Christian Soutou

UML 2
pour les bases de données

2e édition

 ISBN : 978-2-212-13413-1

EYROLLES

Table des matières

Avant-propos

Le but de cet ouvrage est d'expliquer tout d'abord comment utiliser à bon escient le diagramme de classes UML pour concevoir une base de données, puis comment maîtriser la traduction de ce diagramme en script SQL permettant de générer des tables normalisées. La démarche proposée dans ce livre est indépendante de tout éditeur de logiciel et aisément transposable quel que soit l'outil de conception que vous adopterez.

Entièrement réécrite, cette deuxième édition est émaillée de nombreux cas concrets présentés sous forme d'exercices. Le texte est par moments commenté par Frédéric Brouard alias SQLPro, consultant indépendant et expert en bases de données (MS SQL Server en particulier).

Évolution des modèles de données

Avant de rentrer dans le vif du sujet, rappelons brièvement comment nous sommes arrivés à des bases de données essentiellement relationnelles. Avant elles, l'informatique existait bien

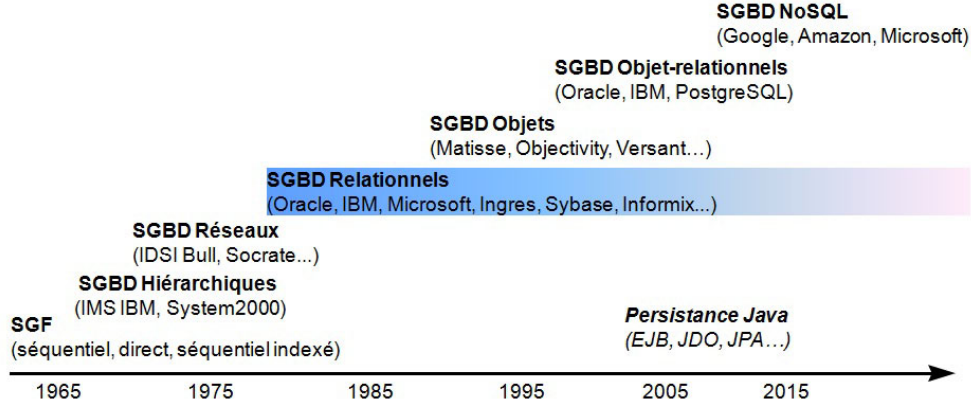

***Figure 0-1**. Historique des bases de données*

sûr ; Apollo 11 a été envoyé sur la lune en juillet 1969 avant qu'E. Codd publie ses écrits sur le modèle relationnel. À l'époque, la conception des données se faisait avec bon sens, depuis, les chercheurs ont amené des cadres plus formels. À vous de toujours conserver votre bon sens, tout en utilisant ces cadres.

Les fichiers et COBOL

Le stockage des données a commencé dans les années 1960 avec les systèmes de gestion de fichiers et le langage COBOL (*Common Business Oriented Language*). Loin d'être dépassé, ce dernier fut le plus utilisé entre 1960 et 1980. En 2002, il permet une programmation de type objet, la gestion des informations Unicode et une intégration avec XML. En 2005, le Gartner Group estimait que COBOL manipulait près de 75 % des données de gestion stockées.

Le principal inconvénient des applications COBOL est la forte dépendance qui existe entre les données stockées et les traitements. En effet, le fait de déclarer dans chaque programme les fichiers utilisés impose une maintenance lourde si la structure d'un fichier doit être modifiée. De plus, les instructions de manipulation (ouverture, lecture, écriture et modification) sont très liées à la structure de chaque fichier. La structure des fichiers de données s'apparente à celle d'une table (suite de champs de types numériques ou alphanumériques).

Le modèle hiérarchique

Les bases de données hiérarchiques ont introduit un modèle de données du même nom. Il s'agit de déterminer une arborescence de données où l'accès à un enregistrement de niveau inférieur n'est pas possible sans passer par le niveau supérieur. Promus par IBM et toujours utilisés dans le domaine bancaire, les SGBD hiérarchiques souffrent toutefois de nombreux inconvénients.

La figure suivante illustre un modèle hiérarchique de données dans lequel des compagnies aériennes peuvent embaucher plusieurs pilotes. Un pilote peut travailler pour le compte de différentes compagnies.

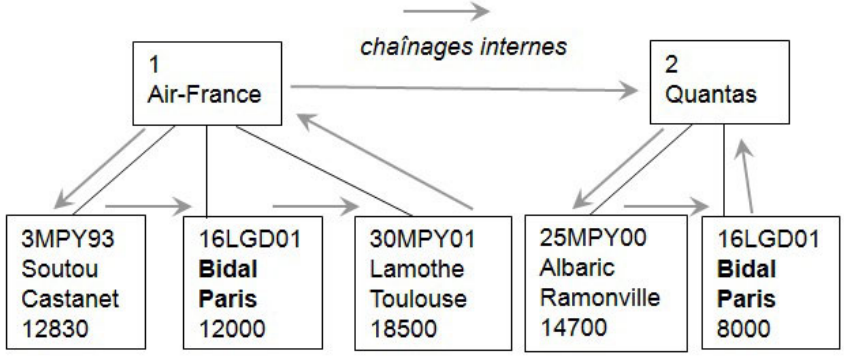

Figure 0-2. *Modèle de données hiérarchique*

Les inconvénients récurrents sont toujours la forte dépendance entre les données stockées et les méthodes d'accès. Les chaînages internes impliquent forcément une programmation complexe. Outre ces problèmes de programmation, ce modèle montre des lacunes lors de l'accès à la base.

- Extraire la liste des pilotes implique le parcours de toutes les compagnies.
- L'insertion peut se révéler problématique : l'ajout d'un pilote sans compagnie n'est pas possible, à moins d'ajouter une compagnie fictive.
- La suppression peut se révéler dangereuse : une compagnie disparaît, alors de fait ses pilotes aussi.
- La modification est souvent problématique : les incohérences proviennent d'éventuelles redondances (le nom ou l'adresse d'un pilote qui change doit se répercuter à tous les enregistrements).

Bien qu'il existe de nombreuses hiérarchies autour de nous, le monde qui nous entoure n'est pas un arbre !

Le modèle réseau

Quelques années plus tard, C. W. Bachman, pionnier dans le domaine de l'informatique, s'est essayé aux bases de données en inventant un modèle brisant cette hiérarchie plutôt arbitraire. Les bases de données réseau étaient nées avec le modèle CODASYL, première norme décidée sans IBM.

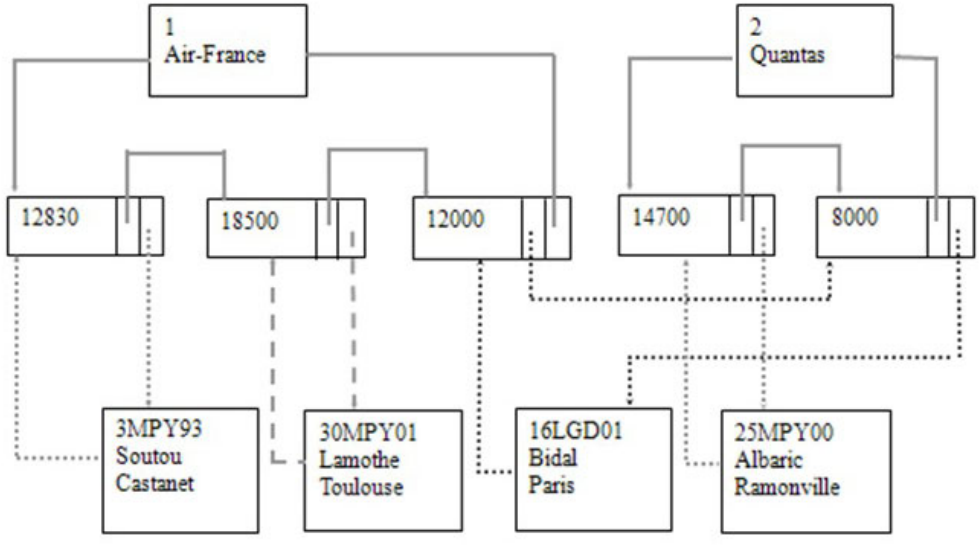

Figure 0-3. *Modèle de données réseau*

Bien que résolvant quelques limitations du modèle hiérarchique et annonçant des performances en lecture honorables, le modèle réseau n'est ni plus ni moins qu'une usine à gaz gavée de pointeurs. Pour preuve, plus personne n'utilise de tels SGBD où la dépendance entre les données stockées et les méthodes d'accès existe toujours, et l'évolution d'une base de données est très coûteuse en termes de recompilation de pointeurs.

Soyons honnêtes, le monde ressemble bien à une telle usine à gaz ! Mais pas question de stocker ainsi les données, ce serait bien trop compliqué de concevoir le bon graphe. Le modèle de données se doit d'être plus simple.

Le modèle relationnel

En 1970, E. Codd publie l'article de référence posant les bases du modèle relationnel [COD 70]. D'un seul coup, toutes les limitations des précédents modèles sont résolues. Le but initial de ce modèle était d'améliorer l'indépendance entre les données et les traitements. Cet aspect des choses est réussi et avec ça d'autres fonctionnalités apparaissent :

- Normalisation (dépendances fonctionnelles) et théorie des ensembles (algèbre relationnelle).
- Cohérence des données (intégrité référentielle).
- Langage SQL (déclaratif et normalisé).
- Accès aux données optimisé (choix du chemin par le SGBD).
- Indexation, etc.

