# 14. L'apport de l'orientation objets

Dans ce chapitre nous discutons de la possibilité et des moyens d'incorporer dans la modélisation des données certains aspects de l'orientation objets (OO), notamment les concepts de spécialisation et d'agrégation.

Le début des années 1990 a vu l'explosion de l'orientation objets, devenue une de ces modes comme on en voit régulièrement passer en informatique.

Au niveau de la programmation, l'apport est évident. Les langages C++, Smalltalk, puis surtout Java et C# ont transformé la manière de développer des programmes et constituent les standards d'aujourd'hui. Les grands "gourous" de l'informatique ont ensuite cherché à étendre les concepts OO à d'autres domaines. CORBA, la gestion d'objets distribués, a connu son heure de gloire, mais est aujourd'hui abandonné au profit d'XML et de SOA. On a également pensé que l'orientation objets allait révolutionner le domaine des systèmes de gestion de bases de données et quelques produits très intéressants ont en effet émergé, mais n'ont jamais réellement pu s'imposer sinon dans des domaines de niche. Oracle a incorporé ces concepts à son produit phare, IBM également et SQL:1999 a sanctionné la chose. Nous estimons que, dans la réalité, l'utilisation de ces fonctions objets reste marginale. Coad, Yourdon, Booch, Rumbaugh, Jacobson et d'autres ont décrit l'analyse et la conception orientée objets Le langage de modélisation UML est aujourd'hui le standard de facto pour représenter un système.

Qu'en est-il au niveau de la modélisation des données? Les concepts de spécialisation (et la notion d'héritage qui lui est associée) et d'agrégation paraissent évidemment très intéressants pour décrire de manière sémantiquement plus précise les relations entre objets.

Pour résumer la situation, affirmons dès le départ que la notion d'héritage, si utile en programmation, l'est malheureusement beaucoup moins pour la modélisation et la conception d'une base de données. Cela ne veut absolument pas dire qu'elle est inapplicable ou ne sert à rien du tout. L'expérience nous montre simplement qu'il arrive fréquemment que la définition d'une hiérarchie de classes d'objets paraisse intéressante à première vue, mais qu'il faille déchanter par la suite et adopter une autre approche parce qu'une telle solution apporte plus de problèmes que de bénéfices ou alors ne permet pas du tout de couvrir les besoins de la situation.

Nous verrons par contre que l'apport de la notion d'agrégation est beaucoup plus intéressant et que la notion d'agrégation peut se substituer avec bénéfice au concept d'héritage. Cela paraîtra peut être bizarre aux personnes familières de la programmation orientée objets, mais présenterons ci-dessous plusieurs exemples parlants.

Si l'on s'en tient par ailleurs aux seules possibilités offertes par les systèmes de base de données non-étendues objets, les deux concepts (spécialisation et agrégation) seront mis en œuvre de manière identique, comme nous allons le voir. Il s'agit donc plutôt de façons différentes de visualiser les choses que de les réaliser.

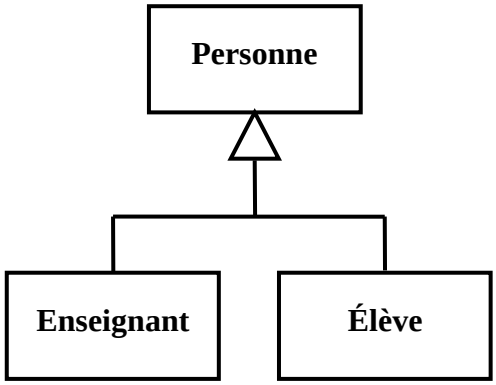
Du fait que cet ouvrage est aussi destiné aux non-informaticiens, nous renonçons ici à expliciter formellement les notions agrégation, généralisation-spécialisation et héritage, préférant les illustrer et les analyser au moyen d'exemples concrets. Voir aussi le chapitre 3 pour la représentation de ces concepts.

## Application du concept de généralisation – spécialisation (héritage)

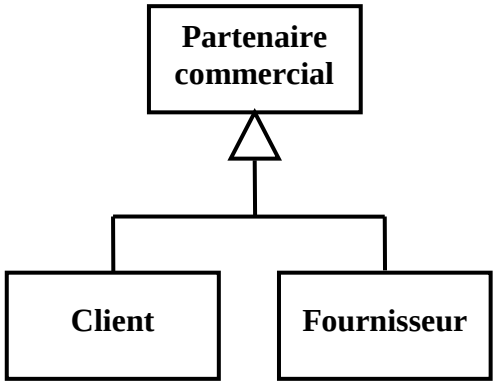
Nous l'avons déjà dit: l'apport du concept d'héritage, donc de spécialisation – généralisation, est malheureusement minime dans le domaine de la modélisation des données. Il permet toutefois souvent de mettre les idées au clair. Nous allons donc tout de même y consacrer quelques paragraphes.

