode OOSE [JAC 92], rejoint cette société pour intégrer sa méthode au projet UML.

Figure 1-8 Évolution d'UML



Les travaux ont continué après son adoption par d'importants acteurs du marché (HP, Microsoft, Oracle, Unisys). La version 1.0 d'UML est sortie en janvier 1997 et, au cours de cette même année, le langage UML a été accepté par l'OMG (Object Management Group). Il est actuellement disponible en version 2.0 (<http://www.uml.org/>). Les créateurs d'UML insistent tout particulièrement sur le fait que leur notation est un langage de modélisation objet et non pas une méthode. La notation UML peut ainsi se substituer sans perte de sémantique aux notations de Booch, d'OMT ou d'OOSE. Le lecteur intéressé dispose d'ouvrages de qualité en français sur la notation UML, citons [BOO 00], [MUL 00] et [ROQ 06].



Attribut (attribute) : donnée élémentaire servant à caractériser les classes et les relations.

Classe (class) : description abstraite d'un ensemble d'objets de même structure et de même comportement extraits du monde à modéliser.

Méthodes (methods) : opérations programmées sur les objets d'une classe.

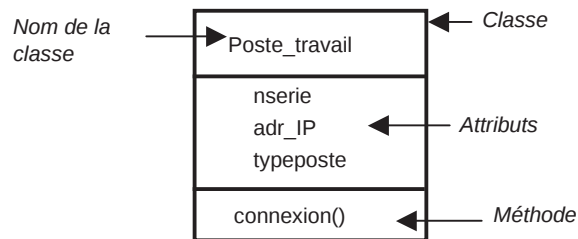
La description des classes UML se divise en trois compartiments contenant respectivement le nom de la classe, les attributs de la classe et la signature des méthodes de la classe.



Bien qu'il n'était pas utile de disposer d'un identifiant pour chaque classe avec UML, il faudra définir un (ou plusieurs) attribut(s) assurant ce rôle dans le but de préparer le passage à SQL. Pensez à disposer l'identifiant en tête des attributs de la classe.

L'exemple 1-9 décrit une classe avec la notation UML. Un poste de travail est caractérisé par trois attributs (ici le numéro de série jouera le rôle de l'identifiant) et dispose d'une méthode.

Figure 1-9 Classe UML

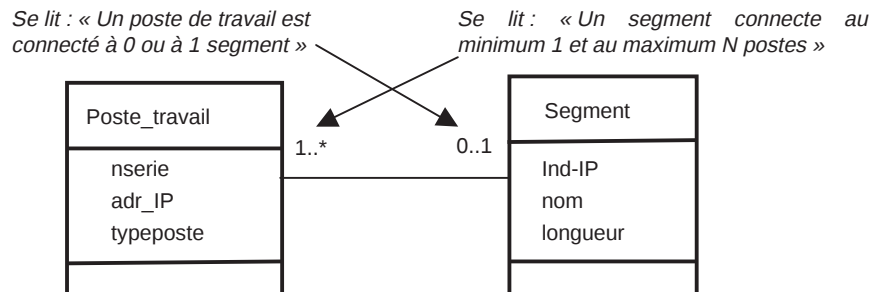


Association (relationship) : l'association permet de relier une classe à plusieurs autres classes.

Multiplicité (multiplicity) : chaque extrémité d'une association porte une indication de multiplicité. Elle exprime le nombre minimum et maximum d'objets d'une classe qui peuvent être reliés à des objets d'une autre classe.

L'exemple 1-10 décrit l'association modélisant la connexion des postes aux segments selon la notation UML. Les classes reliées entre elles sont *Poste_travail* et *Segment*. Comme nous l'avons évoqué précédemment, les cardinalités dans le modèle de Chen et pour le formalisme UML sont positionnées à l'identique sur l'axe de représentation de l'association.

Figure 1-10 Association UML



La spécification UML 2 (Superstructure - version 2.0 - formal/05-07-04) indique qu'une association est représentée par une ligne connectant deux classes (dans le contexte d'un diagramme de classes) ou une classe avec elle-même. Il y est même conseillé de soigner la présentation des segments de droites quand le lien n'est pas rectiligne.

« *A binary association is normally drawn as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance, but they may be graphically meaningful to a tool in dragging or resizing an association symbol* ».

Nous détaillerons plus loin certaines caractéristiques des associations qu'il est intéressant d'utiliser dans un contexte de bases de données (nommage, rôle, classe-association).

Terminologie et conventions

Les tableaux 1.1 et 1.2 établissent un parallèle entre les formalismes du modèle entité-association et de la notation UML.

Tableau 1.1 Terminologie

Entité-association	UML
Entité	Classe
Association (Relation)	Association (Relation)
Occurrence	Objet
Cardinalité	Multiplicité
Modèle conceptuel de données (Merise)	Diagramme de classes

Tableau 1.2 Cardinalités/multiplicités

Cardinalités	Multiplicités d'UML
0, 1	0..1
1, 1	1 ¹
0, N	0..* ou *
1, N	1..*
N, N ²	N..N

1. Ou absence de cardinalité (par défaut)

2. N,N : permet par exemple de spécifier qu'un segment doit connecter entre trois postes et huit postes. Les cardinalités du côté Poste_travail seront (3, 8) dans le modèle entité-association et 3..8 avec UML.

Au niveau de la conception, il est important de déterminer correctement le chiffre maximal des cardinalités. En effet, les méthodes de conception reposent sur la transformation des entités

(ou classes) et des associations en fonction des cardinalités maximales (le processus que nous proposons dans cet ouvrage ne déroge pas à cette règle).

Les cardinalités minimales précisent les liens d'associations et nécessitent parfois l'intervention d'un programmeur, mais elles n'ont pas une grande influence sur la construction de la base de données. Notez que la cardinalité minimale 0 autorise une valeur NULL dans la base, que la cardinalité minimale 1 interdit une valeur NULL et que la cardinalité minimale N induit une vérification de cette contrainte, soit par programme, soit par déclencheur (*trigger*).



Merise appelle CIF (contrainte d'intégrité fonctionnelle) une association ayant un lien de cardinalité maximale égale à 1. Cette CIF est notée d'une flèche sur le lien connecté à l'entité cible.

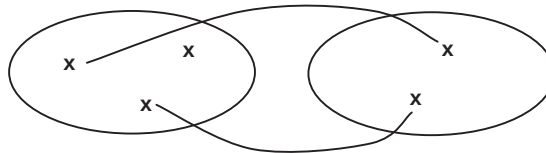
Concernant les CIF de Merise (et Merise/2), prenons l'exemple d'un employé travaillant pour un département qui regroupe plusieurs employés. L'association Travailler est CIF car il y a (0,1) ou (1,1) du côté de l'entité Employe. La flèche sera positionnée sur le lien entre Travailler et Departement et ciblera cette dernière entité. Nous verrons plus loin des exemples construits avec l'outil Win'Design.

Comparons à présent les formalismes en fonction des différents types d'associations. Étudions successivement les associations de type *un-à-un* (*one-to-one*), *un-à-plusieurs* (*one-to-many*), *plusieurs-à-plusieurs* (*many-to-many*) [MAR 88]. Nous utilisons cette classification tout au long de l'ouvrage.

Associations un-à-un

On utilise une association *un-à-un* entre deux entités (classes) si toute occurrence (objet) d'une entité (classe) est en rapport au plus avec une occurrence (objet) de l'autre entité (classe) et inversement. Ce sont les associations les moins répandues.

Figure 1-11 Association un-à-un



Une association *un-à-un* est une association binaire ayant :

- la cardinalité maximale 1 sur chaque lien dans le formalisme entité-association ;
- la multiplicité 0..1 ou 1 sur chaque lien dans le cadre de la notation UML.

Les équivalences entre cardinalités et multiplicités d'une association *un-à-un* sont indiquées dans le tableau 1-3. Les cardinalités 1,1-1,1 sont une anomalie de modélisation car elles expriment une correspondance biunivoque et totale entre les deux entités, à tel point qu'elles sont quasiment « confondues ».

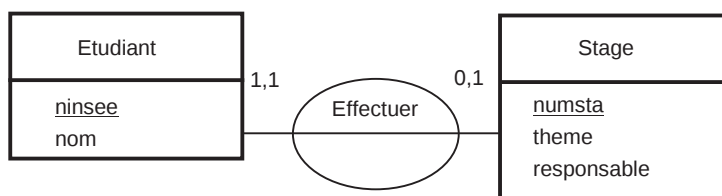
Tableau 1.3 Associations un-à-un

Cardinalités	Multiplicités UML
0,1 - 0,1	0..1 - 0..1
0,1 - 1,1	0..1 - 1
1,1 - 1,1	1 - 1

Modèles entité-association

Dans l'exemple 1-12, un étudiant est caractérisé par un numéro INSEE et un nom. Chaque étudiant doit effectuer un seul stage en entreprise. Un stage est désigné par un numéro, un thème et le nom du responsable ; il est suivi par un seul étudiant.

Figure 1-12 MCD d'une association un-à-un

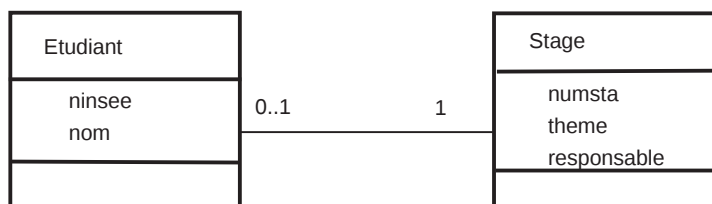


Il suffit d'inverser les cardinalités pour décrire l'association **Effectuer** dans le formalisme de Chen.

Notation UML

L'exemple 1-13 décrit la notation UML qui représente l'association *un-à-un*.

Figure 1-13 Association un-à-un UML



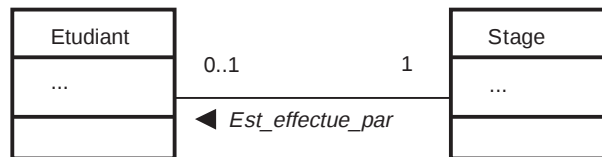


Avec UML, une association peut être *nommée*. Le nom apparaît au milieu du lien d'association.

Il est recommandé de nommer les associations par une forme verbale active ou passive. Dans les deux cas, le sens de la lecture peut être précisé au moyen d'un petit triangle dirigé vers la classe désignée par la forme verbale. Il est à noter qu'on ne retrouve pas toujours ce nom au niveau du code SQL.

Dans notre exemple, nous utilisons une forme passive pour nommer l'association modélisant les affectations des stages, à savoir *Est_effectue_par*. Pour améliorer la lecture du diagramme, il est possible de décorer le nom de l'association par un symbole précisant le sens de lecture.

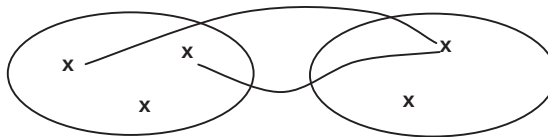
Figure 1-14 Association nommée



Associations un-à-plusieurs

On utilise une association *un-à-plusieurs* entre deux entités/classes si une occurrence/objet d'une entité/classe peut être en rapport avec plusieurs occurrences/objets de l'autre entité/classe et pas inversement.

Figure 1-15 Association un-à-plusieurs



Une *association un-à-plusieurs* (nommée aussi « père-fils »), est une association binaire ayant :

- une cardinalité maximale N et une cardinalité maximale 1 dans le formalisme entité-association ;
- une multiplicité * ou 1..* d'un côté et une multiplicité maximale 1 de l'autre avec UML.

Les équivalences entre cardinalités et multiplicités d'une association *un-à-plusieurs* sont indiquées dans le tableau 1.4.

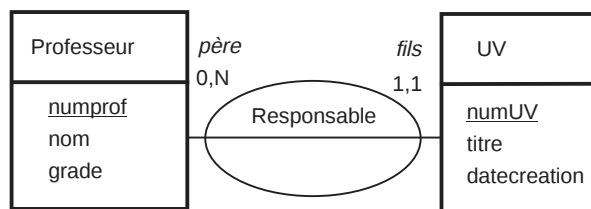
Tableau 1.4 Associations un-à-plusieurs

Cardinalités EA	Multiplicités UML
0, 1 - 0, N	0..1 - *
0, 1 - 1, N	0..1 - 1..*
1, 1 - 0, N	1 - *
1, 1 - 1, N	1 - 1..*

Modèles entité-association

Dans la figure 1-16, un professeur, caractérisé par un numéro, un nom et un grade, peut être responsable de plusieurs unités de valeurs (UV). Chaque UV, désignée par un numéro, un titre et une année de création, est placée sous la responsabilité d'un seul professeur. Il existe des professeurs qui ne sont responsables d'aucune UV. L'entité **Professeur** est dite « père » car un professeur est en relation avec plusieurs UV « fils ». Il suffit d'inverser les cardinalités pour décrire l'association **Responsable** dans le formalisme de Chen.

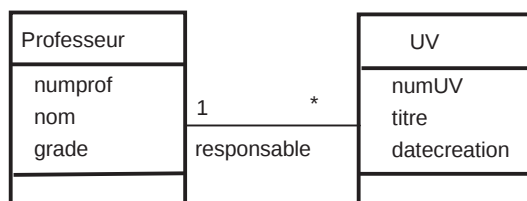
Figure 1-16 MCD d'une association un-à-plusieurs



Notation UML

Le diagramme de classes UML équivalent est représenté dans la figure 1-17.

Figure 1-17 Association un-à-plusieurs UML





Avec UML, l'extrémité d'une association peut être enrichie d'un *rôle*, qui décrit la façon dont la classe perçoit l'autre classe (ou les autres classes pour les associations *n*-aires) *via* l'association.

Un rôle est généralement désigné par une forme nominale ou verbale. Le rôle est placé à une extrémité du lien d'association, il se distingue ainsi du nom de l'association situé au centre du lien.

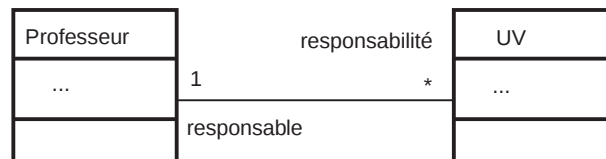
Il est particulièrement intéressant de nommer les rôles ou de nommer les associations lorsque plusieurs associations relient deux mêmes classes. En général, il n'y a pas de corrélation entre les objets qui participent aux différentes associations entre deux mêmes classes. Chaque lien exprime un état de fait distinct.

Le rôle est indispensable pour les associations réflexives (étudiées plus loin dans ce chapitre). Il est à noter que le concept de rôle existait aussi avec Merise/2. Nous verrons au chapitre 4 que cette notion est prise en compte par les outils du marché. Le rôle s'affiche à chaque extrémité du lien d'association.

De même que pour les noms d'associations, on ne retrouve pas toujours le nom des rôles au niveau du code SQL2, mais on peut les retrouver dans du code SQL3 par l'intermédiaire des noms de références ou de collections.

Dans l'exemple 1-18, le professeur perçoit les UV comme une certaine responsabilité et une UV perçoit un professeur comme son responsable.

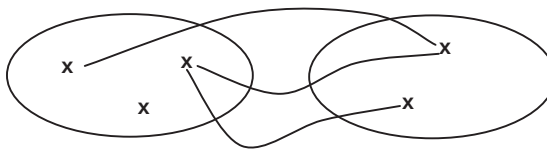
Figure 1-18 Rôles avec UML



Bien qu'il soit possible d'utiliser conjointement les associations nommées et les rôles, il semble préférable d'opter pour l'une ou l'autre de ces deux techniques au sein d'un même diagramme de classes. Les développeurs objet préfèrent souvent le rôle à l'association nommée. Beaucoup d'outils rendent le rôle obligatoire.

Associations plusieurs-à-plusieurs

On utilise une association *plusieurs-à-plusieurs* entre deux entités (classes) si une occurrence (objet) d'une entité (classe) peut être en rapport avec plusieurs occurrences (objets) de l'autre entité (classe) et inversement.

Figure 1-19 Association plusieurs-à-plusieurs

Une association *plusieurs-à-plusieurs* est une association binaire ayant :

- la cardinalité maximale N sur chaque lien dans le modèle entité-association ;
- la multiplicité * ou 1..* sur chaque lien pour la notation UML.

Une association *plusieurs-à-plusieurs* peut, en outre, posséder des attributs.

Les équivalences entre cardinalités et multiplicités d'une association *plusieurs-à-plusieurs* sont indiquées dans le tableau 1-5.

Tableau 1.5 Associations plusieurs-à-plusieurs

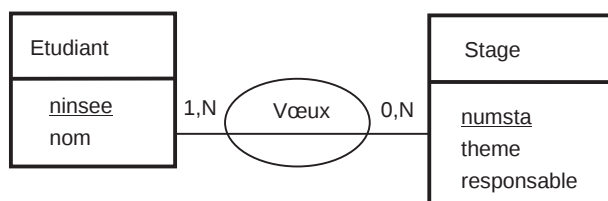
Cardinalités	Multiplicités UML
0, N – 0, N	* – *
0, N – 1, N	* – 1..*
1, N – 0, N	1..* – *
1, N – 1, N	1..* – 1..*

Nous traiterons d'abord des associations sans attributs puis des associations avec attributs. Dans ce dernier cas, il faudra s'assurer du bien-fondé de chaque attribut dans l'association.

Associations *plusieurs-à-plusieurs* sans attribut

Modèles entité-association

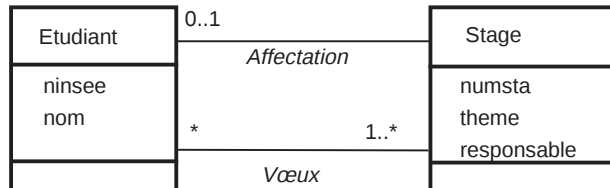
Dans l'exemple 1-20, chaque étudiant émet un ou plusieurs vœux concernant des stages. Un stage peut n'intéresser aucun étudiant ou, au contraire, en attirer plusieurs. Il suffit d'inverser les cardinalités pour décrire l'association Vœux avec le formalisme de Chen.

Figure 1-20 MCD d'une association plusieurs-à-plusieurs

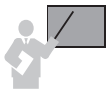
Notation UML

Dans l'exemple 1-21, nous représentons à la fois les vœux et les affectations qui lient les étudiants aux stages en nommant les associations.

Figure 1-21 Associations nommées avec UML



Associations plusieurs-à-plusieurs avec attributs



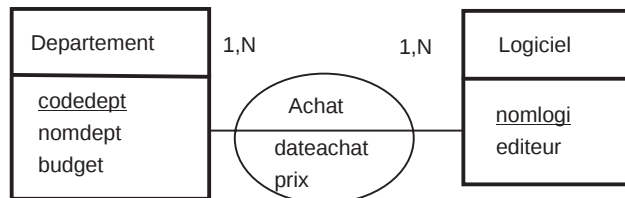
Un attribut *a* est correctement placé dans une association entre les entités/classes *A* et *B* si :

- Une occurrence/objet de *A* peut être en rapport avec plusieurs valeurs de *a* ;
- Une occurrence/objet de *B* peut être en rapport avec plusieurs valeurs de *a* ;
- Un couple d'occurrence/objet de *A* et *B* n'est en rapport qu'avec au plus une valeur de *a*.

Modèles entité-association

Supposons qu'un département achète des logiciels. Un logiciel peut être acheté par un ou plusieurs département(s) à différentes dates et à différents prix. L'exemple 1-22 précise à l'aide des cardinalités minimales que chaque département a acheté au moins un logiciel et que chaque logiciel a été acheté par au moins un département.

Figure 1-22 MCD d'une association plusieurs-à-plusieurs avec attributs



Les attributs *dateachat* et *prix* sont bien des attributs de l'association *Achat*, car ils dépendent à la fois des deux entités dont il est question. Nous verrons que la notion de dépendance aide à déterminer les attributs d'une association.

En plaçant l'attribut `dateachat` dans `Departement` ou dans `Logiciel`, nous exprimons deux choses différentes, à savoir qu'un département n'achète des logiciels qu'à une seule date ou qu'un logiciel ne peut être acheté qu'à un seul moment de l'année. Ces deux remarques valent également pour l'attribut `prix`.

Notation UML

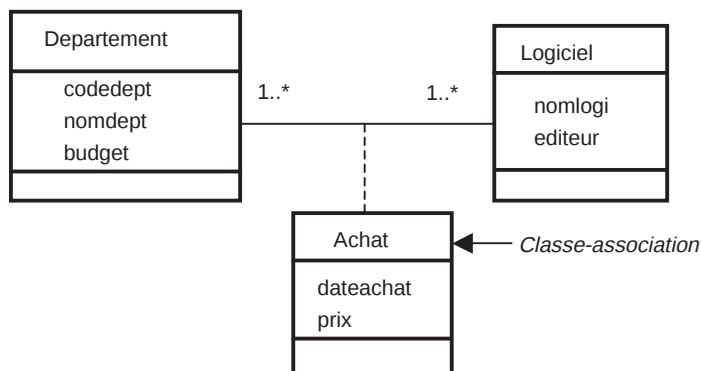


Une association *plusieurs-à-plusieurs* avec attributs est représentée sous UML par une *classe-association*. Cette classe-association contient les attributs de l'association et est connectée au lien d'association par une ligne en pointillé.

Notons qu'il est aussi possible d'utiliser une classe-association pour modéliser une association *un-à-plusieurs* ou *plusieurs-à-plusieurs* sans attributs.

Le diagramme de classes 1-23 décrit les achats de logiciels par les départements.

Figure 1-23 Classe-association avec attribut sous UML



Associations n-aires



Une *association n-aire* connecte *n* entités/classes. Une association *n-aire* peut également posséder ou non des attributs.

Détaillons à présent les interprétations des cardinalités d'une association *n-aire*, qui varient selon les formalismes entité-association. Ces approches se différencient selon le sens de lecture des cardinalités au niveau des entités concernées par l'association *n-aire*.

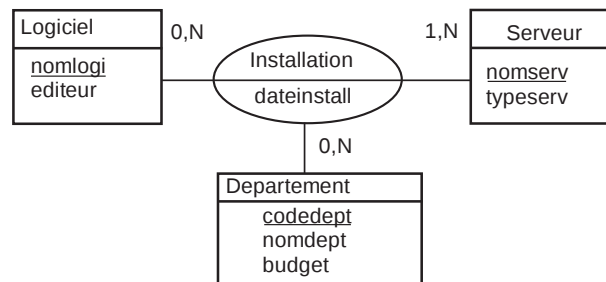
Dans le modèle américain de P. Chen, les contraintes de cardinalités d'une entité donnée sont lues à partir des autres entités (du sens entités connectées→entité concernée) alors qu'avec Merise, elles sont lues du sens entité concernée→entités connectées.

Bien que l'exemple qui nous servira de fil conducteur fasse intervenir une association de degré 3 (3-aire), il est aisé de transposer notre propos à des associations de degré supérieur.

MCD Merise

Le schéma conceptuel 1-24 modélise le fait que des logiciels sont installés sur des serveurs sur l'initiative de départements. On désire savoir la date à laquelle un département a procédé à l'installation d'un logiciel sur un serveur.

Figure 1-24 Association 3-aire Merise



Le formalisme de type Merise impose la lecture des cardinalités suivantes :

- du côté **Logiciel**, *combien de couples (Département, Serveur) peuvent être associés à un logiciel* ? Réponse : $(0, N)$, car un logiciel peut ne pas être installé ou au contraire être installé sur plusieurs serveurs par différents départements ;
- du côté **Serveur**, *combien de couples (Département, Logiciel) peuvent être associés à un serveur* ? Réponse : $(1, N)$, car un serveur doit au moins héberger un logiciel installé par un département et un serveur peut héberger plusieurs logiciels sur l'initiative de différents départements ;
- du côté **Département**, *combien de couples (Serveur, Logiciel) peuvent être associés à un département* ? Réponse : $(0, N)$, car un département peut ne pas être impliqué dans une installation de logiciel. En revanche, il peut être l'instigateur de plusieurs installations mettant en jeu différents logiciels et serveurs.

Par ailleurs, ce diagramme ne laisse apparaître aucune autre contrainte. En d'autres termes, n'importe quel logiciel peut être installé sur l'initiative de n'importe quel département sur n'importe quel serveur. (Transposons ce schéma dans le modèle relationnel : la table **Installation** va être définie avec sa clé primaire composée des identifiants des trois entités. Il est alors possible d'insérer n'importe quel triplet (nomlogi, nomserv, codedept, dans cette table.) Est-ce la réalité à modéliser ? Probablement pas !

La majorité des associations n -aires sans contrainte additionnelle ne permettent pas de représenter convenablement le monde réel. Nous consacrons par la suite un paragraphe à la mise en

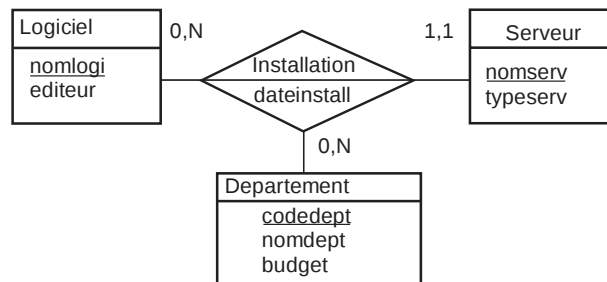
œuvre de contraintes additionnelles sur une association n -aire. Pour raisonner avec des associations de degré supérieur, il suffit de considérer des triplets (pour une association de degré 4) au lieu des couples (comme c'est le cas dans notre exemple), etc.

La notion de cardinalité est insuffisante pour intégrer la contrainte suivante : *un logiciel d'un département ne doit être installé que sur un seul serveur*. La solution consiste à définir une contrainte d'unicité que Merise appelle CIF (contrainte d'intégrité fonctionnelle). Nous détaillerons par la suite le schéma qu'il convient d'élaborer à cet effet.

Formalisme de Chen

Décrivons l'exemple 1-25 d'association 3-aire Installation dans le formalisme de P. Chen.

Figure 1-25 Association 3-aire (formalisme de Chen)



Ce formalisme impose la lecture des cardinalités suivantes :

- du côté *Logiciel*, *combien de logiciels peuvent être associés à un couple (Département, Serveur)* ? Réponse : $(0, N)$, car un logiciel peut ne pas être installé, en revanche plusieurs logiciels peuvent être installés sur le même serveur par un département ;
- du côté *Serveur*, *combien de serveurs peuvent être associés à un couple (Département, Logiciel)* ? Réponse : $(1, 1)$, car nous représentons directement la contrainte qu'un logiciel d'un département ne doit être installé que sur un seul serveur ;
- du côté *Département*, *combien de départements peuvent être associés à un couple (Serveur, Logiciel)* ? Réponse : $(0, N)$, car un département peut ne pas être impliqué dans une installation de logiciel et en revanche plusieurs départements peuvent utiliser le même logiciel sur un serveur donné (on parle alors d'installation de licence).

On constate que ce formalisme permet de représenter intrinsèquement les contraintes d'unicité (appelées aussi CIF dans Merise) que nous détaillerons plus loin.

Ce qu'en disait la littérature

Les associations n -aires ont toujours été très peu mises en avant dans la littérature consacrée à UML. Les exemples sont rares, les explications encore plus ! Les premières versions des

principaux outils anglo-saxons (comme Rational Rose) ne les géraient pas (le chapitre 4 fait le point sur cet aspect des choses). Seul Rumbaugh [RUM95] avec OMT abordait explicitement cet aspect (en le limitant aux ternaires et en niant quasiment les dimensions supérieures). Il y est dit : « La notation symbole de multiplicité (celle d'OMT) fonctionne bien quand il s'agit d'associations binaires... Cependant cette notation est ambiguë pour les associations n -aires ($n > 2$). La meilleure approche est de préciser les clés candidates. Une clé candidate est un ensemble minimum d'attributs qui identifie uniquement un objet... ». Ce dernier conseillait de promouvoir l'association ternaire comme une classe. Si UML avait poursuivi cette voie :

- il ne devrait pas y avoir de multiplicité notée sur les ternaires !
- on devrait introduire la notion d'identifiant ou de clé candidate sur les associations !
- l'interprétation se ferait ensuite au niveau logique !

Notation UML

On lit désormais dans la spécification (Unified Modeling Language : Superstructure - version 2.0 - formal/05-07-04) que la lecture des multiplicités doit se faire selon le mode de lecture du formalisme de Chen (exemple de la figure 1-25).

« For n -ary associations, the lower multiplicity of an end is typically 0. The lower multiplicity for an end of an n -ary association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends ».

On apprend dans cette même spécification que toute association binaire pourrait être représentée à l'aide du symbole losange.

« Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. An association with more than two ends can only be drawn this way ».

Cette notation n'est pas très utilisée en pratique (seul l'outil Together de Borland l'a adoptée). De plus, ce formalisme peut porter à confusion car il ressemble au symbole de l'agrégation partagée.

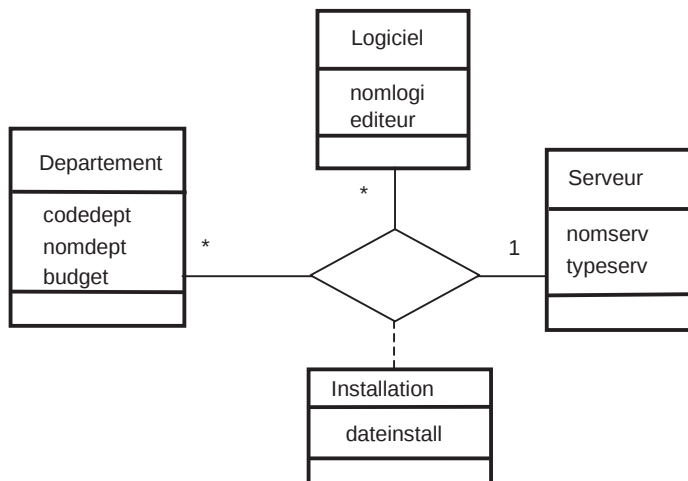


Une association n -aire sans contrainte se représente avec UML soit par un *losange* (symbole de l'association) qui connecte les n classes et une classe-association, soit par une classe *stéréotypée* reliée aux n classes.

Une association n -aire avec contrainte se représente plus facilement avec UML par une ou plusieurs classes-associations reliant les n classes.

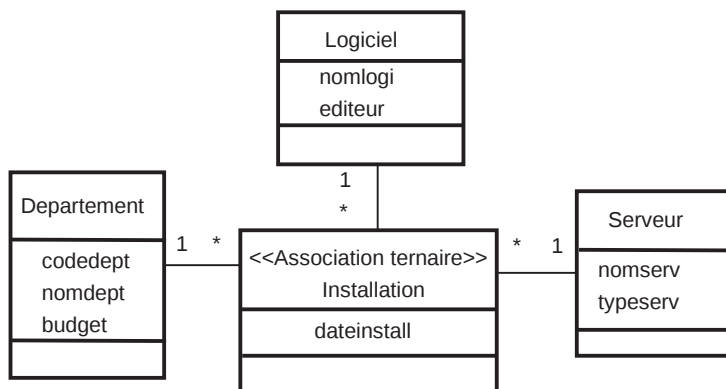
Association en losange

Le diagramme de classes 1-26 modélise l'association *Installation* précédente à l'aide du symbole de l'association n -aire. La contrainte d'unicité apparaît explicitement (multiplicité 1 du côté *Serveur*).

Figure 1-26 Association 3-aire UML avec un losange

Classe stéréotypée

L'exemple 1-27 décrit la même association à l'aide d'une classe stéréotypée `<<Association ternaire>>`. La contrainte d'unicité n'apparaît plus explicitement.

Figure 1-27 Association 3-aire UML avec stéréotype

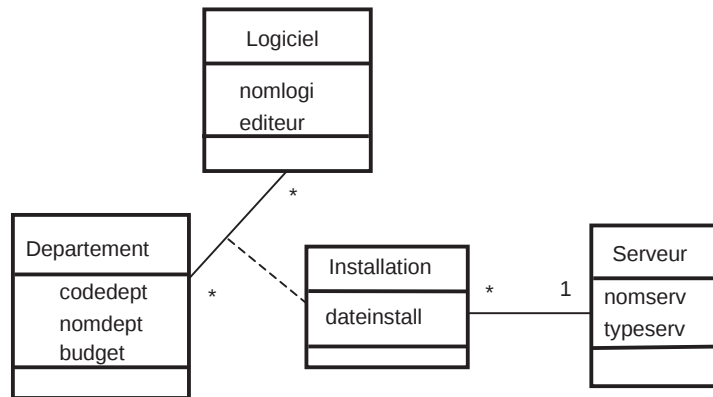
Cette notation n'est pas conseillée à moins d'utiliser un outil qui ne prend pas en compte le symbole losange de l'association et de vouloir toutefois modéliser une association n -aire sans contrainte.

Utilisation d'une classe-association

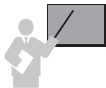
Cette solution va nous permettre de prendre en compte explicitement la contrainte d'unicité qui concerne l'installation des logiciels : *un logiciel d'un département n'est installé que sur un seul serveur*.

Le diagramme 1-28 met en œuvre la classe-association Installation en liaison avec la classe Serveur. Il met en évidence qu'une installation (constituée d'un logiciel et d'un département à une date donnée) est associée à un seul serveur (multiplicité 1 du côté Serveur).

Figure 1-28 Association 3-aire avec contrainte d'unicité



Associations réflexives



Une *association réflexive* est une association binaire ou *n*-aire qui fait intervenir au moins deux fois la même entité/classe.

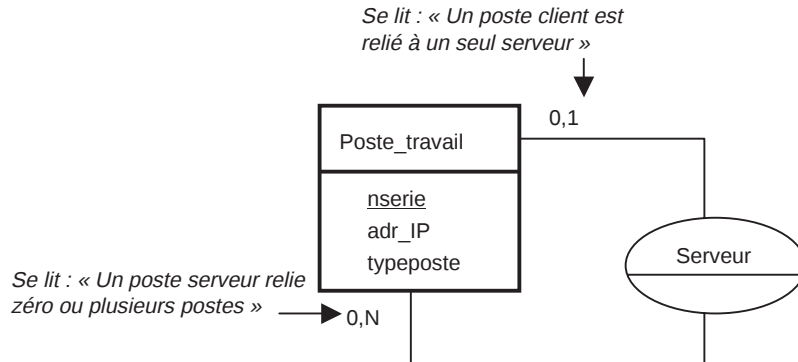
Dans les relations réflexives, il faut préciser le rôle de chaque lien, surtout si les cardinalités ne permettent pas de lever l'ambiguïté. Considérons les deux exemples suivants.

Association un-à-plusieurs réflexive

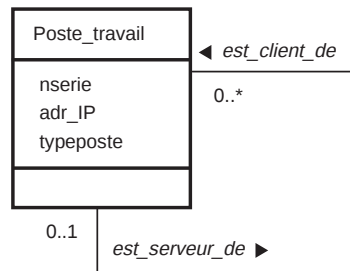
Dans la population des postes de travail, certains éléments sont serveurs et d'autres sont clients. La relation entre les postes clients et serveurs est réflexive, car elle porte sur la même classe : Poste_travail.

Modèles entité-association

Le MCD 1-29 illustre cet état de fait. Il suffirait d'inverser les cardinalités pour décrire l'association Serveur avec le formalisme de Chen.

Figure 1-29 Association un-à-plusieurs réflexive Merise**Notation UML**

Le diagramme 1-30 utilise deux rôles pour faciliter la lecture de l'association.

Figure 1-30 Association un-à-plusieurs réflexive UML**Association plusieurs-à-plusieurs réflexive**

Considérons l'exemple que l'accès à certaines unités de valeurs (UV) est conditionné par des prérequis. Les UV elles-mêmes prérequis constituent éventuellement un prérequis pour la préparation d'une ou de plusieurs autres UV. Ainsi, pour passer l'UV BD objet-relationnelle, il faut avoir les UV BD relationnelle et SQL. Par ailleurs, l'accès aux UV JDBC et Oracle 8 suppose l'obtention de l'UV BD objet-relationnelle. La modélisation Merise 1-32 met en œuvre deux rôles.

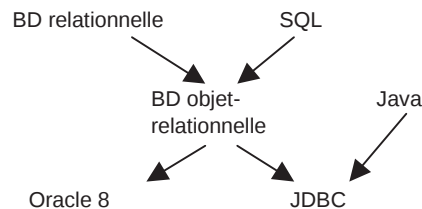
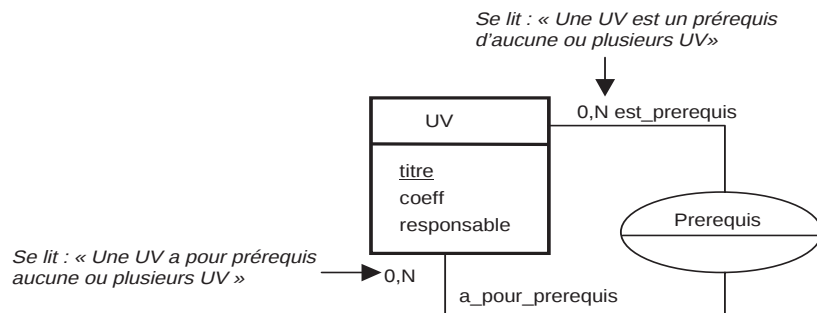
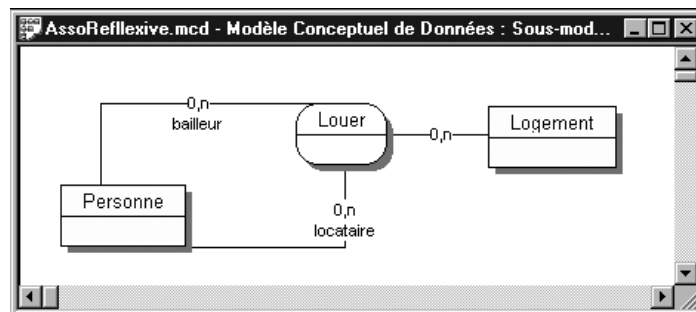
Figure 1-31 UV prérequis

Figure 1-32 Association plusieurs-à-plusieurs réflexive Merise

Association *n*-aire réflexive

Une association *n*-aire est réflexive si plusieurs liens issus de l'association relient la même entité/classe. L'exemple 1-33 modélise les contrats de location d'appartements. L'entité *Personne* modélise à la fois les propriétaires et les locataires.

Figure 1-33 Association *n*-aire réflexive Merise (copie d'écran Win'Design)

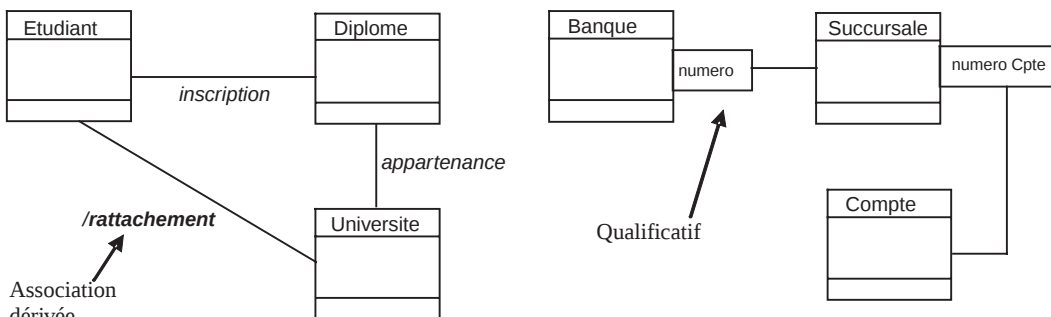
Associations dérivées et qualifiées

La notation UML propose deux notions supplémentaires concernant les associations.

- une association qualifiée est une association qui permet de restreindre les objets référencés dans une association grâce à une clé ;
- une association dérivée est une association déductible d'une ou de plusieurs autres associations existantes.

L'exemple 1-34 illustre deux associations qualifiées et une association dérivée.

Figure 1-34 Association qualifiée et dérivée



On comprend qu'il sera possible de déduire l'université de rattachement d'un étudiant à partir de l'association qui le relie à son diplôme, s'il appartient lui-même à une seule université.

Concernant les associations qualifiées, un compte bancaire sera identifié par un numéro et le numéro de la succursale. Une succursale sera identifiée par son numéro et le numéro de la banque.



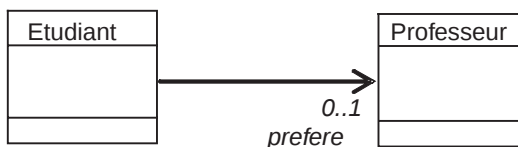
Alors que ces associations peuvent aider à la compréhension d'un schéma, elles ne sont pas intéressantes dans un contexte de base de données car elles auraient plutôt tendance à parasiter le passage au niveau logique (redondances ou incohérences en opposition avec la volonté de normaliser le schéma). Il est donc conseillé de ne pas les représenter (à moins de vraiment bien maîtriser le processus de génération de la base).

Associations navigables

Par défaut, une association UML est navigable dans les deux sens. Cela signifie que si, dans le cadre de la relation, un objet *A* est relié à un objet *B*, par définition *B* est aussi relié à *A* (dans le cadre de cette même relation). UML 2 permet d'exprimer dans un diagramme que les instances d'une classe ne connaissent pas les instances d'une autre bien qu'étant reliées entre elles dans le cadre de l'association.

Considérons l'association 1-35 qui relie un étudiant à son professeur préféré. Par déontologie, il ne doit pas être question que le professeur puisse savoir qui sont ses fans (bien qu'il puisse avoir lui aussi ces chouchous, auquel cas il s'agira d'une autre association navigable dans le sens inverse).

Figure 1-35 Association navigable





La propriété de navigation est vérifiée dans la grande majorité des associations. La réduction de la portée d'une association sera réalisée en phase d'implémentation (par programmation). Il n'y a pas de sens à définir une association précisant la navigation des deux côtés simultanément.

Il existe un débat au sein de la communauté UML pour savoir s'il faut préciser le rôle et la multiplicité du côté du lien non navigable. Bien qu'ils n'aient pas de sens, les renseigner peut aider la compréhension générale. Dans l'exemple, on aurait pu inscrire le rôle fans et la multiplicité 0..* (un professeur peut être le préféré d'aucun ou de plusieurs étudiants, sachant qu'il ne pourra jamais connaître leur identité).

Contraintes

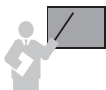
Les contraintes permettent d'améliorer la sémantique d'un schéma conceptuel.

- Merise/2 distingue les contraintes intra-association (qui s'appliquent au sein de l'ensemble des occurrences d'une association), dont fait partie la CIF, et les contraintes inter-associations (qui s'appliquent entre des occurrences de plusieurs associations partageant des entités).
- La spécification de UML 2 ne propose pas beaucoup de contraintes, le chapitre 4 décrit les possibilités actuelles des outils en la matière. En contrepartie, UML permet de définir tout type de contrainte en langage naturel (graphiquement, il s'agira d'un texte encadré d'accolades s'appliquant au niveau d'un attribut, d'un rôle, ou entre associations nommées). Les contraintes qui s'appliquent entre plusieurs éléments doivent être précisées à l'aide d'une ligne pointillée (exemple 1-40) éventuellement assortie d'une note (exemple 1-41).

Nous étudions les contraintes de partition (P), d'exclusivité (X), de totalité (T), d'inclusion (I) et de simultanéité (S). Nous verrons qu'il est possible d'appliquer les contraintes P, X et T aux entités, classes ou associations, alors que les contraintes I et S seront dédiées aux associations.

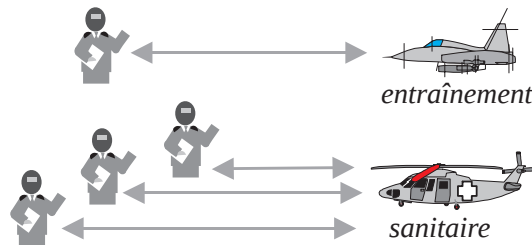
Considérons chaque contrainte sur l'ensemble des pilotes qui partent soit en mission sanitaire, soit en mission d'entraînement. Un pilote est caractérisé par un numéro, un nom et un grade. Une mission sanitaire est désignée par un code et un nom d'organisme, une mission d'entraînement par un numéro, une date et une région concernée.

Contrainte de partition



Selon la contrainte de *partition*, toutes les occurrences d'une entité (ou classe) participent à l'une des deux associations, mais pas aux deux, ni à aucune des deux.

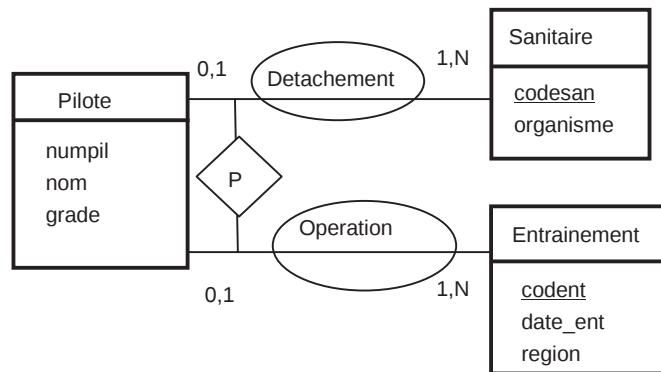
L'union des pilotes en mission sanitaire et en opération d'entraînement donne la totalité du contingent des pilotes (aucun n'est au repos ni ne mène de front des missions des deux types).

Figure 1-36 Exemple de contrainte de partition

En conséquence, il existe deux partitions dans la population des pilotes. Mathématiquement cet état de fait est formalisé par l'opérateur *ou exclusif*.

Merise/2

La notation 1-37 est celle de G. Panet [PAN 94]. L'AFCEt avait proposé une autre notation reprise par Win'Design comme le montre la figure 1-38. Le symbole XT rappelle que la partition est une combinaison de la totalité et de l'exclusivité. Notons l'existence de deux CIF générées par l'outil. Le pointillé indique par rapport à quelle(s) entité(s) s'exprime la contrainte.

Figure 1-37 Contrainte de partition Merise/2

Une contrainte peut concerner plusieurs entités (figure 1-39). Ici, on exprime le fait qu'une même personne ne peut être à la fois propriétaire et locataire d'un même logement.

Notation UML

La contrainte de partition existe avec UML (contrainte prédéfinie dans la spécification) et se note à l'aide d'une ligne en pointillé qui traverse deux liens d'association et du mot {xor} (voir figure 1-40). Il est aussi possible de l'exprimer à l'aide du langage OCL (Object Constraint

Figure 1-38 Contrainte de partition sous Win'Design

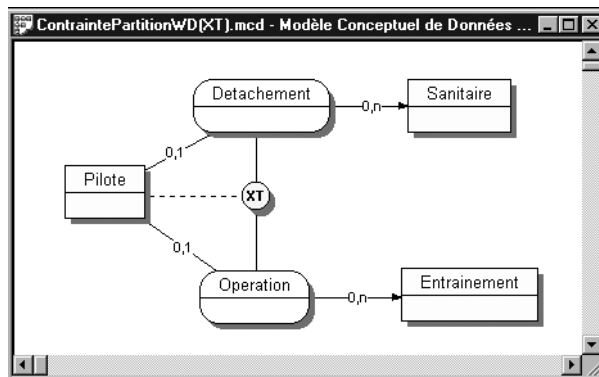


Figure 1-39 Contrainte d'exclusivité sous Win'Design

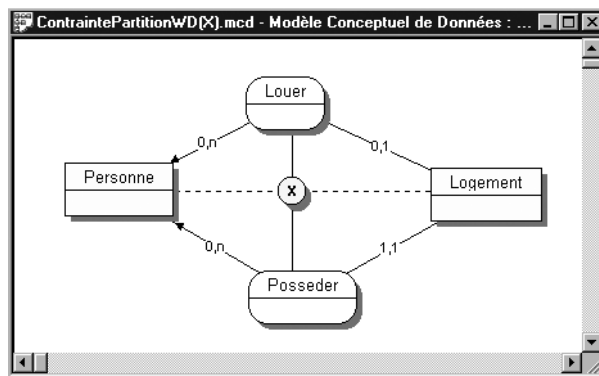
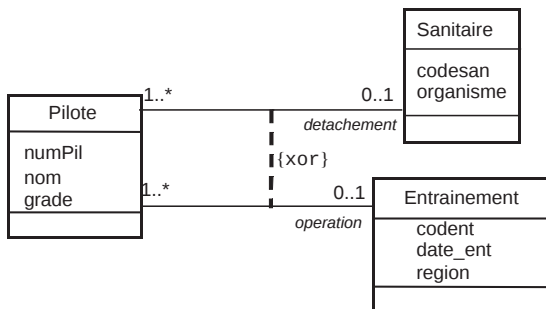


Figure 1-40 Contrainte de partition avec UML



Language). La programmation suivante, disposée dans une note et rattachée à la classe **Pilote** serait une alternative à l'expression {xor}. Notez l'utilisation des rôles pour appliquer les fonctions d'existence : `isEmpty()` qui retourne vrai si l'ensemble est vide et son inverse `notEmpty()`.

```
inv:
((detachement->isEmpty() and operation->notEmpty()) or
 (detachement->notEmpty() and operation->isEmpty()))
```

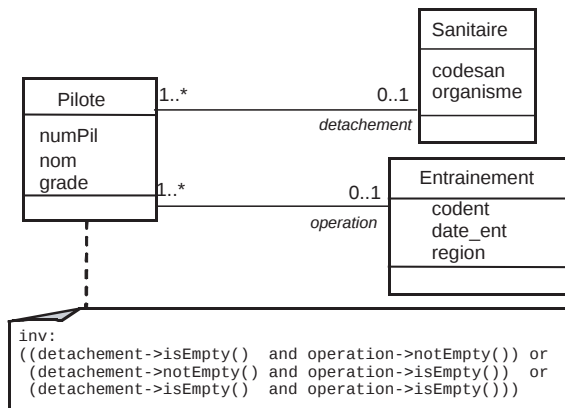
Contrainte d'exclusivité



Selon la contrainte d'*exclusivité*, toutes les occurrences d'une entité (ou classe) peuvent participer à l'une des deux associations, mais pas aux deux à la fois.

Un pilote peut être au repos (il n'est affecté à aucune mission). Si un pilote est affecté à un exercice d'entraînement, alors il ne peut pas être affecté à une mission sanitaire et réciproquement.

Figure 1-41 Contrainte d'exclusivité avec UML



Merise/2

Il faut remplacer la lettre P par la lettre X dans le schéma 1-37.

Notation UML

L'exemple 1-41 décrit la note qu'il faudrait programmer à l'aide du langage OCL (d'autres possibilités de programmation existent).

Contrainte de totalité



Selon la contrainte de *totalité*, toutes les occurrences d'une entité (ou classe) participent au moins à une association.

Considérons :

- qu'un pilote peut être affecté simultanément à une mission sanitaire et à un exercice d'entraînement ;
- que tous les pilotes participent au moins à une mission.

On peut dire ainsi que les pilotes forment la totalité du contingent.

Merise/2

Il faut utiliser la lettre T dans le schéma 1-37.

Notation UML

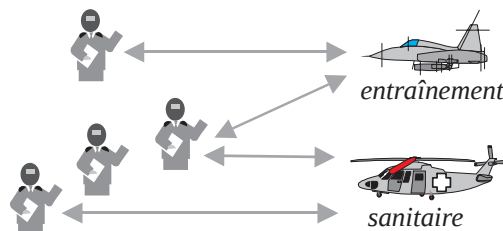
La note à programmer et à rattacher à la classe `Pilote` (de la même manière qu'à l'exemple 1-41) est la suivante.

```
inv:
((detachement->isEmpty() and operation->notEmpty()) or
 (detachement->notEmpty() and operation->isEmpty()) or
 (detachement->notEmpty() and operation->notEmpty()))
```

Absence de contrainte

Dans notre exemple, l'absence de contrainte (illustrée à l'exemple 1-42) signifie qu'il peut exister des pilotes n'étant affectés à aucune mission ou participant simultanément à une mission sanitaire et à un exercice d'entraînement.

Figure 1-42 Exemple sans contrainte



Le schéma Merise serait identique à l'exemple 1-38 sans contrainte. Le diagramme UML serait identique à l'exemple 1-41 sans l'utilisation de la note.

Contrainte d'inclusion



Selon la contrainte d'*inclusion*, toutes les occurrences d'une association doivent être incluses dans les occurrences d'une autre association.

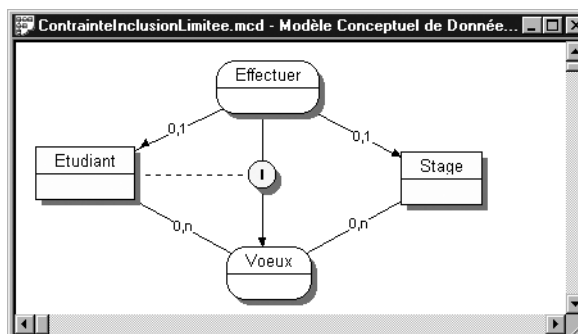
Entre deux associations binaires Merise/2

Des étudiants émettent des vœux concernant des stages sachant qu'ils doivent suivre un seul stage. Un stage peut intéresser aucun ou plusieurs étudiants. Pour modéliser le fait que le stage effectué par un étudiant doit être un stage figurant dans ses vœux, il faut ajouter une contrainte d'inclusion.

Win'Design utilise un lien en pointillé entre la contrainte et l'entité principale (figure 1-43). Cela indique que toute occurrence du couple (Etudiant, Stage) présent dans l'association Effectuer devra appartenir également à l'association Vœux.

Entre trois associations binaires Merise/2

Figure 1-43 Contrainte d'inclusion avec Win'Design



Considérons des départements qui achètent des logiciels pour lesquels on désire conserver la date d'achat. Par ailleurs, un serveur héberge des logiciels et un serveur peut être utilisé par différents départements. Cet état de fait nécessite trois associations binaires (Achat, Installe et Utilisation) ainsi qu'une contrainte d'inclusion entre ces trois associations. Grâce à cette contrainte, on pourra exprimer qu'un logiciel *l* acheté par le département *d* soit installé sur un serveur *s*, utilisé, entre autres, par ce département.

Considérons à présent les occurrences d'entités suivantes : deux départements (D1 et D2), quatre logiciels (L1, L2, L3 et L4) et trois serveurs (S1, S2 et S3). Supposons les occurrences d'associations suivantes :

- le département D1 achète les logiciels L1, L2 et L4 ;
- le département D2 achète le logiciel L1 ;

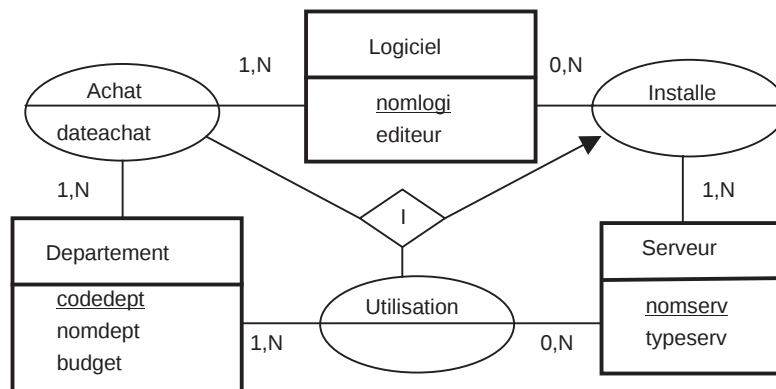
- le département D1 utilise les serveurs S1 et S2 ;
- le département D2 utilise le serveur S3.

Figure 1-44 Représentation tabulaire des occurrences

Achat		Utilisation	
Dept.	Logiciel	Dept.	Serveur
D1	L1	D1	S1
D1	L2	D1	S2
D1	L4		
D2	L1	D2	S3

À présent, considérons les installations de logiciels. La contrainte d'inclusion assure la cohérence des trois associations binaires. Par exemple les logiciels L1 et L4 ne peuvent pas être installés sur n'importe quel serveur. En effet, seuls les serveurs S1 ou S2 peuvent héberger L4 (car seul le département D1 a acheté ce logiciel). En revanche, les trois serveurs peuvent héberger L1 acheté par les deux départements. Représentons la contrainte d'inclusion 1-45.

Figure 1-45 Contrainte d'inclusion entre trois associations binaires Merise/2



Cette contrainte peut être formulée de la manière suivante : les occurrences de l'association *Installe* doivent être incluses dans les occurrences issues de la jointure entre les associations *Achat* et *Utilisation*.

La différence entre l'exemple 1-45 et celui qui concerne l'association *n*-aire *Installation* (figure 1-24) réside dans le fait que la modélisation des trois associations binaires ne permet pas de savoir quel est le département qui a eu l'initiative d'installer le logiciel sur le serveur.

Figure 1-46 Représentation tabulaire d'occurrences

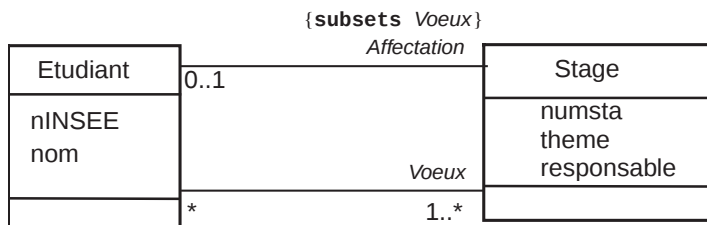
Installe		Jointure entre Achat et Utilisation	
Logiciel	Serveur	Logiciel	Serveur
L1	S1, S2 ou S3	L1	S1
L1		L1	S2
L2		L2	S1
L2		L2	S2
L4		L4	S1
L4		L4	S2
L1		L1	S3

Notation UML

La spécification UML 2 propose la notation `{subsets nom_rôle}` pour exprimer une contrainte d'inclusion. Alors que la spécification présente des exemples basés sur deux associations binaires (figure 1-47), nous verrons qu'il est possible d'étendre cette notation à plus de deux associations (figure 1-48) et d'utiliser les classes-associations pour celles qui concernent les associations *n*-aires (section *Affinage des associations n-aires*).

Entre deux associations binaires

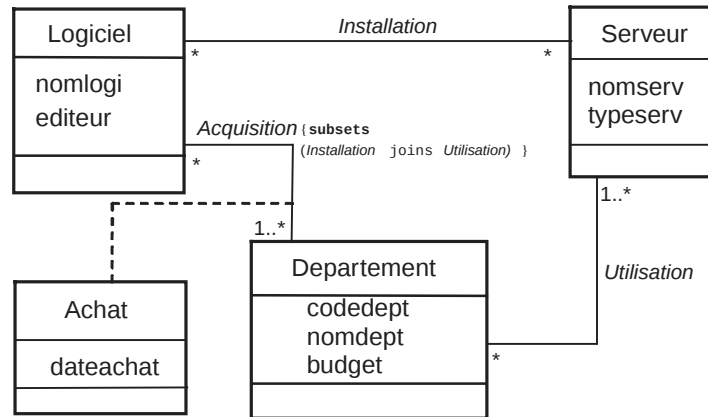
La contrainte d'inclusion entre les associations binaires *Affectation* et *Vœux* se traduit par l'enrichissement du rôle indiquant le sens de l'inclusion : ici *Affectation* est inclus dans *Vœux*.

Figure 1-47 Contrainte d'inclusion entre deux associations binaires UML

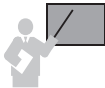
Entre trois associations binaires

La contrainte d'inclusion entre trois associations binaires peut se traduire de manière analogue à l'aide de l'enrichissement du rôle qui est inclus dans la jointure des deux autres.

Figure 1-48 Contrainte d'inclusion entre trois associations binaires avec UML



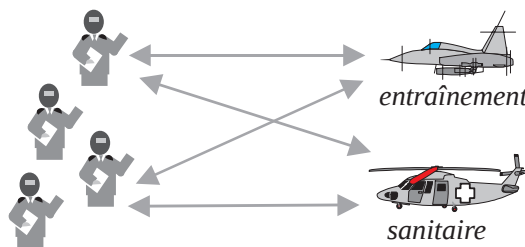
Contrainte de simultanéité



Selon la contrainte de *simultanéité* entre plusieurs associations, toute occurrence d'une entité (ou classe) liée à une association participe également aux autres.

Comme l'illustre l'exemple 1-49, un pilote peut être au repos, ou, s'il est affecté à un exercice d'entraînement, il doit aussi être affecté à une mission sanitaire et réciproquement.

Figure 1-49 Exemple de contrainte de simultanéité



Merise/2

Pour caractériser la simultanéité entre les associations Detachement et Operation de la figure 1-37, il faudrait disposer la lettre S dans le symbole de la contrainte.

Notation UML

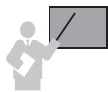
Bien que la contrainte de simultanéité ne soit pas proposée par UML, on la programme à l'aide de l'invariant OCL suivant en l'attachant une note à la classe `Pilote` (de la même manière qu'à l'exemple 1-41).

```
inv:
((detachement->isEmpty() and operation->isEmpty()) or
(detachement->notEmpty() and operation->notEmpty()))
```

Les contraintes prédéfinies de UML 2

La spécification UML 2 définit quelques contraintes qui se notent à l'extrémité d'un lien d'association (au même niveau que le rôle). Nous les citons ici pour mémoire mais elles ne sont pas toutes intéressantes dans un contexte de bases de données.

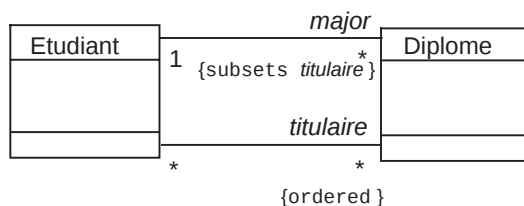
- `{subsets nom_rôle}` exprime l'inclusion d'une association par rapport à une autre (déjà étudié).
- `{redefined nom_rôle}` redéfinit un rôle.
- `{union}` signifie que le rôle rassemble une union de sous-ensembles.
- `{ordered}` exprime un ordre au niveau des objets reliés.
- `{bag}` exprime qu'un même objet cible peut apparaître plusieurs fois dans l'association.
- `{sequence}` notée aussi `{seq}` combine les caractéristiques de `bag` et de `ordered`.
- `{xor}` qui indique le *ou exclusif* entre objets reliés par deux associations (déjà étudié).



En l'absence de ces contraintes (cas le plus général et celui par défaut), la terminaison d'une association est dite de type *set* (chaque objet relié n'est présent au plus qu'une seule fois dans l'association).

Dans l'exemple 1-50, `ordered` précise l'ordre d'obtention d'un diplôme (chronologique par exemple), alors que `subsets` indique que, pour être major d'un diplôme, l'étudiant doit aussi y être associé en tant que titulaire.

Figure 1-50 Contraintes prédéfinies UML

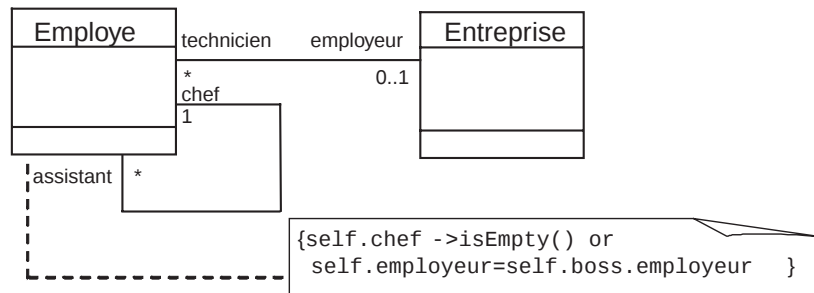


Contrainte personnalisée avec OCL

Pour toute autre contrainte, OCL est utilisable dans une note. L'exemple 1-51 décrit la contrainte qui exprime, à l'aide du langage OCL, que tout employé est soit :

- chef, et dans ce cas il n'a pas de chef (condition isEmpty vérifiée) ;
- sous la responsabilité d'un chef qui doit être employé dans la même entreprise que lui (deuxième partie de la condition).

Figure 1-51 Contrainte UML 2 avec OCL



Toute contrainte prédéfinie par UML2 ou exprimée à l'aide du langage OCL sera à programmer par la suite sous SQL soit par contrainte de vérification (CHECK), soit par déclencheur.

Affinage des associations n-aires

Il est recommandé d'affiner vos associations *n*-aires de façon à préciser la sémantique du schéma et à améliorer l'intégrité de la base de données. Les concepteurs qui n'utilisent pas ces concepts produisent en fait bien plus de programmation (qui pourrait être allégée par des clés étrangères induites par l'adoption de techniques que nous allons présenter).

Pour préciser une association *n*-aire, vous pouvez soit :

- utiliser une ou plusieurs contraintes d'unicité ;
- utiliser une ou plusieurs contraintes d'inclusion ;
- combiner ces deux types de contraintes.

Avec un formalisme de type entité-association, on utilise une CIF (ou une association d'agrégation). Sous UML, on peut recourir aux multiplicités et aux classes-associations. Dans les exemples qui suivent, nous ne considérons que des associations 3-aires. Il est bien sûr possible d'appliquer ce raisonnement à des associations de degré supérieur.

Contrainte d'unicité sur une association n -aire

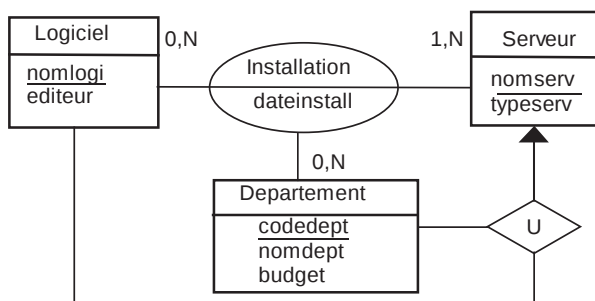
Merise/2



La contrainte d'*unicité* sur une association n -aire indique que pour toute occurrence de $(n-1)$ entité(s) il y a une seule occurrence de l'autre entité.

Revenons à l'installation des logiciels sur des serveurs réalisée sur l'initiative de départements. Ajoutons la contrainte qu'un logiciel acheté par un département ne doit être installé que sur un seul serveur. Avec le formalisme Merise, l'association 1-27 n'exprime pas de contrainte d'unicité. La contrainte qu'il convient de définir sur l'association *Installation* indique que pour un couple (*logiciel*, *département*), un seul *serveur* est associé. La figure 1-52 décrit le formalisme Merise/2 de [PAN 94].

Figure 1-52 Contrainte d'unicité sur une association 3-aire Merise/2



L'autre possibilité consiste à utiliser une CIF sur le lien d'association connecté à l'entité concernée par l'unicité (ici *Serveur*), et terminer ce lien par une flèche. L'explication provient de la notion de dépendance fonctionnelle (étudiée à la section *Règles de validation* et au chapitre 2) : ici, la dépendance fonctionnelle induite par la contrainte d'unicité est $\text{nomlogi}, \text{codedept} \rightarrow \text{nomserv}$. Cette dépendance exprime le fait que pour un logiciel et un département donnés correspond au plus un serveur.

Notation UML



La contrainte d'*unicité* sur une association n -aire se note par une multiplicité maximale valant 1 quand l'association est modélisée par le symbole losange, sinon par une classe-association.

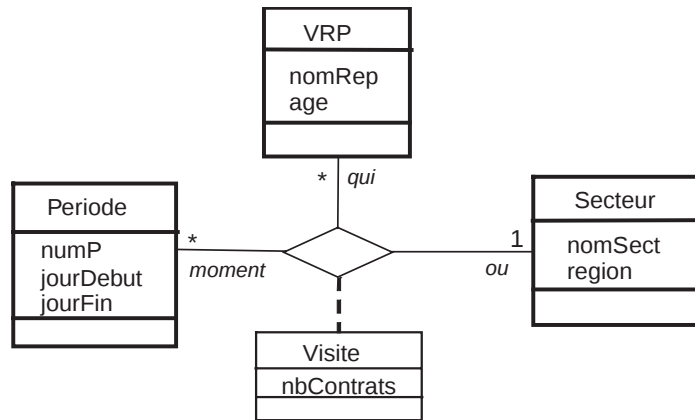


Il est préférable d'utiliser les classes-associations qui font apparaître moins d'éléments sur le diagramme. Par ailleurs, cette solution est mieux adaptée aux associations avec attributs et est plus proche de l'implémentation de la base de données.

Unicité par multiplicité

L'exemple 1-53 modélise la contrainte d'unicité exprimant qu'un représentant de commerce visite un seul secteur à une période donnée (période identifiée par un numéro et étant bornée entre deux dates). On désire connaître le nombre de contrats que le commercial enregistre lors de chaque visite.

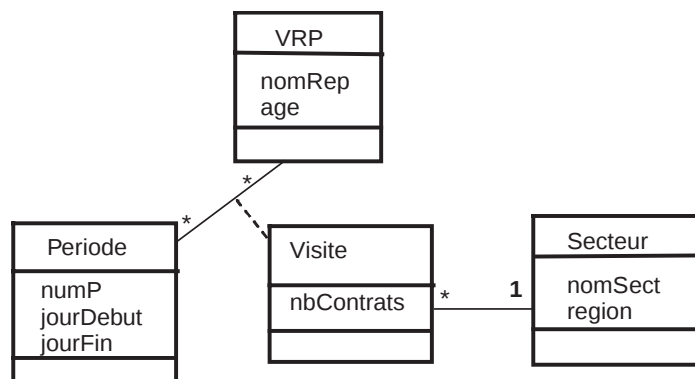
Figure 1-53 Contrainte d'unicité par multiplicité



Unicité avec une classe-association

Modélisons le cas précédent à l'aide de la classe-association 1-54. La classe-association et sa multiplicité 1 du côté de la classe Secteur modélise la contrainte d'unicité. Ce schéma est plus proche de l'implémentation que la solution précédente dans le sens où il est visible que la table Serveur doit jouer un rôle de clé étrangère pour identifier la colonne nbContrats alors que les tables VRP et Periode doivent jouer le rôle de clé primaire.

Figure 1-54 Contrainte d'unicité avec une classe-association



Contrainte d'inclusion sur une association n-aire

Nous verrons, sur la base d'exemples, qu'une contrainte d'inclusion sur une association *n*-aire peut être modélisée dans le formalisme entité-association par :

- une association d'associations, appelée aussi « pseudo-entité » (car il s'agit d'entité sans identifiant mais reconnue à l'aide des entités connectées à l'association) [ACS 90] ou « association d'agrégation » [CHR 87] (car il s'agit de relier une entité à un groupement d'autres entités).
- une entité faible et par l'identification relative, c'est le choix de la méthode Merise et des outils du marché.

Association d'association

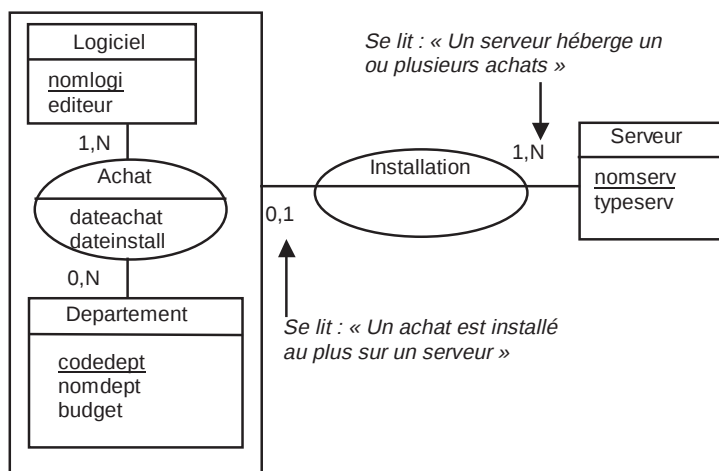


Aussi appelée « association d'agrégation » dans le modèle entité-association, l'association d'associations n'a pas de rapport direct avec le concept d'agrégation de la notation UML et que nous détaillerons ultérieurement. En revanche, l'association d'agrégation du modèle entité-association est pratique car analogue aux classes-associations de la notation UML. Le seul bémol est que la majorité des outils ne la programment pas.

Nous décrirons à la section *Associations d'agrégation* l'analogie avec UML, et dans le chapitre suivant, les moyens de dériver un modèle logique de données en fonction des multiplicités.

Revenons aux installations de logiciels (figures 1-24 et 1-52) et ajoutons la contrainte d'inclusion suivante : *une installation est valide à condition que le département ait acheté le logiciel*.

Figure 1-55 Association d'agrégation



Le diagramme 1-55 décrit à la fois :

- La contrainte d'unicité à l'aide du couple de cardinalités (0, 1) entre l'association Achat appelée « agrégat » et l'entité Serveur (il s'agit de rendre compte qu'un logiciel d'un département doit être installé sur un seul serveur) – définition d'agrégat selon le Larousse : « substance, masse formée d'éléments primitivement distincts, unis intimement et solidement entre eux ».
- La contrainte d'inclusion à l'aide de l'association d'agrégation Installation qui relie l'entité Serveur à l'agrégat.

Si la contrainte d'unicité n'avait pas lieu d'être, les cardinalités devraient être (0, N) du côté de l'agrégat.

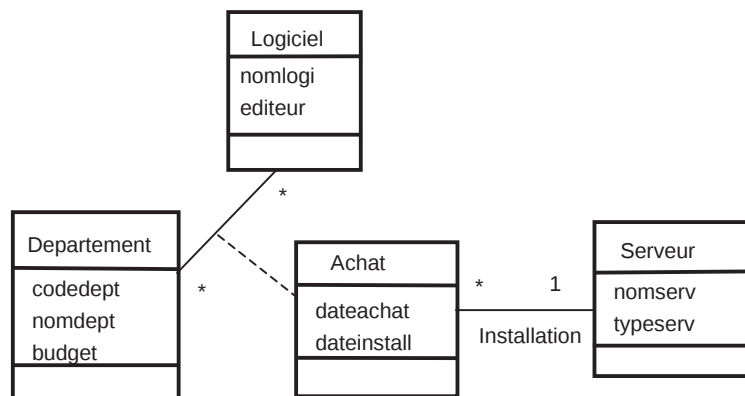
Notation UML



UML propose le concept de classe-association qui convient parfaitement à la modélisation de contrainte d'inclusion sur une association *n*-aire.

L'exemple 1-56 décrit la contrainte d'inclusion par l'existence même de la classe-association Achat. Notons que ce diagramme inclut aussi une contrainte d'unicité de par la multiplicité 1 du côté de la classe Serveur. Si cette contrainte n'avait pas lieu d'exister, la multiplicité deviendrait *.

Figure 1-56 Contrainte d'inclusion avec UML



Entités faibles de Merise

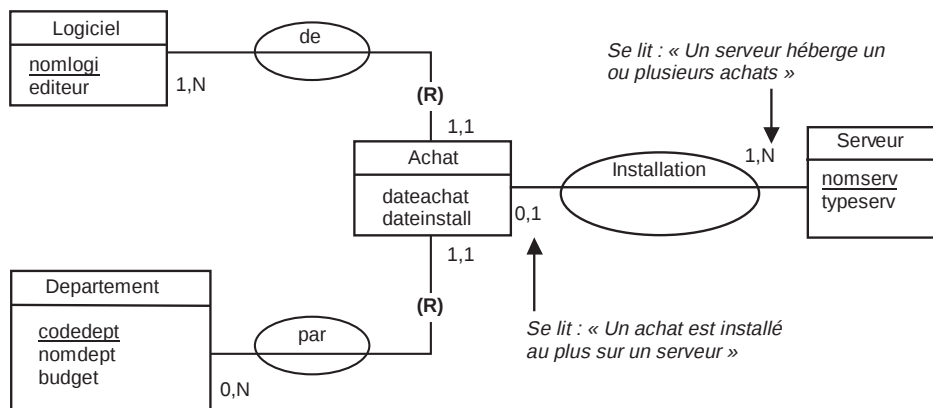


Une entité faible est formalisée comme une entité mais son identification s'effectue en totalité par rapport aux entités de sa collection (comme une association).

Dans notre exemple, le passage du verbe « acheter » au nom « achat » est un peu révélateur de la nécessité d'utiliser une entité faible. Ce phénomène a été appelé réification. Nous détaillerons plus loin la problématique d'identification et son incidence sur la réification qu'a développée D. Nanci à ce sujet.

L'exemple 1-57 illustre l'entité faible Achat identifiée à la fois par les attributs `nomlogi` et `codedept`. Différents formalismes ont été proposés pour caractériser le lien d'identification des entités faibles, nous présentons celui de l'outil Win'Design (symbole R).

Figure 1-57 Entité faible Achat



Associations d'agrégation

[SMI 77] définit le concept d'agrégation ainsi : « l'agrégation est une abstraction qui transforme une association entre objets en un objet agrégé ». Pour information, selon le Larousse : « *agrégation* : action d'agréger, de réunir des éléments distincts pour former un tout homogène ; fait de s'assembler ».



L'agrégation n'a pas vraiment la même signification entre les formalismes entité-association et la notation UML. Pour les premiers, il s'agit de préciser des associations *n*-aires, pour la notation UML, l'agrégation renforce le couplage entre classes et peut s'appliquer à tout type d'association (notamment binaire).

Modèle entité-association

Bien qu'il n'existe pas d'outil qui prenne en compte le concept d'association d'agrégation (au sens de l'exemple 1-55), il est intéressant de bien comprendre ce concept car il est très analogue aux classes-associations de la notation UML. Nous expliquerons, dans les chapitres suivants,

comment transformer une association d'agrégation au niveau logique puis en tables SQL2 ou SQL3.

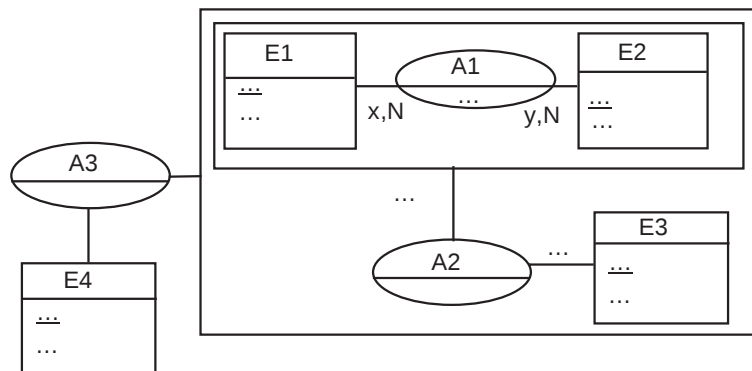
Association d'association



Une association d'agrégation rattache une association *plusieurs-à-plusieurs* à une (ou plusieurs) entité(s) ou à une autre association d'agrégation.

L'agrégat 1-55 est composé de deux entités et l'agrégat 1-58 est composé de trois entités. L'association d'agrégation (A2) forme elle-même un agrégat relié à une autre association d'agrégation (A3). On peut généraliser ainsi ce raisonnement à tout degré supérieur.

Figure 1-58 Agrégat à 3 entités



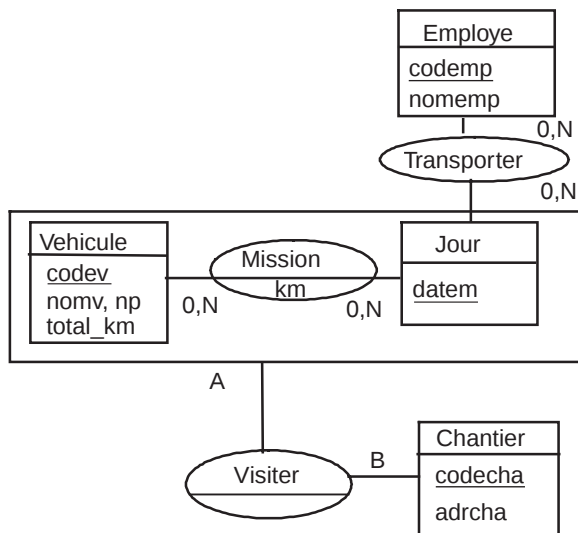
L'exemple 1-59 met en œuvre l'association d'agrégation *Visiter* reliant les entités *Vehicule*, *Jour* (modélisée comme *jour/mois/année*) et *Chantier*. Raisonnons à l'inverse du processus de conception en déterminant les contraintes en fonction des cardinalités maximales. Le tableau 1-6. décrit la classification obtenue.

Tableau 1.6 Cardinalités d'une association d'agrégation

Hypothèse	A	B
1	1,N	0,1
2	1,N	0,N
3	0,1	0,N

Les contraintes déduites de chaque hypothèse sont les suivantes :

- Hypothèse 1 : un chantier sera visité une seule fois par une mission (couple *véhicule-date*). En revanche, plusieurs missions pourront visiter plusieurs chantiers. En d'autres termes, la

Figure 1-59 Possibilités de cardinalités dans une association d'agrégation

base de données ne pourra stocker que la date de la dernière visite pour chaque chantier, un véhicule pouvant sortir à différentes dates et visiter différents chantiers.

- Hypothèse 2 : un chantier peut être visité le même jour par différentes missions. Par ailleurs, une mission ne pourra pas visiter le même jour plus d'une fois le même chantier.
- Hypothèse 3 : un chantier peut être visité le même jour par différentes missions. En revanche, une mission ne pourra pas visiter le même jour un autre chantier. En d'autres termes, la base de données ne pourra stocker au plus qu'une visite par mission.

Entités faibles de Merise/2

Dans cet exemple, il est révélateur que l'association soit appelée **Mission**, elle est devenue mentalement un objet (dont on peut parler, que l'on peut dénombrer... et qui est devenu un sujet dans les phrases), toutefois identifiée par le couple (codev , datem). Pour Merise/2, il est naturel que cette association soit représentée en tant qu'entité faible.

L'exemple 1-60 illustre le MCD de l'hypothèse (1). L'entité faible n'a pas d'identifiant mais « hérite » des identifiants des entités concernées (**Vehicule** et **Date**). Les liens sont marqués de (R) qui signifie le caractère relatif de l'identification.

Une autre modélisation de l'hypothèse (1) est illustrée à la figure 1-61, elle fait abstraction de l'entité **Date** qui n'est pas nécessaire tout en conservant la propriété représentant le jour.

À titre de comparaison, l'exemple 1-62 décrit, avec le formalisme Merise/2 de Win'Design, le cas général illustré à la figure 1-58. PowerAMC utilise une autre forme de notation.