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 ».

REMUNERATION

BREVET	NCOMP	PAYE
3MPY93	1	12830
16LDG01	1	12000
30MPY01	1	18500
25MPY00	2	14700
16LDG01	2	800

COMPAGNIE

NCOMP	NOMCOMP
1	Air-France
2	Quantas

PILOTE

BREVET	NOM	ADRESSE
3MPY93	Soutou	Castanet
16LDG01	Bidal	Paris
30MPY01	Lamothe	Toulouse
25MPY00	Albaric	Ramonville

Figure 0-4. *Modèle de données relationnel*

La force de ce modèle de données réside dans le fait qu'il repose sur des principes simples et permet de modéliser des données complexes. Le modèle relationnel est à l'origine du succès que connaissent aujourd'hui les grands éditeurs de SGBD, à savoir Oracle, IBM, Microsoft et Sybase dans différents domaines :

- OLTP *(OnLine Transaction Processing)* où les mises à jour des données sont fréquentes, les accès concurrents et les transactions nécessaires.
- OLAP *(Online Analytical Processing)* où les données sont multidimensionnelles (cubes), les analyses complexes et l'informatique décisionnelle.
- Systèmes d'information géographiques (SIG) où la majorité des données sont exprimées en 2D ou 3D et suivent des variations temporelles.

Comment concevoir de telles relations ? C'est ce que nous allons voir tout au long de cet ouvrage…

À qui s'adresse cet ouvrage ?

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

- Les architectes, chefs de projet, analystes, développeurs et responsables méthode habitués au modèle entité-association y trouveront les moyens de raisonner avec le diagramme de classes UML.
- Les novices découvriront une méthode de conception, des règles de normalisation et de nombreux exercices mettant en jeu tous les niveaux du processus d'une base de données.

Comment utiliser UML pour les bases de données ?

Depuis plus de 30 ans, la conception des bases de données s'appuie sur un formalisme graphique appelé entité-association que la méthode Merise avait adopté en son temps. Ce formalisme a fait ses preuves et bon nombre d'outils de modélisation destinés aux francophones 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). Les entreprises du consortium initial ayant mis en place UML étaient DEC, HP, IBM, Microsoft, Oracle et Unisys pour parler des plus connues. Le marché a suivi cette tendance car, aujourd'hui, tous les outils de modélisation utilisent cette notation.

L'adoption généralisée de la notation UML dépasse le simple effet de mode. La majorité des nouveaux projets industriels utilisent la notation UML. Tous les cursus universitaires, qu'ils soient théoriques ou plus techniques, incluent l'étude d'UML. 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 mai 2010, est la 2.3 (http://www.omg.org/spec/UML/2.3/). Ce succès s'explique aussi par l'adoption unanime des concepts objet, qui ont des avantages indéniables (réutilisabilité de composants logiciels, facilité de maintenance, prototypage et extension des applications, etc.).

Les diagrammes

Les versions 1.*x* de la notation UML définissent neuf diagrammes : cinq pour les aspects statiques (classes, objets, cas d'utilisation, composants et déploiement) et quatre pour les aspects dynamiques (séquence, collaboration, états-transitions, activités). Les spécifications d'UML 2.*x* ajoutent le diagramme d'interaction, le diagramme de structure composite et le diagramme temporel. Seul le diagramme de classes est intéressant à utiliser pour la modélisation d'une base de données.

UML concerne en premier lieu le développement logiciel et n'a pas été initialement pensé pour les bases de données. La notation UML permet toutefois d'offrir un formalisme aux concepteurs d'objets métier et aux concepteurs de bases de données. D'autre part, les concepts relatifs à la modélisation de données (entités, associations, attributs et identifiants) peuvent être parfaitement intégrés aux diagrammes de classes. De plus, d'autres concepts (notamment les classes-associations, agrégats et contraintes) permettent d'enrichir un schéma conceptuel.

Les outils

De nombreux outils informatiques basés sur la notation UML existent depuis quelques années. Les plus sophistiqués permettent de générer des modèles logiques ou des scripts SQL. Alors que l'automatisation est quasiment assurée (sous réserve de la qualité de l'outil) entre le modèle conceptuel et la base de données, il n'en est pas de même de l'élaboration du diagramme initial qui va conditionner toute la suite. Ici l'humain est au centre de tout et il n'est pas question de penser que cette tâche puisse être automatisée (c'est heureux pour les concepteurs).

Par ailleurs, il est fort probable que les scripts SQL générés devront 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, types de données...), soit tout simplement parce que le concepteur préfère utiliser une autre possibilité d'implémentation pour traduire telle ou telle autre association.

Il est donc préférable de maîtriser les concepts, de comprendre les mécanismes de transformation de modèles et d'adopter une démarche afin d'utiliser l'outil de manière optimale.

Cet ouvrage vous permettra, je l'espère, de mieux appréhender le cheminement de la conception vers le codage en donnant des règles précises à suivre dans l'élaboration des différents modèles pour éviter de graves erreurs au niveau de la base. Le script SQL fera office de véritable révélateur.

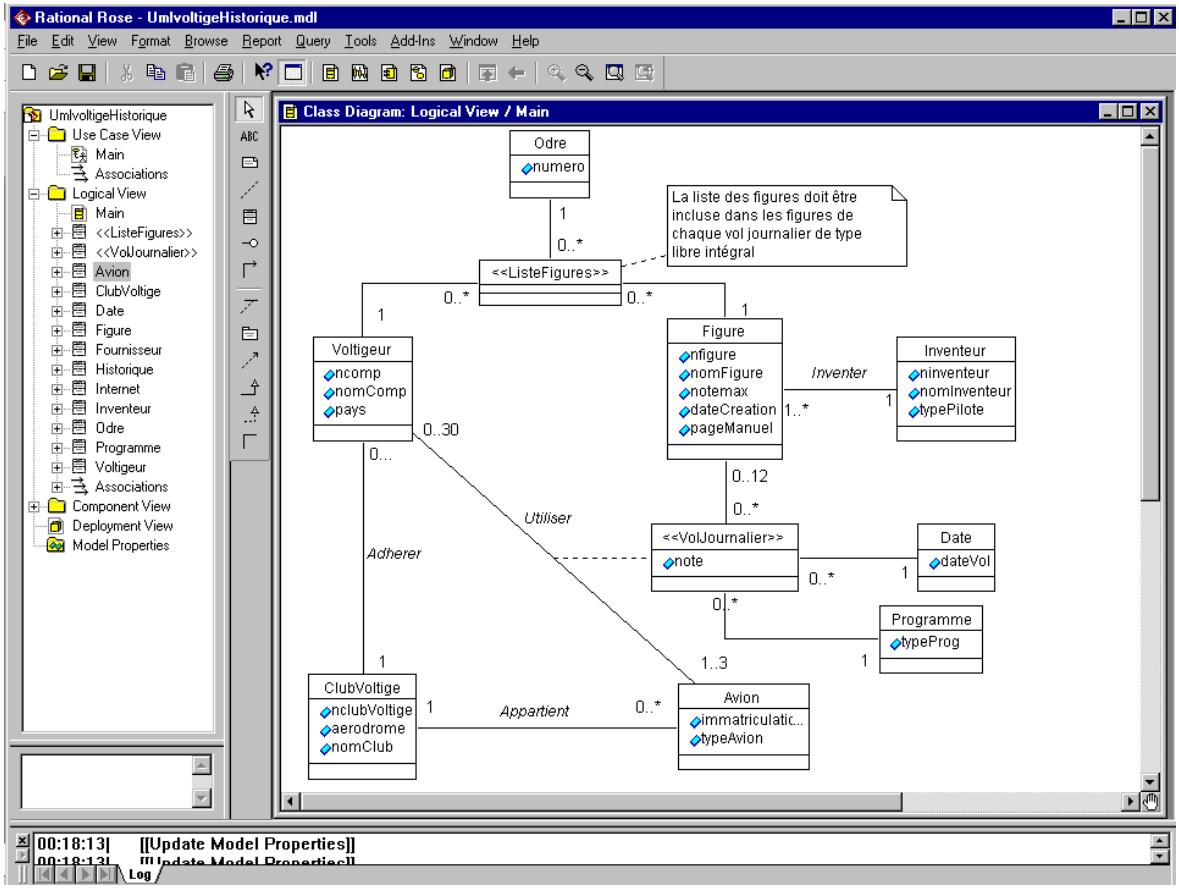

Figure 0-5. *Diagramme de classes*

Guide de lecture

Cet ouvrage s'organise en 5 chapitres qui suivent les étapes de modélisation illustrées dans la figure suivante.

