10. Conception de la base de données

Le présent chapitre est, avec le précédent, le plus important de cet ouvrage. C'est ici que nous apprenons véritablement à concevoir un modèle de données. C'est aussi la raison de sa longueur, car le processus est détaillé de manière aussi complète que possible.

Une fois l'inventaire des données effectué (voir chapitre 9), nous commençons à avoir une idée assez précise de ce qui va figurer dans la base de données. Pourtant nous ne sommes loin du bout de nos peines, car le travail réel d'analyse et de conception n'a même pas commencé.

Objectif: Sur la base de l'inventaire effectué, déterminer la structure, le schéma, d'une base de données apte à recevoir les types d'objet recensés avec leurs informations détaillées.

Résultat: un modèle de données s'exprimant sous forme de types d'objet, d'attributs rattachés à ces types d'objet et d'associations reliant ces types d'objet.

Comment passer de l'inventaire au modèle de données? Les personnes habituées à cet exercice trouveront facilement la voie pour mettre le processus en route, commençant par les aspects les plus évidents et renvoyant la résolution des aspects problématiques à plus tard.

Ici, nous présentons une approche empirique destinée à permettre aux novices de mettre le processus de conception en route. Nous leur conseillons vivement de commencer par là et de ne pas aborder toutes les difficultés en même temps. Les personnes plus expérimentées parcourront évidemment à grands bonds ce parcours pédagogique.

Est-il indispensable de disposer d'abord d'un inventaire des données avant de commencer le travail de modélisation? Nous répondrons OUI s'il s'agit d'un projet complexe et si les responsables de la modélisation ne connaissent pas déjà le domaine et l'entreprise. On peut par contre répondre NON s'il s'agit de petits modèles et que les personnes impliquées connaissent bien le domaine. L'inventaire sera dans ce cas fait "à la volée". Il faudra par contre ne pas omettre de valider en détail avec les futurs utilisateurs le schéma résultant. On se trouve souvent alors confronté à de sacrées surprises: à tort, on croyait que…

Recommandation: lors du travail de modélisation, il vaut mieux ne pas se soucier des difficultés que pourraient poser le modèle élaboré plus tard, lors de l'implémentation et de l'exploitation. Performances, volume de stockage? Oui, ces questions devront être abordées et résolues, mais nous renvoyons leur traitement à plus tard. L'important, pour le moment, est d'élaborer une structure correcte, adaptée à la nature du problème et aux besoins de l'organisation.

1ère étape: Découvrir les types d'objet élémentaires, indépendants

La notion de type d'objet élémentaire (ou indépendant) n'a rien d'absolu ni de précis. Simplement, penser dans cette optique facilite grandement le défrichage du problème. L'approche paraît évidente, même aux débutants. Formellement, elle possède ses limites et contradictions, mais ne constitue de toute manière il qu'une première étape.

Nous appelons type d'objet élémentaire un type d'objet qui peut exister indépendamment d'autres types d'objet. Dans le temps, on parlait de fichiers maîtres, de données de base.

Exemple: dans une application de vente, le type d'objet *client* peut être considéré comme élémentaire, indépendant. Les clients possèdent une existence, ce sont des entreprises ou personnes réelles. On peut définir un objet *client* sans au départ lui associer une commande. De même, les articles que nous vendons possèdent une existence indépendante. Il s'agit d'objets qui peuvent traîner sur une étagère et ne seront peut-être même jamais vendus (nous espérons le contraire, bien sûr!). La commande que nous passe un client, par contre, n'est pas un type d'objet indépendant, élémentaire. Cette commande n'existe qu'en relation avec le client qui nous l'a passée et avec les articles commandés.

Encore une fois, cette distinction n'a rien de formel, mais nous aide à démarrer le processus. Nous proposons donc, dans une première étape de

Découvrir, parmi les types d'objet contenus dans l'inventaire, ceux de nature élémentaire, indépendante.

Rattacher à ces types d'objet les informations élémentaires qui les décrivent (leurs attributs).