Figure 1-60 Modélisation par une entité faible

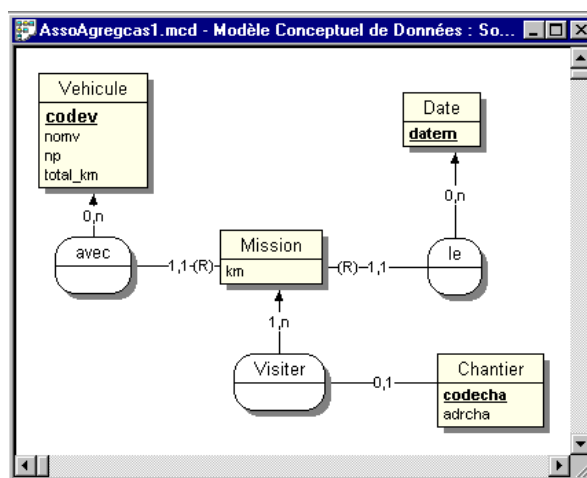


Figure 1-61 Modélisation simplifiée par une entité faible

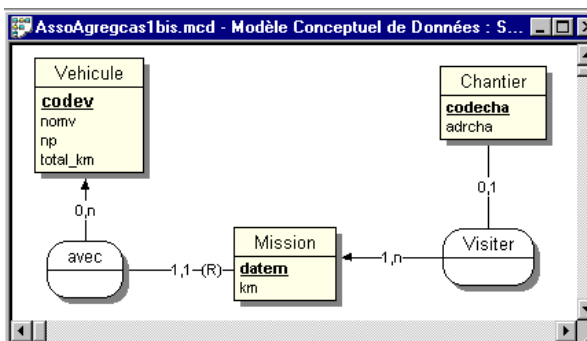
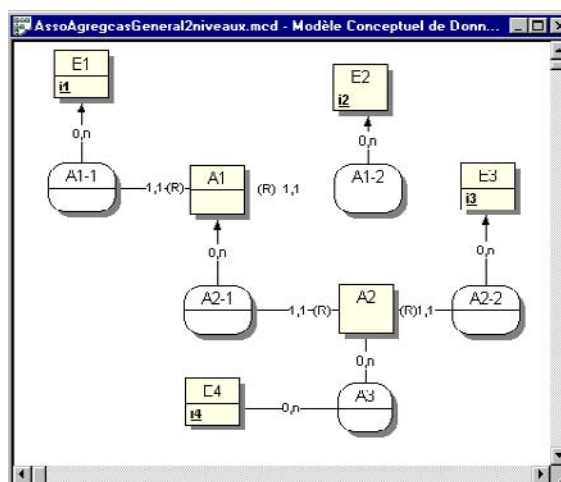


Figure 1-62 Modélisation du cas général des associations d'agrégation sous Win'Design



Notation UML

Les agrégations d'UML représentent des associations qui ne sont pas symétriques et pour lesquelles l'une des extrémités joue un rôle prédominant par rapport à l'autre. L'agrégation concerne seulement les associations binaires (réflexives ou non). Il est préférable d'utiliser une agrégation lorsqu'une classe fait partie d'une autre classe ou lorsqu'une action sur une classe implique une action sur une autre classe [MUL 00].

La notion d'agrégation a été l'un des aspects les plus discutés de la notation UML [KET 98]. Plusieurs types d'associations peuvent être recensés, [MAR 98] en liste six (*component-integral*, *material-object*, *portion-object*, *place-area*, *member-brunch* et *member-partnership*).



UML définit deux formes d'agrégation :

- La *composition* (*composite aggregation*) requiert qu'un objet appartienne au plus à une composition d'objets à un instant donné. Cette appartenance peut changer au cours du temps. La composition implique en outre une forme de propagation entre le composite et le composant (en particulier la destruction du composite entraînera obligatoirement la destruction de ses composants). Cette agrégation est représentée par un *losange noir* du côté du rôle de la classe composite appelée « agrégat ».
- L'*agrégation partagée* autorise qu'un objet appartienne simultanément à différentes compositions d'objets. Cette agrégation est représentée par un *losange clair* du côté du rôle de la classe concernée par l'association appelée « agrégat ».

L'agrégation de composition ne s'impose que lorsque l'association est de type *composite/composant* ou *fait partie de*.

Les agrégations partagées sont un peu plus difficiles à définir, elles renforcent le couplage d'une association binaire et interviennent lorsque la composition ne s'applique pas et que des objets sont fortement dépendants par rapport à d'autres objets dans le cadre de l'association.

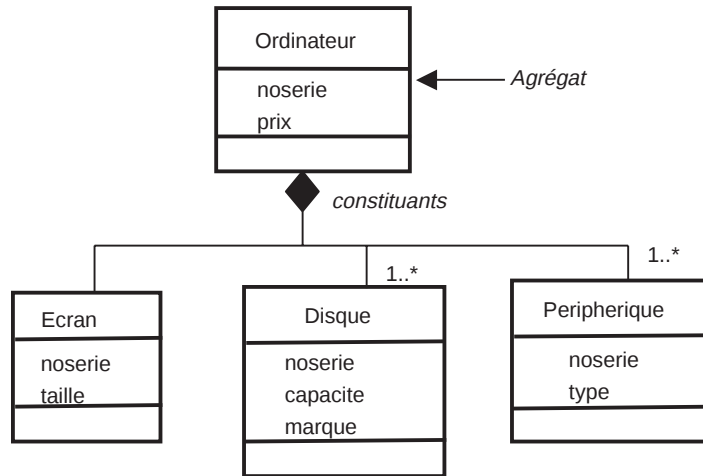
Composition

Modélisons à l'exemple 1-63 un ordinateur qui est composé d'un écran, d'un ou de plusieurs disques et de différents périphériques en utilisant une agrégation de composition. Les composants (objets des classes Ecran, Disque et Périphérique) et les composites (objets de la classe Ordinateur) sont ici considérés comme des pièces réelles. La destruction d'un ordinateur entraînera la destruction de ses composants.



Au niveau de la base de données, la composition se programme avec SQL à l'aide de la propagation de l'action de suppression via les clés étrangères (ON DELETE CASCADE) ou par déclencheurs.

Figure 1-63 Composition avec UML

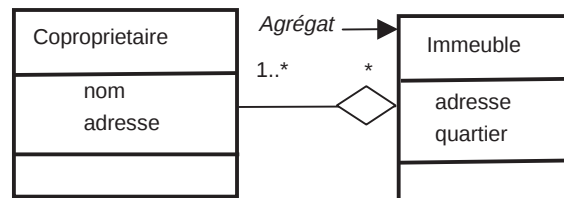


Agrégation partagée

Le losange clair indiquerait que les composants seraient considérés comme des types de pièces et non comme des pièces réelles. Chaque objet des classes **Ecran**, **Disque** et **Peripherique** pourrait ainsi être partagé entre différents objets de type **Ordinateur**. La destruction d'un ordinateur n'entraînerait pas nécessairement la destruction de ses composants.

Considérons l'exemple 1-64, l'existence d'un copropriétaire n'a de sens que si l'immeuble existe. Il ne s'agit pas ici d'une relation composite/composant. La suppression d'un objet de type immeuble n'entraînera pas forcément la destruction de ses copropriétaires (pour ceux qui sont copropriétaires dans d'autres immeubles).

Figure 1-64 Agrégation avec UML



Notion assez subjective, l'agrégation relève plus de la conception détaillée que de la modélisation. Elle se traduira au niveau physique par programmation de déclencheurs ou de contraintes SQL (de vérification CHECK ou de répercussion de clé étrangère CASCADE).

Contre-exemple

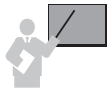
Il n'est pas nécessaire d'utiliser une agrégation pour décrire l'association binaire entre les postes de travail et les segments (figure 1-10). En effet, un poste de travail peut avoir une existence propre indépendamment d'un segment et réciproquement.

Règles de validation

Il est très intéressant d'appliquer les règles de validation inspirées des principes de normalisation du modèle relationnel au niveau du diagramme de classes UML. En l'absence de directives de la spécification UML en la matière, ces règles permettront d'assurer la cohérence de la base de données. Elles sont basées sur les propriétés des dépendances fonctionnelles que nous étudierons au chapitre suivant.

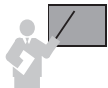
Ces règles préparent correctement le passage à SQL, en limitant les risques d'erreurs de modélisation lourdes de conséquences au niveau de la base de données. La méthode Merise avait à l'époque aussi proposé des règles, en amont des problèmes d'implémentation pour valider des MCD.

Caractère élémentaire d'un attribut



Tous les attributs sont élémentaires dans le sens où ils doivent être non décomposables (exemple de l'attribut *adresse* qui serait composé d'un numéro de rue, du nom de la rue, du code postal, etc. et pour lequel il faudrait définir autant d'attributs que de propriétés).

Vérification

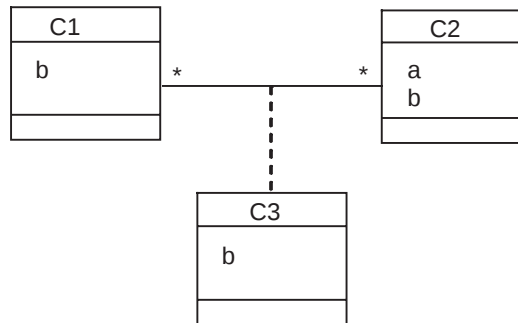


Non-redondance

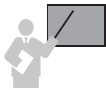
Un attribut doit apparaître une seule fois dans le diagramme, soit au sein d'une classe, soit dans une association (ou classe-association).

Dans l'exemple 1-65, l'attribut *b* apparaît à plusieurs endroits. Il convient de corriger cette modélisation pour ne garder qu'une occurrence en se posant la question : « de quoi dépend *b* ? ». Si *b* permet de caractériser la classe *C1*, il devra se trouver dans *C1*, même chose pour *C2*. S'il dépend à la fois des classes *C1* et *C2*, il devra alors se trouver dans *C3*.

Nous détaillerons plus formellement dans les chapitres suivants cette démarche.

Figure 1-65 Règle de non-redondance

Première forme normale



Première forme normale

Chaque attribut d'une classe, d'une association ou d'une classe-association possède au plus une seule valeur à un instant *t*.

Pour respecter cette normalisation, le schéma ne doit pas contenir d'attribut multivalué (ayant plusieurs valeurs, comme une liste), contrairement aux attributs monovalués qui contiennent au plus une valeur de tout type (chaîne de caractères, date, entier, etc.). Le problème c'est que UML permet de définir des attributs multivalués.



À moins que vous ne maîtrisiez le concept de collections (étudié au chapitre 3) et que vous desiriez l'appliquer à bon escient à une base de données objet-relationnelle, il est conseillé de ne pas utiliser d'attribut multivalué.

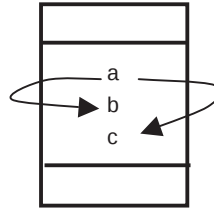
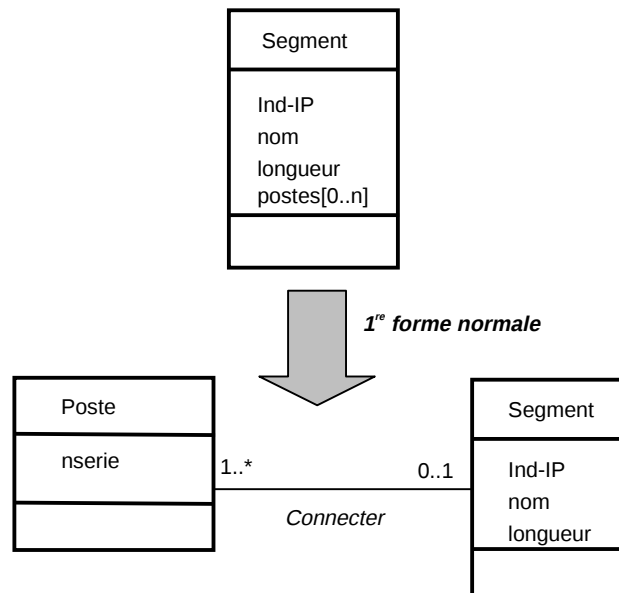
Rapport avec les dépendances fonctionnelles

Une classe est en première forme normale si chaque attribut est en dépendance fonctionnelle avec l'identifiant de la classe. En d'autres termes, à une valeur de l'identifiant, on a au plus une valeur de tout autre attribut de la classe.

Pour que le diagramme 1-66 (dans lequel on suppose que l'attribut *a* est identifiant) soit en première forme normale, il faut qu'à une valeur de *a* soit associé au plus une valeur de *b* et de *c*. Sinon, on est en présence d'attributs multivalués qu'il faut décomposer.

Exemple

Des postes de travail identifiés par un numéro sont connectés au réseau à travers des segments caractérisés par un indicatif IP, un nom et la longueur du câble. L'exemple 1-67 décrit le passage en

Figure 1-66 Première forme normale**Figure 1-67** Passage en première forme normale

première forme normale de la classe UML **Segment**. L'attribut multivalué **postes** est noté comme un tableau de numéros de postes qu'on décompose à l'aide de la classe de même nom.

Contre-exemple

Supposons, à partir de la figure 1-66, que *a* désigne l'immatriculation d'un avion, *b* son type et *c* la date du vol. Si on désire stocker l'historique des vols, cette représentation ne conviendra pas. En effet, pour une immatriculation donnée, plusieurs dates de vol peuvent être associées. Pour rendre ce schéma en première forme normale, l'attribut *b* reste bien placé et il faudrait déplacer l'attribut *c* dans une autre classe (ou dans une classe-association). Nous étudierons, au fil de cet ouvrage, des techniques de raisonnement pour déduire la bonne solution.

Deuxième forme normale



Deuxième forme normale

Chaque attribut d'une classe doit dépendre fonctionnellement de l'identifiant de la classe. Chaque attribut d'une classe-association doit dépendre simultanément des identifiants des classes connectées à la classe-association.

Rapport avec les dépendances fonctionnelles

Pour que le diagramme 1-68 soit en deuxième forme normale, il faut qu'à chaque occurrence du couple (a, b) soit associée au plus une valeur de c .

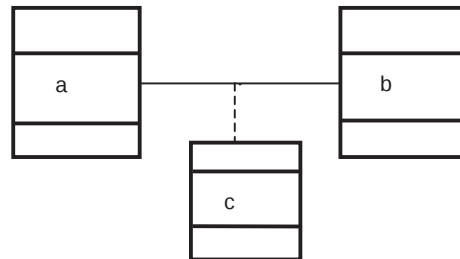


Figure 1-68 Deuxième forme normale

Exemple

Supposons que des serveurs soient des postes de travail connectés au réseau via différents segments. Le besoin est de stocker pour chaque serveur la date de dernière connexion à tout segment (dateCx), tout en conservant la date de première connexion (datePremCx) indépendamment du segment.

Le passage en deuxième forme normale est illustré par la figure 1-69. Le problème réside au niveau de l'attribut datePremCx qui ne dépend pas du segment et doit migrer dans la classe

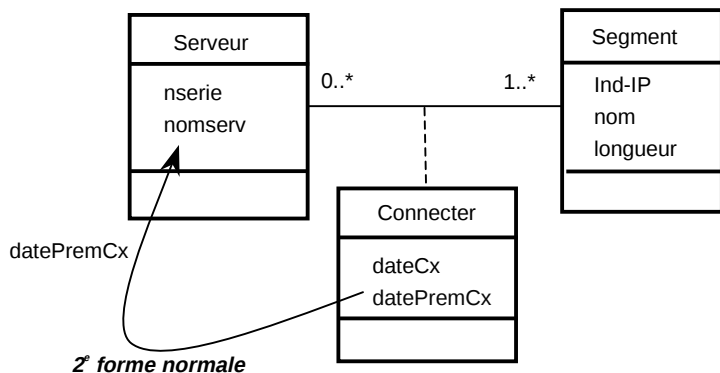


Figure 1-69 Passage en deuxième forme normale

2^e forme normale

Serveur. L'attribut `dateCx` qui dépend simultanément des deux classes doit rester dans la classe-association `Connecter`. L'association peut désormais se lire ainsi : *à un serveur et à un segment donnés est associée une date de connexion.*

Contre-exemple

Supposons, à partir de la figure 1-68, que l'attribut *a* représente l'immatriculation d'un avion, *b* le code de la compagnie propriétaire et *c* le type de l'avion. Le diagramme ne serait pas en deuxième forme normale car seule l'immatriculation d'un avion suffit à déterminer son type. Il faudrait déplacer l'attribut *c* dans la classe de *a* pour satisfaire à cette normalisation.

En revanche, si l'attribut *c* représente la date d'un vol, alors la modélisation exprime le fait qu'on puisse gérer les dates de derniers affrètements de chaque avion par toute compagnie.

Troisième forme normale



Troisième forme normale

Chaque attribut doit dépendre d'un identifiant dans le cas d'une classe, ou de plusieurs identifiants dans le cas d'une classe-association et non d'un autre attribut voisin, lui-même dépendant d'un ou de plusieurs identifiants.

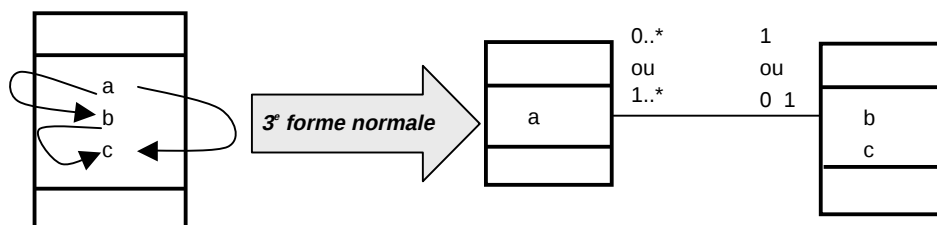
Bien qu'il existe parfois une volonté de dénormalisation (on se contente de la deuxième forme normale) à ce niveau pour optimiser les accès en diminuant les jointures dans les requêtes SQL, il est fortement conseillé de rendre, dans un premier temps, vos diagrammes de classes en troisième forme normale.

Rapport avec les dépendances fonctionnelles

Un diagramme est en troisième forme normale si chaque attribut de classe est en dépendance fonctionnelle directe avec l'identifiant de sa classe.

Dans l'exemple 1-70, il existe un graphe transitif de dépendance qu'il convient de rompre (ici, la dépendance entre *a* et *c* est déduite par transitivité des deux autres). Le fait de supprimer

Figure 1-70 Passage en troisième forme normale



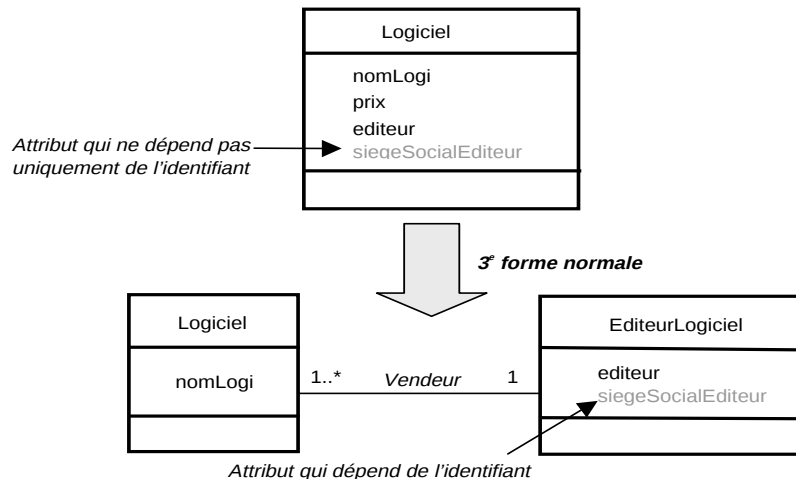
explicitement la dépendance entre *a* et *c* entraîne le déplacement de l'attribut *c*. Pour préserver la dépendance entre *b* et *c*, on dispose ces deux attributs dans une même classe. La dépendance entre *a* et *b* est préservée du fait de l'existence de la multiplicité maximale 1. La dépendance entre *a* et *c* se retrouve par navigation entre les classes.

Un autre exemple est donné à la figure 1-99, il concerne un graphe transitif à travers plusieurs associations.

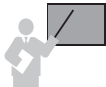
Exemple

Caractérisons un logiciel par son nom, son prix, le nom de son éditeur et le siège social de l'éditeur (exemple, le logiciel *MatLab* coûte 400 €, est édité par *MathWorks* dont le siège social se trouve à Sèvres). La figure 1-71 décrit le passage en troisième forme normale de la classe *Logiciel*. La transformation est nécessaire car le siège social dépend de l'éditeur (indirectement aussi du logiciel dont on suppose que chaque nom est unique).

Figure 1-71 Passage en troisième forme normale



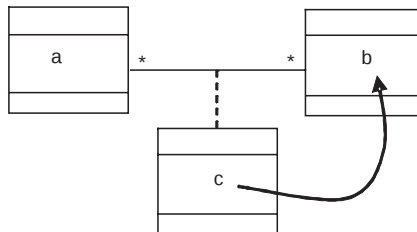
Forme normale de Boyce Codd



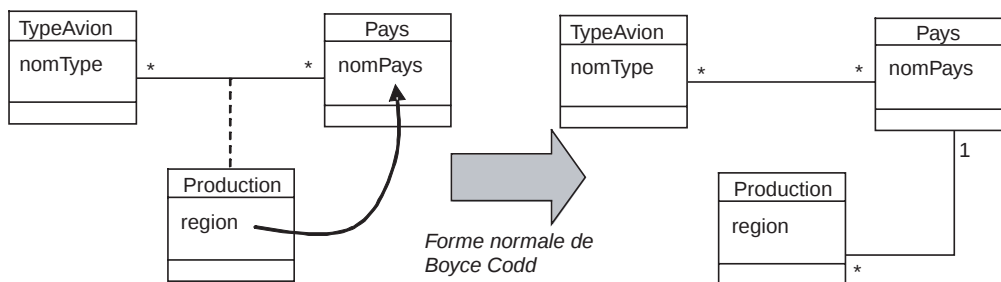
Forme normale de Boyce Codd

Aucun attribut d'une classe-association ne doit être en dépendance avec un identifiant d'une des classes connectée à l'association.

Nous étudions plus formellement cette forme au chapitre 2. L'exemple 1-72 illustre un cas dans lequel cette normalisation n'est pas respectée (l'attribut *c* permet de déterminer l'attribut *b*). Il convient de préserver cette dépendance tout en conservant l'association binaire entre les classes identifiées respectivement par *a* et *b*.

Figure 1-72 Cas de Boyce Codd

L'exemple 1-73 décrit le fait qu'un type d'avion soit produit dans un même pays, dans différentes régions (par exemple, en France, l'A320 est produit à Toulouse et Saint-Nazaire). La dépendance qu'il faut traiter réside dans le fait que chaque région identifie le pays. Une fois normalisée en forme de Boyce Codd, la dépendance fonctionnelle entre la région et le pays est préservée par l'existence de la multiplicité 1 du côté Pays.

Figure 1-73 Résolution de la forme normale de Boyce Codd

Nous verrons au chapitre 2 que cette transformation se réalise sans perte d'information.

Décomposition des *n*-aires

Ce processus décompose une association *n*-aire en plusieurs associations de degré inférieur sans perte de sémantique.



Trois cas imposent de décomposer une association *n*-aire en associations de degré inférieur.

- si l'association est modélisée par une classe stéréotypée ayant une multiplicité maximale valant 1. On obtient des associations binaires ;
- si l'association est modélisée par un losange relié à un lien de multiplicité maximale valant 1. On obtient une classe-association ;
- il existe une dépendance fonctionnelle entre identifiants des classes connectées à l'association.

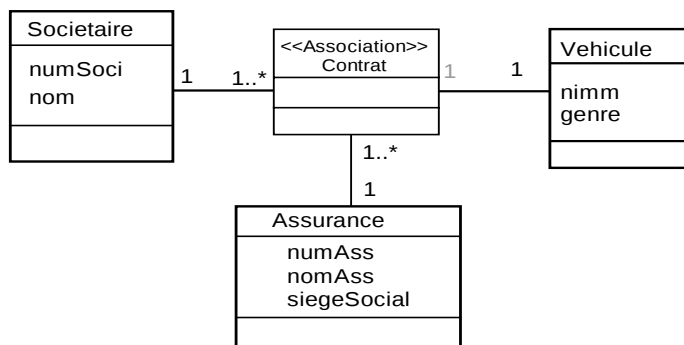
On obtient une classe-association.

Alors que les premiers et derniers cas n'ont pas lieu d'exister dans le formalisme Merise, la méthode française utilise le concept de CIF pour résoudre le dernier cas. Plusieurs notations sont possibles : l'association est décomposée ou la CIF est notée par un lien fléché avec le symbole CIF ou U (figure 1-52).

Décomposition d'une classe stéréotypée

L'exemple 1-74 modélise des sociétaires qui assurent leurs véhicules dans différentes compagnies d'assurances. Un véhicule n'est assuré que par une seule compagnie (multiplicité 1 du côté Contrat). La valeur de cette multiplicité impose la décomposition de l'association 3-aire en deux associations binaires.

Figure 1-74 Association par classe stéréotypée



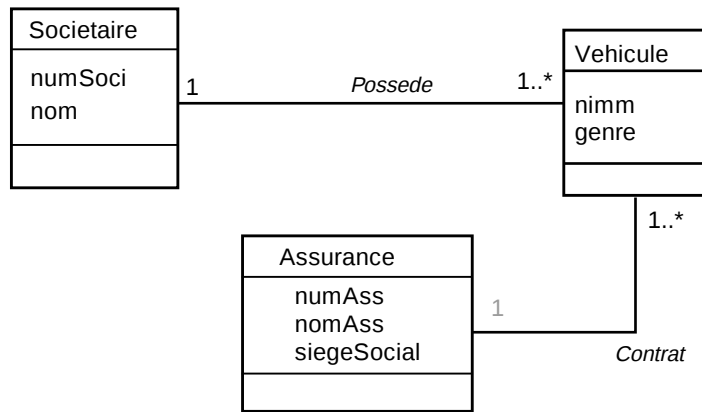
La décomposition 1-75 permet de réduire le degré de l'association Contrat devenue binaire et reliant Assurance à Vehicule. Il reste à relier Vehicule à Societaire pour modéliser l'appartenance.

Décomposition d'une association par losange

Nous avons vu à la figure 1-26 qu'une association n -aire modélisée à l'aide d'un losange relié à un lien de multiplicité maximale valant 1 exprimait une contrainte d'unicité. La transformation de cette association à l'aide d'une classe-association est décrite à la figure 1-28.



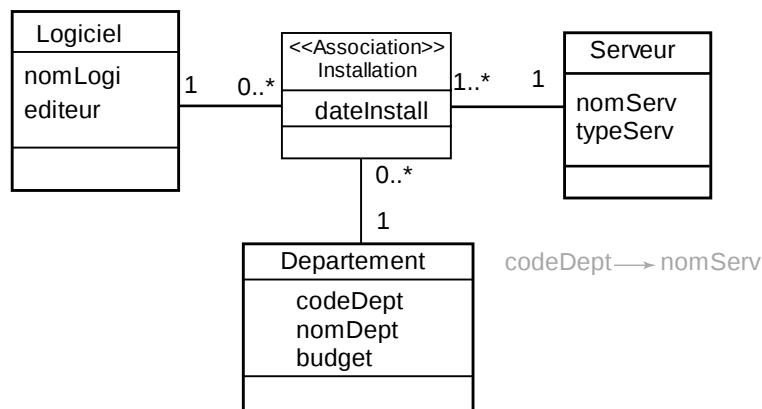
Si deux multiplicités d'une association n -aire modélisée à l'aide d'un losange valent 1, il faudrait, en théorie, généraliser le raisonnement précédent en ajoutant une nouvelle classe-association au schéma. En pratique, on n'utilise qu'une seule classe-association et cette nouvelle contrainte d'unicité pourra se programmer avec SQL à l'aide de la directive UNIQUE.

Figure 1-75 Décomposition de la classe stéréotypée

Décomposition par dépendance fonctionnelle

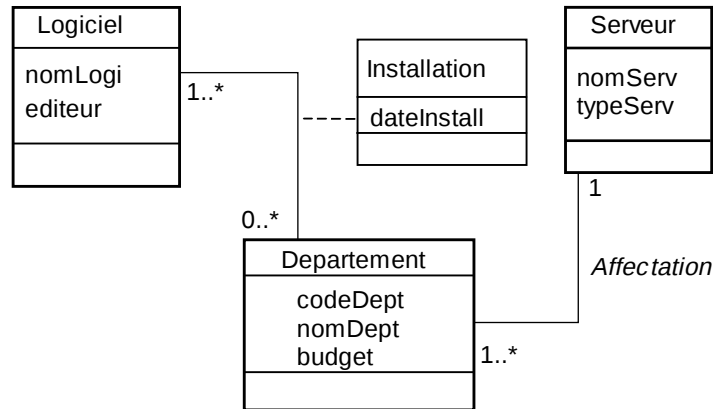
Le dernier cas de décomposition s'impose s'il existe une dépendance fonctionnelle entre identifiants de classes qui composent l'association n -aire.

L'exemple 1-76 suppose que chaque département est désormais lié à un seul serveur qui hébergera ses logiciels (existence de la dépendance fonctionnelle $\text{codeDept} \rightarrow \text{nomServ}$).

Figure 1-76 Association par classe stéréotypée et dépendance

La décomposition 1-77 réduit le degré de l'association Installation (devenue binaire avec classe-association du fait de l'existence de l'attribut `dateInstall`). Il reste à relier Département à Serveur avec la multiplicité 1 pour modéliser la dépendance fonctionnelle.

Figure 1-77 Décomposition de la classe stéréotypée



Héritage

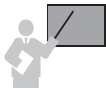
À l'origine, les modèles entité-association ne disposaient pas de ce concept. Des extensions comme Merise/2 ont rendu possible le mécanisme d'héritage qui permet d'organiser les entités en hiérarchies. UML propose l'héritage de classes depuis sa première spécification.



Il faut définir une association d'héritage entre les classes C1 et C2 si on répond affirmativement à la question suivante : *La classe C1 est-elle une sorte de C2 ?*

Les questions *La classe C1 est-elle composée de la classe C2 ?* ou *La classe C1 est-elle en rapport avec la classe C2 ?* permettent de déduire des agrégations ou de simples associations.

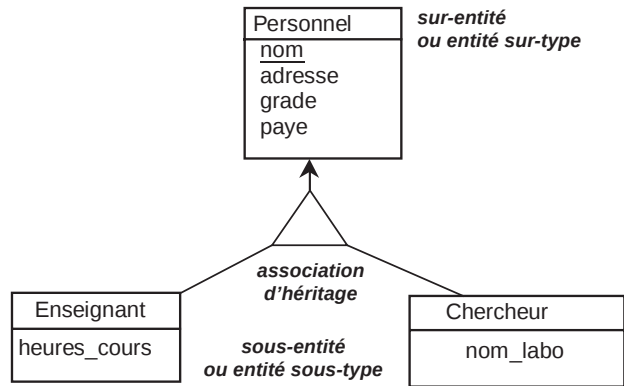
Formalisme



Merise/2 représente une association d'héritage sous la forme d'un triangle contenant une lettre en cas de contrainte.

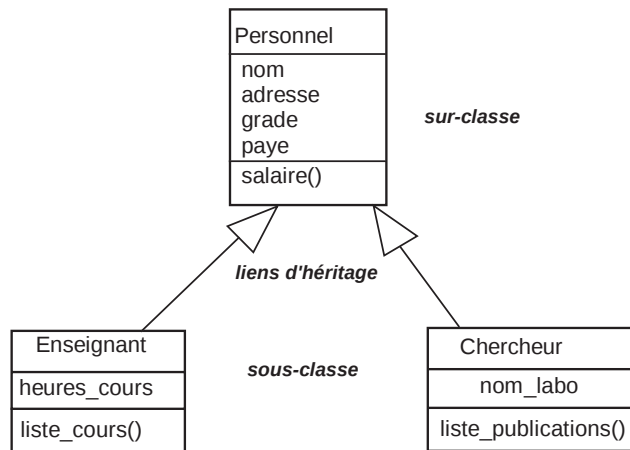
UML représente un lien d'héritage à l'aide d'une flèche partant de la sous-classe vers la sur-classe.

Considérons une population composée d'enseignants et de chercheurs faisant partie de la classe *Personne1*. L'exemple 1-78 décrit une hiérarchie d'héritage simple Merise/2 (deux sous-entités et une sur-entité).

Figure 1-78 Héritage avec Merise/2

Il est à noter qu'une sous-entité peut aussi être nommée « entité sous-type » et une sur-entité « entité sur-type ».

Le diagramme UML 1-79 met en œuvre la sur-classe `Personnel` et les deux sous-classes `Enseignant` et `Chercheur`.

Figure 1-79 Héritage avec UML

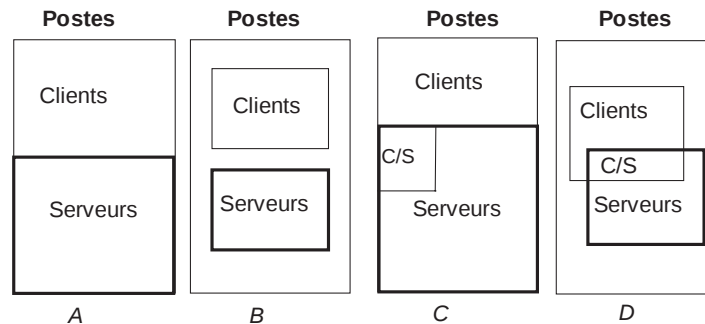
Différents cas d'héritage

Différents cas d'héritage peuvent être recensés en fonction des instances des classes [ADI 93]. L'exemple 1-80 décrit une population de postes de travail qui peut être composée :

- de postes clients ou serveurs exclusivement (cas A) ;

- de postes clients, de postes serveurs et de postes isolés qui ne sont ni clients ni serveurs (cas B) ;
- de postes clients, de postes serveurs et de postes qui sont à la fois clients et serveurs, mais pas de postes isolés (cas C) ;
- de postes clients, de postes serveurs, de postes qui sont à la fois clients et serveurs et de postes isolés (cas D).

Figure 1-80 Différents cas d'héritage



Chacun de ces cas d'héritage peut être modélisé à l'aide de contraintes que nous avons étudiées à la section *Contraintes*. Le tableau 1.7 fait correspondre, pour chaque cas d'héritage, la contrainte à programmer dans les deux formalismes.

Tableau 1.7 Héritage : couverture et disjonction

Couverture	Non-couverture
Disjonction (cas A) Merise/2 : Partition (XT) UML : {complete, disjoint}	(cas B) Merise/2 : Exclusivité (X) UML : {incomplete, disjoint} (par défaut)
Non-Disjonction (cas C) Merise/2 : Totalité (T) UML : {complete, overlapping}	(cas D) Merise/2 : pas de contrainte (par défaut) UML : {incomplete, overlapping}



Notez que l'absence de contrainte dans un graphe d'héritage n'a pas la même signification dans Merise/2 et pour UML. Le premier autorise qu'une occurrence appartienne à plusieurs entités sous-type. UML considère que par défaut un objet n'existe que dans une classe pour un niveau de hiérarchie donné.

Supposons, dans le cadre de notre exemple, qu'un poste de travail soit caractérisé par un numéro de série et un type. Un client sera désigné par une adresse IP et par un masque de sous-

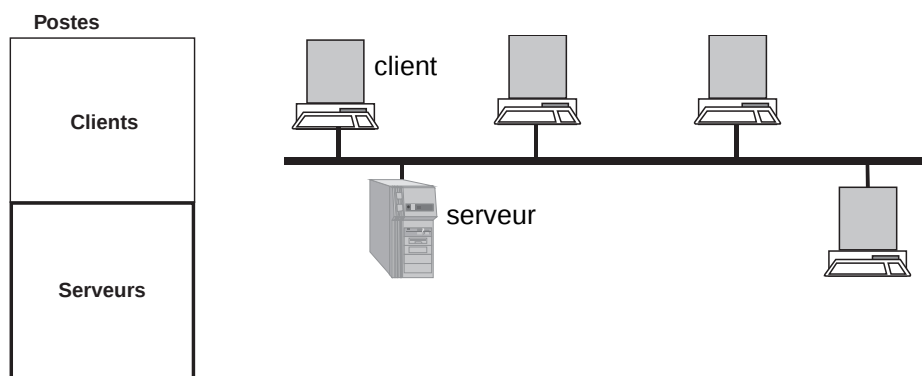
réseau alors qu'un serveur sera défini par un nom et que l'on voudra stocker l'espace disque encore disponible.

Disjonction

Couverture : héritage de partition

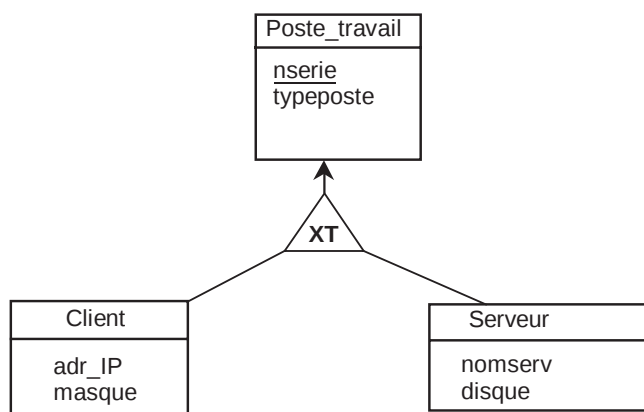
L'exemple 1-81 illustre le cas A : les postes de travail sont soit des clients, soit des serveurs. L'union des clients et des serveurs constitue le parc informatique. On peut dire qu'il y a deux partitions dans le parc informatique.

Figure 1-81 Héritage de partition (cas A)



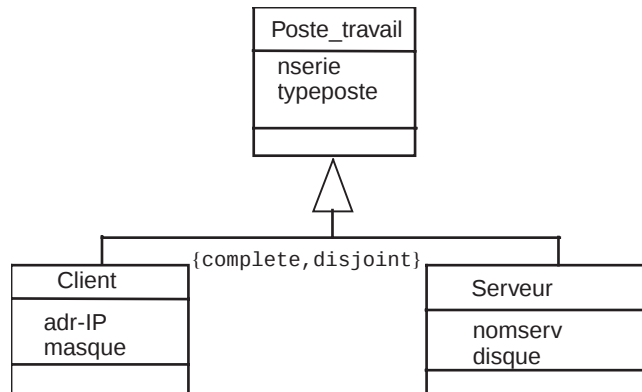
Bien que Merise/2 préconise le symbole P pour la partition, le diagramme 1-82 utilise le symbole XT (comme dans le cas des associations binaires).

Figure 1-82 Héritage de partition Merise/2 (Win'Design)



Le diagramme 1-83 représente la contrainte prédéfinie UML `{complete, disjoint}` qui exprime la partition.

Figure 1-83 Héritage de partition avec UML



On peut également utiliser une programmation OCL sous la forme de trois notes. La première, attachée à la classe `Client` signifierait qu'un client ne peut pas être serveur. Notez l'utilisation de la fonction `allInstances` qui retourne pour un type donné l'ensemble de ses instances, incluant aussi les instances de ses sous-classes.

```

context Client
inv R1 : Serveur.allInstances.noserie->excludes(self.noserie)
  
```

La seconde, attachée à la classe `Serveur` signifierait, d'une manière symétrique, qu'un serveur ne peut pas être client.

```

context Serveur
inv R2 : Client.allInstances.noserie->excludes(self.noserie)
  
```

La troisième, attachée à la classe `Poste` signifierait :

- qu'un poste est soit de type client, soit de type serveur (par le fait du *ou* exclusif). La fonction `oclIsKindOf` permet de statuer à propos de la nature de la classe d'un type passé en paramètre ;
- que le numéro de série est l'identifiant (fonction `isUnique`).

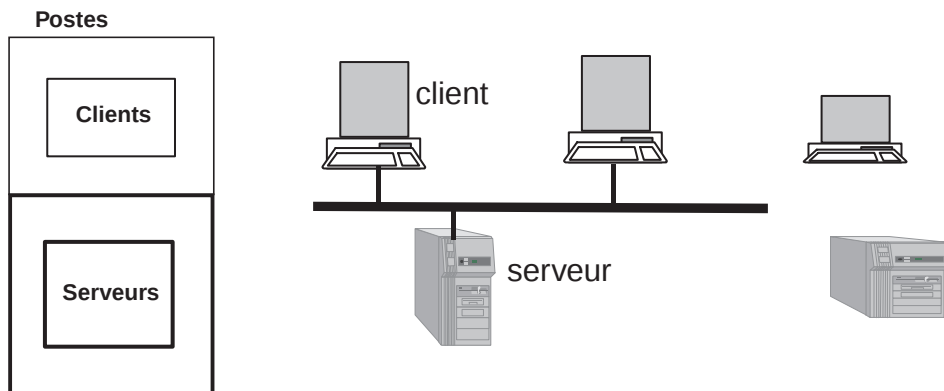
```

context Poste
inv R3 :
  (self.oclIsKindOf(Client) xor self.oclIsKindOf(Serveur)) and
  (Poste.allInstances.noserie->isUnique(c | c.noserie))
  
```

Non-couverture : héritage exclusif

La figure 1-84 illustre le cas B : il existe des postes n'étant ni client, ni serveur (l'union des clients et des serveurs ne compose plus le parc informatique : certains postes en sont exclus).

Figure 1-84 Héritage exclusif (cas B)



Merise/2 note cet héritage par le symbole X dans l'exemple 1-82.

Plusieurs notations UML sont possibles : la plus simple et la plus courante consiste à n'utiliser aucune contrainte (cas par défaut), une autre solution met en œuvre la contrainte prédéfinie {incomplete, disjoint}, enfin les adeptes des règles OCL devront utiliser :

- R1 et R2 précédemment définies ;
- R3 modifiée (qui relâche la contrainte spécifiant qu'un poste est soit client, soit serveur) de la manière suivante.

```
context Poste
```

```
inv R3 : (Poste.allInstances.noserie->isUnique(c | c.noserie))
```

Non-disjonction

Couverture : héritage de totalité

La figure 1-85 illustre le cas C : il existe des postes de travail considérés à la fois comme client et serveur. L'union des clients, des serveurs et des clients-serveurs compose le parc informatique dans sa totalité.

Merise/2 note cet héritage le symbole T dans l'exemple 1-82.

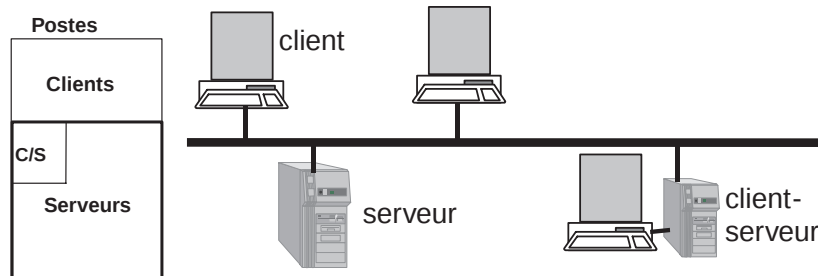
Le plus simple diagramme UML consiste à utiliser la contrainte prédéfinie {incomplete, overlapping}. Pour les adeptes du langage OCL, seule la règle R3 doit être

présente et modifiée (en relâchant la contrainte qu'un numéro de série est unique puisqu'un poste peut être à la fois client et serveur par le fait du *ou* inclusif).

```
context Poste
```

```
inv R3 : (self.oclIsKindOf(Client) or self.oclIsKindOf(Serveur))
```

Figure 1-85 Héritage de totalité (cas C)

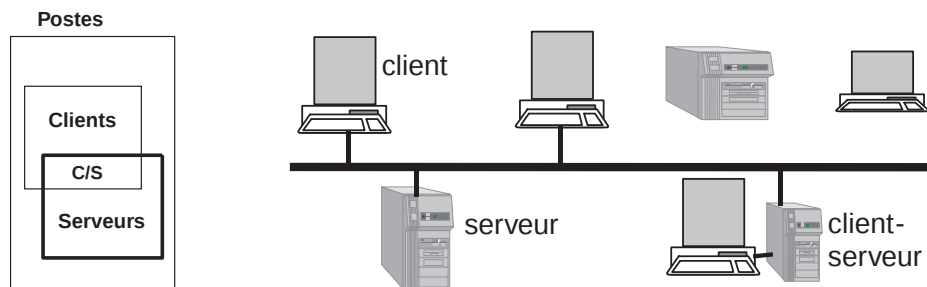


Dans les faits, la contrainte d'unicité du numéro de série sera toujours vérifiée au niveau de la base de données. Nous verrons au chapitre 2 comment peuvent se traduire les associations d'héritage.

Non-couverture : cas général

L'exemple 1-86 décrit le cas le plus général (les postes de travail peuvent être de toute nature).

Figure 1-86 Héritage sans contrainte (cas D)



Le schéma Merise/2 ne dispose d'aucune contrainte sur le graphe d'héritage.

Le diagramme UML consiste à utiliser la contrainte prédéfinie `{incomplete, overlapping}`.

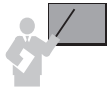
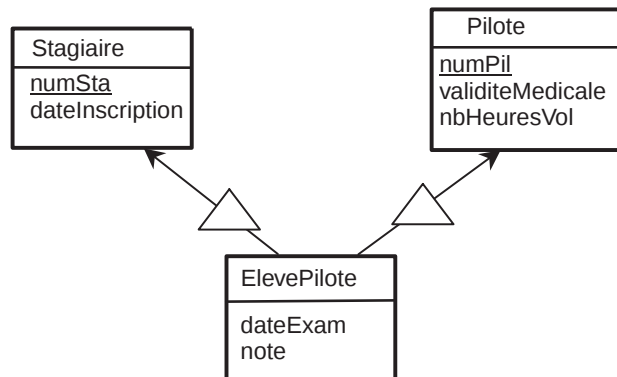
Héritage multiple



On parle d'*héritage multiple* quand une entité (ou classe) hérite simultanément de plusieurs sur-entités (ou sur-classes). Beaucoup de langages objet n'autorisent que l'héritage multiple d'interface (et non de classe).

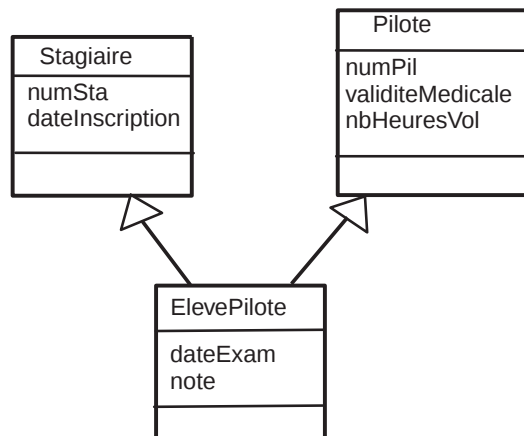
L'exemple 1-87 note, dans le formalisme Merise/2, un élève pilote comme étant à la fois un stagiaire et un pilote à part entière. UML l'écrit d'une manière similaire.

Figure 1-87 Héritage multiple avec Merise/2



Aucune contrainte n'est à ajouter sur des graphes d'héritage multiple car une instance de la sous-classe est obligatoirement instance des sur-classes.

Figure 1-88 Héritage multiple UML



Héritage simple

Dans le cas de l'héritage simple (lorsqu'une seule classe hérite d'une seule autre sur-classe), il n'existe que deux cas : la totalité ou l'absence de contrainte.

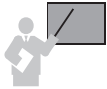
Bilan



Dans tous les cas d'héritage, les contraintes se traduiront au niveau physique par programmation de déclencheurs ou de contraintes de vérification SQL (CHECK).

Encapsulation

Concept bien connu des développeurs objet, le principe de l'encapsulation est d'autoriser l'accès aux données uniquement *via* les méthodes. Une autre édition consiste à considérer la capacité de l'objet à cacher ses méthodes. Depuis l'origine, les modèles entité-association ne décrivent pas les opérations disponibles au niveau des entités. UML note la présence d'une méthode d'une classe au niveau du troisième compartiment de la classe. Merise/2 permet toutefois d'affiner la définition des attributs (public, privé, protégé) au niveau du MOD (Modèle organisationnel des données).



L'encapsulation permet :

- d'augmenter le niveau d'abstraction de la classe ;
- de protéger les objets des classes d'une manipulation interdite de la part des clients de la classe ;
- de protéger les clients vis-à-vis des évolutions de la classe elle-même.

Les méthodes d'une classe représentent les services de la classe en question. Une classe fournisseur propose des services aux autres classes clientes. Conformément aux principes de l'approche objet, il faut vérifier que les classes clientes s'engagent à ne pas faire usage de connaissances autres que celles décrites dans la spécification de la classe fournisseur. Une méthode définit une abstraction de service dont les détails d'implémentation sont cachés.



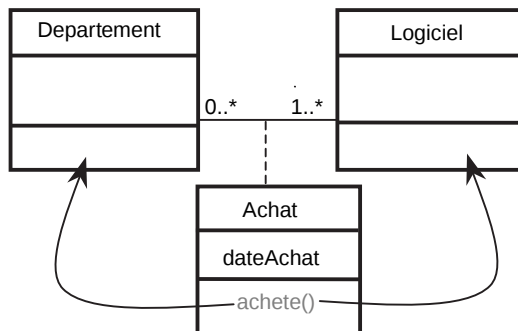
Au niveau d'une base de données, une méthode peut être programmée par une fonction ou une procédure cataloguée (pour les bases objet-relationnelles, ces méthodes sont définies directement au niveau du type de la table).

Positionnement des méthodes

L'exemple 1-89 décrit les achats de logiciels par des départements. Si le concepteur veut définir la méthode `achete()`, il pourra la placer soit dans la classe `Logiciel`, `Departement` ou

dans l'association Achat. Concernant la localisation de la méthode, il n'y a pas vraiment de principe formel à appliquer. Si la méthode est placée dans la classe-association, sa signature devra probablement inclure deux paramètres. De par la sémantique de la méthode, il semblerait plus naturel de la disposer dans la classe Département.

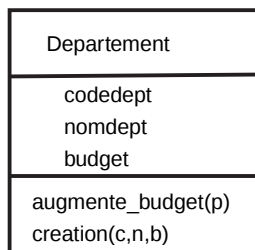
Figure 1-89 Position d'une méthode UML



La signature d'une *méthode* est constituée du nom de l'opération et, éventuellement, des paramètres d'appel.

L'exemple 1-90 décrit la signature des deux méthodes de classe Département. La création d'un département nécessiterait de valuer les trois attributs de la classe.

Figure 1-90 Signature de deux méthodes



Visibilité des attributs et des méthodes

Les règles de visibilité d'un attribut ou d'une méthode viennent compléter ou préciser la notion d'encapsulation. Ainsi, il est possible de préciser le degré de protection au profit de classes utilisatrices bien particulières, désignées implicitement dans la spécification de la classe fournisseur.

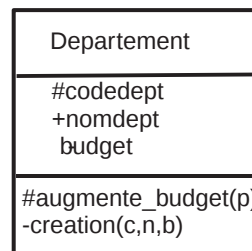


UML définit trois niveaux de *visibilité* pour les attributs et les méthodes (ces niveaux correspondent à ce que propose le langage C++) :

- *public* (symbole +) : niveau le moins strict, qui rend l'élément visible à toutes les autres classes ;
- *protégé* (symbole #) : niveau intermédiaire, qui rend l'élément visible aux sous-classes de la classe fournisseur (aux « classes amies », *friend classes* en C++) ;
- *privé* (symbole -) : niveau le plus strict, qui rend l'élément visible à la classe fournisseur seule (aux classes amies en C++).

L'exemple 1-91 précise la visibilité des attributs et méthode des deux services.

Figure 1-91 Niveaux de visibilité avec UML



Au niveau de la base de données

L'encapsulation et la visibilité devront être programmées manuellement et explicitement au niveau de la base. C'est là où se trouve le plus grand fossé entre classes d'objets et enregistrements de la base. Alors que l'encapsulation et la visibilité sont assurées intrinsèquement par les langages objet (C++ et Java par exemple), ces concepts ne sont pas natifs aux bases de données. Pour l'heure, les outils de conception ne répercutent pas automatiquement ces notions dans les scripts SQL. Et tant que les bases de données cibles seront relationnelles, il est probable que la situation restera en l'état et qu'il faudra se retrousser les manches...

L'encapsulation doit être programmée au niveau de la base par l'ajout de méthodes ou des procédures cataloguées. Il faudra donc prévoir suffisamment de méthodes d'insertion, de modification et de suppression d'objets de la base, ainsi que les services à assurer. Puis il faudra interdire l'accès à la base en révoquant les INSERT, UPDATE et DELETE tout en donnant les droits d'exécution sur les méthodes aux utilisateurs concernés (sous Oracle GRANT EXECUTE ON *nom_type* TO *utilisateur*). Oracle permet en outre d'affiner les prérogatives d'une méthode [SOU 04].

Attributs dérivés

Un attribut dérivé, au sens UML, est une propriété évaluée intéressante pour l'analyste mais redondante, car sa valeur peut être déduite d'autres données présentes dans le modèle. Au sens

des bases de données, il n'est pas très souhaitable de stocker des données calculables. Graphiquement, il suffit de positionner le caractère «/ » devant l'attribut.

L'exemple 1-92 note l'attribut dérivé `dureeVol`. Sa valeur sera déduite à l'aide de la différence de l'heure d'arrivée et de celle du départ.

Figure 1-92 Attribut dérivé UML

VolNavette
numero dateDepart dateArrivee heureDepart heureArrivee / dureeVol



Au niveau logique et physique, il n'y a pas de moyen de différencier un attribut dérivé d'un attribut non dérivé. Ainsi, il n'est pas conseillé d'utiliser cette propriété au niveau conceptuel qui devrait plutôt être inscrite en tant que méthode de la classe.

Identification et incidence sur la réification

Cette section concerne le processus d'identification des entités et des associations pour les inconditionnels de la méthode Merise. Les concepteurs UML feront l'analogie avec leur notation favorite.

Les écrits qui suivent sont un résumé des remarques formulées par Dominique Nanci qui a été un acteur dans l'élaboration du formalisme entité-relation qui est devenu entité-association, puis utilisé dans la méthode Merise. Par ailleurs, Christophe Sibertin-Blanc avait présenté un article entièrement dédié à cette problématique aux journées de l'AFCET en novembre 1990.

Identification absolue d'une entité



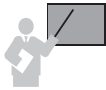
L'identifiant est composé d'un ou plusieurs attributs dont les valeurs permettent de désigner de manière unique chaque occurrence de l'entité. On parle d'identifiant simple quand l'attribut est unique (ex : `immatAvion` pour l'entité `Avion`). On parle d'identifiant composé quand il est composé de plusieurs attributs (ex : `nom+prenom+dateNaissance` pour l'entité `Etudiant`).

Pour assurer une identification absolue, tous les attributs composant l'identifiant de l'entité doivent être intrinsèques à l'entité (sémantiquement rattachés à cette entité). L'entité disposant d'une identification absolue modélise un objet disposant d'une identité propre, d'une indépendance de désignation. On peut faire référence à ses occurrences sans besoin d'autres entités. C'est la modélisation la plus simple, basique de Merise, que l'on cherche souvent à réaliser quitte à « inventer » des identifiants artificiels.

Identification relative

Dans certaines situations, l'identification d'une entité nécessite de se positionner relativement par rapport à une autre entité. Par exemple, l'entité *LigneCommande* s'identifie assez naturellement par le numéro de ligne par rapport à sa commande. Cela sous-entend que les entités *Commande* et *Ligne* sont associées par une relation d'appartenance.

Bien qu'intrinsèque à l'entité *Ligne*, le numéro ne peut être un identifiant absolu. Relativement à sa commande (identifiée par exemple par un numéro de commande), il joue alors le rôle d'identification d'une occurrence de l'entité *Ligne*.



L'identification relative d'une entité est donc construite à partir d'un ou plusieurs attributs intrinsèques à l'entité et d'autres entités auxquelles elle est reliée par des associations binaires de cardinalités 1,1.

Nous avons vu que l'outil Win'Design utilise le symbole (R) sur le lien d'association de cardinalités (1,1) connecté à l'entité faible. Nous verrons au chapitre 4 que l'identifiant relatif est noté au niveau du lien de l'association.

Cette notion d'identification relative a connu une histoire. Elle est abordée dans l'équipe de recherche initiale et figure dans [TAR 79b] sous une forme « confidentielle » de quelques lignes. L'ouvrage de base sur Merise [TAR 83], faisant explicitement référence pour une présentation plus détaillée du formalisme entité-relation au précédent ouvrage, passe sous silence cet aspect. Exit donc l'identification relative... À partir de 1987, Arnold Rochfeld et José Morejon, dans plusieurs articles, réhabilitent cette notion qui s'intègre désormais complètement dans la Merise/2 ainsi que dans la plupart des outils associés.

Identification d'une association

Par définition, une association n'a pas d'identifiant propre, son identification est construite par la composition (concaténation) des identifiants des entités connectées. C'est donc un abus de langage que de parler de l'identifiant d'une association, et encore plus comme certains auteurs, de faire figurer ces propriétés explicitement dans la relation.

Cette définition de l'identification d'une association implique qu'à un n -uplet d'occurrences des entités qui sont connectées (matérialisées par les identifiants), il n'y a qu'une seule occurrence de l'association. On peut le noter ainsi : $E1 \times E2 \times \dots \rightarrow A$.

Par contre, il n'a jamais été dit que la réciproque était systématique ! Ce n'est pas parce qu'un concept perçu est identifiable par la composition d'identifiants d'entités qu'il doit obligatoirement être modélisé comme une association. Certes, c'est souvent le cas mais cela peut aussi être modélisé *via* le concept des entités faibles.

Identifiant alternatif



Une entité peut présenter plusieurs identifiants potentiels. Celui retenu est appelé identifiant principal, les autres, identifiants alternatifs.

Cependant cette situation, compte tenu de la difficulté de trouver un identifiant, reste assez exceptionnelle (sauf dans le cas où cohabitent un identifiant naturel et un identifiant artificiel).

Cette notion apparaît plus souvent dans des graphes d'héritage entre entités. Les sous-entités (entités sous-type), qui généralement préexistent, sont souvent dotées d'identifiant absolu. La sur-entité (l'entité sur-type) possède également son propre identifiant. Dans ce cas, les identifiants des sous-types sont qualifiés *d'alternatifs* car ils possèdent par héritage l'identifiant de leur sur-type.

Considérons l'exemple des entités sous-type Assuré (d'identifiant absolu numINSEE) et Cotisant (d'identifiant absolu numURSSAF) qui héritent de l'entité Adhérent (d'identifiant numAdherent). L'identifiant alternatif de l'entité Assuré devient numINSEE et celui de l'entité Cotisant devient numURSSAF. L'identifiant absolu de ces deux entités devient, par héritage, celui de l'entité sur-type à savoir numAdherent.

Entité faible



On qualifie d'entité faible, une entité dont l'identification est réalisée relativement à plusieurs (2 à n) associations, sans disposer d'attribut intrinsèque intervenant dans l'identification. Son identification est donc identique à celle d'une association. Cependant, ce n'est pas parce que cette entité est identifiée comme une association qu'elle en est une.

Le choix de modélisation en entité ou association n'est pas un problème déductif mais perceptif. C'est avant tout l'intérêt sémantique du concepteur qui décide d'abord de modéliser un concept en tant qu'entité (ou association) et qui se pose ensuite la question de son identification. Cette modélisation par une entité faible correspond à une évolution dans le discours du concepteur.

Au début, le concept à modéliser apparaît comme une forme verbale (une association) : *un avion est affrété par une compagnie*. Puis progressivement, la forme verbale devient substantive : *pour chaque affrètement... les vols affrétés...* Ce groupe nominal (cet objet) intervient à son tour dans des associations.

Dans l'exercice concernant les affrètements d'avions dont vous trouverez l'énoncé plus bas, on ne peut pas parler d'entité faible car le vol dispose d'une propriété composant l'identification relative (la date du vol). En revanche, un cas d'entité faible se présente dans les exemples d'associations d'agrégation à propos de la mission des véhicules qui visitent des chantiers.

Dans la sémantique du discours, sous-tendue par la structure grammaticale, on est passé progressivement d'une association (groupe verbal) à un objet (groupe nominal). Ce phénomène est appelé réification ou substantification. La modélisation conceptuelle, dans l'esprit de Merise, est avant tout une vision sémantique : la perception en entité ou association prime sur la manière de l'identifier. Ce n'est pas l'identification qui détermine la modélisation, mais le choix de modélisation qui implique l'identification.

Le tableau suivant résume l'évolution de la modélisation en entité ou relation (terme utilisé initialement pour parler d'association).

Tableau 1.8 Classification des entités

Relation	Association entre des entités, identification déduite par construction, aucune existence propre, position de groupe verbal
Entité faible	Objet d'intérêt, obtenu souvent par substantification en position de groupe nominal, identification relative multiple sans attribut identifiant intrinsèque
Entité relative	Objet d'intérêt, identification relative simple ou multiple avec toujours un attribut identifiant intrinsèque
Entité dépendante	Objet d'intérêt, identification absolue, impliquée dans une ou plusieurs associations avec une cardinalité 1,1, objet autonome
Entité	Objet d'intérêt, identification absolue, objet indépendant

Exemple récapitulatif

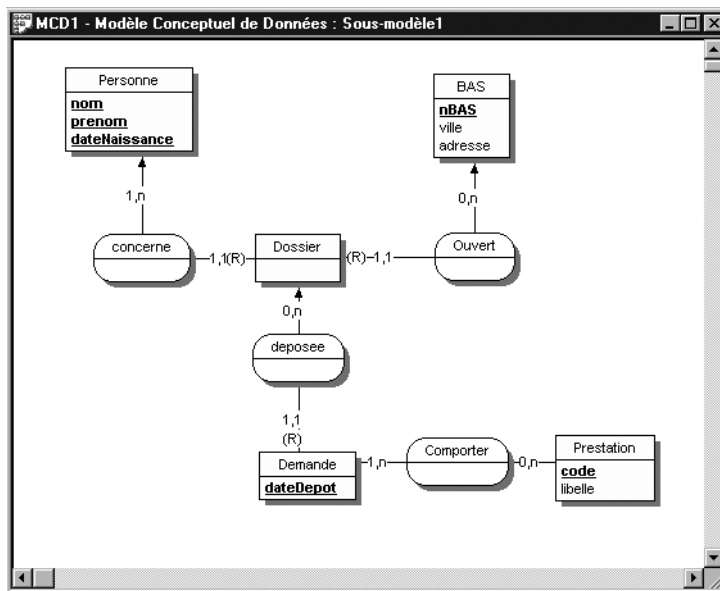
La gestion de l'Aide Sociale (AS) relève de l'autorité des Conseils Généraux (CG). Chaque CG a construit son système d'information. Les personnes souhaitant bénéficier des prestations de l'AS déposent leur dossier auprès du Bureau d'Aide Social (BAS) de leur municipalité.

Les personnes sont identifiées par leurs nom, prénom, date et lieu de naissance. Les BAS sont identifiés par un numéro départemental. Ces personnes déposent, dans le cadre de leur dossier, des demandes (identifiées par la date de dépôt) dont chacune comporte les prestations demandées. Normalement, une personne ne doit ouvrir qu'un dossier. La mise en commun des dossiers au niveau départemental a révélé que dans certaines grandes villes disposant de plusieurs BAS, certaines personnes disposaient, illégalement, de plusieurs dossiers.

Solution Merise/2

La modélisation 1-93 fait apparaître une entité faible **Dossier** et une entité relative **Demande**. L'identifiant relatif de **Dossier** est (nom, prenom, dateNaissance, nBAS). L'identifiant relatif de **Demande** est (nom, prenom, dateNaissance, nBAS, dateDepot).

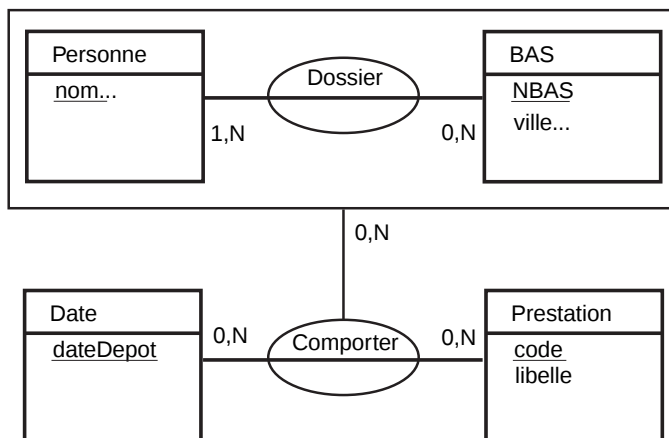
Figure 1-93 MCD Win'Design



Solution par les associations d'agrégation

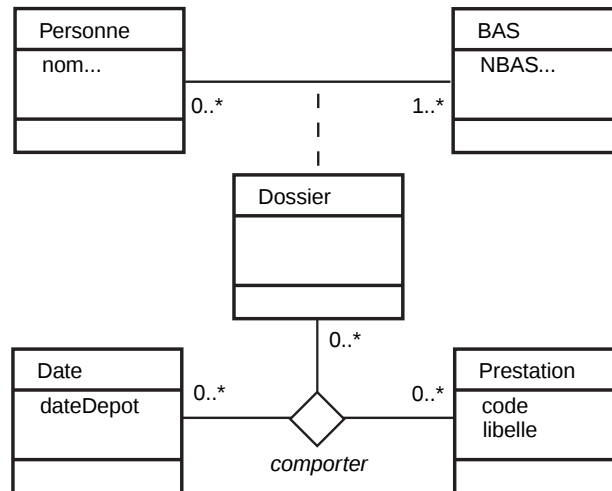
En utilisant les associations d'agrégation, avec lesquelles il est plus facile de faire l'analogie avec le formalisme UML, on obtient le schéma 1-94. Au niveau logique, on économiserait une relation (il n'est pas nécessaire de formuler une demande si elle n'est pas rattachée à une prestation).

Figure 1-94 Solution avec les association d'agrégation



Le diagramme UML 1-95 permet de visualiser l'analogie entre les deux formalismes.

Figure 1-95 Solution avec UML



Aspects temporels

Il est très fréquent d'avoir à prendre en compte des aspects temporels au niveau conceptuel. On peut classer trois modélisations : le moment, la chronologie et l'historisation.



Évitez de nommer un attribut temporel par un mot réservé du langage informatique (date, month, day, etc.). Préférez des identificateurs plus parlants (exemple : dateNaissance, dateVente, moisArrivee, etc.)

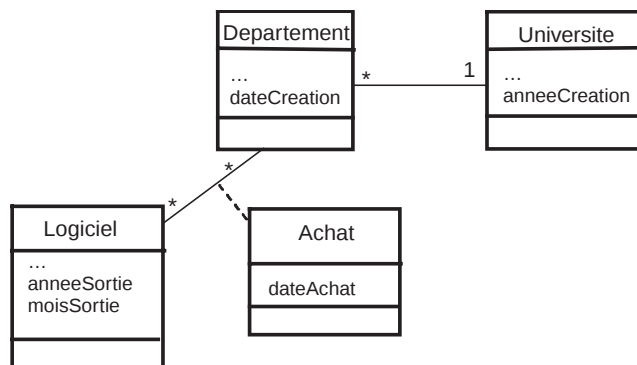
Modélisation d'un moment

Il s'agit de représenter des informations indiquant un moment (le plus souvent une date). Ce moment est représenté sous diverses formes : *jour*, *mois*, *jour/mois*, *année*, *mois/année*, *jour/mois/année*, avec ou sans les heures, minutes et secondes, etc.). Le plus souvent cet attribut sera positionné dans une classe.

L'exemple 1-96 décrit quelques attributs temporels. On peut supposer que *anneeCreation* et *anneeSortie* seront au format *année* (entier sur 4 positions), *dateCreation* au format *mois/année*, *moisSortie* au format *mois* (entier sur 2 positions) et *dateAchat* au format *jour/mois/année*. Ce dernier attribut se trouve dans la classe-association car un département

peut acheter à plusieurs dates, de même un logiciel peut être vendu à plusieurs dates. La modélisation indique ici qu'un département n'achète pas plusieurs fois le même logiciel, et cette date correspond à l'achat d'un logiciel par un département.

Figure 1-96 Modélisation d'un moment

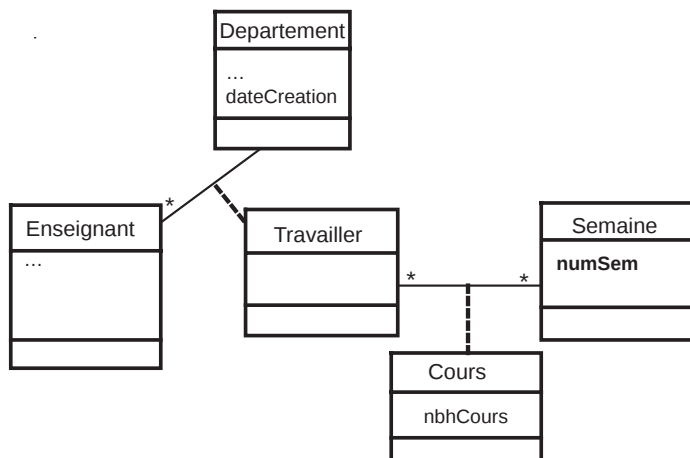


Modélisation de chronologie

Il s'agit de modéliser des situations où le temps intervient de manière régulière (plannings, emploi du temps, prévisions, statistiques, etc.). Le plus souvent cet attribut sera considéré en tant que classe. Les formats de cette donnée peuvent aussi être divers (*jour*, *mois*, *jour/mois*, etc.).

L'exemple 1-97 décrit l'attribut de chronologie `numSem` qui identifie la classe **Semaine**. Ici on suppose que des enseignants peuvent travailler dans plusieurs départements et que l'on souhaite connaître le volume de leurs cours pour chaque semaine.

Figure 1-97 Modélisation d'une chronologie



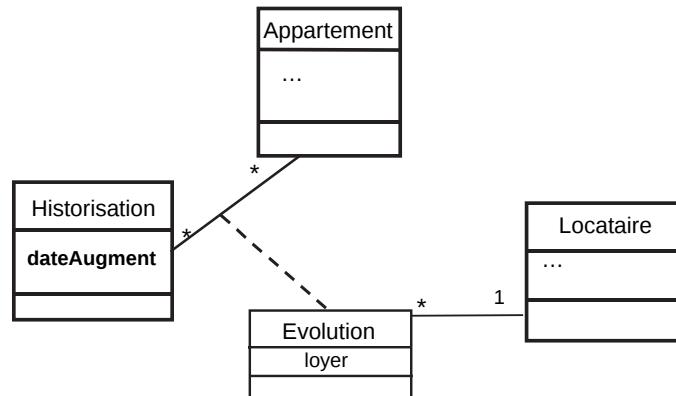
Modélisation de l'historisation

Ce type de modélisation est à utiliser quand on désire conserver les valeurs antérieures d'un attribut. Citons quelques exemples comme le montant d'un loyer (on désire connaître son évolution) ou la valeur du coefficient bonus/malus d'un contrat d'assurance. En contre-exemple, le nombre de passagers d'un vol donné sera plutôt traité comme une chronologie (exemple traité dans les exercices).

Alors que Merise/2 met à disposition un artifice pour noter l'historisation (symbole H sur un lien d'association), il est possible de noter avec UML cette modélisation en ajoutant un attribut de type moment.

L'exemple 1-98 décrit l'historisation des montants de loyer des appartements. L'attribut `dateAugment` identifie la classe `Historisation`.

Figure 1-98 Modélisation d'une historisation



La démarche

Les sources d'information à utiliser lors de la modélisation peuvent être diverses : énoncé en langage naturel (c'est le cas des exemples du livre présentés à l'aide d'un texte introductif), interviews, observation du système, états de sortie papier, copies d'écran, etc.

Dans tous les cas, on doit se ramener à des phrases élémentaires en prenant garde d'éviter de formuler des propositions incomplètes, redondantes, contradictoires ou fausses.

Décomposition en propositions élémentaires

Une proposition élémentaire est assimilée à la composition *sujet-verbe-complément*. Il est courant d'avoir à reformuler certaines phrases lues ou entendues. Ainsi la proposition « Un

avion est caractérisé par une immatriculation, un nombre de places assises et appartient à une compagnie. » devra être décomposée en :

- « Un avion est caractérisé par une immatriculation. »
- « Un avion est caractérisé par un nombre de places assises. »
- « Un avion appartient à une compagnie. »

Un autre exemple, la proposition « Le prix de vente d'un avion devra être proportionnel au nombre de places assises. » devra être décomposée en :

- « Un avion a un prix de vente. »
- « Un avion est caractérisé par un nombre de places assises. »
- « Le prix de vente est proportionnel au nombre de places assises. »

Propositions incomplètes

Il ne doit pas exister de proposition incomplète. Ainsi une seule proposition mettant en jeu trois entités « Un logiciel est installé par un département sur un serveur. » risque d'être incomplète. En effet, il faut préciser ce fait en considérant les entités deux-à-deux.

- « Un département peut-il installer tout logiciel? »
- « Un serveur peut-il accueillir tout logiciel? »
- « Un département peut-il utiliser tout serveur? »

Je doute fort que la réponse soit « oui » à chacune de ces propositions. L'association ternaire représentant l'installation devra donc soit être décomposée, soit inclure une contrainte. Pour les associations de degré supérieur, il est possible de généraliser ce raisonnement.

Propositions redondantes

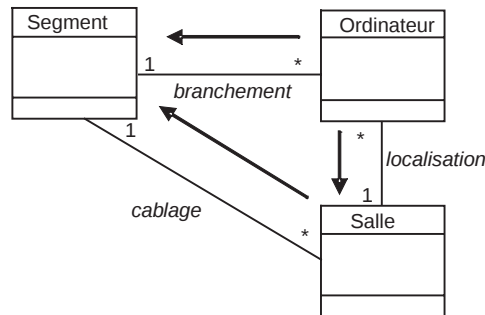
Il faut éviter de conserver des propositions redondantes. Parmi les trois propositions suivantes, une est redondante car déduite des deux autres.

- « Un ordinateur est branché à un segment réseau. »
- « Un ordinateur est situé dans une salle. »
- « Seul un segment réseau passe dans une salle. »

La proposition redondante est la première (qui induirait l'association branchement de l'exemple 1-99). En effet, tout ordinateur étant situé dans une salle où passe un seul segment, il n'est pas nécessaire de relier l'ordinateur à son segment, car il suffit de connaître la salle pour en déduire cette information.

L'explication plus formelle est donnée par les dépendances fonctionnelles. Le schéma 1-99 n'est pas en troisième forme normale en effet la dépendance fonctionnelle ordinateur → segment n'est pas directe car elle est déduite des deux autres (ordinateur → salle puis salle → segment).

Figure 1-99 Exemple de transitivité



Propositions réductibles

Étudions un autre exemple avec la proposition « Un client réalise des achats de produits chez des fournisseurs. ». Décomposons de manière intuitive de sorte à extraire des propositions plus élémentaires.

- « Un achat est effectué par un client. »
- « Un achat concerne plusieurs produits. »
- « Un achat s'effectue chez un fournisseur. »

On peut en déduire que Client, Produit et Fournisseur seront des entités (classes). Achat sera l'association (classe-association) reliant Fournisseur au couple (Client, Produit).

Propositions complexes irréductibles

En considérant à nouveau la proposition « Un logiciel est installé par un département sur un serveur. » et en supposant que les réponses soient « oui » à chacune des propositions binaires, alors l'association représentant l'installation sera ternaire (sans contrainte).

De même, concernant l'exemple précédent, si toutes les propositions suivantes sont vraies, aucune décomposition ne sera possible et l'association représentant l'achat sera ternaire (sans contrainte).

- « Un client peut acheter tout produit. »
- « Un fournisseur dispose de tout produit. »
- « Un client peut se servir chez tout fournisseur. »

Chronologie des étapes

En théorie, la construction d'un schéma conceptuel suit les étapes suivantes :

- Recensement des entités (classe).

- Disposition des attributs dans chaque entité (classe).
- Détermination d'un identifiant par entité (classe).
- Recensement des associations et définition du degré : binaire, classe-association ou *n*-aire.
- Disposition d'éventuels attributs dans chaque classe-association.
- Mise en place des contraintes.
- Vérification par normalisation.



Attention à ne pas surestimer le degré d'une association.

Ne pas réfléchir en terme de fonctionnement ou en termes de traitement, mais simplement exprimer des faits.

Bilan

Alors que les aspects statiques du niveau conceptuel (entité, classe, association, classe-association, héritage) se traduisent naturellement sous SQL2 ou SQL3, les contraintes nécessitent un effort de programmation le plus souvent sous la forme de contraintes SQL de vérification CHECK ou de déclencheurs. L'aspect dynamique du niveau conceptuel n'est pas natif aux bases de données et il restera à programmer l'encapsulation par les méthodes.

Les méthodes et outils de conception objet qui permettent de transiter du conceptuel au physique (spécifications UML traduites automatiquement en classes C++ ou Java) sont bien plus adaptés à l'élaboration de programmes que de schémas de bases de données. Les principaux inconvénients de ces démarche sont les suivants :

- le souci d'intégrité des objets stockés n'est pas suffisamment pris en compte ;
- la traduction des associations au niveau physique n'est ni clairement décrite, ni totalement maîtrisée.

UML 2 ou Merise/2 ?

Après avoir examiné différents formalismes : le modèle de Chen, le MCD de Merise/2 et la notation UML, on est en droit de savoir quel formalisme convient le plus à la modélisation d'une base de données. La notation UML 2 est bien adaptée à la modélisation d'une base de données (et si elle est utilisée à bon escient !) pour les raisons suivantes :

- les bases de données sont majoritairement relationnelles, mais certaines migreront vers l'objet ;
- les concepts fondamentaux de Merise/2 peuvent être représentés dans le diagramme de classes d'UML qui offre en plus la possibilité de définir des stéréotypes et des contraintes personnalisées ;

- la majorité des contraintes sur les associations n -aires sont implicites si on utilise les classes-associations ;
- les concepteurs travaillent dans un même environnement (comme Eclipse), avec la possibilité d'interfacer plus facilement les langages évolués C++ ou Java.

Bien qu'il reste encore des fonctionnalités absentes des outils utilisant la notation UML, notamment en ce qui concerne les contraintes et des associations n -aires, les versions à venir de ces produits devraient être de plus en plus performantes.

Cette édition fait encore référence au modèle entité-association dans les prochains chapitres pour que les inconditionnels y trouvent leur compte.

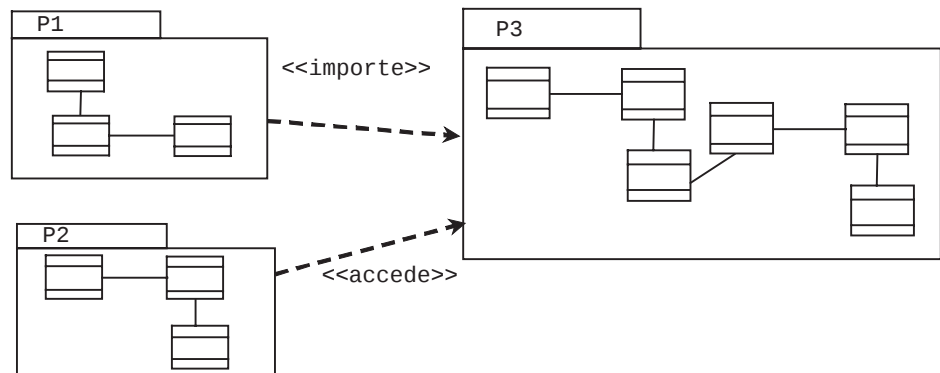
Quelques règles à respecter avec UML

La spécification UML 2 n'est pas très restrictive: « *Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice. Generalizations between associations are best drawn using a different color or line width than what is used for the associations.* »

L'adoption des règles suivantes vous permettra de construire des diagrammes de classes UML les plus lisibles et évolutifs possible :

- Évitez de croiser les liens d'association (si c'est toutefois inévitable, utilisez la notation du « pont » électrique).
- Tracez les liens d'association horizontalement ou verticalement.
- Concernant les associations *un-à-plusieurs*, essayez de placer la classe qui contient une multiplicité * en haut ou à gauche de la classe qui contient une multiplicité 1.
- N'hésitez pas à construire des classes grandes et fines ou, au contraire, des classes petites et larges afin d'éviter les croisements de liens d'association.

Figure 1-100 Utilisation de paquets



- Utilisez la notion de paquetages pour former des sous-schémas (regroupement de classes reliées entre elles) dans le cas de diagrammes volumineux. L'exemple 1-100 illustre deux relations de dépendance entre paquetages. Le paquetage source est dit client, le paquetage cible est dit fournisseur. Le stéréotype <<importe>> permet d'ajouter des éléments du paquetage fournisseur au paquetage client. Le stéréotype <<accède>> permet de référencer des éléments du paquetage fournisseur.

Et après ?

Il est utile d'établir plusieurs niveaux de conception, car ils permettront de séparer les spécifications formelles de l'implémentation et rendront les schémas indépendants des matériels et des logiciels, dans la mesure du possible. Nous énonçons au chapitre suivant les règles d'évolution pour définir un modèle de données (niveau logique) à partir d'un diagramme conceptuel.

Exercices

Composer les schémas conceptuels de type Merise/2 ou UML 2 afin de modéliser les faits suivants. Pour faciliter les corrections qui se trouvent sur le site d'accompagnement, utilisez les noms d'attributs qui sont indiqués en police courriel. Les exercices sont de difficulté croissante.

Exercice

1.1 Inscription des étudiants

Lors de son inscription en début d'année scolaire, chaque étudiant remplit une fiche sur laquelle il indique certains renseignements comme son numéro d'identification nationale (*ninsee*), ses nom et prénom (*nom*, *prenom*), son adresse (*adresse*) et la liste des unités de valeurs (UV) qu'il s'engage à suivre (8 au plus sur les 15 possibles). Un code lui est automatiquement attribué (*codetu*).