Dans ce premier exemple, il s'agit de gérer des personnes dans une école. Le concept de personne couvre ici deux catégories d'individus: les enseignants et les élèves. (Dans la réalité, on pourrait en avoir encore d'autres: personnel administratif et technique, membres du conseil de surveillance, etc.). Mais commençons par une situation très simple.

Le schéma suivant décrit ce fait. Il indique que le type d'objets *personne* est une généralisation de deux types plus spécialisés: *enseignant* et *élève*. Un enseignant **est** une personne et un élève **est** une personne. En orientation objets, on s'intéresse aussi bien aux attributs des objets qu'aux opérations s'appliquant à ces objets. Ce livre ayant pour sujet la modélisation des données, nous ne parlons ici que des attributs. Justement: *enseignant* et *élève* ont des attributs communs: nom, prénom, adresse, etc. Ces attributs communs sont définis dans *personne*. Ils ont également des attributs distincts: fonction, liste de branches, diplôme, date d'engagement, etc. pour *enseignant* et année d'étude, voie d'étude, nom des représentants légaux, etc. pour *élève*.




Présentons un deuxième exemple semblable:




Dans *partenaire commercial*, on spécifiera tous les attributs communs aux deux sous-types: l'adresse avec tous ses détails, la personne de contact, etc. Par contre *client* et *fournisseur* contiennent les attributs spécifiques au deux sous-types: un délai de livraison et une note d'appréciation chez le fournisseur, un rabais accordé et la fidélité de paiement chez le client.

**Implémentation de la généralisation – spécialisation dans un modèle relationnel simple**

Comment implémenter de telles situations dans une base de données relationnelle? Une solution évidente est de créer trois tables. Pour l'exemple de l'école:

Personne (#-personne, nom, prénom, sexe, date de naissance, adresse, …

Type enseignant (*#-personne*, fonction, diplôme, date diplôme, entrée en fonction, liste (branches), …

Type élève (*#-personne*, voie, année d'étude, …

Nous appelons la deuxième et la troisième table *type...* parce que ces types d'objets ne sont en fait que des compléments à la table *personne* et ne contiennent pas toutes les données d'un enseignant ou d'un élève (il manque les données liées à la personne).

Nous avons ensuite la possibilité de définir des vues: *élève* contenant les élèves seulement avec l'ensemble de leurs données provenant de *personne* et de *type_élève*; *enseignant* contenant les enseignants avec toutes leurs données.

```
CREATE VIEW élève AS SELECT * FROM personne, type élève
        WHERE personne.#-personne = type élève.#-personne
```

Dans les exemples qui suivent, nous dirons simplement:

Élève = jointure (personne, type élève)

L'opération mathématique de jointure consiste à réunir les attributs des deux tables au moyen de la valeur commune du numéro de personne.

**Plusieurs commentaires s'imposent:**

L'avantage principal de la solution envisagée est de nous fournir une table de personnes, de toutes les personnes, qu'elles soient élèves, enseignants ou autre chose. Ensuite, seuls les élèves possèdent une entrée dans la table *type élève* et les enseignants une entrée dans la table *type enseignant*. Mais pour arriver à cette solution, nul besoin d'être maître ès orientation objets!

Dans *type enseignant* et *type élève*, la clé primaire *#-personne* est également clé étrangère. Il n'est donc pas être possible de définir d'enseignant ou d'élève qui ne soit pas d'abord défini comme personne. Le langage de définition de données (DDL) nous permet d'imposer cette règle, comme pour toute clé étrangère.

Techniquement, le schéma ci-dessus décrit des associations entre un objet *personne* et un objet *type enseignant* et/ou *élève*. Mais il ne nous oblige pas (seule la programmation de règles pourrait le forcer) à associer un objet *type élève* ou *enseignant* à chaque objet *personne*; la base pourrait donc contenir des personnes n'étant ni élève ni enseignant. Le modèle ne nous empêche pas non plus de définir des personnes qui sont à la fois élève et enseignant.