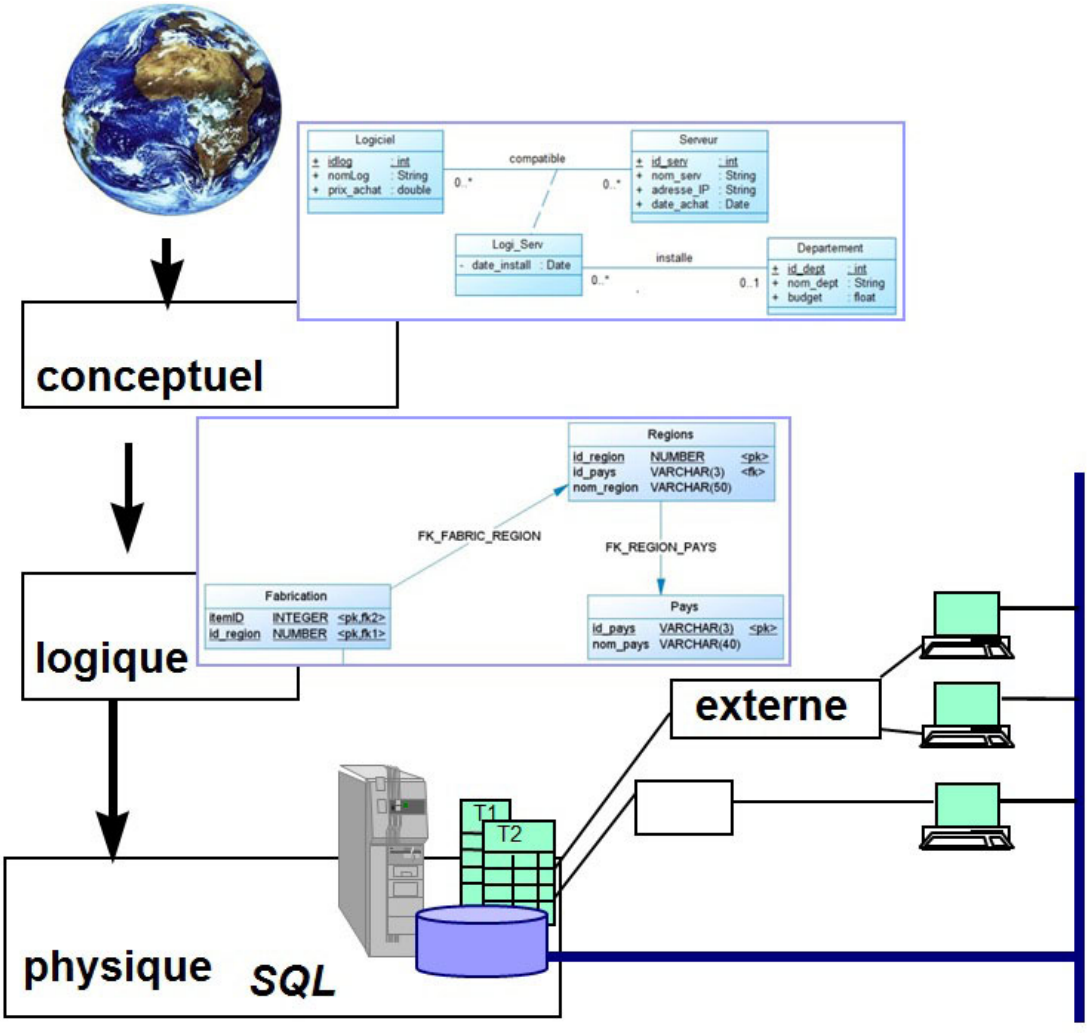

Figure 0-6. *Niveaux de conception et d'implémentation*

L'ouvrage commence par décrire la construction d'un diagramme de classes UML en respectant des règles qui permettent de le valider et de le normaliser. Les mécanismes de dérivation d'un modèle conceptuel dans un schéma de données relationnel sont clairement expliqués à l'aide d'exemples concrets. Le modèle logique est ensuite optimisé avant l'écriture des scripts SQL. Il s'agit ensuite d'implémenter les règles métier en programmant des contraintes ou des déclencheurs SQL. La dernière étape consiste à préparer des vues qui composeront l'interface de la base aux utilisateurs extérieurs.

Niveau conceptuel

Le chapitre 1 décrit la première étape du processus de conception d'une base de données, à savoir la construction d'un schéma conceptuel. Le formalisme graphique préconisé est le diagramme de classes de la notation UML qui permet des équivalences des modèles de type entité-association.

Transformation et normalisation

Le chapitre 2 décrit les concepts du modèle relationnel, puis présente les règles qui permettent de dériver un schéma logique à partir d'un modèle conceptuel. La dernière partie traite de la normalisation et du calcul de volumétrie.

Écriture des scripts SQL et programmation des contraintes

Le chapitre 3 décrit la mise en œuvre pour écrire un script SQL dérivé d'un modèle de données relationnel. Le niveau physique présenté dans ce chapitre correspond à la définition des tables, des clés étrangères ainsi que l'implémentation des éventuelles règles métier par des contraintes de clés, de validation ou par déclencheurs.

Les vues SQL2 et SQL3

Le niveau externe décrit au chapitre 4 correspond à la définition de vues qui agissent comme des fenêtres sur la base de données. Ce chapitre décrit les différents types de vues existantes (vues relationnelles SQL2, vues matérialisées et vues objet SQL3).

Les outils du marché

Le chapitre 5 confronte l'offre des principaux outils UML du marché (MagicDraw, MEGA Designer, Modelio, Objecteering, PowerAMC, Rational Rose, Visual Paradigm et Win'Design). Chaque outil est évalué sur différents critères (saisie d'un diagramme de classes, génération d'un modèle relationnel, d'un script SQL et rétroconception d'une base de données).

Annexes

Les annexes contiennent les corrigés détaillés des exercices, une webographie et une bibliographie. L'index propose les termes utilisés dans la définition des concepts et de certaines instructions SQL.

Les identifiants

L'étape d'identification est fondamentale, elle va conditionner par la suite la composition des index et des clés étrangères de vos tables. C'est aussi grâce aux identifiants que vous découvrirez des nouvelles classes à définir.

Qui dit libellé, dit identifiant

« Qui dit libellé, dit identifiant », à force de le dire je l'ai écrit. Un libellé, c'est une donnée le plus souvent textuelle que vous allez rencontrer et pour laquelle vous devrez vous demander si cette dernière est susceptible d'être utilisée à travers différents contextes.

Voici quelques exemples.

- Le nom d'un produit (`nom_produit`) est susceptible d'exister au niveau du fournisseur, du détaillant, de la facture, du grossiste, du transporteur, etc.
- Le type d'une voiture (`type_voiture`) est susceptible d'être utilisé dans les catalogues du constructeur, au sein du service des mines, sur les cartes grises, etc.
- Les prénom et nom (`prenom_client` et `nom_client`) du passager d'un vol se trouveront à coup sûr au niveau des cartes d'embarquement, dans les e-mailings publicitaires, les programmes de fidélité, etc.

Afin d'éviter en amont des probables redondances d'informations au niveau de la base de données, vous devrez, pour chaque attribut de la sorte, y associer un identifiant ayant un nom explicite. Ainsi, `id_produit` devra identifier `nom_produit`, `id_type_voiture` identifiera `type_voiture`, `id_client` identifiera `nom_client` et `prenom_client`, etc.

Dans l'exemple suivant, il apparaît au premier abord trois attributs. Le titre des thèmes tels « L'entretien » et « Les pièces d'usure » (`lib_theme`), le titre des sujets (`lib_sujet`) et le numéro des pages du catalogue (`num_page`).

***Figure 1-8**. Des libellés sans identifiant*

Deux de ces attributs se trouvent être des libellés qui impliquent la définition des identifiants associés (`id_theme` et `id_sujet`). Le schéma conceptuel sera donc composé d'au moins deux classes, illustrées comme suit. L'outil PowerAMC permet de déclarer un identifiant pour chaque classe (et permet de le repérer visuellement).

***Figure 1-9**. Schéma initial des classes UML*

Pourquoi avoir disposé l'attribut `num_page` dans la classe `Sujets` ? Parce qu'il « dépend » de cette classe. En effet, d'après l'analyse qui a été établie, chaque sujet est doté d'un libellé et une seule page du catalogue le concerne. Nous reviendrons plus loin sur cette notion de dépendance.

Concrets ou abstraits ?

Abstrait ou concret, tout objet se doit d'être identifiable.

- Si une classe regroupe des objets concrets, l'identifiant est naturellement concret, mais vous pourrez le considérer également de nature abstraite.
- Si une classe regroupe des objets abstraits, l'identifiant est naturellement abstrait, mais vous pourrez le considérer également de nature concrète.

Le contexte et le point de vue doivent vous conduire à différencier, pour tout objet que vous modéliserez, le concret de l'abstrait et ce que vous désirez stocker. Dans certains cas, les deux aspects d'une même chose peuvent vous intéresser simultanément.

En considérant les objets suivants, examinons les différents points de vue et contextes afin de déterminer, pour chaque classe, l'identifiant adéquat.