Contrôler, à ce niveau déjà, l'absence de violations de la 1ère et de la 3ème forme normale.

Attribuer à ces types d'objet une clé primaire, de préférence sous forme d'un attribut neutre.

À ce stade, toutes les définitions de types d'objet restent "ouvertes". Elles seront complétées par de nouveaux attributs au fur et à mesure de la progression du travail de modélisation. Effectuant ce travail, on cochera dans l'inventaire des données tous les types d'objet et toutes les informations détaillées qui ont déjà été prises en compte.

Exemple: système de fabrication

Analysant un système de fabrication dans plusieurs usines, nous découvrons dans le cadre de l'inventaire les types d'objet élémentaires suivants:

Usine (Nom, adresse, …)

Département (Nom, …)

Employé (No d'employé, nom, prénom, sexe, date de naissance, date d'entrée, fonction, …)

Article (Code article, libellé, catégorie, poids, prix de revient, …)

Machine (No de machine, type de machine, no de série, …)

Notons que nous avons soigneusement évité à ce stade de décrire des associations entre types d'objet, par exemple entre *employé* et *département* ou *usine* ou encore entre *article* et *usine* ou *machine*.

Lors de l'attribution de clés primaires à ces définitions, nous décidons de n'accepter à ce titre que des attributs existants qui soient parfaitement neutres (donc pas de clés primaires "parlantes", contenant d'autres informations). Et, si un tel attribut n'existe pas déjà, d'introduire un nouvel attribut, sans signification pour l'utilisateur, dont la valeur sera attribuée par le système de façon automatique. Voir chapitre 5: Le choix de la clé primaire. Nous désignons une clé primaire neutre de la façon suivante: #-clé ("numéro de" …) et nous la soulignons.

Dans l'exemple ci-dessus, nous devons attribuer une clé primaire à *Usine* et *Département* qui ne possèdent pas d'attribut unique, invariable, déterminant. Une discussion avec les utilisateurs révèle que le *no d'employé* officiel est un pur numéro attribué au fur et à mesure des engagements. Nous le promouvons au rang de clé primaire. Par contre *code article* ne possède pas ces qualités: il commence par deux lettres décrivant la catégorie d'article; nous introduisons en conséquence ici une nouvelle clé neutre. Le numéro de machine est neutre.

Nous obtenons donc:

Usine (#-usine, nom, adresse, …)

Département (#-département, nom, …)

Employé (#-employé, nom, prénom, sexe, date de naissance, date d'entrée, fonction, …)

Article (#-article, code article, libellé, catégorie, poids, prix de revient, …)

Machine (#- machine, type de machine, no de série, …)

Codification d'attributs

Examinons à présent, toujours à titre d'exemple, les attributs qui pourraient poser problème.

L'attribut *adresse* dans *usine* est suspect. Que va-t-il contenir? Une adresse réelle avec tous ses éléments? Dans ce cas, il n'est pas atomique et doit être détaillé, c'est le dernier moment et cela aurait dé-

jà dû être fait lors de l'inventaire. À discuter avec les utilisateurs! Pour l'exemple, on nous a répondu à ce propos qu'il ne s'agissait pas de gérer l'adresse postale de l'usine, mais que *nom* contient en fait la localité et *adresse* le pays où se trouve l'usine. Exemple: l'usine Montbéliard en France. Nous adaptons donc la définition comme suit:

Usine (#usine, localité, pays)

Autre attribut sujet à controverse dès le départ: *catégorie* dans *article*. Précisons tout de suite que la gestion des classifications d'articles (avec catégories, sous-catégories, etc.) relève dans toute application presque du travail de doctorat. Pour rester simple ici, constatons qu'un utilisateur ne sera certainement pas autorisé à saisir n'importe quelle valeur dans le champ catégorie, faute de quoi des statistiques par catégories d'article pourraient ne pas produire les résultats escomptés. Il est donc absolument nécessaire de formaliser la définition des catégories et, après discussion avec les utilisateurs, nous créons un nouveau type d'objet:

Catégorie d'article (#-catégorie article, libellé catégorie)

Dans *article*, l'attribut *catégorie* ne sera donc plus un texte, mais un numéro qui renvoie à la table des catégories. Déjà nous avons décrit une association, alors que nous voulions éviter de le faire à ce stade! La définition d'*article* devient alors:

Article (#-article, code article, libellé, *#-catégorie*, poids, prix de revient, …)

Et *article* contient à présent une clé étrangère, présentée en italique comme convenu au chapitre 3.