Cette dernière remarque soulève une question de principe. Les sous-types d'un type d'objets doivent-ils être disjoints, ou peut-il y avoir des recoupements? Une personne peut-elle à la fois être enseignant et élève. Dans une école, généralement pas. Dans l'exemple des partenaires commerciaux illustré ci-dessus, c'est par contre possible: un client d'une entreprise peut parfaitement être aussi fournisseur.

L'orientation objets pure et dure interdit les recoupements de sous-types (certaines modélisations envisagent cette possibilité, mais aucun langage réellement utilisé ne le fait). Le schéma *partenaire commercial, client, fournisseur* présenté ci-dessus est donc inadéquat pour décrire la réalité.

**À éviter absolument:**

Réunir toutes les données *personne*, *type élève* et *type enseignant* en une seule table comprenant l'ensemble des attributs avec le choix

a) de laisser les attributs *élève* vides chez un enseignant et les attributs *enseignant* vides chez un élève

ou

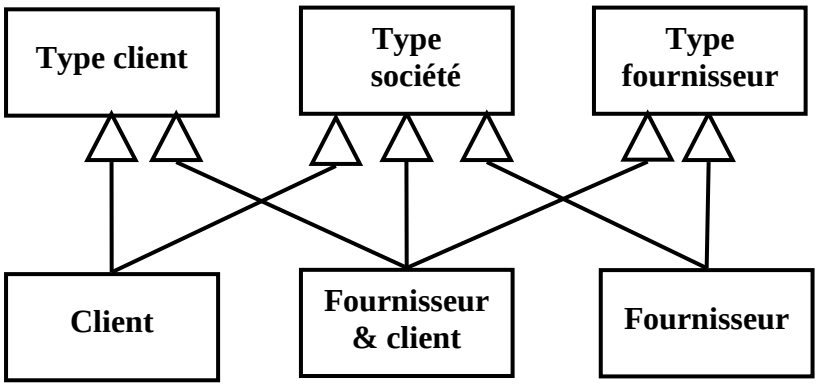
b) pire encore, d'utiliser les mêmes colonnes dans la table *personne* pour y loger les attributs *élève* chez un élève et les attributs *enseignant* chez un enseignant (pratique courante aux temps héroïques des bases de données).

**L'héritage multiple pourrait-il nous être utile?**

Ce paragraphe n'est pas destiné aux débutants!

L'héritage multiple représentait le nec plus ultra de l'orientation objets il y a une vingtaine d'années. Le langage C++, en particulier, permettait de le mettre en œuvre. Entre-temps, cette approche est tombée en disgrâce et Java, par exemple, ne l'implémente plus.

Mais l'héritage multiple nous permet, en principe, de voir d'une autre manière le problème des clients – fournisseurs mentionné plus haut.




*Client* hérite donc de *type client* et *type société*, *fournisseur & client* de *type client*, *type société* et *type fournisseur*, *fournisseur* de *type société* et *type fournisseur*.

SQL:1999 nous permet bien de définir des sous-tables d'une table, mais chaque table ne peut avoir qu'une super-table. Et diviser les partenaires commerciaux en trois catégories comme ci-dessus (et donc faire trois tables distinctes) nous paraît particulièrement indésirable. Surtout qu'un partenaire exclusivement client un jour peut devenir fournisseur le lendemain. Et inversement.

Voie sans issue!

**Cas de l'hôpital**

Il s'agit ici de gérer les personnes dans un système hospitalier. On pourrait penser à première vue que ces personnes se sous-divisent en patients, médecins, soignants, techniciens, etc. Et la tentation sera peut être forte de dessiner un schéma hiérarchique tel que nous l'avons fait pour l'exemple élèves – enseignants ci-dessus. Seulement, un soignant peut devenir patient, de même qu'un médecin. Et si nous voulons gérer en plus la notion d'employé, on aura des médecins employés et des médecins externes qui, chacun, peuvent devenir patients, etc. Plus on y réfléchit, plus on se rend compte qu'aucune hiérarchie simple ou complexe n'est en mesure de décrire la réalité. La sous-division des personnes en catégories (les spécialiser au sens de l'orientation objets) n'est pas une solution applicable.

**Conclusion**

Nous l'avons déjà dit: le concept d'héritage (spécialisation – généralisation) de l'orientation objets, si puissant et utile en programmation, n'est pas d'une grande utilité lors de la modélisation des données. Tous les exemples présentés dans les manuels, semblables à ceux qui figurent ici, ne fonctionnent malheureusement que pour les cas les plus triviaux.

Si un lecteur connaissait un bon exemple du contraire de ce que nous affirmons, nous le prions instamment de nous contacter (voir le site www.jlpi.ch) pour nous faire changer d'avis.