***Figure 1-10**. Objets concrets ou abstraits ?*

- L'avion (ici, un A320) : du point de vue de la compagnie aérienne, chaque objet est concret, doté d'une immatriculation, d'un numéro de série, d'une date d'acquisition, et associé à des événements (vols et révisons) etc. L'immatriculation ou le numéro de série sont de bons candidats pour devenir identifiants. Du point de vue d'une agence de voyages, cet objet est abstrait ; il importe peu de savoir quel aéronef de la compagnie réalise le vol. Dans ce cas, le type de l'avion (ici, A320) suffirait à l'identifier. Pour distinguer tout vol réel, l'objet concret avion devient un composant de l'identifiant (il faudra utiliser l'identification relative).
- La voiture (ici, une BMW 320Ci type E46) : du point de vue de la concession, chaque objet est concret, doté d'un numéro de série et de bien d'autres options. Le numéro de série est un bon candidat pour devenir identifiant de chaque véhicule. D'un point de vue du département des ventes, habitué des statistiques, chaque objet devient abstrait et associé à un type catalogue (ici, 320Ci E46). Ce type suffirait à identifier toute voiture perçue comme un modèle. Dans le contexte des cartes grises ou des services de police, plusieurs identifiants seront nécessaires, le numéro de série (qui ne varie jamais) et le numéro d'immatriculation.
- Le livre (ici, *SQL 3e édition*) : du point de vue de l'éditeur, chaque objet peut être perçu de manière abstraite, doté notamment d'un numéro ISBN unique (il existe en fait plusieurs classifications ISBN). Du point de vue d'un libraire ou d'un bibliothécaire, chaque objet est concret et sera vendu ou prêté. En conséquence, un numéro d'exemplaire artificiel devra être ajouté afin d'identifier tout livre perçu ici comme un exemplaire.
- Le plat (ici, un steak-frites) : dans le contexte d'un restaurant, chaque objet sera plutôt perçu de manière abstraite (plusieurs steaks-frites correspondent au même plat qui est vendu à la carte à un certain prix). Un code suffira donc à identifier ce plat. S'il advenait qu'on doive tracer la provenance des viandes, il faudrait considérer l'objet comme concret et associer à chaque steak un numéro unique afin de pouvoir relier ce dernier à l'exploitation d'origine. Dans ce cas, les deux identifiants (celui du steak et celui de l'exploitation) seraient utilisés conjointement.

J'appelle cela les notions « générique » et « spécifique ». Alors que Livre est une classe générique (qui décrit une œuvre littéraire, par exemple Les misérables de Victor Hugo), Livre peut être aussi spécifique (Les misérables, collection Folio, Gallimard). Enfin, Livre peut être un exemplaire d'une bibliothèque. Cela rejoint les notions de classe et d'instance.

Artificiels ou naturels ?

Avant de parler de la nature d'un identifiant, évoquons les deux critères de qualité à respecter :

- stabilité (un objet ne doit pas changer d'identifiant au cours du temps) ;
- minimalité (en termes de taille et de complexité de valeurs).

Lorsque l'attribut d'un objet remplit les deux critères fondamentaux, on parle « d'identification naturelle » (l'identifiant a une sémantique métier ; on parle de « clé métier »).

Lorsqu'aucun attribut présent dans le dictionnaire des données ne remplit les deux critères fondamentaux, il est nécessaire d'ajouter un attribut identifiant (le plus souvent un numérique qui deviendra séquence ou auto-incrément). On parlera « d'identification artificielle ». Cette technique, très utilisée, est simple et favorise le respect des critères de qualité au cours du temps.

Si vous optez pour l'identification artificielle, vous devez, dans la majorité des cas, ajouter à la classe un attribut naturel d'identification (clé métier). Cette identification secondaire permettra par la suite de pouvoir rechercher un objet à l'aide d'une sémantique métier.

Plusieurs identifiants peuvent être déduits de la figure suivante (extrait du site Internet de la banque LCL).

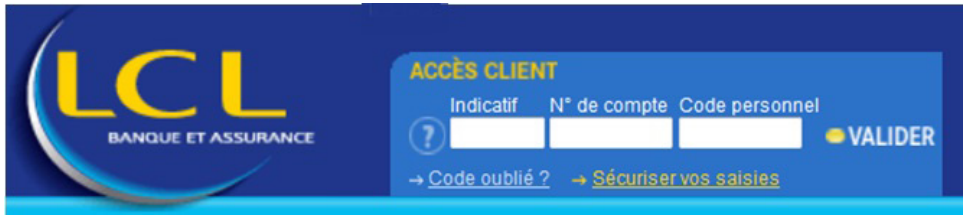

***Figure 1-11**. Identifiants*

- L'identifiant naturel de la banque LCL (`id_banque`) : il est aussi possible d'identifier une banque par un numéro artificiel (c'est le cas du code attribué par la Banque de France : 30002 pour le Crédit Lyonnais).
- L'identifiant naturel de l'agence (`id_agence`), indiqué en tant qu'indicatif du compte : c'est souvent un numéro ou un code de la succursale (chaque établissement bancaire est responsable de cette codification : 4068 pour l'agence de Ramonville Saint-Agne du Crédit Lyonnais).
- L'identifiant naturel du compte (`id_compte`) : c'est souvent un code composé de numéros et de lettres (chaque client connaît ses numéros ou est capable de les retrouver).
- L'identifiant articiel du client (`id_client`) : le client ne le connaît pas, mais le banquier est capable de le retrouver.

Le code personnel ne peut ni jouer le rôle d'identifiant articiel du client (plusieurs clients peuvent choisir le même code, peu probable, mais possible), ni permettre d'identifier un compte (pour la raison précédente et parce qu'un client peut être propriétaire de plusieurs comptes).

Bannissez l'identifiant naturel !

Bien que d'un point de vue scolaire, il faille chercher l'identifiant parmi les attributs de l'entité, cette vision conduit à des problématiques de performances particulièrement sensibles…

Une clé naturelle peut ne pas être connue lors de la saisie primale… Que se passe-t-il si le patient que vous avez modélisé avec comme identifiant son numéro de Sécurité sociale arrive à l'hôpital sans ses papiers et dans le coma ?

Une clé naturelle peut évoluer dans le temps… Que se passe-t-il si ayant modélisé vos adhérents par leurs nom et prénom, Marie DUPONT se marie avec Albert DUVAL ? Quid des données des nombreuses tables filles ?

Une clé naturelle est souvent longue… Un numéro de Sécurité sociale (13 caractères), comme le couple nom et prénom (en moyenne, 15 caractères), c'est 3 à 4 fois plus long qu'un entier et permet environ 6 fois moins de combinaisons (en pratique, on ne peut utiliser qu'une quarantaine de caractères sur les 25 possibles). C'est donc facilement 24 fois moins performant !

Préférez une clé arbitraire la plus courte possible. Par exemple, un entier auto-incrémenté (`SEQUENCE`*,* `IDENTITY` *en SQL). L'utilisation des GUID (Globally Unique IDentifier) ou UUID (Universally Unique Identifier) posant d'autres problèmes en plus de ne pas garantir l'unicité absolue…*

Plusieurs, c'est possible ?

Un objet peut être identifié par différents attributs. L'exemple suivant illustre cette possibilité : tout livre peut être référencé par plusieurs identifiants (un code éditeur, des numéros ISBN, des codes-barres…). Ce raisonnement vaut pour la majorité des autres types de produits.

***Figure 1-12**. Plusieurs identifiants*

Il est possible de définir dans une classe plusieurs identifiants à la condition qu'un seul soit principal.

Les autres identifiants potentiels (appelés secondaires ou alternatifs) n'apparaîtront pas forcément visuellement sur vos schémas conceptuels (cela dépend de l'outil que vous utiliserez). Ces identifiants secondaires se rendront plus utiles au niveau logique en jouant le rôle de clé candidate pouvant se substituer à la clé primaire de la table. Des références supplémentaires pourront être mises en place sans nécessairement utiliser la clé primaire de la table.

La figure suivante présente la classe UML (incomplète) que l'on peut déduire de cet exemple. L'ensemble des données qu'il faudrait recenser n'est pas pris en compte (nom de l'auteur, traducteur…). En revanche, les attributs identifiants potentiels sont tous répertoriés. Seul l'identifiant principal est aisément repérable visuellement.

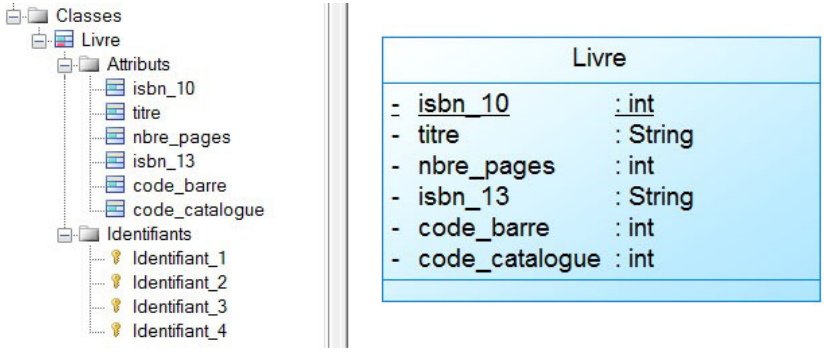

***Figure 1-13**. Plusieurs identifiants dans une classe UML (PowerAMC)*

Les associations binaires

Comme son nom l'indique, une association binaire relie deux classes. Quand la classe est reliée à elle-même, on parle d'association réflexive. Les associations sont souvent déduites des verbes du discours. Suivant le nombre d'objets concernés par l'association, la terminologie permettant de classifier ces associations est la suivante : *un-à-un, un-à-plusieurs, plusieurs-à-plusieurs.*

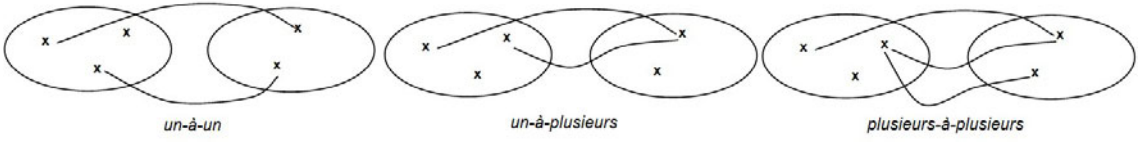

***Figure 1-14**. Classification des associations binaires*

- Les associations *un-à-un* sont les moins courantes. À titre d'exemple, dans le contexte des assurances, citons le contrat (classe) qui concerne (association) un véhicule (classe). Tout véhicule n'est concerné que par un seul contrat.