Une UV est caractérisée par un code (*codeuv*) et un intitulé (*intuv*). Par exemple le code *UV3* identifie *Électronique numérique*. Chaque UV est placée sous la responsabilité d'un enseignant identifié par ses initiales (*initens*) et caractérisé par un nom (*nomens*), un numéro de bureau (*bureauens*) et un numéro de téléphone (*telens*). Cet enseignant se rend disponible un jour de la semaine (*jsem*) et durant une plage horaire précise (*hrens*) afin de donner tout renseignement concernant les UV qu'il dirige.

- Que faut-il modifier pour qu'un enseignant se rende disponible à différents moments (un seul créneau par UV qu'il dirige) ?
- Que faut-il modifier pour que différents créneaux soient disponibles par UV qu'il dirige ?

Exercice

1.2 Gestion des stages

Une école de journalistes veut organiser des stages à l'intention des étudiants en dernière année (*codetu*, *nom*, *prenom*, *ninsee*). Elle fait appel à des entreprises (*codent*, *adressent*) qui proposent des stages. Les stages sont caractérisés par un code stage, une durée, une date de début et de fin (*nstage*, *duree*, *ddeb*, *dfin*).

Chaque stage est consacré à un thème d'étude (nomtheme) répertorié par un code (codetheme). Exemple : le code *eco1* identifie le thème *économie générale*. Les étudiants indiquent les stages qu'ils souhaitent suivre en priorité en leur attribuant un numéro de préférence (npriorite). Le chef d'établissement décide ensuite des attributions. Chaque stagiaire est placé sous la responsabilité d'un enseignant (utiliser l'entité Enseignant du précédent exercice).

On désire obtenir des statistiques selon le thème du stage effectué par les étudiants. On désire stocker les diplômes (codedip, nomdip) qu'ont obtenus les étudiants (bacs et diplômes de premier cycle). Les éventuelles mentions aux diplômes (mentiondip) doivent aussi être stockées.

Précisez la nature de chaque stage par un pourcentage associé à chaque thème (par exemple, le stage *c1* est composé de 20 % de *eco1* et 80 % de *marketing2*).

On désire connaître les antécédents des étudiants avant l'obtention de leur diplôme final. En particulier, il sera utile de connaître pour chaque étudiant le nombre d'années passées (nbanpasse) dans les différents établissements fréquentés pour avoir un diplôme (utiliser l'entité Etablissement avec codetab, nometab, nomdirecteur).

Exercice 1.3 Affrètement d'avions

Un aéroport toulousain désire gérer les compagnies, leurs avions et les vols affrétés. Une compagnie est caractérisée par un code et un nom (comp, nomComp). Chaque avion est désigné par une immatriculation (immatriculation), un type (typeAvion), une capacité (capacite). Un avion est la propriété d'une compagnie.

Un avion peut être affrété par une compagnie à différentes dates (dateVol), même plusieurs fois par jour par différentes compagnies. Pour chaque affrètement il faudra stocker le nombre de passagers transportés (nbPax) et le coût du vol pour la compagnie (cout). On ne pose pas de contrainte, donc, chaque compagnie peut affréter n'importe quel avion à n'importe quel moment. On suppose toutefois qu'une compagnie ne peut pas affréter le même avion plusieurs fois dans la même journée.

L'aéroport décide de stocker les caractéristiques de chaque type d'avion : le code de la désignation commerciale, le nombre maximum de passagers (npMax) et la désignation commerciale (nomAvion). Exemple : l'A320 peut transporter au maximum 180 passagers et se dénomme « Airbus A320 ».

- Comment décrire la contrainte UML 2 qui interdit à une compagnie d'affréter des avions qui ne lui appartiennent pas ?
- Décrire la modification à faire pour qu'une compagnie puisse affréter un avion plusieurs fois dans la même journée.

Exercice 1.4 Gestion des partiels

Compléter le schéma de la scolarité (exercice 1.1) en tenant compte de nouvelles spécifications :

En cours d'année, dans le cadre du contrôle continu des connaissances, chaque étudiant subit deux partiels pour chaque UV qu'il a choisie. En fin d'année il passe un examen obligatoire. Pour chaque épreuve (partiel ou examen) on désire mémoriser :

- un numéro chronologique qui identifie chaque épreuve quelle que soit la matière (ncont) ;
- la date (datecont), l'heure de début (debcont) et sa durée (dureecont) ;

- le type de l'épreuve (typecont).

Après correction des copies, il est affecté une note à chaque étudiant (notetu). Une UV est décernée sous réserve d'une moyenne supérieure à 10.

Les épreuves sont surveillées par un ou plusieurs enseignants et peuvent se dérouler simultanément dans différentes salles. Les salles sont caractérisées par un code (codesal) et un nombre de places (capacité). Les enseignants sont caractérisés par un code, un nom et un grade. Chaque enseignant rédige un rapport concernant sa surveillance sous la forme d'une note de quelques lignes (rapport). Un enseignant peut passer d'une salle à l'autre durant un contrôle. On désire connaître le temps de présence (tempspresence) passé dans chaque salle au total.

Exercice 1.5 Castanet Télécoms

Castanet Télécoms désire utiliser une base de données afin de gérer les factures de ses abonnés. Il est nécessaire de pouvoir stocker dans la base de données tous les éléments relatifs au calcul d'une facture. On suppose que les différentes consommations sont automatiquement importées d'un auto-commutateur (par exemple, une communication d'une heure trente sera automatiquement transformée en 1,5 dans un champ numérique de la base de données).

On ne souhaite pas conserver l'historique de l'année en cours, mais seulement les consommations courantes de périodes de deux mois. Tous les abonnés de Castanet Télécoms reçoivent une facture tous les deux mois. Il est nécessaire de connaître les différentes consommations (locales, nationales, Internet, vers mobiles et autres appels). Les coordonnées (nom et adresse) d'un abonné devront être connues (pour lui envoyer ses factures). L'abonné peut choisir le débit de sa ligne pour chaque période de deux mois (1, 2, 8 ou 20 Mo/s) – bien sûr, le montant de la facture sera modifié en conséquence.

Les abonnés peuvent souscrire un contrat pour Internet. Plusieurs formules de contrat sont possibles :

- la formule 'F1', qui coûte 45 euros par mois avec 3 heures de connexion comprises, l'heure supplémentaire étant taxée 19 euros ;
- la formule 'F2', qui coûte 95 euros par mois avec un nombre d'heures de connexion illimité ;
- d'autres formules toutes basées sur un montant par mois.

Castanet Télécoms fait appel à des fournisseurs de services Internet pour prendre en charge ces formules. Ainsi, il est possible qu'une même formule soit proposée par différents prestataires et que des abonnés soient connectés à différents fournisseurs de services pour la même formule de contrat. Castanet Télécoms désire conserver les dates de premier contact et de fin de contrat avec chaque prestataire Internet pour les différentes formules qu'il a proposées. Il est aussi nécessaire de conserver l'URL, l'adresse et le nom du responsable pour tous les prestataires Internet. Un abonné peut également opter pour un téléphone portable. Une fois le contrat signé, un numéro de portable est attribué au client et un combiné portable lui est donné. Le type du combiné peut être basique, moderne ou de luxe. L'abonné peut choisir trois numéros privilégiés pour lesquels une réduction importante sera calculée en fonction des durées d'appel, lors de l'édition de la facture.

Les attributs à prendre en compte sont les suivants : numAbonne, nomAbonne, adresseAbonne, numPortable, typeCombine, telPrefere, dureePreferee, numTel, consoLocale, consoNationale, consoInternet, consoAutres, debitLigne, consoMobiles, formule, prixParMois, heureSupp, URL, adresseFourni, responsable, premierContact, finContrat.

Exercice 1.6 Voltige aérienne

Compétition journalière

Les organisateurs du championnat du monde de voltige aérienne décident de concevoir une base de données relationnelle afin de gérer la compétition. Celle-ci se déroulera sur période de 9 jours, elle rassemblera 25 avions et 30 compétiteurs. Chaque voltigeur pourra utiliser au plus trois avions différents lors de la compétition (même des avions qui n'appartiennent pas à son club de voltige). Pour cela, il devra notifier aux juges avant le début de la compétition la liste des avions qu'il envisage d'utiliser. Il ne sera donc pas possible pour un voltigeur de voler sur n'importe quel avion. On suppose qu'un pilote ne vole qu'une fois au plus par jour.

Chaque vol de compétition comprend 7 figures qui seront notées distinctement par les juges. On suppose que l'on ne stocke pas la date des vols (la base est vidée chaque matin). Il faudra stocker en plus les éléments suivants :

- numéro, nom et aérodrome de base de chaque club de voltige (exemples : LFBO pour Blagnac, LFCL pour Lasbordes, LFCI pour Albi...) ;
- immatriculation, type des avions et rattachement au club propriétaire ;
- identité du voltigeur (numéro et nom du compétiteur, pays et club de voltige) ;
- pour chaque figure de voltige référencée dans un ouvrage de référence (qui en comporte plus de 1 500), il faudra connaître son nom, sa cotation maximale (note qui varie de 5 à 10), la date de création et le numéro de la page du livre la décrivant ;
- les inventeurs des figures doivent être également mémorisés avec leur nom et le type de personnage (militaire, pilote civil ou autre).

Les attributs sont les suivants : `nclubVoltige`, `aerodromeBase`, `nomClub`, `immatriculation`, `typeAvion`, `ncomp`, `nomComp`, `pays`, `nfigure`, `nomFigure`, `noteMax`, `dateCreation`, `pageManuel`, `ninventeur`, `nomInventeur`, `typePilote`, `note`.

Historique des compétitions

Il est nécessaire maintenant de stocker les résultats pour chaque jour de la compétition. On suppose toujours qu'un pilote ne vole qu'une fois au plus par jour.

Chaque vol de compétition est soit un programme connu, un des deux programmes inconnus ou un libre intégral (où les compétiteurs peuvent faire 12 figures). Lors du libre intégral, les juges doivent avoir reçu de la part du compétiteur avant le vol la liste des figures avec leur chronologie.

Les attributs à ajouter sont les suivants : `numeroChrono`, `dateVol` et `typeProgramme`.

Chapitre 2

Le niveau logique : du relationnel à l'objet

*Nous ne cesserons pas d'explorer -
L'aboutissement de toutes nos quêtes -
Sera d'atteindre l'endroit d'où nous étions partis -
Et pour la première fois de le reconnaître.*

The Magus, J. Fowles, Jonathan Cape, 1967

Ce chapitre se divise en trois parties. La première décrit le modèle relationnel et traite de la normalisation. La deuxième décrit les concepts objet présents au niveau logique. La troisième, enfin, expose les règles permettant de dériver un schéma relationnel ou objet à partir d'un diagramme entité-association ou UML.

Modèle relationnel

Bien que tous les outils permettent de modéliser un schéma logique relationnel, aucun ne peut offrir un processus de normalisation automatique. Il est donc nécessaire de maîtriser à la fois le modèle de données et les principes de normalisation afin de concevoir des bases de données cohérentes.

Historique, généralités

Le modèle relationnel a vu le jour dans les années 1970 avec les travaux de E.F. Codd [COD 69, 70]. Il est fondé sur de solides bases théoriques, car il propose des opérateurs issus de la théorie des ensembles. Par ailleurs, le processus de normalisation permet de construire des bases de données sans redondance d'informations, tout en préservant l'intégrité des données lors d'ajouts, de modifications ou de suppressions d'enregistrements.

Selon C. Date, consultant et auteur d'ouvrages édités à plus d'un demi-million d'exemplaires, c'est cette théorie qui fait que ce modèle peut être considéré comme une science [DAT 00]. Le modèle relationnel recouvre trois aspects : l'aspect structurel (développé dans ce chapitre), l'aspect intégrité (plus précisément traité dans le chapitre 3, lors de l'étude des contraintes) et, enfin, l'aspect manipulation au sens des opérateurs relationnels (cet aspect sort du contexte de la conception, le lecteur intéressé pourra consulter [BOU 99] à ce propos). Ces différents aspects sont programmés à l'aide du langage SQL que nous étudierons aux chapitres suivants (niveaux physique et externe).

Le modèle relationnel est à l'origine du succès que connaissent aujourd'hui les grands éditeurs de SGBD (système de gestion de bases de données), à savoir Oracle, IBM, Microsoft, Informix, Sybase et CA-Ingres. Le but initial de ce modèle était d'améliorer l'indépendance données/traitements. Les premiers systèmes informatiques ayant mis en œuvre les concepts du modèle relationnel ont été System-R d'IBM [AST 76] et Ingres [STO 76], tous deux apparus en 1976. La première version commerciale d'Oracle, devenu leader du marché, est sortie en 1979 (la société est née en 1977).

Modèle de données

Le modèle de données relationnel repose sur des principes simples et permet néanmoins de modéliser des données complexes. La description des données repose sur le concept de relations. Nous utilisons dans cet ouvrage une syntaxe simple (figure 2-1). Nous verrons plus loin qu'il est aussi possible de modéliser un schéma relationnel avec le formalisme UML.

Terminologie



Une *relation* possède des *attributs*. Chaque attribut prend sa valeur dans un *domaine* (ensemble de valeurs).

Chaque élément d'une relation est appelé *uplet* ou *n-uplet* (*n* désignant le nombre d'attributs de la relation et étant aussi appelé *degré* de la relation).

L'ensemble des *n-uplets* désigne la relation en *extension*.

Notations



$R[a_1, \dots, a_n]$ désigne la relation *R* composée de *n* attributs définis sur leurs domaines $D_1, \dots, D_n$ en *intention*.

R est un sous-ensemble du produit cartésien $D_1 \times D_2 \times \dots \times D_n$.

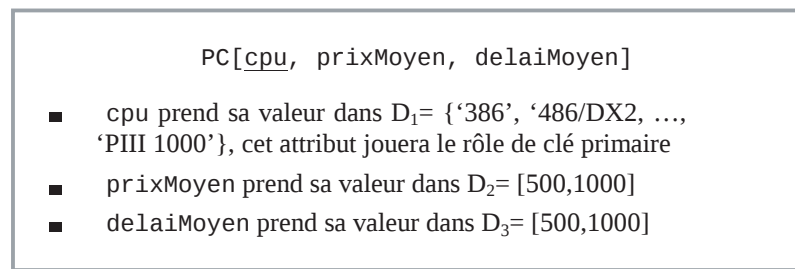
Pour travailler formellement avec le modèle relationnel, on considère que seuls n -uplets de chaque relation sont présents dans la base (c'est ce qu'on appelle l'hypothèse du monde clos).

Définition en intention

Considérons l'exemple suivant : les micro-ordinateurs du marché sont caractérisés par le code du processeur, un prix moyen et un délai de livraison moyen. Les processeurs des plus anciens micro-ordinateurs sont des 386 et les plus récents sont des Pentium V cadencés à 1 GHz. Les prix s'échelonnent de 500 € à 1 000 € et les délais de livraison varient de 0 à 8 jours.

La figure 2-1 définit la relation en intention :

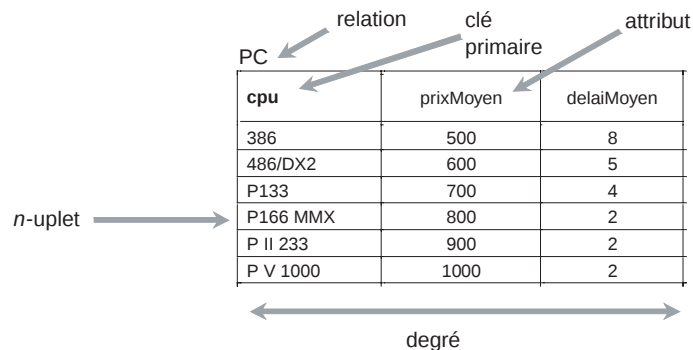
Figure 2-1 Relation en intention



Définition en extension

La définition en extension de la même relation est illustrée à la figure 2-2. Elle peut être considérée comme le contenu de cette relation. On voit bien qu'il s'agit d'un sous-ensemble du produit cartésien des trois domaines de valeurs. En effet, le tableau ne contient pas toutes les combinaisons de lignes qu'il est possible de composer avec les trois domaines de valeurs D_1 , D_2 et D_3 .

Figure 2-2 Relation en extension



Équivalences avec le modèle de données du SGBD

La quasi-totalité des concepts du modèle relationnel a été implantée dans les SGBD *via* le langage SQL.

Terminologie



La *table* est l'équivalent de la relation. Une table contient des données appelées *enregistrements* (aussi appelées « lignes »).

Une table est composée de *colonnes* (aussi appelées « champs »).

La *clé primaire* d'une table est l'ensemble minimal de colonnes qui permet d'identifier de manière unique chaque enregistrement.

Une table contient ou non des *clés candidates*. Une clé est dite candidate si elle peut se substituer à la clé primaire.

Une table contient ou non des *clés étrangères*. Une clé étrangère est composée d'une ou de plusieurs colonnes et permet d'identifier un enregistrement d'une autre table.

Les liens entre les enregistrements de la base de données sont réalisés non pas à l'aide de pointeurs physiques, mais à l'aide des valeurs des clés étrangères et des clés primaires. Pour cette raison, le modèle relationnel est dit « modèle à valeurs ».

Exemple

Dans l'exemple 2-3, la table *Etudiant* possède une clé étrangère (notée *cpu#*). Cet attribut permet de mettre en association la table *Etudiant* avec la table *PC*. Ainsi, nous représentons le fait que chaque étudiant possède un seul PC.

Figure 2-3 Clés primaire et étrangère

Etudiant		
codetu	nomEtu	cpu#
E01	Jean	486/DX2
E02	Pierre	P166 MMX
E03	Pascal	P II 233
E04	Michel	P133
E05	Georges	P II 233

clé primaire

clé étrangère

Classification

Un système est dit « relationnel minimal » s'il satisfait à trois règles :

- les informations sont représentées par des valeurs dans des tables ;
- les tables ne contiennent aucun pointeur (aussi appelé référence) ;
- les opérateurs algébriques de restriction (sélection de lignes), projection (sélection de colonnes) et jointure (procédé permettant de relier deux tables) doivent être mis en œuvre.

Tous les SGBD du marché respectent ces règles.

Un système est dit complètement relationnel si, outre le respect des règles de minimalité énoncées précédemment, il répond aux deux conditions suivantes :

- les opérateurs algébriques autres que la restriction, la sélection et la jointure naturelle doivent être mis en œuvre ;
- la contrainte d'unicité de clé primaire d'une relation et la contrainte référentielle doivent être vérifiées.

Étant donné le caractère ouvert de la première condition, il est difficile de positionner chaque SGBD du marché. Par exemple, bien que le langage SQL d'Oracle permette les unions, les différences et l'intersection, il ne permet pas de programmer une jointure non symétrique ou une division (opérateur encore indisponible quel que soit le SGBD).

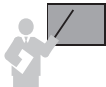
La seconde condition est maintenant respectée par tous les éditeurs (il aura fallu attendre la version 5.1 de MySQL en 2006 pour les tables MyISAM et la version 7 d'Oracle en 1992).

Dépendances fonctionnelles

Le processus de normalisation permet de construire des bases de données relationnelles en évitant les redondances et en préservant l'intégrité des données. Il est préférable de normaliser les relations au moins jusqu'à la troisième forme normale.

La normalisation est basée sur les dépendances fonctionnelles (DF). E.F. Codd fut le premier à publier des écrits sur les DF [COD 72].

Définition



Un attribut b dépend fonctionnellement d'un attribut a si à une valeur de a correspond au plus une valeur de b . La *dépendance fonctionnelle* est notée $a \rightarrow b$.

Le membre droit de l'écriture s'appelle le *dépendant*, le membre gauche s'appelle le *déterminant*.

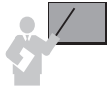
Plusieurs attributs peuvent apparaître dans la partie gauche d'une DF. Dans ce cas, il convient de considérer le couple (si deux attributs figurent dans la partie gauche), le triplet (s'il y a trois attributs), etc. Plusieurs attributs peuvent apparaître dans la partie droite d'une DF. Dans ce

cas, il convient de considérer chaque DF en gardant la partie gauche et en faisant intervenir un seul attribut dans la partie droite.

Supposons les exemples suivants, qui concernent des pilotes ayant un numéro, un nom, une fonction (copilote, commandant, instructeur...) :

- l'écriture $\text{numPilote}, \text{jour} \rightarrow \text{nbHeuresVol}$ est une DF, car à un couple $(\text{numPilote}, \text{jour})$ correspond au plus un nombre d'heures de vol ;
- l'écriture $\text{numPilote} \rightarrow \text{nomPilote}, \text{fonction}$ est équivalente aux écritures $\text{numPilote} \rightarrow \text{nomPilote}$ et $\text{numPilote} \rightarrow \text{fonction}$ qui sont deux DF. En conséquence $\text{numPilote} \rightarrow \text{nomPilote}, \text{fonction}$ est une DF ;
- l'écriture $\text{nomPilote} \rightarrow \text{fonction}$ est une DF s'il n'y a pas d'homonymes dans la population des pilotes enregistrés dans la base de données. Dans le cas contraire, ce n'est pas une DF, car à un nom de pilote peuvent correspondre plusieurs fonctions ;
- l'écriture $\text{fonction} \rightarrow \text{nomPilote}$ n'est pas une DF, car à une fonction donnée correspondent éventuellement plusieurs pilotes.

DF élémentaire

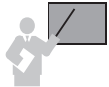


Une DF $a, b \rightarrow c$ est *élémentaire* si ni $a \rightarrow c$, ni $b \rightarrow c$ ne sont des DF.

Considérons les exemples suivants :

- la dépendance fonctionnelle $\text{numPilote}, \text{jour} \rightarrow \text{nbHeuresVol}$ est élémentaire, car $\text{numPilote} \rightarrow \text{nbHeuresVol}$ n'est pas une DF (un pilote vole différents jours, donc pour un pilote donné, il existe plusieurs nombres d'heures de vol), pas plus que $\text{jour} \rightarrow \text{nbHeuresVol}$ (à un jour donné, plusieurs vols sont programmés) ;
- la dépendance fonctionnelle $\text{numPilote}, \text{nomPilote} \rightarrow \text{fonction}$ n'est pas élémentaire, car le numéro du pilote suffit pour retrouver sa fonction ($\text{numPilote} \rightarrow \text{fonction}$ est une DF).

DF directe

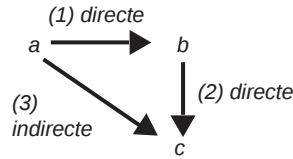


Une DF $a \rightarrow c$ est *directe* si elle n'est pas déduite par transitivité, c'est-à-dire s'il n'existe pas de DF $a \rightarrow b$ et $b \rightarrow c$.

L'expérience montre qu'il est difficile de savoir si une DF est directe, mais il est aisé de voir si elle est indirecte. En conséquence si une DF n'est pas indirecte, elle est directe (ouf !).

Illustrons cette définition par la figure suivante. Soit les trois dépendances fonctionnelles (1), (2) et (3). La dépendance fonctionnelle (3) est non directe et les dépendances (1) et (2) sont directes.

Figure 2-4 Dépendances fonctionnelles directes et non directes



Considérons l'exemple qui fait intervenir les attributs suivants : (immat : numéro d'immatriculation d'un avion ; typeAvion : type de l'aéronef ; nomConst : nom du constructeur de l'aéronef).

- La dépendance $\text{immat} \rightarrow \text{nomConst}$ n'est pas une DF directe.
- La dépendance $\text{immat} \rightarrow \text{typeAvion}$ est une DF directe.
- La dépendance $\text{typeAvion} \rightarrow \text{nomConst}$ est une DF directe.

Propriétés

Les propriétés suivantes sont appelées « axiomes d'Armstrong » ou « règles d'inférence ». L'article [ARM 74] est à l'origine de ces axiomes.



Réflexivité : si b est un sous-ensemble de a alors $a \rightarrow b$ est une DF. Une autre écriture de la réflexivité (appelée « autodétermination ») $a \rightarrow a$ est une DF, cette écriture peut aider à rendre minimal un ensemble de DF.

Augmentation : si $a \rightarrow b$ est une DF, alors $a, c \rightarrow b, c$ est une DF.

Transitivité : si $a \rightarrow b$ et $b \rightarrow c$ sont des DF, alors $a \rightarrow c$ est une DF.

Union : si $a \rightarrow b$ et $a \rightarrow c$ sont des DF, alors $a \rightarrow b, c$ est une DF.

Pseudo-transitivité : si $a \rightarrow b$ et $b, c \rightarrow d$ sont des DF, alors $a, b \rightarrow d$ est une DF.

Décomposition : si $a \rightarrow b, c$ est une DF, alors $a \rightarrow b$ et $a \rightarrow c$ sont des DF.

Fermeture d'un ensemble de DF

Nous avons vu précédemment qu'on pouvait déduire des DF à partir de DF par règles d'inférence.



La **fermeture** F^+ d'un ensemble de DF F est l'ensemble des DF qu'on peut obtenir par applications successives des axiomes d'Armstrong.

La **couverture minimale** C est l'ensemble des DF élémentaires issues de F^+ tel que :

- le membre droit (*dépendant*) de chaque DF ne contient qu'un seul attribut ;
- le membre gauche (*déterminant*) de chaque DF est irréductible ;
- aucune DF ne peut être supprimée.

En considérant seulement $a \rightarrow b$ et $b \rightarrow c$, la fermeture obtenue contient 21 DF ! ($a \rightarrow a$, $b \rightarrow b$, $a \rightarrow c$, $a, c \rightarrow b \dots$).

L'exemple suivant décrit la résolution de la fermeture et de la couverture minimale de l'ensemble de DF $F = \{\text{immat} \rightarrow \text{compagnie}, \text{immat} \rightarrow \text{typeAvion}, \text{typeAvion} \rightarrow \text{capacite}, \text{typeAvion} \rightarrow \text{nomConst}\}$.

On applique la transitivité entre :

- la DF $\text{immat} \rightarrow \text{typeAvion}$ et $\text{typeAvion} \rightarrow \text{nomConst}$ et on obtient la DF $\text{immat} \rightarrow \text{nomConst}$;
- la DF $\text{immat} \rightarrow \text{typeAvion}$ et $\text{typeAvion} \rightarrow \text{capacite}$ et on obtient la DF $\text{immat} \rightarrow \text{capacite}$.

On pourrait appliquer l'union entre $\text{immat} \rightarrow \text{compagnie}$ et $\text{immat} \rightarrow \text{typeAvion}$ de manière à obtenir la DF $\text{immat} \rightarrow \text{typeAvion}, \text{compagnie}$.

De même, l'union entre $\text{typeAvion} \rightarrow \text{capacite}$ et $\text{typeAvion} \rightarrow \text{nomConst}$ donne $\text{typeAvion} \rightarrow \text{capacite}, \text{nomConst}$.

On pourrait appliquer l'augmentation entre chaque DF. Ainsi la DF $\text{immat} \rightarrow \text{compagnie}$ donnerait lieu à l'écriture des DF suivantes :

- $\text{immat}, \text{typeAvion} \rightarrow \text{compagnie}, \text{typeAvion}$
- $\text{immat}, \text{capacite} \rightarrow \text{compagnie}, \text{capacite}$
- $\text{immat}, \text{nomConst} \rightarrow \text{compagnie}, \text{nomConst}$

On pourrait également appliquer les autres propriétés des DF aux DF initiales de manière à obtenir un nombre important de DF pas nécessairement intéressantes. La fermeture de cet ensemble s'écrirait de la sorte :

$F^+ = \{\text{immat} \rightarrow \text{compagnie} ; \text{immat} \rightarrow \text{typeAvion} ; \text{typeAvion} \rightarrow \text{capacite} ; \text{typeAvion} \rightarrow \text{nomConst} ; \text{immat} \rightarrow \text{capacite} ; \text{immat} \rightarrow \text{nomConst} ; \text{immat} \rightarrow \text{typeAvion}, \text{compagnie} ; \text{immat}, \text{typeAvion} \rightarrow \text{compagnie}, \text{typeAvion} ; \text{immat}, \text{capacite} \rightarrow \text{compagnie}, \text{capacite} ; \text{immat}, \text{nomConst} \rightarrow \text{compagnie}, \text{nomConst} ; \dots\}$

Ce qui intéresse le concepteur est l'ensemble des DF composant la couverture minimale de F^+ noté :

$C = \{\text{immat} \rightarrow \text{compagnie} ; \text{immat} \rightarrow \text{typeAvion} ; \text{typeAvion} \rightarrow \text{capacite} ; \text{typeAvion} \rightarrow \text{nomConst}\}$.

Nous étudierons en fin de chapitre deux méthodes de conception de schémas relationnels normalisés basées sur les DF.

Alors que les DF vont servir à classer un schéma relationnel en première, deuxième, troisième ou BCFN (Boyce-Codd forme normale), d'autres formes de dépendances vont permettre de définir les quatrième et cinquième formes normales. Ces familles de dépendances sont les dépendances multivaluées et les dépendances de jointure. Néanmoins, la majorité des schémas relationnels en entreprise sont en deuxième (on dénormalise des relations volontairement pour des contraintes d'optimisation) ou en troisième forme normale.

Dépendances multivaluées

Les dépendances multivaluées (DM) permettent d'appréhender certaines dépendances existant entre attributs d'une même relation.



La DM entre les attributs a et b et c d'une relation R est notée $a \twoheadrightarrow b \mid c$ (a multidétermine b pour c). Pour toute valeur de a , il existe un ensemble de valeurs de b indépendant des valeurs de c .

$$\text{Si } (a_1, b_1, c_1) \in R \text{ et } (a_1, b_2, c_2) \in R \Rightarrow (a_1, b_2, c_1) \in R \text{ et } (a_1, b_1, c_2) \in R$$

La relation suivante décrit une DM $\text{typeAvion} \twoheadrightarrow \text{qualif} \mid \text{nomPilote}$. En effet, les triplets (A320, PP-IFR, Bidal) et (A320, QT-JAR25, Soutou) de la relation `QualifsAvions1` entraînent (A320, PP-IFR, Soutou) et (A320, QT-JAR25, Bidal) dans la même relation.

Le lecteur aura compris que pour qu'un pilote soit lâché (terme aéronautique indiquant qu'un pilote est apte à voler sur une machine) sur A320, il faut qu'il ait à la fois les qualifications PP-IFR et QT-JAR25. Le type de machine A340 requiert quant à lui trois qualifications (PP-IFR, PL et QT-JAR25).

Figure 2-5 Dépendances multivaluées

QualifsAvions1

nomPilote	typeAvion	qualif
Bidal	A320	PP-IFR
Bidal	A320	QT-JAR25
SanFilippo	A340	PL
SanFilippo	A340	QT-JAR25
SanFilippo	A340	FH
Soutou	A320	PP-IFR
Soutou	A320	QT-JAR25

$\text{typeAvion} \twoheadrightarrow \text{qualif} \mid \text{nomPilote}$

Les dépendances multivaluées existent toujours par paires [DAT 00]. En effet, $a \twoheadrightarrow b \mid c$ peut être aussi noté avec les deux écritures $a \twoheadrightarrow b$ et $a \twoheadrightarrow c$.

Dans notre exemple, nous aurions donc les notations des DM $\text{typeAvion} \twoheadrightarrow \text{qualif}$ et $\text{typeAvion} \twoheadrightarrow \text{nomPilote}$. Il semble cependant préférable d'utiliser la première notation ($a \twoheadrightarrow b \mid c$).

Les axiomes d'Armstrong [ARM 74] ont été étendus par [BEE 77] pour les DM. On découvre notamment que les propriétés de réflexivité, d'augmentation, de transitivité, de pseudo-transitivité, d'union et de décomposition peuvent s'appliquer aux DM.

Par ailleurs, une DM $a \rightarrow b$ est triviale si b est un sous-ensemble de a ou si l'union des deux colonnes donne relation entière [TEO 94].

Dépendances de jointure

Les dépendances de jointure (DJ) permettent de résoudre certaines redondances assez rares, qu'on ne peut pas traiter avec les DF et DM, pour mener ainsi à la cinquième forme normale. Le principe est de pouvoir recomposer une relation contenant des redondances à partir de jointures sur n relations ($n > 2$).

Dans l'exemple suivant, la relation **Employer** comporte des redondances au niveau de la qualification du pilote (certaines qualifications sont répétées pour différentes compagnies).

Figure 2-6 Relation **Employer**

Employer

nomPilote	compagnie	qualif
Bidal	Quantas	PP-IFR
Bidal	Quantas	QT-JAR25
Soutou	Singapore AL	PP-IFR
SanFilippo	JAL	PL
Bidal	JAL	QT-JAR25
Bidal	Singapore AL	PP-IFR

On ne peut pas décomposer cette relation en deux relations. Il est nécessaire d'utiliser une troisième relation pour éviter des redondances (voir la figure 2-7). En contrepartie, il faudra mettre en œuvre de nombreuses jointures au niveau du langage SQL pour interroger et manipuler ces relations.

Figure 2-7 Relations définies en extension permettant de reconstituer **Employer**

QualifsAvions2

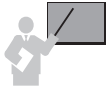
nomPilote	qualif
Bidal	PP-IFR
Bidal	QT-JAR25
Soutou	PP-IFR
SanFilippo	PL

QualifsCompagnies

compagnie	qualif
Quantas	PP-IFR
Quantas	QT-JAR25
Singapore AL	PP-IFR
JAL	PL
JAL	QT-JAR25

Travailler

nomPilote	compagnie
Bidal	Quantas
Bidal	Singapore AL
Bidal	JAL
Soutou	Singapore AL
SanFilippo	JAL



Une *dépendance de jointure* est vérifiée dans une relation R entre les attributs si la jointure¹ de ses projections² sur chaque attribut reconstitue R .

- Soit $R[a,b,c]$, une dépendance de jointure existe si $R = \pi(a) \bowtie \pi(b) \bowtie \pi(c)$.

Bilan des dépendances

Il est utile au concepteur de connaître ces notions de base. En effet, même s'il construit le schéma avec un formalisme entité-association ou UML, il peut toujours en extraire des relations entre attributs et déduire s'il s'agit de DF ou non. Ainsi, il peut concevoir des schémas normalisés. Les clés primaires et étrangères des relations peuvent être également définies à l'aide des DF. Reprenons l'exemple des ordinateurs et des étudiants décrit en début de chapitre. On aurait pu déduire les DF élémentaires et directes suivantes :

- $\text{cpu} \rightarrow \text{prixMoyen}$, $\text{cpu} \rightarrow \text{delaiMoyen}$;
- $\text{codetu} \rightarrow \text{nomEtu}$, $\text{codetu} \rightarrow \text{cpu}$, cette dernière DF permet de statuer sur l'existence d'une clé étrangère dans la relation Etudiant.

La figure 2-8 indique la notation de ce schéma relationnel en intention :

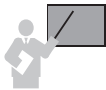
Figure 2-8 Schéma relationnel

```
PC[cpu, prixMoyen, delaiMoyen]
Etudiant[codetu, nomEtu, cpu#]
```

Formes normales

La normalisation permet de construire des schémas optimaux en terme d'intégrité et de cohérence des données stockées.

Première forme normale



Une relation est en *première forme normale* si chaque n -uplet contient une seule valeur par attribut. En d'autres termes, au niveau de l'extension d'une relation, à l'intersection d'une ligne et d'une colonne, on ne doit trouver qu'une et une seule valeur (qui peut être diverse : nombre, chaîne de caractères, date, image, etc.)

1. Notez que la jointure entre deux relations permet de constituer une troisième relation sur la base d'une expression (le plus souvent l'égalité) entre attributs des relations à mettre en jointure.
2. Remarquez que la projection d'une relation R sur un attribut a consiste à ne garder que la colonne a dans la relation résultat.

La relation **VolS1** respecte la règle de la première forme normale. À l'intersection de chaque ligne et pour chaque colonne il n'existe qu'une seule valeur. Il y a cependant des redondances (nom de la compagnie, sexe du pilote et type de l'aéronef). Dans cet exemple, des pilotes peuvent voler pour le compte de différentes compagnies.

Si on devait définir une clé pour la relation **VolS1**, ce serait la concaténation de **ncomp**, **nomPilote**, **immat**. Cette combinaison permet en effet d'identifier chaque ligne. Le nombre d'heures de vol dépend au minimum des valeurs de ces trois attributs.

Figure 2-9 Relations en première forme normale

VolS1

ncomp	compagnie	nomPilote	sexe	typeAvion	immat	nbHeuresVol
1	Air-France	Bidal	F	CRJ	F-CLAR	600
1	Air France	Bidal	F	A320	F-ROMA	345
1	Air France	Bidal	F	A320	F-GLDX	120
1	Air France	Bidal	F	B727	F-CSTU	150
1	Air France	Labat	F	A320	F-GLDX	70
1	Air France	Labat	F	A320	F-ROMA	340
1	Air France	Labat	F	B727	F-CSTU	900
2	Quantas	SanFilippo	G	A330	F-STEF	7500
2	Quantas	SanFilippo	G	A320	F-GLDX	250
2	Quantas	Bidal	F	B727	F-CSTU	2500
2	Quantas	Soutou	G	B747	F-PAUL	750
2	Quantas	Soutou	G	A340	F-ABDL	2500

Les tables relationnelles sont nativement toutes en première forme normale, car les attributs de type tableau ne sont pas autorisés au niveau de la base de données.

Les modèles de données ne respectant pas la première forme normale sont appelés modèles NF2 (Non First Normal Form) [MAK 77]. Le modèle objet-relationnel permet de s'affranchir de la première forme normale.

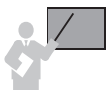
La figure 2-10 présente une relation NF2 équivalente à **VolS1** (il en existe d'autres [SOU 99]). Il existe ici une double imbrication : la collection **Avions** est imbriquée dans la collection **Pilotes**. À l'intersection d'une ligne et d'une colonne, il y a plusieurs valeurs (à l'intersection de la deuxième ligne et de la troisième colonne, il y a SanFilippo, Bidal et Soutou).

Figure 2-10 Relation NF2

VolsNF2

ncomp	compagnie	{Pilotes}				
		nomPilote	sexe	{Avions}		
				typeAvion	immat	nbHeuresVol
1	Air France	Bidal	F	CRJ	F-CLAR	600
				A320	F-ROMA	345
					F-GLDX	120
				B727	F-CSTU	150
		Labat	F	A320	F-GLDX	70
					F-ROMA	340
				B727	F-CSTU	900
2	Quantas	SanFilippo	G	A330	F-STEF	7500
				A320	F-GLDX	250
		Bidal	F	B727	F-CSTU	2500
		Soutou	G	B747	F-PAUL	750
				A340	F-ABDL	2500

Deuxième forme normale



Une relation est en *deuxième forme normale* si elle est en première forme normale, d'une part, et si tout attribut n'appartenant pas à la clé primaire est en dépendance fonctionnelle élémentaire avec la clé, d'autre part.

Cette définition n'est exacte que si la relation ne possède qu'une clé candidate. En effet, considérons la relation $R[a, b, c, d, e]$ pour laquelle deux clés sont candidates (a, b et c, d) avec la DF $a \rightarrow e$. Si on choisit a, b comme clé primaire, la relation $R[\underline{a, b}, c, d, e]$ n'est pas en deuxième forme normale, car dans la DF $a \rightarrow e$, l'attribut e , n'appartenant pas à la clé primaire, n'est pas en dépendance fonctionnelle élémentaire avec la clé. En revanche si on choisit c, d comme clé primaire, la relation $R[\underline{c, d}, a, b, e]$ est en deuxième forme normale mais ne sera pas en troisième forme normale (voir plus loin).

La relation $Avions2$ n'est pas en deuxième forme normale, car la dépendance $immat \rightarrow nomConst$ n'est pas une DF directe. Il suffit en effet de connaître le type de l'aéronef pour en déduire le constructeur. Les DF $immat \rightarrow typeAvion$ et $typeAvion \rightarrow nomConst$ sont directes.

Figure 2-11 Relation n'étant pas en deuxième forme normale

Avions2

immat	typeAvion	nomConst
F-CLAR	CRJ	Canadian Regional Jet
F-ROMA	A320	Airbus
F-GLDX	A320	Airbus
F-CSTU	B727	Boeing
F-STEF	A330	Airbus
F-PAUL	B747	Boeing
F-ABDL	A340	Airbus

Troisième forme normale



Une relation est en *troisième forme normale* si elle respecte la deuxième forme normale et si les dépendances fonctionnelles entre la clé primaire et les autres attributs sont directes.

Le seul inconvénient d'un schéma relationnel en troisième forme normale, c'est qu'il fait intervenir un nombre plus important de relations du fait de la décomposition. Cette inflation entraîne de nombreuses jointures coûteuses lors des interrogations de la base. En contrepartie, il n'existe plus de redondances d'informations, sources d'erreurs lors de manipulations sous SQL (ajout, modification ou suppression d'enregistrements).

Reprenons à présent l'exemple précédent de la relation $R[\underline{c, d}, a, b, e]$ avec la DF $a \rightarrow e$. Cette relation n'est pas en troisième forme normale, car il existe un graphe de transitivité avec les DF suivantes : $c, d \rightarrow a$; $c, d \rightarrow e$ et $a \rightarrow e$. La DF $c, d \rightarrow e$ n'est pas directe. Le schéma en troisième forme normale est donc composé des deux relations $R1[\underline{c, d}, a\#, b]$ et $R2[\underline{a}, e]$.

Considérons à présent l'exemple des vols. Le schéma relationnel en troisième forme normale fait apparaître six relations à partir d'une seule (Vols1). La relation Avions2 a été décomposée en Avions3 et Constructeurs3 de manière à éliminer la DF indirecte $immat \rightarrow nomConst$.

Les colonnes compagnie, sexe et typeAvion ont disparu de la relation Vols3 car les DF :

- $ncomp, nomPilote, immat \rightarrow compagnie$
- $ncomp, nomPilote, immat \rightarrow sexe$
- $ncomp, nomPilote, immat \rightarrow typeAvion$

n'étaient pas minimales en partie gauche. Pour la première DF, par exemple, il suffit de connaître le numéro de compagnie pour avoir le nom de cette dernière. Cette DF se simplifie ainsi en $ncomp \rightarrow compagnie$ (qui donne lieu à la définition de la relation Compagnie3).

Figure 2-12 Relations en troisième forme normale

Compagnie3

ncomp	compagnie
1	Air France
2	Quantas

Pilotes3

nomPilote	sexe
Bidal	F
Labat	F
SanFilippo	G
Soutou	G

Employer3

ncomp#	nomPilote#
1	Bidal
1	Labat
2	SanFilippo
2	Bidal
2	Soutou

Avions3

immat	typeAvion
F-CLAR	CRJ
F-ROMA	A320
F-GLDX	A320
F-CSTU	B727
F-STEF	A330
F-PAUL	B747
F-ABDL	A340

Vols3

ncomp#	nomPilote#	immat#	nbHeuresVol
1	Bidal	F-CLAR	600
1	Bidal	F-ROMA	345
1	Bidal	F-GLDX	120
1	Bidal	F-CSTU	150
1	Labat	F-GLDX	70
1	Labat	F-ROMA	340
1	Labat	F-CSTU	900
2	SanFilippo	F-STEF	7500
2	SanFilippo	F-GLDX	250
2	Bidal	F-CSTU	2500
2	Soutou	F-PAUL	750
2	Soutou	F-ABDL	2500

Constructeurs3

typeAvion	nomConst
CRJ	Canadian Regional Jet
A320	Airbus
B727	Boeing
A330	Airbus
B747	Boeing
A340	Airbus

Forme normale de Boyce-Codd



Une relation est en forme normale de *Boyce-Codd* [COD 74] si elle est en troisième forme normale et que le seul déterminant (membre gauche des dépendances fonctionnelles) existant dans la relation est la clé primaire.

Au niveau conceptuel, cette forme est illustrée aux figures 1-72 et 1-73. L'exemple suivant décrit en extension la relation *ProductionRegion*. Les DF existantes sont :

- typeAvion, pays → region

- $\text{region} \rightarrow \text{pays}$

L'existence de la dernière DF permet de statuer que cette relation n'est pas en forme normale de Boyce-Codd.

Figure 2-13 Relation n'étant pas en forme normale de Boyce-Codd

ProductionRegion

typeAvion	pays	region
A320	France	Blagnac
ATR-42	France	Blagnac
A319	France	Blagnac
A320	Allemagne	Hambourg
A340	Allemagne	Francfort

La décomposition de cette relation en forme normale de Boyce-Codd entraîne la définition d'une relation additionnelle, car la DF à préserver est $\text{region} \rightarrow \text{pays}$

Figure 2-14 Relations en forme normale de Boyce-Codd

AvionRegionBC

typeAvion	region
A320	Blagnac
ATR-42	Blagnac
A319	Blagnac
A320	Hambourg
A340	Francfort

RegionPaysBC

region	pays
Blagnac	France
Hambourg	Allemagne
Francfort	Allemagne

Toute relation a une décomposition sans perte en forme normale de Boyce-Codd. Par contre, une telle décomposition ne préserve pas toutes les DF. Dans notre exemple, la DF $\text{typeAvion}, \text{pays} \rightarrow \text{region}$ a été supprimée mais sans perte, car il est possible de la retrouver en réalisant une jointure entre les deux relations résultantes.

Quatrième forme normale



Une relation est en *quatrième forme normale* si elle respecte la forme normale de Boyce-Codd. S'il n'existe pas de dépendances multivaluées ou si $x \twoheadrightarrow y \mid z$ existe, alors tous les autres attributs $a, b, c, \dots$ de la relation sont en DF avec x ($x \rightarrow a, b, c, \dots$).

Une autre définition [TEO 94] précise qu'une relation est en quatrième forme normale si elle respecte la forme normale de Boyce-Codd. S'il existe une DM $x \twoheadrightarrow y$, elle est soit triviale (voir la section *Dépendances multivaluées*), soit x est clé pour la relation.

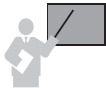
Nous avons donné un exemple de relations en quatrième forme normale dans la section *Dépendances multivaluées* (figure 2-5).

Puisqu'il existe la DM $\text{typeAvion} \twoheadrightarrow \text{qualif} \mid \text{nomPilote}$ dans la relation *QualifsAvions1*, cette relation n'est pas en quatrième forme normale. La décomposition en deux relations, en quatrième forme normale, est détaillée figure 2-15.

Figure 2-15 Relations en quatrième forme normale

QualifsTypeAvions		AvionsPilotes	
typeAvion	qualif	typeAvion	nomPilote
A320	PP-IFR	A320	Bidal
A320	QT-JAR25	A340	SanFilippo
A340	PL	A320	Soutou
A340	QT-JAR25		
A340	FH		

Cinquième forme normale



Une relation est en *cinquième forme normale* (aussi appelée « forme normale de projection-jointure ») si chaque dépendance de jointure respecte la deuxième forme normale et si les dépendances fonctionnelles entre la clé primaire et les autres attributs sont directes.

Approche par décomposition

On trouve dans [BER 76], une première approche de la décomposition en formes normales. Bien que cette approche ne soit pas beaucoup utilisée, il est intéressant de connaître ce processus, car il fait intervenir les DF et permet d'obtenir rapidement un schéma relationnel en troisième forme normale.

Principe

Le principe est de raffiner successivement une relation en formes normales de plus en plus avancées. La relation unique composant le point de départ de la décomposition est appelée « relation universelle ». Selon la littérature relative à cette approche, la relation de départ peut être en première forme normale ou non. On suppose ainsi que les attributs sont tous extraits. Les dépendances fonctionnelles sont à extraire par la suite en examinant les attributs entre eux.

Exemple

Considérons l'exemple des assurances de véhicules. L'extension de la relation est illustrée par la figure 2-16.

Relation universelle

Figure 2-16 Relation universelle

Ventes							
num	nom	{Vehicules}					
		nimm	type	ncons	nomCons	dateAchat	prixAchat
1	Soutou	6748-XW-31	320 i	1	BMW	12-05-1999	62 500
		734-AJH-31	1200 GSF	3	SUZUKI	19-07-2000	46680
		358-ALZ-31	320 Ci	1	BMW	25-01-2001	213 970
2	Bidal	955-NEH-75	Scenic	2	RENAULT	16-01-1995	105 000
		52-AIM-31	Beetle	4	VOLKSWAGEN	16-01-2002	109 500

On va maintenant procéder à la décomposition de cette relation de manière à obtenir un schéma normalisé. Si la relation n'est pas en première forme normale, il faut *aplatir* cette relation de manière à supprimer tous les attributs multivalués.

Première forme normale

L'aplatissement de cette relation donne la relation *Ventes1* en première forme normale suivante. En partant de l'hypothèse qu'un assuré puisse vendre puis racheter sa voiture, la clé de cette relation sera la concaténation des attributs *num*, *nimm* et *dateAchat*.

Figure 2-17 Relation aplatie

Ventes1							
num	nom	nimm	type	ncons	nomCons	dateAchat	prixAchat
1	Soutou	6748-XW-31	320 i	1	BMW	12-05-1999	62 500
1	Soutou	734-AJH-31	1200 GSF	3	SUZUKI	19-07-2000	46680
1	Soutou	358-ALZ-31	320 Ci	1	BMW	25-01-2001	213 970
2	Bidal	955-NEH-75	Scenic	2	RENAULT	16-01-1995	105 000
2	Bidal	52-AIM-31	Beetle	4	VOLKSWAGEN	16-01-2002	109 500

Deuxième forme normale

La question à se poser ensuite sera : « De quoi dépend élémentairement chaque attribut de la relation ? ».

Considérons tous les attributs :

- prixAchat dépend de la clé (num, nimm, dateAchat $\rightarrow$ prixAchat) ;
- ncons, type, nomCons dépendent du numéro d'immatriculation (nimm $\rightarrow$ ncons, type, nomCons) ;
- nom dépend du numéro d'assuré (num $\rightarrow$ nom).

Les relations déduites de ces DF sont par définition en deuxième forme normale.

Figure 2-18 Relations en deuxième forme normale

Ventes2

num#	nimm#	dateAchat	prixAchat
1	6748-XW-31	12-05-1999	62 500
1	734-AJH-31	19-07-2000	46680
1	358-ALZ-31	25-01-2001	213 970
2	955-NEH-75	16-01-1995	105 000
2	52-AIM-31	16-01-2002	109 500

Assures2

num	nom
1	Soutou
2	Bidal

Vehicules2

nimm	type	ncons	nomCons
6748-XW-31	320 i	1	BMW
734-AJH-31	1200 GSF	3	SUZUKI
358-ALZ-31	320 Ci	1	BMW
955-NEH-75	Scenic	2	RENAULT
52-AIM-31	Beetle	4	VOLKSWAGEN

Troisième forme normale

Il faut à présent répondre à la question : « De quoi dépend directement chaque attribut de la relation ? ».

Cherchons les DF indirectes. On trouve un graphe de transitivité avec (nimm $\rightarrow$ nomCons, ncons $\rightarrow$ nomCons, nimm $\rightarrow$ nomCons) dans lequel la DF nimm $\rightarrow$ nomCons est indirecte car issue d'une transitivité. Cette dépendance est donc à supprimer, ce qui revient à enlever l'attribut nomCons de la relation Vehicules2. En contrepartie, il faut conserver les DF directes nimm $\rightarrow$ ncons et ncons $\rightarrow$ nomCons. La première DF directe est représentée car l'attribut reste dans la relation Vehicules2, qu'on renomme pour l'occasion en Vehicules3. Afin de représenter la deuxième DF directe, il faut définir une relation Constructeurs3. On renomme aussi Ventes2 en Ventes3 et Assures2 en Assures3.

Les relations déduites de ces DF sont par définition en troisième forme normale.

Figure 2-19 Relations en troisième forme normale

Ventes3				Assures3	
num#	nimm#	dateAchat	prixAchat	num	nom
1	6748-XW-31	12-05-1999	62 500	1	Soutou
1	734-AJH-31	19-07-2000	46680	2	Bidal
1	358-ALZ-31	25-01-2001	213 970		
2	955-NEH-75	16-01-1995	105 000		
2	52-AIM-31	16-01-2002	109 500		

Vehicules3			Constructeurs3	
nimm	type	ncons#	ncons	nomCons
6748-XW-31	320 i	1	1	BMW
734-AJH-31	1200 GSF	3	3	SUZUKI
358-ALZ-31	320 Ci	1	2	RENAULT
955-NEH-75	Scenic	2	4	VOLKSWAGEN
52-AIM-31	Beetle	4		

On peut ainsi poursuivre le raffinage en des formes normales plus avancées, en examinant l'existence éventuelle de dépendances multivaluées et dépendances de jointure. Dans cet exemple, il n'y a pas lieu de poursuivre la normalisation.

Approche par synthèse

On trouve dans [WON 78] une première approche de normalisation par synthèse. Ce processus est intéressant, car on peut s'en servir en conception ou dans le cas d'une base de données déjà existante qu'on désire faire évoluer. On suppose que les dépendances fonctionnelles sont toutes extraites. On affine ensuite ces DF en les regroupant de manière à aboutir à un schéma relationnel normalisé.

Principe

À partir d'un ensemble $F = \{a \rightarrow b ; b \rightarrow c ; \dots\}$ de DF, il faut :

- Supprimer les DF redondantes (déduites de transitivité et non minimales en partie gauche). Pour ce faire, on peut utiliser les propriétés des DF vues précédemment (réflexivité, augmentation...).

- Regrouper les DF ayant la même partie gauche dans des sous-ensembles F_i (les DF de type $a \rightarrow b$ et $b \rightarrow a$ doivent être regroupées dans le même sous-ensemble et on doit choisir parmi les deux attributs lequel sera la clé et lequel sera la clé candidate). On déduira les relations de ces sous-ensembles. Dans le même temps les clés étrangères seront extraites.
- S'il existe des attributs qui ne sont ni source ni cible d'aucune DF, il faut les regrouper au sein d'une relation dont tous les attributs seront clés.

Exemple

À partir de l'ensemble suivant constitué de sept DF, $F = \{a \rightarrow b^1; a \rightarrow c^2; a, b, h \rightarrow e, g^3; h \rightarrow j^4, j \rightarrow k^5; h \rightarrow k^6, b \rightarrow a^7\}$, nous allons suivre les étapes énoncées précédemment.

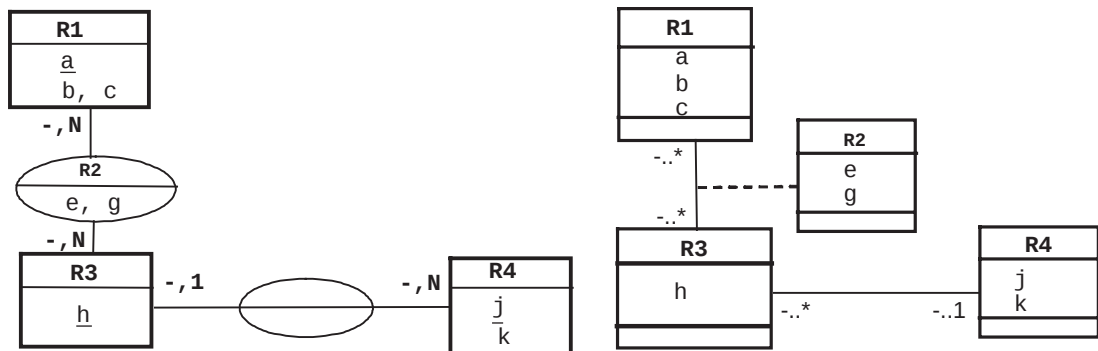
- Supprimer les DF redondantes déduites de transitivité : la dépendance 6 peut être supprimée car elle est déduite de 4 et 5.
- Supprimer les DF redondantes non minimales en partie gauche : la dépendance 3 peut être réduite à $a, h \rightarrow e, g$ à cause de la dépendance 1. Par symétrie, on aurait pu réduire à $b, h \rightarrow e, g$ à cause de la dépendance 7, cependant il est préférable de garder a en partie gauche du fait de l'existence de la DF 2.
- Regrouper les DF ayant la même partie gauche dans des sous-ensembles : $F_1 = \{a \rightarrow b^1; a \rightarrow c^2; b \rightarrow a^7\}$; $F_2 = \{a, h \rightarrow e, g^3\}$; $F_3 = \{h \rightarrow j^4\}$; $F_4 = \{j \rightarrow k^5\}$.

Les relations déduites sont les suivantes. On peut en même temps définir les clés étrangères.

- R1[a, b, c]
- R2[a#, h#, e, g]
- R3[h, j#]
- R4[j, k]

Il est intéressant d'en déduire les schémas conceptuels Merise et UML. Le symbole « - » au niveau des cardinalités minimales signifie qu'il peut s'agir d'un 0 ou d'un 1.

Figure 2-20 Schémas conceptuels équivalents



Bilan

Nous avons étudié les aspects du modèle relationnel intéressant la conception. Il existe deux autres aspects importants du modèle relationnel. Le premier concerne l'intégrité des données, que nous verrons au chapitre suivant quand nous traiterons de la traduction des différentes contraintes avec SQL. Le second couvre l'aspect de manipulation au sens des opérateurs relationnels, qui sort du sujet de ce livre.

Les méthodes de conception basées sur l'analyse des dépendances fonctionnelles sont rarement mises en œuvre dans le milieu industriel. La démarche de conception la plus courante consiste en effet à élaborer un diagramme conceptuel, puis à le transformer dans le modèle de données par un outil informatique, qui utilise les règles que nous verrons à la fin de ce chapitre. La connaissance des DF et de leurs caractéristiques peut néanmoins aider à la validation en amont et à l'évolution d'un modèle de données.

Modèles objet

En se plaçant à un niveau logique, on ne peut pas à proprement parler d'un seul modèle de données objet. La littérature en a proposé de nombreux qui proviennent des modèles de données sémantiques. Nous serons plus pragmatiques en présentant d'une part comment la notation UML peut modéliser un schéma logique. D'autre part, par analogie au modèle relationnel, nous présenterons une syntaxe qui permet de décrire un schéma de base de données objet-relationnel ou objet.

Notation UML

La notation UML (Rational Rose parle de profil UML pour les bases de données) permet de modéliser un schéma relationnel (le diagramme de classes représentant un ensemble de tables). Pour préciser qu'une classe représentera une table, on utilise le stéréotype <<Table>>. La classe contient des attributs. On peut relier plusieurs classes entre elles en prenant garde d'insérer convenablement les clés étrangères (nous verrons les règles à employer à la fin de ce chapitre). Il est aussi possible d'utiliser les agrégations pour renforcer le couplage d'une association.

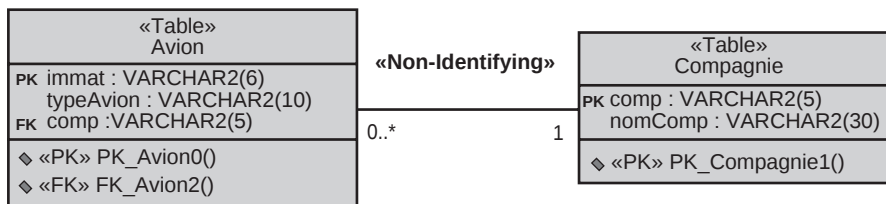
Association un-à-plusieurs

L'exemple 2-21 illustre une association *un-à-plusieurs* au niveau logique entre deux tables (copie d'écran de Rational Rose). Les clés primaires et étrangères apparaissent. Les noms des contraintes sont situés dans le troisième compartiment des classes.



Les règles de normalisation étudiées précédemment peuvent être appliquées manuellement en examinant chaque attribut des classes du diagramme.

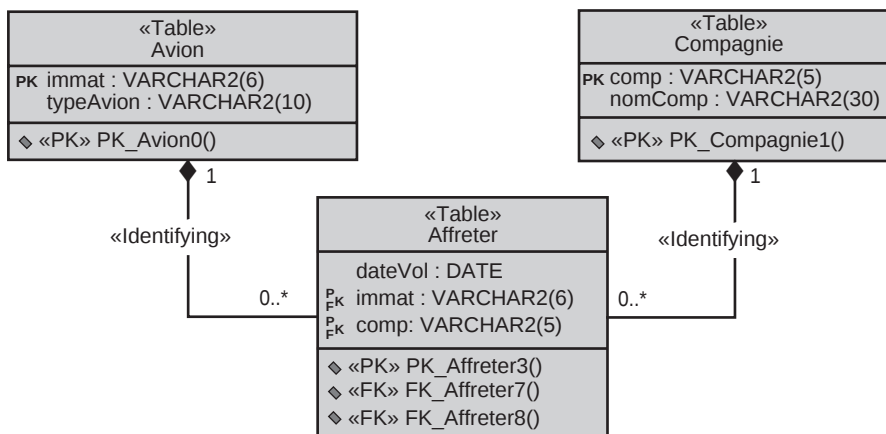
Figure 2-21 Association logique un-à-plusieurs avec UML



Association plusieurs-à-plusieurs

L'exemple 2-22 décrit l'association *plusieurs-à-plusieurs* par une troisième classe (Affreter). Deux agrégations précisent qu'il ne peut y avoir d'affrètement s'il n'existe pas de compagnie ou d'avion correspondant.

Figure 2-22 Association N-N avec UML



Les concepts objet au niveau logique

Cette section décrit une notation objet au niveau logique en faisant une analogie au modèle relationnel. Bien que nous ne proposons pas ici de normalisation formelle comme il en existe dans le monde relationnel, cette notation peut être considérée comme une aide au passage d'un modèle conceptuel objet à un modèle de base de données tout en limitant les risques d'incohérences.

Avec l'objet, les attributs d'une classe peuvent être simples (chaîne de caractères, nombre etc.), de type structure (composition d'attributs), de type référence (au sens Java du terme : adresse d'un autre objet), de type collection de structures ou collection de références.

Structure et référence

L'exemple 2-23 illustre deux modélisations. La structure de données de nom adr est utilisée dans la première figure. Dans la seconde figure , la référence ref_adr relie l'objet à un autre objet de la classe Adresse.

Figure 2-23 Structure et référence

Chercheur

numero	nom	adr		
		norue	rue	ville
15	E.F. Codd	1020	Broadway	New York
...				

Chercheur

numero	nom	ref_adr
15	E.F. Codd	
...		

Adresse		
norue	rue	ville
1020	Broadway	New York
...		

Les notations au niveau logique de ces exemples sont les suivantes.

Figure 2-24 Représentation logique d'un schéma objet

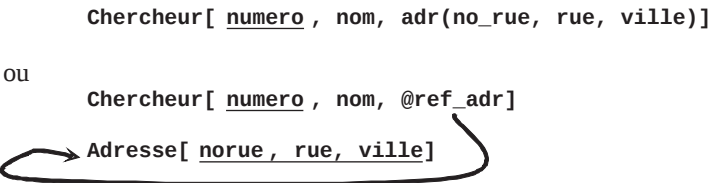
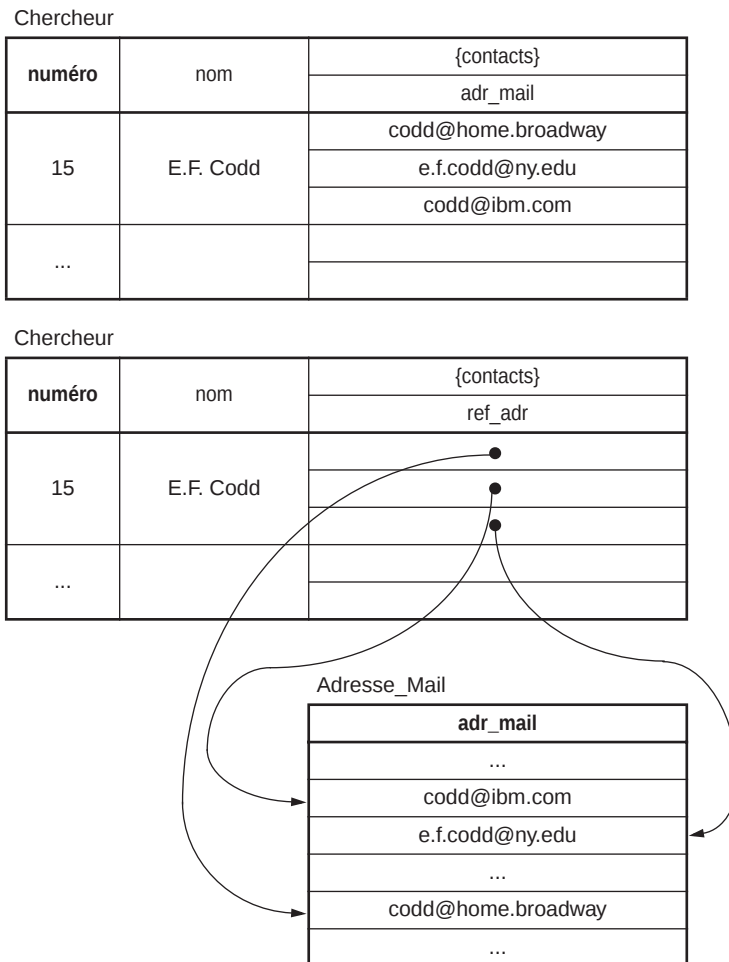


Figure 2-25 Collections



Les notations au niveau logique de ces exemples sont les suivantes.

Figure 2-26 Représentation logique d'un schéma objet

```

Chercheur[ numero , nom, contacts{adr_mail}]

ou

Chercheur[ numero , nom, contacts{@ref_adr} ]

→ Adresse_Mail [ adr_mail ]
    
```

Du conceptuel au logique

Cette section présente les règles permettant de décrire un schéma logique dans les modèles relationnel et objet-relationnel à partir d'un schéma entité-association ou à partir d'un diagramme de classes UML.

D'un schéma entité-association/UML vers un schéma relationnel

Nous donnons ci-après quatre règles (de R1 à R4) pour traduire un schéma conceptuel entité-association ou UML en un schéma relationnel équivalent. Il existe d'autres solutions de transformation, mais ces règles sont les plus simples et les plus opérationnelles.

Transformation des entités/classes



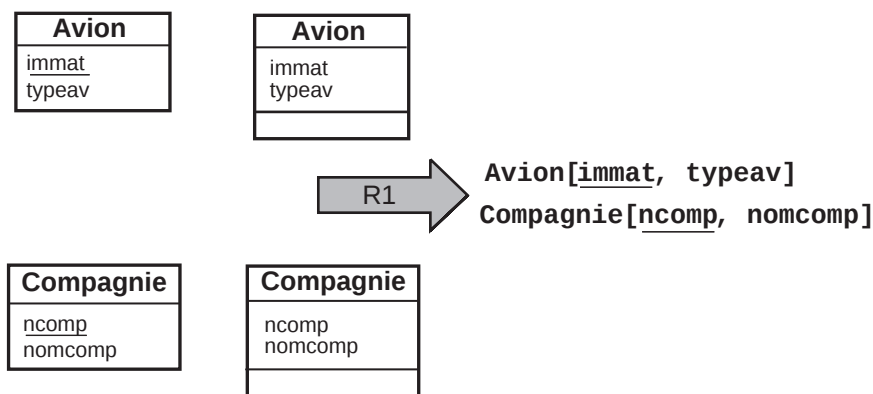
Chaque entité devient une relation. L'identifiant de l'entité devient clé primaire pour la relation.

Chaque classe du diagramme UML devient une relation. Il faut choisir un attribut de la classe pouvant jouer le rôle d'identifiant.

Si aucun attribut ne convient en tant qu'identifiant, il faut en ajouter un de telle sorte que la relation dispose d'une clé primaire (les outils proposent l'ajout de tels attributs).

La règle R1 a été appliquée à l'exemple 2-27 de manière à dériver deux relations.

Figure 2-27 Transformation d'entités/classes



La plupart des entités (classes) temporelles ne doivent pas se transformer en relation (exemple figure 2-30)

Transformation des associations

Les règles de transformation que nous allons voir dépendent des cardinalités/multiplicités maximales des associations. Nous distinguons trois familles d'associations :

- *un-à-plusieurs* ;
- *plusieurs-à-plusieurs* ou classes-associations, et *n-aires* ;
- *un-à-un*.

Associations *un-à-plusieurs*

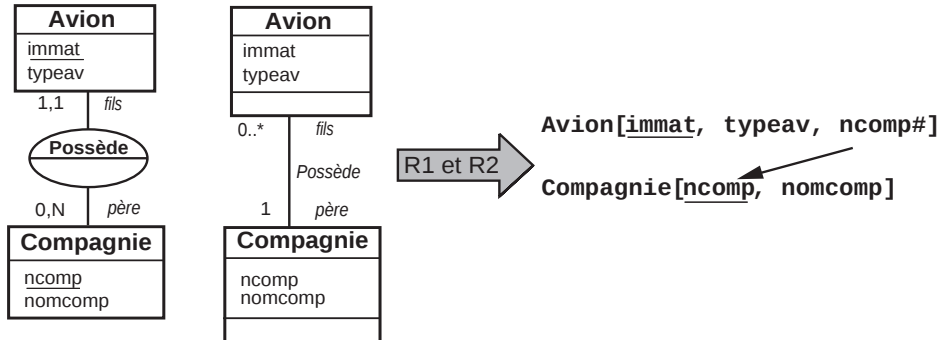


Il faut ajouter un attribut de type clé étrangère dans la relation *Avion* de l'association. L'attribut porte le nom de la clé primaire de la relation *Compagnie* de l'association.

On peut se rappeler cette règle de la manière suivante : *la clé de la relation père migre dans la relation fils*.

Les règles R1 et R2 ont été appliquées à l'exemple 2-28. La règle R2 fait apparaître la clé étrangère *ncomp* dans la relation *Avion* qui a migré de la relation *père* *Compagnie*.

Figure 2-28 Transformation d'une association *un-à-plusieurs*



Associations *plusieurs-à-plusieurs* et *n-aires*

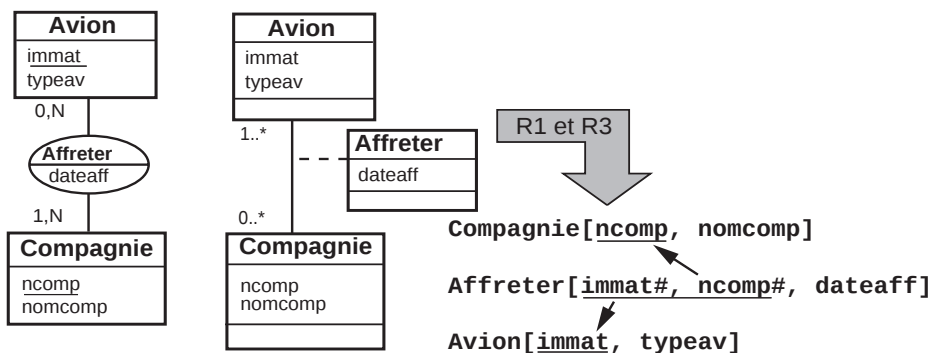


L'association (classe-association) devient une relation dont la clé primaire est composée par la concaténation des identifiants des entités (classes) connectés à l'association. Chaque attribut devient clé étrangère si l'entité (classe) connectée dont il provient devient une relation en vertu de la règle R1.

Les attributs de l'association (classe-association) doivent être ajoutés à la nouvelle relation. Ces attributs ne sont ni clé primaire, ni clé étrangère.

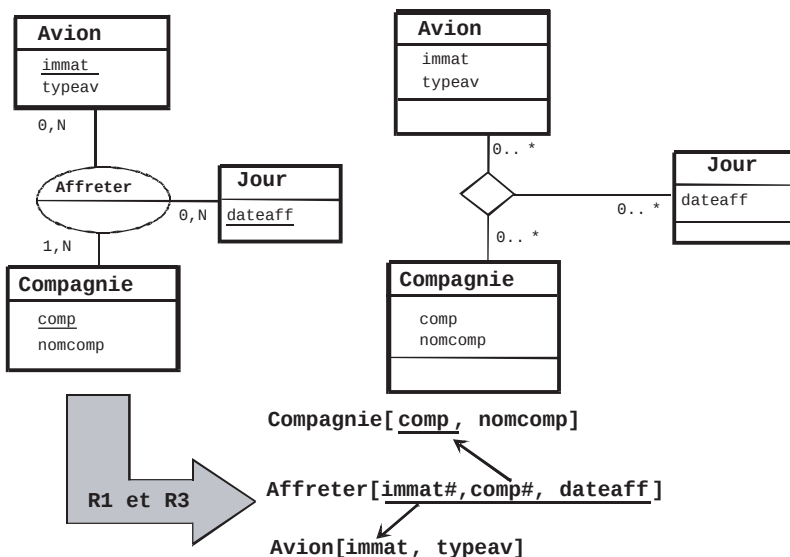
Les règles R1 et R3 ont été appliquées à l'exemple 2-29. La règle R3 crée la relation **Affreter** dont la clé primaire est composée de deux clés étrangères. L'attribut **dateaff** de l'association est ajouté à la nouvelle relation.

Figure 2-29 Transformation d'une association plusieurs-à-plusieurs



Les règles R1 et R3 sont appliquées à l'association 3-aire **Affreter**. On ne dérive pas la relation **Jour** car c'est une entité (classe) temporelle. En conséquence, l'attribut **dateaff** ne devient pas une clé étrangère, mais il est nécessaire pour composer la clé primaire de la relation modélisant les affrètements.

Figure 2-30 Transformation d'une association n-aire



Associations un-à-un

La règle est la suivante, elle permet d'éviter les valeurs NULL dans la base de données.

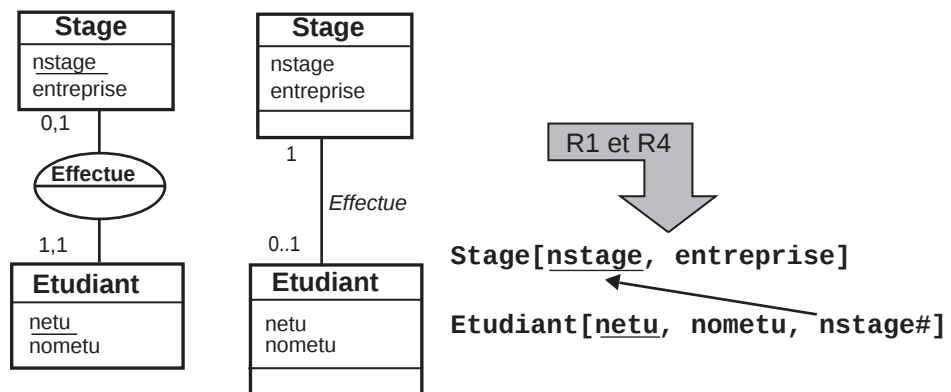


Il faut ajouter un attribut clé étrangère dans la relation dérivée de l'entité ayant la cardinalité minimale égale à zéro. Dans le cas de UML, il faut ajouter un attribut clé étrangère dans la relation dérivée de la classe ayant la multiplicité minimale égale à un. L'attribut porte le nom de la clé primaire de la relation dérivée de l'entité (classe) connectée à l'association.

Si les deux cardinalités (multiplicités) minimales sont à zéro, le choix est donné entre les deux relations dérivées de la règle R1. Si les deux cardinalités minimales sont à un, il est sans doute préférable de fusionner les deux entités (classes) en une seule.

Les règles R1 et R4 sont appliquées à l'exemple 2-31.

Figure 2-31 Transformation d'une association un-à-un



Transformation de l'héritage

Trois décompositions sont possibles pour traduire une association d'héritage en fonction des contraintes existantes :

- décomposition par distinction ;
- décomposition descendante (*push-down*). S'il existe une contrainte de totalité ou de partition sur l'association d'héritage, il y aura deux cas possibles de décomposition ;
- décomposition ascendante (*push-up*).

Nous verrons plus loin que l'héritage multiple se traduit également au niveau logique à l'aide de ces principes. Nous utilisons ici des diagrammes UML que le lecteur pourra aisément traduire sous Merise/2.

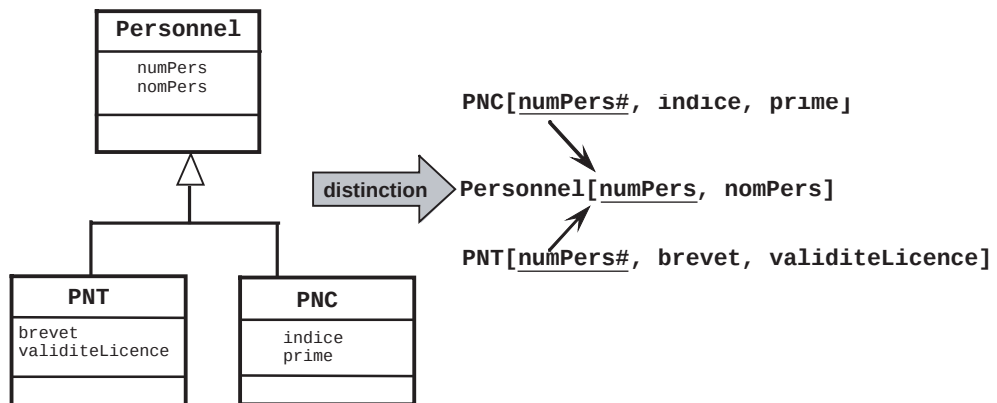
Décomposition par distinction



Il faut transformer chaque sous-classe en une relation. La clé primaire de la sur-classe migre dans la (les) relation(s) issue(s) de la (des) sous-classe(s) et devient à la fois clé primaire et clé étrangère.

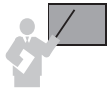
L'exemple 2-32 applique la règle R1. L'association d'héritage est traduite en faisant migrer l'identifiant de la sur-classe (*Personnel*) dans les deux relations déduites des sous-classes. Cet attribut devient aussi clé étrangère.

Figure 2-32 Décomposition par distinction d'une association d'héritage



Décomposition descendante (push-down)

Deux cas sont possibles :

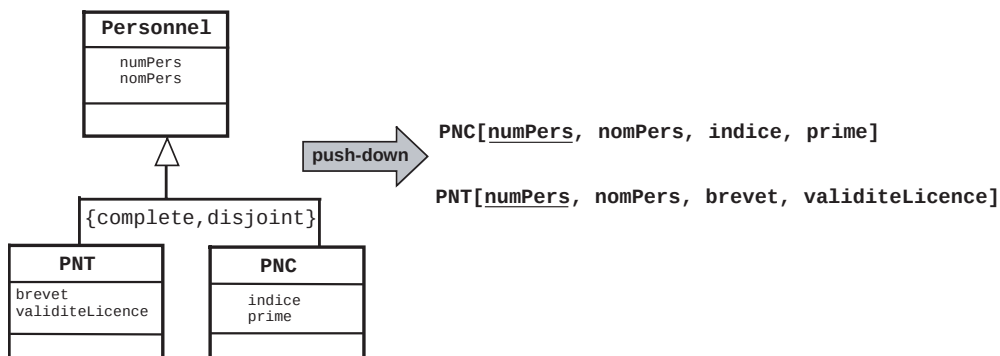


- S'il existe une contrainte de totalité ou de partition sur l'association, il est possible de ne pas traduire la relation issue de la sur-classe. Il faut alors faire migrer tous ses attributs dans la (les) relation(s) issue(s) de la (des) sous-classe(s).
- Dans le cas contraire, il faut faire migrer tous ses attributs dans la ou les relation(s) issue(s) de la (des) sous-classe(s) dans la (les) relation(s) issue(s) de la (des) sous-classe(s).

L'exemple 2-33 décrit une contrainte de partition dans l'association d'héritage (aucun personnel ne peut être à la fois PNT et PNC et il n'existe pas un personnel n'étant ni PNT ni PNC). Les deux relations héritent du contenu intégral de la relation issue de la sur-classe (*Personnel*). La relation *Personnel* n'apparaît plus au niveau logique et n'est pas nécessaire, car aucun

personnel ne peut être ni PNT ni PNC. Ici, on peut dire que `Personnel` est une entité abstraite (il n'existera pas d'instance de cette entité).

Figure 2-33 Décomposition descendante (push-down) d'une association d'héritage



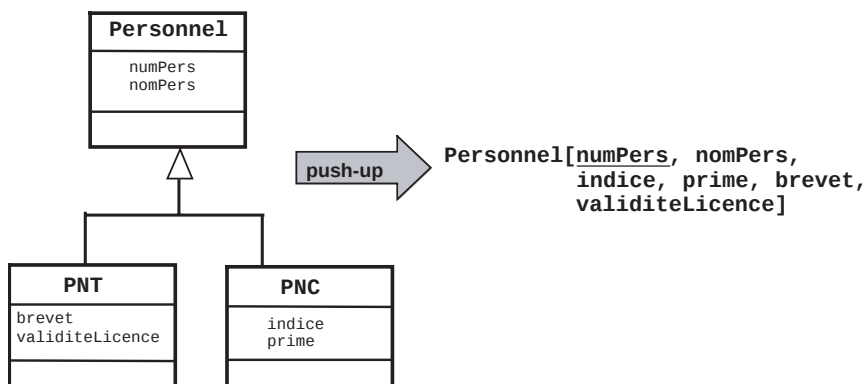
Décomposition ascendante (push-up)



Il faut supprimer la (les) relation(s) issue(s) de la (des) sous-classe(s) et faire migrer les attributs dans la relation issue de la sur-classe.

L'exemple 2-34 décrit une association d'héritage UML sans contrainte. En appliquant le principe de décomposition ascendante, nous obtenons une relation issue de la sur-classe dans laquelle se trouve le contenu des relations issues des sous-classes.