Le même argument vaut pour *fonction* dans *employé* et *type de machine* dans *machine*. Il faudra déterminer avec les utilisateurs si une codification séparée de ces informations est requise. S'il s'agit simplement de codifier les valeurs permises, il suffit alors de créer une nouvelle table dans la table des codes. Voir chapitre 7.

Les concepteurs expérimentés auront probablement déjà réagi à de tels problèmes lors de l'inventaire des données. Il est en fait sage de se demander systématiquement au sujet de presque chaque attribut si ses valeurs doivent être soumises à une codification rigoureuse.

Prenons l'exemple d'un fichier de personnes avec leur adresse:

Personne (#-personne, nom, prénom, rue, no, code postal, localité, pays)

Codifier les noms de famille et les prénoms (donc créer une table des noms et prénoms valables) nous paraît irréaliste au vu du brassage des populations qui règne aujourd'hui. Celui des rues non plus. Par contre, la nécessité de créer une table des localités avec leur code postal et leur nom, de même qu'une table des pays pourrait, selon l'application, certainement se justifier. À ce propos: le duo *code postal* et *localité* viole-t-il la 3ème forme normale (dépendance transitive)? La réponse est OUI dans certains pays dans lesquels le code postal définit exactement la localité, NON dans d'autres, comme la Suisse, où notre NPA (numéro postal d'acheminement) désigne en fait l'office postal et non pas la localité. Un même code peut ainsi désigner plusieurs localités et une localité posséder plusieurs codes. Mieux vaut, face à de telles ambigüités, ne pas se fixer et traiter, comme ici par exemple, les deux attributs comme étant indépendants l'un de l'autre.

Autre exemple de résolution d'un problème de normalisation:

Région-vente (#-région, libellé région, bureau1, bureau 2, bureau 3)

La présence d'une table de bureaux (bureau1, bureau2, …) dans cette définition indique une violation de la 1ère forme normale au sens de la doctrine traditionnelle. Nous avons expliqué au chapitre 6 qu'une telle situation n'est aujourd'hui plus strictement interdite et peut même être gérée avec certains SGBD. Pour des raisons de simplicité conceptuelle, nous déconseillons cependant de procéder de la sorte. De toute manière il est clair ici que la notion de bureau doit être explicitée: bureau est probablement un type d'objet en soi avec ses propres attributs. Nous préférerons donc normaliser:

Région-vente (#-région, libellé région)

Bureau de vente (#-bureau, nom, adresse, …, *#-région*)

Et déjà nous avons créé un nouveau type d'objet élémentaire (bureau) et une autre association!

Soulignons à propos de cet exemple que, dans une application commerciale, la gestion de territoires/régions présente la même complexité (hiérarchie) que celle évoquée au sujet de la classification des articles.

Résumé de la méthode et commentaires

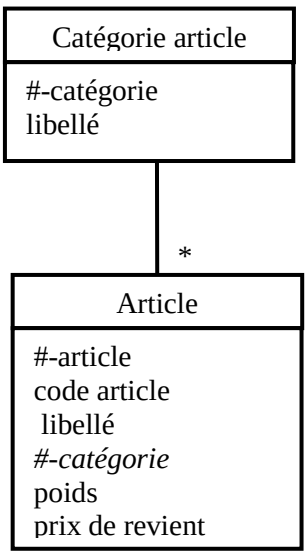
Le but de cette première étape est donc de découvrir les types d'objet de base du système, les "piliers" de la base de données, les tables principales auxquelles viendront se lier toutes les autres. Généralement ce n'est pas très difficile. Plus compliqué par contre est de spécifier correctement la liste de leurs attributs.