Transformation des classes-associations

La transformation d'une classe-association s'apparente à celle de l'association *plusieurs-à-plusieurs*. Ainsi, une nouvelle relation est générée (et porte en principe le nom de la classe-association). De plus, si la classe-association contient des attributs, ils apparaissent dans la nouvelle relation (sans être clé toutefois).

En appliquant cette règle à l'exemple suivant, la nouvelle relation contient l'attribut de la classe-association.

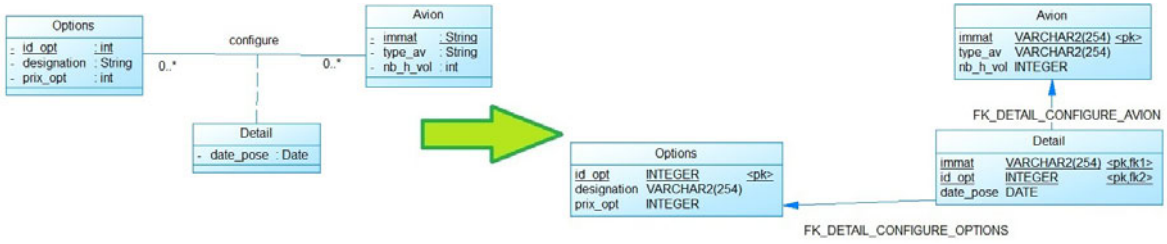

***Figure 2-14**. Transformation d'une classe-association*

Vous devez vous demander comment dériver un schéma relationnel lorsqu'une classe-association est reliée à une autre classe. Eh bien, le plus naturellement du monde : en considérant la classe-association comme une classe à part entière et en adoptant les règles précitées en fonction des multiplicités de l'association à traduire.

Nous verrons plus loin quelques exemples qui incluent des classes-associations connectées à d'autres classes.

Transformation de l'héritage

Il existe trois transformations possibles pour chaque association d'héritage.

Il n'existe pas de solution miracle et vous devrez choisir la meilleure option en fonction de la nature de la contrainte qui concerne l'héritage (partition, exclusion, totalité ou absence de contrainte).

Vous pouvez traduire une association d'héritage par :

- distinction : en créant autant de relations que de classes (surclasse et sous-classes). L'identifiant de la surclasse est propagé dans toutes les relations et devient la clé primaire pour chacune d'elles. Ce même identifiant joue également le rôle de clé étrangère vers la relation déduite de la surclasse (qui préfigure la table de référence) ;
- *push-down* : en migrant tous les attributs de la surclasse dans les sous-classes ;
- *push-up* : en migrant tous les attributs des sous-classes dans la surclasse.

Le premier exemple concerne la classification des personnels en trois populations : commerciaux, managers (chacun responsable de plusieurs personnels) et administratifs. En l'absence de contrainte, ce diagramme UML signifie qu'il existe des personnels non classifiés (ni commerciaux, ni managers, ni administratifs) et qu'aucun d'entre eux ne peut appartenir à plus d'une classification.

La transformation par distinction implémente correctement cet état de fait.

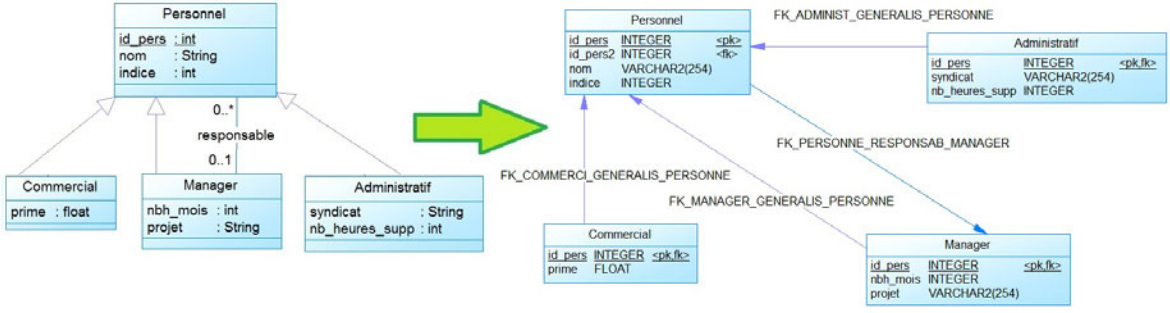

***Figure 2-15**. Transformation d'un héritage par distinction*

La transformation par distinction d'une association d'héritage est la plus polyvalente. Sous réserve de codage, elle permet d'implémenter n'importe quelle autre contrainte d'héritage (partition, exclusion et totalité).

Le deuxième exemple considère qu'il n'existe aucun personnel n'étant ni commercial, ni manager, ni administratif et que chacun est cantonné à un seul type d'emploi (contrainte UML `complete, disjoint`). La transformation par *push-down* implémente correctement cette hypothèse.

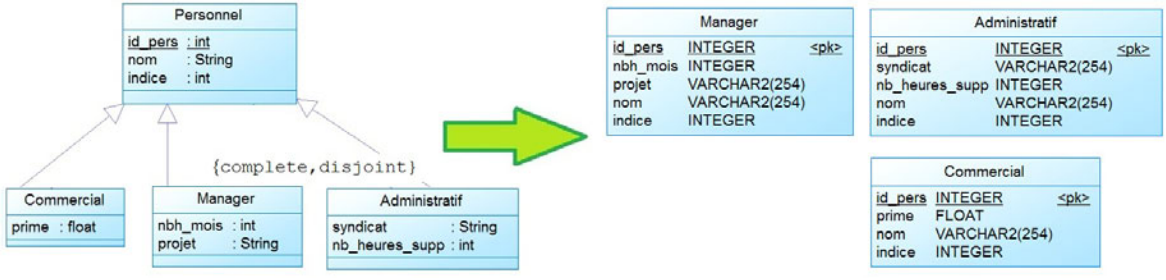

***Figure 2-16**. Transformation d'un héritage par push-down*

Le troisième exemple présente la transformation d'un héritage simple par *push-up*. Cette solution permet d'implémenter toutes les contraintes, mais convient vraiment mieux lorsque chaque personnel peut occuper plus d'un type d'emploi.

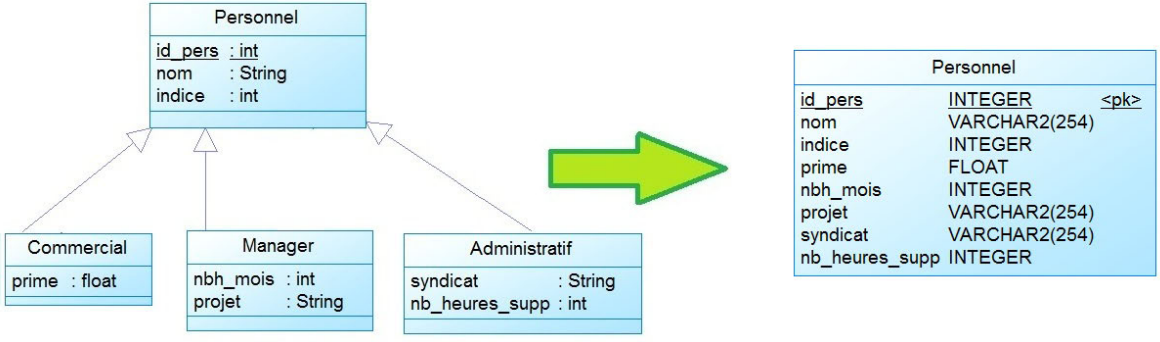

***Figure 2-17**. Transformation d'un héritage par push-up*

Le tableau suivant précise les options souhaitables pour chaque contrainte d'héritage. Quelle que soit la solution que vous choisirez, vous devrez programmer la contrainte sous la forme de directives SQL `CHECK`, déclencheurs ou à l'intérieur de vos procédures cataloguées (voir le chapitre 3), sauf en ce qui concerne la transformation par distinction et lorsqu'il n'est pas nécessaire d'assurer l'exclusivité.

Tableau 2-1 : Transformation de l'héritage

Options de transformation	Compatibilité avec les contraintes
Distinction	☺ Exclusivité : `{incomplete,disjoint}` (par défaut) ☺ Pas de contrainte : `{incomplete,overlapping}`
Push-down	☺ Partition : `{complete,disjoint}` ☹ Totalité : `{complete,overlapping}`, redondance d'informations
Push-up	☺ Totalité : `{complete,overlapping}` ☹ Autres cas : présence de nombreuses valeurs `NULL`

En pratique, il est rare d'utiliser la technique de rassemblement des données qu'est le push-down, car le modèle relationnel est fait pour éviter les redondances. Or, la migration de tous les attributs de la surclasse dans les sous-classes entraîne systématiquement de la redondance, sauf dans le cas ou la surclasse ne contient que la clé (et dans ce cas, on en revient à la distinction).