Figure 2-34 Décomposition ascendante (push-up) d'une association d'héritage

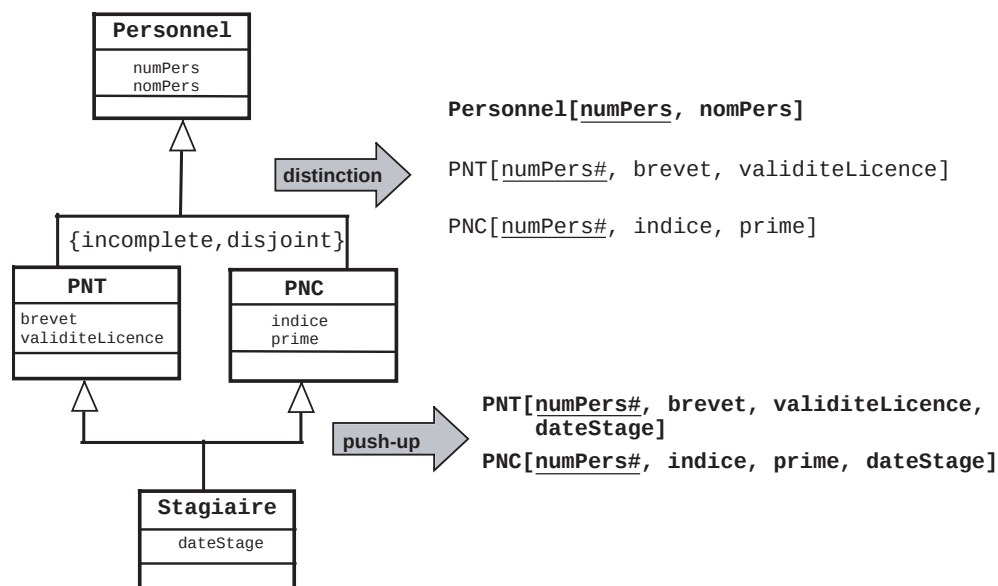


Transformation de l'héritage multiple

Les trois décompositions que nous avons étudiées peuvent s'appliquer à l'héritage multiple. Chaque association présente dans la hiérarchie pourra être traduite par distinction, de manière descendante ou de manière ascendante. Nous discuterons des avantages et des inconvénients de chaque décomposition dans le chapitre suivant (SQL).

Ainsi, pour le schéma 2-35, il est possible de combiner différentes décompositions pour le même graphe d'héritage. Dans notre exemple, nous choisissons d'utiliser la décomposition par distinction pour le premier niveau d'héritage, et la décomposition ascendante (*push-up*) pour le second niveau d'héritage. Les relations finales sont notées en gras.

Figure 2-35 Décomposition d'une association d'héritage multiple



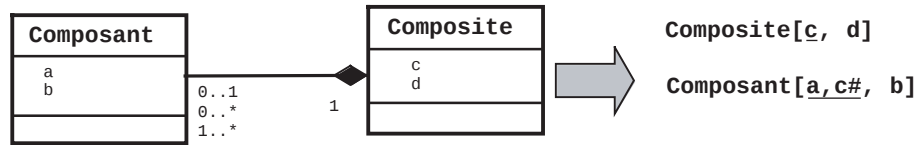
Agrégations UML

Alors que l'agrégation partagée de UML se traduit au niveau logique comme une simple association, il n'en est pas de même pour la composition. L'exemple 2-36 décrit le schéma relationnel déduit d'une composition (on suppose que l'attribut *a* identifie la classe composante et que l'attribut *c* identifie la classe composite).



La clé primaire des relations déduites des classes composantes doit contenir l'identifiant de la classe composite (quelles que soient les multiplicités).

Figure 2-36 Transformation d'une composition



D'un schéma entité-association/UML vers un schéma objet

Dans cette section, nous expliquons des règles (numérotées de R5 à R9) de passage d'un schéma conceptuel (entité-association ou UML) à un schéma objet (ou objet-relationnel) convenant à des bases de données de type SQL3.

Transformation des entités



Chaque entité devient une classe. L'identifiant de l'entité devient celui de la classe.

Transformation des associations

Les règles de transformation dépendent des cardinalités (multiplicités) maximales des associations. Nous distinguons trois familles d'associations *un-à-plusieurs*, *plusieurs-à-plusieurs* (ou classes-associations et *n-aires*) et *un-à-un*.

Associations *un-à-plusieurs*



Une association *un-à-plusieurs* peut se modéliser de trois manières dans un modèle objet :

- la classe *fils* migre dans la classe *père* et devient une collection ;
- avec deux classes dont l'une contient éventuellement une collection avec ou sans références ;
- avec trois classes ne contenant aucune collection.

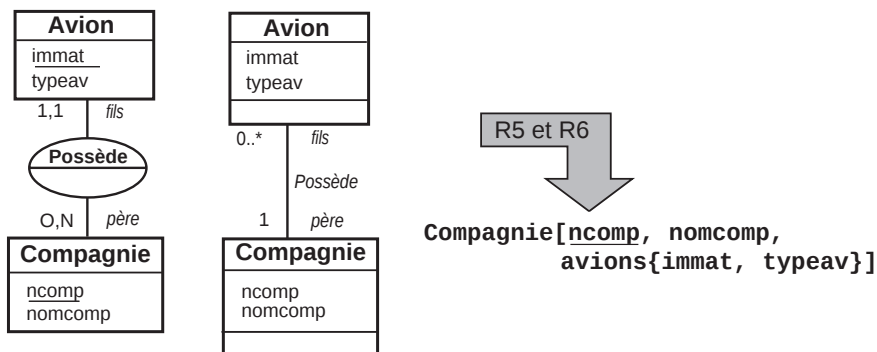
L'exemple 2-37 illustre la première possibilité de transformation.



Notez les collections entre accolades *uneCollection{liste d'attributs}* et les références *@ref_nomRelationCible*.

La classe *Avion* est ici transformée en une collection nommée *avions* située dans la classe *Compagnie*. Cette collection est composée de deux attributs.

Figure 2-37 Transformation d'une association un-à-plusieurs



Cette structure de données imbriquée rappelle le modèle des SGBD hiérarchiques. Elle présente un inconvénient majeur au niveau de SQL dans la mesure où l'ajout d'un avion qui n'est pas encore propriété d'une compagnie sera impossible. Cette solution est seulement adaptée aux associations pour lesquelles les objets *fils* n'ont pas d'existence propre.

Lors du bilan, nous reviendrons sur ce cas de transformation et expliquerons dans quels cas il convient de l'employer.

Les schémas de la figure 2-38 illustrent la deuxième possibilité de transformation avec une collection sans référence. Les solutions sont basées sur une collection dans la classe *père*. La table *fils* contient ou non un lien inverse. Dans notre exemple, la classe *Compagnie* héberge la collection *flotte* qui ne contient que l'identifiant *immat* de la table *Avion*. Il faudra définir aussi la classe *Avion* qui peut éventuellement contenir une clé étrangère *ncomp#* réalisant un lien inverse. L'inconvénient majeur réside dans la gestion des références inverses lorsqu'elles existent.

Figure 2-38 Transformation avec deux classes sans référence

Compagnie[ncomp, nomcomp, flotte{immat}]

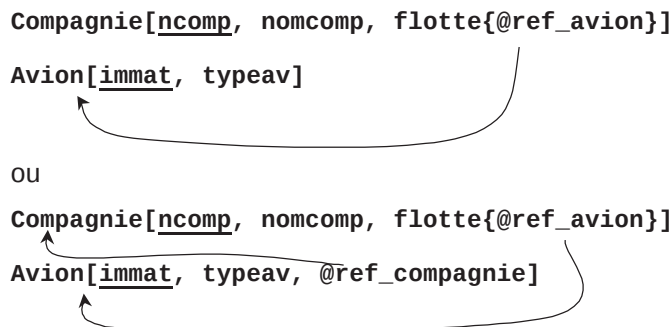
Avion[immat, typeav]

OU

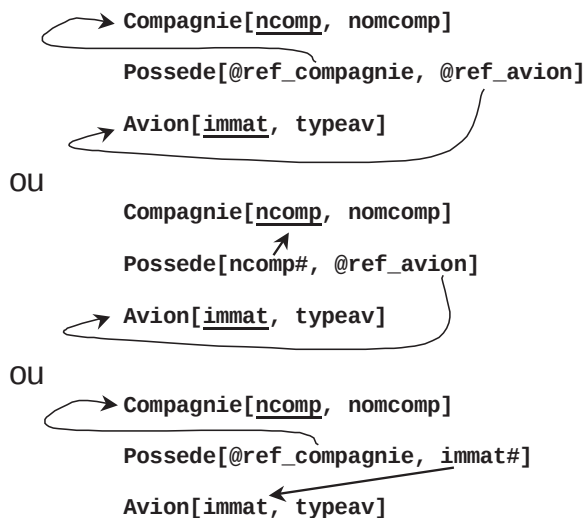
Compagnie[ncomp, nomcomp, flotte{immat}]

Avion[immat, typeav, ncomp#]

Les schémas de la figure 2-39 illustrent la deuxième possibilité de transformation avec une collection de références.

Figure 2-39 Transformation avec deux classes et références

Les schémas de la figure 2-40 décrivent la troisième possibilité de transformation. Une troisième classe est définie, elle contient soit deux références (une vers chaque classe à mettre en association), soit une référence et un attribut de type clé étrangère (deux possibilités).

Figure 2-40 Transformation avec trois classes avec références

Pour une unique solution avec le modèle relationnel, nous avons répertorié huit solutions avec le modèle objet (et il en existe d'autres !). Le fait d'utiliser des collections va privilégier l'accès aux données par l'une ou l'autre des classes.

Associations *plusieurs-à-plusieurs*

Les possibilités que nous retenons sont les suivantes :



Une association *plusieurs-à-plusieurs* entre deux entités (classes) peut se modéliser de deux manières dans un modèle objet :

- avec deux classes, dont l'une contenant une collection de références ;
- avec une troisième classe contenant éventuellement une collection de références.

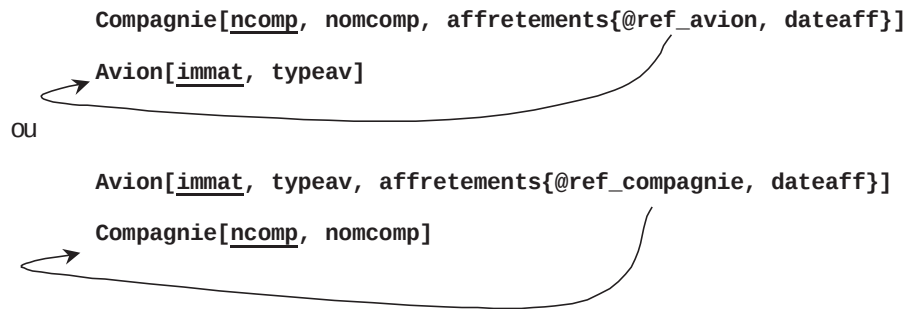
Il est aussi possible de privilégier chaque attribut de l'association [SOU 04]. L'exemple que nous utilisons est celui des affrètements d'avions par des compagnies à une date donnée (figure 2-30).

Les schémas basés sur deux classes décrits à la figure 2-41 consistent à définir dans la classe privilégiée (au niveau de l'accès aux données) une collection de références composée des attributs `dateaff` et d'une référence vers la classe à mettre en association.



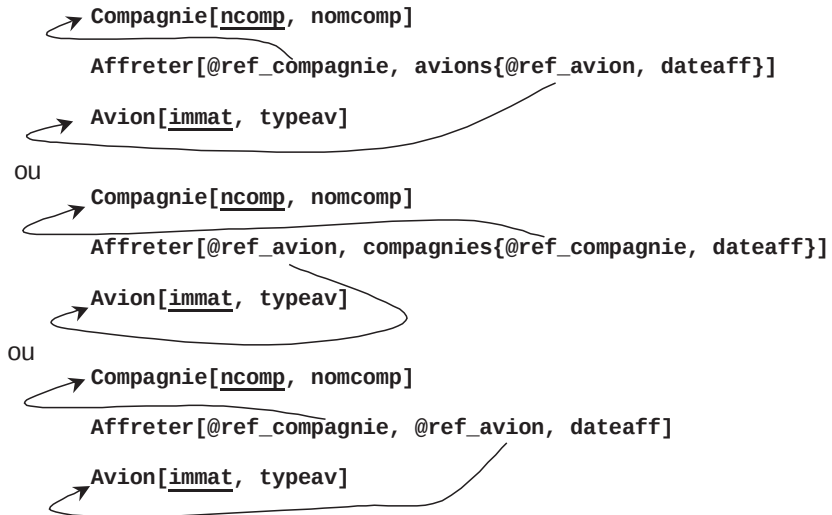
On choisira d'utiliser une collection de références si on désire privilégier l'accès par telle ou telle autre classe. Dans le cas inverse, la solution mettant en œuvre trois classes sans collection doit être préférée.

Figure 2-41 Transformation avec une collection de références



Les autres solutions consistent à définir une troisième classe. Dans notre exemple, la troisième classe est `Affreter`. Elle contient tantôt la collection `avions`, tantôt la collection `compagnies`. Ces collections sont composées des attributs de l'association (ici `dateaff`) et d'une référence (ou de plusieurs si l'association est *n*-aire) vers la classe à mettre en association.

Figure 2-42 Transformation avec trois classes



On choisira une des deux premières solutions si on désire privilégier l'accès par telle ou telle autre classe. Si on ne désire privilégier aucune des classes, alors on utilisera la dernière solution (celle qui se rapproche le plus du modèle relationnel).

Associations n -aires

Le raisonnement est similaire à celui des associations précédentes. Il convient d'utiliser ou non des collections de références. Ces collections contiendront les attributs de l'association et des références vers les classes à relier.

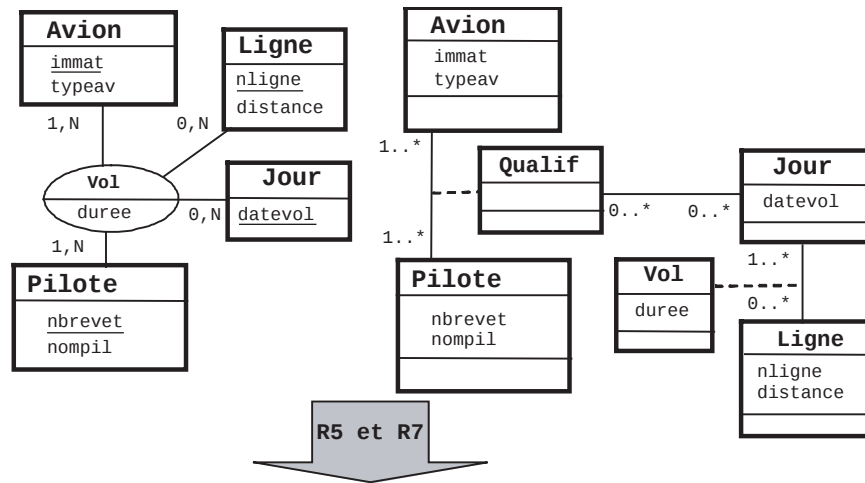


Une association n -aire peut se modéliser de plusieurs manières dans un modèle objet :

- avec n classes dont l'une contient une collection de références ;
- avec une $n+1^e$ classe contenant éventuellement une collection de références.

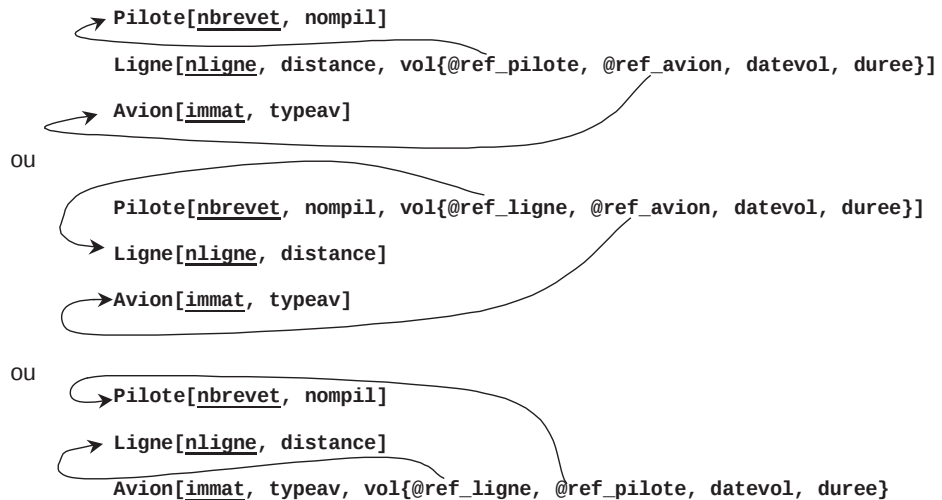
Dans l'exemple 2-43, il devrait y avoir deux bases de travail (en utilisant soit quatre classes, soit cinq). L'entité (classe) **Jour** ne donnant pas lieu à la création d'une classe (règle R5), deux bases de travail seront à étudier (en utilisant soit trois classes, soit quatre classes).

Figure 2-43 Exemple d'une association 4-aire



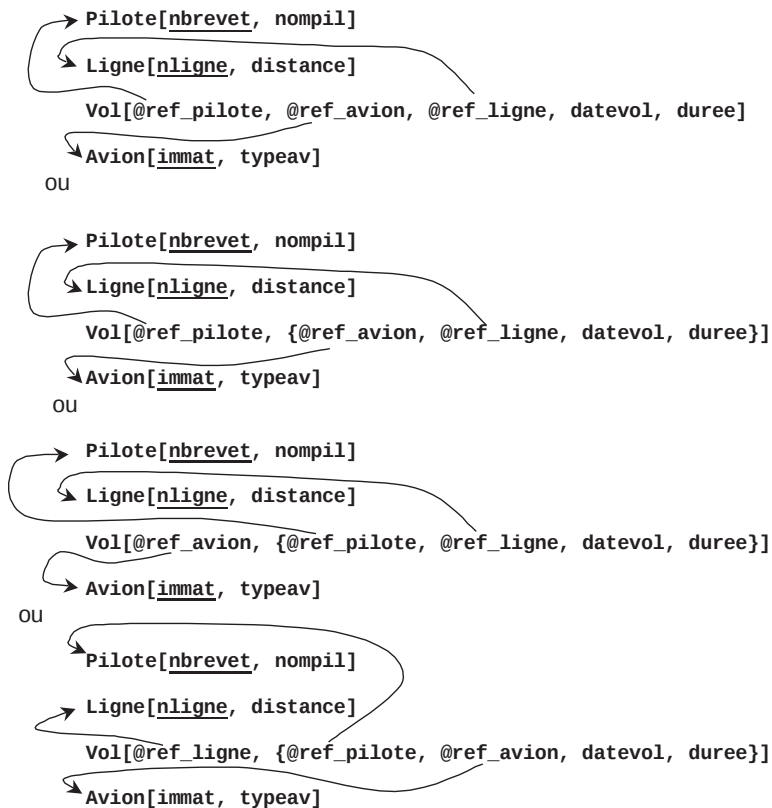
Les solutions sur la base de trois classes sont décrites à la figure 2-44. Une collection est définie dans une des trois classes (Avion, Pilote ou Ligne). Le choix se fera en fonction de l'accès aux données.

Figure 2-44 Transformation d'une association 4-aire avec trois classes



Les solutions mettant en oeuvre quatre classes sont décrites figure 2-45. Il est possible de ne pas utiliser de collection (première solution). Les trois autres privilégient l'accès aux données par chaque classe.

Figure 2-45 Transformation d'une association 4-aire avec quatre classes



Associations un-à-un

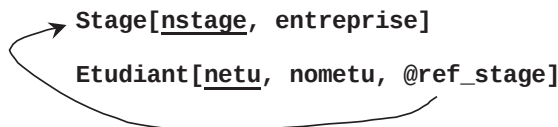


Une association *un-à-un* peut se modéliser de deux manières dans un modèle objet :

- avec deux classes sans collection et mettant en œuvre des références ;
- avec une troisième classe contenant des références mais sans collection.

L'exemple 2-46 illustre le premier cas en considérant la modélisation de la figure 1-14. En examinant les cardinalités minimales, on choisira la classe pour éviter de stocker des références nulles. Au niveau de SQL, cela se traduira par une contrainte CHECK de type NOT NULL sur la référence.

Figure 2-46 Transformation avec deux classes



La deuxième solution de transformation est illustrée à la figure 2-47. Il faudra, au niveau du code SQL, vérifier que les cardinalités maximales ne dépassent pas 1. Cela se traduira par la définition de contrainte CHECK de type UNIQUE sur les références.

Figure 2-47 Transformation avec trois classes



Transformation de l'héritage

Le mécanisme d'héritage de classes s'apparente naturellement à la décomposition descendante, (*push-down*) que nous avons précédemment étudiée. En effet, une sous-classe héritant des attributs et des méthodes d'une sur-classe est considérée comme une classe à part entière.

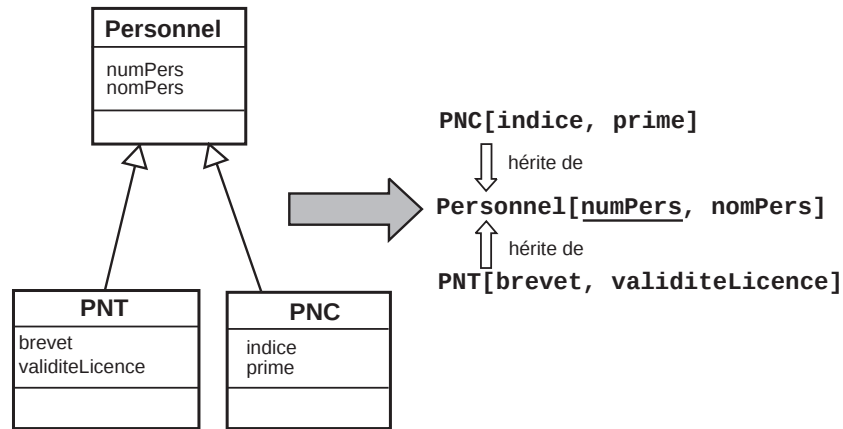
L'héritage par distinction ne peut pas s'appliquer à un schéma objet, car tous les attributs d'une sur-classe sont hérités par la ou les sous-classe(s), alors que la distinction préconise la migration de l'identifiant seul.

La décomposition ascendante peut à la rigueur s'appliquer. On ne parlera alors plus d'héritage mais de migration d'attribut et de méthodes d'une classe vers une autre. De plus les méthodes de la sous-classe pourront être exécutées illégalement sur des objets de la sur-classe.

De même que pour l'héritage simple, la traduction de l'héritage multiple s'apparente naturellement à la décomposition descendante (*push-down*).

L'exemple 2-48 illustre la traduction d'un graphe d'héritage. Le schéma logique est composé de trois classes. On peut se représenter ainsi la structure des deux sous-classes : PNC[numPers, nomPers, indice, prime] et PNT[numPers, nomPers, brevet, valideLicence].

Figure 2-48 Traduction d'une association d'héritage



Transformation des contraintes

Les éventuelles contraintes des associations d'héritage n'ont pas d'influence sur la structure des classes obtenues, mais devront être prises en compte au niveau de la programmation (contraintes SQL CHECK, déclencheurs ou méthodes).

Supposons qu'on désire programmer la contrainte de partition sur l'héritage précédent. Il faudra s'assurer, par la méthode de création d'un objet PNT, qu'il n'existe pas déjà un objet PNC ayant le même numéro (numPers), et réciproquement en ce qui concerne la méthode de création d'un objet PNC.

Associations d'agrégation

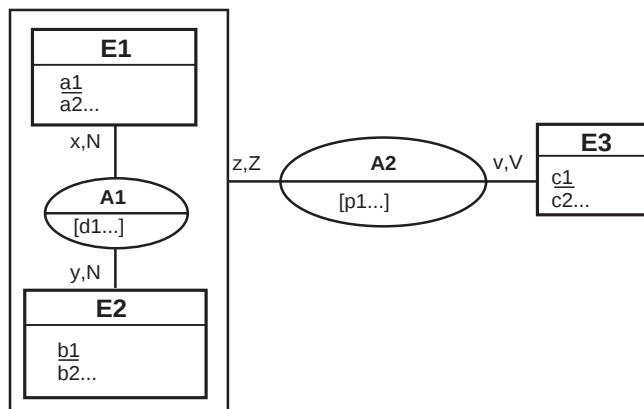
Ce n'est pas l'agrégation UML (qui a été étudiée figure 2-36) que nous détaillons ici, il s'agit d'exposer les règles de traduction des associations d'agrégation du modèle entité-association et des classes-associations UML.

Modèle entité-association

Les notations que nous employons sont les suivantes :

- l'association d'agrégation (ici A2) relie une entité (ici E3) à une liaison appelée « agrégat » (ici A1). Les identifiants des entités sont soulignés (ex : a1,b1...) ;
- les entités peuvent inclure des attributs supplémentaires (ex : a2,...,b2...) ;
- les cardinalités minimales des associations (x, y, z, v) peuvent prendre les valeurs 0 ou 1 ;
- les cardinalités maximales des liaisons entre l'association d'agrégation et l'agrégat (Z, V) peuvent prendre les valeurs 1 ou N ;
- les associations peuvent inclure des attributs (ex : d1,..., p1...).

Figure 2-49 Associations d'agrégation

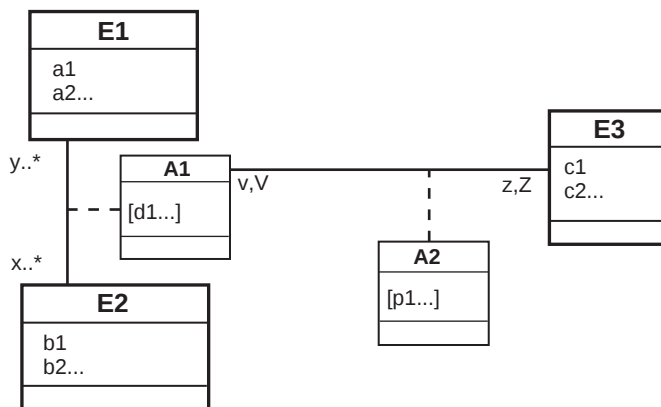


Notation UML

Pour réaliser un diagramme UML équivalent, il faut mettre en œuvre :

- l'association d'agrégation (ici A2), qui est représentée par une classe-association reliant une classe (ici E3) à une autre classe-association (ici A1) ;
- les multiplicités minimales des associations (x, y, z, v), qui peuvent prendre les valeurs 0 ou 1 ;
- les multiplicités maximales des liaisons entre l'association d'agrégation et l'agrégat (Z, V), qui peuvent prendre les valeurs 1 ou *.

Figure 2-50 Classes-associations UML



Classification

Quatre familles d'associations d'agrégation peuvent être recensées en fonction de la valeur des cardinalités (multiplicités) maximales entre l'agrégat (classe-association) et la troisième entité (classe). Ainsi, nous étudierons les associations d'agrégation *plusieurs-à-plusieurs*, *un-à-plusieurs*, *plusieurs-à-un* et *un-a-un*.



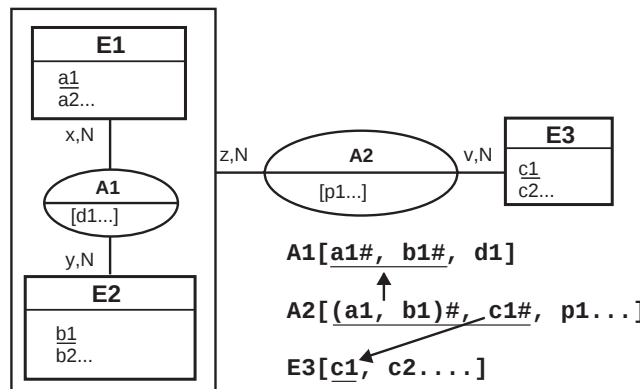
Les contraintes de cardinalités (multiplicités) minimales d'une association d'agrégation n'ont pas d'influence sur les structures de données du niveau logique.

Associations d'agrégation plusieurs-à-plusieurs

Les associations d'agrégation plusieurs-à-plusieurs ($Z = N, V = N$) sont caractérisées par le fait que les cardinalités maximales sont égales à N . L'exemple 2-51 décrit le schéma relationnel qui traduit l'association d'agrégation. Les remarques que nous pouvons faire sont les suivantes :

- L'association d'agrégation *plusieurs-à-plusieurs* peut inclure ou non des attributs ($p1, \dots$).
- La représentation au niveau logique des associations fait apparaître que la clé primaire des relations décrivant les associations doit inclure les identifiants des entités concernées par l'association. Une contrainte supplémentaire concernant la relation décrivant l'association d'agrégation (ici A2) existe. Elle exprime le fait qu'une partie de la clé primaire doit être en dépendance d'inclusion avec la clé primaire de l'association comprise dans l'agrégat (ici A1). Cette dépendance se traduira au niveau physique par la définition d'une clé étrangère.
- La valeur de la cardinalité minimale du côté de l'agrégat (notée z) sera déterminée par les valeurs de l'attribut $c1$ dans la relation A2 (s'il peut être NULL alors $z = 0$, sinon $z = 1$).
- La valeur de la cardinalité minimale du côté de l'entité (notée v) sera déterminée à la fois par les valeurs de l'attribut $c1$ dans la relation A2 et par les valeurs de l'attribut $c1$ dans la relation E3 (s'il existe un enregistrement de la relation E3 qui ne soit pas référencé dans la relation A2, alors $v = 0$, sinon $v = 1$).

Figure 2-51 Associations d'agrégation plusieurs-à-plusieurs



L'exemple 2-52 décrit l'association d'agrégation `PassagerHabituel`, qui exprime le fait qu'un agent puisse faire partie de plusieurs missions et qu'une mission puisse transporter plusieurs passagers. La notation UML équivalente est donnée figure 2-53.

La différence entre cette association d'agrégation et une association ternaire reliant `Vehicule`, `Jour` et `Agent` s'explique par le fait qu'un triplet (`Vehicule`, `Jour`, `Agent`), prenant ses valeurs dans les trois entités composant l'association 3-aire, n'est pas forcément valide. En revanche, en utilisant l'association d'agrégation, on exprime le fait que, pour qu'un agent soit transporté, il faut que la mission existe en tant que telle.

Figure 2-52 Association d'agrégation plusieurs-à-plusieurs

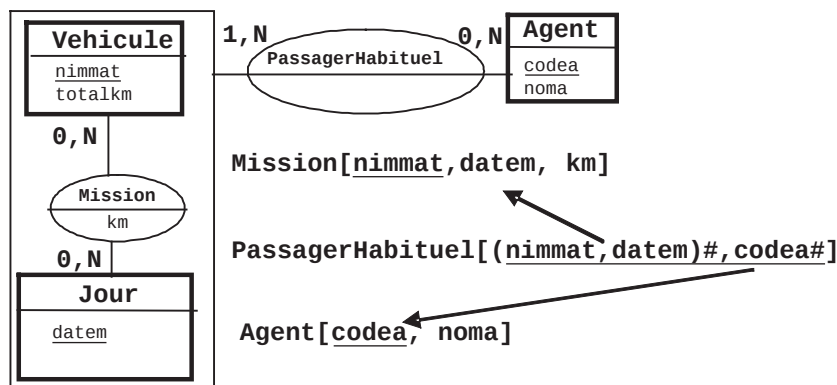
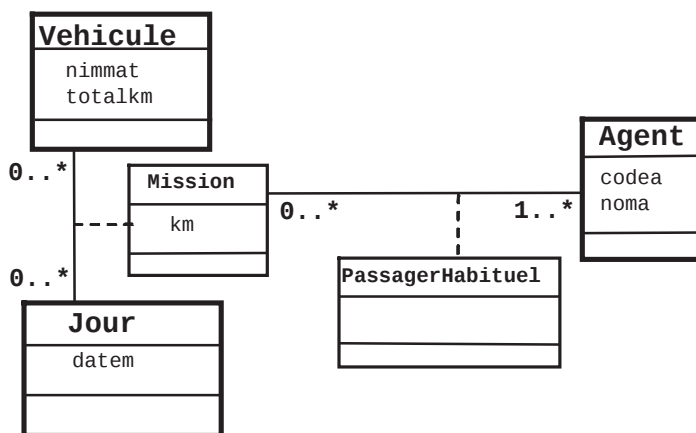


Figure 2-53 Classe-association plusieurs-à-plusieurs

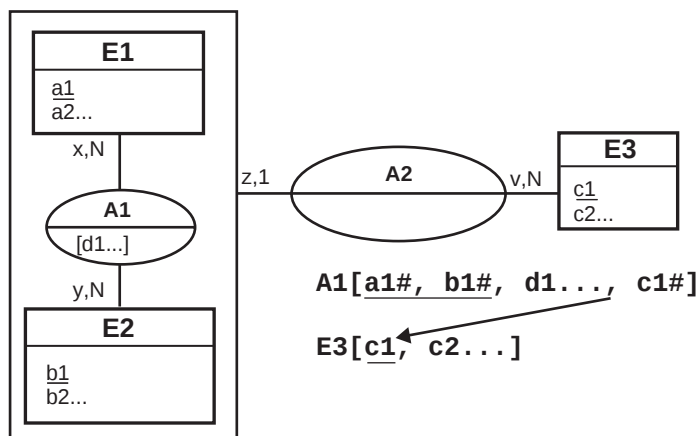


Associations d'agrégation un-à-plusieurs

Les associations d'agrégation *un-à-plusieurs* ($Z=1, V=N$) sont caractérisées par le fait que la cardinalité maximale du côté de l'agrégat est égale à 1. L'exemple 2-54 décrit le schéma relationnel qui traduit une telle association d'agrégation. Les remarques que nous pouvons faire sont les suivantes :

- L'agrégat représente le fils pour l'association d'agrégation A2, alors que l'entité E3 représente le père. En effet, un individu de E3 peut être en relation avec plusieurs individus de l'agrégat. Par contre un individu de l'agrégat ne peut être en relation qu'avec au plus un individu de E3.
- L'association d'agrégation *un-à-plusieurs* ne possède pas d'attributs.
- La structure des tables qu'on pourrait trouver dans un schéma relationnel (relations A1 et E3) est cohérente avec la démarche de conception habituelle qui indique que l'attribut clé de la relation père doit se retrouver dans la relation fils. Au niveau physique on retrouvera l'existence d'une clé étrangère.
- La valeur de la cardinalité minimale du côté de l'agrégat (notée z) sera déterminée par les valeurs de l'attribut $c1$ dans la relation A1 (s'il peut être NULL alors $z = 0$, sinon $z = 1$).
- La valeur de la cardinalité minimale du côté de l'entité (notée v) sera déterminée à la fois par les valeurs de l'attribut $c1$ dans la relation A1 et par les valeurs de l'attribut $c1$ dans la relation E3 (s'il existe un enregistrement de la relation E3 qui ne soit pas référencé dans la relation A1, alors $v = 0$ sinon $v = 1$).

Figure 2-54 Associations d'agrégation un-à-plusieurs



L'exemple 2-55 décrit l'association d'agrégation Conducteur, qui exprime le fait que pour chaque mission, il y a un agent conducteur, et qu'un agent peut être conducteur pour différentes missions. La notation UML équivalente est donnée figure 2-56.

La différence entre cette association d'agrégation et une association 3-aire (ternaire) s'explique par le fait qu'un triplet (Vehicule, Jour, Agent) prenant ses valeurs dans les trois entités composant l'association ternaire n'est pas forcément valide. En revanche, en utilisant l'association d'agrégation, on exprime le fait que pour qu'un agent soit conducteur, il faut que la mission existe en tant que telle.

Figure 2-55 Association d'agrégation un-à-plusieurs

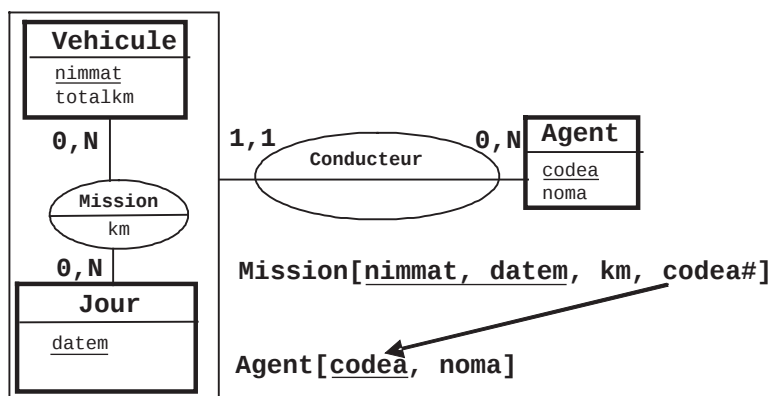
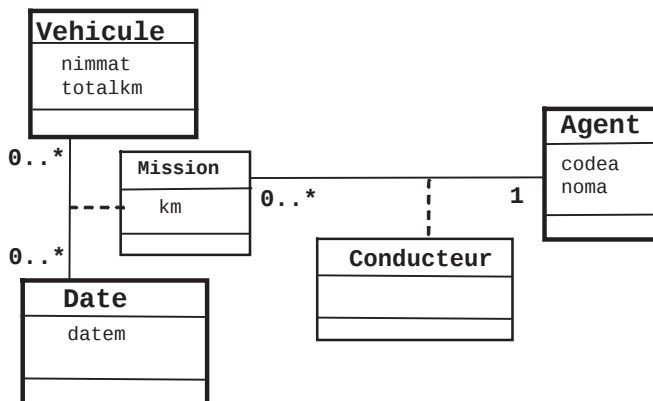


Figure 2-56 Classe-association un-à-plusieurs

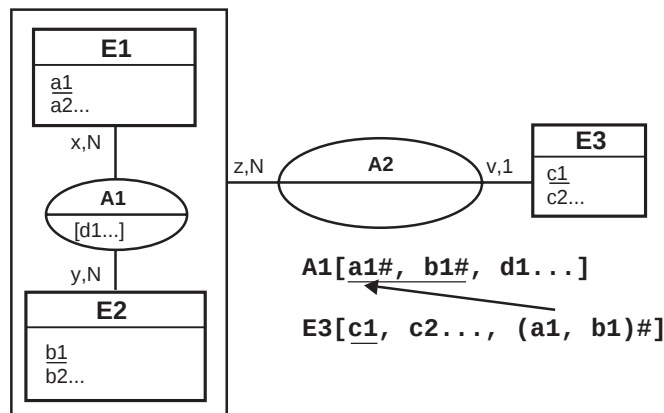


Associations d'agrégation plusieurs-à-un

Les associations d'agrégation *plusieurs-à-un* ($Z = N, V = 1$) sont caractérisées par le fait que la cardinalité maximale du côté de l'entité non concernée par l'agrégat est égale à 1. L'exemple 2-57 décrit le schéma relationnel qui traduit une telle association d'agrégation. Les remarques que nous pouvons faire sont les suivantes :

- L'agrégat représente le père pour l'association d'agrégation A2 alors que l'entité E3 représente le fils. En effet, un individu de l'agrégat peut être en relation avec plusieurs individus de E3. Par contre un individu de E3 ne peut être en relation qu'avec au plus un individu de l'agrégat.
- L'association d'agrégation *plusieurs-à-un* ne possède pas d'attributs.
- La structure des relations A1 et E3 est cohérente avec la démarche de conception habituelle qui indique que l'attribut clé de la relation père doit se retrouver dans la relation fils. Au niveau physique on retrouvera l'existence d'une clé étrangère composée de deux attributs.
- La valeur de la cardinalité minimale du côté de l'agrégat (notée z) sera déterminée par les valeurs de l'attribut $c1$ dans la relation A1 (s'il peut être NULL alors $z = 0$, sinon $z = 1$).

Figure 2-57 Associations d'agrégation plusieurs-à-un



- La valeur de la cardinalité minimale du côté de l'entité (notée v) sera déterminée à la fois par les valeurs de l'attribut $c1$ dans la relation A1 et par les valeurs de l'attribut $c1$ dans la relation E3 (s'il existe un enregistrement de la relation E3 qui ne soit pas référencé dans la relation A1 alors $v = 0$ sinon $v = 1$).

L'exemple 2-58 décrit l'association d'agrégation *BonusMission*, qui exprime le fait qu'un agent puisse choisir une seule mission pour un calcul de prime particulier. Une telle mission peut être choisie par plusieurs agents. En utilisant l'association d'agrégation, on précise que, pour qu'un agent puisse choisir une mission bonus, cette même mission doit au préalable exister en tant que telle. La notation équivalente est donnée figure 2-59.

Figure 2-58 Association d'agrégation plusieurs-à-un

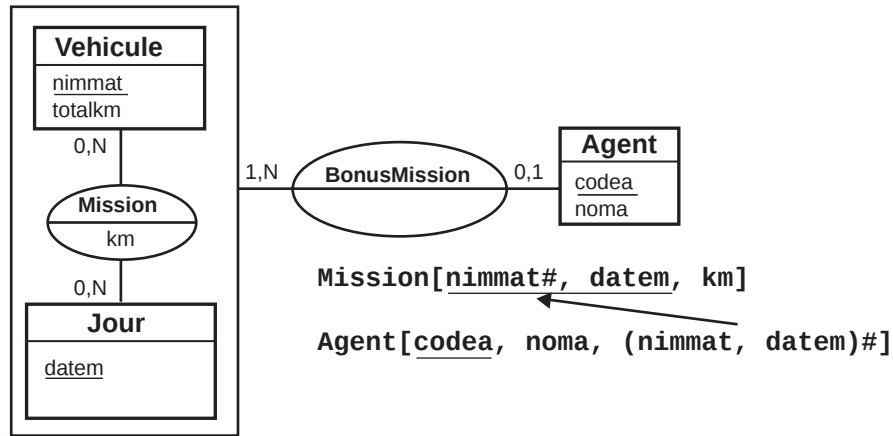
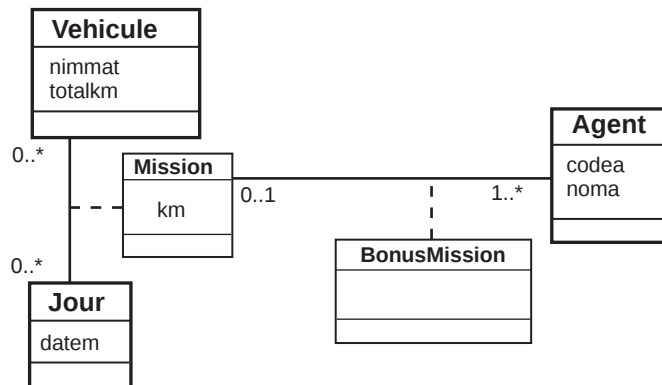


Figure 2-59 Classe-association plusieurs-à-un



Associations d'agrégation un-à-un

Les associations d'agrégation *un-à-un* ($Z = 1, V = 1$) sont caractérisées par le fait que les cardinalités maximales de l'association d'agrégation soit égales à 1. L'exemple 2-60 décrit la forme générale d'une telle association.

Le tableau 2.1 décrit les schémas relationnels possibles en fonction des cardinalités minimales (z et v). Ces schémas permettent d'éviter des valeurs de type NULL dans la base de données.

Le cas où $z = v = 1$ n'est pas envisageable (fusion des entités). Dans le cas où $z = v = 0$, il faudra choisir l'un des deux schémas.

Figure 2-60 Associations d'agrégation un-à-un

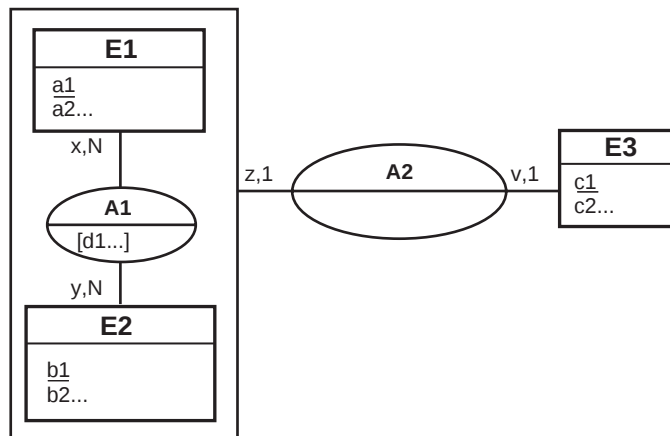


Tableau 2.1 Schémas relationnels pour une association d'agrégation un-à-un

Schéma relationnel	Valeur de v	Valeur de z
A1[a1#, b1#, d1...] E3[c1, c2..., (a1, b1)#]	(A) 1	0
A1[a1#, b1#, d1..., c1#] E3[c1, c2...]	(B) 0	1

L'exemple 2-61 illustre le cas A. Cette association d'agrégation exprime le fait qu'une mission peut donner lieu ou non à l'établissement d'un rapport, et qu'un rapport ne peut concerner qu'une seule mission. La notation UML est donnée figure 2-62.

Figure 2-61 Association d'agrégation un-à-un

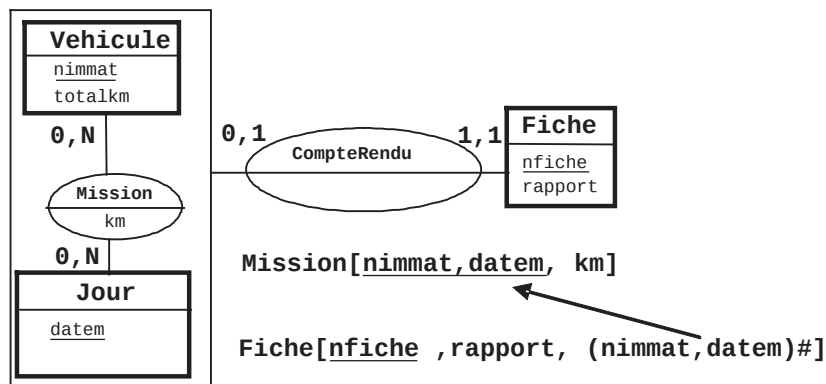
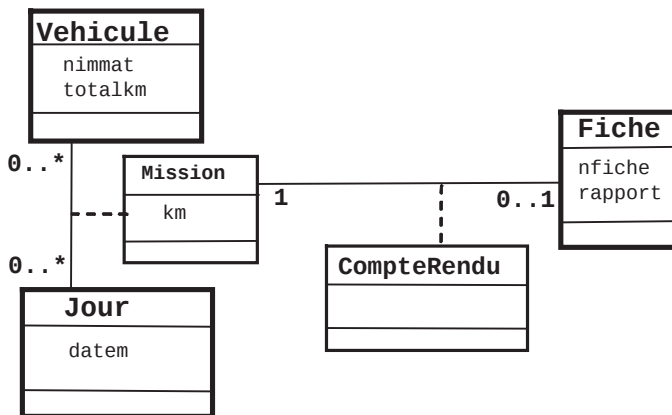


Figure 2-62 Classe-association un-à-un



Raisonnement par rétroconception

Nous avons vu dans le chapitre précédent quelles étaient les structures relationnelles possibles pour chaque type d'association d'agrégation. Dans un contexte de rétroconception, il faut adopter le processus inverse, à savoir déduire les entités et les associations d'agrégation potentielles à partir de l'étude d'un schéma relationnel.

Les quatre tableaux suivants décrivent les types d'associations d'agrégation (en termes de cardinalités maximales) qu'il est possible d'extraire automatiquement à partir du schéma relationnel de la base de données. Nous avons recensé 43 cas. Ces types d'associations d'agrégation sont potentiels, car seul l'examen des instances (données des tables) nous permettra de statuer sur le type d'associations d'agrégation (contraintes de cardinalités minimum et maximum).

On remarquera que, pour un schéma relation donné, il existe plusieurs types d'associations d'agrégation possibles. Par ailleurs, pour un type d'association d'agrégation donné, plusieurs schémas relationnels sont possibles.

Certaines associations incluent des références inverses. Par conséquent un même schéma peut exprimer soit deux associations d'agrégation distinctes, soit une seule association d'agrégation en fonction des données qu'il contient. Enfin, si les attributs clés primaires de la relation A1 (notés ici `a1#`, `b1#`) sont tous deux clés étrangères, le schéma conceptuel inclura des entités, sinon des pseudo-entités se trouveront dans l'agrégat.

Deux relations en liaison

Le tableau 2.2 décrit les cardinalités maximum potentielles des associations d'agrégation à partir de deux relations en liaison. Nous avons numéroté les schémas relationnels en lettres majuscules italiques (*A*, *B*, *C*). Le cas *C* traite des références inverses, par conséquent ce schéma peut exprimer soit deux associations d'agrégation distinctes (cas 5, 6, 7, 8) soit une seule association d'agrégation (cas 9).

Tableau 2.2 Cardinalités maximales potentielles des associations d'agrégation à partir de deux relations

Schéma relationnel	Cardi. max. A1/E3	Condition sur les données	N°
A1[<u>a1#</u> , <u>b1#</u> , d1...]	N-1	(a1, b1) non unique dans E3	1
E3[c1, c2..., (a1, b1)#] (<i>A</i>)	1-1	(a1, b1) unique dans E3	2
A1[<u>a1#</u> , <u>b1#</u> , d1..., c1#]	1-N	c1 non unique dans A1	3
E3[c1, c2...] (<i>B</i>)	1-1	c1 unique dans A1	4
A1[<u>a1#</u> , <u>b1#</u> , d1..., c1#]	1-N	c1 non unique dans A1	5
E3[c1, c2..., (a1, b1)#]	N-1	(a1, b1) non unique dans E3	6
	1-1	(a1, b1) unique dans E3	7
	N-1	(a1, b1) non unique dans E3	8
	1-N	c1 non unique dans A1	9
	1-1	(a1, b1) unique dans E3	10
	1-1	c1 unique dans A1	11
	1-1	(a1, b1) unique dans E3	12
	1-1	(a1, b1) unique dans E3	13
	1-1	(a1, b1) unique dans E3	14
	1-1	(a1, b1) unique dans E3	15
	1-1	(a1, b1) unique dans E3	16
	1-1	(a1, b1) unique dans E3	17
	1-1	(a1, b1) unique dans E3	18
	1-1	(a1, b1) unique dans E3	19
	1-1	(a1, b1) unique dans E3	20
	1-1	(a1, b1) unique dans E3	21
	1-1	(a1, b1) unique dans E3	22
	1-1	(a1, b1) unique dans E3	23
	1-1	(a1, b1) unique dans E3	24
	1-1	(a1, b1) unique dans E3	25
	1-1	(a1, b1) unique dans E3	26
	1-1	(a1, b1) unique dans E3	27
	1-1	(a1, b1) unique dans E3	28
	1-1	(a1, b1) unique dans E3	29
	1-1	(a1, b1) unique dans E3	30
	1-1	(a1, b1) unique dans E3	31
	1-1	(a1, b1) unique dans E3	32
	1-1	(a1, b1) unique dans E3	33
	1-1	(a1, b1) unique dans E3	34
	1-1	(a1, b1) unique dans E3	35
	1-1	(a1, b1) unique dans E3	36
	1-1	(a1, b1) unique dans E3	37
	1-1	(a1, b1) unique dans E3	38
	1-1	(a1, b1) unique dans E3	39
	1-1	(a1, b1) unique dans E3	40
	1-1	(a1, b1) unique dans E3	41
	1-1	(a1, b1) unique dans E3	42
	1-1	(a1, b1) unique dans E3	43
	1-1	(a1, b1) unique dans E3	44
	1-1	(a1, b1) unique dans E3	45
	1-1	(a1, b1) unique dans E3	46
	1-1	(a1, b1) unique dans E3	47
	1-1	(a1, b1) unique dans E3	48
	1-1	(a1, b1) unique dans E3	49
	1-1	(a1, b1) unique dans E3	50
	1-1	(a1, b1) unique dans E3	51
	1-1	(a1, b1) unique dans E3	52
	1-1	(a1, b1) unique dans E3	53
	1-1	(a1, b1) unique dans E3	54
	1-1	(a1, b1) unique dans E3	55
	1-1	(a1, b1) unique dans E3	56
	1-1	(a1, b1) unique dans E3	57
	1-1	(a1, b1) unique dans E3	58
	1-1	(a1, b1) unique dans E3	59
	1-1	(a1, b1) unique dans E3	60
	1-1	(a1, b1) unique dans E3	61
	1-1	(a1, b1) unique dans E3	62
	1-1	(a1, b1) unique dans E3	63
	1-1	(a1, b1) unique dans E3	64
	1-1	(a1, b1) unique dans E3	65
	1-1	(a1, b1) unique dans E3	66
	1-1	(a1, b1) unique dans E3	67
	1-1	(a1, b1) unique dans E3	68
	1-1	(a1, b1) unique dans E3	69
	1-1	(a1, b1) unique dans E3	70
	1-1	(a1, b1) unique dans E3	71
	1-1	(a1, b1) unique dans E3	72
	1-1	(a1, b1) unique dans E3	73
	1-1	(a1, b1) unique dans E3	74
	1-1	(a1, b1) unique dans E3	75
	1-1	(a1, b1) unique dans E3	76
	1-1	(a1, b1) unique dans E3	77
	1-1	(a1, b1) unique dans E3	78
	1-1	(a1, b1) unique dans E3	79
	1-1	(a1, b1) unique dans E3	80
	1-1	(a1, b1) unique dans E3	81
	1-1	(a1, b1) unique dans E3	82
	1-1	(a1, b1) unique dans E3	83
	1-1	(a1, b1) unique dans E3	84
	1-1	(a1, b1) unique dans E3	85
	1-1	(a1, b1) unique dans E3	86
	1-1	(a1, b1) unique dans E3	87
	1-1	(a1, b1) unique dans E3	88
	1-1	(a1, b1) unique dans E3	89
	1-1	(a1, b1) unique dans E3	90
	1-1	(a1, b1) unique dans E3	91
	1-1	(a1, b1) unique dans E3	92
	1-1	(a1, b1) unique dans E3	93
	1-1	(a1, b1) unique dans E3	94
	1-1	(a1, b1) unique dans E3	95
	1-1	(a1, b1) unique dans E3	96
	1-1	(a1, b1) unique dans E3	97
	1-1	(a1, b1) unique dans E3	98
	1-1	(a1, b1) unique dans E3	99
	1-1	(a1, b1) unique dans E3	100

Trois relations en liaison

Deux attributs dans la clé primaire

Le tableau décrit les cardinalités maximales potentielles des associations d'agrégation à partir de trois relations en liaison avec le degré des clés primaires inférieur à trois (nombre d'attributs composant la clé primaire). Les cas *F* et *G* traitent des références inverses, par conséquent ces schémas peuvent exprimer soit deux associations d'agrégation distinctes, soit une seule association d'agrégation (cas 18 pour le cas *F* et 23 pour le cas *G*).

Tableau 2.3 Cardinalités maximales potentielles des associations d'agrégation à partir de trois relations

Schéma relationnel	Cardi. max. A1/E3	Condition sur les données	N°
E3[c1, c2..., (a1, b1)#] A2[(a1, b1)#, p1...] A1[a1#, b1#, d1...] (D)	N-1	(a1, b1) non unique dans E3	10
E3[c1, c2...] A2[(a1, b1)#, c1#, p1...] A1[a1#, b1#, d1...] (E)	1-N	c1 non unique dans A2	12
E3[c1, c2..., (a1, b1)#] A2[(a1, b1)#, c1#, p1...] A1[a1#, b1#, d1...] (F)	1-N	c1 non unique dans A2	14
	N-1	(a1, b1) non unique dans E3	15
	1-1	(a1, b1) unique dans E3	16
	N-1	(a1, b1) non unique dans E3	17
	1-N	c1 non unique dans A2	18
	1-1	(a1, b1) unique dans E3	19
	1-1	c1 unique dans A2	20
	1-1	(a1, b1) unique dans E3	21
	1-1	(a1, b1, c1) se retrouvent simultanément dans E3 et A2	22
E3[c1, c2..., (a1, b1)#] A2[(a1, b1)#, c1#, p1...] A1[a1#, b1#, d1...] (G)	1-N	c1 non unique dans A2	23
	N-1	(a1, b1) non unique dans E3	24
	1-1	(a1, b1) unique dans E3	25
	N-1	(a1, b1) non unique dans E3	26
	1-N	c1 non unique dans A2	27
	1-1	(a1, b1) unique dans E3	28
	1-1	c1 unique dans A2	29
	1-1	(a1, b1) unique dans E3	30
	1-1	(a1, b1, c1) se retrouvent simultanément dans E3 et A2	31

Trois attributs dans la clé primaire

Le tableau suivant décrit les cardinalités maximales potentielles des associations d'agrégation à partir de trois relations en liaison avec le degré des clés primaires égal à trois (nombre d'attributs composant la clé primaire). Le cas *I* traite de références inverses, par conséquent ce schéma peut exprimer soit deux associations d'agrégation distinctes soit une seule association d'agrégation (cas 36).

Tableau 2.4 Cardinalités maximales potentielles des associations d'agrégation à partir de trois relations

Schéma relationnel	Cardi. max. A1/E3	Condition sur les données	N°
E3[c1, c2...]	N-N	c1 non unique dans A2 pour (a1, b1) donné et (a1, b1) non unique dans A2 pour c1 donné	24
A2[(a1, b1)#, c1#, p1...]	N-1	c1 unique dans A2 pour (a1, b1) donné et (a1, b1) non unique dans A2 pour c1 donné	25
A1[a1#, b1#, d1...]	1-N	c1 non unique dans A2 pour (a1, b1) donné et (a1, b1) unique dans A2 pour c1 donné	26
(H)	1-1	c1 unique dans A2 pour (a1, b1) donné et (a1, b1) unique dans A2 pour c1 donné	27
E3[c1, c2..., (a1, b1)#]	N-1	(a1, b1) non unique dans E3	28
	N-N	cf. 24	
A2[(a1, b1)#, c1#, p1...]	N-1	(a1, b1) non unique dans E3	29
	N-1	cf. 25	
A1[a1#, b1#, d1...]	N-1	(a1, b1) non unique dans E3	30
	1-N	cf. 26	
	1-N	(a1, b1) non unique dans E3	31
	1-1	cf. 27	
	1-1	(a1, b1) unique dans E3	32
	N-N	cf. 24	
	1-1	(a1, b1) unique dans E3	33
	N-1	cf. 25	
	1-1	(a1, b1) unique dans E3	34
	1-N	cf. 26	
	1-1	(a1, b1) unique dans E3	35
(I)	1-1	(a1, b1, c1) se retrouvent simultanément dans E3 et A2	36

Le tableau 2.5 décrit les cardinalités maximales potentielles des associations d'agrégation avec la relation qui représente l'association d'agrégation contenant deux clés étrangères composées.

Tableau 2.5 Cardinalités maximales potentielles des associations d'agrégation à partir de trois relations

Schéma relationnel	Cardi. max. A1/E3	Condition sur les données	N°
A1[a1#, b1#, d1...]	N-1	c1 unique dans A3 pour (a1, b1) donné et (a1, b1) non unique dans A3 pour c1 donné	37
A3[(a1, (b1)#, c1)#, m1...]	1-N	c1 non unique dans A3 pour (a1, b1) donné et (a1, b1) unique dans A3 pour c1 donné	38
A2[(b1, c1)#, p1...] (J)	1-1	c1 unique dans A3 pour (a1, b1) donné et (a1, b1) unique dans A3 pour c1 donné	39
A1[a1#, b1#, d1...]	N-N	c1 non unique dans A3 pour (a1, b1) donné et (a1, b1) non unique dans A3 pour c1 donné	40
A3[(a1, (b1)#, c1)#, m1...]	N-1	cf. 37	41
A2[(b1, c1)#, p1...]	1-N	cf. 38	42
(K)	1-1	cf. 39	43

Je ne parle pas des cardinalités minimales car elles n'influencent pas la structure des relations.

Du conceptuel à l'objet

Nous décrivons dans cette section le processus de passage d'un schéma conceptuel (entité-association ou UML) au modèle objet. Pour chaque association du schéma conceptuel, nous préconisons des transformations dans le modèle objet, qui faciliteront par la suite la déclaration du script SQL3 de la base de données.

Transformation des entités/classes

Classes du schéma navigationnel



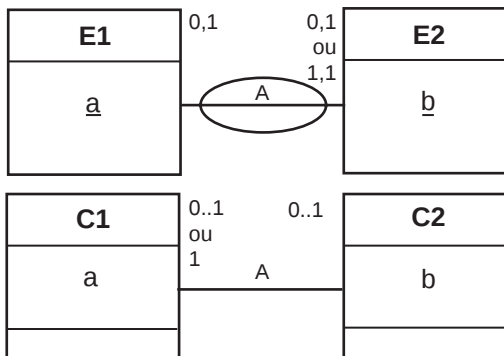
Chaque entité du schéma conceptuel entité-association devient une classe du schéma navigationnel.

Chaque classe UML, excepté les classes-associations, devient une classe du schéma navigationnel.

Transformation des associations un-à-un

On recense les quatre possibilités de transformation d'une association *un-à-un* du modèle conceptuel dans le modèle objet.

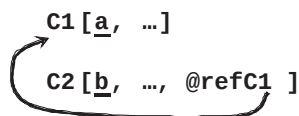
Figure 2-63 Associations un-à-un



Solutions avec une référence

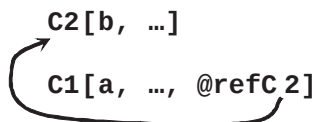
La première solution de transformation met en œuvre une référence entre les deux classes. Dans l'exemple 2-64, ce lien est `refC1`. La référence pourra être nulle (traduction de la multiplicité minimale 0).

Figure 2-64 Une référence



La deuxième solution est symétrique de la précédente.

Figure 2-65 L'autre référence

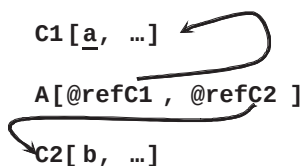


Solution avec deux références

La troisième solution met en œuvre une référence inverse.

Figure 2-66 Référence et référence inverse

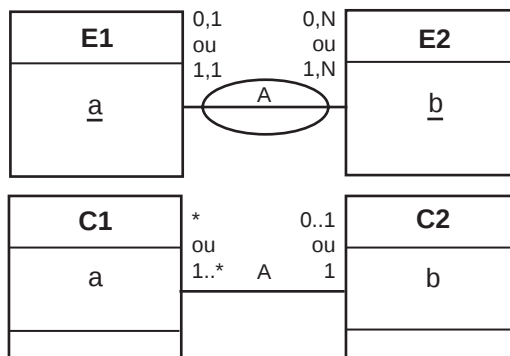
La dernière solution est basée sur l'utilisation d'une troisième classe. Cette classe porte le nom de l'association et contient deux références.

Figure 2-67 Solution universelle

Cette solution présente un avantage : elle est valable pour tous les types d'associations (on la dit ainsi universelle). La structure de la base n'a pas à être modifiée si les cardinalités sont modifiées (activation ou désactivation de contraintes SQL UNIQUE au niveau des références).

Associations un-à-plusieurs

On recense quatre possibilités de transformation d'une association *un-à-plusieurs* du modèle conceptuel dans le modèle objet.

Figure 2-68 Associations un-à-plusieurs

Collection de références

La première solution de transformation met en œuvre une collection de références. Dans l'exemple 2-69, la collection s'appelle `lesC1`, elle contient une référence `refC1`.

Figure 2-69 Collection de références

```

C1[a, ...] ←
C2[b, ..., lesC1{ @refC1 } ]

```

La deuxième solution met en œuvre la première solution avec une référence inverse (du fils vers le père). Dans notre exemple, la référence inverse est `refC2`.

Figure 2-70 Collection de références avec référence inverse

```

C1[a, ..., @refC2 ] ←
C2[b, ..., lesC1{ @refC1 } ]

```

Solutions sans collection

La troisième solution est analogue au modèle relationnel : il s'agit de recourir uniquement à une référence du fils vers le père.

Figure 2-71 Une référence

```

C1[a, ..., @refC2 ]
C2[b, ...]

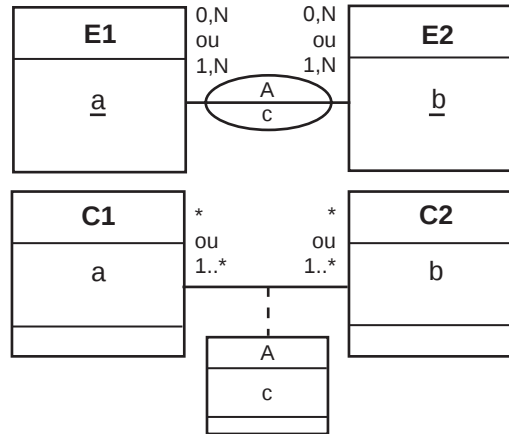
```

La quatrième alternative est la solution universelle, basée sur l'utilisation de deux références (figure 2-67).

Associations plusieurs-à-plusieurs

On recense deux possibilités de transformation d'une association *plusieurs-à-plusieurs* du modèle conceptuel dans le modèle objet.

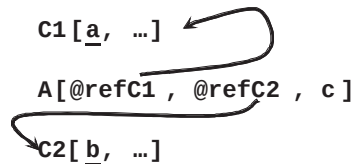
Figure 2-72 Associations plusieurs-à-plusieurs



Première solution (solution universelle)

La solution universelle met en œuvre une classe contenant les attributs de l'association et deux références.

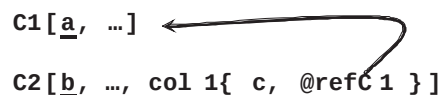
Figure 2-73 Solution universelle



Deuxième solution : une collection de références

La deuxième solution met en œuvre une collection qui contient les attributs de l'association et une référence. L'accès aux données est privilégié par la classe qui héberge la collection. Dans l'exemple 2-74, la collection **col1** contient l'attribut **c** et la référence **refC1**.

Figure 2-74 Transformations par une collection de références



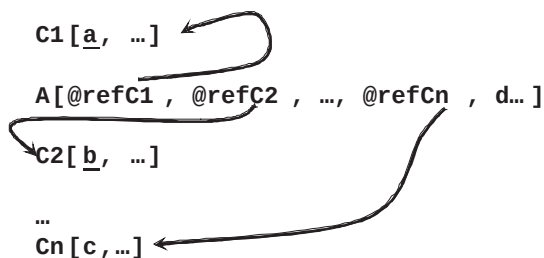
Associations n -aires

On recense $n+1$ possibilités de transformation d'une association n -aire du modèle conceptuel dans le modèle objet. Il est possible de privilégier chaque entité/classe qui compose l'association n -aire ou de n'en privilégier aucune (par la solution universelle).

Première solution (solution universelle)

La solution universelle consiste à mettre en œuvre une $n+1^e$ classe qui porte le nom de l'association, et contient les attributs de l'association et n références (figure 2-75).

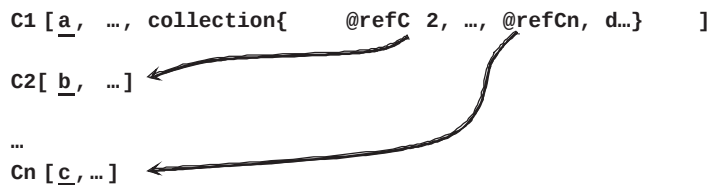
Figure 2-75 Solution universelle pour les associations n -aires



Collection de références

Chacune des autres solutions privilégie une des classes qui contiendra une collection incluant les attributs de l'association et $n-1$ références. L'exemple 2-76 illustre la traduction de l'association n -aire A qui contient l'attribut d et est reliée à n entités/classes. Ici la classe C1 est privilégiée.

Figure 2-76 Solution dans laquelle la classe C1 est privilégiée



Bilan

On peut se demander à juste titre quelle solution utiliser pour chacune des associations du schéma conceptuel qu'il faut implanter dans la base de données.

Avantages et inconvénients

Les collections privilégient l'accès aux données *via* une table au détriment d'une autre. Considérons l'exemple 2-76, l'accès aux données est privilégié par la classe C1. La requête qui extrait les attributs des classes C2 à Cn en fonction d'un objet C1 donné sera immédiate. L'utilisation de références simplifie l'expression des requêtes dans le langage d'interrogation. En ce sens, il est intéressant d'utiliser la solution universelle, même si ce n'est pas la panacée.

Le problème des références, c'est qu'elles ne fournissent pas encore totalement les fonctionnalités des clés étrangères. L'intégrité référentielle est à programmer explicitement. De plus, il n'est pas encore possible d'exprimer certaines contraintes sur une référence (comme définir un index de type clé primaire sous Oracle). L'avenir nous dira comment vont évoluer ces techniques quand même prometteuses.

Les solutions les plus relationnelles

Le tableau 2.6 indique la solution du modèle objet la plus proche d'une solution relationnelle classique. Par exemple, dans le cadre d'une association *un-à-plusieurs*, le modèle relationnel impose l'utilisation d'un attribut de type clé étrangère dans la table *fils* référençant la table *père* . La solution la plus proche au niveau navigationnel est la troisième, c'est-à-dire celle qui met en œuvre une référence.

Tableau 2.6 Correspondances objet/relationnel

Association	Modèle navigationnel
<i>un-à-un</i>	solution 2
<i>un-à-plusieurs</i>	solution 3
<i>plusieurs-à-plusieurs</i>	solution 1
<i>n-aire</i>	solution 1

Contraintes du niveau conceptuel

Ces contraintes seront réellement prises en compte au niveau de l'implémentation des méthodes. Il est difficile de déterminer à ce niveau la classe qui hébergera la méthode. Tout dépend du SGBD qui rendra certains cas de figure impossibles. Par exemple, Oracle (version 10g R2) ne permet pas de programmer un déclencheur (*trigger*) sur une collection (*nested table*). En conséquence, des méthodes seront probablement déplacées dans d'autres classes lors de l'implantation.

Considérons la contrainte de partition qui impose à tout pilote d'être affecté soit à un exercice d'entraînement, soit à une mission sanitaire. Lors de la création (ou modification) d'un objet *Pilote*, il faudra s'assurer que ledit objet est relié soit à un objet *Sanitaire*, soit à un objet *Entraînement*. Ici deux associations *un-à-plusieurs* sont à contraindre.

Collections de références

L'exemple 2-77 illustre le premier cas de transformation avec une collection de références. La collection s'appelle `contingent` dans la classe `Sanitaire` et `equipe` dans la classe `Entrainement`.

Figure 2-77 Collection de références

```
Entrainement[codent, date_ent, region, equipe{ @refPil }]
Pilote[numpil, nom, grade]
Sanitaire[codesan, organisme, contingent{ @refPil }]
```

La contrainte de partition impose de vérifier que chaque référence de la classe `Pilote` (OID) ne doit apparaître qu'une fois au plus dans l'union des collections `contingent` et `equipe`.

Le deuxième cas de transformation d'une association *un-à-plusieurs* fait intervenir une collection de références et une référence inverse. La contrainte de partition porte alors sur les trois

Figure 2-78 Deuxième solution de transformation

```
Entrainement[codent, date_ent, region, equipe{ @refPil }]
Pilote[numpil, nom, grade, @refSani, @refEntraîne]
Sanitaire[codesan, organisme, contingent{ @refPil }]
```

classes. Il faut s'assurer que :

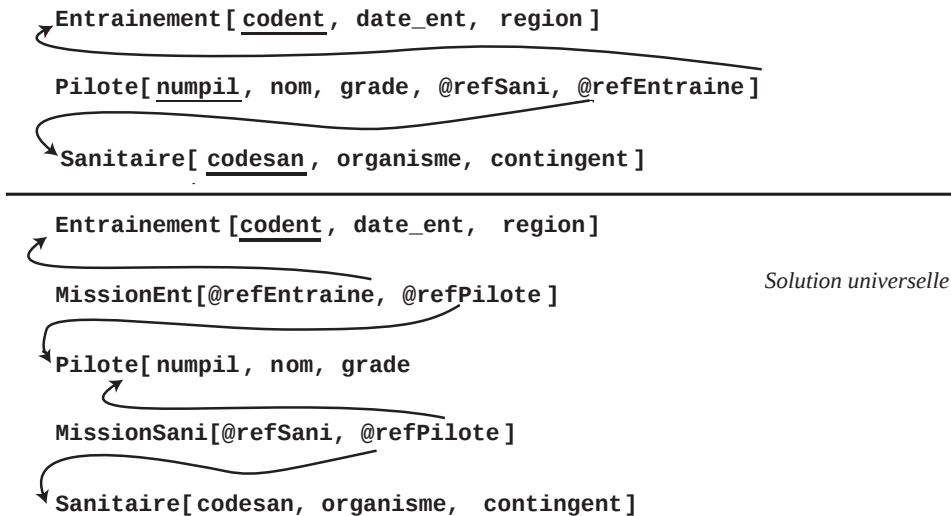
- chaque référence de la classe `Pilote` (OID) ne doit apparaître qu'une fois au plus dans l'union des collections `contingent` et `equipe` ;
- pour chaque objet `Pilote`, l'une des références `refSani` et `refEntraîne` est nulle.

Solutions sans collection

Les autres transformations mettent en œuvre des références sans collection. La figure 2-79 décrit ces deux solutions. Dans le premier cas, il faut s'assurer que, pour chaque objet `Pilote`, l'une des références `refSani` et `refEntraîne` est toujours nulle.

Le second cas correspond à la solution universelle. Les références `refPil` doivent être toutes distinctes et non nulles.

Figure 2-79 Solutions sans collection

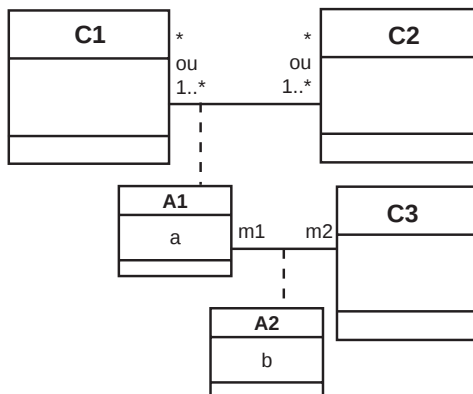


Le même type de raisonnement vaut pour toute autre contrainte (exclusivité, totalité, inclusion, unicité) qu'il faudra programmer en fonction de la nature de l'association (*un-à-un*, *un-à-plusieurs*, etc.).

Classes-associations UML

Nous étudions ici la transformation d'une classe-association reliée à une autre classe-association (classe A2 de la figure 2-80). Ce cas se présente lors de la présence de contraintes d'inclusion ou d'unicité sur une association *n*-aire. Bien que nous considérons un exemple générique ($n = 3$), le même raisonnement vaut pour un degré d'association supérieur.

Figure 2-80 Classes-associations UML



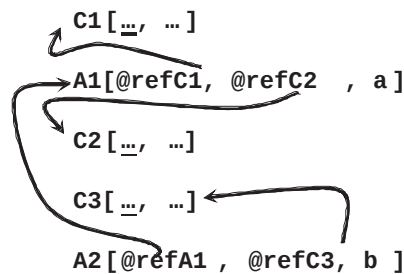
La classe-association A2 traduit une contrainte d'inclusion. Plusieurs multiplicités m1 et m2 de l'association A2 peuvent se présenter. Raisonnons sur le chiffre maximal des multiplicités (1 pour les multiplicités 1 et 0..1, * pour les multiplicités * et 1..*). La multiplicité maximale 1 permet de traduire une contrainte d'unicité.

En fonction des multiplicités, plusieurs schémas objet seront possibles. Nous en retenons deux : la solution universelle et une solution basée sur les collections. Nous modéliserons donc une association *plusieurs-à-plusieurs* (A1) et une association binaire (entre A1 et C3) en fonction des multiplicités m1 et m2. Il faudra programmer la contrainte en fonction des valeurs m1 et m2 sur les références du schéma navigationnel concernées par l'association A2.

Solution universelle

La solution universelle est illustrée à la figure 2-81, la contrainte va porter sur les références refA1 et refC3.

Figure 2-81 Solution universelle pour les classes-associations



La contrainte d'inclusion induite par l'association A2 ne requiert pas de contrainte explicite sur les liens, car la classe A2 est directement reliée à la classe A1 par la référence refA1. Les conditions induites par la contrainte d'unicité sont décrites dans le tableau 2.7.

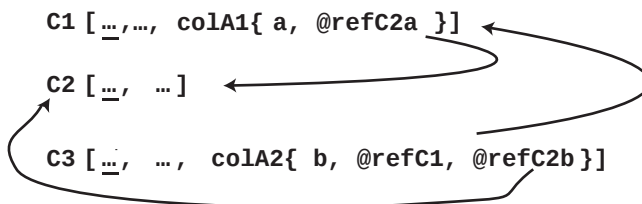
Tableau 2.7 Contraintes à respecter avec la solution universelle

m1	m2	Contrainte
1	1	Pour tous les objets de la classe A2, les références refA1 et refC3 sont toutes distinctes.
1	*	Pour tous les objets de la classe A2, le lien refC3 est distinct. Plusieurs objets de la classe A2 peuvent avoir le même lien refA1.
*	1	Pour tous les objets de la classe A2, le lien refA1 est distinct. Plusieurs objets de la classe A2 peuvent avoir le même lien refC3.
*	*	Aucune : plusieurs objets de la classe A2 peuvent avoir le même lien refC3 ou refA1.

Solution par collections

L'exemple 2-82 représente l'association *plusieurs-à-plusieurs* (A1) et une association binaire entre A1 et C3. L'association A1 se compose d'une collection de deux références, et l'association A2 d'une structure et de trois références. La contrainte va porter sur toutes les références.

Figure 2-82 Collections pour les classes-associations



La contrainte d'inclusion induite par l'association A2 s'énonce ainsi : pour tout objet A2, la référence `refC1` pointe un objet C1 qui possède une référence `refC2a` sur un objet C2 d'OID `x`, et la référence `refC2b` doit pointer l'objet d'OID `x`. Cette contrainte est notée `ci`. Les conditions sont décrites dans le tableau 2.8.

Tableau 2.8 Contraintes à respecter pour la solution avec structures

m1	m2	Contraintes
1	1	Pour tous les objets de type A2, les couples de références (<code>refC1</code> , <code>refC2b</code>) sont distincts deux à deux et on n'utilise pas la collection <code>colA2</code> . La contrainte <code>ci</code> doit aussi être respectée.
1	*	On n'utilise pas la collection <code>colA2</code> et la contrainte <code>ci</code> doit être respectée.
*	1	Pour tous les objets A2, les couples de références (<code>refC1</code> , <code>refC2b</code>) sont distincts deux à deux et on utilise la collection <code>colA2</code> . La contrainte <code>ci</code> doit aussi être respectée.
*	*	On utilise la collection <code>colA2</code> et la contrainte <code>ci</code> doit être respectée.

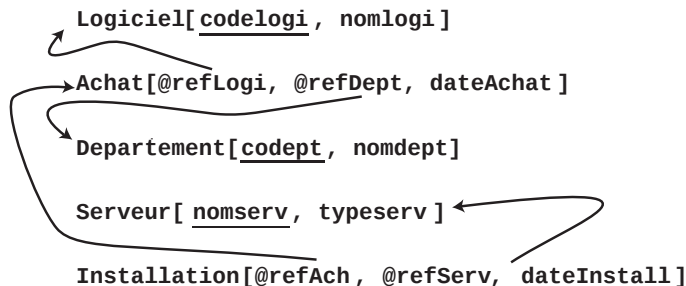
Exemple

Considérons les installations de logiciels avec la contrainte d'unicité (un logiciel d'un département ne doit être installé que sur un seul serveur) et la contrainte d'inclusion (une installation est valide seulement si le logiciel a été acheté par le département). L'exemple 2-83 met en œuvre la solution universelle. Le tableau 2.9 décrit les contraintes avec les multiplicités `m1=*` et `m2=1`.

Tableau 2.9 Contraintes à respecter pour la solution universelle

m1	m2	Contrainte
*	1	Pour tous les objets <code>Installation</code> , la référence <code>refAch</code> est distincte. Plusieurs objets <code>Installation</code> peuvent avoir la même référence <code>refServ</code> .

Figure 2-83 Solution universelle



L'exemple 2-84 met en œuvre des collections. Le tableau 2.10 décrit les contraintes avec les mêmes multiplicités.

Figure 2-84 Solution basée sur les collections

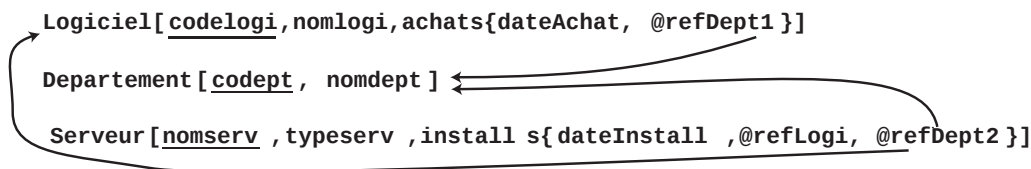


Tableau 2.10 Contraintes à respecter pour la solution avec collections

m1	m2	Contraintes
*	1	Inclusion : pour tous les éléments de la collection <code>installs</code> , la référence <code>refLogi</code> pointe un objet <code>Logiciel</code> qui possède une référence <code>refDept1</code> sur un objet <code>Département</code> d'OID <code>x</code> , et la référence <code>refDept2</code> pointe l'objet d'OID <code>x</code> . Unicité : pour tous les éléments de la collection <code>installs</code> , les couples de références (<code>refLogi</code> , <code>refDept2</code>) sont distincts deux à deux.

Agrégations UML

Une agrégation peut être modélisée de plusieurs manières :

- par des collections ou des structures dans la classe composite ;
- par une classe composite reliée aux différentes classes composantes par des références ;
- une combinaison de ces deux approches.

La première solution convient si aucune autre classe que la classe composite n'est reliée à des objets des structures de la classe composite. La seconde solution semble mieux adaptée au partage d'objets de classes composantes entre différentes classes composites. La dernière solution permet de mélanger à la demande les deux premières approches.

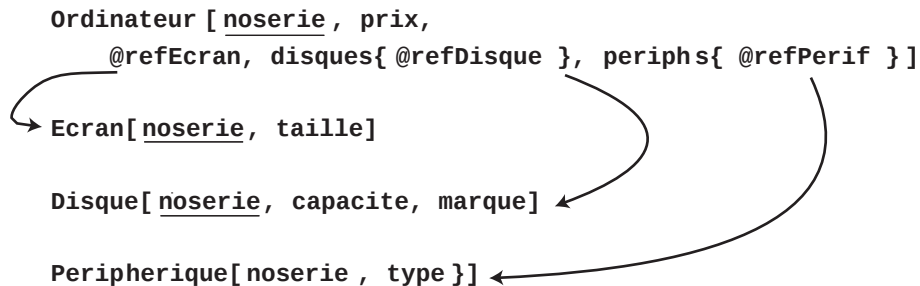
L'exemple 2-85 met en œuvre une composition avec les composants `ecran`, `disques` et `peripheriques`, parties intégrantes de la classe `Ordinateur`. Les valeurs composantes n'ont pas d'existence propre, elles doivent appartenir à un objet de la classe `Ordinateur`.