Répétons: personne n'est obligé de procéder strictement de cette manière, il ne s'agit ici que d'une proposition pour faciliter aux débutants le démarrage du processus de conception.

Il est presque certain, d'ailleurs, que d'autres types d'objet de base seront découverts par la suite.

Jusqu'à présent, nous n'avons pas dessiné de schéma graphique comprenant les éléments présentés au chapitre 3. Ceci simplement parce que de tels schémas ne présentent vraiment un intérêt qu'à partir du moment où ils contiennent des associations. Et puisque nous avons déjà défini des associations, lançons nous:

Certaines personnes aiment bien, et certains outils de modélisation de données permettent, de faire figurer la liste des attributs dans le rectangle représentant le type d'objet. Ainsi:



Nous n'avons aucune difficulté avec cette façon de procéder. Le problème ne se présente que lorsque la liste des attributs devient très longue, ce qui est souvent le cas avec les types d'objet les plus fondamentaux. Nous n'inclurons ici en général pas la liste des attributs dans le rectangle représentant le type d'objet, mais établirons, séparément du schéma (comme nous l'avons fait plus haut), une liste des différents attributs de chaque type.

Une autre question se pose alors, dont nous avons déjà parlé au chapitre 2. Faut-il, ose-t-on, dans une telle représentation faire figurer les clés étrangères, les attributs qui décrivent une liaison, dans la liste des attributs (exemple: *#-catégori*e dans *article* dans l'exemple ci-dessus)? "NON", hurlent certains théoriciens, "les associations sont décrites par la représentation graphique!"

Nous estimons pour notre part qu'il s'agit d'un reliquat du temps des SGBD hiérarchiques et en réseau. Aujourd'hui, presque 100% des conceptions se font en vue d'une implémentation avec un système relationnel et, dans le relationnel, les associations sont décrites au moyen des clés étrangères. De plus, nous ne faisons pas de distinction entre schémas conceptionnel, logique et physique. Le modèle terminé est constitué par la liste des définitions des types d'objet avec leurs attributs, clés étrangères comprises.

Cela ne signifie pourtant pas que le schéma graphique n'est pas utile ou important. Bien au contraire! Nous l'établissons en général en même temps ou même avant la liste des types d'objet. C'est un instrument très puissant pour concevoir et pour valider la structure de la base de données. Nous en donnerons maints exemples.

Dans cet ouvrage, nous présentons donc les spécifications de schémas de base de données sous une double forme. D'une part un schéma montrant les types d'objet et leurs associations. D'autre part la liste des types d'objet, chacun avec sa liste d'attributs, y compris la clé primaire et les clés étrangères décrivant les associations. Complétée plus tard par les types des attributs et les indications nécessaires pour préciser les rôles des clés (PRIMARY KEY pour une clé primaire, REFERENCES Table(attribut) pour une clé étrangère), cette liste peut directement servir à générer la base de données.

2. Étude et description des types d'objet associés

Une fois les principaux types d'objet indépendants extraits de l'inventaire des données, il faut intégrer dans le schéma les types d'objet plus complexes figurant dans cet inventaire. Les types donc qui ne peuvent pas avoir d'existence indépendante, mais sont pas définition liés à d'autres types. Il faut ensuite définir les associations de ces types d'objet avec les autres types du modèle et spécifier la nature (cardinalité) de ces associations. Ces types d'objet sont ajoutés au schéma graphique avec leurs associations et à la liste des types d'objet avec tous leurs attributs.

À ce stade, également, il s'agit de spécifier chaque type d'objet au moyen de sa liste d'attributs, de lui attribuer une clé primaire et de faire figurer dans la liste des attributs les clés étrangères qui décrivent les associations avec d'autres types d'objet. Et ici également, il faut contrôler l'absence de violations d'une forme normale.