De même, la technique du push-up est rare, car elle crée des tables avec un fort taux de `NULL` *si les sous-classes sont exclusives. Et dans ce cas, il convient de gérer proprement les conditions de* `NULL`*ification à l'aide de contraintes souvent complexes.*

Enfin, même si la technique de distinction est la plus souvent implémentée, il ne faut pas oublier qu'elle nécessite l'ajout de contraintes complexes, lorsqu'il y a exclusivité entre les membres des sous-classes.

Pour autant, la technique de l'héritage induit d'excellents bénéfices en termes de performances, en réduisant le nombre de colonnes des tables et en permettant l'élision de jointures (dans le cas d'héritages successifs, il n'est pas utile de passer par toutes les classes descendantes pour joindre la racine à une feuille).

La solution « universelle »

Il existe une transformation qui peut convenir à toute association binaire (*un-à-un*, *un-à-plusieurs*, *plusieurs-à-plusieurs* et réflexive). Il s'agit de générer une nouvelle relation munie de deux clés étrangères (qui préfigure la table d'association ou « table de jointure »). La clé primaire de cette relation est composée soit par un des identifiants, soit par le couple des identifiants des deux relations initiales.

Appliquons cette règle à l'exemple de deux associations binaires. Deux relations doivent être générées. L'association *un-à-un* est traduite par la relation `diriger` et l'association *un-à-plusieurs* est traduite par la relation `affecter`.

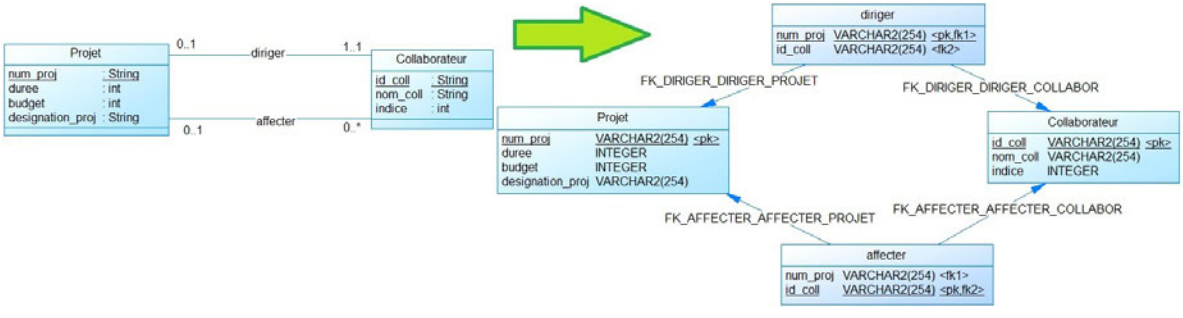

***Figure 2-18**. Solution universelle*

Pour s'assurer qu'un collaborateur ne dirige qu'un projet, il conviendra d'ajouter, au niveau physique, une contrainte `UNIQUE` sur la clé étrangère (`id_coll`) dans la table `diriger`.

Dans l'autre table d'association, la clé primaire `id_coll` permettra de s'assurer qu'un collaborateur ne puisse être affecté à plusieurs projets.

Ce principe de transformation est idéal lorsque les multiplicités de vos associations sont susceptibles de changer avec le temps (dans notre exemple, le fait qu'un collaborateur puisse diriger plusieurs projets ou participer à différents projets). En ce cas, et même avec des données en base, il suffira d'activer ou de désactiver telle contrainte ou telle clé.

Ainsi, pour l'association universelle de type un-à-un, il faut choisir la clé primaire de la table de jointure qui peut venir indifféremment de l'une ou l'autre des entités. Mon conseil est donc le suivant : prenez toujours celle de l'entité qui existe avant l'autre. Dans l'exemple proposé, la table de jointure `diriger` *se voit dotée d'une clé primaire venant de l'entité* `Projet`*. C'est à mon sens une erreur, car il me semble difficile qu'un projet existe dans votre entreprise, si vous n'avez aucun collaborateur !*

Les transformations à éviter

Je vais encore dire du mal des associations *n*-aires, mais la plupart d'entre elles le méritent. Considérons à nouveau le premier exemple de la section « Les associations *n*-aires » du chapitre 1, l'association *n*-aire `Installer` exprimait le fait que des *logiciels* soient installés sur des *serveurs* par des *départements*.

La transformation de toute association *n*-aire entraîne l'apparition d'une nouvelle relation. Si les multiplicités sont toutes de type *plusieurs-à-plusieurs*, la clé primaire de la nouvelle relation sera composée des clés de toutes les relations connectées. Si la multiplicité d'un lien de l'association *n*-aire vaut 1, le degré de la clé primaire est réduit d'une colonne.

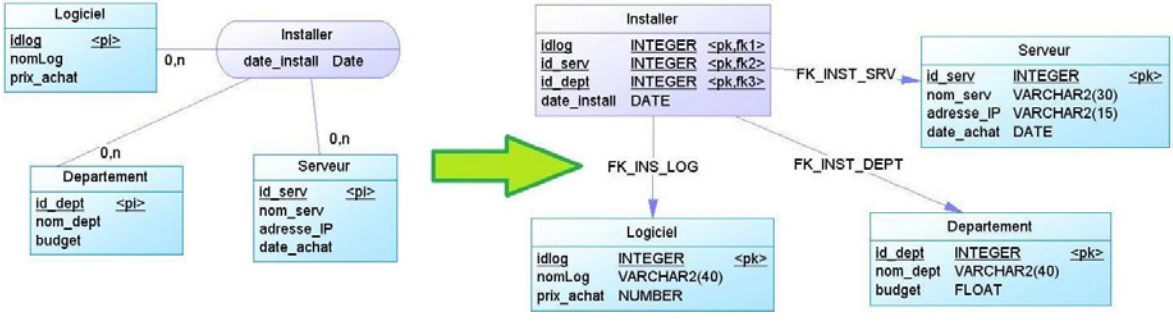

Figure 2-19*. Une transformation à éviter*

Cette transformation est à éviter, car la table d'association qu'on obtiendra ne permettra aucun contrôle de cohérence. Ainsi, avec ce schéma, il est possible d'installer un logiciel qui n'est pas compatible avec un serveur ou qu'un département ne détient pas par exemple.

Les pictogrammes

Ce pictogramme introduit une définition, un concept ou une remarque importante.

Ce pictogramme indique une astuce ou un conseil personnel.

Ce pictogramme introduit une remarque, un avis divergent, un complément ou un coup de gueule de Frédéric Brouard.

Contact avec l'auteur

Si vous avez des remarques à formuler sur le contenu de cet ouvrage, n'hésitez pas à m'écrire (christian.soutou@gmail.com). Vous trouverez sur le site d'accompagnement de cet ouvrage accessible par la fiche du livre sur www.editions-eyrolles.com, les errata ainsi que d'éventuelles discussions.

Vous pouvez aussi poster des questions sur vos modèles sur http://www.developpez.net/forums/f940/general-developpement/conception/modelisation/, de nombreux contributeurs s'y retrouvent.

Traduction des agrégations (composition)

Le tableau suivant décrit la transformation d'une agrégation de composition. L'association doit se traduire par deux mécanismes : migration d'une clé étrangère du composite et enrichissement de la clé primaire du composant.

Tableau 3-5 : Transformation d'un agrégat

Schéma relationnel	Script SQL

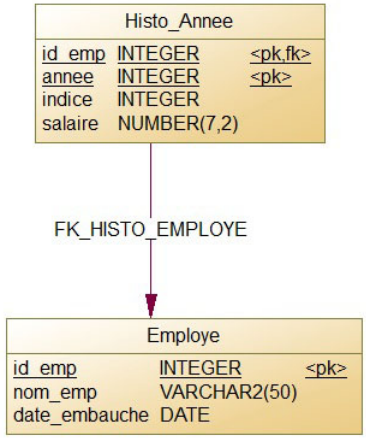	`ALTER TABLE histo_annee`
	`  DROP CONSTRAINT fk_histo_employe;`
	`DROP TABLE employe CASCADE CONSTRAINTS;`
	`DROP TABLE histo_annee CASCADE CONSTRAINTS;`
	
	`CREATE TABLE employe  (`
	`  id_emp  INTEGER NOT NULL,`
	`  nom_emp  VARCHAR2(50),`
	`  date_embauche DATE,`
	`  CONSTRAINT pk_employe PRIMARY KEY (id_emp));`
	
	`CREATE TABLE histo_annee  (`
	`  id_emp  INTEGER NOT NULL,`
	`  annee  INTEGER NOT NULL,`
	`  indice  INTEGER,`
	`  salaire  NUMBER(7,2),`
	`  CONSTRAINT pk_histo_annee`
	`  PRIMARY KEY (id_emp, annee));`
	
	`ALTER TABLE histo_annee`
	`  ADD CONSTRAINT fk_histo_employe`
	`  FOREIGN KEY (id_emp)`
	`  REFERENCES employe (id_emp);`

***Figure 3-6**. Agrégation à traduire*

Vérifiez que la clé primaire de la table « composant » est constituée en partie de la clé primaire de la table « composite ».