Figure 2-85 Composition

```
Ordinateur[ noserie, prix, ecran(noserie, taille),
           disques{ noserie, capacite, marque},
           peripheriques{noserie, type}]
```

Supposons qu'un objet composant puisse appartenir à différentes compositions. La figure 2-86 illustre le schéma objet, qui utilise deux collections de références. Des objets des classes `Disque`, `Ecran` et `Peripherique` peuvent avoir une existence distincte (indépendamment d'un objet de la classe `Ordinateur`). La contrainte de composition stricte est abandonnée.

Figure 2-86 Agrégation partagée



Il sera possible de combiner ces deux approches (composition stricte et agrégation partagée) en ne partageant que certains objets.

Exercices

Exercice 2.1 Dépendances fonctionnelles

On suppose les hypothèses suivantes :

- Les professeurs portent tous un nom différent.
- Des élèves peuvent avoir le même nom.
- Il y a un seul téléphone par bureau.
- Un bureau héberge plusieurs enseignants.
- Un enseignant n'est affecté qu'à un seul bureau.
- Un contrôle peut avoir lieu dans différentes salles en même temps.
- On ne stockera que le premier prénom.
- Un étudiant peut s'inscrire à 3 UV au maximum.
- Il y a plusieurs contrôles par UV.
- Un enseignant écrit un rapport pour chaque contrôle qu'il surveille, même s'il passe dans plusieurs salles.
- Plusieurs enseignants peuvent surveiller un contrôle.
- On désire savoir combien de temps l'enseignant est resté dans chaque salle.

Les attributs sont les suivants : (*note* d'un étudiant à un contrôle, *noteUV* : moyenne de l'UV pour l'étudiant, *moyUV* : moyenne de l'UV).

Classification des DF

Classifier chaque dépendance suivante (fonctionnelle, élémentaire). Parmi les dépendances fonctionnelles, préciser aussi celles qui sont directes.

1. `codeEtu` → `nom`
2. `nom` → `nInsee`
3. `nInsee` → `adresse`, `prenom`
4. `nInsee` → `codeEtu`
5. `codeEtu` → `nInsee`
6. `telEns` → `nomEns`
7. `nomEns` → `telEns`
8. `codeEns` → `nomEns`
9. `codeEns` → `telEns`
10. `codeEtu` → `codeUv`
11. `codeEtu`, `codeUv` → `nom`
12. `codeEtu`, `codeUv` → `moyUv`
13. `codeEtu`, `codeUv` → `titreUv`
14. `codeEtu` → `noteUv`
15. `codeEtu`, `codeUv` → `note`

16. $\text{codeEtu}, \text{codeUv} \rightarrow \text{note_uv}$
17. $\text{nCont} \rightarrow \text{codeUv}$
18. $\text{codeUv} \rightarrow \text{nCont}$
19. $\text{nCont} \rightarrow \text{note}$
20. $\text{nCont}, \text{codeEtu} \rightarrow \text{note}$
21. $\text{codeEns} \rightarrow \text{rappSurveillance}$
22. $\text{codeSalle}, \text{nCont} \rightarrow \text{rappSurveillance}$
23. $\text{codeEns}, \text{codeSalle}, \text{nCont} \rightarrow \text{rappSurveillance}$
24. $\text{codeEns}, \text{codeSalle}, \text{nCont} \rightarrow \text{tempsPasse}$

Élaboration du schéma relationnel

Définir le schéma relationnel en troisième forme normale à partir de l'ensemble de DF. Utiliser l'approche par synthèse.

Exercice 2.2 Propriétés des dépendances fonctionnelles

Démontrer les implications suivantes :

- Soient les deux DF $x \rightarrow y$ et $y \rightarrow z$ alors $x \rightarrow y, z$
- Soient les deux DF $x \rightarrow y$ et $z \rightarrow w$ alors $x, z \rightarrow y, w$
- Soit la DF $x \rightarrow y$ alors $x, z \rightarrow y, z$

Exercice 2.3 Dépendances fonctionnelles

Déduire les DF de la situation suivante :

- Une bibliothèque dispose d'ouvrages (numlivre , titre , editeur , anneedit , nbexemplaires). Les ouvrages qui existent en plusieurs exemplaires ont le même titre mais pas le même numéro.
- Les prêts sont limités à 3 livres par étudiant (codetu , nometu , adretu) et le bibliothécaire souhaite mémoriser la dernière date du dernier prêt pour chaque livre emprunté (datederpret).
- Un titre est unique et n'est édité que par un seul éditeur.
- Si un livre est écrit par plusieurs auteurs (codeauteur , nomauteur), il faut connaître l'ordre (ordreauteur) d'apparition des auteurs sur la couverture du livre.
- Le bibliothécaire désire mémoriser les prêts (datepret) de manière à connaître les emprunts en cours.

Constuire le diagramme UML équivalent.

Comment stocker l'historique des emprunts ?

La modélisation obtenue permet-elle à un emprunteur de louer plusieurs fois, dans la même année, le même ouvrage ?

Exercice 2.4 Dépendances fonctionnelles

Déduire les DF de la situation suivante.

Soient les dépendances fonctionnelles suivantes :

- (1) $a \rightarrow b$
- (2) $b \rightarrow c$
- (3) $a \rightarrow e$
- (4) $a \rightarrow c$
- (5) $b \rightarrow f$
- (6) $b, a \rightarrow \delta$
- (7) $d \rightarrow g$
- (8) $a, b \rightarrow c$
- (9) $a, b, f \rightarrow h$
- (10) $a, i \rightarrow j$

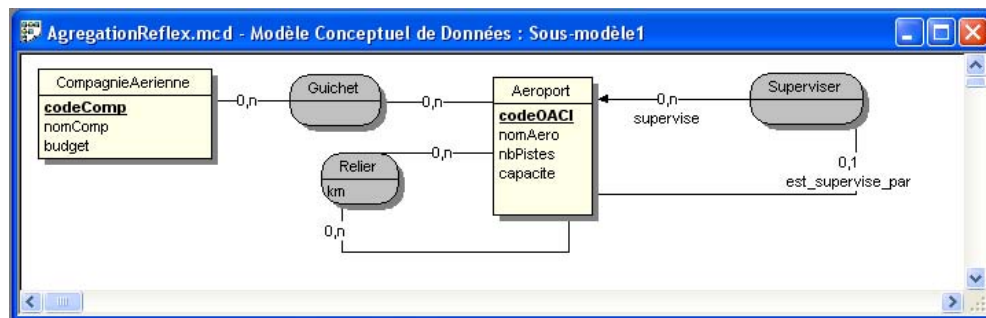
Déterminer le schéma relationnel en utilisant l'approche par synthèse.

Déterminer le MCD Merise et le diagramme UML équivalents.

Exercice 2.5 Associations réflexives

Décrire le schéma relationnel à partir de la modélisation Merise suivante.

Figure 2-87 MCD Merise

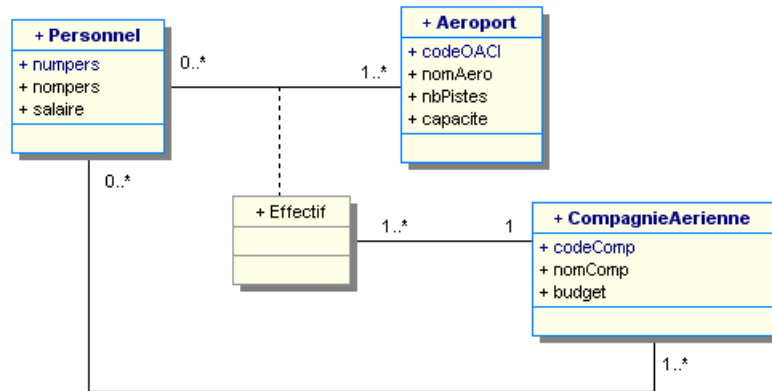


Déduire les dépendances fonctionnelles issues des trois associations.

Exercice 2.6 Classe-association

Décrire le schéma relationnel à partir de la modélisation UML suivante.

Figure 2-88 Classe association UML

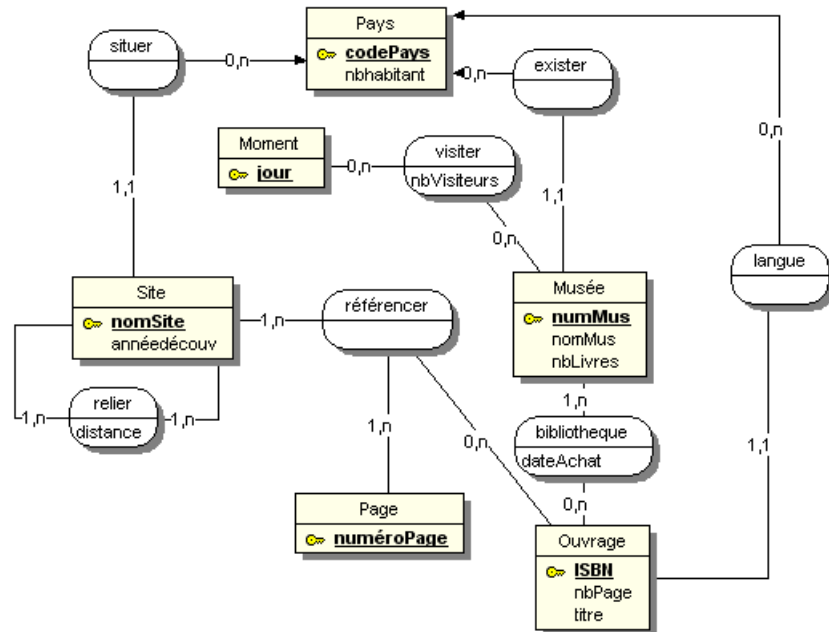


Déduire les dépendances fonctionnelles issues des deux associations.

Exercice 2.7 Association n-aire

Décrire le schéma relationnel à partir de la modélisation Merise suivante.

Figure 2-89 Association n-aire Merise



Cette modélisation permet-elle de répondre aux assertions suivantes :

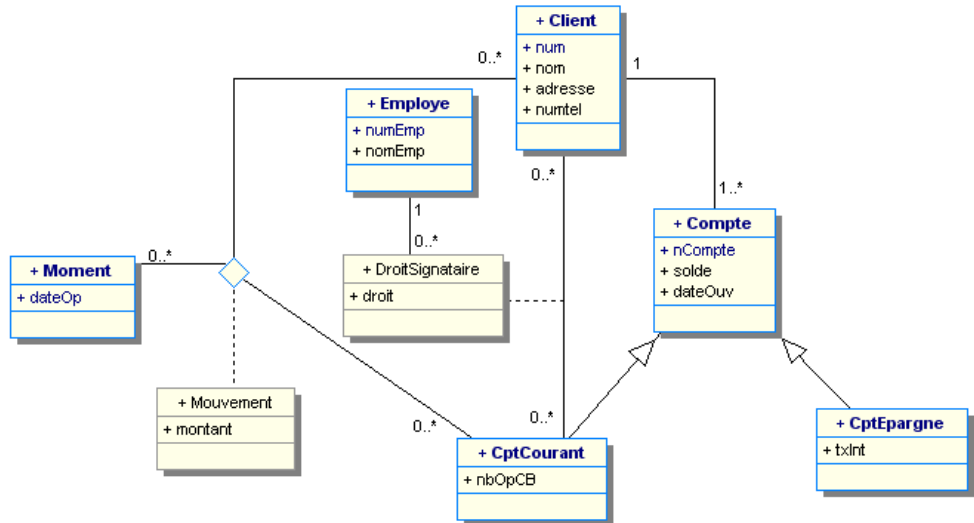
- Chaque musée est dédié à un pays.
- Le nombre de visiteurs par site dépend du jour.
- Une page d'un ouvrage ne référence qu'un site.
- Un site est référencé par plusieurs livres d'une bibliothèque.

Déduire toutes les dépendances fonctionnelles qui concernent le site.

Exercice 2.8 Héritage

Décrire un schéma relationnel à partir de la modélisation UML suivante. On suppose qu'un compte courant ne peut pas être rémunéré.

Figure 2-90 Association n-aire UML



Expliquer en quelques phrases la situation décrite.

Cette modélisation permet-elle de répondre aux assertions suivantes ?

- Un compte courant permet d'alimenter d'autres comptes courants.
- Un compte courant permet d'alimenter un compte épargne.
- Un client possède un seul compte épargne.
- Un compte courant est géré par différents clients.
- Un client peut avoir différents droits sur un compte courant.

Déduire toutes les dépendances fonctionnelles qui concernent le compte courant.

Exercice 2.9 Du conceptuel au logique

Décrire le schéma relationnel et objet à partir de certaines modélisations du chapitre 1 (exercices 1.5, 1.6 et 1.7).

1. Affrètement d'avions

Concernant le schéma objet, privilégiez l'accès aux affrètements par l'immatriculation de l'avion. Il sera intéressant de connaître rapidement les affrètements qu'un avion a effectués plutôt que la liste des affrètements de tous les avions confondus.

2. Castanet Télécoms

Concernant le schéma objet, privilégiez l'accès aux fournisseurs d'abonnement par le type de formule.

3. Voltige aérienne

Compétition journalière

Concernant le schéma objet, privilégiez l'accès aux données aux figures et aux notes d'un vol par le compétiteur.

Historique des compétitions

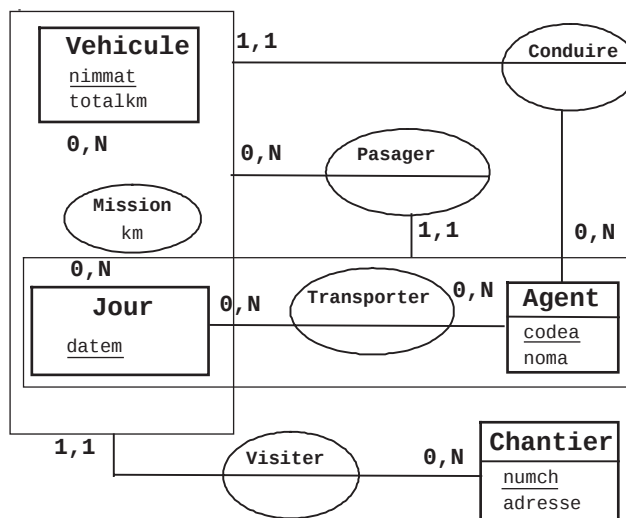
Concernant le schéma objet, privilégiez l'accès :

- aux figures et aux notes (d'un vol) par le compétiteur et le type de programme ;
- à la liste des figures (données aux juges) par le compétiteur.

Exercice 2.10 Associations d'agrégation

Dériver le schéma relationnel et un schéma objet à partir du diagramme conceptuel de la figure 2-91. Concernant le schéma objet, on privilégie l'accès aux données par la date de mission pour les personnes transportées, les chantiers et les véhicules concernés.

Figure 2-91 Exemple d'associations d'agrégation à traduire au niveau logique



Décrire le diagramme de classes UML équivalent.

Chapitre 3

Le niveau physique : de SQL2 à SQL3

*Ce que j'ai fait, balbutia-t-il, je te le jure, aucune bête
n'aurait pu le faire...*

Plus tard, Guillaumet lui expliqua.

Je t'ai vu, figure-toi, mais toi tu ne pouvais me voir.

*Mais comment as-tu su que c'était moi ? lui demanda
Saint-Ex.*

Personne d'autre n'aurait osé voler si bas.

*Antoine de Saint-Exupéry, Laboureur du ciel,
C. Cate, G.P. Putnam's Sons, 1970*

Le niveau physique que nous décrivons ici correspond à la définition des structures de données et à la programmation SQL nécessaires à mettre en œuvre. Nous décrivons dans ce chapitre les transformations à effectuer afin de dériver un schéma logique relationnel ou objet à l'aide du langage du SGBD.

Nous utiliserons une syntaxe SQL2 (Oracle et MySQL) pour les scripts s'appliquant aux bases de données relationnelles, et une syntaxe de type SQL3 (Oracle) pour les scripts s'appliquant aux bases de données objet-relationnelles.

Nous n'abordons pas ici des aspects relatifs à la structure interne de la base (types de fichiers, blocs et pages, chemins d'accès aux données, types d'index, chaînage, etc.). Nous n'abordons pas non plus les aspects liés à l'optimisation.

Le langage SQL

C'est IBM, avec System-R [AST 76], qui a mis en œuvre le premier le modèle relationnel à travers le langage SEQUEL (Structured English as QUery Language), rebaptisé par la suite SQL (Structured Query Language). SQL provient également du langage SQUARE (Specifying Queries As Relational Expressions), développé avant System-R, qui était plus structuré que SQL, mais utilisait moins de commandes.

SQL est inclus dans des logiciels commerciaux dès la fin des années 1970 avec la première version d'Oracle et de SQL/DS d'IBM. Les derniers SGBD *open source* du marché (MySQL, PostgreSQL, MaxDB, Firebird) ont adopté ce langage âgé de plus de 30 ans.

Les normes

SQL1

Le langage SQL est normalisé depuis 1986. Cette norme s'est enrichie au fil du temps. Cette première mouture, appelée SQL86 ou SQL1 (norme décrite en une centaine de pages), est le résultat de compromis entre constructeurs, bien que l'empreinte IBM soit forte. En 1989, d'importantes mises à jour sont faites en matière d'intégrité référentielle.

SQL2

La norme SQL2 (appelée aussi SQL92 [MAR 94] est présentée en 600 pages) est finalisée en 1992. Elle définit quatre niveaux de conformité : le niveau d'entrée (*entry level*), les niveaux intermédiaires (*transitional* et *intermediate levels*) et le niveau supérieur (*full level*). Les langages SQL des principaux éditeurs sont tous conformes au premier niveau, et ont beaucoup de caractéristiques relevant des niveaux supérieurs.

SQL3

Les groupes de travail X3H2 de l'ANSI et WG3 de l'ISO se penchent à partir de 1993 sur les extensions à apporter à la précédente norme. Les optimistes prévoient SQL3 pour 1996 mais rien ne se passe comme prévu. C'est en 1999 que naît SQL:1999, appelé aussi SQL3 (présenté dans un volume de 1 600 pages). Ce retard est probablement dû aux nombreux protagonistes (Oracle, IBM, Microsoft, Digital, Computer Associates...) qui rechignent à remettre en cause leur mode de pensée, au risque de ne pouvoir assurer la compatibilité des bases de leurs clients.

Les évolutions de l'ancienne norme ne sont pas limitées qu'aux extensions objet. Bien d'autres mécanismes sont introduits dans SQL:1999 (géographie, temps réel, séries temporelles, multi-média, OLAP, données et routines externes).

La dernière version de SQL est référencée SQL:2003. Ses apports concernent l'auto-incrément des clés, de nouvelles fonctions d'agrégation, de ranking et de calculs statistiques, les colonnes

calculées, et surtout une prise en charge de XML. Il est prévu de s'occuper des aspects relatifs aux bases de données déductives dans SQL4.

Les raisons du succès

Le succès que connaissent les grands éditeurs de SGBDR repose notamment sur SQL :

- SQL peut s'interfacer avec des langages de troisième génération (C, Ada ou Cobol), mais aussi avec des langages plus évolués (C++, Java, Delphi, C#).
- L'indépendance entre les programmes et les données (la modification d'une structure de données n'entraîne pas forcément une importante refonte des programmes).
- Ces systèmes sont bien adaptés aux grandes applications informatiques de gestion et ont acquis une maturité sur le plan de la fiabilité et des performances, même avec de forts volumes (actuellement plusieurs dizaines de téraoctets).
- Ils intègrent des outils de développement comme les précompilateurs, les générateurs de code, d'états, de formulaires, et des outils d'administration, de réplication, de sauvegarde, de surveillance, etc.
- Ils offrent la possibilité de stocker des informations non structurées (texte, images...) dans des champs appelés BLOB (*Binary Large Object*).

Résumons les principales caractéristiques de SQL. Le lecteur intéressé par des aspects plus précis pourra consulter [BRO 05].



Afin de rendre les scripts plus clairs, les instructions SQL seront notées en MAJUSCULES et les noms de tables, de colonnes et de contraintes en minuscules.

Définition des données

Le langage SQL permet de déclarer tous les éléments d'une base de données, en particulier les tables, qui sont les conteneurs d'informations.

Création de tables

Dans notre premier exemple, nous allons travailler sur deux tables : une qui contiendra les compagnies aériennes, et l'autre qui décrira les pilotes rattachés à leur compagnie. Les commandes suivantes créent les tables `Compagnie` et `Pilote` avec les syntaxes SQL d'Oracle et Microsoft. La clause `PRIMARY KEY` permet de déclarer une clé primaire, la clause `FOREIGN KEY`, une clé étrangère.

Avec la clause `CONSTRAINT`, vous pouvez programmer tout autre type de contrainte en la nommant (valable également pour les clés primaires ou étrangères). Nous avons limité à l'aide d'une contrainte le domaine de valeurs de l'âge des pilotes (ici la contrainte se nomme `ck_age_pilote` et assurera que l'âge de chaque pilote sera toujours compris entre 20 et 60 ans).

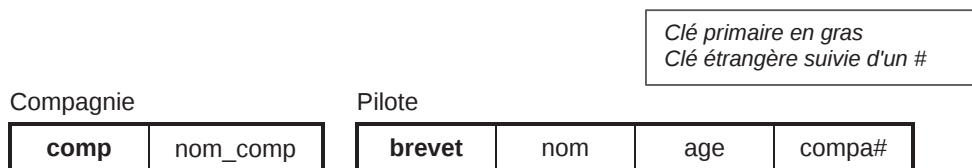
```

CREATE TABLE compagnie
(comp VARCHAR(4), nom_comp VARCHAR(30),
 CONSTRAINT pk_compagnie PRIMARY KEY(comp))
CREATE TABLE pilote
(brevet VARCHAR(8), nom VARCHAR(30),
 age INTEGER, compa VARCHAR(4),
 CONSTRAINT pk_pilote PRIMARY KEY(brevet),
 CONSTRAINT fk_pilote_compa_compagnie FOREIGN KEY(compa)
 REFERENCES compagnie(comp),
 CONSTRAINT ck_age_pilote CHECK (age BETWEEN 20 AND 60))

```

Afin d'exécuter ces scripts, sous Oracle, il faudra ajouter un point-virgule après chaque instruction et sous Microsoft la directive GO. La structure des tables est illustrée figure 3-1.

Figure 3-1 Structure des tables



Remarque

Il faut d'abord supprimer avec la directive DROP TABLE les tables *fil*s puis *père*, ensuite créer les tables *père* puis les tables *fil*s. Cela permet de pouvoir relancer le script à la demande. Le script précédent devrait donc contenir en en-tête les instructions :

Tableau 3.1 Destruction du schéma

Oracle	Microsoft
DROP TABLE pilote;	DROP TABLE pilote
DROP TABLE compagnie;	GO
	DROP TABLE compagnie
	GO

Noms des contraintes



Nommez `pk_nomtable` la contrainte clé primaire de la table.

Nommez `fk_T1_ce_T2` la contrainte clé étrangère ce de la table T1 vers la table T2 .

Manipulation des données

Les commandes INSERT, UPDATE, et DELETE permettent respectivement d'insérer, de modifier et de supprimer des enregistrements d'une table. Concernant les deux dernières fonctionnalités, on peut filtrer ces instructions selon des critères définis avec la directive WHERE.

Le script SQL suivant insère des compagnies et des pilotes rattachés chacun à une compagnie.

```
INSERT INTO compagnie VALUES('AF','Air France')
INSERT INTO compagnie VALUES('CAST','Castanet Air Lines')
INSERT INTO pilote VALUES ('3MPY93', 'Soutou', 36, 'CAST')
INSERT INTO pilote VALUES ('16AGN65', 'Bidal', 36, 'CAST')
INSERT INTO pilote VALUES ('9PAR64', 'Rival', 37, 'AF')
INSERT INTO pilote VALUES ('30MPY67', 'Lamothe', 34, 'AF')
INSERT INTO pilote VALUES ('25MPY67', 'Albaric', 34, 'CAST')
```

Le contenu des tables est illustré figure 3-2 :

Figure 3-2 Contenu des tables

Compagnie

comp	nom_comp
AF	Air France
CAST	Castanet Air Lines

Pilote

brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
9PAR64	Rival	37	AF
30MPY67	Lamothe	34	AF
25MPY67	Albaric	34	AF

Le script SQL suivant affecte le pilote de code 'Lamothe' à la compagnie de code 'CAST' :

```
UPDATE pilote SET compa = 'CAST' WHERE nom = 'Lamothe'
```

Le script SQL suivant supprime les pilotes qui ont plus de 36 ans et qui appartiennent à la compagnie de code 'AF' :

```
DELETE FROM pilote WHERE (age > 35 AND compa = 'AF')
```

Le contenu des tables est maintenant illustré par la figure 3-3 :

Figure 3-3 Contenu des tables

Compagnie

comp	nom_comp
AF	Air France
CAST	Castanet Air Lines

Pilote

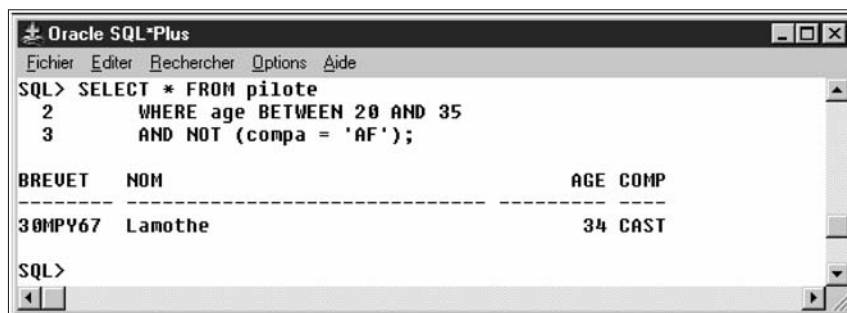
brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
30MPY67	Lamothe	34	CAST
25MPY67	Albaric	34	AF

Interrogation des données

Le langage SQL permet d'extraire des informations de la base de données en fonction de critères. Pour ce faire, il faut recourir à une instruction de type **SELECT** appelée « requête ». À titre d'exemple, nous cherchons les pilotes âgés de 20 à 35 ans, qui n'appartiennent pas à la compagnie de code 'AF'. La requête qu'il convient d'utiliser est la suivante (le signe « * » sélectionne toutes les colonnes de la table).

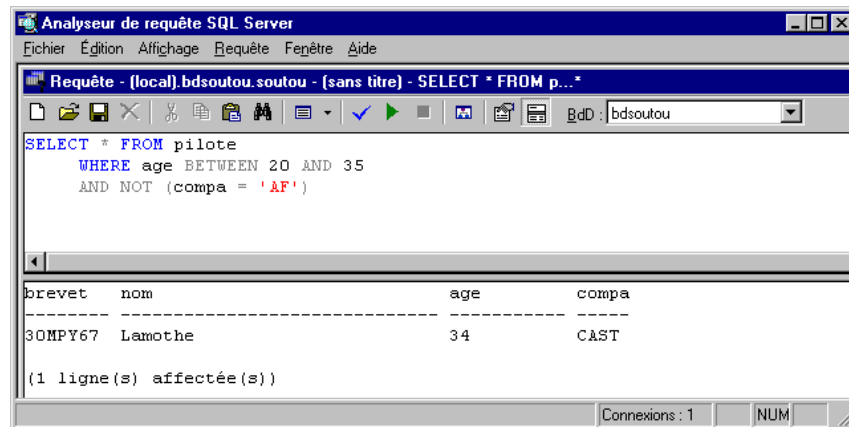
Interface SQL*Plus d'Oracle

Figure 3-4 Résultat sous Oracle



Interface ISQL/W de Microsoft

Figure 3-5 Résultat sous Microsoft



Requête avec jointures

Il est possible de rédiger des questions plus complexes mettant en jeu plusieurs tables, et faisant la plupart du temps intervenir des jointures. Supposons que le contenu des tables soit à présent celui de la figure 3-6, et que vous souhaitiez connaître le nom des pilotes âgés de 35 à 37 ans, qui appartiennent à une compagnie embauchant plus de deux pilotes, et dont le budget dépasse la moyenne des budgets des compagnies.

Figure 3-6 Contenu des tables

Compagnie

comp	nom_comp	budget
AF	Air France	300 000
CAST	Castanet Air Lines	400 001
ASO	Air Sud-Ouest	500 000

Pilote

brevet	nom	age	compa#
3MPY93	Soutou	36	CAST
2MTB98	Laroche	39	CAST
30MPY67	Lamothe	34	AF
25MPY67	Albaric	34	AF
16AGN65	Bidal	36	ASO
21PAU99	Labat	33	ASO
6MPY97	Tauzin	34	ASO

La requête qu'il conviendra d'utiliser sera alors la suivante :

```
SELECT nom, age, compa FROM pilote
WHERE compa IN (SELECT comp FROM compagnie WHERE budget >=
                (SELECT AVG(budget) FROM compagnie))
```

```

AND compa IN      (SELECT compa FROM pilote GROUP BY compa
                   HAVING COUNT(*) >= 2)
AND age BETWEEN 35 AND 38

```

Le résultat retourne :

NOM	AGE	COMP
Bidal	36	ASO
Soutou	36	CAST

On se rend compte que le concepteur de cette base n'a pas forcément pris en compte la nécessité d'effectuer cette requête lors de la conception. Cela illustre une des fonctionnalités des bases de données, à savoir le fait de ne pas connaître exhaustivement les requêtes à soumettre, et qu'on peut déduire des informations en rapprochant des faits élémentaires entre eux.

Contrôle des données

Dans un contexte multi-utilisateur, SQL a dû s'adapter pour :

- contrôler l'accès aux données (privileges en lecture, modification ou suppression) contenues dans les tables ;
- assurer la confidentialité et l'intégrité des informations. La confidentialité est souvent réalisée par l'utilisation de vues (*views*).

Attribution de privilèges

Les instructions SQL qui définissent les privilèges sur les données sont fondées sur les directives **GRANT** (pour autoriser) et **REVOKE** (pour interdire).

Exemple

Tableau 3.2 Privilèges

Utilisateur <i>Soutou</i>				Utilisateur <i>Tremont</i>	
Pilote				Compagnie	
brevet	nom	age	compa	comp	nom_comp
3MPY93	Soutou	36	CAST	AF	Air France
16AGN65	Bidal	36	CAST	CAST	Castanet Air Lines
30MPY67	Lamothe	34	CAST		
25MPY67	Albaric	34	AF		

Soutou désire autoriser l'utilisateur *Tremont* à lire et à modifier la table *Pilote* et à modifier la colonne *age*.

Tremont autorise l'utilisateur *Soutou* à lire et à insérer des enregistrements dans la table *Compagnie*.
Tremont désire interdire désormais à l'utilisateur *Soutou* d'insérer des enregistrements dans la table *Compagnie*.

Mise en œuvre sous Oracle

Tableau 3.3 Privilèges sous Oracle

Utilisateur <i>Soutou</i>	Utilisateur <i>Tremont</i>
Autorisation à l'utilisateur <i>Tremont</i> de lire la table <i>Pilote</i> . GRANT SELECT ON Pilote TO Tremont; Autorisation de privilèges (GRANT) acceptée.	
Autorisation à l'utilisateur <i>Tremont</i> de modifier la colonne <i>age</i> de la table <i>Pilote</i> . GRANT UPDATE(age) ON Pilote TO Tremont; Autorisation de privilèges (GRANT) acceptée.	
	Lecture de la table <i>Pilote</i> de <i>Soutou</i> . SELECT nom,age,compa FROM Soutou.Pilote; NOMAGECOMPACT ----- Soutou36CAST Bidal36CAST Lamothe34AF Albaric34AF
	Modification de l'âge d'un des pilotes. UPDATE Soutou.Pilote SET age = age+1 WHERE nom = 'Bidal'; 1 ligne mise à jour. Tentative de suppression de tous les pilotes. DELETE FROM Soutou.Pilote; * ERREUR à la ligne 1 : ORA-01031: privilèges insuffisants <i>Tremont</i> autorise <i>Soutou</i> à lire et à insérer des enregistrements dans la table <i>Compagnie</i> . GRANT SELECT,INSERT ON Compagnie TO Soutou; Autorisation de privilèges (GRANT) acceptée.

Tableau 3.3 Privilèges sous Oracle (suite)

Utilisateur <i>Soutou</i>	Utilisateur <i>Tremont</i>
Lecture de la table Compagnie de l'utilisateur <i>Tremont</i>. <pre>SELECT * FROM Tremont.Compagnie; COMP NOM_COMP ----- AF Air France CAST Castanet Air Lines</pre> Insertion d'une nouvelle compagnie dans la table de l'utilisateur <i>Tremont</i>. <pre>INSERT INTO Tremont.Compagnie VALUES('IFID','Aéro-IFIDEC'); 1 ligne créée.</pre> <pre>SELECT * FROM Tremont.Compagnie; COMP NOM_COMP ----- AF Air France CAST Castanet Air Lines IFID Aéro-IFIDEC</pre> Tentative de supprimer toutes les compagnies. <pre>DELETE FROM Tremont.Compagnie; *</pre> ERREUR à la ligne 1 : ORA-01031: privilèges insuffisants	<i>Tremont</i> désire interdire désormais à l'utilisateur <i>Soutou</i> d'insérer des enregistrements dans la table Compagnie. <pre>REVOKE INSERT ON Compagnie FROM Soutou;</pre> Suppression de privilèges (REVOKE) acceptée. Insertion d'une nouvelle compagnie dans la table Compagnie de l'utilisateur <i>Tremont</i>. <pre>INSERT INTO Tremont.Compagnie VALUES('Essai','Aéro Test'); *</pre> ERREUR à la ligne 1 : ORA-01031: privilèges insuffisants

Intégrité des données

Les SGBD prennent en compte l'intégrité des données définies *via* la déclaration de contraintes ou la programmation de fonctions ou de procédures cataloguées, de paquetages (*packages*) ou de déclencheurs (*triggers*). Le principe est simple : assurer la cohérence de la base après chaque modification (par INSERT, UPDATE ou DELETE).

Exemples de contraintes

Dans l'exemple, trois contraintes ont été déclarées sur la table *Pilote*, elles permettent de programmer les domaines d'attributs du modèle relationnel. La première (*pk_pilote*) indique que le numéro du brevet est unique, la deuxième (*fk_pilote_compa_compagnie*) indique que la compagnie du pilote doit être référencée dans la table *compagnie*, la troisième (*ck_age_pilote*) définit un intervalle d'âge possible pour tout pilote. Chaque contrainte est nommée. Ce principe facilite la désactivation temporaire et la réactivation des contraintes avec la commande ALTER TABLE.

```
CREATE TABLE pilote
(brevet VARCHAR(8), nom VARCHAR(30), age NUMBER, compa VARCHAR(4),
CONSTRAINT pk_pilote PRIMARY KEY(brevet),
CONSTRAINT fk_pilote_compa_compagnie FOREIGN KEY(compa)
REFERENCES compagnie(comp),
CONSTRAINT ck_age_pilote CHECK (age BETWEEN 20 AND 60))
```

Supposons que l'état des tables soit le suivant :

```
SQL> SELECT * FROM compagnie;
COMP NOM_COMP
```

```
-----
AF    Air France
CAST Castanet Air Lines
```

```
SQL> SELECT * FROM pilote;
```

BREVET	NOM	AGE	COMP
3MPY93	Soutou	36	CAST
16AGN65	Bidal	36	CAST
9PAR64	Rival	37	AF
30MPY67	Lamothe	34	AF
25MPY67	Albaric	34	AF

Essayons à présent d'insérer des enregistrements de cette table en ne respectant pas les règles de gestion déclarées, c'est-à-dire en donnant un numéro de brevet, un code compagnie et un âge incorrects.

```
SQL> INSERT INTO pilote VALUES ('3MPY93', 'Tuffery', 32, 'CAST');
ERREUR à la ligne 1 :
ORA-00001: violation de contrainte unique (SOUTOU.PK_PILOTE)

SQL> INSERT INTO pilote VALUES ('1MPY93', 'Tuffery', 32, 'RIEN');
ERREUR à la ligne 1 :
ORA-02291: violation de contrainte (SOUTOU.FK_PILOTE_COMPAGNIE) d'intégrité - touche parent introuvable

SQL> INSERT INTO pilote VALUES ('1MPY93', 'Tuffery', 62, 'CAST');
ERREUR à la ligne 1 :
ORA-02290: violation de contraintes (SOUTOU.CK_AGE_PILOTE) de vérification
```

Notons que le SGBDR renvoie chaque fois une erreur, qui évite au programmeur de prendre en compte toute une série de tests. Avec Oracle, le nom de la contrainte est préfixé par le nom du propriétaire de la table sur laquelle se porte cette contrainte (ici l'utilisateur Soutou).

Intégrité référentielle

L'intégrité référentielle, mise en œuvre à l'aide des clés étrangères, est l'une des caractéristiques majeures des SGBD. Elle permet d'assurer la cohérence d'une base de données.

Dans notre exemple, il était incorrect d'insérer un pilote d'une compagnie non référencée dans la base. Il est tout aussi incorrect de vouloir supprimer une compagnie à laquelle des pilotes sont rattachés. La suppression d'une compagnie est envisageable en effectuant en contrepartie des actions compensatoires, comme modifier tous les pilotes qui dépendent de cette compagnie (en affectant la valeur nulle à la colonne compa) ou en supprimer tous les pilotes concernés ! Ces actions peuvent être définies avec la directive CASCADE et sont déclarées avec la table concernée comme une contrainte.

Dans notre exemple, nous n'avons pas pris en compte de telles contraintes. La suppression d'une compagnie ayant des pilotes entraîne donc l'incohérence de la base. Le script suivant montre qu'Oracle gère intrinsèquement l'intégrité référentielle.

```
-- Modif de la compagnie du premier pilote
SQL> UPDATE pilote SET compa = 'AOM' WHERE brevet = '3MPY93';
ERREUR à la ligne 1 :
ORA-02291: violation de contrainte (SOUTOU.FK_PILOTE_COMPAGNIE) d'intégrité - touche parent introuvable

--Suppression d'une compagnie ayant des pilotes
SQL> DELETE FROM compagnie WHERE comp = 'AF';
ERREUR à la ligne 1 :
ORA-02292: violation de contrainte (SOUTOU.FK_PILOTE_COMPAGNIE) d'intégrité - enregistrement fils existant
```

Dans le premier message d'erreur, Oracle signale qu'un *fils* n'a pas de *père*. Dans le second message, il signale qu'un *père* possède un *fils*.

Passage du logique à SQL2

Cette section décrit la traduction SQL2 d'un schéma logique relationnel. Nous expliquons comment traduire les associations et les contraintes découvertes au niveau conceptuel.

Traduction des relations

Les instructions SQL que nous allons détailler seront à placer dans un script qui permettra la création de la base de données relationnelle.

Principe



Une relation devient une table, ses attributs sont les colonnes de la table. La clé primaire est définie sur les colonnes traduites de l'identifiant de la relation avec la contrainte PRIMARY KEY.

Exemple

L'exemple décrit un pilote caractérisé par un numéro de brevet, un nom et un âge. La figure 3-7 illustre cet exemple. La règle R1 de transit du niveau conceptuel au niveau logique est décrite au chapitre 2 (section *Du conceptuel au logique*).

Figure 3-7 Premier exemple

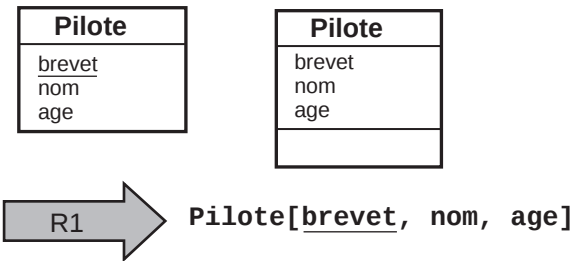


Tableau 3.4 Règle R1

Schéma logique	Script SQL2
Pilote[<u>brevet</u> , nom, age]	CREATE TABLE pilote (brevet VARCHAR(8), nom VARCHAR(30), age NUMBER, CONSTRAINT pk_pilote PRIMARY KEY(brevet))

Traduction des associations binaires

Nous étudions la traduction des associations *un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs* et réflexives. Nous illustrons nos exemples avec les formalismes Merise et UML de manière à ce que le lecteur puisse mieux appréhender les niveaux conceptuel, logique et physique.



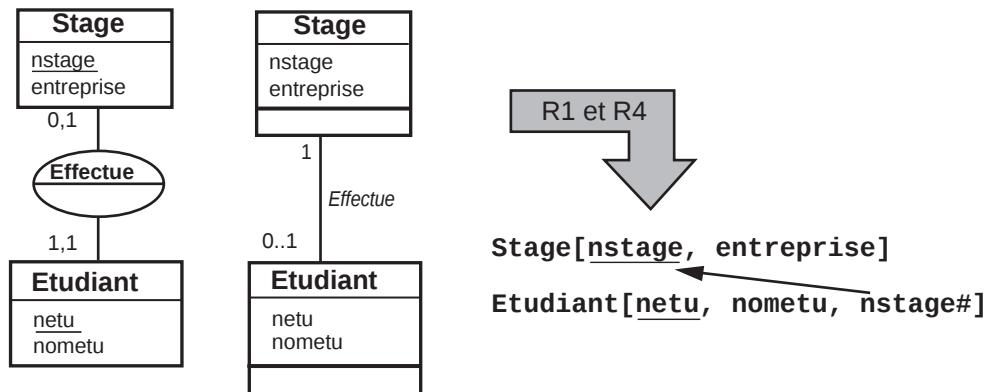
Les clés étrangères sont traduites avec des contraintes FOREIGN KEY... REFERENCES...

Associations un-à-un

L'exemple 3-8 décrit l'inscription d'un étudiant à un stage. Les règles R1 et R4 doivent être appliquées. Il est préférable de placer la clé étrangère dans la table *Etudiant* pour éviter les valeurs nulles en base.

La particularité du script SQL réside dans la définition des deux dernières contraintes qui

Figure 3-8 Exemple d'association un-à-un



traduisent les deux cardinalités minimales (à 1) de l'association *Effectue*. Ces contraintes expriment le fait qu'un étudiant doit être affecté à un seul stage (contrainte NOT NULL), et qu'un stage n'est attribué qu'à un seul étudiant (contrainte UNIQUE). La contrainte NOT NULL interdira les valeurs nulles dans la colonne *nstage*. La contrainte UNIQUE interdira qu'une valeur dans la colonne *nstage* puisse apparaître pour différents étudiants.

Tableau 3.5 Association un-à-un

Schéma logique	Script SQL2
Stage[<u>nstage</u> , entreprise]	CREATE TABLE stage (nstage VARCHAR(4), entreprise VARCHAR(30), CONSTRAINT pk_stage PRIMARY KEY(nstage))
Etudiant[<u>netu</u> , nometu, nstage#]	CREATE TABLE etudiant (netu VARCHAR(2), nometu VARCHAR(30), nstage VARCHAR(4), CONSTRAINT pk_etudiant PRIMARY KEY(netu), CONSTRAINT fk_etudiant_nstage_stage FOREIGN KEY(nstage) REFERENCES stage(nstage), CONSTRAINT nn_etudiant_nstage CHECK (nstage IS NOT NULL), CONSTRAINT unique_etudiant_nstage UNIQUE (nstage))



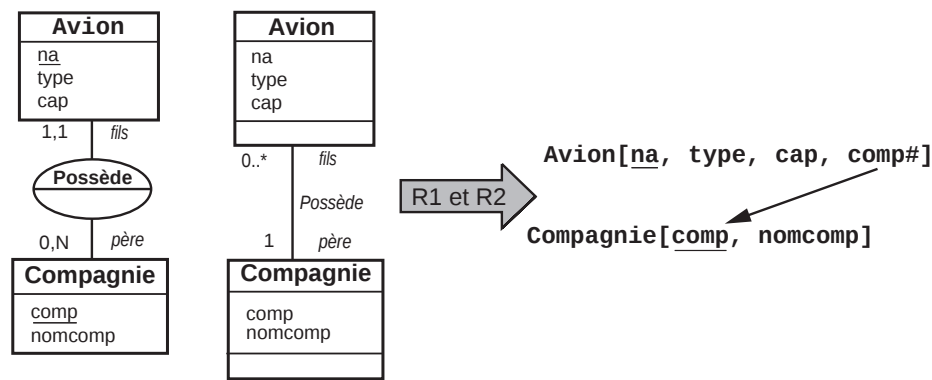
Nommez nn_T1_col une contrainte de type NOT NULL sur la colonne col de la table T1.

Nommez unique_T1_col une contrainte de type UNIQUE sur la colonne col de la table T1.

Associations un-à-plusieurs

L'exemple 3-9 décrit une association *un-à-plusieurs* entre une compagnie et ses avions. Les règles à adopter sont R1 et R2.

Figure 3-9 Exemple d'association un-à-plusieurs



La cardinalité minimale 1 de l'association Possède se traduit à l'aide d'une contrainte de type NOT NULL sur la clé étrangère.



La majorité des outils génèrent les clés étrangères après les tables par l'instruction ALTER TABLE.

Tableau 3.6 Association un-à-plusieurs

Schéma logique	Script SQL2
Compagnie[comp, nomcomp]	CREATE TABLE compagnie (comp VARCHAR(4), nomcomp VARCHAR(30), CONSTRAINT pk_compagnie PRIMARY KEY(comp))
Avion[na, type, cap, comp#]	CREATE TABLE avion (na VARCHAR(2), type VARCHAR(4), cap NUMBER(3), comp VARCHAR(4), CONSTRAINT pk_avion PRIMARY KEY(na), CONSTRAINT fk_avion_comp_compagnie FOREIGN KEY(comp) REFERENCES compagnie(comp), CONSTRAINT nn_avion_comp CHECK (comp IS NOT NULL))

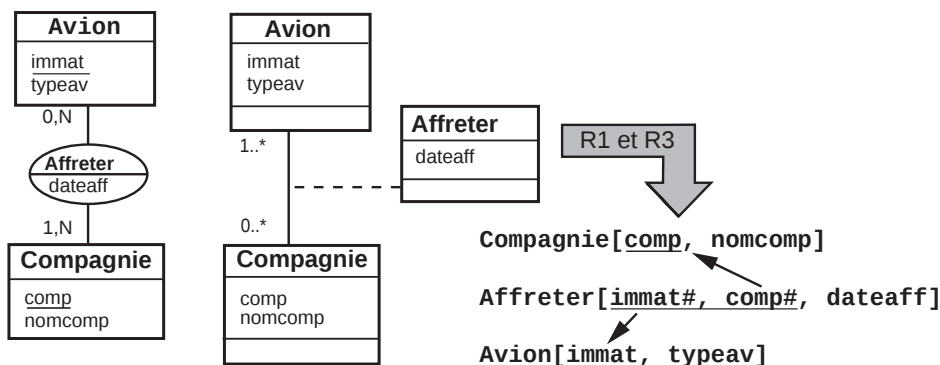
Associations plusieurs-à-plusieurs



Une relation dont l'identifiant est composé de plusieurs attributs devient une table pour laquelle la clé primaire se compose de plusieurs colonnes à l'aide de la contrainte de type PRIMARY KEY(col1, col2, ...).

Les règles R1 et R3 ont été appliquées à l'exemple 3-10 de manière à dériver trois relations, dont l'une (Affreter) possède une clé primaire composée.

Figure 3-10 Exemple d'association plusieurs-à-plusieurs





La cardinalité (multiplicité) minimale 1 d'une association *plusieurs-à-plusieurs* ne se traduit pas au niveau physique.

Dans l'exemple, cette cardinalité indique que tous les codes des compagnies référencées dans la table *Compagnie* doivent se trouver dans la table *Affreter*. Il n'est pas possible que cette contrainte soit respectée au début du cycle de vie de la base (lors du premier affrètement par exemple). En revanche, il sera possible de programmer, ultérieurement, que toute compagnie est référencée par un affrètement.

Tableau 3.7 Association plusieurs-à-plusieurs

Schéma logique	Script SQL2
	<pre>CREATE TABLE compagnie (comp VARCHAR(4), nomcomp VARCHAR(30), CONSTRAINT pk_compagnie PRIMARY KEY(comp))</pre>
Compagnie[comp, nomcomp]	
	<pre>CREATE TABLE avion (immat VARCHAR(6), typav VARCHAR(10), CONSTRAINT pk_avion PRIMARY KEY(immat))</pre>
Affreter[immat#, comp#, dateaff]	
Avion[immat, typav]	
	<pre>CREATE TABLE affreter (immat VARCHAR(6), comp VARCHAR(4), dateaff DATE, CONSTRAINT pk_affreter PRIMARY KEY (immat, comp), CONSTRAINT fk_affreter_immat_avion FOREIGN KEY(immat) REFERENCES avion(immat), CONSTRAINT fk_affreter_comp_compagnie FOREIGN KEY(comp) REFERENCES compagnie(comp))</pre>

Associations *n*-aires

Une association *n*-aire généralise une association *plusieurs-à-plusieurs*. La clé primaire de la table réalisant l'association sera donc composée de *n* colonnes. Comme pour les associations *plusieurs-à-plusieurs*, toute cardinalité minimale 1 de l'association ne se traduit pas au niveau physique.

Les règles R1 et R3 sont appliquées à l'association 3-aire de l'exemple 3-11. On ne dérive pas une relation de l'entité temporelle Jour (explication au chapitre 2). En conséquence, seuls deux attributs de la table *Affreter* sont des clés étrangères (*immat* et *comp*). La différence avec le schéma précédent réside dans le fait qu'un avion donné pour une compagnie donnée peut être affrété à différentes dates.

Figure 3-11 Exemple d'association n-aire

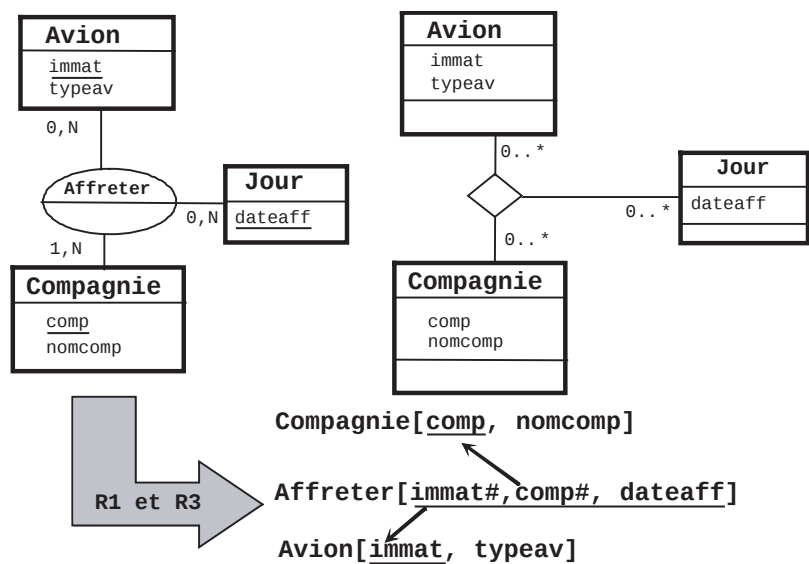


Tableau 3.8 Association n-aire

Schéma logique	Script SQL2
	<pre>CREATE TABLE compagnie (comp VARCHAR(4), nomcomp VARCHAR(30), CONSTRAINT pk_compagnie PRIMARY KEY(comp))</pre>
	<pre>CREATE TABLE avion (immat VARCHAR(6), typav VARCHAR(10), CONSTRAINT pk_avion PRIMARY KEY(immat))</pre>
Compagnie[<u>comp</u> , nomcomp]	
Affreter[<u>immat#</u> , <u>comp#</u> , dateaff]	<pre>CREATE TABLE affreter (immat VARCHAR(6), comp VARCHAR(4), dateaff DATE, CONSTRAINT pk_affreter PRIMARY KEY (immat, comp, dateaff), CONSTRAINT fk_affreter_immat_avion FOREIGN KEY(immat) REFERENCES avion(immat), CONSTRAINT fk_affreter_comp_compagnie FOREIGN KEY(comp) REFERENCES compagnie(comp))</pre>
Avion[<u>immat</u> , typav]	

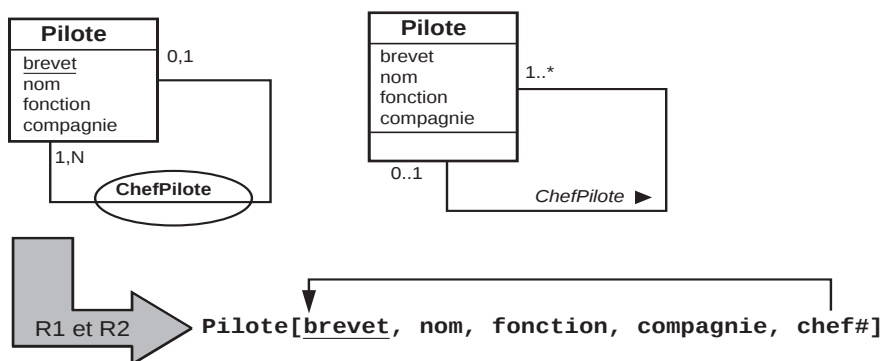
Associations réflexives

Les associations réflexives sont des associations binaires (*un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs*) ou *n*-aires. Les transformations sont analogues aux associations non réflexives.

Un-à-plusieurs

Les règles R1 et R2 sont appliquées à l'association réflexive *un-à-plusieurs* de l'exemple 3-12. La clé étrangère contiendra le code du chef pilote pour chaque pilote. Pour tout chef, cette clé étrangère ne contiendra pas de valeur (NULL).

Figure 3-12 Exemple d'association réflexive un-à-plusieurs



La partie SQL indiquée en gras souligne la particularité de l'association réflexive.

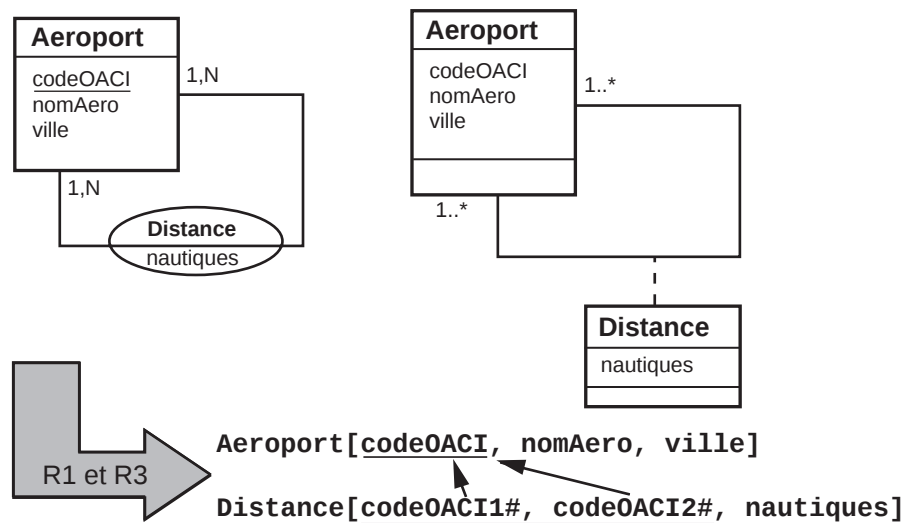
Tableau 3.9 Association réflexive un-à-plusieurs

Schéma logique	Script SQL2
Pilote[brevet, nom, fonction, compagnie, chef#]	<pre>CREATE TABLE pilote (brevet VARCHAR(8), nom VARCHAR(30), fonction VARCHAR(4), compagnie VARCHAR(4), chef VARCHAR(8), CONSTRAINT pk_pilote PRIMARY KEY(brevet), CONSTRAINT fk_pilote_chef_pilote FOREIGN KEY(chef) REFERENCES pilote(brevet))</pre>

Plusieurs-à-plusieurs

Les règles R1 et R3 sont appliquées à l'association réflexive *plusieurs-à-plusieurs* de l'exemple 3-13 (modélisation de la distance entre deux aéroports).

Figure 3-13 Exemple d'association réflexive plusieurs-à-plusieurs



Le script SQL est le suivant.

Tableau 3.10 Association réflexive plusieurs-à-plusieurs

Schéma logique	Script SQL2
Aeroport[<u>codeOACI</u>, nomAero, ville] Distance[<u>OACI1#</u>, <u>OACI2#</u>, nautiques]	<pre>CREATE TABLE Aeroport (codeOACI VARCHAR(8), nomAero VARCHAR(30), ville VARCHAR(20), CONSTRAINT pk_Aeroport PRIMARY KEY(codeOACI)) CREATE TABLE Distance (OACI1 VARCHAR(8),OACI2 VARCHAR(8), nautiques NUMBER(4), CONSTRAINT pk_Distance PRIMARY KEY(OACI1,OACI2), CONSTRAINT fk_Distance_Aeroport1 FOREIGN KEY(OACI1) REFERENCES Aeroport(codeOACI), CONSTRAINT fk_Distance_Aeroport2 FOREIGN KEY(OACI2) REFERENCES Aeroport(codeOACI))</pre>

Solution universelle

Il est possible de modéliser toute association par une table supplémentaire, qui contiendra autant de clés étrangères qu'il y a de tables à relier, et sur lesquelles existera ou non une contrainte de type UNIQUE. Ce principe s'apparente à la règle R3.



Cette solution présente l'avantage de pouvoir faire évoluer le schéma plus facilement si les cardinalités viennent à changer dans le temps. En effet, il n'y a besoin de modifier la structure d'aucune table, seules des contraintes UNIQUE devront être désactivées.

Considérons à nouveau l'exemple des stages des étudiants. La table supplémentaire (Effectuer) contient deux colonnes réalisant l'association. Sur chaque colonne, il sera impératif de définir une contrainte UNIQUE qui garantira les cardinalités maximales de l'association (ici *un-à-un*).

Tableau 3.11 Solution universelle pour une association un-à-un

Schéma logique	Script SQL2
<pre> graph TD Stage["Stage[nstage, entreprise]"] <--> Effectuer["Effectuer[netu#, nstage#]"] Effectuer --> Etudiant["Etudiant[netu, nometu]"] </pre>	<pre> CREATE TABLE Stage (nstage VARCHAR(4), entreprise VARCHAR(30), CONSTRAINT pk_stage PRIMARY KEY(nstage)) CREATE TABLE Etudiant (netu VARCHAR(2), nometu VARCHAR(30), CONSTRAINT pk_etudiant PRIMARY KEY(netu)) CREATE TABLE Effectuer (netu VARCHAR(2), nstage VARCHAR(4), CONSTRAINT pk_Effectuer PRIMARY KEY(netu,nstage), CONSTRAINT fk_Effectuer_nstage_Stage FOREIGN KEY(nstage) REFERENCES Stage(nstage), CONSTRAINT fk_Effectuer_netu_Etudiant FOREIGN KEY(netu) REFERENCES Etudiant(netu), CONSTRAINT unique_Effectuer_netu UNIQUE (netu), CONSTRAINT unique_Effectuer_nstage UNIQUE (nstage)) </pre>

Ce schéma est évolutif. En effet, si un stage peut être effectué par plusieurs étudiants, il faudra simplement désactiver la contrainte (ALTER TABLE Effectuer DISABLE CONSTRAINT unique_Effectuer_nstage). On retrouvera une association *un-à-plusieurs*. Si un étudiant peut effectuer plusieurs stages, il faudra désactiver l'autre contrainte. On se retrouvera alors avec une association *plusieurs-à-plusieurs*.

Traduction des associations d'héritage

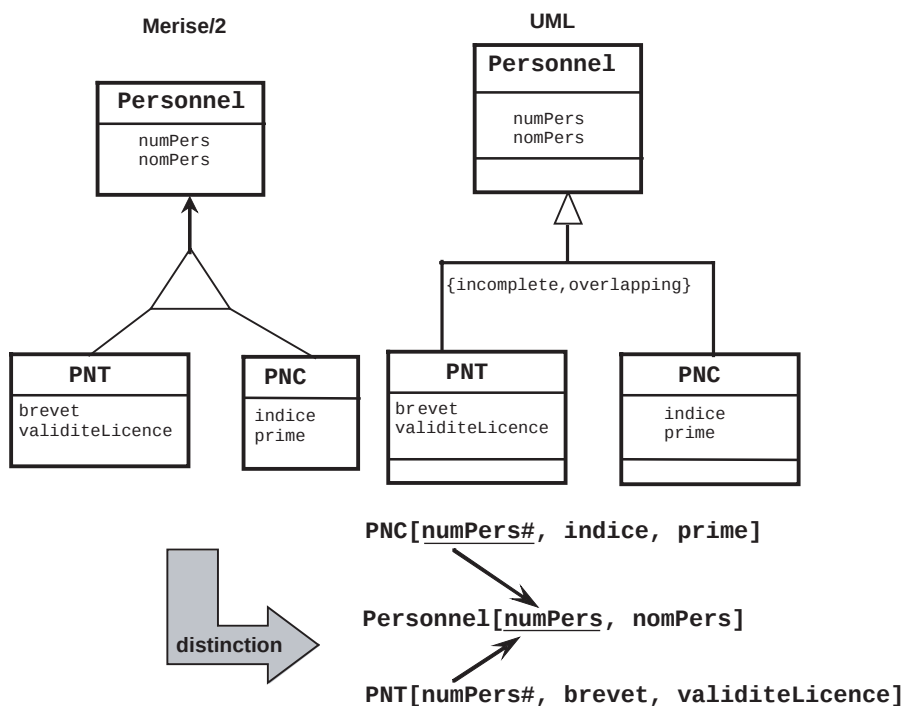
Nous expliquerons dans un premier temps la traduction SQL2 des associations d'héritage en fonction de la décomposition choisie (chapitre 2, section *Héritage*). Nous détaillerons ensuite la traduction d'éventuelles contraintes (partition, totalité et exclusivité).

Nous avons recensé au chapitre précédent trois familles de décomposition au niveau logique pour traduire une association d'héritage : décomposition par distinction, descendante (*push-down*) et ascendante (*push-up*).

Décomposition par distinction

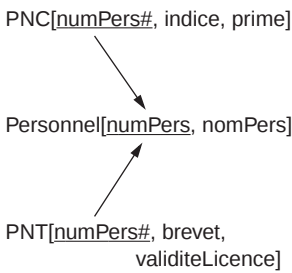
L'héritage 3-14 (sans contrainte avec Merise/2 se modélisant par une contrainte UML) est traduit suivant le principe de décomposition par distinction. La clé primaire issue de la sur-entité (sur-classe) est dupliquée dans les deux tables déduites des sous-entités (sous-classes).

Figure 3-14 Décomposition par distinction d'une association d'héritage



Le schéma physique est composé de trois tables dotées de la même clé primaire. La table **Personnel** stockera notamment les personnels n'étant ni **PNT** ni **PNC**.

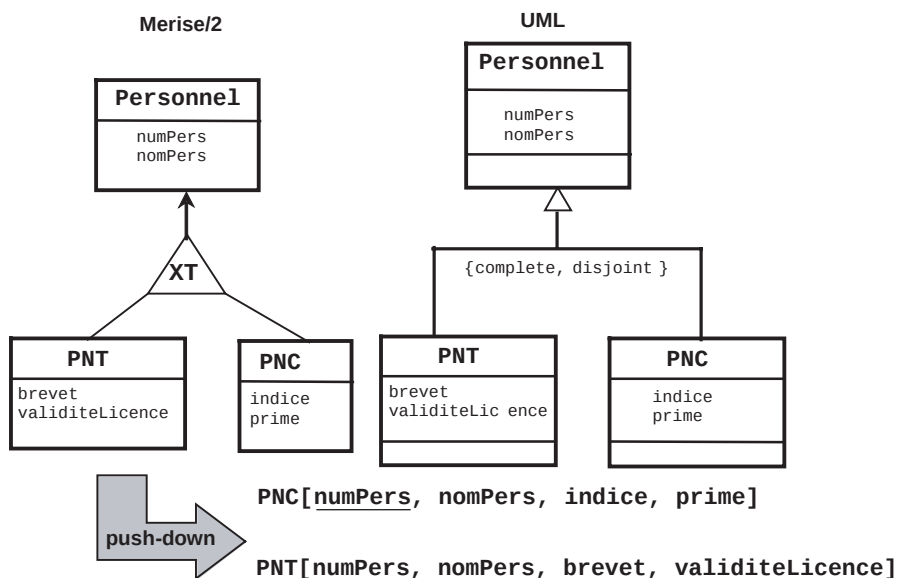
Tableau 3.12 Héritage par distinction

Schéma logique	Script SQL2
	<pre> CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), CONSTRAINT pk_Personnel PRIMARY KEY(numPers)) CREATE TABLE PNC (numPers NUMBER, indice NUMBER, prime NUMBER(8,2), CONSTRAINT pk_PNC PRIMARY KEY(numPers), CONSTRAINT fk_PNC_Personnel FOREIGN KEY(numPers) ➔REFERENCES Personnel(numPers)) CREATE TABLE PNT (numPers NUMBER, brevet VARCHAR(10), valideLicence DATE, CONSTRAINT pk_PNT PRIMARY KEY(numPers), CONSTRAINT fk_PNT_Personnel FOREIGN KEY(numPers) ➔REFERENCES Personnel(numPers)) </pre>

Décomposition descendante (push-down)

L'héritage de partition 3-15 exprime le fait qu'aucun personnel ne peut être à la fois PNT et PNC, et qu'il n'existe pas non plus de personnel n'étant ni PNT ni PNC. Nous verrons plus loin comment traduire cette contrainte. Il est question ici uniquement de traduire les relations déduites au niveau logique en tables.

Figure 3-15 Exemple de décomposition descendante (push-down) d'une association d'héritage



Le schéma physique est composé de deux tables dans lesquelles le contenu de la relation issue de la sur-entité (ici *Personnel*) a été intégralement dupliqué. La table *Personnel* n'est pas nécessaire car aucun personnel ne peut être ni PNT ni PNC.

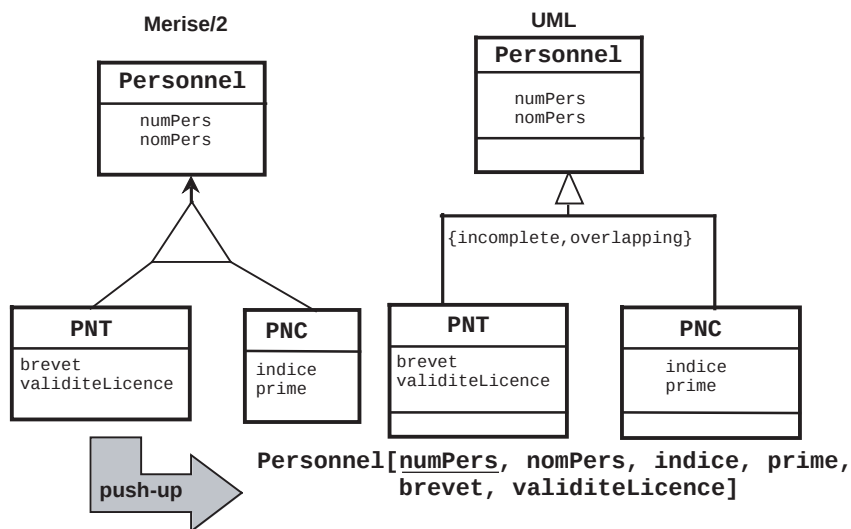
Tableau 3.13 Héritage push-down

Schéma logique	Script SQL2
PNC[numPers, nomPers, indice, prime]	CREATE TABLE PNC (numPers NUMBER, nomPers VARCHAR(20), indice NUMBER, prime NUMBER(8,2), CONSTRAINT pk_PNC PRIMARY KEY(numPers))
PNT[numPers, nomPers, brevet, valideLicence]	CREATE TABLE PNT (numPers NUMBER, nomPers VARCHAR(20), brevet VARCHAR(10), valideLicence DATE, CONSTRAINT pk_PNT PRIMARY KEY(numPers))

Décomposition ascendante (push-up)

L'exemple 3-16 illustre la décomposition ascendante du graphe d'héritage.

Figure 3-16 Décomposition ascendante (push-up) d'une association d'héritage



Le schéma physique est constitué d'une seule table dans laquelle se trouvent tous les attributs des sous-entités (sous-classes). La table *Personnel* contiendra des valeurs nulles dans certains cas (par exemple pour un personnel seulement PNT, les colonnes *indice* et *prime* vaudront NULL).

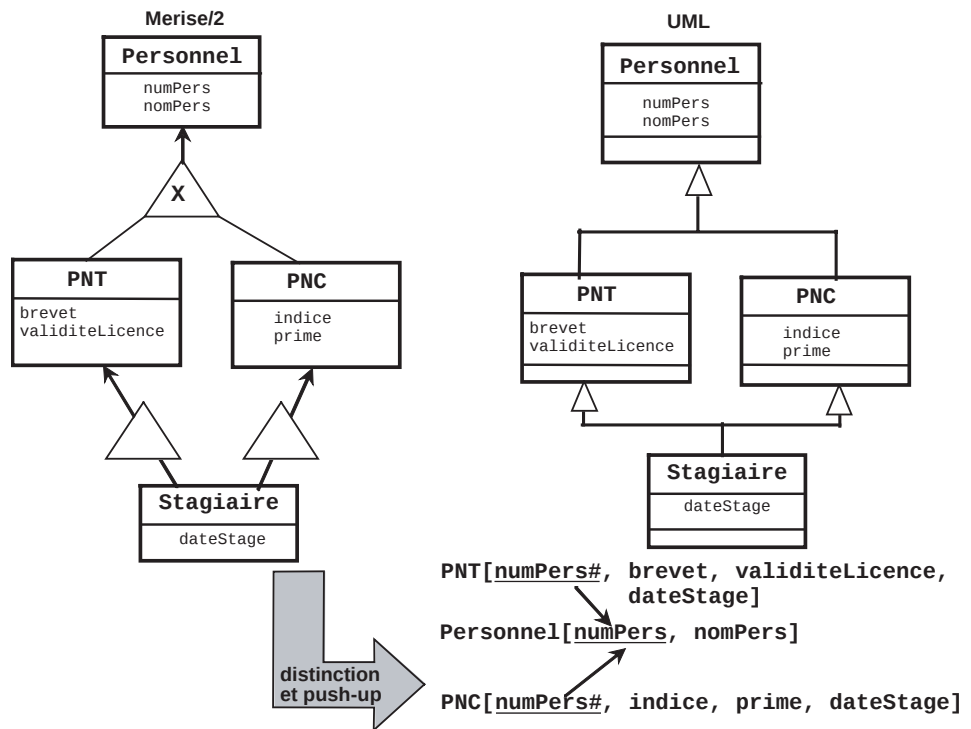
Tableau 3.14 Héritage push-up

Schéma logique	Script SQL2
Personnel[numPers, nomPers, indice, prime, brevet, validiteLicence]	CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), indice NUMBER, prime NUMBER(8,2), brevet VARCHAR(10), validiteLicence DATE, CONSTRAINT pk_Personnel PRIMARY KEY(numPers))

Traduction des associations d'héritage multiple

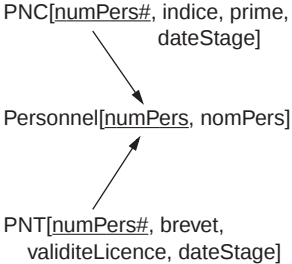
Les trois familles de décomposition précédemment étudiées peuvent s'appliquer aux associations d'héritage multiple. Chaque association du graphe d'héritage pourra ainsi être traduite soit par distinction, soit de manière descendante, soit de manière ascendante, dans la mesure du possible en fonction des contraintes existantes par ailleurs (contraintes d'héritage mais aussi d'autres contraintes comme une sous-entité connectée à une entité non rattachée au graphe d'héritage, etc.). Dans le graphe d'héritage suivant, nous choisissons d'utiliser la décomposition par distinction pour le premier niveau d'héritage, et la décomposition ascendante (*push-up*) pour le second niveau d'héritage.

Figure 3-17 Décomposition d'une association d'héritage multiple



Le schéma physique sera constitué de trois tables. Les enregistrements des tables PNT et PNC, dont la colonne `dateStage` sera non nulle, seront des stagiaires.

Tableau 3.15 Héritage multiple

Schéma logique	Script SQL2
 <pre>graph TD PNC --> Personnel PNC --> PNT</pre>	<pre>CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), CONSTRAINT pk_Personnel PRIMARY KEY(numPers)) CREATE TABLE PNC (numPers NUMBER, indice NUMBER, prime NUMBER(8,2), dateStage DATE, CONSTRAINT pk_PNC PRIMARY KEY(numPers), CONSTRAINT fk_PNC_Personnel FOREIGN KEY(numPers) REFERENCES Personnel(numPers)) CREATE TABLE PNT (numPers NUMBER, brevet VARCHAR(10), valideLicence DATE, dateStage DATE, CONSTRAINT pk_PNT PRIMARY KEY(numPers), CONSTRAINT fk_PNT_Personnel FOREIGN KEY(numPers) REFERENCES Personnel(numPers))</pre>

Traduction des contraintes d'héritage

Nous étudions à présent les moyens qu’offre le langage SQL pour traduire les contraintes possibles d’un graphe d’héritage. Cela faisant, nous pourrons discuter des avantages et inconvénients des différentes décompositions possibles précédemment citées.

Le tableau 3.16 résume les contraintes à mettre en place pour les décompositions d’héritage possibles, à partir de l’exemple en cours. Nous expliquerons comment programmer avec SQL les différents prédicats (numérotés de A à D) composant chaque contrainte, en fonction du schéma relationnel choisi. Nous considérons que les tables relationnelles sont déjà créées.

Décomposition par distinction

Contrainte de partition

La contrainte de partition exprime le fait qu’il n’existe aucun personnel pouvant être à la fois PNT et PNC (prédicat A), et qu’il n’existe pas non plus de personnel n’étant ni PNT ni PNC (prédicat B).

Le prédicat A se programme à l’aide de déclencheurs (*triggers*) sur les tables PNT et PNC. Concernant les ajouts dans la table PNT et la modification du numéro (`numPers`), le déclencheur devra s’assurer qu’il n’existe pas d’enregistrement PNC ayant le même numéro.

Tableau 3.16 Différents cas d'héritage

Héritage	Décomposition	distinction	descendante	ascendante
Partition Merise/2 :  UML : {complète, disjoint}		Personnel[numPers, nomPers] PNC[numPers#, indice, prime] PNT[numPers#, brevet, valideLicence]	PNC[numPers, nomPers, indice, prime] PNT[numPers, nomPers, brevet, valideLicence]	Personnel[numPers, nomPers, indice, prime, brevet, valideLicence]
Totalité Merise/2  UML : {complète, overlapping}				
∄ un personnel à la fois PNT et PNC (A) ∄ un personnel ni PNT ni PNC (B)				
Exclusivité Merise/2 :  UML : {incomplète, disjoint} ou rien			Personnel[numPers, nomPers] PNC[numPers#, nomPers, indice, prime] PNT[numPers#, nomPers, brevet, valideLicence]	
∄ un personnel à la fois PNT et PNC (A) ∃ un personnel ni PNT ni PNC (D)				
Sans contrainte Merise/2 :  UML : {incomplète, overlapping}				
∃ un personnel à la fois PNT et PNC (C) ∃ un personnel ni PNT ni PNC (D)				

Il en va de même pour le déclencheur de la table PNC qui devra s'assurer qu'il n'existe pas d'enregistrement de PNT ayant même numéro.



Bien que les deux déclencheurs soient écrits avec une syntaxe Oracle, il est aisé de les programmer par analogie avec un autre SGBD.

La clause `RAISE_APPLICATION_ERROR` génère un `ROLLBACK` et envoie un message applicatif qui n'est pas dans la plage réservée des erreurs Oracle. Nous verrons dans le cas de la décomposition descendante un exemple d'exécution de ce déclencheur.

La clause `NO_DATA_FOUND` permet de lever l'exception lors d'une sélection ne ramenant aucun enregistrement.

```
-- déclencheur sur PNT
CREATE TRIGGER Tri_B_IU_PNT
    BEFORE INSERT OR UPDATE OF numPers ON PNT FOR EACH ROW
DECLARE
    num NUMBER;
BEGIN
    SELECT numPers INTO num FROM PNC WHERE numPers = :NEW.numPers;
    RAISE_APPLICATION_ERROR(-20001, 'Le personnel' || TO_CHAR(num) || '
est déjà PNC...');
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        NULL;
END;
```

Étiquette prédéfinie d'Oracle

Artifice pour éviter que le déclencheur ne renvoie pas d'erreur si le `SELECT` ne renvoie aucune ligne

```
-- déclencheur sur PNC
CREATE TRIGGER Tri_B_IU_PNC
    BEFORE INSERT OR UPDATE OF numPers ON PNC FOR EACH ROW
DECLARE
    num NUMBER;
BEGIN
    SELECT numPers INTO num FROM PNT WHERE numPers = :NEW.numPers;
    RAISE_APPLICATION_ERROR(-20001, 'Le personnel' || TO_CHAR(num) || '
est déjà PNT...');
EXCEPTION
    WHEN NO_DATA_FOUND THEN NULL;
END;
```

Le prédicat *B* se programme à l'aide de deux procédures cataloguées et d'un déclencheur. L'ajout d'un personnel sera répercuté dans la table `Personnel` et dans la table `PNT` ou dans la table `PNC`. Il en va de même pour la modification du numéro d'un personnel ou pour la suppression d'un personnel qui devra répercuter la suppression dans la table `PNT` ou dans la table `PNC`.

Les deux procédures cataloguées réalisent respectivement l'ajout d'un personnel `PNT` et d'un personnel de type `PNC`. L'appel `EXECUTE ajout_PNT(6, 'Laurent', '4MPY01', SYSDATE)` déclenchera deux insertions : une dans la table `Personnel` avec les valeurs (6, 'Laurent'), et l'autre dans la table `PNT` avec les valeurs (6, '4MPY01', SYSDATE). SYSDATE désignant la date du jour Oracle.

```
-- Ajout d'un PNT
CREATE PROCEDURE ajout_PNT(num NUMBER, nom VARCHAR,
    brevet VARCHAR, licence DATE) IS
BEGIN
    INSERT INTO Personnel VALUES (num,nom);
    INSERT INTO PNT VALUES (num,brevet,licence);
END;

-- Ajout d'un PNC
CREATE PROCEDURE ajout_PNC(num NUMBER, nom VARCHAR, ind NUMBER,
    prim NUMBER) IS
BEGIN
    INSERT INTO Personnel VALUES (num,nom);
    INSERT INTO PNC VALUES (num,ind,prim);
END;
```

Les procédures suivantes réalisent respectivement la suppression d'un personnel PNT et d'un personnel de type PNC. On aurait pu également gérer la suppression à l'aide de déclencheurs sur les trois tables.

```
-- Suppression d'un PNT
CREATE PROCEDURE enleve_PNT(num NUMBER) IS
BEGIN
    DELETE FROM PNT      WHERE numPers=num;
    DELETE FROM Personnel WHERE numPers=num;
END;

-- Suppression d'un PNC
CREATE PROCEDURE enleve_PNC(num NUMBER) IS
BEGIN
    DELETE FROM PNC      WHERE numPers=num;
    DELETE FROM Personnel WHERE numPers=num;
END;
```

La modification d'un numéro de personnel au niveau de la table `Personnel` doit être répercutée dans la table `PNT` ou `PNC`. Pour réaliser ces répercussions, nous programmons le déclencheur suivant.

```
-- Répercussion de Personnel vers PNT ou PNC
CREATE TRIGGER Tri_B_U_Personnel
    BEFORE UPDATE OF numPers ON Personnel FOR EACH ROW
BEGIN
    UPDATE PNC SET numPers = :NEW.numPers
        WHERE numPers = :OLD.numPers;
    UPDATE PNT SET numPers = :NEW.numPers
        WHERE numPers = :OLD.numPers;
END;
```

Contrainte de totalité

La contrainte de totalité exprime le fait qu'il n'existe aucun personnel n'étant ni PNT ni PNC (prédicat *B*), et qu'il peut exister un personnel à la fois PNT et PNC (prédicat *C*). Le prédicat *B* a été programmé plus avant. Le prédicat *C* revient à ne pas programmer le prédicat *A* précédent (ne pas mettre en œuvre les déclencheurs des tables PNT et PNC).

Contrainte d'exclusivité

La contrainte d'exclusivité exprime le fait qu'il n'existe aucun personnel pouvant être à la fois PNT et PNC (prédicat *A*), et qu'il peut exister un personnel ni PNT ni PNC (prédicat *D*). Le prédicat *A* a été programmé plus avant. Le prédicat *D* revient à ne pas programmer le prédicat *B* précédent (ne pas mettre en œuvre les procédures d'ajout et de suppression et le déclencheur de la table `Personnel`).

Sans contrainte

Aucune contrainte n'est à programmer.

Décomposition descendante

Contrainte de partition

Le prédicat *A* se programme à l'aide des mêmes déclencheurs sur les tables PNT et PNC que pour la décomposition ascendante. Le code suivant illustre une tentative d'ajout d'un PNT, qui est déjà PNC lorsque les déclencheurs sont actifs.

```
-- État des tables
SQL> SELECT * FROM PNT;
  NUMPERS  NOMPERS                BREVET      VALIDITE
-----
          1 Tanguy                4MPY68      10/12/02
          3 Laverdure             3MPY69      11/12/03
SQL> SELECT * FROM PNC;
  NUMPERS  NOMPERS                INDICE      PRIME
-----
          2 Natacha                567        15000,2
-- tentative d'ajout d'un PNT déjà PNC
SQL> INSERT INTO PNT values (2, 'Natacha', '3MPY99', '11-12-2003');
ERREUR à la ligne 1 : ORA-20001: Le personnel 2 est déjà PNC...
ORA-06512: à "SOUTOU.TRI_B_IU_PNT", ligne 6
```

Le prédicat *B* n'a pas à être programmé si la table `Personnel` est inexistante (voir chapitre 2, section *Décomposition ascendante* : quand il existe une contrainte de totalité ou de partition sur l'association d'héritage, il est possible de ne pas traduire la table issue de la sur-entité). En effet, avec les tables PNT et PNC seules, il n'est pas possible d'avoir un enregistrement ni PNT ni PNC. En revanche, si la table `Personnel` est présente, il faut procéder de la même façon que pour la décomposition ascendante (procédures et déclencheur sur la table `Personnel`).