Exemple

À titre d'exemple nous reprenons l'exemple présenté au chapitre 9. Nous allons effectuer le travail de conception pas à pas et présenter tous les raisonnements qui s'effectuent normalement lors d'un tel travail. L'inventaire des données d'une facture nous avait fourni les types d'objet suivants:

Facture avec: client, en-tête de facture, plusieurs lignes de facture, bas de facture

Client avec: raison sociale, rue, code postal, localité.

En-tête facture avec: no facture, date facture, no commande client, référence client, data commande client, no livraison, date de livraison.

Ligne facture avec: no article, libellé article, quantité, prix unitaire, rabais, montant.

Article avec: no article, libellé, prix unitaire.

Bas de facture avec: total marchandise, frais d'emballage, frais d'envoi, total soumis TVA, taux TVA, montant TVA, total facture, conditions paiement avec escompte éventuel, délai de paiement.

Cet inventaire est évidemment incomplet puisqu'établi à partir d'un seul document. Nous nous en contenterons pourtant pour ce premier exemple et verrons à la fin que le résultat n'est pas vraiment satisfaisant parce que nous avons omis de voir le problème dans son ensemble.

Nous effectuons donc le raisonnement suivant:

Facture dépend de *client*, *en-tête*, *ligne* et *bas de facture* font partie de *facture*, *ligne de facture* dépend également d'*article*, ce ne sont pas des type d'objet indépendants.

Par contre client et article le sont. Nous les faisons donc figurer dans le schéma et dans la liste (avec une clé primaire!). Tous les types d'objet restent ouverts pour leur ajouter des attributs par la suite!

Article (#-article, code article, libellé, prix unitaire

Client (#-client, numéro client, raison sociale, rue, code postal, localité

Nous avons créé de nouveaux attributs neutres comme clés primaires de ces types, ni code article (alphanumérique) ni numéro client (attribué par région) étant neutres, quoique uniques.

Le graphique ne contient pour le moment que ces deux types d'objet.

Ajoutons à présent les types d'objet comprenant des dépendances.

Facture dépend de *client*. Nous reprenons la liste des attributs provenant de l'inventaire et y ajoutons la clé étrangère *#-client*.

Facture (#-facture, no facture, *#-client*, date facture, no commande client, référence client, date commande client, no livraison, date livraison

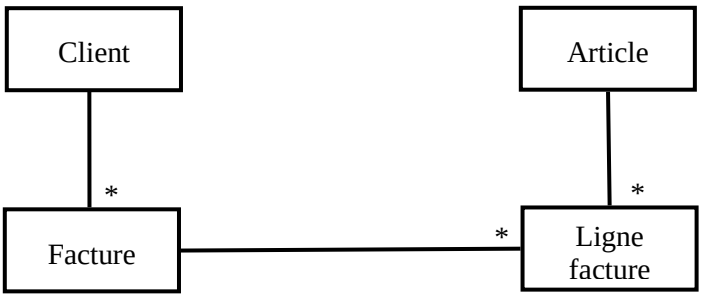
Ici également, nous avons créé comme clé primaire un nouveau numéro de facture neutre, celui utilisé dans l'entreprise étant alphanumérique et contenant comme deux premiers chiffres l'année dans la décennie (AA-nnnn), un système qui pausera problème si l'on garde les factures pendant plus que 10 ans.

Le schéma se présente pour le moment ainsi:

En-tête, *pied* et *ligne de facture* sont des composants de *facture*.