## Application du concept d'agrégation (composition)

La nécessité de décrire une agrégation (le fait qu'un objet se compose de ou contienne d'autres objets) est évidemment beaucoup plus vieille que l'orientation objets. Mais c'est dans le cadre de ce paradigme que l'agrégation a été reconnue et formalisée.

Dans plusieurs situations, il est sémantiquement plus précis de parler d'agrégation que d' association.

Aux chapitres 10 et 11, nous avons déjà présenté des exemples d'agrégation.

Schéma relationnel classique:

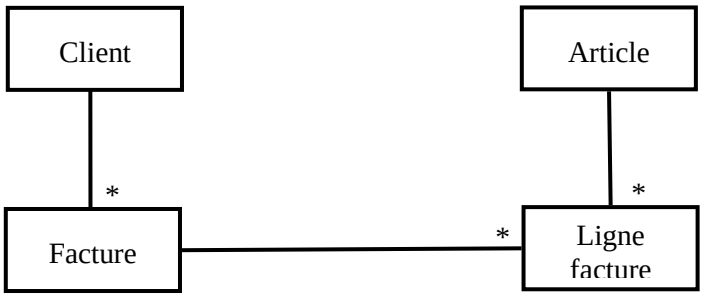
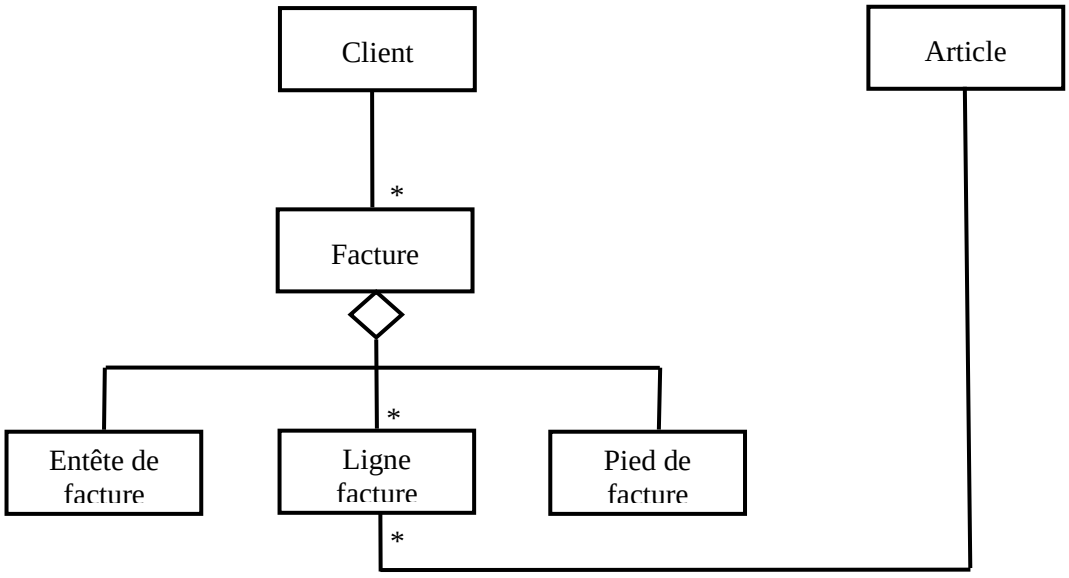



Schéma prenant en compte la notion d'agrégation:




Dans ce deuxième schéma, nous précisons qu'une facture se **compose** d'un entête, de plusieurs lignes et d'un pied, alors que dans le premier, on **associe** *ligne de facture* à *facture*.

Traditionnellement, sans la notion d'agrégation, on définit un type *facture*, un type *ligne facture* et une association 1 à plusieurs entre eux. Les attributs d'*entête facture* et *pied de facture* sont intégrés à *facture*. La présence de *#-facture* sous forme de clé étrangère dans *ligne facture* garantit qu'aucune ligne n'existe sans être rattachée à une facture.

La traduction du schéma ci-dessus intégrant la notion d'agrégation en définitions de types d'objets n'est pas vraiment différente de la solution traditionnelle. Sauf besoin exceptionnel que nous ne pouvons véritablement imaginer, nous ne voyons pas l'utilité de définir quatre types d'objets différents: *facture, entête, ligne, pied*. *Entête* et *pied* ayant certainement pour clé primaire simplement #-*fact*ure, il n'existe aucune raison de ne pas intégrer les attributs de ces types à une table unique. Même si le second schéma ci-dessus est donc plus précis que le premier, il n'apporte pas vraiment de plus-value au niveau de la définition des tables.