Contrainte de totalité

La contrainte de partition doit rendre valide le prédicat *B* et il peut exister un personnel à la fois PNT et PNC (prédicat *C*). Nous avons parlé plus haut du prédicat *B*. Le prédicat *C* revient à ne pas programmer le prédicat *A* précédent (ne pas mettre en œuvre les déclencheurs des tables PNT et PNC).

Contrainte d'exclusivité

Dans ce cas et dans le suivant, la table `Personnel` doit exister pour contenir les employés n'étant ni PNT ni PNC (prédicat *D*). Pour l'exclusivité, le prédicat *A* se programme comme pour la décomposition ascendante. Le prédicat *D* revient à ne pas programmer le prédicat *B* précédente (ne pas mettre en œuvre les procédures d'ajout et de suppression et le déclencheur de la table `Personnel`).

Sans contrainte

Aucun prédicat n'est à programmer et la table `Personnel` doit être présente dans la base.

Décomposition ascendante (push-up)

Dans cette décomposition, on ne retrouve qu'une seule relation au niveau logique, qui devient une table (`Personnel`) au niveau physique. Cette table contient toutes les colonnes issues des sous-entités (sous-classes).

Contrainte de partition

Les prédicats *A* et *B* se programment au niveau de la table `Personnel` à l'aide de contraintes de validation `CHECK` :

- Pour le prédicat *A*, il faut que les colonnes `indice`, `prime`, `brevet` et `validite` ne soient pas toutes initialisées.
- Pour le prédicat *B*, il faut que les colonnes `indice`, `prime`, `brevet` et `validite` ne soient pas toutes nulles.

L'ordre `ALTER TABLE` modifie la structure d'une table (colonne ou contrainte SQL), ici on ajoute deux contraintes.

```
--- prédicat A
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_A
    CHECK ((indice IS NULL AND prime IS NULL)
           OR (brevet IS NULL AND validiteLicence IS NULL));
--- prédicat B
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_B
    CHECK ((indice IS NOT NULL OR prime IS NOT NULL)
           OR (brevet IS NOT NULL OR validiteLicence IS NOT NULL));
```

Les tentatives d'ajout d'un personnel à la fois PNT et PNC et d'un personnel n'étant ni l'un ni l'autre se traduiront par un échec.

```
-- État de la base
SQL> SELECT * FROM Personnel;
  NUMPERS  NOMPERS          INDICE      PRIME  BREVET      VALIDITE
-----
      1  Tanguy
      2  Natacha          567    15000,2
      3  Laverdure
-- Ajout d'un personnel PNT et PNC
SQL> INSERT INTO Personnel VALUES (4,'Soutou', 780, 100.00,
  '3MPY93', '11-12-2003');
ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOU-
TOU.CK_PREDICAT_A) de vérification
-- Ajout d'un personnel ni PNT ni PNC
SQL> INSERT INTO Personnel VALUES (5,'Bidal',NULL,NULL,NULL,NULL);
ERREUR à la ligne 1 : ORA-02290: violation de contraintes (SOU-
TOU.CK_PREDICAT_B) de vérification
```

Contrainte de totalité

Les prédicats *B* et *C* sont à programmer. Nous avons parlé plus haut de la contrainte *B*. La contrainte *C* revient à supprimer la contrainte *A* précédente avec l'option **DROP CONSTRAINT**.

```
--- Désactivation du prédicat A
ALTER TABLE Personnel DROP CONSTRAINT ck_predicat_A;
```

On peut désormais insérer un personnel à la fois PNT et PNC comme le montre le script suivant :

```
SQL> INSERT INTO Personnel VALUES (4,'Soutou', 780, 100.00,
  '3MPY93', '11-12-2003');
SQL> SELECT * FROM Personnel;
  NUMPERS  NOMPERS          INDICE      PRIME  BREVET      VALIDITE
-----
      1  Tanguy
      2  Natacha          567    15000,2
      3  Laverdure
      4  Soutou          780    100 3MPY93    11/12/03
```

Contrainte d'exclusivité

Il faut redéfinir le prédicat *A* en supprimant au préalable les enregistrements ne répondant pas à ce critère, et programmer le prédicat *D* en supprimant la contrainte vérifiant le prédicat *B*.

```
--- Activation du prédicat A
ALTER TABLE Personnel ADD CONSTRAINT ck_predicat_A
  CHECK ((indice IS NULL AND prime IS NULL)
    OR (brevet IS NULL AND validiteLicence IS NULL));
--- Activation du predicat D
ALTER TABLE Personnel DROP CONSTRAINT ck_predicat_B;
```

Sans contrainte

Aucune contrainte de validation (CHECK) n'est à programmer.

Bilan

Dressons un bilan des avantages et des inconvénients de chacune de ces approches. Aucune des solutions ne constitue la panacée. Il semblerait que la décomposition descendante soit la plus pénalisante. La décomposition ascendante semble être la plus souple au point de vue de la programmation des contraintes, mais n'est pas optimale d'un point de vue stockage des informations. La décomposition par distinction peut paraître comme un compromis entre ces deux solutions.

Tableau 3.17 Bilan des traductions de l'héritage

Décomposition	distinction	descendante	ascendante
Avantages	Pas de redondances de données. Prédicats <i>C</i> et <i>D</i> implicites.	Diminution du nombre de jointures en exploitation. Prédicats <i>C</i> et <i>D</i> implicites.	Simplicité des prédicats <i>A</i> et <i>B</i> à programmer. Prédicats <i>C</i> et <i>D</i> implicites.
Inconvénients	Prédicats <i>A</i> et <i>B</i> à programmer.	Redondances possibles de données. Prédicats <i>A</i> et <i>B</i> à programmer.	Possibilité de valeurs nulles dans les données.

Transformation des agrégations

Pour UML, l'agrégation renforce le couplage d'une association. Pour le modèle entité-association, il s'agit d'affiner une association *n*-aire (qui se traduit en classes-associations dans la notation UML).

Agrégations UML

UML distingue une agrégation partagée (ou simple) d'une composition. L'agrégation partagée se traduit sous SQL comme une simple association. La composition nécessite d'inclure dans la clé primaire de la table composant l'identifiant de la table composite.

L'exemple 3-18 décrit une agrégation simple *plusieurs-à-plusieurs* et une composition *un-à-plusieurs* (une composition est toujours *un-à-plusieurs* ou *un-à-un*).

La première association est agrégée dans le sens où la suppression d'un immeuble doit se répercuter sur la suppression des achats relatifs à cet immeuble. Il ne s'agit pas de détruire les copropriétaires de l'immeuble supprimé, car ils peuvent être copropriétaires d'autres immeubles. En appliquant les règles R1 et R3, on obtient trois relations *CoPropriétaire*, *Immeuble* et *Achat*.



La programmation SQL de l'agrégation UML (partagée ou composition) se réalise à l'aide des directives `UPDATE CASCADE` et `DELETE CASCADE` qui permettent de propager la suppression (ou la modification) d'un enregistrement composite au niveau du composant.

Figure 3-18 Agrégations UML

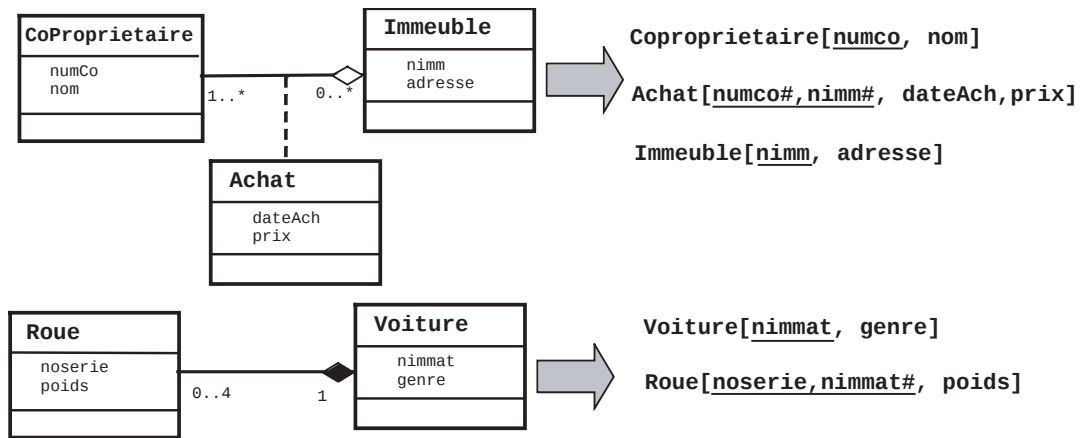


Tableau 3.18 Agrégations UML

Schéma logique	Script SQL2 (programmation MySQL)
	<pre>CREATE TABLE Coproprietaire (numCo INTEGER, nom CHAR(30), CONSTRAINT pk_Coproprietaire PRIMARY KEY(numCo));</pre>
Coproprietaire [<u>numCo</u> , nom]	<pre>CREATE TABLE Immeuble (nimm INTEGER, adresse CHAR(40), CONSTRAINT pk_Immeuble PRIMARY KEY(nimm));</pre>
Achat [<u>numCo#</u> , <u>nimm#</u> , dateAch, prix]	<pre>CREATE TABLE Achat (numCo INTEGER, nimm INTEGER, dateAch DATETIME, prix INTEGER, CONSTRAINT pk_Achat PRIMARY KEY (numCo,nimm), CONSTRAINT fk_Achat_numCo_Coproprietaire FOREIGN KEY(numCo)REFERENCES Coproprietaire(numCo) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT fk_Achat_nimm_Immeuble FOREIGN KEY(nimm) REFERENCES Immeuble(nimm) ON DELETE CASCADE ON UPDATE CASCADE);</pre>
Immeuble [<u>nimm</u> , adresse]	
Voiture [<u>nimmat</u> , genre]	<pre>CREATE TABLE Voiture (nimmat CHAR(10), genre CHAR(30), CONSTRAINT pk_Voiture PRIMARY KEY(nimmat));</pre>
Roue [<u>noserie#</u> , <u>nimmat#</u> , poids]	<pre>CREATE TABLE Roue (noserie INTEGER, nimmat CHAR(10), poids INTEGER, CONSTRAINT pk_Roue PRIMARY KEY(noserie,nimmat), CONSTRAINT fk_Roue_nimmat_Voiture FOREIGN KEY (nimmat) REFERENCES Voiture(nimmat) ON DELETE CASCADE ON UPDATE CASCADE);</pre>

Modèle entité-association

En considérant l'agrégat comme une entité dans le passage au niveau logique, il n'y a pas de difficulté particulière au niveau de SQL pour traduire une association reliant l'agrégat à une entité. Puisque les associations d'agrégation du modèle entité-association se traduisent sous UML à l'aide de classes-associations, il est intéressant d'étudier leur transformation avec SQL2.

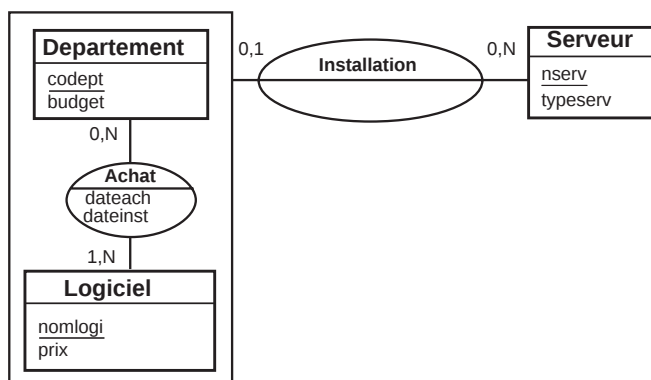
L'exemple 3-19 décrit les installations des logiciels d'un département sur des serveurs. L'association 3-aire s'affine à l'aide d'une association d'agrégation dans le sens où la base de données ne devra pas stocker des installations de logiciels non achetés. Nous poserons par la suite des conditions supplémentaires de manière à examiner les trois autres cardinalités possibles.

Associations d'agrégation un-à-plusieurs

Un logiciel acheté n'est installé que sur un seul serveur qui peut héberger plusieurs logiciels achetés.

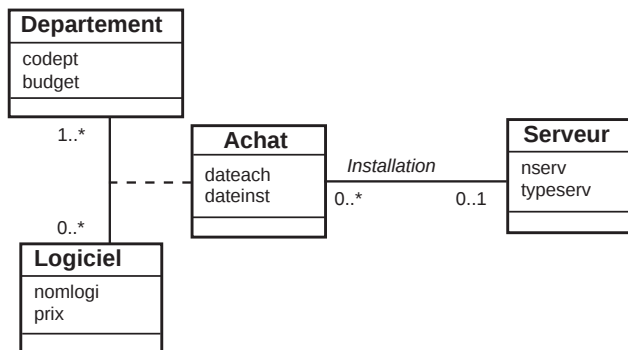
Modèle entité-association

Figure 3-19 Association d'agrégation un-à-plusieurs



Notation UML

Figure 3-20 Classe-association un-à-plusieurs



Script SQL2

En appliquant la règle R1, on obtient trois relations `Departement`, `Logiciel` et `Serveur`. En appliquant la règle R3, on obtient la relation `Achat`. La règle R2 traduit l'association `Installation` en faisant migrer dans la relation *fils* (ici l'agrégat `Achat`) l'identifiant de la relation *père* (ici `Serveur`).

Tableau 3.19 Association d'agrégation un-à-plusieurs

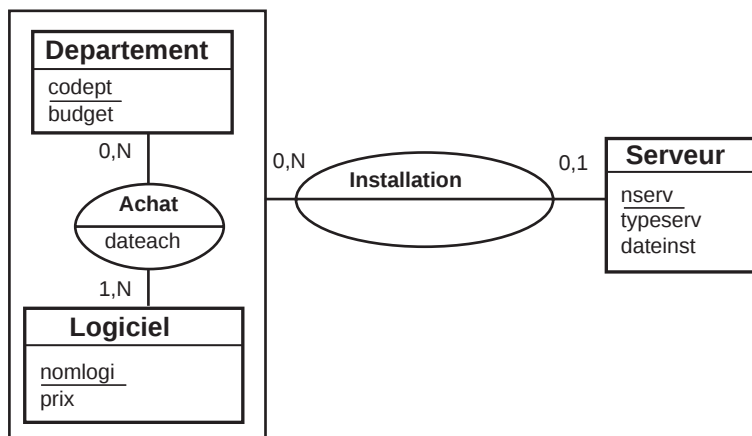
Schéma logique	Script SQL2 (Oracle)
<pre>classDiagram class Logiciel { <u>nomlogi</u> prix } class Departement { <u>codept</u> budget } class Achat { <u>codept#</u> <u>nomlogi#</u> dateach <u>nserv#</u> dateinst } class Serveur { <u>nserv</u> typeserv } Logiciel "1" -- "1" Achat Achat "1" *-- "1" Serveur</pre>	<pre>CREATE TABLE logiciel (nomlogi VARCHAR(20), prix NUMBER, CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi)) CREATE TABLE departement (codept VARCHAR(6), budget NUMBER, CONSTRAINT pk_departement PRIMARY KEY(codept)) CREATE TABLE serveur (nserv NUMBER, typeserv VARCHAR(20), CONSTRAINT pk_serveur PRIMARY KEY(nserv)) CREATE TABLE achat (codept VARCHAR(6), nomlogi VARCHAR(20), dateach DATE, nserv NUMBER, dateinst DATE, CONSTRAINT pk_achat PRIMARY KEY (codept,nomlogi), CONSTRAINT fk_achat_codept_departement FOREIGN KEY(codept) REFERENCES departement(codept), CONSTRAINT fk_achat_nomlogi_logiciel FOREIGN KEY(nomlogi) REFERENCES logiciel(nomlogi), CONSTRAINT fk_achat_nserv_serveur FOREIGN KEY(nserv) REFERENCES serveur(nserv))</pre>

Associations d'agrégation plusieurs-à-un

Un serveur n'héberge qu'un seul logiciel acheté qui peut être installé sur plusieurs serveurs.

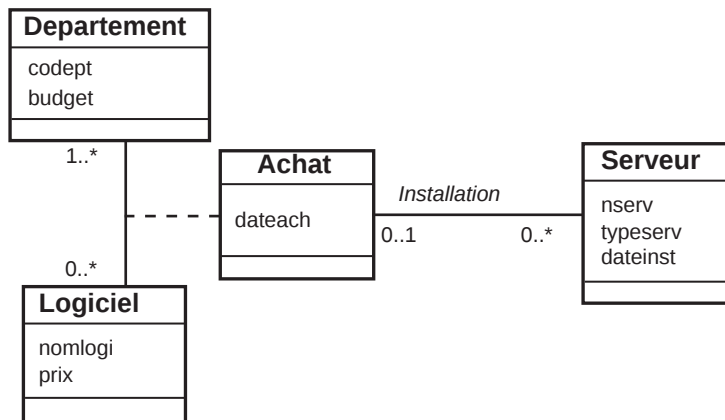
Modèle entité-association

Figure 3-21 Association d'agrégation plusieurs-à-un



Notation UML

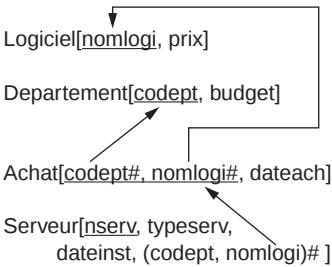
Figure 3-22 Classe-association plusieurs-à-un



Script SQL2

En appliquant la règle R1, on obtient les relations `Departement`, `Logiciel` et `Serveur`. En appliquant la règle R3, on obtient la relation `Achat`. La règle R2 traduit l'association `Installation` en faisant migrer dans la relation *fils* (ici `Serveur`) l'identifiant de la relation *père* (le couple `codept,nomlogi` de l'agrégat `Achat`). Nous indiquons en gras cette migration dans le script SQL2 du tableau 3.20.

Tableau 3.20 Association d'agrégation plusieurs-à-un

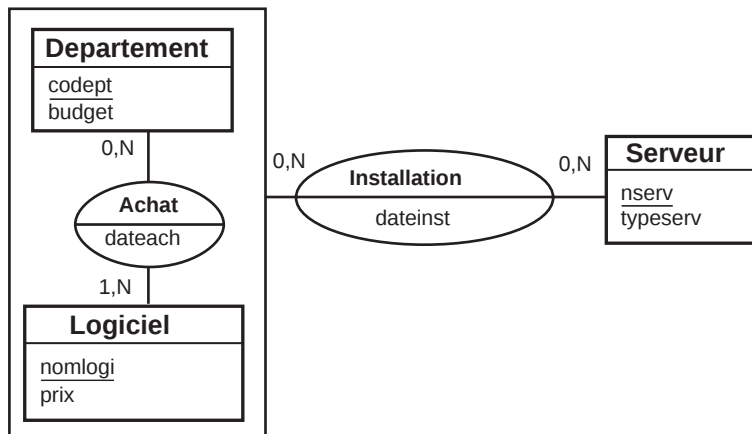
Schéma logique	Script SQL2 (Oracle)
	<pre>CREATE TABLE logiciel (nomlogi VARCHAR(20), prix NUMBER, CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi)) CREATE TABLE departement (codept VARCHAR(6), budget NUMBER, CONSTRAINT pk_departement PRIMARY KEY(codept)) CREATE TABLE achat (codept VARCHAR(6), nomlogi VARCHAR(20), dateach DATE, CONSTRAINT pk_achat PRIMARY KEY (codept,nomlogi), CONSTRAINT fk_achat_codept_departement FOREIGN KEY(codept) REFERENCES departement(codept), CONSTRAINT fk_achat_nomlogi_logiciel FOREIGN KEY(nomlogi) REFERENCES logiciel(nomlogi)); CREATE TABLE serveur (nserv NUMBER, typeserv VARCHAR(20), dateinst DATE, codept VARCHAR(6), nomlogi VARCHAR(20), CONSTRAINT pk_serveur PRIMARY KEY(nserv), CONSTRAINT fk_serveur_achat FOREIGN KEY(codept,nomlogi) REFERENCES achat(codept,nomlogi))</pre>

Associations d'agrégation plusieurs-à-plusieurs

Un serveur héberge plusieurs logiciels achetés. Un logiciel pouvant être installé sur différents serveurs.

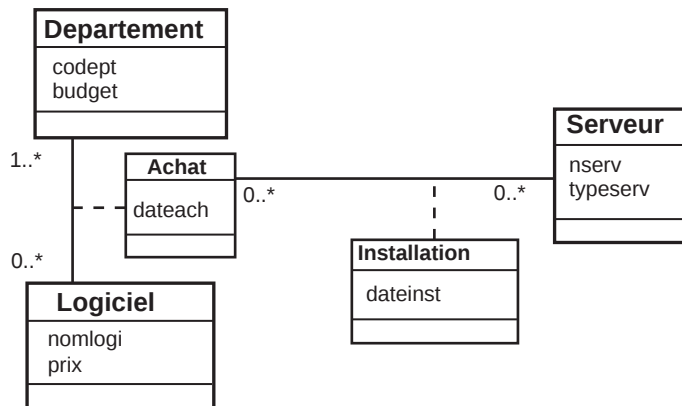
Modèle entité-association

Figure 3-23 Association d'agrégation plusieurs-à-plusieurs



Notation UML

Figure 3-24 Classe-association plusieurs-à-plusieurs



Script SQL2

En appliquant la règle R1, on obtient les relations **Departement**, **Logiciel** et **Serveur**. La règle R3 s'applique pour les deux associations *plusieurs-à-plusieurs* (**Achat** et **Installation**). La clé primaire de la relation **Installation** contient les identifiants des entités/classes connectées (nserv avec le couple codept,nomlogi de l'agrégat représenté par la classe-association **Achat**). Comme ce couple est par ailleurs clé primaire de la relation **Achat**, il est devenu clé étrangère dans la relation **Installation**.

C’est cette contrainte de clé étrangère qui renforce l’intégrité de la base de données, car il ne sera pas possible d’ajouter l’installation d’un logiciel si ce précédent n’a pas été acheté au préalable par le département. Ce schéma est plus intègre qu’un schéma traduisant l’association 3-aire sans contrainte entre Département, Logiciel et Serveur. La clé étrangère qui traduit l’association d’agrégation est indiquée en gras.

Tableau 3.21 Association d’agrégation plusieurs-à-plusieurs

Schéma logique	Script SQL2 (Oracle)
<pre>classDiagram class Logiciel { nomlogi prix } class Departement { codept budget } class Achat { codept# nomlogi# dateach } class Installation { nserv# "codept,nomlogi"# dateinst } class Serveur { nserv typeserv } Logiciel --> Achat Departement --> Achat Achat --> Installation Installation --> Serveur</pre>	<pre>CREATE TABLE logiciel (nomlogi VARCHAR(20), prix NUMBER, CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi)) CREATE TABLE departement (codept VARCHAR(6), budget NUMBER, CONSTRAINT pk_departement PRIMARY KEY(codept)) CREATE TABLE serveur (nserv NUMBER, typeserv VARCHAR(20), CONSTRAINT pk_serveur PRIMARY KEY(nserv)) CREATE TABLE achat (codept VARCHAR(6), nomlogi VARCHAR(20), dateach DATE, CONSTRAINT pk_achat PRIMARY KEY(codept,nomlogi), CONSTRAINT fk_achat_codept_departement FOREIGN KEY(codept) REFERENCES departement(codept), CONSTRAINT fk_achat_nomlogi_logiciel FOREIGN KEY(nomlogi) REFERENCES logiciel(nomlogi)) CREATE TABLE installation (nserv NUMBER, codept VARCHAR(6), nomlogi VARCHAR(20), dateinst DATE, CONSTRAINT pk_installation PRIMARY KEY(nserv,codept,nomlogi), CONSTRAINT fk_installation_serveur FOREIGN KEY(nserv) REFERENCES serveur(nserv), CONSTRAINT fk_installation_achat FOREIGN KEY(codept,nomlogi) REFERENCES achat(codept,nomlogi))</pre>

Traduction des contraintes

Cette section est consacrée à la programmation SQL2 des différentes contraintes du niveau conceptuel (partition, exclusivité, totalité, inclusion et simultanéité).

Contrainte de partition

Considérons l'exemple des pilotes (caractérisés par un numéro, un nom et un grade) qui partent soit en mission sanitaire, soit en mission d'entraînement. La contrainte de partition détermine le fait qu'aucun pilote n'est au repos ou ne mène de front des missions des deux types. Les modèles conceptuels sont illustrés figures 1-37 et 1-40.



La partition se programme par une contrainte SQL de validation (CHECK).

Tableau 3.22 Contrainte de partition

Schéma logique	Script SQL2 (Oracle)
	<pre>CREATE TABLE sanitaire (codesan VARCHAR(10), organisme VARCHAR(20), CONSTRAINT pk_sanitaire PRIMARY KEY(codesan))</pre>
Sanitaire[<u>codesan</u> , organisme]	
	<pre>CREATE TABLE entrainement (codent VARCHAR(10), datent DATE, region VARCHAR(20), CONSTRAINT pk_entraîne PRIMARY KEY(codent))</pre>
Pilote[<u>numpil</u> , nom, grade, sani#, entraîne#]	<pre>CREATE TABLE pilote (numpil NUMBER, nom VARCHAR(10), grade VARCHAR(10), sani VARCHAR(10), entraîne VARCHAR(10), CONSTRAINT fk_pilote_sanitaire FOREIGN KEY(sani) REFERENCES sanitaire(codesan), CONSTRAINT fk_pilote_entrainement FOREIGN KEY(entraîne) REFERENCES entrainement(codent),</pre>
Entrainement[<u>codent</u> , datent, region]	<pre>CONSTRAINT ck_partition CHECK ((sani IS NOT NULL OR entraîne IS NOT NULL) AND NOT (sani IS NOT NULL AND entraîne IS NOT NULL)), CONSTRAINT pk_pilote PRIMARY KEY(numpil))</pre>

Contrainte d'exclusivité

Dans notre exemple, un pilote peut être au repos (affecté à aucune mission). Si un pilote est affecté à un exercice d'entraînement, alors il ne peut pas être affecté à une mission sanitaire et réciproquement.

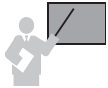


L'exclusivité se programme par une contrainte SQL de validation (CHECK).

```
CREATE TABLE pilote
(numpil NUMBER, nom VARCHAR(10),
grade VARCHAR(10), sani VARCHAR(10), entraine VARCHAR(10),
CONSTRAINT fk_pilote_sanitaire
    FOREIGN KEY(sani) REFERENCES sanitaire(codesan),
CONSTRAINT fk_pilote_entrainement
    FOREIGN KEY(entraine) REFERENCES entrainement(codent),
CONSTRAINT ck_exclusivite CHECK
    (sani ISNULL OR entraine IS NULL),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte de totalité

Dans notre exemple, un pilote peut être affecté à la fois à une mission sanitaire et à un exercice d'entraînement, et tous les pilotes participent à au moins une mission.

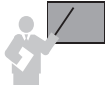


La totalité se programme par une contrainte SQL de validation (CHECK).

```
CREATE TABLE pilote
(numpil NUMBER, nom VARCHAR(10),
grade VARCHAR(10), sani VARCHAR(10), entraine VARCHAR(10),
CONSTRAINT fk_pilote_sanitaire
    FOREIGN KEY(sani) REFERENCES sanitaire(codesan),
CONSTRAINT fk_pilote_entrainement
    FOREIGN KEY(entraine) REFERENCES entrainement(codent),
CONSTRAINT ck_totalité CHECK
    (NOT (sani IS NULL AND entraine IS NULL)),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte de simultanéité

Dans notre exemple, un pilote peut être affecté à la fois à une mission sanitaire et à un exercice d'entraînement. En outre, il peut n'être affecté à aucune mission.



La simultanéité se programme par une contrainte SQL de validation (CHECK).

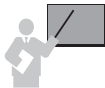
```
CREATE TABLE pilote
(numpil NUMBER, nom VARCHAR(10), grade VARCHAR(10),
sani VARCHAR(10), entraine VARCHAR(10),
CONSTRAINT fk_pilote_sanitaire FOREIGN KEY(sani)
REFERENCES sanitaire(codesan),
CONSTRAINT fk_pilote_entrainement FOREIGN KEY(entraine)
REFERENCES entrainement(codent),
CONSTRAINT ck_simultaneite CHECK
  ➤((sani IS NULL AND entraine IS NULL) OR
  ➤(sani IS NOT NULL AND entraine IS NOT NULL)),
CONSTRAINT pk_pilote PRIMARY KEY(numpil))
```

Contrainte d'inclusion

Selon la contrainte d'inclusion, toutes les occurrences d'une association doivent être incluses dans les occurrences d'une autre association.

Entre deux associations binaires

Considérons l'exemple 1-43 (1-47 pour la notation UML) dans lequel chaque étudiant formule des vœux concernant des stages. Pour modéliser le fait qu'un étudiant accède à un stage à condition qu'il le demande explicitement, il faut utiliser une contrainte d'inclusion. Les règles R1 et R3 s'appliquent pour traduire l'association Vœux. Les règles R1 et R4 s'appliquent pour traduire l'association Effectuer.



L'inclusion entre deux associations se programme par une contrainte SQL de clé étrangère (FOREIGN KEY).

Puisqu'il n'est pas possible de déclarer la contrainte lors de la création de la table *Etudiant* (car la table *Vœux* référence la table *Etudiant*), il faut utiliser l'instruction `ALTER TABLE`.

Entre trois associations binaires

Considérons l'exemple 1-45 dans lequel des départements achètent des logiciels. La contrainte d'inclusion exprime qu'un logiciel *l* acheté par le département *d* est installé sur un serveur *s*, destiné, entre autres, à ce département.

Tableau 3.23 Contraintes d'inclusion

Schéma logique	Script SQL2 (Oracle)
<pre> classDiagram class Stage { numsta theme responsable } class Etudiant { ninsee nom numsta# } class Voeux { ninsee# numsta# } Stage --> Etudiant : numsta to numsta# Etudiant --> Voeux : ninsee to ninsee# Voeux --> Stage : numsta to numsta </pre>	<pre> CREATE TABLE stage (numsta VARCHAR(10), theme VARCHAR(20), responsable VARCHAR(20), CONSTRAINT pk_stage PRIMARY KEY(numsta)) CREATE TABLE etudiant (ninsee VARCHAR(13), nom VARCHAR(30), numsta VARCHAR(10), CONSTRAINT pk_etudiant PRIMARY KEY(ninsee), CONSTRAINT fk_etudiant_stage FOREIGN KEY(numsta) REFERENCES stage(numsta)) CREATE TABLE voeux (ninsee VARCHAR(13), numsta VARCHAR(10), CONSTRAINT fk_voeux_stage FOREIGN KEY(numsta) REFERENCES stage(numsta), CONSTRAINT fk_voeux_etudiant FOREIGN KEY(ninsee) REFERENCES etudiant(ninsee), CONSTRAINT pk_voeux PRIMARY KEY(ninsee,numsta)) ALTER TABLE etudiant ADD CONSTRAINT fk_inclusion FOREIGN KEY(ninsee,numsta) REFERENCES voeux(ninsee,numsta)) </pre>

On peut formuler cette contrainte de la manière suivante : les occurrences de l'association Installe doivent être incluses dans les occurrences issues de la jointure entre les associations Achat et Utilisation (voir la figure 3-25).



L'inclusion entre trois associations (et plus) se programme à l'aide d'un déclencheur (situé dans la table ciblée par la contrainte).

```

CREATE TABLE Logiciel
(nomlogi VARCHAR(20), editeur VARCHAR(30),
 CONSTRAINT pk_logiciel PRIMARY KEY(nomlogi));
CREATE TABLE Departement
(codedept VARCHAR(8), nomdept VARCHAR(30), budget NUMBER,
 CONSTRAINT pk_departement PRIMARY KEY(codedept));
CREATE TABLE Serveur
(nomserv VARCHAR(8), typeserv VARCHAR(30),
 CONSTRAINT pk_serveur PRIMARY KEY(nomserv));

CREATE TABLE Achat

```

```

(codedept VARCHAR(8), nomlogi VARCHAR(20), dateachat DATE,
CONSTRAINT pk_Achat PRIMARY KEY(codedept,nomlogi),
CONSTRAINT fk_Achat_dept FOREIGN KEY(codedept)
REFERENCES Departement(codedept),
CONSTRAINT fk_Achat_logi FOREIGN KEY(nomlogi)
REFERENCES Logiciel(nomlogi));
CREATE TABLE Utilisation
(codedept VARCHAR(8), nomserv VARCHAR(8),
CONSTRAINT pk_Utilisation PRIMARY KEY(codedept,nomserv),
CONSTRAINT fk_Utilisation_serv FOREIGN KEY(nomserv)
REFERENCES Serveur(nomserv),
CONSTRAINT fk_Utilisation_dept FOREIGN KEY(codedept)
REFERENCES Departement(codedept));
CREATE TABLE Installe
(nomserv VARCHAR(8), nomlogi VARCHAR(20),
CONSTRAINT pk_Installe PRIMARY KEY(nomserv,nomlogi),
CONSTRAINT fk_Installe_serv FOREIGN KEY(nomserv)
REFERENCES Serveur(nomserv),
CONSTRAINT fk_Installe_logi FOREIGN KEY(nomlogi)
REFERENCES Logiciel(nomlogi));

CREATE TRIGGER tri_contrainte_inclusion
BEFORE INSERT ON Installe FOR EACH ROW
DECLARE
    V1 VARCHAR(20);
    V2 VARCHAR(8);
BEGIN
SELECT a.nomlogi,u.nomserv INTO v1,v2
FROM Achat a, Utilisation u
WHERE a.codedept = u.codedept
AND a.nomlogi = :NEW.nomlogi
AND u.nomserv = :NEW.nomserv;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR(-20100, 'Le logiciel doit être
placé sur un serveur du département acheteur');
END;
```

Le code suivant décrit deux insertions dans la table `Installe` (la première est correcte du fait des données de l'exemple 3-25, la seconde, incohérente, est rejetée automatiquement par le déclencheur).

```
SQL> INSERT INTO Installe VALUES ('S2','L1');
1 ligne créée.
SQL> INSERT INTO Installe VALUES ('S3','L2');
ERREUR à la ligne 1 : ORA-20100: Le logiciel doit être placé sur un
serveur du département acheteur
ORA-06512: à "SOUTOU.TRI_CONTRAINTE_INCLUSION", ligne 12
```

Figure 3-25 Représentation tabulaire des occurrences

Achat		Utilisation	
Dept.	Logiciel	Dept.	Serveur
D1	L1	D1	S1
D1	L2	D1	S2
D1	L4		
D2	L1	D2	S3

Du modèle objet à SQL3

Alors que la majorité des outils de conception sont capables de générer des classes C++, Java et C# à partir de spécifications UML, aucun n'est adapté à la génération de script SQL3 pour la conception de bases de données objet-relationnelles.

Cette section présente les techniques permettant de dériver un diagramme UML en langage SQL3. Nous utilisons la syntaxe d'Oracle mais l'analogie avec un autre SGBD incluant les références, collections et l'héritage est réalisable à moindre frais.

Traduction des classes UML



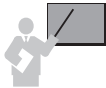
Une classe UML se traduit en un type de données avec la syntaxe CREATE TYPE... et une table objet-relationnelle issue de ce type.

Création d'un type

La commande CREATE TYPE déclare une structure de données simple ou complexe pouvant contenir des déclarations de méthodes. Cette structure est utilisée dans un programme pour définir des objets non persistants ou pour former des tables objet-relationnelles. Le code suivant définit trois types SQL3. Les deux premiers types sont des structures de données simples. Le troisième type inclut deux références (attributs REF) vers des types prédéfinis.

```
CREATE TYPE sanitaire_type AS OBJECT
    (codesan CHAR(10), organisme CHAR(20))
CREATE TYPE entraine_type AS OBJECT
    (codent CHAR(10), datent DATE, region CHAR(20))
CREATE TYPE pilote_type AS OBJECT
    (numpil NUMBER, nom CHAR(10), grade CHAR(10),
     refSani REF sanitaire_type, refEntraîne REF entraine_type)
```

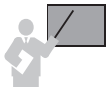
Création d'une table objet-relationnelle



La commande `CREATE TABLE nomtable OF nomdtype` déclare une table objet-relationnelle issue d'un type. Chaque enregistrement de la table est doté d'un unique OID (Object Identifier) qui pourra être utilisé par d'autres objets comme une référence.

La table objet-relationnelle `Pilote`, qui pourra stocker des objets de type `pilote_type`, est déclarée sous SQL3 de la manière suivante.

```
CREATE TABLE Pilote OF pilote_type
```



Le script permettant de supprimer le schéma entier doit détruire d'abord les tables puis les types dans l'ordre inverse de leur création.

```
DROP TABLE Pilote
DROP TYPE pilote_type
DROP TYPE entraine_type
DROP TYPE sanitaire_type
```

La commande `DESC` affiche la structure d'un type ou d'une table. On retrouve la déclaration des deux références au niveau de la table.

```
SQL> DESC pilote
```

Nom	Null ?	Type
NUMPIL		NUMBER
NOM		CHAR(10)
GRADE		CHAR(10)
SANI		REF OF SANITAIRE_
ENTRAINE		REF OF ENTRAINE_T

Associations un-à-un

Examinons les différentes possibilités qu'offre SQL3 pour implanter une association *un-à-un* entre deux classes C1 et C2. Nous recensons deux familles de possibilités.



Solutions 1 et 2 Définition d'un attribut REF ou FOREIGN KEY dans le type qui décrit C1. Cet attribut pointe vers le type qui décrit C2. La deuxième solution est symétrique à la première.

Solution 3 Définition d'un type qui contient deux attributs REF ou FOREIGN KEY vers les types décrivant C1 et C2.

En utilisant deux attributs REF dans la dernière solution, on obtient la solution universelle, qui convient à tous les types d'associations.

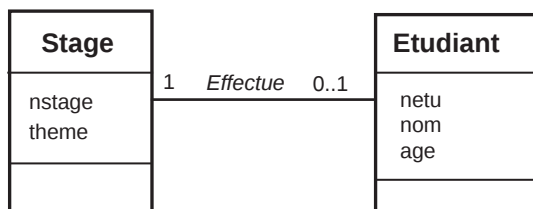


Afin de maintenir la cohérence des cardinalités maximales 1, il conviendrait de définir une contrainte UNIQUE sur chaque référence. Pour les attributs de type FOREIGN KEY, l'opération est possible. En revanche, pour les attributs de type REF, ce n'est pas toujours permis par le SGBD.

L'exemple suivant concerne l'association *un-à-un* entre Stage et Etudiant.

Notation UML

Figure 3-26 Association un-à-un UML



Script SQL3

Adoptons la première solution d'implantation. La multiplicité minimale 1 côté Stage se traduit à l'aide d'une contrainte de type NOT NULL sur la référence. Oracle permet en outre de limiter la portée par la directive SCOPE IS ou de poser une contrainte de clé étrangère REFERENCES (voir tableau 3-25).

Tableau 3.24 Association un-à-un SQL3

Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE stage_type AS OBJECT (nstage CHAR(4), theme CHAR(30))
Stage[nstage, theme]	CREATE TYPE etudiant_type AS OBJECT (netu CHAR(4), nom CHAR(30), age NUMBER, refSta REF stage_type)
	CREATE TABLE Stage OF stage_type (CONSTRAINT pk_stage PRIMARY KEY(nstage));
Etudiant[netu, nom, age, refSta]	CREATE TABLE Etudiant OF etudiant_type (CONSTRAINT pk_etudiant PRIMARY KEY(netu), CONSTRAINT nn_refSta CHECK (refSta IS NOT NULL), CONSTRAINT scope_refSta refSta SCOPE IS Stage)

À propos des références



Bien que les références permettent d'implanter une association entre deux tables, comme le faisaient les clés étrangères, elles n'offrent pas encore toutes les fonctionnalités de ces dernières. Il est possible que l'intégrité référentielle soit à programmer partiellement ou totalement.

Avec Oracle, il est possible de savoir si l'objet *père* d'un objet *fils* donné a été supprimé par la directive DANGLING [SOU 04]. En revanche, il n'est pas encore possible de définir une clé primaire, une contrainte UNIQUE ou un index sur une référence.

Que reste-t-il aux références ? La limitation du nombre de jointures entre des tables et la possibilité de définir des vues objet-relationnelles de tables SQL2.

Associations un-à-plusieurs

Nous recensons trois bases de travail sous SQL3 pour décrire une association *un-à-plusieurs* entre deux classes C1 (*père*) et C2 (*fils*).



Solution 1 Définition d'une collection (NESTED TABLE ou VARRAY pour Oracle) dans le type dérivé de C1. Cette collection contient une référence vers le type dérivé de C2.

Solution 2 Définition d'un attribut REF ou FOREIGN KEY dans le type dérivé de C2 qui référence le type dérivé de C1.

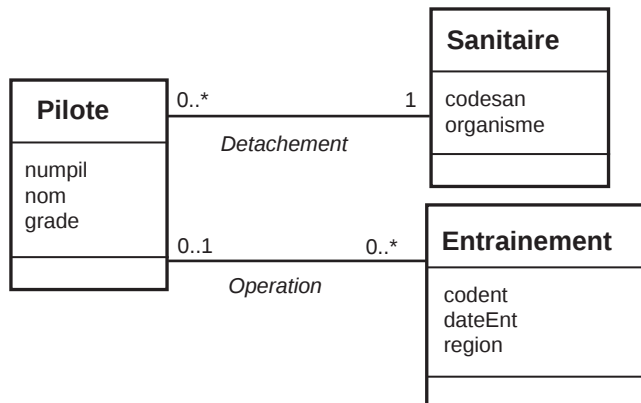
Solution 3 Définition d'un troisième type contenant deux attributs REF ou FOREIGN KEY qui référencent les types dérivés de C1 et C2.

La dernière solution correspond à la solution universelle.

Dans l'exemple 3-27, choisissons de dériver l'association **Detachement** par la première solution, et l'autre avec la deuxième solution.

Notation UML

Figure 3-27 Associations un-à-plusieurs UML



Script SQL3

Pour traduire l'association **Detachement**, la référence `refSani` est placée dans le type `fil` (`pilote_type`) et une contrainte de clé étrangère est définie au niveau de la table **Pilote**. La multiplicité 1 du côté **Sanitaire** est programmée à l'aide d'une contrainte `NOT NULL`. Une collection de références est utilisée pour l'association **Operation** (voir tableau 3-25).

À propos des collections



L'utilisation d'une collection peut privilégier l'accès aux données au niveau de la table qui la contient. Il n'est en général pas possible de définir des contraintes de clé étrangère au niveau d'une référence dans une collection.

Dans notre exemple, l'extraction des missions d'entraînement pour un pilote donné sera plus aisée que l'obtention de la liste des pilotes concernés par une mission d'entraînement.



Si on utilise la solution universelle, il convient de définir une contrainte `UNIQUE` sur la référence qui répertorie la classe affectée de la multiplicité `*`. Oracle ne permet pas de définir une telle contrainte sur un attribut `REF`.

Tableau 3.25 Association un-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)
Sanitaire[codesan, organisme] Pilote[numpil, nom, grade, refSani, Operation{refEnt}] Entrainement[codent, datent, region]	CREATE TYPE sanitaire_type AS OBJECT (codesan CHAR(10), organisme CHAR(20)) CREATE TYPE entrainement_type AS OBJECT (codent CHAR(10), dateEnt DATE, region CHAR(20))
	CREATE TYPE elt_collection_ent AS OBJECT (refEnt REF entrainement_type)
	CREATE TYPE collection_ent AS TABLE OF elt_collection_ent
	CREATE TYPE pilote_type AS OBJECT (numpil NUMBER, nom CHAR(10), grade CHAR(10), refSani REF sanitaire_type, COperation collection_ent)
	CREATE TABLE Sanitaire OF sanitaire_type (CONSTRAINT pk_sanitaire PRIMARY KEY(codesan)) CREATE TABLE Entrainement OF entrainement_type (CONSTRAINT pk_ent PRIMARY KEY(codent))
	CREATE TABLE Pilote OF pilote_type (CONSTRAINT pk_pilote PRIMARY KEY(numpil), CONSTRAINT nn_refSani CHECK (refSani IS NOT NULL), CONSTRAINT fk_refSani refSani REFERENCES Sanitaire) NESTED TABLE Operation STORE AS temp



Pour profiter des avantages des collections de références (accès rapide aux données par une table sans jointures) sans en subir les inconvénients (requête qui interroge la table non privilégiée et qui nécessite une jointure avec la table contenant la collection), il est possible de définir des vues objet-relationnelles [SOU 04], qui privilégieront tantôt l'accès via une table, tantôt l'accès par une autre. Les données pourront être stockées sous la forme relationnelle ou suivant la solution universelle.

Associations plusieurs-à-plusieurs

Nous recensons deux bases de travail sous SQL3 pour décrire une association *plusieurs-à-plusieurs* entre deux classes C1 et C2. La première famille de solutions privilégie un accès aux données par une table.



Solutions 1 et 2 Définition d'une collection dans le type dérivé de C1. Cette collection contient une référence vers le type dérivé de C2. La deuxième solution est l'inverse de la première, à savoir que la collection se trouve dans C2 et référence C1.

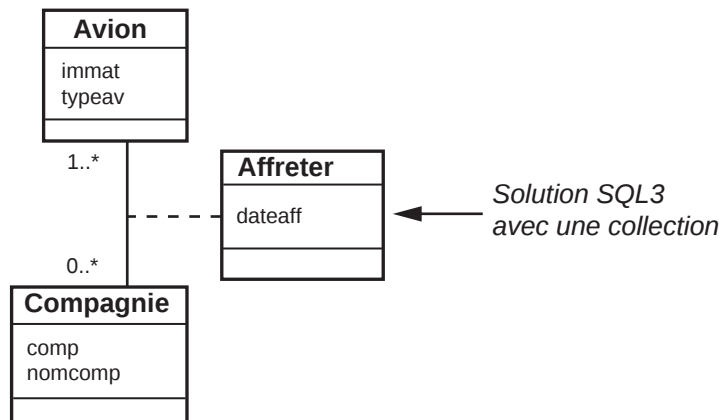
Solution 3 Définition d'un type contenant deux attributs REF ou FOREIGN KEY, qui référencent les types dérivés de C1 et C2.

La dernière solution avec deux références correspond à la solution universelle.

Dérivons l'association *Affreter* de l'exemple 3-28 à l'aide de la première solution (en privilégiant les accès aux données par les compagnies).

Notation UML

Figure 3-28 Association plusieurs-à-plusieurs UML



Script SQL3

La collection collAvi contient la référence refAvi et l'attribut de l'association.

Tableau 3.26 Association plusieurs-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)
Compagnie[comp, nomcomp, collAvi{refAvi, dateaff}] ↙ Avion[immat, typeav]	CREATE TYPE avion_type AS OBJECT (immat VARCHAR(6), typav VARCHAR(10))
	CREATE TYPE elt_coll_avi AS OBJECT (refAvi REF avion_type, dateaff DATE)
	CREATE TYPE coll_avi_type AS TABLE OF elt_coll_avi
	CREATE TYPE compagnie_type AS OBJECT (comp VARCHAR(4), nomcomp VARCHAR(30), collAvi coll_avi_type)
	CREATE TABLE Avion OF avion_type (CONSTRAINT pk_avion PRIMARY KEY(immat))
	CREATE TABLE Compagnie OF compagnie_type (CONSTRAINT pk_compagnie PRIMARY KEY(comp)) NESTED TABLE collAvi STORE AS tempAvi

Associations *n*-aires

Nous recensons *n*+1 bases de travail sous SQL3 pour décrire une association *n*-aire entre *n* classes C1, C2... Cn. La première solution ne privilégie aucun accès aux données, il s'agit de la solution universelle. Les autres solutions privilégient l'accès aux données par une des *n* tables.

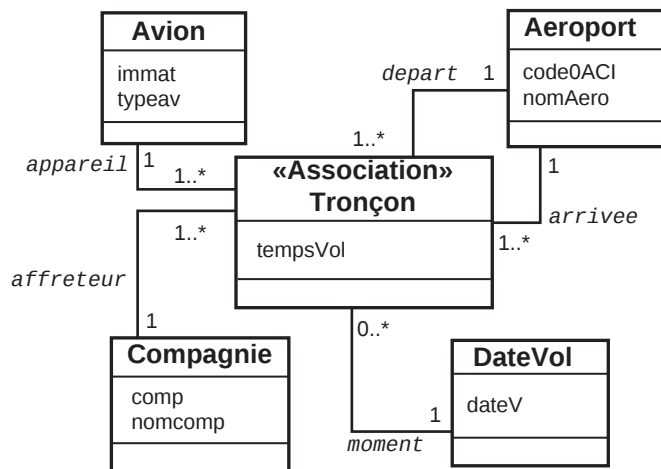


- Solutions 1** Définition d'un type contenant *n* attributs REF ou FOREIGN KEY, qui référencent les *n* types dérivés des classes C1 à Cn.
- Solutions 2 à *n*+1** Définition d'une collection dans un type dérivé de la classe Cj. Cette collection contient *n*-1 références vers les types dérivés des *n*-1 autres classes.

L'exemple 3-29 modélise les temps de vol des avions sur chaque tronçon. Un tronçon est caractérisé par un aéroport de départ et un aéroport d'arrivée. Supposons qu'on désire aussi connaître la compagnie qui a affrété le vol, la date du vol et l'appareil.

Notation UML

Figure 3-29 Association n-aire UML

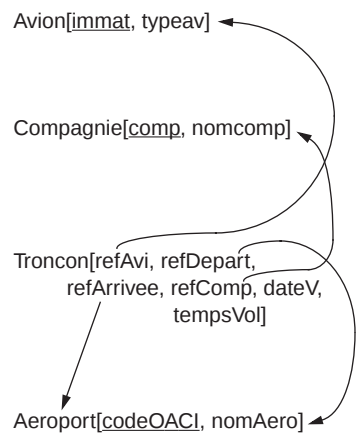


Dérivons l'association Tronçon dans un premier temps avec la première solution, puis dans un second temps en privilégiant les accès aux données par la compagnie.

Solution universelle

Le type `troncon_type` contient autant de références qu'il faut relier de tables concernées par l'association et les éventuels attributs de l'association (ici `tempsVol`). La table associée à ce type doit mettre en œuvre autant de clés étrangères que nécessaire (tableau 3-27).

Tableau 3.27 Association *n*-aire par la solution universelle SQL3

Schéma logique	Script SQL3 (Oracle)
	<pre>CREATE TYPE avion_type AS OBJECT (immat VARCHAR(6), typeav VARCHAR(10)) CREATE TYPE aeroport_type AS OBJECT (codeOACI VARCHAR(6), nomAero VARCHAR(30)) CREATE TYPE compagnie_type AS OBJECT (comp VARCHAR(4), nomcomp VARCHAR(30))</pre>
 Avion[<u>immat</u>, typeav] Compagnie[<u>comp</u>, nomcomp] Troncon[refAvi, refDepart, refArrivee, refComp, dateV, tempsVol] Aeroport[<u>codeOACI</u>, nomAero]	<pre>CREATE TYPE troncon_type AS OBJECT (refAvi REF avion_type, refDepart REF aeroport_type, refArrivee REF aeroport_type, refComp REF compagnie_type, dateV DATE, tempsVol NUMBER) CREATE TABLE Avion OF avion_type (CONSTRAINT pk_avion PRIMARY KEY(immat)) CREATE TABLE Aeroport OF aeroport_type (CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI)) CREATE TABLE Compagnie OF compagnie_type (CONSTRAINT pk_compagnie PRIMARY KEY(comp)) CREATE TABLE Troncon OF troncon_type (CONSTRAINT nn_refAvi CHECK (refAvi IS NOT NULL), CONSTRAINT fk_refAvi refAvi REFERENCES Avion, CONSTRAINT nn_refDepart CHECK(refDepart IS NOT NULL), CONSTRAINT fk_refDep refDepart REFERENCES Aeroport, CONSTRAINT nn_refArrivee CHECK (refArrivee IS NOT NULL), CONSTRAINT fk_refArr refArrivee REFERENCES Aeroport, CONSTRAINT nn_refComp CHECK (refComp IS NOT NULL), CONSTRAINT fk_refCom refComp REFERENCES Compagnie, CONSTRAINT nn_dateV CHECK (dateV IS NOT NULL))</pre>

Avec une collection

La collection collVols contient les références vers les classes concernées par l'association ainsi que les attributs tempsVol et dateV. Aucune contrainte de clé étrangère ne peut être programmée pour l'heure avec Oracle.

Tableau 3.28 Association n-aire par une collection SQL3

Schéma logique	Script SQL3 (Oracle)
<pre>classDiagram class Avion { immat typeav } class Aeroport { codeOACI nomAero } class Compagnie { comp nomcomp collVols refArrivee dateV tempsVol } Avion --> Compagnie Aeroport --> Compagnie Compagnie --> Avion Compagnie --> Aeroport</pre>	<pre>CREATE TYPE avion_type AS OBJECT (immat VARCHAR(6), typav VARCHAR(10)) CREATE TYPE aeroport_type AS OBJECT (codeOACI VARCHAR(6), nomAero VARCHAR(30)) CREATE TYPE elt_collVols AS OBJECT (refAvi REF avion_type, refDepart REF aeroport_type, refArrivee REF aeroport_type, dateV DATE, tempsVol NUMBER) CREATE TYPE collVols_type AS TABLE OF elt_collVols CREATE TYPE compagnie_type AS OBJECT (comp VARCHAR(4), nomcomp VARCHAR(30), collVols collVols_type) CREATE TABLE Avion OF avion_type (CONSTRAINT pk_avion PRIMARY KEY(immat)) CREATE TABLE Aeroport OF aeroport_type (CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI)) CREATE TABLE Compagnie OF compagnie_type (CONSTRAINT pk_compagnie PRIMARY KEY(comp)) NESTED TABLE collVols STORE AS tempVols</pre>

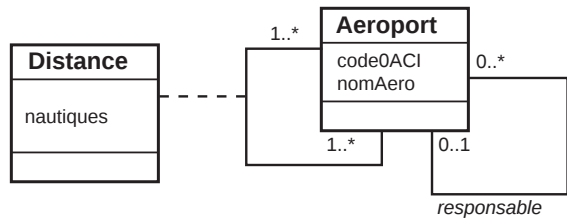
Associations réflexives

Pour traduire une association réflexive, il suffit d'adopter une des solutions précédemment recensées selon la nature de l'association (*un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs* ou *n-aire*). La solution universelle peut convenir dans tous les cas.

Dans l'exemple 3-30, l'association réflexive *plusieurs-à-plusieurs* modélise la distance séparant deux aéroports. Par ailleurs, l'association réflexive *un-à-plusieurs* modélise le fait qu'un aéroport peut avoir sous sa responsabilité plusieurs autres aéroports.

Notation UML

Figure 3-30 Associations réflexives UML



Choisissons de traduire l’association réflexive *plusieurs-à-plusieurs* par la solution universelle et l’association réflexive *un-à-plusieurs* par une collection de références (première solution d’implantation d’une association *un-à-plusieurs*).

Script SQL3

L’association Distance met en œuvre des références et une table précisant les clés étrangères. L’association responsable nécessite une collection (collAero) de références (refAero-Respons).

Tableau 3.29 Associations réflexives SQL3

Schéma logique	Script SQL3 (Oracle)
	<pre>CREATE TYPE aeroport_type CREATE TYPE elt_collAero AS OBJECT (refAeroRespons REF aeroport_type) CREATE TYPE collAero_type AS TABLE OF elt_collAero CREATE TYPE aeroport_type AS OBJECT (codeOACI VARCHAR(6), nomAero VARCHAR(30), collAero collAero_type) CREATE TYPE distance_type AS OBJECT (refAero1 REF aeroport_type, refAero2 REF aeroport_type, nautique NUMBER) CREATE TABLE Aeroport OF aeroport_type (CONSTRAINT pk_aeroport PRIMARY KEY(codeOACI)) NESTED TABLE collAero STORE AS tempAeros</pre>

Tableau 3.29 Associations réflexives SQL3 (suite)

Schéma logique	Script SQL3 (Oracle)
	<pre>CREATE TABLE Distance OF distance_type (CONSTRAINT nn_refAero1 CHECK (refAero1 IS NOT NULL), CONSTRAINT fk_Aero1 refAero1 REFERENCES Aeroport, CONSTRAINT nn_refAero2 CHECK (refAero2 IS NOT NULL), CONSTRAINT fk_Aero2 refAero2 REFERENCES Aeroport)</pre>

Notez qu’il est nécessaire d’utiliser un artifice pour définir une collection de références vers le type contenant la collection. Avec Oracle, la solution consiste à déclarer un type incomplet (première instruction du script). Le type est ensuite redéfini plus loin. Pour régénérer ce schéma, il faut utiliser la directive `DROP TYPE... FORCE` afin de supprimer un type faisant référence à lui-même. Le script de destruction de la base est le suivant.

```
DROP TABLE Distance ;
DROP TABLE Aeroport ;
DROP TYPE distance_type ;
DROP TYPE aeroport_type
DROP TYPE collAero_type;
DROP TYPE aeroport_type;
DROP TYPE elt_collAero;
```

Classes-associations UML

Cette section décrit la traduction SQL3 des associations connectées à des classes-associations UML. Pour chaque association, il faut utiliser une des solutions précédemment étudiées en fonction de la nature de l’association (*plusieurs-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs*).

Considérons pour chacun des cas un exemple. Nous utiliserons arbitrairement une des solutions d’implantation SQL3, que nous avons énoncées précédemment, pour chaque association. D’autres schémas sont possibles en employant d’autres solutions SQL3 de traduction pour chaque association.

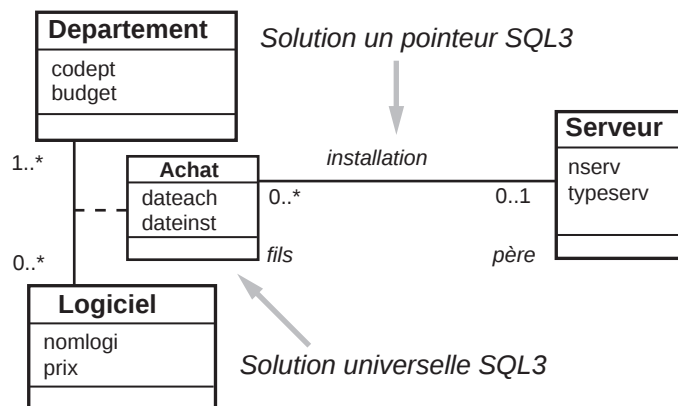
Classe-association un-à-plusieurs

Considérons des serveurs qui hébergent des logiciels achetés par des départements. Un logiciel acheté ne s'installe que sur un seul serveur.

Diagramme UML

La classe-association Achat est reliée à la classe Serveur par une association *un-à-plusieurs*.

Figure 3-31 Classe-association un-à-plusieurs



Script SQL3

Supposons que nous ne privilégions aucun accès aux données, la solution universelle traduit la classe-association Achat avec deux références refLog et refDept et les attributs de l'association. Traduisons à l'aide de la deuxième solution l'association *un-à-plusieurs* Installation. Il en résulte une référence (refServ) dans le type *fils* (Achat) vers le type *père* (Serveur). La table associée inclut une clé étrangère sur la référence.

Tableau 3.30 Classe-association un-à-plusieurs SQL3

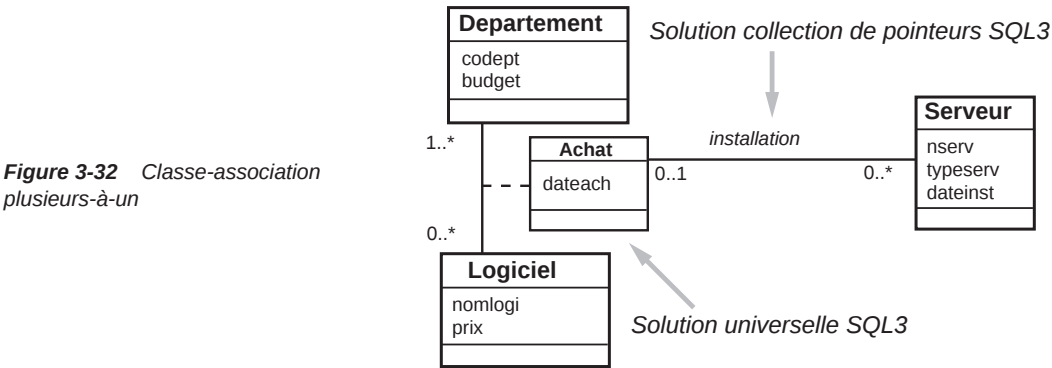
Schéma logique	Script SQL3 (Oracle)
<pre> classDiagram class Departement { <u>codept</u> budget } class Achat { refDept refLog refServ datechat dateinst } class Logiciel { <u>nomlogi</u> prix } class Serveur { <u>nserv</u> typeserv } Departement "1" --> "N" Achat Achat --> Logiciel Achat --> Serveur </pre>	<pre> CREATE TYPE logiciel_type AS OBJECT (nomlogi CHAR(20), prix NUMBER) CREATE TYPE departement_type AS OBJECT (codept CHAR(6), budget NUMBER) CREATE TYPE serveur_type AS OBJECT (nserv NUMBER, typeserv CHAR(20)) CREATE TYPE achat_type AS OBJECT (refDept REF departement_type, refLog REF logiciel_type, refServ REF serveur_type, datechat DATE, dateinst DATE) CREATE TABLE Departement OF departement_type (CONSTRAINT pk_dept PRIMARY KEY(codept)) CREATE TABLE Serveur OF serveur_type (CONSTRAINT pk_serveur PRIMARY KEY(nserv)) CREATE TABLE Logiciel OF logiciel_type (CONSTRAINT pk_log PRIMARY KEY(nomlogi)) CREATE TABLE Achat OF achat_type (CONSTRAINT nn_refDept CHECK (refDept IS NOT NULL), CONSTRAINT fk_refDept refDept REFERENCES Departement CONSTRAINT nn_refLog CHECK (refLog IS NOT NULL), CONSTRAINT fk_refLog refLog REFERENCES Logiciel, CONSTRAINT nn_refServ CHECK (refServ IS NOT NULL), CONSTRAINT fk_refServ refServ REFERENCES Serveur) </pre>

Classe-association plusieurs-à-un

Supposons qu'un serveur n'héberge qu'un seul logiciel et qu'un logiciel acheté par un département puisse être installé sur différents serveurs.

Diagramme UML

La classe-association Achat est liée à la classe Serveur par une association *plusieurs-à-un*.



Script SQL3

Utilisons la solution universelle pour traduire l'association Achat (en gras dans le script). Privilégions l'accès aux données par les logiciels pour l'association d'installation par une collection (collServ) de références (refServ). Cette collection est située dans le type père (Achat) et permet de référencer le type fils (Serveur).

Tableau 3.31 Classe-association plusieurs-à-un SQL3

Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE logiciel_type AS OBJECT (nomlogi CHAR(20), prix NUMBER)
	CREATE TYPE departement_type AS OBJECT (codept CHAR(6), budget NUMBER)
Departement[<u>codept</u> , budget]	CREATE TYPE serveur_type AS OBJECT (nserv NUMBER, typeserv CHAR(20), dateinst DATE)
Achat[refDept, refLog, collServ{refServ}, datechat, dateinst]	CREATE TYPE elt_coll_serv AS OBJECT (reServ REF serveur_type)
	CREATE TYPE coll_serv_type AS TABLE OF elt_coll_serv
Logiciel[nomlogi, prix]	CREATE TYPE achat_type AS OBJECT (refDept REF departement_type, refLog REF logiciel_type, dateach DATE, collServ coll_serv_type)
Serveur[nserv, typeserv]	

Tableau 3.31 Classe-association plusieurs-à-un SQL3 (suite)

Schéma logique	Script SQL3 (Oracle)
	<pre>CREATE TABLE Departement OF departement_type (CONSTRAINT pk_dept PRIMARY KEY(codept))</pre>
	<pre>CREATE TABLE Serveur OF serveur_type (CONSTRAINT pk_serveur PRIMARY KEY(nserv))</pre>
	<pre>CREATE TABLE Logiciel OF logiciel_type (CONSTRAINT pk_log PRIMARY KEY(nomlogi))</pre>
	<pre>CREATE TABLE Achat OF achat_type (CONSTRAINT nn_refDept CHECK (refDept IS NOT NULL), CONSTRAINT fk_refDept refDept REFERENCES Departement, CONSTRAINT nn_refLog CHECK (refLog IS NOT NULL), CONSTRAINT fk_refLog refLog REFERENCES Logiciel)</pre>
	<pre>NESTED TABLE collServ STORE AS tabServ</pre>

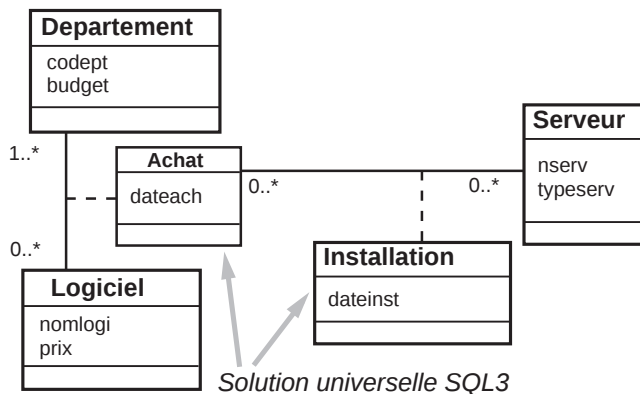
Classe-association plusieurs-à-plusieurs

Supposons qu'un serveur héberge plusieurs logiciels et qu'un logiciel acheté puisse être installé sur différents serveurs.

Diagramme UML

La classe-association Achat est liée à la classe Serveur par une association *plusieurs-à-plusieurs* qui se modélise par la classe-association Installation.

Figure 3-33 Classe-association plusieurs-à-plusieurs



Script SQL3

Supposons que ne nous privilégions aucun accès aux données en utilisant la solution universelle pour traduire les deux classes-associations. Concernant Achat, nous obtenons deux références refLog et refDept avec l'attribut dateach (en surligné dans le script SQL3). Nous traduisons Installation de la même manière. Il en résulte deux références refAchat et refServ avec l'attribut dateinst (en gras dans le script).

Tableau 3.32 Classe-association plusieurs-à-plusieurs SQL3

Schéma logique	Script SQL3 (Oracle)
<pre> graph TD Departement[Departement
[codept, budget]] Achat[Achat
[refDept, refLog, dateach]] Logiciel[Logiciel
[nomlogi, prix]] Installation[Installation
[refServ, refAch, dateinst]] Serveur[Serveur
[nserv, typeserv]] Departement --> Achat Departement --> Logiciel Achat --> Logiciel Installation --> Achat Installation --> Serveur </pre>	<pre> CREATE TYPE logiciel_type AS OBJECT (nomlogi CHAR(20), prix NUMBER) CREATE TYPE departement_type AS OBJECT (codept CHAR(6), budget NUMBER) CREATE TYPE serveur_type AS OBJECT (nserv NUMBER, typeserv CHAR(20)) CREATE TYPE achat_type AS OBJECT (refDept REF departement_type, refLog REF logiciel_type, dateach DATE) CREATE TYPE installation_type AS OBJECT (refServ REF serveur_type, refAch REF achat_type, dateinst DATE) CREATE TABLE Departement OF departement_type (CONSTRAINT pk_dept PRIMARY KEY(codept)) CREATE TABLE Serveur OF serveur_type (CONSTRAINT pk_serveur PRIMARY KEY(nserv)) CREATE TABLE Logiciel OF logiciel_type (CONSTRAINT pk_log PRIMARY KEY(nomlogi)) CREATE TABLE Achat OF achat_type (CONSTRAINT nn_refDept CHECK (refDept IS NOT NULL), CONSTRAINT fk_refDep refDep REFERENCES Departement CONSTRAINT nn_refLog CHECK (refLog IS NOT NULL), CONSTRAINT fk_refLog refLog REFERENCES Logiciel) CREATE TABLE Installation OF installation_type (CONSTRAINT nn_refServ CHECK (refServ IS NOT NULL), CONSTRAINT fk_refServ refServ REFERENCES Serveur, CONSTRAINT nn_refAch CHECK (refAch IS NOT NULL), CONSTRAINT fk_refAch refAch REFERENCES Achat) </pre>

Transformation des associations d'héritage

SQL3 prend en compte l'héritage de types et l'héritage de tables. Nous ne parlons pas ici de décomposition pour traduire une association d'héritage, car la notion d'héritage est intrinsèque au modèle objet.

Héritage de types

En considérant l'exemple 2-48, le schéma SQL3 est composé des types `PNC_type` et `PNT_type` qui héritent du type `Personnel_type` à l'aide de la directive `UNDER`. Les tables `PNC` et `PNT` permettront de stocker des objets persistants. La table `Personnel` contiendra notamment les personnels n'étant ni `PNT` ni `PNC` (c'est un cas possible car l'héritage est sans contrainte).

Tableau 3.33 Héritage de types SQL3

Schéma logique	Script SQL3 (Oracle)
	CREATE TYPE Personnel_type (numPers NUMBER, nomPers VARCHAR(20))
PNC[indice, prime]	CREATE TYPE PNC_type UNDER Personnel_type (indice NUMBER, prime NUMBER(8,2))
↓	
Personnel[numPers, nomPers]	CREATE TYPE PNT_type UNDER Personnel_type (brevet VARCHAR(10), valideLicence DATE)
↑	
PNT[brevet, valideLicence]	CREATE TABLE Personnel OF Personnel_type (CONSTRAINT pk_Personnel PRIMARY KEY(numPers))
	CREATE TABLE PNC OF PNC_type
	CREATE TABLE PNT OF PNT_type

L'avantage de l'héritage de types par rapport à l'héritage de tables réside dans le fait qu'un type, une fois déclaré, peut entrer dans la composition d'un autre type ou peut permettre de définir plusieurs tables.



L'inconvénient de l'utilisation des types se situe au niveau de la modification de la structure des tables en exploitation (Oracle, tout en l'autorisant par `ALTER TYPE`, ne maîtrise pas encore très bien l'ajout ou la suppression de colonnes d'un type composant une table objet-relationnelle).

Héritage de tables

Dans le schéma SQL3 suivant, les tables `PNC` et `PNT` héritent de la table objet-relationnelle `Personnel` à l'aide de la directive `UNDER`.

Tableau 3.34 Héritage de tables SQL3

Schéma logique	Script SQL3
PNC[indice, prime] ↓	CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), CONSTRAINT pk_Personnel PRIMARY KEY(numPers))
Personnel[numPers, nomPers] ↑	CREATE TABLE PNC UNDER Personnel (indice NUMBER, prime NUMBER(8,2))
PNT[brevet, validiteLicence]	CREATE TABLE PNT UNDER Personnel (brevet VARCHAR(10), validiteLicence DATE)

Contraintes

Quelle que soit la contrainte d'une association d'héritage (partition, exclusivité et totalité) à programmer, le schéma SQL3 est identique. En revanche, chaque contrainte devra être programmée soit par des méthodes, soit avec les principes étudiés dans ce chapitre pour SQL2 (déclencheurs, procédures et contraintes de vérification).

Héritage multiple

De même que pour l'héritage simple, la traduction de l'héritage multiple sous SQL3 se programme à l'aide de la directive **UNDER**.

Le schéma SQL3 implémentant l'exemple 2-35 est composé de quatre types et quatre tables objet-relationnelles. Le type `Stagiaire_type` hérite simultanément des types `PNT_type` et `PNC_type`.

Tableau 3.35 Héritage multiple de types SQL3

Schéma logique	Script SQL3
PNC[indice, prime] ↓	CREATE TYPE Personnel_type (numPers NUMBER, nomPers VARCHAR(20))
Personnel[numPers, nomPers] ↑	CREATE TYPE PNC_type UNDER Personnel_type (indice NUMBER, prime NUMBER(8,2))
PNT[brevet, validiteLicence] ↑	CREATE TYPE PNT_type UNDER Personnel_type (brevet VARCHAR(10), validiteLicence DATE)
Stagiaire[dateStage] ↑	CREATE TYPE Stagiaire_type UNDER PNC_type, PNT_type (dateStage DATE)
	CREATE TABLE Personnel OF Personnel_type (CONSTRAINT pk_Personnel PRIMARY KEY(numPers))
	CREATE TABLE PNC OF PNC_type
	CREATE TABLE PNT OF PNT_type
	CREATE TABLE Stagiaire OF Stagiaire_type

Héritage de tables

La clause UNDER concerne ici deux tables.

Tableau 3.36 Héritage multiple de tables SQL3

Schéma logique	Script SQL3
PNC[indice, prime]	CREATE TABLE Personnel (numPers NUMBER, nomPers VARCHAR(20), CONSTRAINT pk_Personnel PRIMARY KEY(numPers))
Personnel[numPers, nomPers]	CREATE TABLE PNC UNDER Personnel (indice NUMBER, prime NUMBER(8,2))
PNT[brevet, valideLicence]	CREATE TABLE PNT UNDER Personnel (brevet VARCHAR(10), valideLicence DATE)
Stagiaire[dateStage]	CREATE TABLE Stagiaire_type UNDER PNC, PNT (dateStage DATE)



Oracle comme IBM (et bien d’autres langages objet en commençant par Java) n’autorisent pas la programmation de l’héritage multiple.

Exercices

Exercice 3.1 Du logique à SQL2

Ecrire le script SQL2 (création des tables) du schéma relationnel extrait de l'exercice 2.4. Considérez les types de colonnes suivants :

- *a* et *g* CHAR(12)
- *b*, *h* et *i* NUMBER
- *e*, *f* CHAR(40)
- *c*, *d* et *j* DATE

Dans quel ordre ces créations doivent-elles être effectuées ?

Exercice 3.2 Du logique à SQL2

Traduisez les modèles logiques de l'exercice 2.9 (affrètement d'avions, Castanet Telecoms et voltige aérienne) en tables SQL2. Considérez les types de données suivants.

1. Affrètement d'avions

Ajouter les contraintes qui assurent que la capacité d'un avion et le nombre de passagers transportés sont compris entre 50 et 500 places.

attribut	type
comp	CHAR(8)
nomComp	CHAR(30)
typeAvion	CHAR(10)
npMax	NUMBER
nomAvion	CHAR(30)
immatriculation	CHAR(6)
capacite	NUMBER
dateVol	DATE
nbPax	NUMBER
cout	NUMBER

2. Castanet Telecoms

Ne traduisez pas la relation Privileges en une table relationnelle.

attribut	type
formule	CHAR(8)
prixParMois	NUMBER

attribut (<i>suite</i>)	type (<i>suite</i>)
heureSupp	NUMBER
numAbonne	CHAR(10)
nomAbonne	CHAR(30)
adresseAbonne	CHAR(40)
numTel	CHAR(14)
consoLocale	NUMBER
consoNationale	NUMBER
consoInternet	NUMBER
consoAutres	NUMBER
debitLigne	NUMBER
consoMobiles	NUMBER
numPortable	CHAR(14)
typeCombine	CHAR(20)
URL	CHAR(200)
adresseFourni	CHAR(40)
responsable	CHAR(30)
premierContact	DATE
finContrat	DATE
telPrefere	CHAR(14)
dureePreferee	NUMBER

3. Voltige aérienne

Ne traduisez pas la relation Ordre en une table relationnelle.

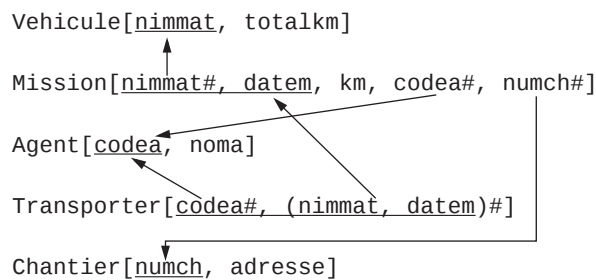
attribut	type
nclubVoltige	NUMBER
aerodromeBase	CHAR(40)
nomClub	CHAR(30)
ncomp	CHAR(8)
nomComp	CHAR(30)
pays	CHAR(20)
immatriculation	CHAR(6)
typeAvion	CHAR(20)
ninventeur	NUMBER
nomInventeur	CHAR(30)
typePilote	CHAR(5)
nfigure	NUMBER
nomFigure	CHAR(40)

attribut (suite)	type (suite)
noteMax	NUMBER
dateCreation	DATE
pageManuel	NUMBER
numeroChrono	NUMBER
dateVol	DATE
typeProgramme	CHAR(5)
note	NUMBER

Exercice 3.3 Associations d'agrégation

Traduisez avec SQL2 le schéma relationnel des associations d'agrégation de l'exercice 2.10. Considérez les types de données suivants :

attribut	type
nimmat	CHAR(10)
totalkm	NUMBER
datem	DATE
codea	CHAR(5)
noma	CHAR(40)
numch	NUMBER
adresse	CHAR(50)



Chapitre 4

Outils du marché : de la théorie à la pratique

Non mais t'as déjà vu ça ? En pleine paix. Y chante et puis crac, un bourre-pif ! Mais il est complètement fou ce mec ! Mais moi, les dingues j'les soigne. J'm'en vais lui faire une ordonnance, et une sévère ! J'vais lui montrer qui c'est Raoul. Aux quatre coins de Paris qu'on va le retrouver, éparpillé par petits bouts, façon puzzle. Moi quand on m'en fait trop, j'correctionne plus, j'dynamite, j'disperse, j'ventile...

Les Tontons flingueurs, B. Blier
G. Lautner, dialogues M. Audiard, 1963

Ce chapitre valide la démarche théorique de l'ouvrage en la comparant aux principales solutions informatiques du marché qui mettent en œuvre la notation UML et l'interconnexion à une base de données (le plus souvent par un pilote ODBC ou JDBC). Les 14 outils étudiés sont : Enterprise Architect, MagicDraw, MEGA Designer, ModelSphere, MyEclipse, Objecteering, Poseidon for UML, PowerAMC, Rational Rose Data Modeler, Together, Visio, Visual Paradigm, Visual UML et Win'Design. Le lecteur trouvera en annexe les adresses Internet de ces produits.

Sont exclus de ce comparatif, les outils qui ne prennent pas en compte UML pour l'instant, citons DB Designer, Database Design Studio, DeZign, AllFusion ERWin, xCase, CASE Studio et ER/Studio.

La partie consacrée aux bases de données n'est pas prépondérante pour la majorité des outils. D'autres fonctionnalités sont offertes en ce qui concerne la modélisation de processus métier BPM (*Business Process Models*) qui peuvent être importés ou exportés conformément au langage BPEL4WS (*Business Process Execution Language for Web Services*). Bon nombre d'entre eux fournissent un référentiel pour le contrôle des données métiers utiles aux architectes des systèmes d'information qui en seront les principaux utilisateurs. Les plus récents s'inscrivent davantage dans l'architecture MDA (*Model Driven Architecture*) en incluant le standard QVT (*Query View Transformation*) pour la transformation de modèles.

La majorité des outils proposent un processus de conception basé sur les différents niveaux du conceptuel au physique (*forward engineering*), un processus de rétroconception (*reverse engineering*) et un processus d'interéchange (*round-trip engineering*) entre chaque modèle. Les outils permettent également la génération de code en différents langages (SQL, PowerBuilder, C++, C#, Java, XML, IDL-CORBA, Visual Basic, IHM ou site Web).

Seuls Together et MyEclipse sont totalement intégrés à Eclipse (Objectteering et MagicDraw proposent toutefois un plug-in).

Chaque logiciel sera évalué selon la qualité d'implémentation de différents diagrammes de classes incluant les critères suivants :

- associations binaires et *n*-aires, classes-associations et agrégations ;
- contraintes (partition, inclusion et relatives à l'héritage) ;
- héritage (décomposition au niveau logique et héritage multiple) ;
- rétroconception d'une base de données.

Pour chaque critère, un tableau illustre la mise en œuvre des outils avec une signalisation à 248248010 états (satisfaisante : feu vert, moyenne : feu orange, insatisfaisante et absente : feu rouge). Certaines particularités intéressantes seront illustrées par des copies d'écran au fil des exemples. Le classement final, qui prend également en compte la robustesse et l'ergonomie de chaque logiciel, n'engage que moi naturellement.

Associations binaires

Les outils sont évalués ici sur la capacité à représenter, avec la notation UML, des associations binaires, de les transformer au niveau logique puis de générer un script SQL. La prise en compte des identifiants de classes, multiplicités, rôles, etc. est également étudiée.

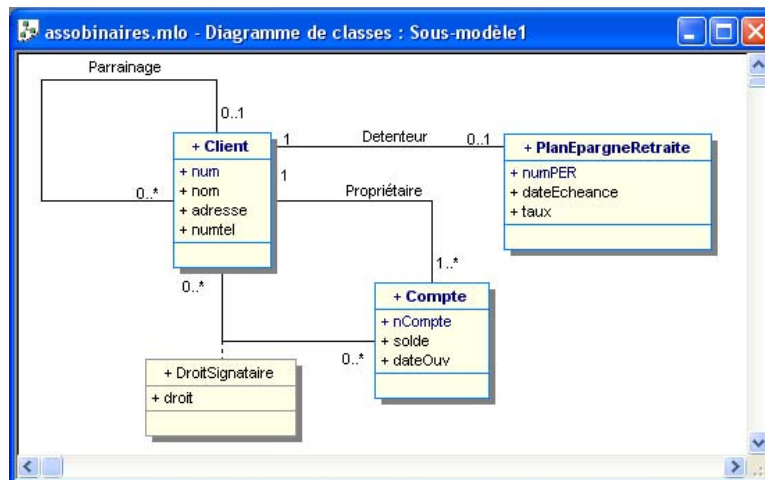


Figure 4-1 Associations binaires
(Win'Design)

L'exemple 4-1 décrit une modélisation UML de comptes bancaires qui sont la propriété de clients. Un client peut parrainer d'autres clients. Un client n'a le droit de souscrire qu'à un seul plan d'épargne retraite. Enfin, un client peut être désigné comme signataire de comptes ne lui appartenant pas (un seul droit lui est alors affecté).

Niveau conceptuel

Les outils sont évalués sur les moyens mis en œuvre pour représenter une classe (avec ses attributs et son identifiant), une association binaire ou réflexive (nommage, désignation des rôles et des multiplicités).

Les copies d'écran suivantes illustrent l'implémentation des identifiants de classe selon Objecteering, MEGA, PowerAMC et Rational Rose (Win'Design le permet également).