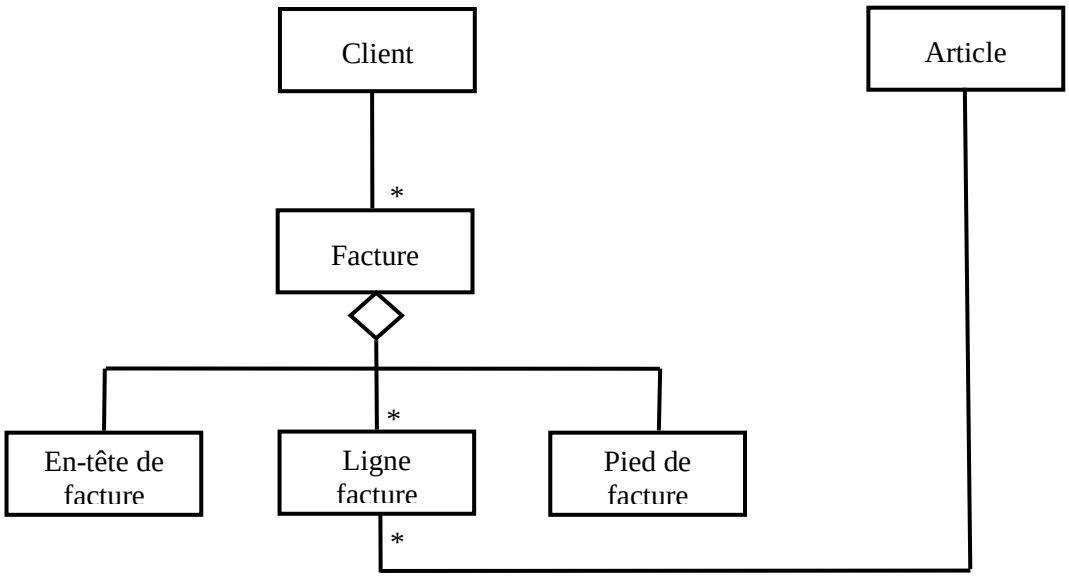
Du fait qu'une facture comprend plusieurs lignes de facture, nous devons considérer la ligne de facture comme un type d'objet séparé. *Ligne de facture* est également associé à *article*. Chaque ligne de facture ne comprend qu'un article, mais le même article peut évidemment apparaître dans plusieurs lignes de factures (de factures différentes ou même d'une seule facture). Pour ce qui est de l'en-tête et du pied, deux possibilités de solution se présentent: d'une part nous pouvons simplement intégrer leurs attributs à *facture*: *en-tête* et *pied* ne deviennent alors pas des types d'objet dans le schéma de la base. C'est l'approche relationnelle pure.

Le schéma se présente ainsi comme suit:



La deuxième solution fait usage de la notion sémantiquement plus précise de composition, reprise de l'approche objets et de la notation UML. Elle dit: une facture se *compose* d'un en-tête, de plusieurs lignes et d'un pied. Nous avons déjà étudié cette notion au chapitre 3, où nous l'avons nommé agrégation, synonyme de composition.

Nous représenterions alors les choses ainsi:



Dans ce cas précis, cette seconde solution n'apporte pas vraiment de plus-value. Nous nous contenterons de la version purement relationnelle de la page précédente.

La définition du type d'objet *facture* reste ouverte et nous avons un nouveau type d'objet: *ligne facture*.

Ligne facture (*#-facture*, #-ligne, *#-article*, quantité, prix unitaire, rabais, montant

Clés étrangères

La clé étrangère *#-facture* décrit l'association avec une facture et la clé étrangère *#-article* décrit l'association avec un article. Petit truc qui aidera les débutants: la clé étrangère se trouve toujours du côté "plusieurs" d'une association.

Choix de la clé primaire

Nous aurions pu définir une nouvelle clé primaire, un numéro de ligne universel (unique, jamais réutilisé) et ne pas faire figurer *#-facture* dans la clé primaire. Nous préférons la solution présentée ici: à savoir une clé concaténée composée du numéro de la facture et d'un numéro de ligne qui débute à 1 pour chaque facture. Cela nous permet en même temps de donner une séquence définie aux lignes dans chaque facture, de les réorganiser si nécessaire. Notons au passage qu'une clé primaire peut parfaitement être composée d'une ou plusieurs clés étrangères

Normalisation.