Traduction des associations *plusieurs-à-plusieurs*

Le tableau suivant décrit la transformation d'une association *plusieurs-à-plusieurs* (résultant d'une classe-association au niveau conceptuel).

Tableau 3-6 : Transformation d'une association plusieurs-à-plusieurs

Schéma relationnel

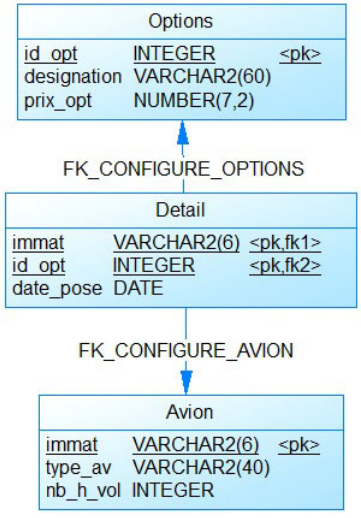

***Figure 3-7**. Association plusieurs-à-plusieurs à traduire*

Script SQL

```
ALTER TABLE detail
   DROP CONSTRAINT fk_configure_avion;
ALTER TABLE DETAIL
   DROP CONSTRAINT fk_configure_options;
DROP TABLE avion CASCADE CONSTRAINTS;
DROP TABLE detail CASCADE CONSTRAINTS;
DROP TABLE options CASCADE CONSTRAINTS;

CREATE TABLE avion  (
   immat       VARCHAR2(6) NOT NULL,
   type_av     VARCHAR2(40),
   nb_h_vol    INTEGER,
   CONSTRAINT pk_avion PRIMARY KEY (immat));

CREATE TABLE detail  (
   immat       VARCHAR2(6) NOT NULL,
   id_opt      INTEGER     NOT NULL,
   date_pose   DATE,
   CONSTRAINT PK_DETAIL
               PRIMARY KEY (immat, id_opt));

CREATE TABLE OPTIONS  (
   id_opt       INTEGER NOT NULL,
   designation VARCHAR2(60),
   prix_opt    NUMBER(7,2),
   CONSTRAINT pk_options
               PRIMARY KEY (id_opt));

ALTER TABLE detail
   ADD CONSTRAINT fk_configure_avion
      FOREIGN KEY (immat)
      REFERENCES avion (immat);
ALTER TABLE detail
   ADD CONSTRAINT fk_configure_options
      FOREIGN KEY (id_opt)
      REFERENCES options (id_opt);
```

Vérifiez que la clé primaire de la table d'association est composée de deux clés étrangères.

Solution universelle

Le tableau suivant décrit la transformation de deux associations binaires entre les relations `Projet` et `Collaborateur`. L'association `affecter` est de type de *un-à-plusieurs* l'association `diriger` est de type de *un-à-un*.

Chaque association doit se traduire par une table d'association munie de deux clés étrangères. Les contraintes `UNIQUE` et `NOT NULL` sur chacune de ces clés permettront d'implémenter telle ou telle multiplicité.

Tableau 3-7 : Transformation de deux associations binaires par la solution universelle

Schéma relationnel	Script SQL

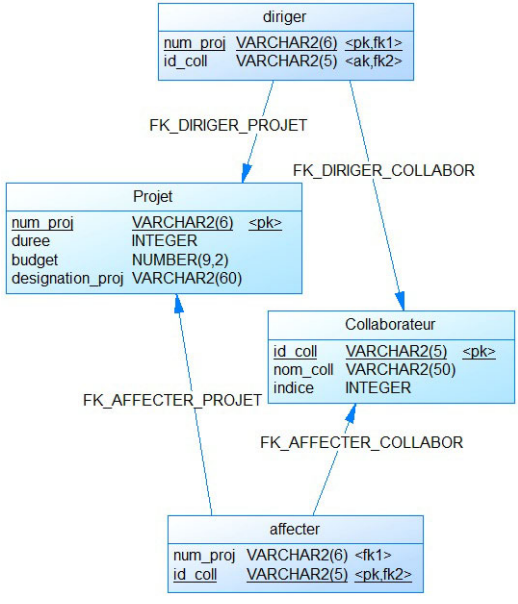

***Figure 3-8**. Associations binaires à traduire*

```
ALTER TABLE affecter
   DROP CONSTRAINT fk_affecter_collabor;
ALTER TABLE affecter
   DROP CONSTRAINT fk_affecter_projet;
ALTER TABLE diriger
   DROP CONSTRAINT fk_diriger_collabor;
ALTER TABLE diriger
   DROP CONSTRAINT fk_diriger_projet;
DROP TABLE collaborateur CASCADE CONSTRAINTS;
DROP TABLE projet CASCADE CONSTRAINTS;
DROP TABLE affecter CASCADE CONSTRAINTS;
DROP TABLE diriger CASCADE CONSTRAINTS;

CREATE TABLE collaborateur  (
   id_coll   VARCHAR2(5) NOT NULL,
   nom_coll  VARCHAR2(50),
   indice    INTEGER,
   CONSTRAINT pk_collaborateur
              PRIMARY KEY (id_coll));
CREATE TABLE projet  (
   num_proj        VARCHAR2(6) NOT NULL,
   duree           INTEGER,
   budget          NUMBER(9,2),
   designation_proj  VARCHAR2(60),
   CONSTRAINT pk_projet
              PRIMARY KEY (num_proj));
CREATE TABLE affecter  (
   num_proj   VARCHAR2(6),
   id_coll    VARCHAR2(5) NOT NULL,
   CONSTRAINT pk_affecter
              PRIMARY KEY (id_coll);
```

Schéma relationnel (*suite*)	Script SQL (*suite*)

```
CREATE TABLE diriger  (
   num_proj  VARCHAR2(6) NOT NULL,
   id_coll   VARCHAR2(5) NOT NULL,
   CONSTRAINT pk_diriger
              PRIMARY KEY (num_proj),
   CONSTRAINT AK_DIRIGER UNIQUE(id_coll));

ALTER TABLE affecter
   ADD CONSTRAINT fk_affecter_collabor
      FOREIGN KEY (id_coll)
      REFERENCES collaborateur (id_coll);
ALTER TABLE affecter
   ADD CONSTRAINT fk_affecter_projet
      FOREIGN KEY (num_proj)
      REFERENCES projet (num_proj);
ALTER TABLE diriger
   ADD CONSTRAINT fk_diriger_collabor
      FOREIGN KEY (id_coll)
      REFERENCES collaborateur (id_coll);
ALTER TABLE diriger
   ADD CONSTRAINT fk_diriger_projet
      FOREIGN KEY (num_proj)
      REFERENCES projet (num_proj);
```

La contrainte `NOT NULL` sur la colonne `id_coll` dans la table `diriger` signifie que tout projet est dirigé par un collaborateur (multiplicité minimale 1). La contrainte `UNIQUE` sur cette même colonne implémente le fait qu'un collaborateur ne peut diriger plusieurs projets (multiplicité maximale 1).

L'absence de contrainte `NOT NULL` sur la colonne `num_proj` dans la table `affecter` signifie qu'un collaborateur ne peut participer à aucun projet (multiplicité minimale 0). La contrainte `UNIQUE` (du fait de la clé primaire) sur la colonne `id_coll` de cette même table implémente le fait qu'un collaborateur ne peut participer à plusieurs projets (multiplicité maximale 1).

Cette solution présente l'avantage d'être la plus évolutive si les multiplicités viennent à changer. En effet, nul besoin de modifier la structure d'une table, seules des contraintes seront à activer ou à désactiver.

Mise en pratique

Les exercices 3.1 « La création de tables (carte d'embarquement) » et 3.2 « La création de tables (horaires de bus) » vous proposent de déduire le script SQL à partir de différents modèles conceptuels.

Et voici les données présentées par cette vue.

```
TVA_TYPE        TVA_DATE_DEBUT_APPLICATION TVA_DATE_FIN_APPLICATION  TVA_TAUX
--------------- -------------------------- -------------------------- --------
Super Réduit    1982-07-01                 1988-12-31                  5,5
Super Réduit    1989-01-01                 9999-12-31                  NULL
Réduit          1968-01-01                 1968-11-30                  6,38
Réduit          1968-12-01                 1969-12-31                  7,53
Réduit          1970-01-01                 1972-12-31                  7,5
Réduit          1973-01-01                 1988-12-31                  7
Réduit          1989-01-01                 9999-12-31                  5,5
Intermédiaire   1968-01-01                 1968-11-30                  14,92
Intermédiaire   1968-12-01                 1969-12-31                  15,65
Intermédiaire   1970-01-01                 1982-06-30                  17,6
Intermédiaire   1982-07-01                 9999-12-31                  NULL
Normal          1968-01-01                 1968-11-30                  20
Normal          1968-12-01                 1969-12-31                  23,46
...
```

Ce serait une énorme faute que de rajouter à la table une date de fin d'application, car en cas d'erreur de saisie, on pourrait se retrouver soit avec plusieurs taux différents pour un même jour, soit aucun…

À noter : la présence d'une ligne avec un taux à `NULL` *signifie tout simplement la disparition de ce taux de TVA !*

Contrôles d'intégrité référentielle

Une vue peut contrôler la validité des références entre tables. En l'absence du mécanisme de clés étrangères, vous pouvez programmer l'intégrité référentielle (un autre moyen plus coûteux est d'utiliser des déclencheurs).