Figure 4-2 Définition d'un identifiant de classe (Objecteering)

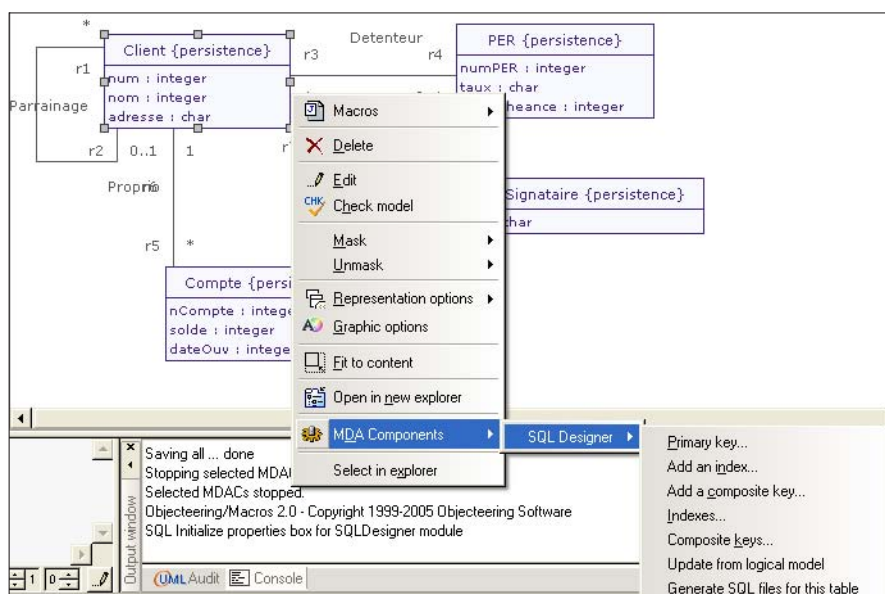


Figure 4-3 Définition d'un identifiant de classe (MEGA)

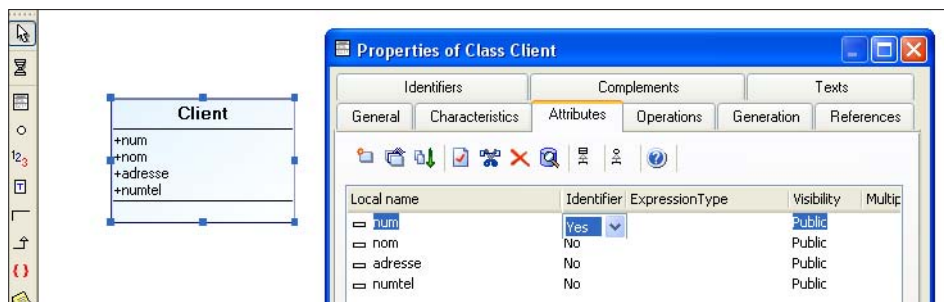


Figure 4-4 Définition d'un identifiant de classe (PowerAMC)

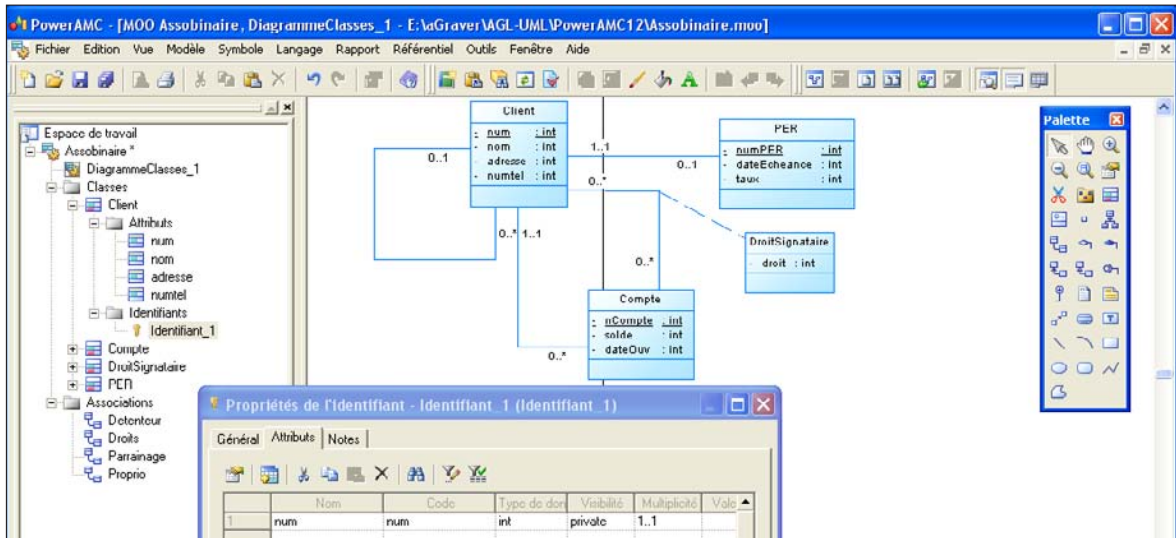
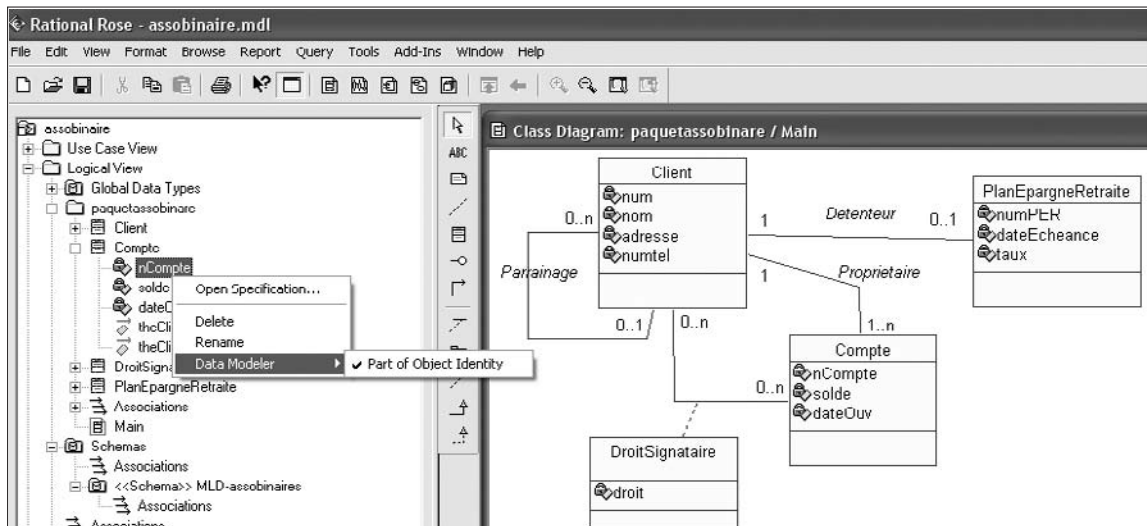


Figure 4-5 Définition d'un identifiant de classe (Rational Rose)



Niveau logique

Les outils sont évalués sur leur capacité à générer un schéma relationnel correct et sur la possibilité de définir d'éventuelles contraintes de répercussion sur les clés étrangères (CASCADE) pour la suite de processus de conception.

Le modèle relationnel attendu est le suivant.

Figure 4-6 Modèle relationnel attendu

Client[num, nom, adresse, numtel, numParrain#]

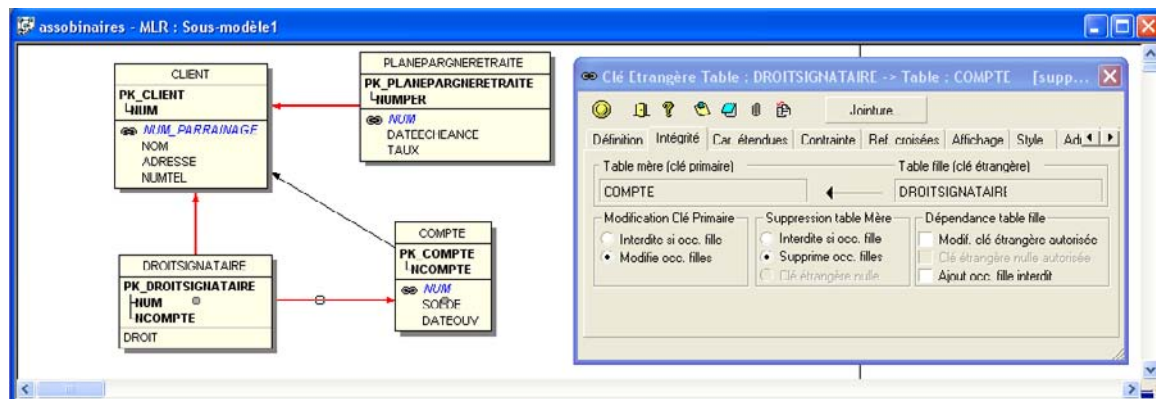
PlanEpargneRetraite[numPER, dateEcheance, taux, num#]

DroitSignataire[num#, nCompte#, droit]

Compte[nCompte, solde, dateOuv, num#]

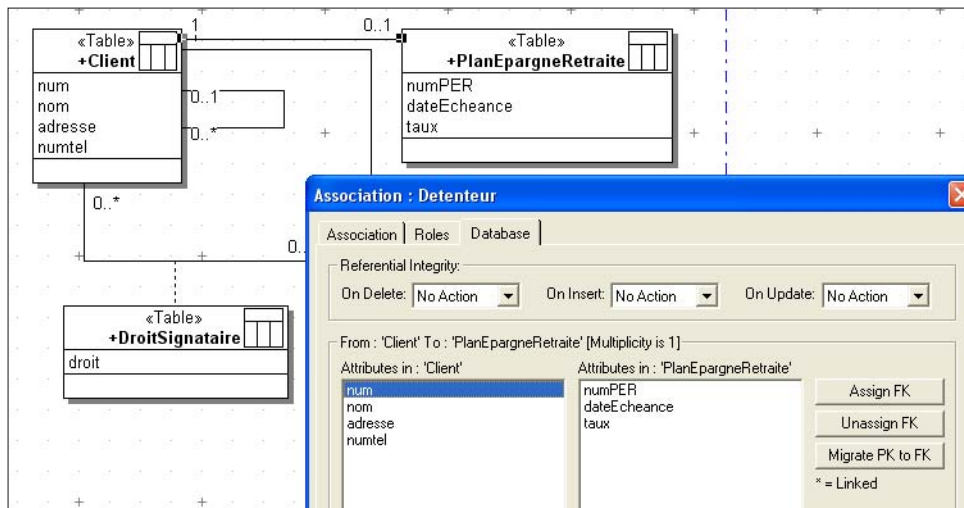
Bon nombre d'outils permettent d'ajouter des contraintes au niveau des clés étrangères. Win'Design note en rouge les liens qui symbolisent la répercussion d'une suppression (ici la suppression d'un compte devrait entraîner celle des lignes dans la table des signataires, de même pour la suppression d'un client).

Figure 4-7 Modèle logique Win'Design



Un mauvais point pour Visual UML qui nécessite (comme JDeveloper d'Oracle) de définir manuellement les clés étrangères pour chaque association ! Dans le cas de l'association présente, il faudrait ajouter un attribut dans la classe PlanEpargneRetraite puis le lier à l'identifiant de la classe Client.

Figure 4-8 Définition d'une clé étrangère avec Visual UML



Script SQL

Les scripts SQL générés sont évalués sur la dénomination des contraintes (utilisation des noms des associations ou des rôles) et la position de la clé étrangère de l'association *un-à-un* (qui devrait, en théorie, se situer dans la table *PlanEpargneRetraite*).

Bilan intermédiaire

La majorité des outils permettent une saisie correcte au niveau conceptuel. ModelSphere et MyEclipse pêchent par manque de fonctionnalités tandis que Visual Paradigm nécessite une manipulation complexe inter-modèles. Au niveau logique, des différences d'implémentation (ou des insuffisances) apparaissent. Seuls Win'Design, Rational Rose et Objectteering font un sans faute. Un bémol pour MagicDraw, Visual Paradigm, Together et Enterprise Architect qui ne disposent pas du mécanisme d'identifiant de classe. À noter que Together représente une classe-association avec le même symbole que celui de l'association *n-aire*.

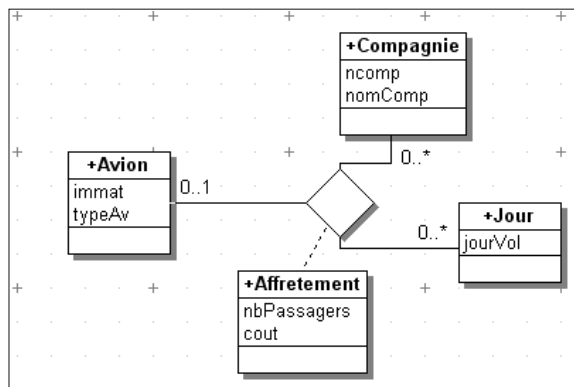
Tableau 4.1 Associations binaires

Associations binaires	Niveau conceptuel avec UML (identifiant, associations, multiplicités, rôles, etc.)	Modèle logique	Code SQL (contraintes de clés étrangères)
Enterprise Architect			
MagicDraw			
MEGA Designer			
ModelSphere			
MyEclipse			
Objectteering			
Poseidon			
PowerAMC			
Rational Rose Data Modeler			
Together			
Visio			
Visual Paradigm			
Visual UML			
Win'Design			

Associations *n*-aires

L'exemple 4-9 décrit une association 3-aire (avec attributs) qui modélise les affrètements d'avions par différentes compagnies au cours du temps.

Figure 4-9 Association *n*-aire (Visual UML)



Niveau conceptuel

Les outils sont évalués sur la possibilité de représenter une association *n*-aire à l'aide du symbole losange (*diamond*), de nommer l'association et d'y apposer différentes multiplicités maximales (1 ou plus généralement *).

Figure 4-10 Association *n*-aire (Together)

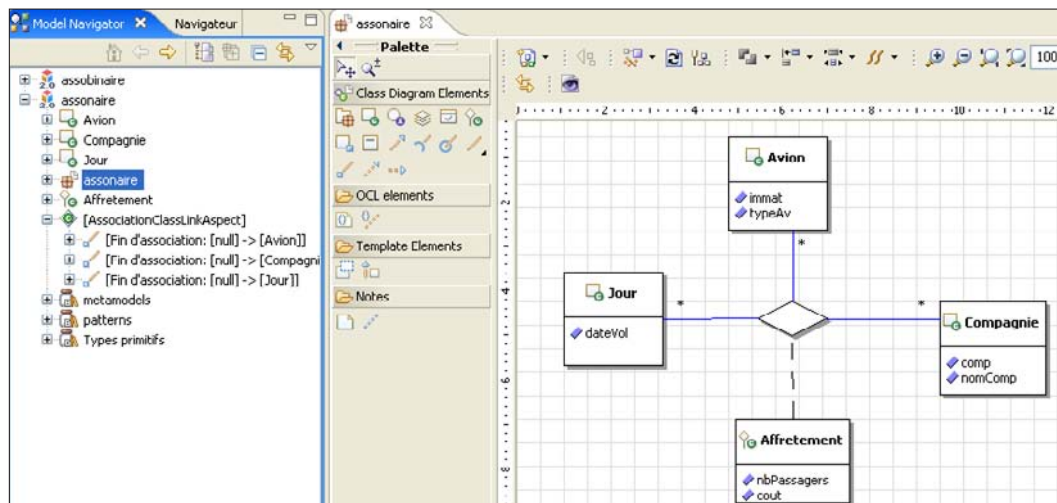
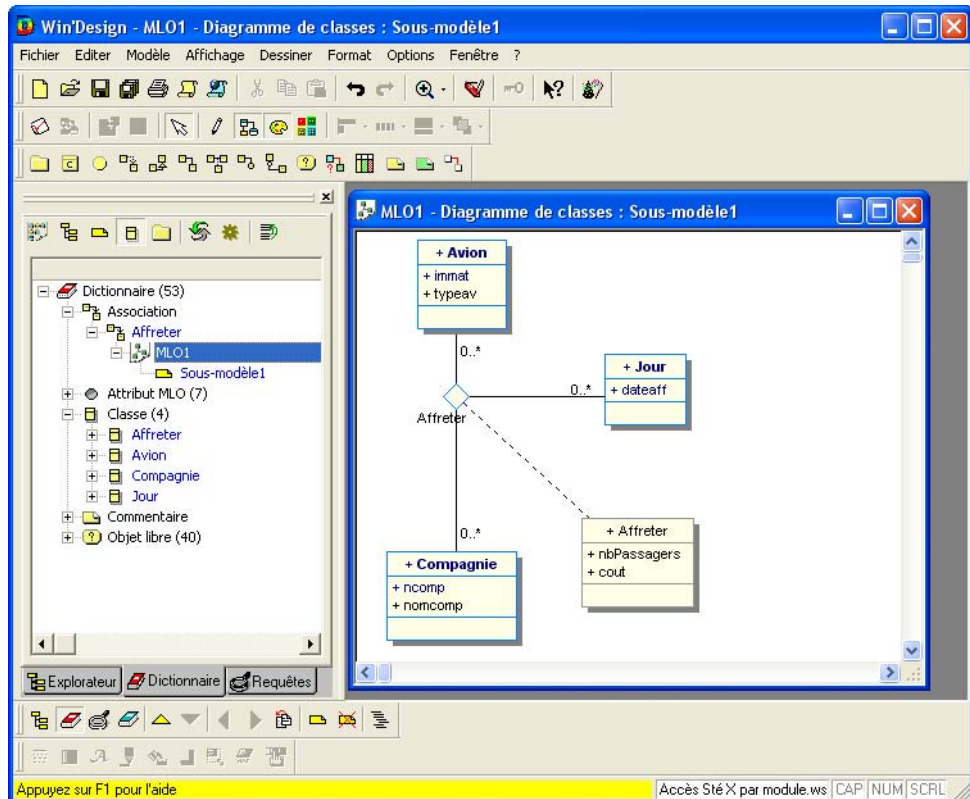


Figure 4-11 Association n-aire (Win'Design)



Niveau logique

Les outils sont évalués sur leur capacité à transformer une association *n*-aire selon la valeur des multiplicités (en particulier pour 0..1 et 1, le degré de la clé primaire de la relation dérivée doit être réduit).

Le modèle relationnel attendu est le suivant.

Figure 4-12 Modèle relationnel attendu

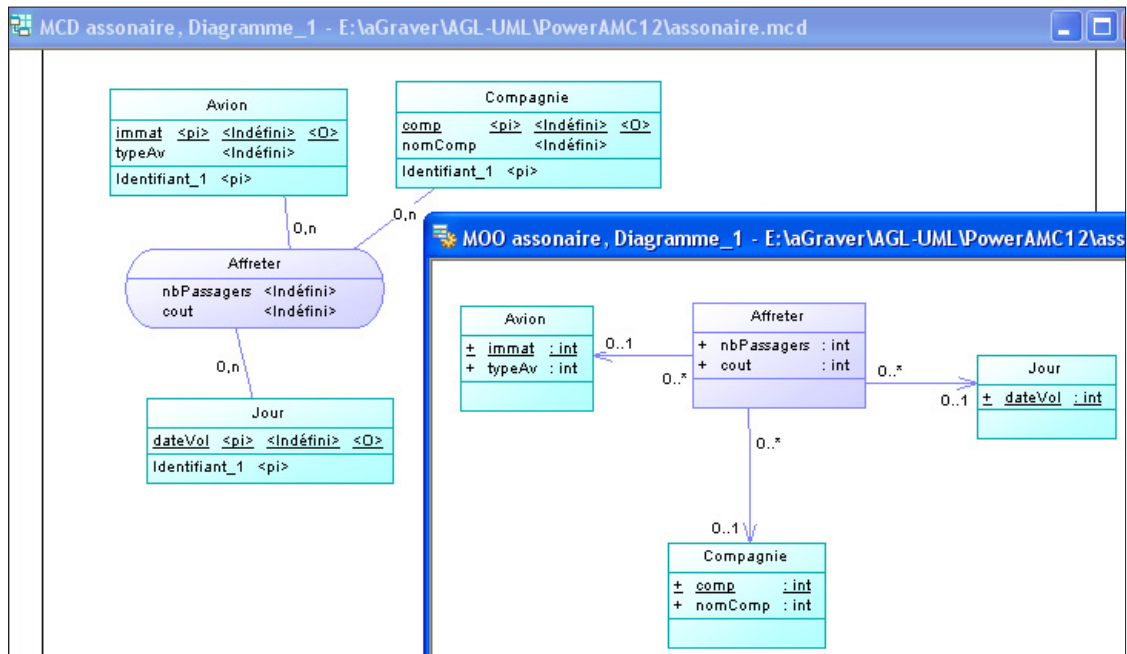
Avion[immat, typeAv]

Compagnie[comp, nomComp]

Affretement[immat#, comp#, dateAff#, nbPassagers, cout]

Jour[dateAff]

Figure 4-13 Association n-aire (Power AMC)



Script SQL

Les outils sont évalués sur la possibilité de ne pas traduire une relation en table tout en la conservant au niveau logique (dans notre exemple, il n'est pas souhaitable de transformer la relation JOUR en une table) et la traduction d'éventuelles multiplicités 0..1 et 1 (clé primaire réduite ou contrainte UNIQUE).

Bilan intermédiaire

Seuls cinq outils sont vraiment à l'aise au niveau conceptuel avec les associations *n*-aire. Le diagramme test ajoutait à la difficulté le fait de connecter une classe-association. Le grand gagnant de cet exercice est Win'Design qui maîtrise le processus de bout en bout.

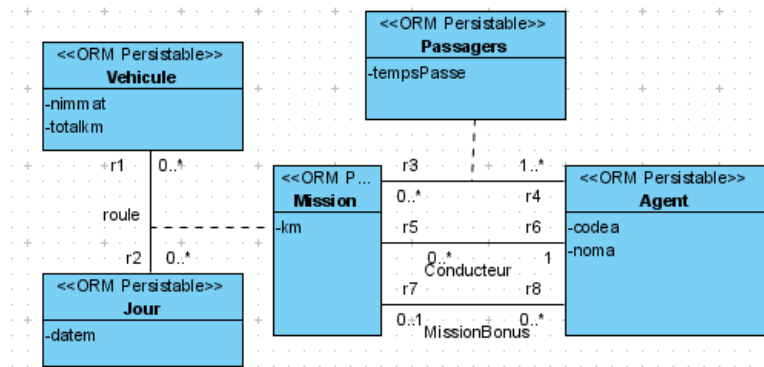
Tableau 4.2 Associations *n*-aires

Associations <i>n</i> -aires	Niveau conceptuel avec UML (symbole losange, multiplicités)	Modèle logique	Code SQL
Enterprise Architect			
MagicDraw			
MEGA Designer			
ModelSphere			
MyEclipse			
Objectteering			
Poseidon			
PowerAMC			
Rational Rose Data Modeler			
Together			
Visio			
Visual Paradigm			
Visual UML			
Win'Design			

Classes-associations

L'exemple 4-14 regroupe sur une modélisation plusieurs associations sur une classe-association (compilation des exemples 2-52, 2-56 et 2-59). Différentes missions peuvent être suivies par un employé (en tant que passager ou conducteur). De plus, un employé pourra choisir une mission particulière pour calculer un bonus. Il est à noter que cette même base de données nous servira de test pour le processus de rétroconception.

Figure 4-14 Classes-associations
(Visual Paradigm)



Niveau conceptuel

Les outils sont évalués sur la capacité de représenter toutes les multiplicités et plusieurs liens sur la même classe-association.

Figure 4-15 Classes-associations (Together)

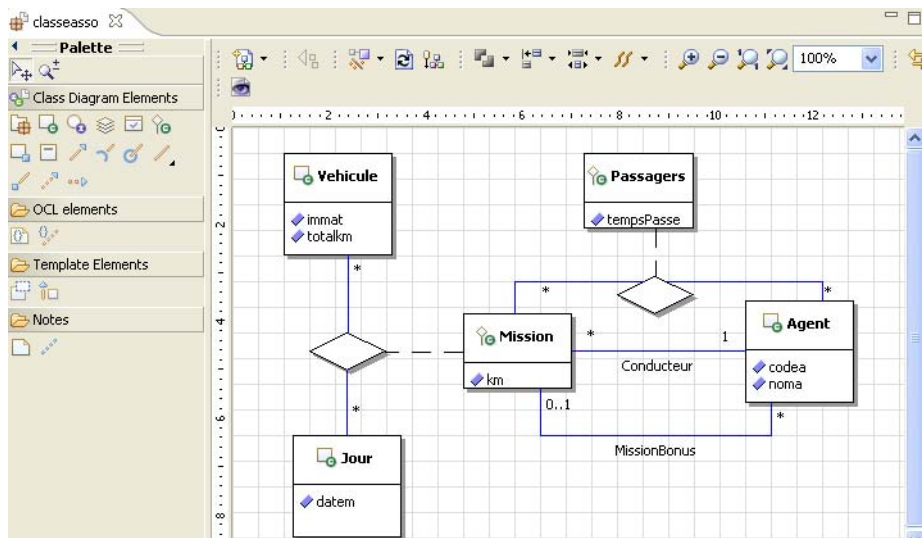
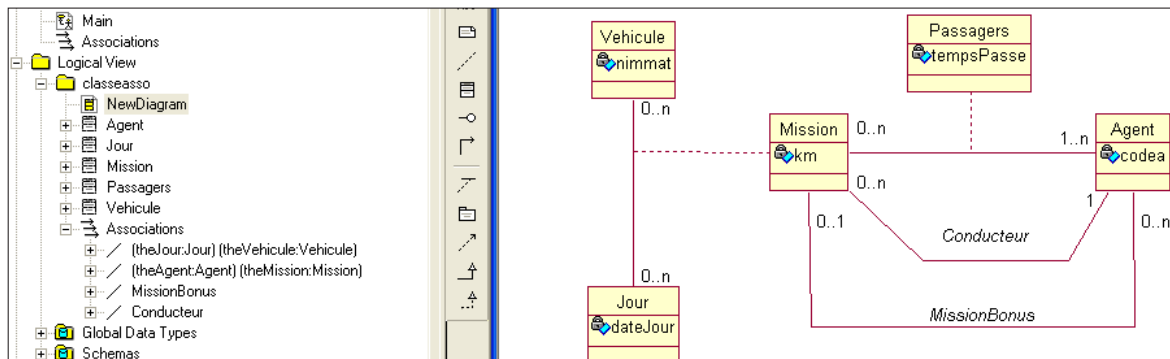


Figure 4-16 Classes-associations (Rational Rose)

Niveau logique

Les outils sont évalués sur la capacité de générer correctement un schéma relationnel en fonction des multiplicités des classes-associations.

Le modèle relationnel attendu est le suivant.

Figure 4-17 Modèle relationnel attendu

Vehicule[nimmat, totalkm]

Agent[codea, noma, (nimmat, dateJour)#]

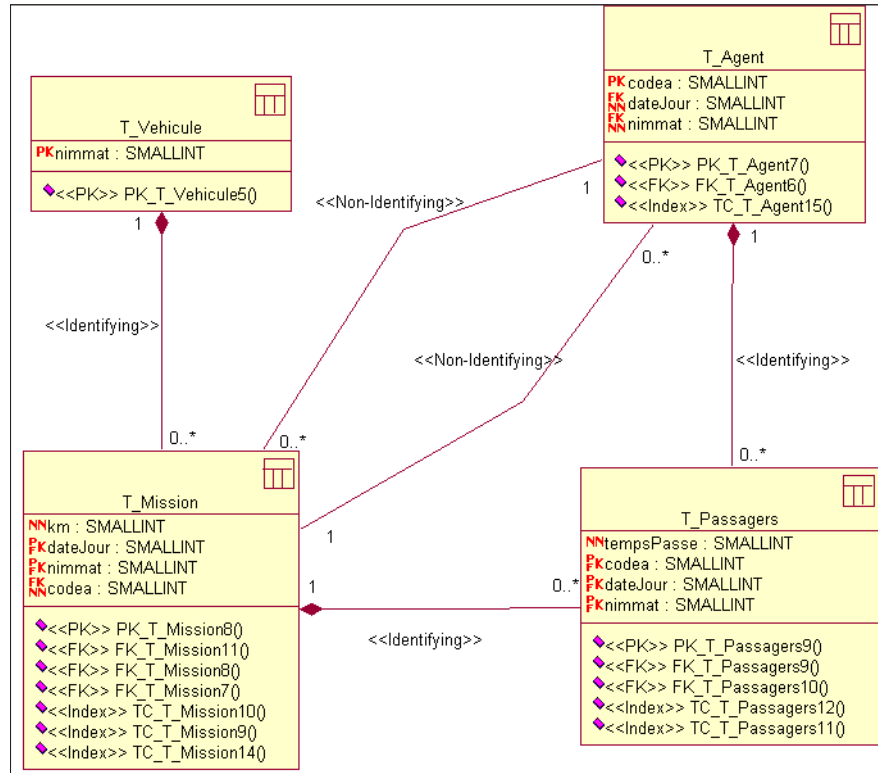
Mission[nimmat#, dateJour#, km, codea#]

Jour[dateJour]

Passagers[(nimmat, dateJour)#, codea#, tempsPasse]

Le modèle logique de Rational Rose est décrit à l'aide du profil UML.

Figure 4-18 Classes-associations (Rational Rose)



Script SQL

Les outils sont évalués sur la dénomination des clés étrangères (on devrait pouvoir retrouver le nom de certaines associations liées à la classe-association).

Bilan intermédiaire

C'est au niveau logique, durant la phase de migration des clés étrangères, que les disparités se produisent. Seuls trois outils passent ce test haut la main, il s'agit de Objecteering, Rational Rose et PowerAMC.

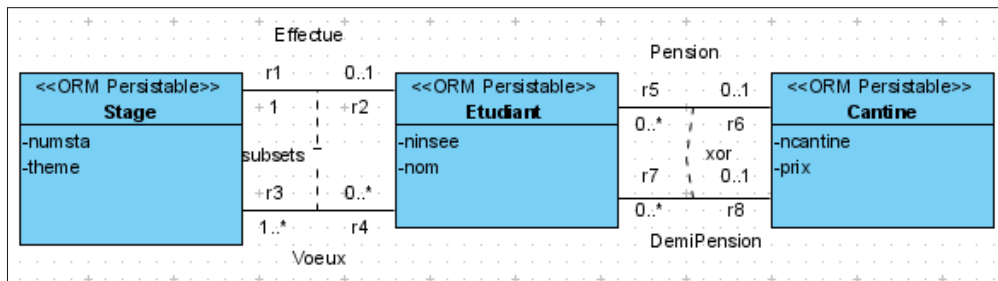
Tableau 4.3 Classes-associations

2	Niveau conceptuel avec UML (plusieurs liens sur la classe-association)	Modèle logique	Code SQL
Enterprise Architect			
MagicDraw			
MEGA Designer			
ModelSphere			
MyEclipse			
Objectteering			
Poseidon			
PowerAMC			
Rational Rose Data Modeler			
Together			
Visio			
Visual Paradigm			
Visual UML			
Win'Design			

Contraintes

L'exemple 4-19 illustre deux contraintes prédéfinies de UML 2 (partition : xor et inclusion : subsets). La partition exprime qu'un étudiant est soit pensionnaire soit demi-pensionnaire, l'inclusion signifie que le stage d'un étudiant doit être au préalable souhaité.

Figure 4-19 Contraintes (Visual Paradigm)



Niveau conceptuel

Les outils sont évalués sur la capacité de représenter ces contraintes (stéréotypes prédéfinis, menu contextuel, etc.). Dans la majorité des cas, la contrainte doit être saisie manuellement ou le stéréotype doit être créé pour pouvoir être réutilisé par la suite. Peu d'outils proposent des stéréotypes prédéfinis comme le montrent les écrans suivants.

Figure 4-20 Contraintes (Objectteering)

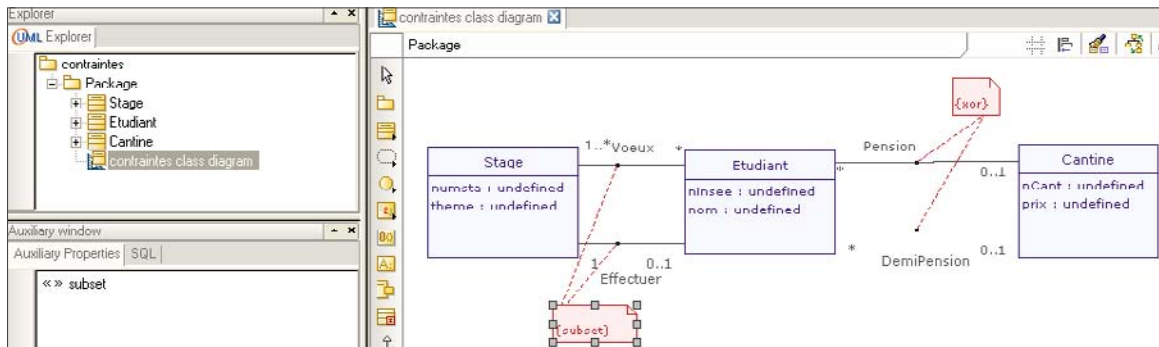
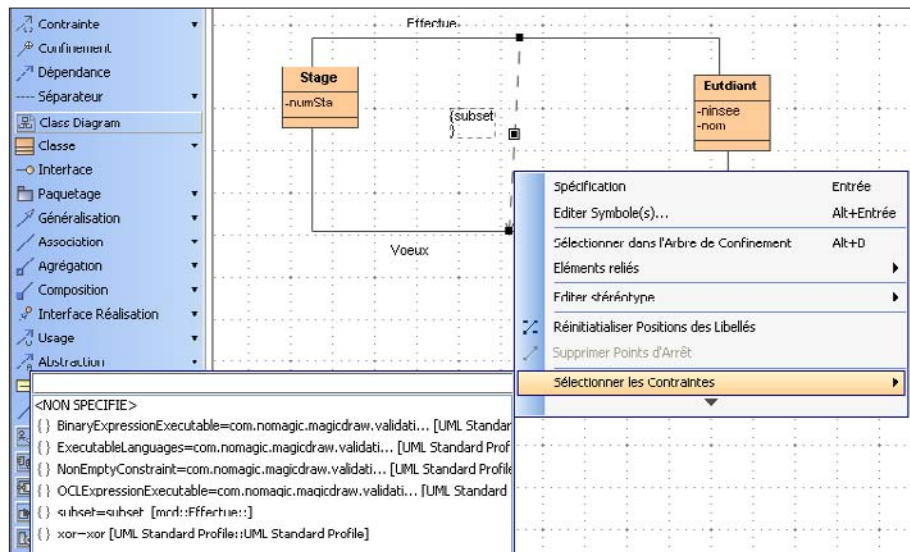


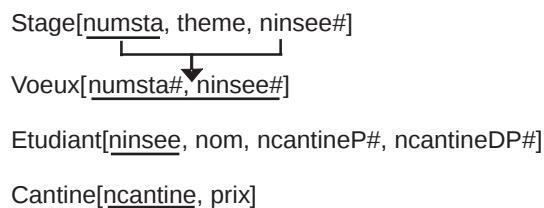
Figure 4-21 Contraintes (MagicDraw)



Niveau logique

Le modèle relationnel attendu est représenté à la figure 4-22. Ces deux contraintes n'ont pas d'influence sur la structure des relations générées (idéalement, une clé étrangère devrait être toutefois ajoutée entre **Stage** et **Voeux** pour assurer l'inclusion).

Figure 4-22 Modèle relationnel attendu



Script SQL

Les outils sont évalués sur la capacité à générer la clé étrangère induite de la contrainte d'inclusion et la directive SQL de vérification (CHECK) pour implémenter la contrainte de partition (non-nullité exclusive des clés étrangères liant **Etudiant** à **Cantine**). Cette dernière contrainte pourrait aussi être assurée par un déclencheur (qui pourrait être déclaré automatiquement mais qu'il faudrait par la suite programmer manuellement).

Bilan intermédiaire

Tableau 4.4 Contraintes

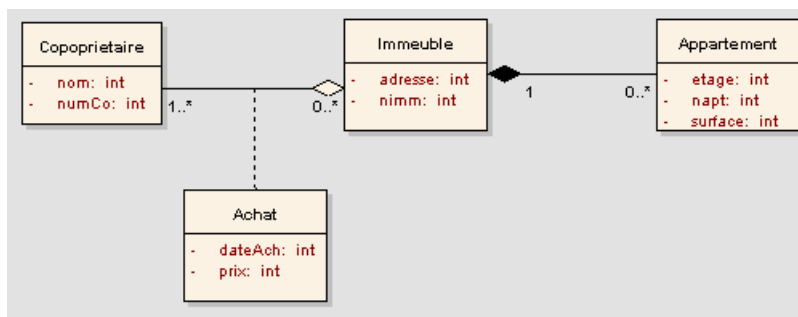
Contraintes	Niveau conceptuel avec UML (partition et inclusion)	Code SQL (clé étrangère, contrainte CHECK ou déclencheur)
Enterprise Architect		
MagicDraw		
MEGA Designer		
ModelSphere		
MyEclipse		
Objectteering		
Poseidon		
PowerAMC		
Rational Rose Data Modeler		
Together		
Visio		
Visual Paradigm		
Visual UML		
Win'Design		

Bien que certaines contraintes prédéfinies de UML 2 soient déjà répertoriées par certains outils, la traduction du conceptuel au physique n'est pas encore automatisée. Il vous incombera donc de programmer explicitement au niveau du code SQL l'enrichissement sémantique de vos diagrammes de classes.

Agrégations

L'exemple 4-23 met en œuvre une composition (qui exprime qu'un appartement est un composant d'un immeuble) et une agrégation partagée (qui renforce l'association d'achat entre un copropriétaire et un immeuble).

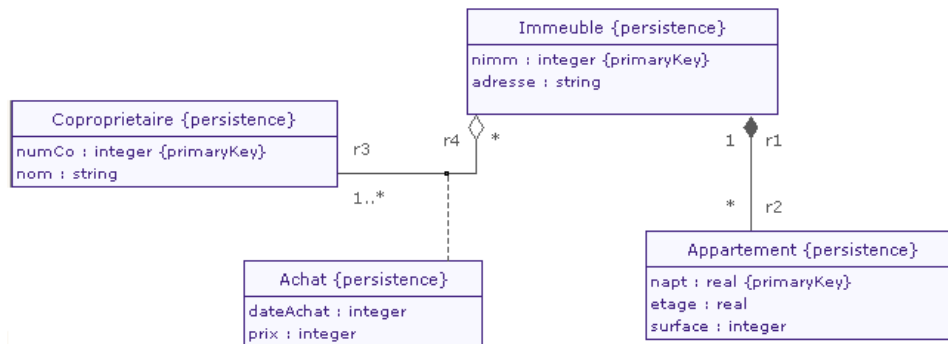
Figure 4-23 Agrégations (Enterprise Architect)



Niveau conceptuel

Les outils sont évalués sur la capacité de représenter ces deux formes d'agrégation.

Figure 4-24 Agrégations (Objectteering)



Niveau logique

Les outils sont évalués sur la capacité de générer un schéma relationnel prenant en compte la composition des tables (clé primaire de la table composant enrichie par la clé primaire de la table composite). Le modèle relationnel attendu est le suivant.

Coproprietaire[numco, nom]

Immeuble[nimm, adresse]

Achat[nimm#, numco#, prix, dateAchat]

Appartement[nimm#, napt, surface, etage]

Figure 4-25 Modèle relationnel attendu

Les captures d'écran suivantes illustrent un tel modèle logique.

Figure 4-26 Agrégations (Win'Design)

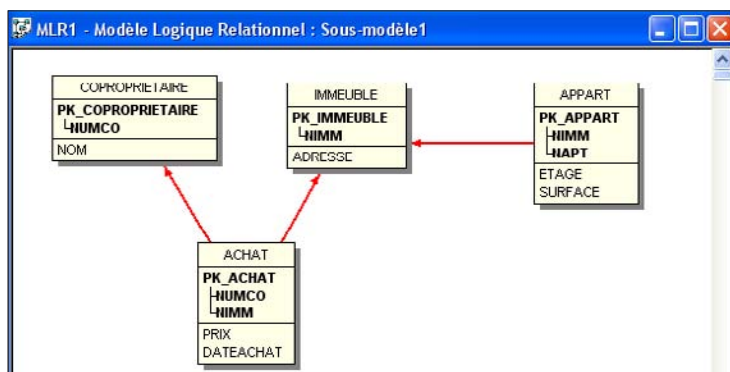
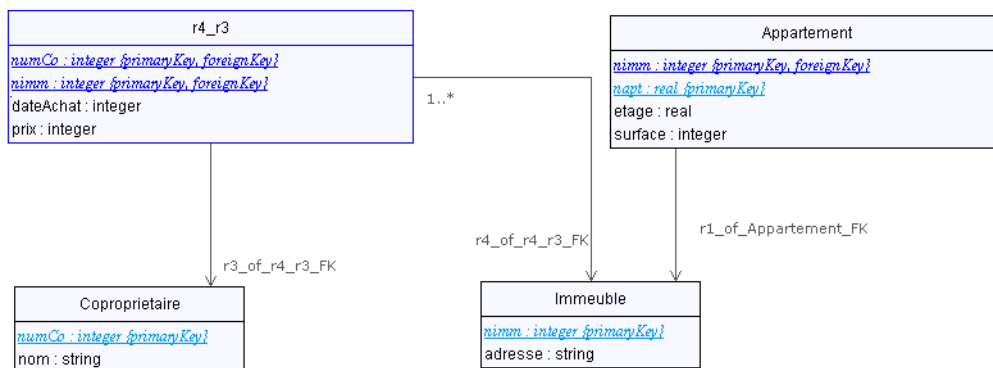


Figure 4-27 Agrégations (Objecteering)



Script SQL

Les scripts sont évalués sur la capacité d'inclure la directive CASCADE au niveau de la clé étrangère de la table composite et de celle liée à l'agrégation partagée (ici Achat et Appartement).

Bilan intermédiaire

Tableau 4.5 Agrégations

Agrégation	Niveau conceptuel avec UML (symboles de l'agrégation partagée et de la composition)	Modèle logique (clé composite)	Code SQL (contraintes CASCADE)
Enterprise Architect			
MagicDraw			
MEGA Designer			
ModelSphere			
MyEclipse			
Objectteering			
Poseidon			
PowerAMC			
Rational Rose Data Modeler			
Together			
Visio			
Visual Paradigm			

Tableau 4.5 Agrégations (suite)

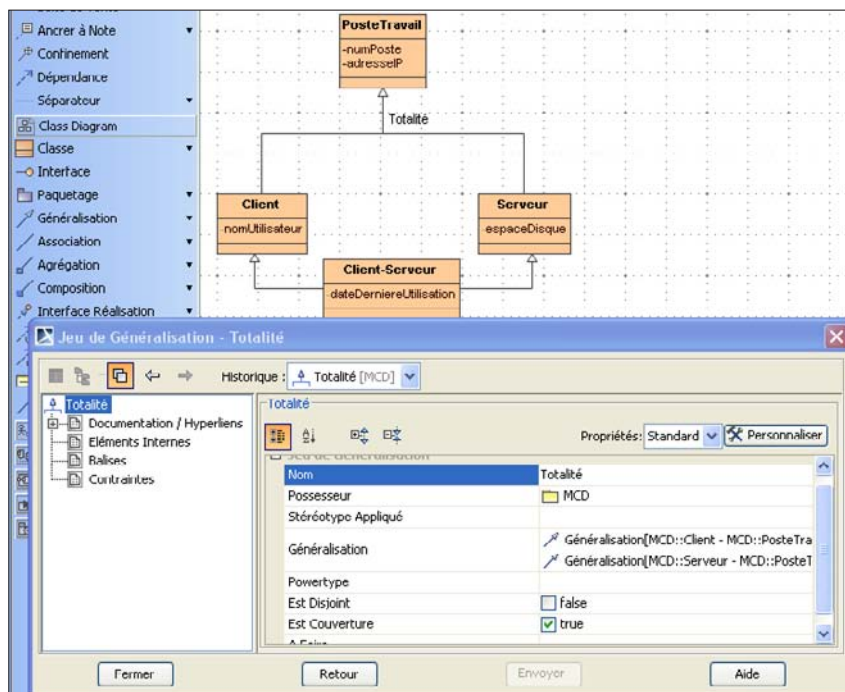
Agrégation	Niveau conceptuel avec UML (symboles de l'agrégation partagée et de la composition)	Modèle logique (clé composite)	Code SQL (contraintes CASCADE)
Visual UML			
Win'Design			

C'est encore une fois au niveau logique, durant la phase de transformation des agrégations (dont une couplée à une classe-association), que les disparités apparaissent. Seuls trois outils réussissent à construire un modèle logique correct : il s'agit de Objecteering, PowerAMC et Win'Design. Rational Rose et Visio ne proposent qu'une seule forme d'agrégation. En revanche, aucun outil ne génère de directive CASCADE au niveau du code SQL.

Héritage

L'exemple 4-28 décrit une hiérarchie à deux niveaux avec un héritage multiple. On devra pouvoir définir la contrainte de totalité au premier niveau.

Figure 4-28 Héritage (Magic Draw)



Niveau conceptuel

Les outils sont évalués sur la capacité de représenter d'une part la contrainte de totalité d'autre part l'héritage multiple.

Figure 4-29 Héritage (MEGA)

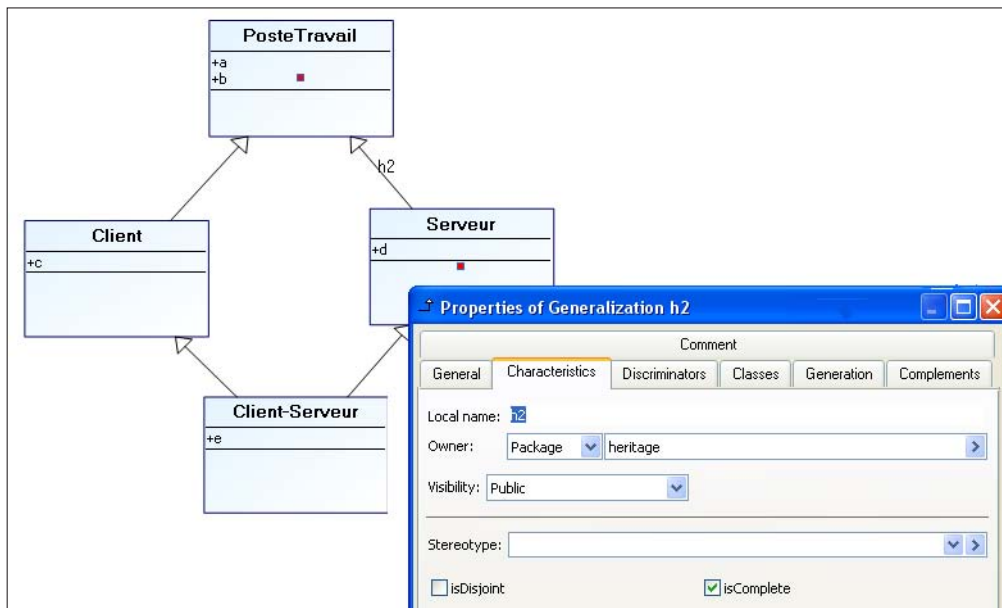
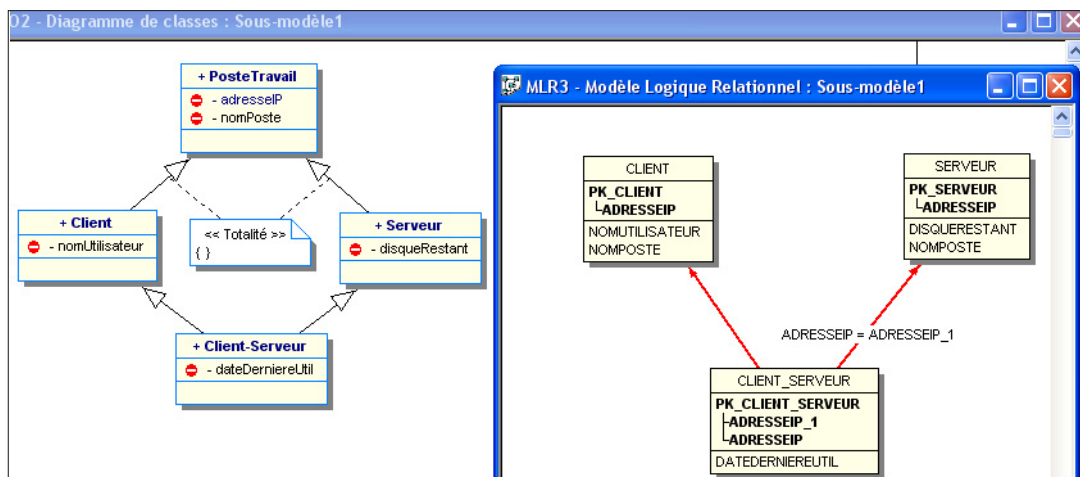


Figure 4-30 Héritage (Win'Design)



Niveau logique

Les outils sont évalués sur la capacité de proposer les trois cas de décompositions pour chaque niveau du graphe d'héritage. On devrait pouvoir choisir, par exemple, de définir une contrainte de totalité au premier niveau, de transformer par *push-down* le premier niveau et par distinction le second niveau.

Le modèle relationnel attendu est le suivant.

Figure 4-31 Modèle relationnel attendu

Client[adresseIP, nomPoste, nomUtilisateur]

Serveur[adresseIP, nomPoste, disqueRestant]

Client-Serveur[adresseIPC#, adresseIPS#, dateDerniereUtil]

Les captures d'écran suivantes illustrent les menus relatifs aux choix de décomposition du premier niveau du graphe d'héritage.

Figure 4-32 Héritage (PowerAMC)

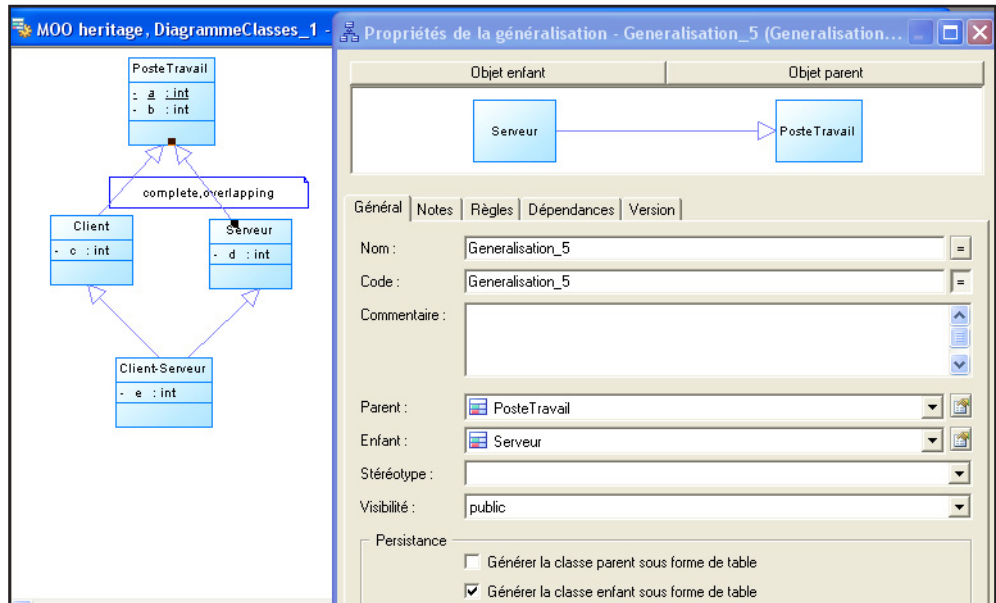


Figure 4-33 Décomposition avec Visual Paradigm

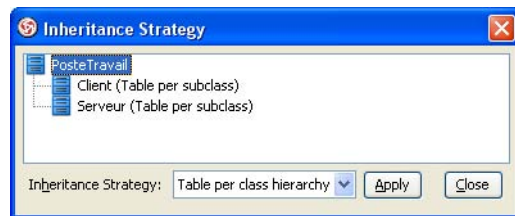


Figure 4-34 Décomposition avec Win'Design

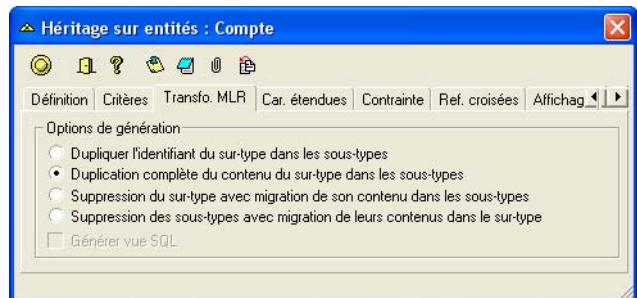
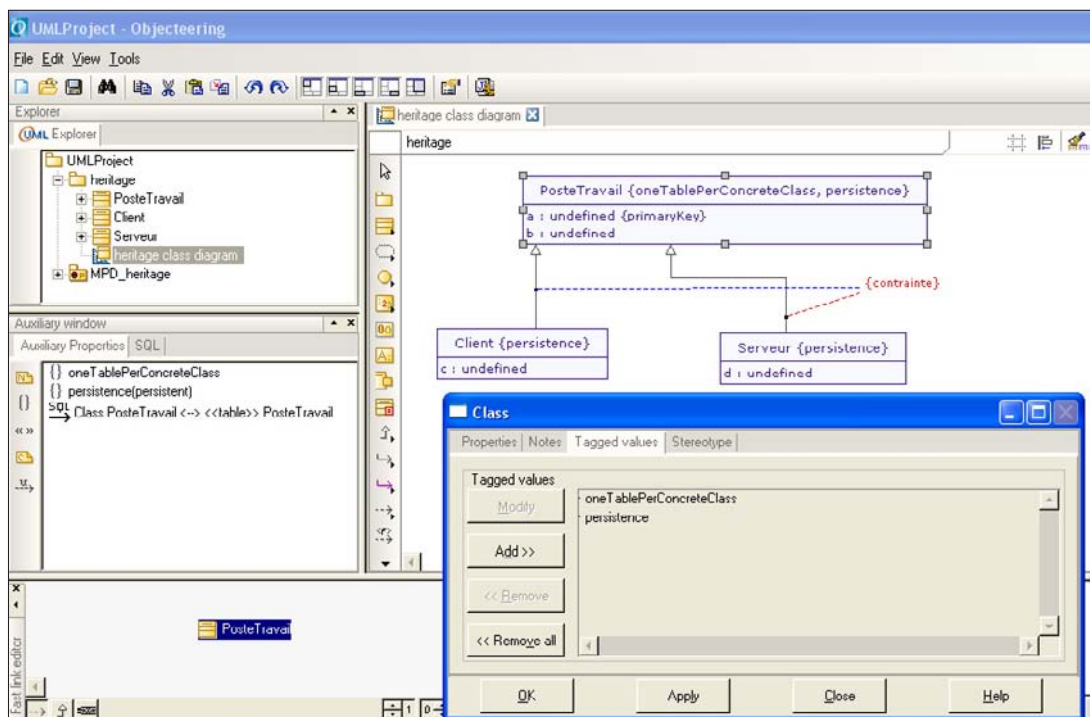


Figure 4-35 Décomposition avec Objectteering



Script SQL

Aucun outil ne génère actuellement du code SQL implémentant les contraintes d'héritage (idéalement, il faudrait générer soit un déclencheur, soit des directives CHECK en fonction du type de la contrainte).

Bilan intermédiaire

Tableau 4.6 Héritage

Héritage	Niveau conceptuel avec UML (contrainte <i>complete-disjoint-incomplete-overlapping</i> et héritage multiple)	Modèle logique (3 cas de décomposition)
Enterprise Architect		
MagicDraw		
MEGA Designer		
ModelSphere		
MyEclipse		
Objectteering		
Poseidon		
PowerAMC		
Rational Rose Data Modeler		
Together		
Visio		

Tableau 4.6 Héritage (suite)

Héritage	Niveau conceptuel avec UML (contrainte <i>complete-disjoint-incomplete-overlapping</i> et héritage multiple)	Modèle logique (3 cas de décomposition)
Visual Paradigm		
Visual UML		
Win'Design		

Seuls Magic Draw et MEGA maîtrisent les contraintes prédéfinies de UML 2 relatives à l'héritage. Seuls PowerAMC et Win'Design (en passant par un MCD Merise) construisent un modèle logique correct en proposant les trois types de transformation. Visual Paradigm et Objectteering ne proposent que deux cas de décomposition.

La rétroconception

Les outils sont évalués sur la capacité à générer un diagramme UML à partir de la base de données MySQL suivante. Le processus devrait produire, en théorie, un diagramme contenant trois associations sur une classe-association voir figure 4-14).

```
CREATE TABLE vehicule (immat CHAR(10), genre CHAR(20),
    kilometrage INTEGER, CONSTRAINT pk_veh1 PRIMARY KEY(immat));
CREATE TABLE agent (codea CHAR(10), noma CHAR(20), immat CHAR(10),
    jour DATETIME, CONSTRAINT pk_agent PRIMARY KEY(codea));
CREATE TABLE mission (immat CHAR(10), jour DATETIME, km INTEGER,
    codea CHAR(10), CONSTRAINT pk_mision PRIMARY KEY(immat,jour),
    CONSTRAINT fk_mission_agent FOREIGN KEY(codea)
        REFERENCES agent(codea),
    CONSTRAINT fk_mission_veh1 FOREIGN KEY(immat)
        REFERENCES vehicule(immat));
ALTER TABLE agent ADD CONSTRAINT fk_agent_mission
    FOREIGN KEY(immat,jour) REFERENCES mission(immat,jour);
CREATE TABLE passagers (immat CHAR(10), jour DATETIME,
    codea CHAR(10), tempsPasse INTEGER,
    CONSTRAINT pk_passagers PRIMARY KEY(immat,jour,codea),
    CONSTRAINT fk_pax_mission FOREIGN KEY(immat,jour)
        REFERENCES mission(immat,jour),
    CONSTRAINT fk_pax_agent FOREIGN KEY(codea)
        REFERENCES agent(codea));
```

Les captures d'écran suivantes illustrent quelques résultats de rétroconception au niveau conceptuel. Bon nombre de solutions implémentent une composition. Ce faisant, une mission est identifiée par un numéro de véhicule et par un jour. Chaque mission pourra être reliée à un agent de trois manières distinctes.

Figure 4-36 Rétroconception (Visual Paradigm)

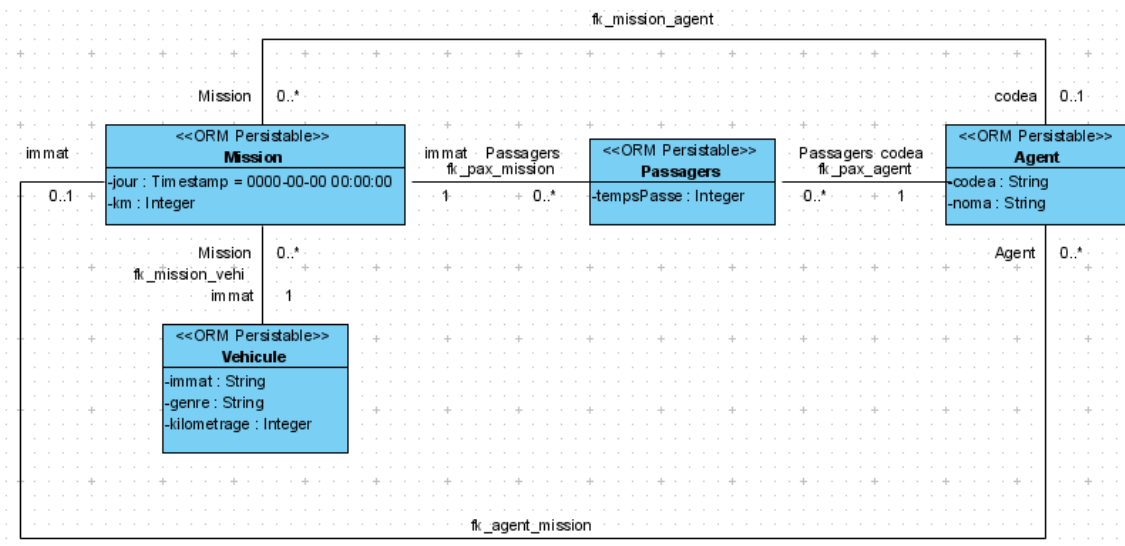


Figure 4-37 Rétroconception (Rational Rose)

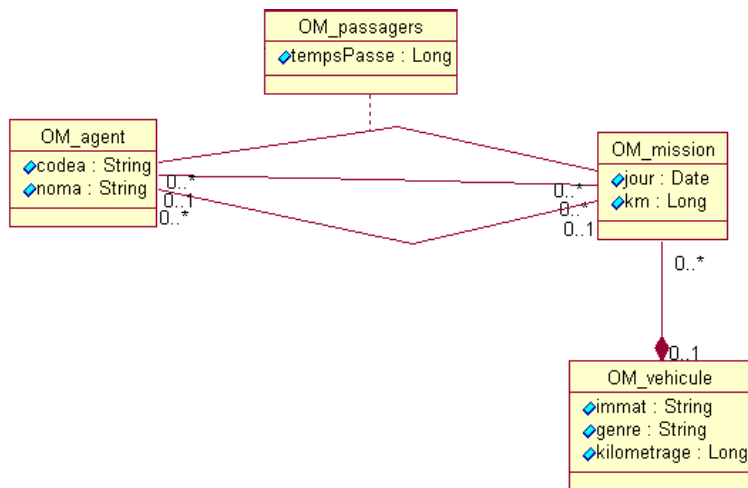


Figure 4-38 Rétroconception (Win'Design)

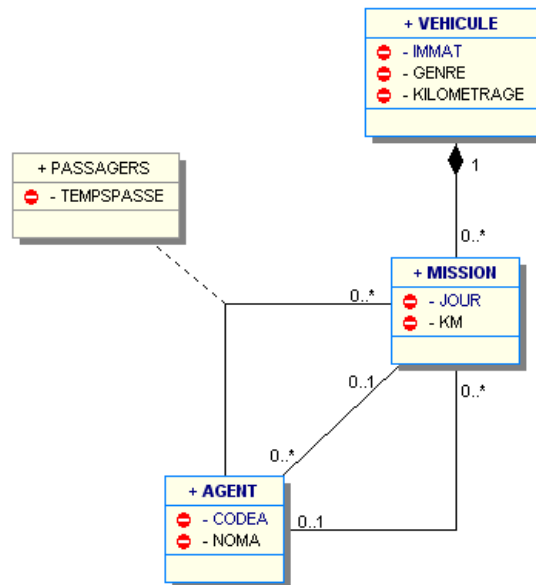


Figure 4-39 Rétroconception (PowerAMC)

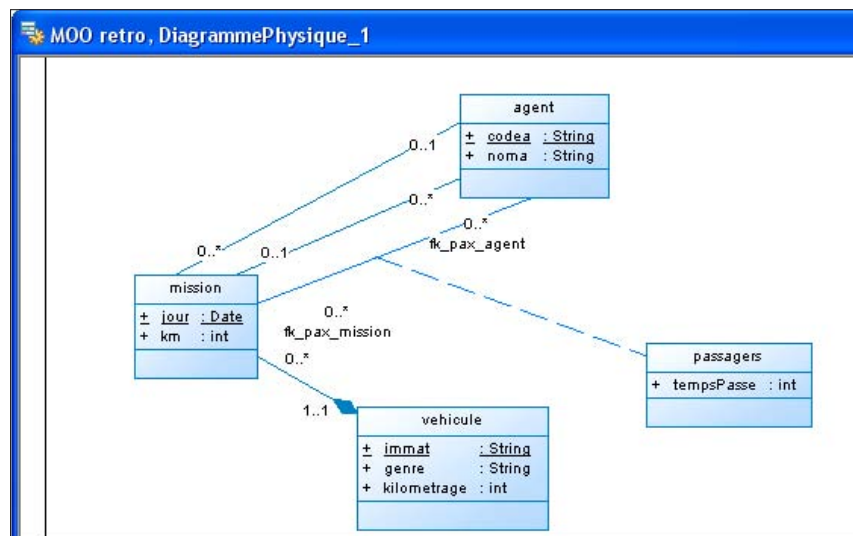
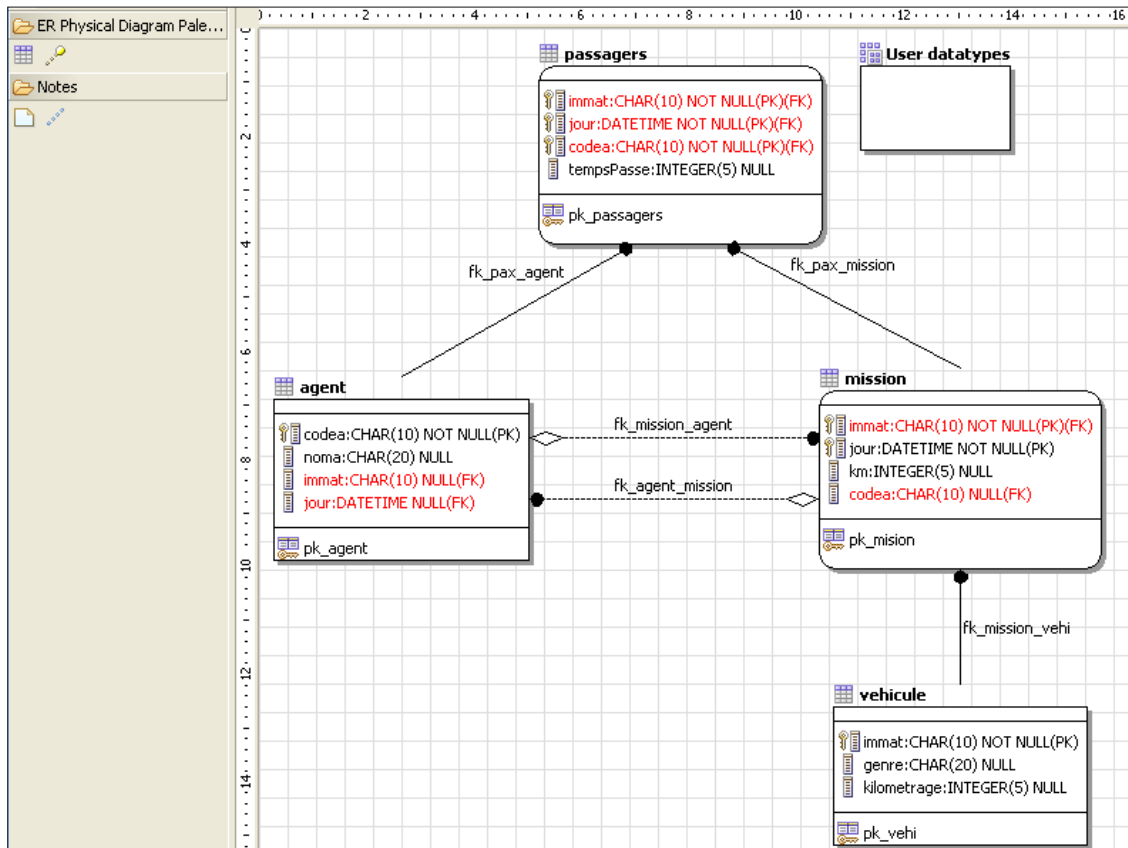


Figure 4-40 Rétroconception (Together)



Bilan intermédiaire

Seuls quatre outils sont capables de produire un diagramme UML correct suite à la rétroconception de la base. La majorité des outils permettent une connexion via ODBC ou JDBC. Certains sont encore plus puissants car ils permettent d'analyser en plus d'une base, le script SQL de création des tables, index et contraintes (il s'agit de PowerAMC, Rational Rose, Together et de Win'Design).

Tableau 4.7 Rétroconception

Rétroconception	Sources de données (connexion base, script SQL)	Modèle logique (3 cas de décomposition)
Enterprise Architect		
MagicDraw		
MEGA Designer		
ModelSphere		
MyEclipse		
Objectteering		
Poseidon		
PowerAMC		
Rational Rose Data Modeler		
Together		
Visio		
Visual Paradigm		
Visual UML		
Win'Design		

Bilan général

La note finale est déterminée à partir des résultats des tests précédents, de la robustesse, de l'ergonomie, de la documentation et du support éventuel que j'ai dû solliciter.

Tableau 4.8 Produit

278	278	Prix indicatif pour 1 licence	Version évaluée
PowerAMC	★★★★☆	2 800 €	12
Win'Design	★★★★☆	1 800 €	7
Rational Rose Data Modeler	★★★★☆	3 500 €	7.0
Objectteering	★★★★☆	2 000 €	6
MagicDraw	★★★☆☆	4 250 €	12.0
Enterprise Architect	★★★★☆	118 €	6.5
Visual Paradigm	★★★★☆	540 €	3
MEGA Designer	★★★☆☆	NC	2005 SP3
Together	★★★★☆	3 500 €	2006 R2
Visual UML	★★★★☆	230 €	5.0
MyEclipse	★★★☆☆	45 € /an	5.1
Visio	★★★☆☆	630 €	2007
Poseidon	★★★☆☆	700 €	5.x
ModelSphere	★★★☆☆	1 160 €/an	2.5

Quelques mots sur les outils

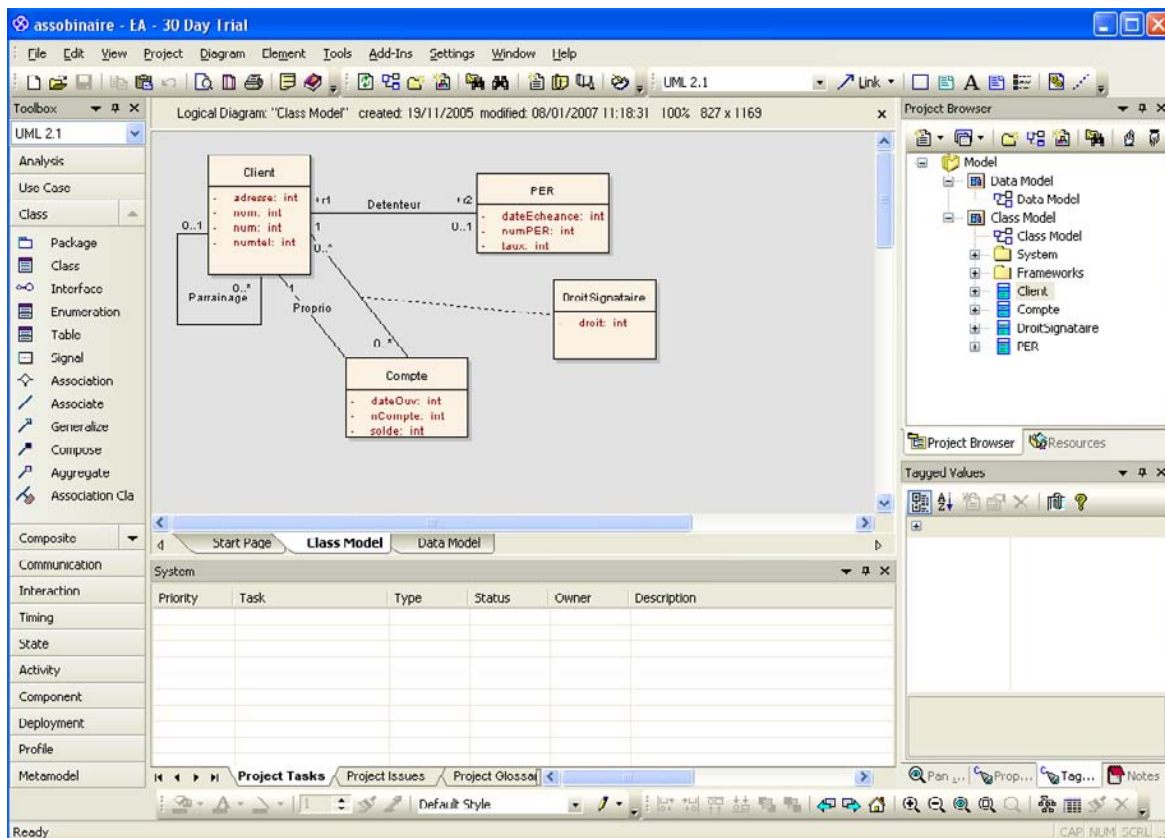
Si vous envisagez d'évaluer vous-même un des outils, ces quelques remarques vous feront sans doute gagner du temps lors de la mise en œuvre.

Enterprise Architect

Ergonomie soignée et robustesse pour cet outil Australien, qui pêche malheureusement dans les transformations de modèles comme nous le verrons par la suite. Créez un nouveau projet, puis choisissez les modules *Class* et *Database*. Sélectionnez *Data Architect* dans la fenêtre *Toolbox* pour disposer des icônes nécessaires à la construction de diagrammes de classes.

Ajoutez un nouveau diagramme dans la fenêtre de droite. Il n'est pas possible de définir un attribut identifiant par classe. Pour créer une classe-association (clic droit sur la classe Advanced/Make Association Class...). Si vous supprimez des éléments, pensez à vérifier dans la fenêtre du *browser* qu'ils sont bien effectivement retirés du modèle.

Figure 4-41 Enterprise Architect

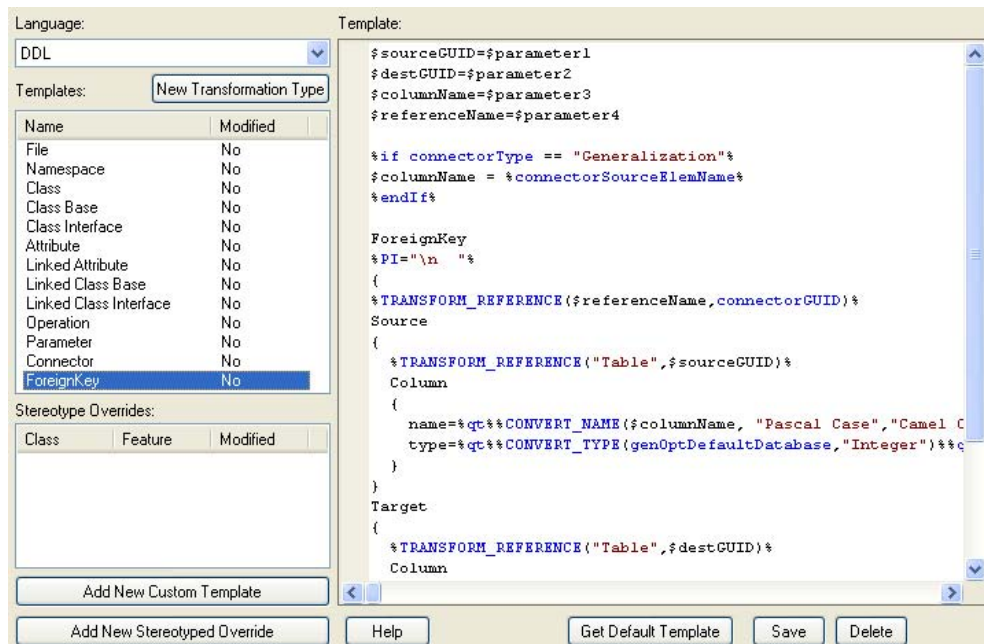


Pour générer un modèle logique exprimé dans le formalisme UML, utilisant le profil pour les bases de données (celui initié par Rational Rose), il faut d'abord le transformer en paquetage (Project/Source Code Engineering/Generate Package Source Code...) puis Project/Model Transformation/Transform Current Package (choisir DDL et le répertoire cible).

Pour générer un modèle physique suivez l'assistant qui apparaît après avoir sélectionné : Project/Database Engineering/Generate Package DDL...

En ce qui concerne les points forts de ce produit, citons sa documentation de bonne qualité avec un lien direct vers le forum Internet dédié, la possibilité de personnaliser un grand nombre de paramètres et processus et la facilité de travailler avec différents paquetages au sein d'un même diagramme de classes.

Figure 4-42 Programmation du mapping



Les points faibles principaux sont l'absence de transformation native entre modèles logiques et conceptuels et la faible qualité du mécanisme de transformation de modèles (génération très basique des clés étrangères). En effet, seules les associations binaires sont correctement traduites. Pour les associations *n*-aires, les classes-associations ou les agrégations, le concepteur doit enrichir le processus de transformation (reprogrammation du *mapping*). Chose compliquée et hasardeuse. L'héritage ne propose qu'un seul cas de décomposition. La rétro-conception est correcte, seul un schéma logique au formalisme UML est produit (pas de remontée d'un modèle conceptuel).

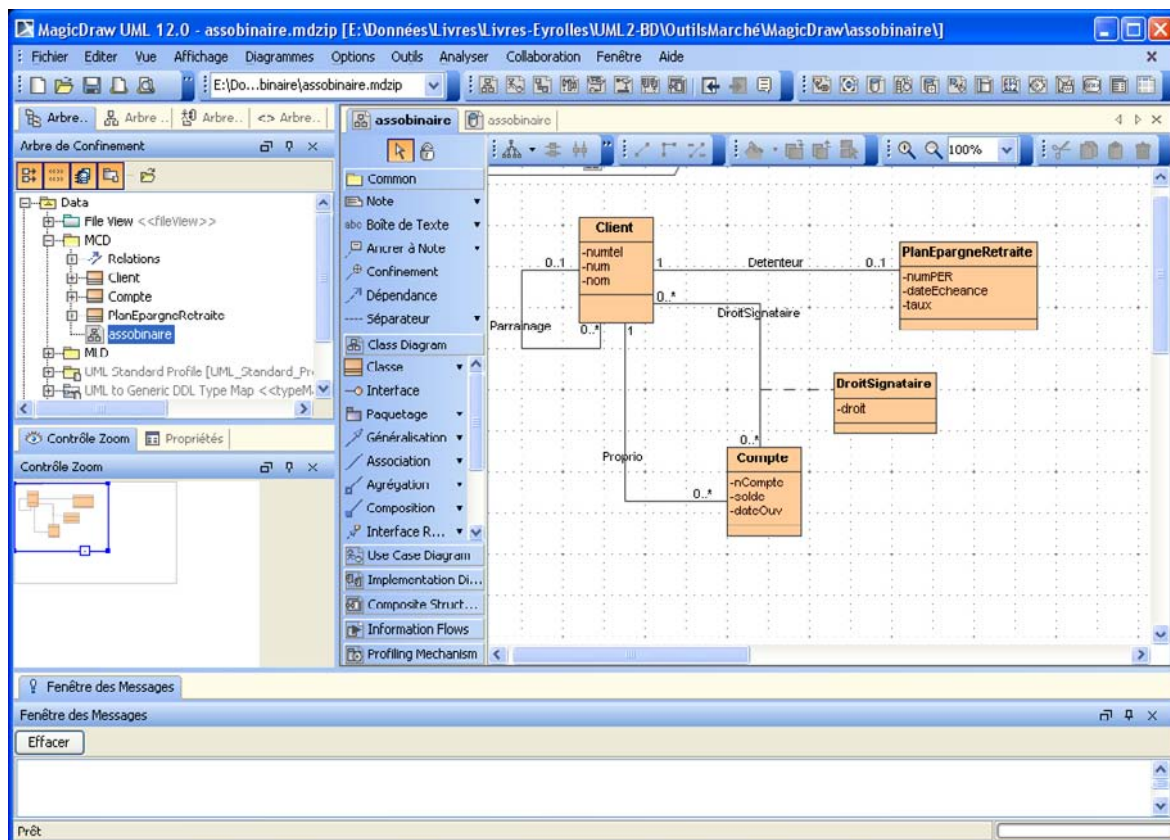
MagicDraw

Outil robuste à l'ergonomie semblable à celle de Visual Paradigm, MagicDraw de l'éditeur No Magic, Inc. se targue sur son site Web, par de nombreuses nominations, d'avoir été élu à plusieurs reprises meilleur outil de modélisation. Bien qu'il propose des assistants de

conversion puissants, son mécanisme de transformation de classes-associations n'est pas encore au point.

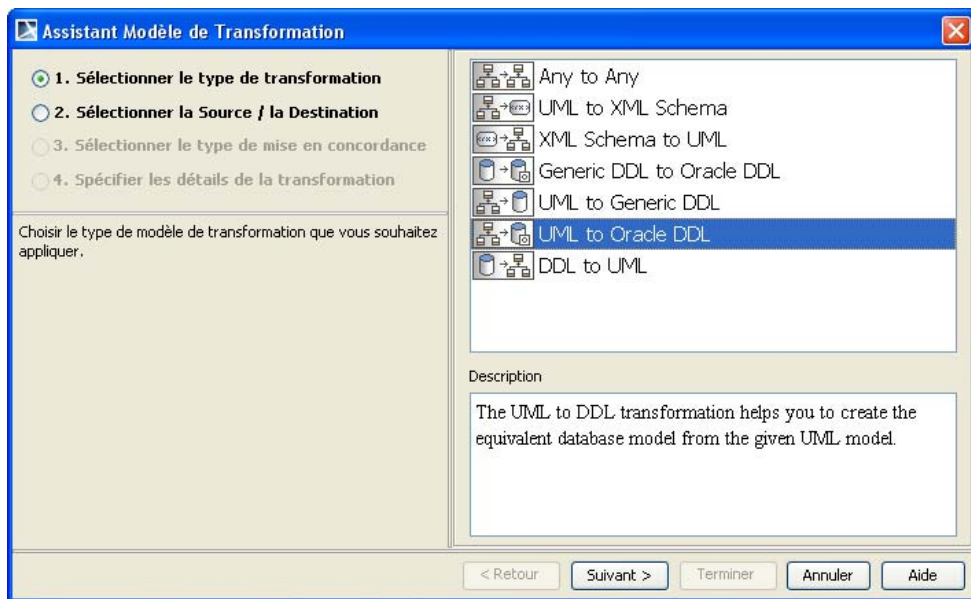
Créez un nouveau projet dans lequel vous installerez un nouveau paquetage contenant votre diagramme de classes. La saisie des éléments du diagramme de classes est intuitive mais il n'y a malheureusement pas de possibilité de définir d'identifiant de classe. Pensez à créer un autre paquetage qui contiendra le modèle logique (modèle relationnel exprimé au profil UML).