Nous avons déjà éliminé de *ligne facture* l'attribut *libellé article* qui dépend de *#-article* et non pas directement de la clé primaire (violation de 3ème forme normale). Nous ferons figurer les libellés articles en un seul endroit: dans la table des articles où nous irons les chercher lors de l'affichage ou de l'impression d'une facture. Par le passé, on aurait répété ce libellé dans chaque ligne de facture, ceci pour des questions de rapidité d'exécution. Aujourd'hui cette pratique est bannie. Faire figurer le libellé de l'article dans la ligne de facture serait acceptable si, dans le contexte de l'application envisagée, ce libellé pouvait dans certains cas être différent de celui stocké avec l'article, par exemple si un client très important exigeait qu'on fasse figurer ses propres libellés sur la facture. Mais, même dans ce cas, il existerait d'autres solutions pour résoudre ce problème. Et l'impression sur la facture du libellé dans la langue du client n'est pas non plus une bonne raison pour le faire figurer au sein de *ligne facture*.

Qu'en est-il de *prix unitaire*? Peut-on y appliquer le même raisonnement? La réalité nous apprend que la situation est différente. *Prix unitaire* dans *ligne facture* est le prix unitaire facturé au client dans la présente facture. *Prix unitaire* dans article est un prix catalogue. Depuis que la facture a été émise, ce prix catalogue a peut-être changé. Or nous devons absolument conserver la trace du prix, ne serait-ce que pour des raisons de révision comptable ou pour pouvoir réagir en cas de contestation. Le prix facturé pourrait aussi être différent suite à un accord avec le client. Bien sûr on a toujours, dans cet exemple, la possibilité de modifier le montant facturé au moyen de la valeur de *rabais*. Mais cet attribut décrit en pratique souvent un rabais de quantité, mieux vaut donc indiquer le prix de base appliqué et le rabais appliqué. Les discussions en cas de contestation seront ainsi simplifiées.

Qu'en est-il de *montant*? Il s'agit bien sûr d'un attribut dérivé, dont la valeur peut en principe se calculer à partir de celles d'autres attributs. À éliminer, donc, à moins que, dans certains cas, on veuille garder la possibilité de faire figurer sur la facture des lignes ne contenant pas vraiment un article. Mais il existe pour cela aussi la possibilité de définir des articles fictifs.

Par contre nous garderons dans facture, à qui nous avons intégré tous les éléments du pied de facture suite à la discussion qui précède: *total marchandise*, *total soumis TVA*, *taux TVA*, *montant TVA* et *total facture*. Taux TVA est indispensable, car ce taux peut changer d'une année à l'autre et il faut connaître le taux appliqué à *cette* facture. Les autres grandeurs pourraient effectivement être recalculées, mais restent indispensables parce que représentant des valeurs contractuelles et comptables. Elles seront nécessaires pour la révision, où l'on ne s'intéressera pas aux lignes de la facture, mais où l'on voudra extraire de la base de données avec un outil de requête neutre les principales données de la facture. Considérons également que des règles sous-jacentes ont pu être été utilisées pour arrondir ces valeurs et que ces règles pourraient changer.

Les discussions qui précèdent illustrent qu'il est important lors de l'examen de la normalisation de tenir compte de la réalité du monde dans lequel nous vivons et de ne pas se laisser piéger par une approche purement scolaire.

La liste des types d'objet avec leurs attributs se présente donc ainsi:

Article (#-article, code article, libellé, prix unitaire

Client (#-client, no client, raison sociale, rue, code postal, localité

Facture (#-facture, *#-client*, date facture, no commande client, référence client, date commande client, no livraison, date livraison, total marchandise, frais d'emballage, frais d'envoi, total soumis TVA, taux TVA, montant TVA, total facture, escompte, délai de paiement

Ligne facture (*#-facture*, #-ligne, *#-article*, quantité, prix unitaire, rabais, montant

Malheureusement nous ne sommes pas encore au bout de nos peines!