La cohérence référentielle entre deux tables `t1` (table « parent ») et `t2` (table « enfant ») se programme :

- du parent vers l'enfant par l'utilisation d'une vue `v1` de la table `t1` définie avec la clause `NOT EXISTS` ;
- de l'enfant vers le parent par l'utilisation d'une vue `v2` de la table `t2` définie avec la clause `WITH CHECK OPTION`.

Considérons la table des départements (renommée `t1` et jouant le rôle de parent : `CREATE TABLE t1 AS SELECT * FROM hr.departments`) et celle des employés (renommée `t2` et jouant le rôle de l'enfant : `CREATE TABLE t2 AS SELECT * FROM hr.employees`). Puisqu'il n'existe aucune clé étrangère entre ces tables, les vues à créer sont les suivantes.

Tableau 4-18 : Vues à créer

Code SQL	Commentaire

`CREATE VIEW v1`
`AS SELECT * FROM t1 pere`
`WHERE NOT EXISTS`
`(SELECT employee_id`
`FROM t2`
`WHERE department_id=pere.department_id);`
	Vue pour assurer la cohérence du parent vers l'enfant.

`CREATE VIEW v2`
`AS SELECT * FROM t2`
`WHERE (department_id IN`
`(SELECT department_id FROM t1)`
`OR department_id IS NULL)`
`WITH CHECK OPTION;`
	Vue pour assurer la cohérence de l'enfant vers le parent.

La vue `v2` restitue les départements qui n'embauchent aucun employé. La vue `v1` restitue les employés associés à un département existant et ceux qui ne sont associés à aucun département (la condition `IS NULL` peut être omise dans la vue si chaque enfant doit être obligatoirement rattaché à un parent).

Les règles que vous devez respecter pour manipuler les données sont les suivantes.

- Côté *parent* : modifications, insertions et suppressions par la vue `v1` et lecture de la table `t1`.
- Côté *enfant* : modifications, insertions, suppressions et lecture par la vue `v2`.

Manipulons à présent les données de notre exemple par ces vues.

Tableau 4-19 : Manipulations des vues pour l'intégrité référentielle

Cohérence fils → père	Cohérence père → fils

Insertion correcte :
`INSERT INTO v2`
`(employee_id, first_name, last_name, email,`
`hire_date, job_id,salary, commission_pct,`
`manager_id, department_id )`
`VALUES (99,'Fred','Brouard','sqlpro',`
`SYSDATE,'IT_PROG',3400,0,100,90);`
	Toute insertion à travers la vue `v1` est possible (sous réserve de la validité du type des colonnes et de l'existence d'autres contraintes référentielles).
	Insertion correcte :
	`INSERT INTO v1`
	`(department_id, department_name,`
	`manager_id, location_id)`
	`VALUES (800,'Eyrolles info.', 100,1700);`

Cohérence fils → père (*suite*)	Cohérence père → fils (*suite*)

Insertion incorrecte (le département 999 est un parent absent) :
`INSERT INTO v2`
`(employee_id, first_name, last_name, email,`
`hire_date, job_id, salary, commission_pct,`
`manager_id, department_id )`
`VALUES (98,'Fred','Comby','bezier',`
`SYSDATE,'IT_PROG',5400,0,100,`**`999`**`);`	

Modification correcte :
`UPDATE v2`
`SET department_id = 10`
`WHERE employee_id=99;`
 	
Modification incorrecte (parent absent) :
`UPDATE v2`
`SET department_id =` **`999`**
`WHERE employee_id=99;`
	Modification correcte (le département 900 n'a aucun employé) :
	`UPDATE v1`
	`SET department_id = 77`
	`WHERE department_id = 900;`
	
	Modification incorrecte (enfant présent : le département 90 a des employés et le mécanisme de cascade n'est pas pris en compte) :
	`UPDATE v1`
	`SET department_id = 77`
	`WHERE department_id = 90;`

Toute suppression est possible à travers la vue `v2` (sous réserve de l'existence d'autres contraintes référentielles).	Suppression correcte (enfants absents) :
	`DELETE FROM v1`
	`WHERE department_id = 900;`
	
	Suppression incorrecte (enfants présents) :
	`DELETE FROM v1`
	`WHERE department_id = 90;`

Le raisonnement sur deux tables peut se généraliser à une hiérarchie d'associations ou à différentes associations issues de la même table.

Un avantage spectaculaire et peu connu des vues : les contraintes dynamiques

Dans une base de données, les contraintes sont statiques, c'est-à-dire qu'elles empêchent d'introduire des données qui ne respectent pas les règles. Mais comment faire pour stocker quand même certaines données intermédiaires et ne présenter aux utilisateurs que les ensembles de données valides ? La réponse est une vue… Imaginons un géomètre qui doit stocker dans une table ses relevés de mesure de parcelles de terrain. Voici la table :

```
CREATE TABLE T_PARCELLE_GEOMETRIE_PCG
(PCL_NUMERO_CADASTRAL     VARCHAR(16) NOT NULL,
 PCG_NUMERO_POINT         INT NOT NULL,
 PCG_LATITUDE             FLOAT NOT NULL,
 PCG_LONGITUDE            FLOAT NOT NULL,
 PCG_ALTITUDE             FLOAT,
 CONSTRAINT PK_PCG PRIMARY KEY (PCL_NUMERO_CADASTRAL,
 PCG_NUMERO_POINT));
```

Chacun sait qu'une parcelle de terrain est un polygone et qu'un polygone se doit d'avoir au moins 3 sommets. Tant que l'on n'a pas saisi 3 points, pas de polygone. Or, aucune contrainte ne peut « attendre » l'arrivée du troisième point pour valider ou invalider les données. Dans ce cas, la solution réside dans la vue :

```
CREATE VIEW V_PARCELLE_GEOMETRIE_PCG AS
SELECT * FROM T_PARCELLE_GEOMETRIE_PCG AS P1
WHERE  EXISTS(SELECT 1
       FROM   T_PARCELLE_GEOMETRIE_PCG AS P1 AS P2
       WHERE  P1.PCL_NUMERO_CADASTRAL = P2.PCL_NUMERO_CADASTRAL
       AND  P1.PCG_NUMERO_POINT = P2.PCG_NUMERO_POINT
       GROUP  BY P2.PCG_NUMERO_CADASTRAL, P2.PCG_NUMERO_POINT
                          HAVING COUNT(*) >= 3);
```

Cette dernière ne fera apparaître que les données des points des parcelles pour celles formant des polygones… Mais comme on ne sait pas à l'avance le nombre de points d'un polygone, la manière la plus pratique est de rajouter cette information de « fermeture » du polygone à un niveau supérieur :

```
CREATE TABLE T_PARCELLE_PCL
(PCL_NUMERO_CADASTRAL     VARCHAR(16) NOT NULL PRIMARY KEY,
 PCL_SAISIE_FINALISE      BOOLEAN NOT NULL DEFAULT 0);

ALTER TABLE T_PARCELLE_GEOMETRIE_PCG
   ADD CONSTRAINT FK_PCG_PCL
      FOREIGN KEY (PCL_NUMERO_CADASTRAL)
      REFERENCES T_PARCELLE_GEOMETRIE_PCG (PCL_NUMERO_CADASTRAL);
```

Dès lors, lorsque le géomètre a terminé sa saisie, il fait passer l'information `PCL_SAISIE_FINALISE` *à* `'true'` *et le polygone représentant la parcelle devient visible à travers la vue suivante :*

```
CREATE VIEW V_PARCELLE_GEOMETRIE_PCG AS
SELECT * FROM T_PARCELLE_GEOMETRIE_PCG AS PCG
       INNER JOIN T_PARCELLE_PCL AS PCL
       ON PCL.PCL_NUMERO_CADASTRAL = PCG.PCL_NUMERO_CADASTRAL
WHERE  PCL_SAISIE_FINALISE;
```

Un autre exemple concerne les clients et les prospects. Un prospect n'est autre qu'un client qui n'a jamais effectué de commande. Dès lors, clients et prospects peuvent figurer dans la même table physique (`CONTACT`*) ce qui simplifie beaucoup les choses. Il n'y aura qu'à créer deux vues modifiables, chacune d'elles avec toutes les colonnes de la table des contacts, l'une présentant les données des prospects, l'autre celle des clients, avec les définitions de vues suivantes :*

```
CREATE VIEW V_CLIENT_CLI AS
SELECT * FROM  T_CONTACT_CTC
WHERE  CTC_ID IN (SELECT CTC_ID FROM T_COMMANDE_CMD);

CREATE VIEW V_PROSPECT_PSP AS
SELECT * FROM T_CONTACT_CTC
WHERE   CTC_ID NOT IN (SELECT CTC_ID FROM T_COMMANDE_CMD);
```

Dénormalisation

Les vues peuvent aider à dénormaliser un schéma relationnel. Il est possible de créer volontairement des résultats tabulaires qui ne respectent aucune des formes normales (de la 1re à la 5e).

L'exemple suivant ne respecte pas la deuxième forme normale. En effet, la clé étant `id_emp`, la présence de la colonne `department` (nom du département qui dépend davantage d'un code département) fait abandonner la troisième forme normale.