Figure 4-43 MagicDraw



Pour transformer les classes en relations, passer par Outils/Transformer qui lance un assistant. Pour définir une contrainte d'un graphe héritage, vous devez créer un ensemble de généralisations reliant les liens des sous-classes, donner un nom à votre contrainte et choisir parmi les options relatives à la disjonction et à la couverture.

Figure 4-44 Assistant de transformation (Magic Draw)



La génération du script SQL s'opère via le répertoire **Jeux de Code d'Ingénierie (New/DDL...)** : créez une base puis faites glisser vos relations du paquetage du niveau logique. Ensuite, il suffit de sélectionner la base puis de lancer la génération.

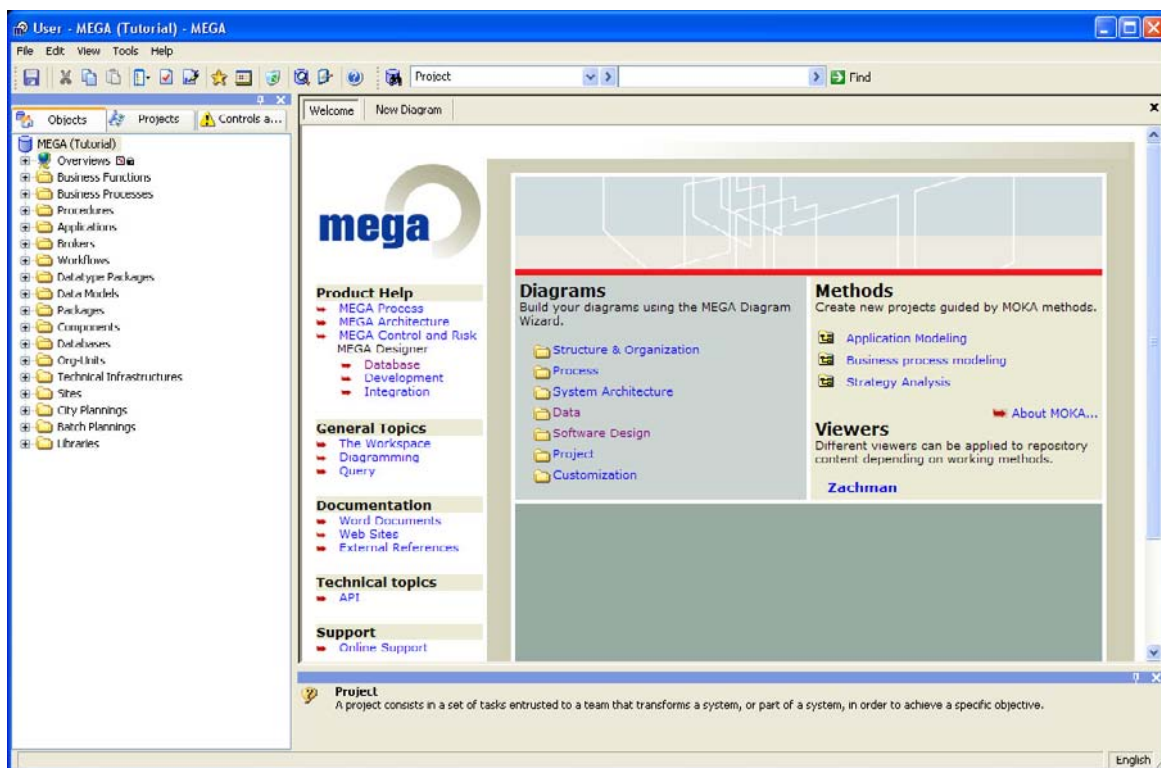
Pour la rétroconception, il faut créer une base dans ce même répertoire, puis **Editer...** lance un menu qui précise l'accès à la base (souvent une connexion JDBC). Ensuite, il suffit de lancer le processus par le menu **Inverser**. Le schéma relationnel obtenu par rétroconception est au profil UML. Un assistant vous permet de le convertir en diagramme (conceptuel) de classes UML (menu **Outils/Transformer/DDL to UML**). Pour visualiser les associations, positionnez-vous sur une des classes du schéma conceptuel (clic droit **Elements reliés/Afficher les éléments liés**). Le diagramme risque fort d'être de qualité moyenne si votre base de données implémente des associations plusieurs-à-plusieurs ou d'agrégation que vous vous attendiez légitimement à visualiser en tant que classes-associations.

Les points forts résident dans la possibilité de transformation d'une base à une autre, dans la prise en compte des contraintes d'un graphe d'héritage, dans le choix varié des pilotes de SGBD et des animations des fonctionnalités téléchargeables sur le site. Les limitations concernant UML sont l'absence d'identifiant de classe et la transformation des classes-associations (qui n'est pas du tout prise en compte). De plus, aucun cas de décomposition n'est prévu pour l'héritage, l'éditeur semble attendre la tendance du marché pour se décider à implémenter une solution.

MEGA Designer

Faisant partie des solutions relatives à l'architecture et à la gouvernance du système d'information de la société Française MEGA International, l'outil Designer se consacre aux différents modèles de données servant à la conception.

Figure 4-45 MEGA Designer



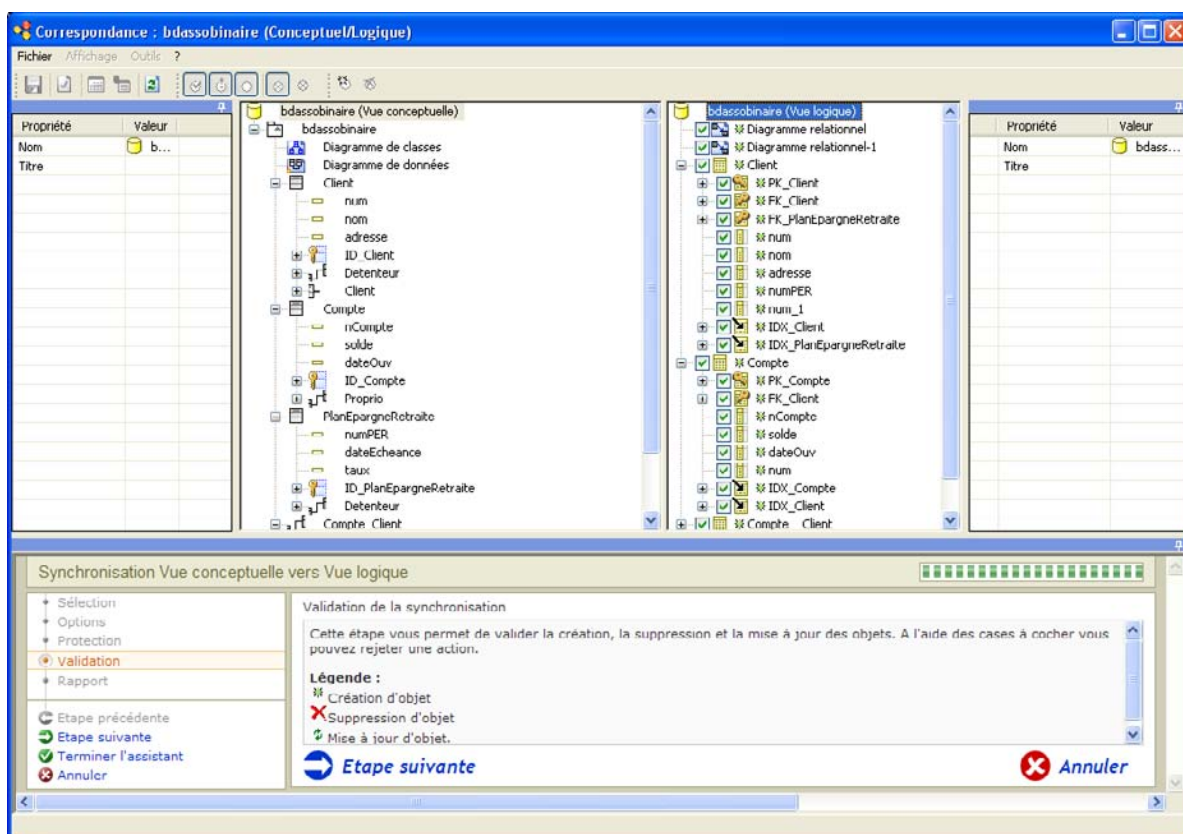
Pour débiter, créez une base de données (dans le répertoire Databases) dans laquelle vous définirez un nouveau modèle de données (clic droit New...). La création de ce modèle de données va entraîner automatiquement la création d'un paquetage (portant le nom de la base). Dans le répertoire Paquetages, sélectionnez ce paquetage et créez un diagramme de classes (New/Generate a Class Diagram).

Pour créer une classe-association, ne définissez pas de classe mais cliquez sur l'association puis Association class... Au même titre que PowerAMC et Win'Design, MEGA permet l'identification relative.

Afin de générer un modèle logique, positionnez-vous sur votre base dans le répertoire Data-bases et choisissez **Editer** puis **Outils** (synchroniser du conceptuel au logique). Un assistant en quatre étapes est alors lancé. Le script SQL est créé via l'option **Générer**.

Pour la rétroconception, créez une base puis **Reverse Database**, sélectionnez l'entrée ODBC. Il faudra définir un diagramme (**New/Diagram/Relational Diagram**) pour faire un glisser-déposer des tables obtenues. Pour obtenir l'équivalent UML du modèle, il faut synchroniser (clic droit sur la base puis **Edit/Tools/Synchronisation...**). Le schéma obtenu par rétroconception peut être de qualité moyenne. À noter enfin l'existence d'un modèle de données hybride entre UML et le niveau logique relationnel appelé *Data diagram*.

Figure 4-46 Synchronisation (MEGA)



La principale limitation concernant UML est l'absence d'association *n*-aire et la qualité de transformation des classes-associations et de l'héritage multiple. Par ailleurs, aucun pilote de SGBD *Open Source* n'est pris en compte dans le processus de conception.

Les points forts sont la robustesse de son ergonomie, son outil de recherche d'éléments inter-objets (processus, paquetages, bases de données, etc.), son outil de synchronisation par étapes guidées par un assistant et les passerelles possibles avec PowerAMC et Rational Rose.

ModelSphere

Outil de la société québécoise Grandite, ModelSphere permet de créer des modèles conceptuels de type entité-association, des modèles relationnels (associés ou non à une base) selon différents formalismes et notamment UML. La partie UML de l'outil permet de créer un diagramme de classes. Le processus de transformation vers le niveau logique n'est pas finalisé notamment au niveau des identifiants et des clés étrangères car des modèles de liens (pas encore opérationnels) doivent être mis en œuvre. Le niveau logique est donc composé de tables sans clés liées entre elles graphiquement.

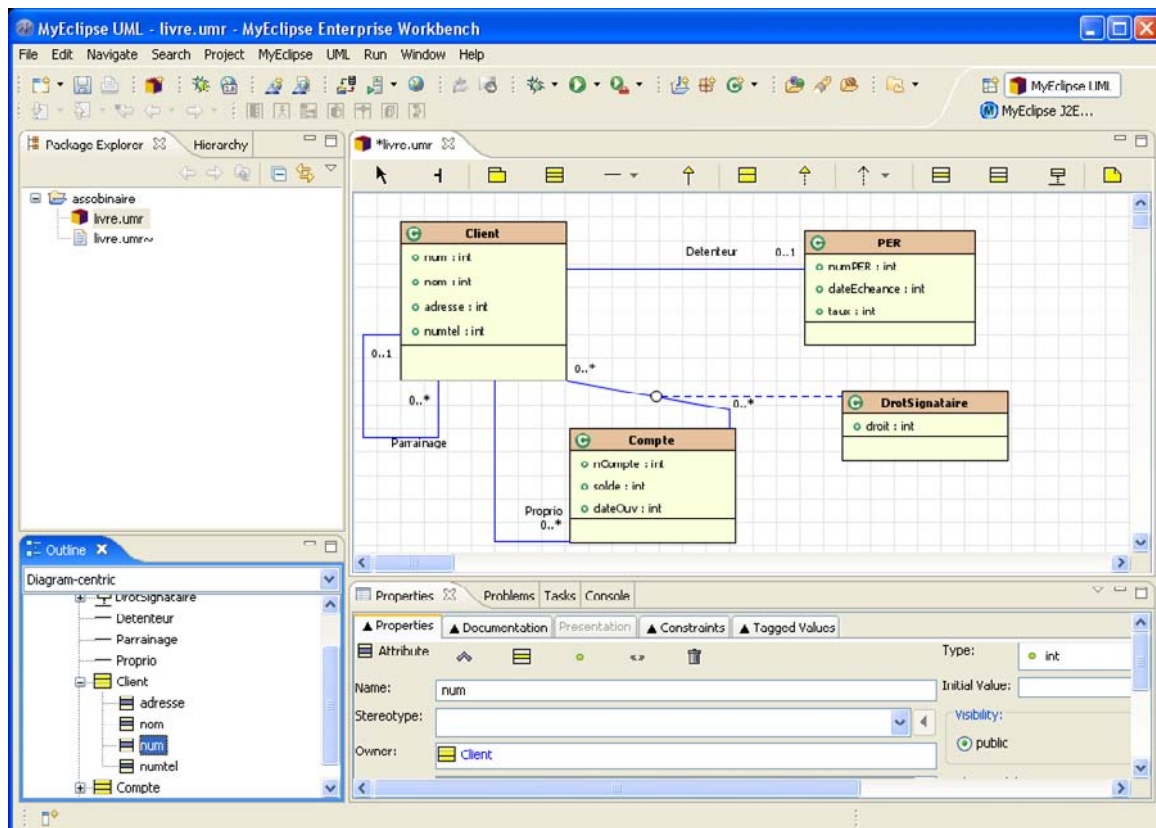
Pour générer un modèle logique, sélectionnez le modèle de classes (clic droit **Générer Modèle de données...**). Par la suite, positionnez-vous dans le répertoire contenant les tables générées et ajoutez un diagramme (clic droit **Ajouter**). Faites glisser les tables et les associations générées pour découvrir le schéma relationnel généré.

Les principales limitations concernant UML sont l'absence d'identifiant de classe, de notation graphique pour les classes-associations et les associations *n*-aires. Le processus de rétro-conception d'une base est correct (via ODBC ou JDBC) mais le schéma généré UML est de qualité moyenne puisque le processus traduit toute table en une classe. Pas de moyen pour l'heure de traduire un MCD en diagramme de classes et inversement.

MyEclipse

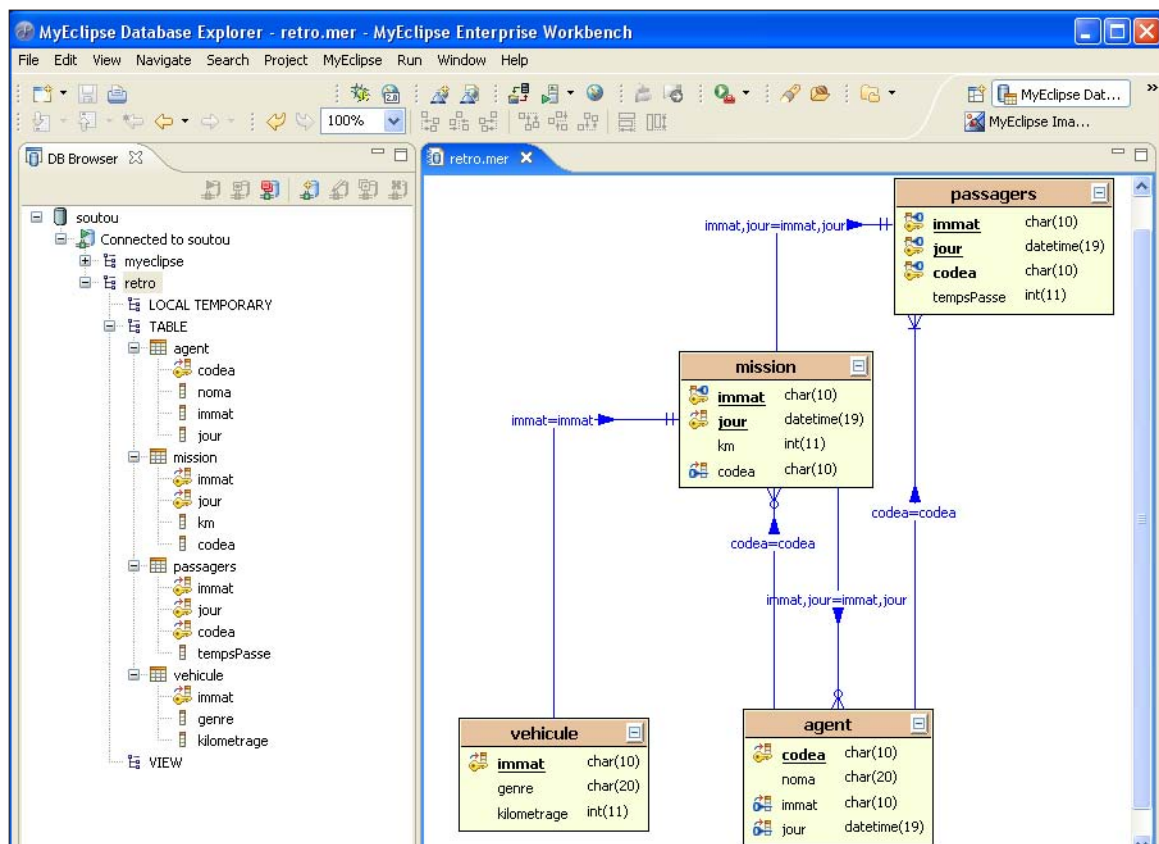
Comme Poseidon, cet outil n'est pas vraiment fait pour les bases de données (niveau logique inexistant de même que la génération de code SQL à partir d'un diagramme UML). L'ergonomie est toutefois mieux réussie que Poseidon. Cet outil oblige l'utilisateur à se servir du modèle *Entity-Relationship* pour concevoir ses schémas conceptuels.

Figure 4-47 MyEclipse



Les principales limitations côté UML se rapportent à l'absence d'identifiant de classe et de notation pour les associations *n*-aires. Le processus de rétroconception d'une base est correct (grand choix de SGBD) mais le schéma généré est de type *Entity-Relationship* (notation *crow's foot*). Pour vous faire une idée de ces possibilités Preferences/MyEclipse/Database Explorer, configurez votre accès à la base puis créez une connexion Window/Open Perspective/Other/MyEclipse/Database Explorer. Créez ensuite un diagramme (clic droit New ER Diagram), enfin sélectionnez les tables à rétroconcevoir.

Figure 4-48 Niveau logique (MyEclipse)



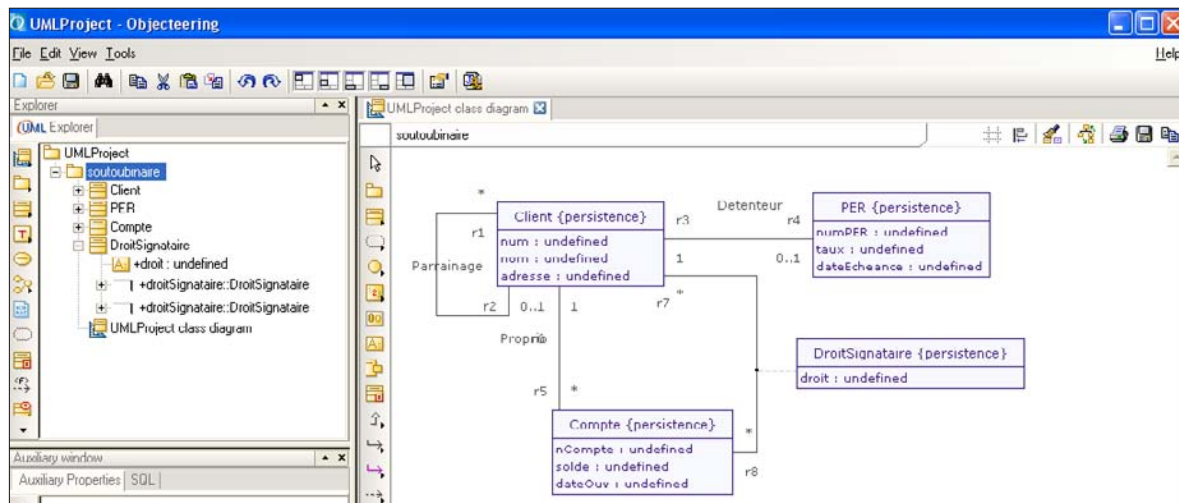
Objecteering

Objecteering est un outil français de la société éponyme, filiale de SOFTEAM, qui a été acteur majeur dans la communauté des technologies objet et premier membre Européen de l'OMG. Dans ce logiciel, le terme *physical model* désigne un schéma relationnel (niveau logique) et *logical model* désigne un script SQL.

Avant tout, créez un projet puis configurez l'affichage des tags qui serviront à annoter un diagramme de classes pour sa transformation (Tools/Diagram graphic options/Properties Class Diagram, rendre visible les *tagged values*). Ensuite déployez le module SQL (Tools/Deploy an MDAC... choisir SQLDesigner). Concernant ce module (Tools/MDAC options...), configurez l'accès à votre base. Dans l'option Diagram generation... cochez la case Generates diagram...

Vous pouvez créer un paquetage (dans la racine) qui contiendra votre diagramme de classes. La transformation nécessite d'enrichir le diagramme UML de *tagged values*. Ainsi vous devrez annoter chaque classe du tag *persistence* (avec la propriété *persistent*). Les identifiants sont choisis via un clic droit sur la classe MDA Components/SQL Designer/Primary key... Pendant la saisie du diagramme de classes servez-vous plutôt de la fenêtre de gauche (*UML Explorer*) pour supprimer les éléments superflus.

Figure 4-49 Objecteering



La génération d'un schéma relationnel se fait au niveau du paquetage contenant le diagramme UML (clic droit MDA Components/SQL Designer/Generate physical model). La génération d'un script SQL se fait au niveau du paquetage contenant le modèle physique (clic droit MDA Components/SQL Designer/Generate SQL files).

Un grand nombre de points forts sont à noter : rigueur de la démarche et ergonomie à la fois sobre et puissante, documentation détaillée du module SQL Designer (principes de transformations), réactivité et efficacité du support ainsi que l'existence d'un forum dédié.

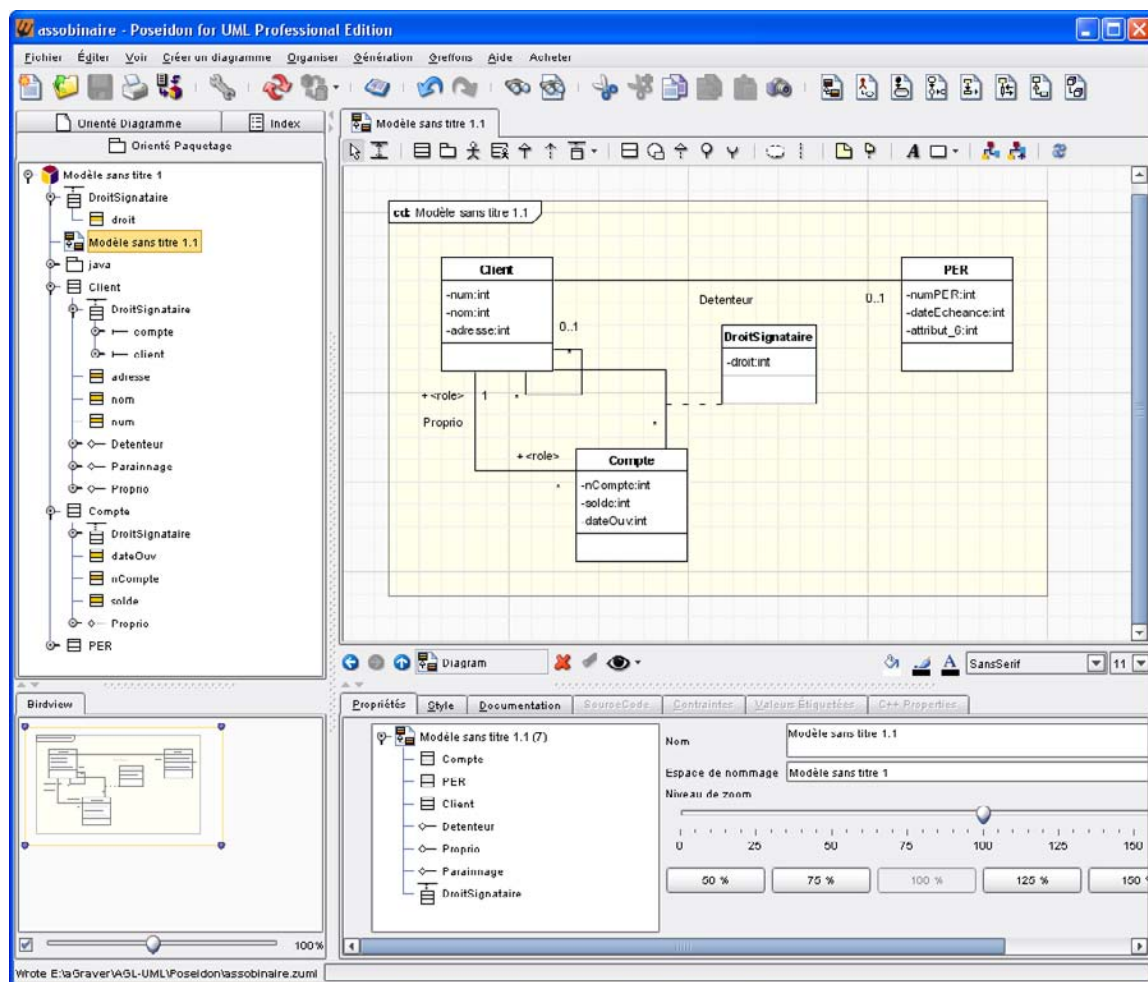
Les points faibles concernent le faible nombre de SGBD pris en compte (notamment SQL92 et tous les *Open Source*), l'absence de processus de rétroconception et la non-prise en compte de l'héritage multiple. Enfin, certaines transformations de diagrammes ont posé des problèmes qui sont désormais identifiés et en cours de correction.

Poseidon

Cet outil n'est pas vraiment fait pour les bases de données. Le niveau logique n'est pas supporté, la génération de code SQL est très contestable (les classes-associations ne sont pas traduites et les associations sont parfois inexistantes). L'ergonomie surprend au début mais on

s'y fait. La création de classes-associations et d'associations réflexives n'est pas aisée. Les options ne sont pas toutes traduites en français, les associations *n*-aires ne sont pas prises en compte, il n'existe pas de processus de rétroconception d'une base, n'en jetez plus !

Figure 4-50 Poseidon

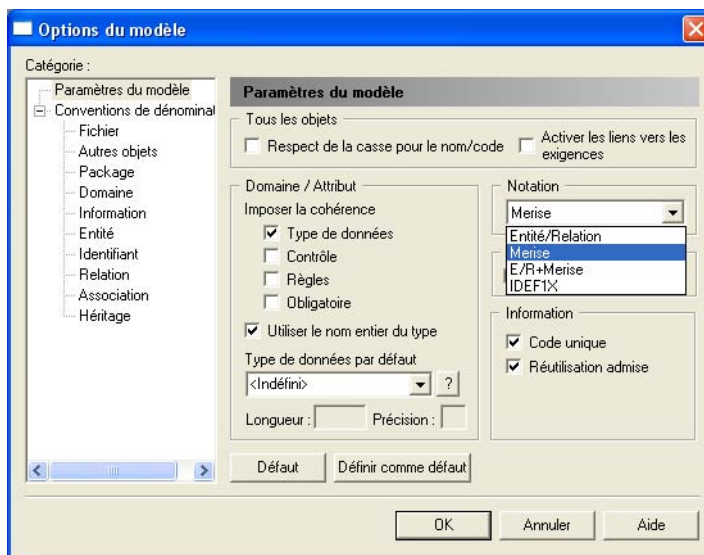


PowerAMC

PowerAMC (anciennement AMC*Designer) est la version française de l'outil de modélisation PowerDesigner de Sybase. Concernant les bases de données, l'outil prend en charge trois types de modèles qu'on peut transformer entre eux :

- le modèle conceptuel de données (MCD) qu'on pourra construire avec la notation Merise, entité-relation ou IDEF1X ;
- le modèle physique de données (MPD) qui correspond au niveau logique ;
- le modèle orienté objet (MOO) au formalisme UML.

Figure 4-51 Formalismes du MCD (PowerAMC)



L'ergonomie de l'outil est très réussie : il est très intuitif de créer différents diagrammes, de les transformer entre eux, de générer le script SQL ou de rétroconcevoir une base de données (à partir des tables ou d'un script).

Les transformations de modèles se font très facilement en ouvrant le diagramme puis Outils/ Générer un Modèle...). Pour manipuler une base de données (génération ou rétro-conception), il faut travailler avec un modèle physique pour que l'option SGBD soit active.

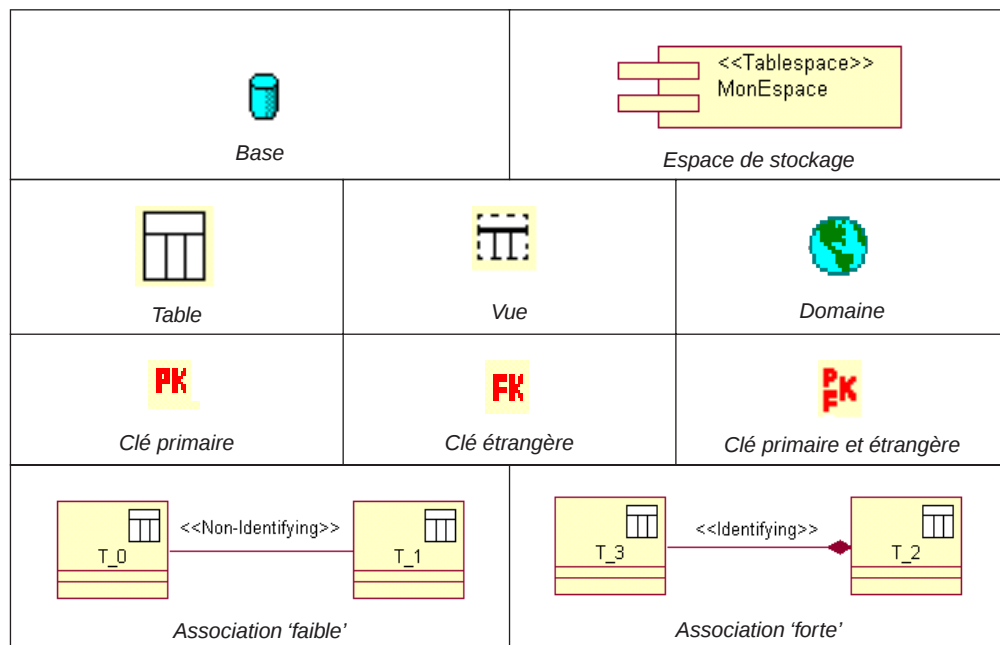
Quelques points forts sont à mettre en exergue : un grand choix de SGBD est pris en compte, la robustesse et la qualité de l'ergonomie qui permet de travailler facilement avec différents diagrammes dans le même environnement. Les seuls points faibles se réduisent à la non-prise en compte des contraintes et des associations *n*-aires avec la notation UML.

Rational Rose Data Modeler

À l'origine de la standardisation d'UML, Rational a proposé un profil convenant aux bases de données. Un profil, proposition d'une communauté, regroupe un ensemble d'éléments (composants, stéréotypes, icônes, propriétés...) qui s'appliquent à un contexte particulier tout

en conservant le métamodèle d'UML intact. Le profil UML, pour la modélisation de données (*UML profile for Data Modeling*), permet de manipuler des bases de données à l'aide notamment de classes UML possédant des stéréotypes prédéfinis (*Table*, *RelationalTable*, *View*...). Les icônes de ce profil sont illustrées figure 4-52.

Figure 4-52 Icônes du profil UML



Il est nécessaire dans un premier temps de créer un composant *Database* dans le répertoire *Component View* (clic droit sur un répertoire, *Data Modeler/New/Database*). On donne ensuite un nom à la base et on choisit la nature du SGBD utilisé en cible (*Name* et *Target*). Il faut ensuite créer un schéma dans le répertoire *Logical View* (clic droit sur le répertoire, option *Data Modeler/New/Schema*). Associez ensuite ce schéma à la base de données cible précédemment créée.

Le diagramme de classes UML doit être situé dans un paquetage (clic droit sur le compartiment *Logical View*, *New/Package*). Les classes doivent être marquées *Persistent* (clic droit, *Open Specification*, onglet *Detail*). Tous les éléments du modèle objet ne se transforment pas forcément en classes de stéréotype *<<Table>>* au niveau logique. La documentation stipule qu'il est prudent d'examiner en détail le résultat de la transformation de manière à s'assurer de la bonne structure de la base. On retrouve ici l'idée de départ de l'ouvrage, à savoir la maîtrise des concepts pour une meilleure utilisation de l'outil.

La transformation du modèle objet se fait en sélectionnant le paquetage des classes UML à transformer (clic droit puis *Data Modeler/Transform to Data Model...*). Pour visualiser le modèle logique créé dans le schéma un diagramme (*Data Modeler/New/Data Model Diagram*) puis faites glisser chaque relation obtenue par la transformation.

Le processus de rétroconception démarre du schéma (compartiment *Logical View/Schemas*) qu'on sélectionne par un clic droit (*Data Modeler/Transform to Object Model*). Un schéma se transforme en un paquetage à créer initialement. Un assistant permet de sélectionner le nom du paquetage en sortie, et si oui ou non, les clés primaires doivent être transformées en identifiant de classe. Le fait de transformer un modèle de données dans un paquetage modifie les nouveaux éléments du paquetage mais ne détruit ni ne modifie les éléments mis à jour au niveau du modèle de données. Si cette option du logiciel fonctionne bien pour les associations binaires, classes-associations et l'héritage, elle n'est pas opérationnelle pour les associations *n*-aires.

Bien que peu de modifications ont été apportées à cet outil depuis la première version de cet ouvrage, le point fort du logiciel est sa très grande robustesse. On peut toutefois regretter qu'il n'inclue pas encore de pilotes pour les SGBD *Open Source* et qu'il ne propose pas l'agrégation simple et les associations *n*-aires.

Together

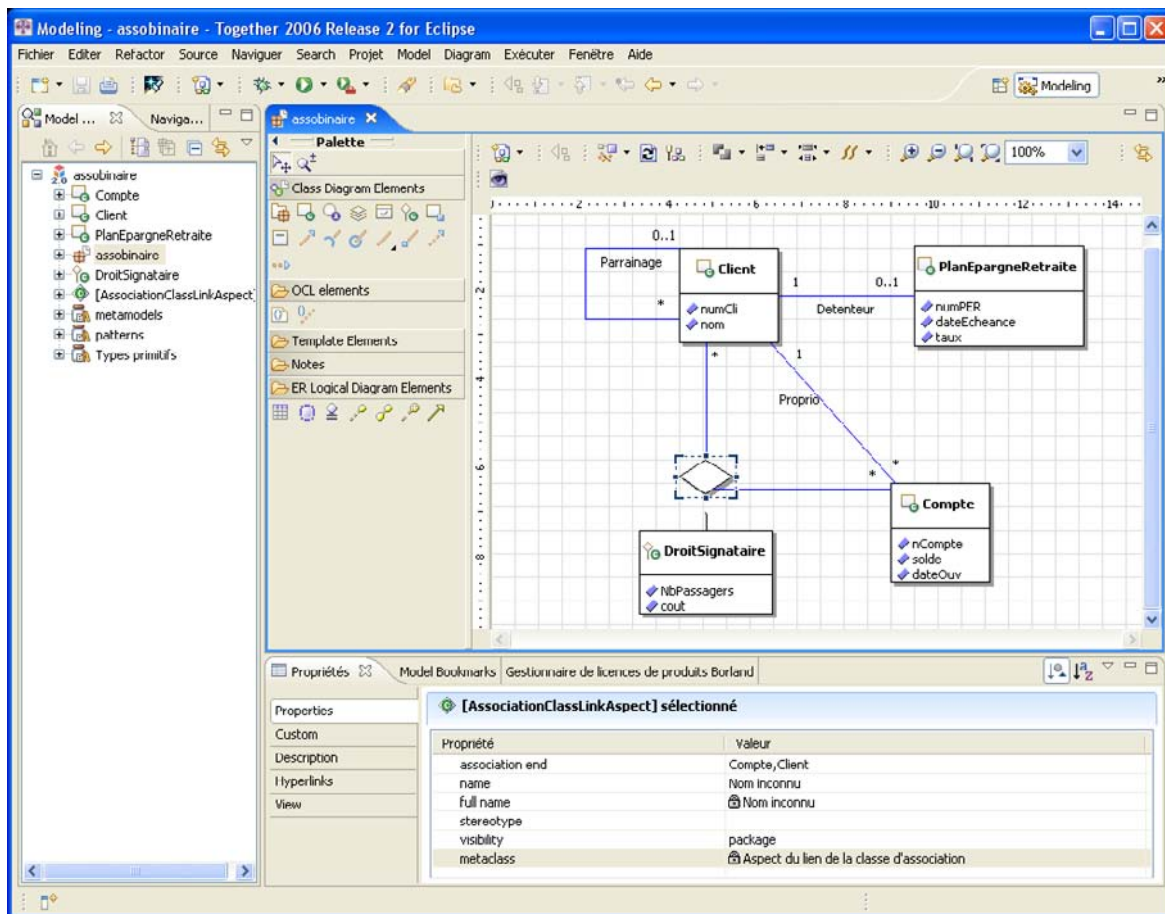
Together est l'outil de conception de Borland. La partie relative aux bases de données est nommée *Data Modeling* et est présente dans la version pour Eclipse. Une autre version existe pour VisualStudio mais elle ne prend pas en compte l'aspect modélisation de bases de données.

Dans la partie *Data Modeling*, seuls les niveaux logique et physique sont présents. Le formalisme des modèles logiques s'apparente au profil UML pour les bases de données. Dans ce modèle, deux types d'associations sont prévues : celles qui doivent donner lieu à la création d'une table (*many-to-many relationship*) et celles qui génèrent seulement une clé étrangère (*relationship*). Pour ces dernières, l'entité cible du lien créé graphiquement détermine l'entité parent. Les modèles physiques font apparaître les clés étrangères sous la forme graphique (formalisme IDEF1X). La transformation entre les deux modèles se fait par la fonction *Importer/DB schema from ER Logical Diagram*. La génération de script SQL s'opère par la fonction *Exporter/DDL SQL script*.

La création d'un diagramme de classes nécessite de créer un projet (*Nouveau/Autres/ULML 2.0 Project*). Une palette de symboles est alors mise à disposition. Pour créer une classe-association (ou une association *n*-aire), sélectionnez l'icône *Association Class*. Ensuite, reliez le symbole losange (*diamond*) aux différentes classes avec le lien *Association End*.

La transformation automatique du diagramme de classes en modèle logique n'est pas native-ment proposée. On ne peut avoir accès à la base de données que par le modèle logique et UML n'est pas encore exploité au niveau conceptuel. Pour ce faire, il faudra programmer des

Figure 4-53 Together



transformateurs de modèles (similaires aux règles de mapping de Enterprise Architect). La documentation fait toutefois preuve d'optimisme "The concept of entities and relationships in logical data modeling maps to the concept of classes and associations in the UML 2.0 class diagram". Il n'empêche que la programmation des règles de transformation des classes-association, associations *n*-aires et l'héritage par les éléments du modèle logique profilé par Borland risque de ne pas être de tout repos.

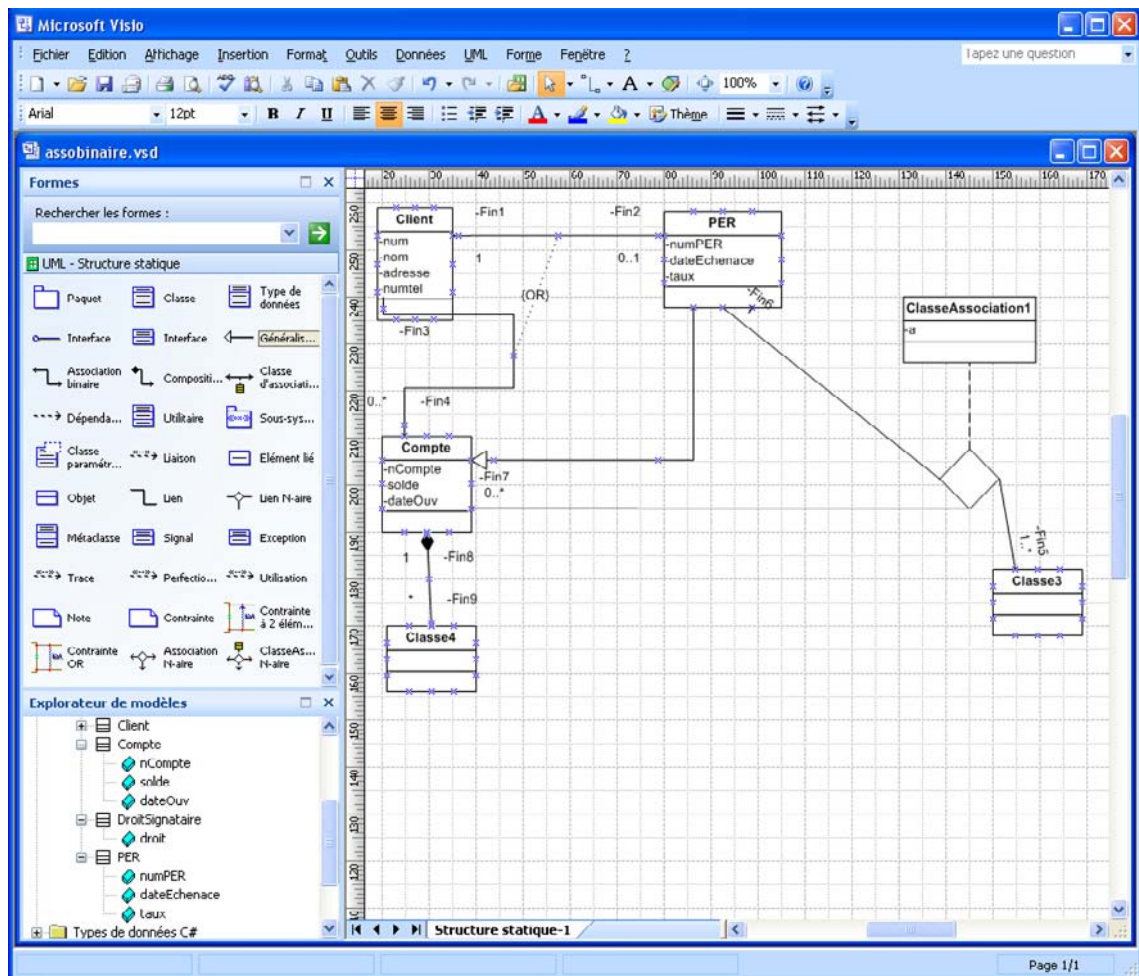
La rétroconception consiste à importer (Fichier/Importer, la source étant soit un script, soit une connexion JDBC à une base qu'il faudra paramétrer). Le schéma généré est un modèle physique dont le formalisme s'apparente au profil UML et qui fait apparaître explicitement les clés étrangères.

Les points forts de l'outil : sa qualité et sa robustesse, l'efficacité du support et l'expertise dans les fonctionnalités de transformation de modèles par l'utilisation de QVT et de EMF (Eclipse Metamodel Framework). Le point faible majeur concerne l'absence de processus natif de transformation entre schémas conceptuels et modèles logiques.

Visio

Visio fait partie de la suite Office de Microsoft. En utilisant la notation UML, il n'est pas possible de générer de modèle ni de code. Pour ce faire, vous devrez travailler soit avec le modèle ORM (modèle conceptuel binaire graphique), soit avec IDEF1X (modèle relationnel

Figure 4-54 Visio



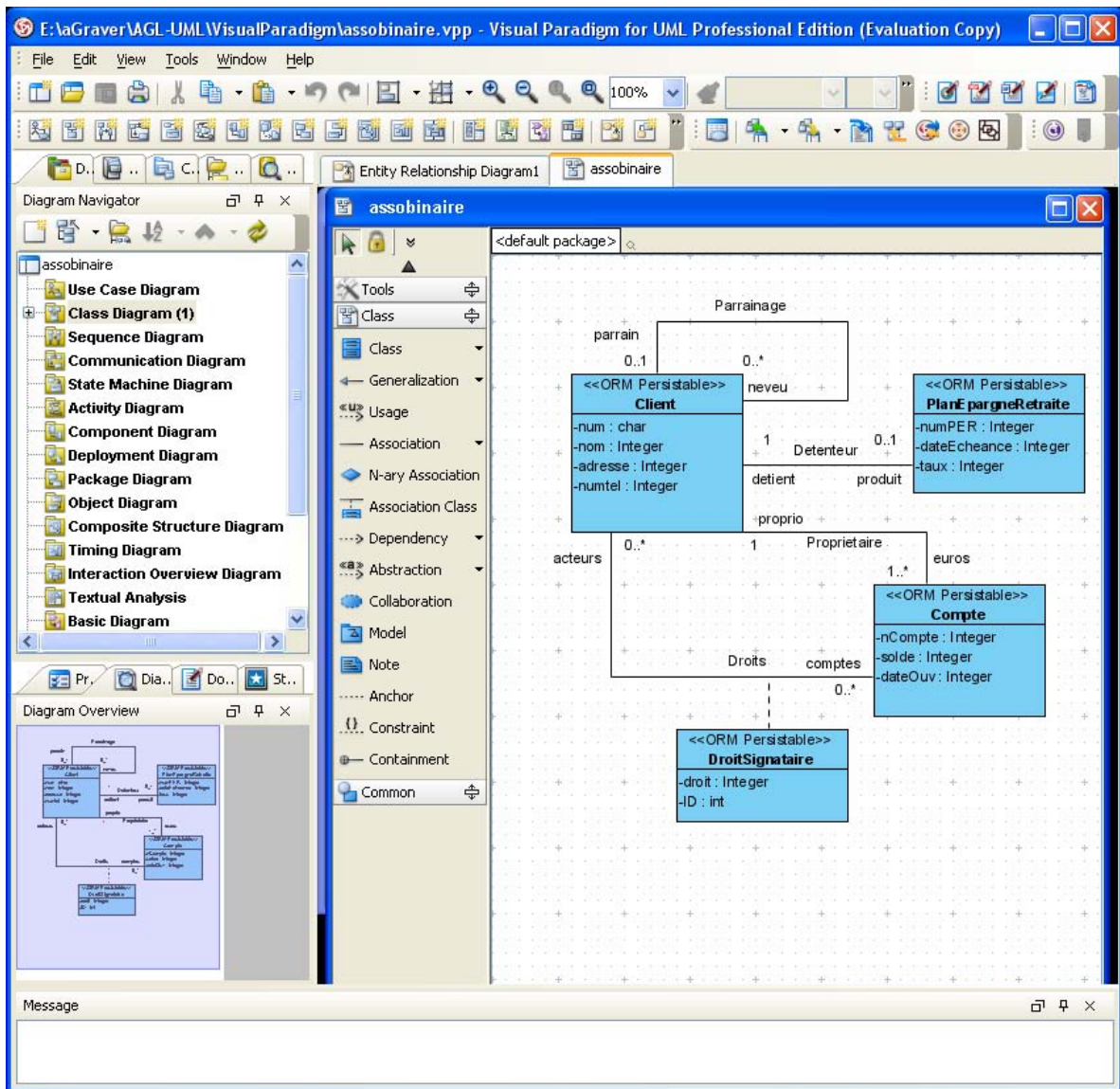
graphique). Ce sont les seuls formalismes qui permettent de se connecter à une base. Dans ces deux cas, vous devrez utiliser Visio au sein de l'environnement de Visual Studio. Si vous utilisez une base *Open Source*, il faudra trouver le bon pilote (en utilisant un pilote générique ODBC pour MySQL j'ai eu la surprise de constater que les colonnes de mes tables étaient inscrites en caractères japonais !).

Visual Paradigm

Une fois saisi le diagramme de classes UML, nommez impérativement tous les rôles des liens d'associations (sinon la génération au niveau logique ne sera pas possible). Ensuite configurez votre transformation (Tools/Object... (ORM)/Wizards...). Rendez persistantes vos classes, choisissez un identifiant pour chacune et paramétrez votre connexion à la base de données cible. Une fois le modèle logique généré, il est nécessaire de le synchroniser (Tools/Object... (ORM)/Synchronize...) avec le diagramme de classes avant de pouvoir générer un script SQL. De toute façon, vous devrez souvent agir sur le modèle logique (surtout pour les identifiants des classes-associations) avant de générer une base de données.

Quelques points forts : un grand choix de SGBD est pris en compte ; la réactivité et l'efficacité du support. Les points faibles concernent le déplacement des objets et des liens sur un graphique du niveau logique (*Entity-Relationship*) qui n'est pas toujours réussi et la non-prise en compte des identifiants et des associations *n*-aires.

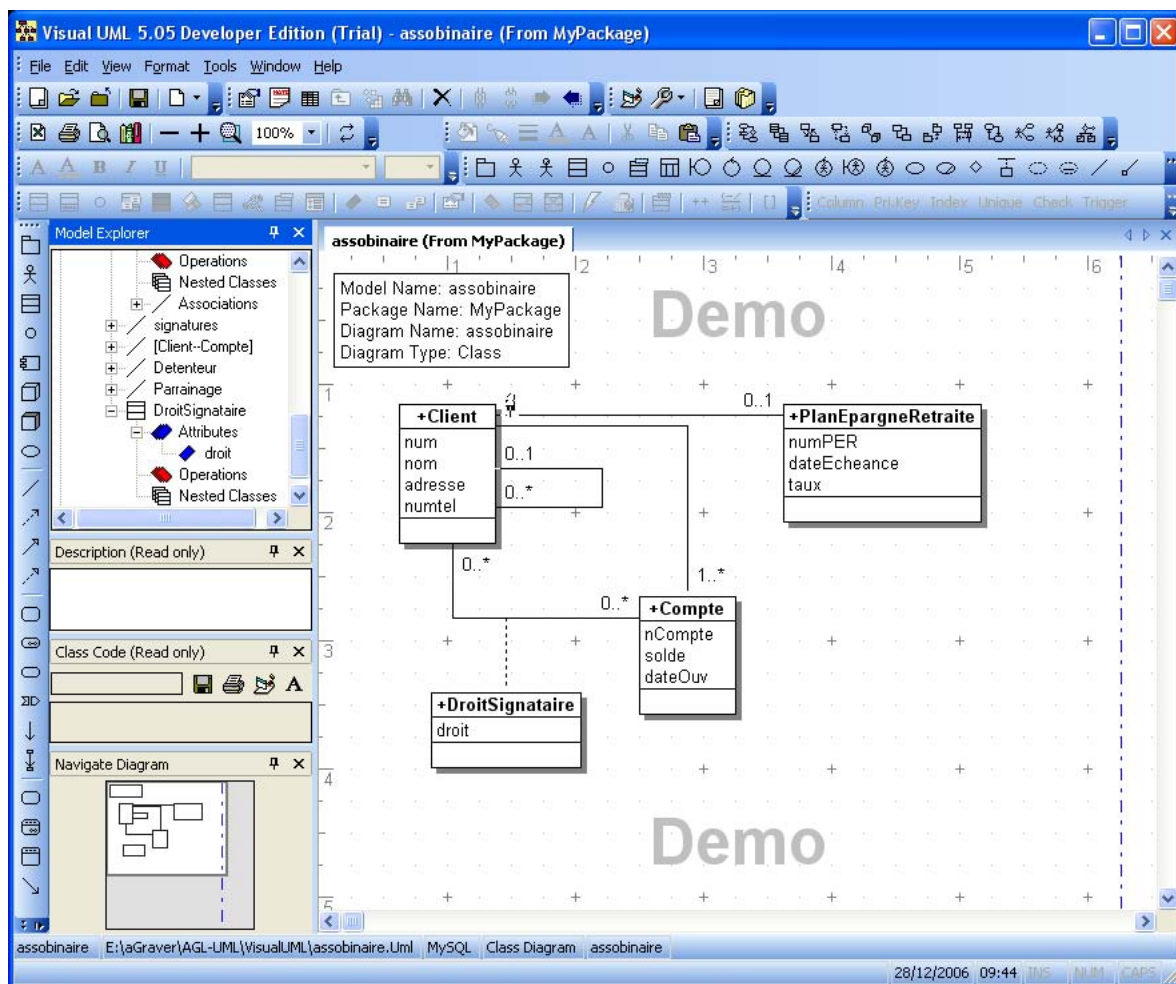
Figure 4-55 Visual Paradigm



Visual UML

Pas grand-chose à dire sur cet outil qui ne prend en compte que le niveau logique. Si vous êtes devenu adepte de la saisie manuelle des clés étrangères, rendez vos classes persistantes, allez chercher un pilote ODBC pour votre base de données et bon courage pour la suite.

Figure 4-56 Visual UML

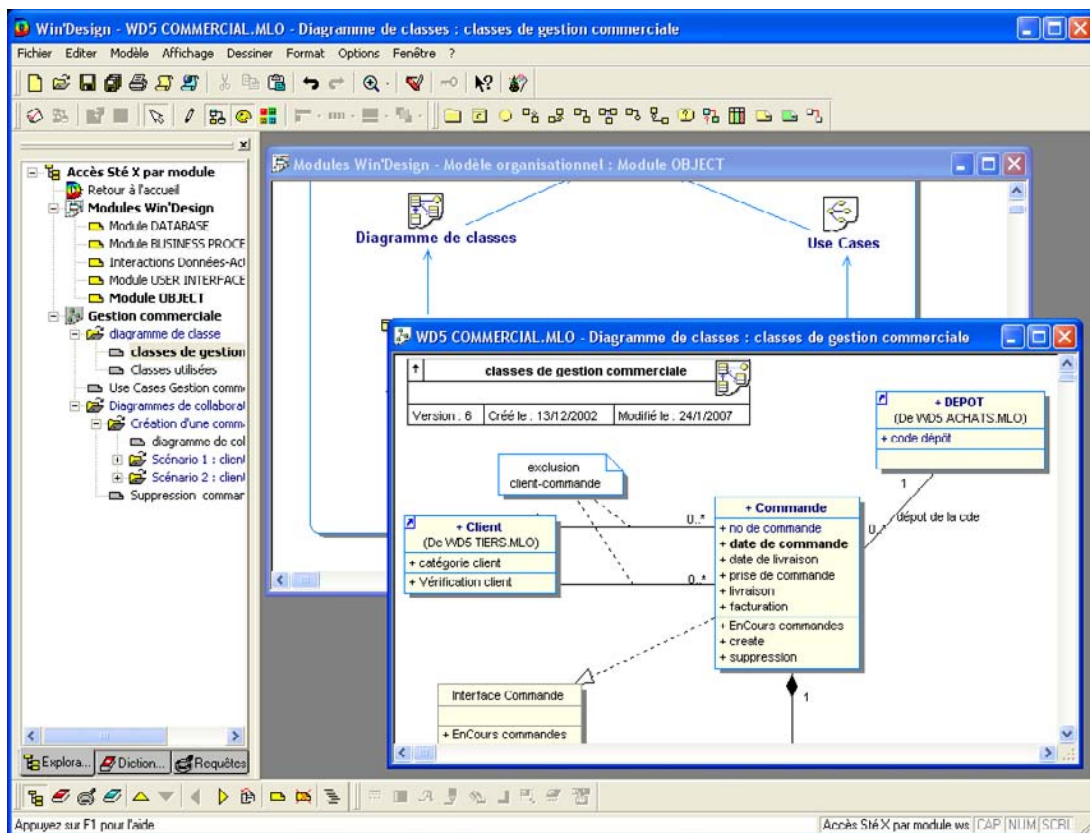


Win'Design

Win'Design est le produit de la société CECIMA basée à Aix en Provence. Il est présent sur le marché français depuis 1995. Développé initialement pour Merise/2, la notation UML arrive en 2002 avec la version 5. Depuis l'outil est en évolution constante.

La Gamme comprend quatre modules autonomes et complémentaires, qui s'articulent autour d'un référentiel (*Database* pour la conception et le reverse des bases de données, *Business Process* pour la modélisation des processus métier, *Object* pour la modélisation UML et *UserInterface* pour le maquetage des IHM). Vous devrez disposer du premier et du troisième module pour traduire des diagrammes de classes en script SQL. Comme PowerAMC, l'outil permet la double notation Merise/2 et UML 2 (sans le mode mixte de PowerAMC qui peut porter à confusion). Cet outil est le plus complet en ce qui concerne les contraintes Merise/2. Win'Design est probablement l'outil le plus facile à prendre en main et son interface est très intuitive.

Figure 4-57 Win'Design



Citons d'autres points forts : un très grand choix de SGBD est pris en compte, un grand nombre d'options de transformation de modèles (à tous les niveaux), la réactivité et l'efficacité du support. Notez que Win'Design est le seul à traiter correctement la transformation des associations n -aires et qu'avec PowerAMC et Rational, il est capable d'extraire un modèle conceptuel sans connexion à la base (à partir du seul script SQL de création des tables et contraintes). Le point faible de la version évaluée concerne la transformation de diagramme UML complexes (quelques cas spéciaux relevés lors de ces tests) en MLD et MCD. Les problèmes sont connus et en cours de correction.

Conclusion

Depuis 5 ans, le nombre d'outils a plus que doublé et les fonctionnalités deviennent de plus en plus puissantes. L'évolution d'UML semble se concrétiser car l'OMG a proposé en 2006 une RFP (*Request For Proposal*) nommée *Information Management Metamodel* dont le but est de définir un métamodèle incluant différentes sous-parties : *Common Warehouse Metamodel*, *UML2 Profile for Relational Data Modeling*, *UML2 Profile for Logical (Entity Relationship) Data Modeling*, *UML2 Profile for XML Data Modeling* et *UML2 Profile for Record Modeling (COBOL)*. De nouvelles spécifications à propos de la modélisation sont sans doute à attendre prochainement comme par exemple la notion d'identifiant de classe. Les prochaines versions des outils devraient s'enrichir de ces nouvelles fonctionnalités. Enfin, sachez qu'il existe un forum francophone sur <http://www.developpez.net> relatifs aux outils Rational Rose, PowerAMC, Together et Win'Design.

Annexe 1

URL utiles

Outils

EnterPrise Architect	http://www.sparxsystems.com.au/products/ea.html
MagicDraw UML	http://www.magicdraw.com/
MEGA Designer	http://www.mega.com/en/product/mega_designer/
ModelSphere	http://www.silverrun.com/modelsphere.html
MyEclipse	http://myeclipseide.com
Objecteering	http://www.objecteering.com/
Poseidon	http://gentleware.com/index.php?id=30
PowerAMC	http://www.sybase.com/products/developmentintegration/poweramc
Rational Rose	http://www-306.ibm.com/software/awdtools/developer/datamodeler/
Together	http://www.borland.com
Visio	http://www.microsoft.com/france/office/visio
Visual Paradigm	http://www.visual-paradigm.com/product/vpuml/productinfovpumlse.jsp
Visual UML	http://www.visualuml.com/Products.htm
Win'Design	http://www.win-design.com/fr/

UML

http://www.omg.org/technology/documents/modeling_spec_catalog.htm
<http://www.uml.org>
<http://uml.developpez.com/faq/>

Annexe 2

Bibliographie

- [ABR 74] J.R. ABRIAL, « Data Semantics », *Data Base Management*, North Holland, 1974.
- [ACS 90] ACSIOME, *Modélisation dans la conception des systèmes d'information*, Masson, 1990.
- [AST 76] M. ASTRAHAN *et al.*, « System R : Relational Approach to Database Management », *ACM Transactions on Database Systems*, Vol. 1, N° 2, 1976.
- [ABI 87] S. ABITEBOUL, R. HULL, « IFO : A Formal Semantic Database Model System », *ACM Transactions on Database Systems*, Vol. 12, N° 4, 1987.
- [ADI 93] M. ADIBA, C. COLLET, *Objets et Bases de données, le SGBD O2*, Hermès, 1993.
- [AKO 01] J.AKOKA, I. COMYN-WATIAU, *Conception des bases de données relationnelles*, Vuibert, 2001.
- [ANS 75] ANSI/X3/SPARC, « American National Standard Institute Study Group on DBMS : Interim report », *Bulletin of the ACM SIGMOD*, 1975.
- [ANS 74] W.W. ARMSTRONG, « Dependencies Structures of Data Base Relationships », *Proceedings IFIP*, 1974.
- [ARM 93] D. BEECH, « Collections of objects in SQL3 », *Proceedings of the Conference Very Large DataBases*, 1993.
- [AST 76] M. ASTRAHAN *et al.*, « System R : Relational Approach to Database Management », *ACM Transactions on Database Systems*, Vol. 1, N° 2, 1976.
- [ATK 89] M. ATKINSON, F. BANCILHON, D. DE WITT, K. DITTRICH, D. MAIER, S. ZDONIK, « The Object-Oriented database manifesto », *Proceedings of the Conference Deductive and Object-Oriented DataBases*, 1989.
- [BAC 69] C.W. BACHMAN, « Data structure diagrams », *Data Base Journal of the ACM SIGBDP*, Vol. 1, N° 2, 1969.
- [BEE 93] D. BEECH, « Collections of objects in SQL3 », *Proceedings of the Conference Very Large DataBases*, Dublin, 1993.

- [BEE 77] C. BEERI, R. FAGIN, J.H. HOWARD, « A Complete Axiomatization for Functional and Multivalued Dependencies », *Proceedings of the Conference ACM SIGMOD*, Toronto, 1977.
- [BER 76] P.A. BERNSTEIN, « Synthesizing Third Normal Form from Functional Dependencies », *ACM Transactions on Database Systems*, Vol. 1, N° 4, 1976.
- [BOO 91] G. BOOCH, *Object Oriented Design with Application*, Benjamin Cummings, 1991.
- [BOO 00] G. BOOCH, I JACOBSON, J. RUMBAUGH, *Le guide de l'utilisateur UML*, Eyrolles, 2000.
- [BOR 97] S. BOBROWSKI, *Oracle8 Architecture*, Oracle Press, 1997.
- [BOU 99] N. BOUDJILDA, *Bases de données et systèmes d'informations, le modèle relationnel : langages, systèmes et méthodes*, Dunod, 1999.
- [BRO 05] F. BROUARD, C. SOUTOU, *SQL*, Pearson Education, 2005.
- [CAT 91] R. G. G. CATTELL, *Object Data Management*, Addison-Wesley, 1991.
- [CAT 93] R. G. G. CATTELL, *Object Databases : The ODMG-93 Standard*, Morgan-Kaufman, 1993.
- [CAV 00] J.L. CAVARERO, R. LECAT, *La conception orientée objet, évidence ou fatalité*, Ellipses, 2000.
- [CHA 96] D. CHAMBERLIN, Using the new DB2 : IBM's object-relational database system, Morgan & Kaufman, 1996.
- [CHE 76] P.P. CHEN, « The Entity-Relationship Model : Towards a Unified View of Data », *ACM Transactions on Database Systems*, Vol. 1, N° 1, 1976.
- [CHR 87] C. CHRISMENT, *Mise en œuvre des bases de données*, Eyrolles, 1987.
- [COA 91] P. COAD, E. YOURDON, *Object Oriented Analysis*, Prentice Hall, 1991.
- [COD 69] E.F. CODD, « Derivability, Redundancy and Consistency of Relations Stored in Large Data Banks », *IBM research report RJ599*, 19 août 1969.
- [COD 70] E.F. CODD, « A Relational Model for Large Shared Data Banks », *Communications of the ACM*, Vol 13, N°6, 1970.
- [COD 72] E.F. CODD, « Further Normalization of the Data Base Relational Model », *Data Base Systems, Courant Computer Science Symposia Series 6*, Prentice Hall, 1972.
- [COD 74] E.F. CODD, « Recent Investigations into Relational Database Systems », *Proceedings of the IFIP*, Stockholm, 1974.
- [CON 05] T. CONNOLY, C. BEGG, *Systèmes de bases de données*, Reynald Goulet-Eyrolles, 2005.
- [COP 84] G. COPELAND, D. MAIER, « Making Smalltalk a database system », *Proceedings of the Conference ACM SIGMOD*, Boston, 1984.

- [DAH 66] O.DAHL, K. NYGAARD, « An algol-based simulation langage », *Communications of the ACM*, Vol. 9, N° 9, 1966.
- [DAT 96] C.J. DATE, *An Introduction to Database Systems*, Addison-Wesley, 6th edition, 1996.
- [DAT 98] C.J. DATE, H. DARWEN, *Foundation for Object/Relational Databases, The Third Manifesto*, Addison-Wesley, 1998.
- [DAT 00] C.J. DATE, *Introduction aux bases de données*, International Thomson Publishing, traduction de la septième édition de l'ouvrage d'Addison-Wesley, 2000.
- [DEL 82] C. DELOBEL, M. ADIBA, *Bases de données et systèmes relationnels*, Dunod, 1982.
- [DEL 91] C. DELOBEL, C. LÉCLUSE, P. RICHARD, *Bases de données : des systèmes relationnels aux systèmes à objets*, InterEditions, 1991.
- [DEL 94] C. DELOBEL, « Convergence and/or divergence between SQL3 standard evolution and ODMG-93 standard for object-oriented database systems », *Proceedings of the Basque International Workshop on Information Technology (BIWIT'94)*, 1994.
- [DEL 00] P. DELMAL, *SQL2-SQL3, Applications à Oracle*, De Boeck Université, 2000.
- [DIO 94] D. DIONISI, *L'essentiel sur Merise*, Eyrolles, 2^e édition, 1998.
- [DOR 99] P. DORSEY, J. R. HUDICKA, *Oracle8, Design Using UML Object Modeling*, Oracle Press, 1999.
- [ELM 04] R. ELMASRI, S. NAVATHE, *Conception et architecture des bases de données*, Pearson Education, 2004.
- [FRE 93] P. FRESNAIS, « Comparaison des SGBD relationnels et orientés objets à travers une étude de cas », *Actes du congrès biennal de l'AFCET*, Versailles, 1993.
- [GAR 94] G. GARDARIN, « Translating relational to object databases », *Ingénierie des systèmes d'information*, Vol. 2, N° 3, Hermès, 1994.
- [GRI 98a] S. GRIMES, « Modeling Object Relational Databases », *Object-Relational Database Summit*, <http://www.dbsummit/or/grimes.html>, septembre 1998.
- [GRI 98b] S. GRIMES, « Object Relational Reality Check », *Database Programming and Design on line*, <http://www.dbpd.com/9807feat.htm>, 1998.
- [HAI 05] J.L. HAINAULT, *Bases de données et modèles de calculs*, Dunod, 2005.
- [HAL 01] T. HALPIN, *Information Modeling and Relational Databases*, Mogan Kaufmann, 2001.
- [ISO 92] ISO/IEC 9075:1992, «Database Langage SQL ».
- [JAC 92] I. JACOBSON, M. CHRISTERSON, P. JONSSON, G. OVERGAARD, *Object-Oriented Software Engineering : a Use Case Driven Approach*, Addison-Wesley, 1992.

- [KET 98] N. KETTANI, D. MIGNET, P. PARE, C. ROSENTHAL-SABROUX, *De Merise à UML*, Eyrolles, 1998.
- [KIM 97] W. KIM, « Bringing Object Relational Down to Earth », *Database Programming and Design on line*, <http://www.dbpd.com/9707kim.htm>, 1997.
- [LAR 06] N. LAROUSSE, *Création de bases de données*, Pearson Education, 2006.
- [LOH 91] G. LOHMAN *et al.*, « Extension to Starbust : Object, Types, Functions and Rules », *Communications of the ACM*, Vol 34, N°10, 1991.
- [MAK 77] A. MAKINOUCI, « A consideration of normal form of not necessarily-normalized relation in the relational model of data », *Proceedings of the Conference Very Large DataBases*, 1977.
- [MAR 88] J.P. MARTHERON, *Comprendre Merise*, Eyrolles, 1988.
- [MAR 94] C. MAREE, G. LEDANT, *SQL 2 Initiation Programmation*, Armand Colin, 1994.
- [MAR 98] J. MARTIN, J. ODELL, *Object Oriented Methods : A Foundation, UML Edition*, Prentice Hall, 1998.
- [MAT 00] P. MATHIEU, *Des bases de données à l'Internet*, Vuibert, 2000.
- [MEL 93] J. MELTON, A.R. SIMON, *Understanding the new SQL : A complete guide*, Morgan-Kaufman, 1993.
- [MEY 90] B. MEYER, *Conception et programmation par objets*, InterÉditions, 1990.
- [MIR 94] S. MIRANDA, A. RUOLS, *Client-Serveur, moteurs SQL, middleware et architectures parallèles*, Eyrolles, 1994.
- [MOU 76] P. MOULIN, J. RANDON, S. SAVOYSKY, S. SPACCAPIETRA, H. TARDIEU, M. TEBOUL, « Conceptual model as database design tool », *Proceedings of the IFIP Working conference on Modelling in Database Management Systems*, G.M. Nijssen Ed., North-Holland, 1976.
- [MOR 92] J. MOREJON, *Principes et conception d'une base de données relationnelle*, Collection Ingénierie des systèmes d'information, Les Éditions d'Organisation, 1992.
- [MUL 00] P.A. MULLER, N. GAERTNER, *Modélisation objet avec UML*, Eyrolles, 2^e édition, 2000.
- [MUL 99] R.J. MULLER, *Database Design for Smarties, Using UML for Data Modeling*, Morgan-Kaufman, 1999.
- [NAI 01] E. NAIBURG, R. MAKSIMCHUK, *UML for database design*, Addison-Wesley, 2001.
- [NAM 74] Rapport introductif *Modèles de structure de données dans les systèmes d'information*, Séminaire international, Namur 1974.
- [NAN 01] D. NANJI, B. ESPINASSE, B. COHEN, *Ingénierie des systèmes d'information : Merise deuxième génération*, Vuibert, 4^e édition, 2001.

- [OMG 97] OBJECT MANAGEMENT GROUP, « The Unified Modeling Language », *Release 1.1 Reference Manual*, 1997.
- [PAN 94] G. PANET, R. LETOUCHE, *Merise/2 : modèles et techniques Merise avancés*, Les Éditions d'Organisation, 1994.
- [PIC 90] E. PICHAT, R. BODIN, *Ingénierie des données*, Masson, 1990.
- [ROC 93] A. ROCHFELD, M. BOUZEGHOUB, « From Merise to OOM », *Ingénierie des systèmes d'information*, Vol. 2, N° 1, Hermès, 1993.
- [ROQ 06] P. ROQUES, *UML2 par la pratique*, Eyrolles, 2006.
- [RUM 91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, *Object Oriented Modeling and Design*, Prentice Hall, 1991.
- [RUM 95] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, W. LORENSEN, *Modélisation et conception orientées objet*, Masson, 1995.
- [SAR 98] C. M. SARACCO, *Universal Database Management, A Guide to Object/Relational Technology*, Morgan-Kaufman, 1998.
- [SCH 88] S. SCHLAER, S.J. MELLOR, *Object Oriented Systems Analysis : Modeling the World in Data*, Prentice Hall, 1988.
- [SIM 05] G. SIMSION, G. WITT, *Data Modeling Essentials*, Mogan Kaufmann, 2005.
- [SMI 77] H.A. SMITH, D.C.P. SMITH, «Database Abstractions : Aggregation and Generalization», *ACM Transactions on Database Systems*, Vol. 2, N° 2, 1977.
- [SPA 93] S. SPACCAPIETRA, C. PARENT, « ERC+ : an Object Based Entity Relationship Approach », dans *Conceptual Modeling, Databases and CASE : An Integrated View of Information Systems Development*, Wiley, 1993.
- [SOU 98] C. SOUTOU, « Relational Database Reverse Engineering Extraction of Cardinality Constraints », *Data and Knowledge Engineering*, Vol. 28, N° 2, Elsevier Science, North-Holland, 1998.
- [SOU 98b] C. SOUTOU, « Inference of Aggregate Relationships through Database Reverse Engineering », *Proceedings of the 17th International Conference on Conceptual Modeling (ER'98)*, Lecture Note in Computer Science, Editions Springer Verlag, volume 1507, pages 135-149.
- [SOU 99] C. SOUTOU, *Objet-relationnel sous Oracle8 - Modélisation avec UML*, Eyrolles, 1999.
- [SOU 01] C. SOUTOU, « Modeling Relationships in Object-relational Databases », *Data and Knowledge Engineering*, Vol. 36, N° 1, Elsevier Science, North-Holland, 2001.
- [SOU 04] C. SOUTOU, *Programmer Objet avec Oracle*, Vuibert, 2004.

- [STE 98] J. STEIN, D. MAIER, « Concepts in Object-Oriented Data Management », *Database Programming and Design*, Vol. 1, N° 4, 1998.
- [STE 01] R. STEPHENS, R. PLEW, *Conception de bases de données*, Campus Press, 2001.
- [STO 76] M. STONEBRAKER, E. WONG, P. KREPS, G. HELD, « The Design and Implementation of Ingres », *ACM Transaction on Database Systems*, Vol. 1, N° 3, 1976.
- [STO 86] M. STONEBRAKER, L.A. ROWE, « The design of Postgres », *Proceedings of the Conference ACM SIGMOD*, San Diego, 1986.
- [STO 96] M. STONEBRAKER, D. MOORE, *Object Relational DBMSs, the next great wave*, Morgan-Kaufman, 1996.
- [TAR 79] H. TARDIEU, D. NANJI, D. PASCOT, « A Method, A Formalism and Tools for Database Design (three years of Experimental Practice) », *Proceedings of the International Conference on Entity-Relationship Approach to Systems Analysis and Design*, 1979.
- [TAR 79b] H. TARDIEU, D. NANJI, D. PASCOT, *Conception d'un système d'information, Conception de la base de données*, Les Éditions d'Organisation, 1979.
- [TAR 83] H. TARDIEU, A. ROCHFELD, R. COLLETI, *La méthode MERISE tome 1*, Les Éditions d'Organisation, 1983.
- [TAR 91] H. TARDIEU, A. ROCHFELD, R. COLLETI, *La méthode MERISE tome 1*, Eyrolles, 1991.
- [TEO 94] T.J. TEOREY, *Database Modeling and Design : The Fundamental Principles*, Ed. Morgan-Kaufmann, 1994.
- [VAL 87] P. VALDURIEZ, « Objets complexes dans les systèmes de bases de données relationnels », *Techniques et science informatique*, Vol. 6, N° 5, 1987.
- [WON 78] E. WONG, K. YOUSSEFI, « Decomposition Strategy for Query Processing », *ACM Transactions on Database Systems*, Vol. 1, N° 3, 1978.

A

- agrégation
 - modèle entité-association 60, 61
 - UML 64, 209
- agrégation plusieurs-à-plusieurs
 - SQL2 214
- agrégation plusieurs-à-un
 - modèle entité-association 149
 - SQL2 213
- agrégation un-à-plusieurs
 - modèle entité-association 147
 - SQL2 211
- agrégation un-à-un
 - modèle entité-association 150
- allInstances 79
- ALTER TABLE 207
- AMC*Designor 289
- association 22
 - arité 24
 - degré 24
 - dérivée 43
 - navigable 44
 - nommée
 - Voir UML
- association d'agrégation 143
 - plusieurs-à-plusieurs 145
 - plusieurs-à-un 149
 - un-à-plusieurs 147
 - un-à-un 150
 - Voir agrégation
- association *n*-aire 36
 - affinage 55
 - décomposition 72
 - formalisme de Chen 38
 - Merise 37
 - modèle conceptuel 36
 - modèle objet 139, 161
 - modèle relationnel 129
 - réflexive 43
 - SQL2 193
 - SQL3 229
 - UML 39
- association plusieurs-à-plusieurs
 - modèle entité-association 34, 35
 - modèle objet 138
 - modèle relationnel 129
 - SQL2 192
 - SQL3 228
 - UML 35, 36
- association qualifiée 43
- association réflexive 41
 - modèle conceptuel 41
 - modèle entité-association 41
 - SQL2 195
 - SQL3 232
- association un-à-plusieurs
 - modèle entité-association 32
 - modèle objet 158
 - modèle relationnel 129
 - SQL2 191
 - SQL3 225
 - UML 32

- association un-à-un 29
 - modèle conceptuel 29
 - modèle entité-association 30
 - modèle objet 141, 157
 - modèle relationnel 131
 - SQL2 190
 - SQL3 224
 - UML 30
- attribut 21, 85, 104
 - monovalué 67
 - multivalué 67
- attribut dérivé 85
- augmentation 109

B

- backup 10
- bag 54
- base de données 1
- BLOB 12
- BPEL4WS 247
- BPM 247

C

- cardinalités 24
- CASCADE 188
- champs 106
- CHECK 207
- CIF 29, 38
- classe 26
- classe-association
 - SQL2 192
 - SQL3 234
 - Voir UML
- clé
 - candidate 106
 - étrangère 10, 106
 - primaire 10, 106
- Codd E. 7
- collection 24

- colonnes 106
- complete 79
- CONSTRAINT 179
- contrainte d'exclusivité 48
 - SQL2 218
- contrainte d'inclusion 58
 - modèle conceptuel 50
 - SQL2 219
- contrainte d'unicité 56
- contrainte de partition
 - modèle conceptuel 45
 - SQL2 217
- contrainte de simultanéité
 - modèle conceptuel 53
 - SQL2 219
- contrainte de totalité
 - modèle conceptuel 49
 - SQL2 218
- contrainte référentielle 10
- couverture minimale 109
- CREATE TABLE 179
- CREATE TYPE 222

D

- déclencheur 202
- décomposition 109
- degré 104
- DELETE 181
- DELETE CASCADE 209
- dépendance de jointure 112
- dépendance fonctionnelle
 - définition 107
 - dépendant 107, 109
 - déterminant 107, 109, 117
 - directe 108
 - élémentaire 108
 - formes normales 67, 69, 70
 - multivaluée 111
 - union 109
- desc 223
- dimension 24

DISABLE CONSTRAINT 197
disjoint 79
domaine 104, 187
DROP CONSTRAINT 208
DROP TABLE 180

E

encapsulation 83
Enterprise Architect 278
entité 21
 abstraite 133
 faible 59, 88
 sous-type 76
 sur-type 76
entité-association 20
exclusivité 48
extension 104

F

fermeture 109
FOREIGN KEY 179
forme normale
 Boyce-Codd 117
 cinquième 119
 deuxième 69, 115
 première 67, 113
 quatrième 118
 troisième 70, 116

G

GRANT 184

H

héritage 75
 avec partition 78
 avec totalité 80
 classification 76, 202
 de tables 242
 de types 241
 distinction 202

multiple 82, 134, 201
niveau logique 131, 142
push-down 132, 142, 199, 206
push-up 133, 200, 207
SQL2 198
SQL3 240

I

identifiant 21
 alternatif 88
 composé 86
 relatif 87, 89
 simple 86
identification
 absolue 87
 d'une association 87
incomplete 80
INSERT 181
intégrité des données 187
intégrité référentielle 188
intention 104
isUnique 79

J

jointure 183

M

MagicDraw 280
mapping 14
MCD 21
MDA 247
MEGA 283
Merise 21
 association 22
 association n-aire 37
 association réflexive 41
 attribut 21
 cardinalités 24
 entité 21
 identifiant 21
 occurrence 23

Merise/2 21
 exclusivité 48
 héritage 75
 héritage multiple 82
 inclusion 50, 219
 partition 46
 totalité 49
méthode 84
modèle conceptuel 19
modèle entité-association 20
modèle objet
 association n-aire 161
 association un-à-plusieurs 158
 association un-à-un 157
 contrainte d'inclusion 165
 contrainte d'unicité 165
modèle relationnel
 cinquième forme normale 119
 deuxième forme normale 115
 forme normale de Boyce-Codd 117
 première forme normale 113
 quatrième forme normale 118
 troisième forme normale 116
ModelSphere 285
multiplicités 27
MyEclipse 285

N

NF2 12, 114
NIAM 19
niveau conceptuel 19
niveau logique 103
niveau physique 177
normalisation 107, 113
 par décomposition 119
 par synthèse 122
NOT NULL 190
n-uplet 104, 113

O

Objectteering 287

occurrence 23
OCL 46, 55
oclIsKindOf 79
Oadapter 13
OID 223
ordered 54
overlapping 80

P

Poseidon 288
PowerAMC 289
PowerDesigner 289
PRIMARY KEY 179
procédure cataloguée 204
profil UML 3, 290
pseudo-entité 58
pseudo-transitivité 109
push-down 131
push-up 131

Q

QVT 247

R

recovery 10
redefined 54
REF 222
référence inverse 159
réflexivité 109
règles de gestion 187
réification 60, 89
relation 104
 degré 104
 extension 104
 intention 104
relation universelle 119
requête 182
REVOKE 184
RFP 299

S

- schéma conceptuel 19
- SCOPE IS 224
- sequence 54
- SQL 178
 - contrôle des données 184
 - définition des données 179
 - interrogation des données 182
 - manipulation des données 181
- SQL2 178
- SQL3 178
- stéréotype 124
 - Voir* UML
- subsets 52, 54
- substantification 89

T

- table
 - objet-relationnelle 223
 - relationnelle 106
- Together 292
- transitivité 109
- tuple 104
- type 222

U

- UML 26
 - agrégation 64
 - association 27
 - n*-aire 39
 - navigable 44
 - nommée 31
 - réflexive 42
 - attribut 26
 - classe 26
 - classe-association 36
 - composition 64

- contrainte
 - d'exclusivité 48
 - d'inclusion 52, 59
 - d'unicité 56
 - de partition 46
 - de simultanéité 54
 - de totalité 49
- héritage 75, 78
 - multiple 82
- méthode 26
- multiplicités 27
- privé 85
- profil 3, 290
- protégé 85
- public 85
- rôle 33
- stéréotype 40
- visibilité 85
- UNDER 240
- union 54
- UNIQUE 190
- UPDATE 181
- UPDATE CASCADE 209

V

- visibilité 84
- Visio 294
- Visual Paradigm 295
- Visual UML 297

W

- WHERE 181
- Win'Design 298

X

- xor 46, 54