Examinant *facture* avec un œil critique, on voit qu'il y apparaît différents attributs liés à une *commande* et à une *livraison*. Un inventaire des données correctement effectué aurait déjà dû nous mettre la puce à l'oreille. À moins que le système envisagé soit ici une pure application de facturation sans lien avec d'autres domaines de l'entreprise, nous avons manqué d'analyser le domaine des commandes et celui des livraisons. Apparemment, si l'on se fie totalement à l'inventaire effectué, une facture ne se rapporte qu'à une seule commande et à une seule livraison. Chaque livraison donne lieu à une facture. Par contre une commande pourrait susciter plusieurs livraisons partielles, chacune avec sa facture. Si cela n'a pas été fait précédemment, il est urgent de clarifier ces points.

Pour en finir avec cet exemple, nous ajoutons donc au modèle les deux types d'objet *commande* et *livraison*, le premier rattachés à *client*, le second à *commande*. La tâche n'est évidemment pas terminée puisqu'il faudrait à présent spécifier ce que ces types contiennent en dehors des attributs découverts par l'analyse de la facture.

Notre base de données (incomplète) se présente à présent comme suit, Nous avons relié chaque facture à une livraison et chaque livraison à une commande et chaque commande à un client. Nous avons aussi supprimé de *facture* les attributs dépendant de la commande et de la livraison.

Article (#-article, libellé, prix unitaire

Client (#-client, raison sociale, rue, code postal, localité

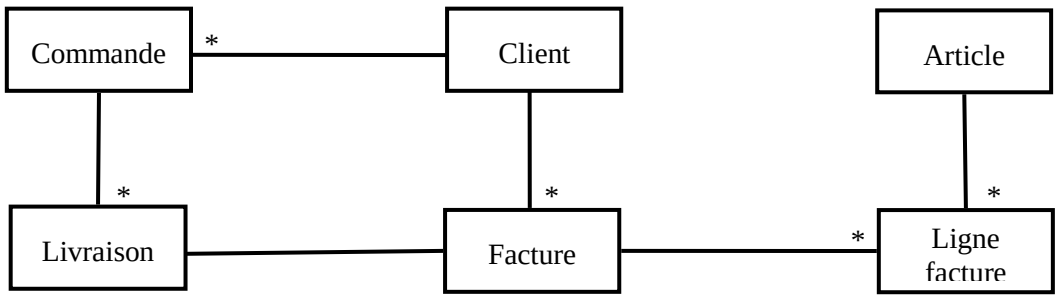
Commande (#-commande, *#-client*, date commande, référence client)

Livraison (#-livraison, *#-commande*, date livraison)

Facture (#-facture, *#-client*, date facture, *#-livraison*, total marchandise, frais d'emballage, frais d'envoi, total soumis TVA, taux TVA, montant TVA, total facture, escompte, délai de paiement

Ligne facture (*#-facture*, #-ligne, *#-article*, quantité, prix unitaire, rabais, montant

Et le schéma:



Deux remarques encore:

1. Nous avons une association de type *un à un* entre facture et livraison. Une telle situation n'est en générale pas bienvenue et la tentation existe de fusionner les deux types d'objet. Au niveau de la gestion, il s'agit pourtant bien de deux objets différents et nous refusons de les fusionner. La philosophie de l'orientation objets laisse ses traces: on cherche aujourd'hui à décrire dans le système les objets de la réalité, alors que la normalisation pure et dure a beaucoup perdu de son importance. Le problème est cependant que la solution adoptée n'empêche pas, au niveau de la base de données, du moins, d'associer plusieurs factures à une livraison. Nous reviendrons sur ce problème au chapitre prochain.

2. Théoriquement il n'est plus nécessaire de lier la facture au client, vu que chaque facture est liée à une livraison, donc à une commande, donc à un client. Mais rappelons qu'une facture est une pièce comptable complète. Elle doit contenir tous les éléments qui permettent son analyse.

Dans le chapitre qui suit nous allons étudier de plus près les associations entre types d'objet.