Au chapitre 11, nous avons parlé de l'agrégation en relation avec les listes décrivant la composition d'un agrégat (montage, article composé): c'est le problème des nomenclatures.

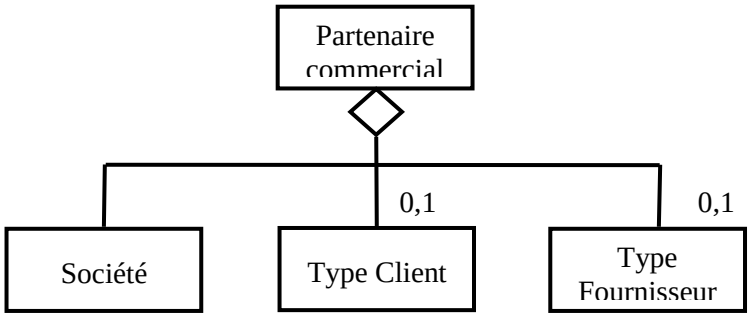
**Utilisation du concept d'agrégation pour décrire des sous-ensembles d'objets.**

Nous avons constaté dans la première partie de ce chapitre l'inaptitude du concept de spécialisation (héritage) à décrire de façon exploitable la catégorisation d'objets en sous-classes. Par contre, le concept d'agrégation vient ici fortement à notre aide.

**Principe: Décrire les propriétés d'un sous-ensemble d'objets non pas au moyen du concept de spécialisation, mais avec le concept d'agrégation.**

Dire: un élève est une personne qui possède aussi les attributs du type élève (élève = personne + type élève). Un enseignant est une personne qui possède aussi les attributs du type enseignant. Un client est une société qui possède également les attributs du type client, un client – fournisseur est une société qui possède aussi les attributs type client et type fournisseur.

Nous pouvons décrire cette situation ainsi:




Lire ce schéma ainsi: un partenaire commercial est une société qui peut aussi être client et/ou fournisseur.

Remarque: ce schéma n'empêche pas de définir une société qui ne serait ni client ni fournisseur. Par exemple une banque ou une assurance avec laquelle travaille cette entreprise (techniquement ce sont aussi des fournisseurs, mais pas dans le cadre de l'activité commerciale quotidienne).

Pour implémenter ce schéma au plan relationnel, nous définissons les tables *société, type client, type fournisseur* ainsi que des vues *client* et *fournisseur* qui réunissent les attributs de *société* et de *type client* respectivement *type fournisseur*.

Société (#-société, raison sociale, adresse, personne contact…

Type client (*#-société*, #-client, rabais, région de vente, …

Type fournisseur (*#-société*, #-fournisseur, délai d'approvisionnement, conditions de paiement, …

Client = Jointure (société, type client)

Fournisseur = Jointure (société, type fournisseur)

**Remarque:**

Notons que *#-client* dans *type client* pourrait être clé primaire de ce type d'objets. De même *#-fournisseur* dans *type fournisseur*. Nous préférons le choix de *#-société* (clé primaire et clé étrangère) parce que chaque objet de *type client* ou *type fournisseur* est vraiment rattaché à un objet de *société*. Les objets *type ...* ne sont pas des objets complets, mais des composants qui viennent s'ajouter à un objet *société*.

Cela n'empêche évidemment pas d'identifier un client à la fois par son numéro de société et son numéro de client qui pourraient être différents.

**Autre exemple: personnes dans un hôpital**

Nous pouvons à présent appliquer la même recette à l'exemple de l'hôpital, cité plus haut:

Un *médecin* est une *personne* qui possède également les attributs du *type médecin*.

Un *médecin patient* est une *personne* qui possède également les attributs *type médecin* et de *type patient*.

Un *médecin employé* est une *personne* qui possède également les attributs *type médecin* et *type employé*.

Et ainsi de suite.

L'exemple ci-dessus (sociétés, clients, fournisseurs) nous montre une façon d'implémenter la solution.

**Exemple: attributs techniques d'un objet**

Un problème sur le sujet duquel des tonnes d'encre ont déjà été répandues est celui des caractéristiques techniques des articles dans un système de gestion des articles. Le problème est extrêmement complexe et la solution que nous esquissons ici n'est pas forcément la meilleure ou la plus appropriée pour une application spécifique. Elle sert d'exemple plutôt que de solution prête à l'emploi.

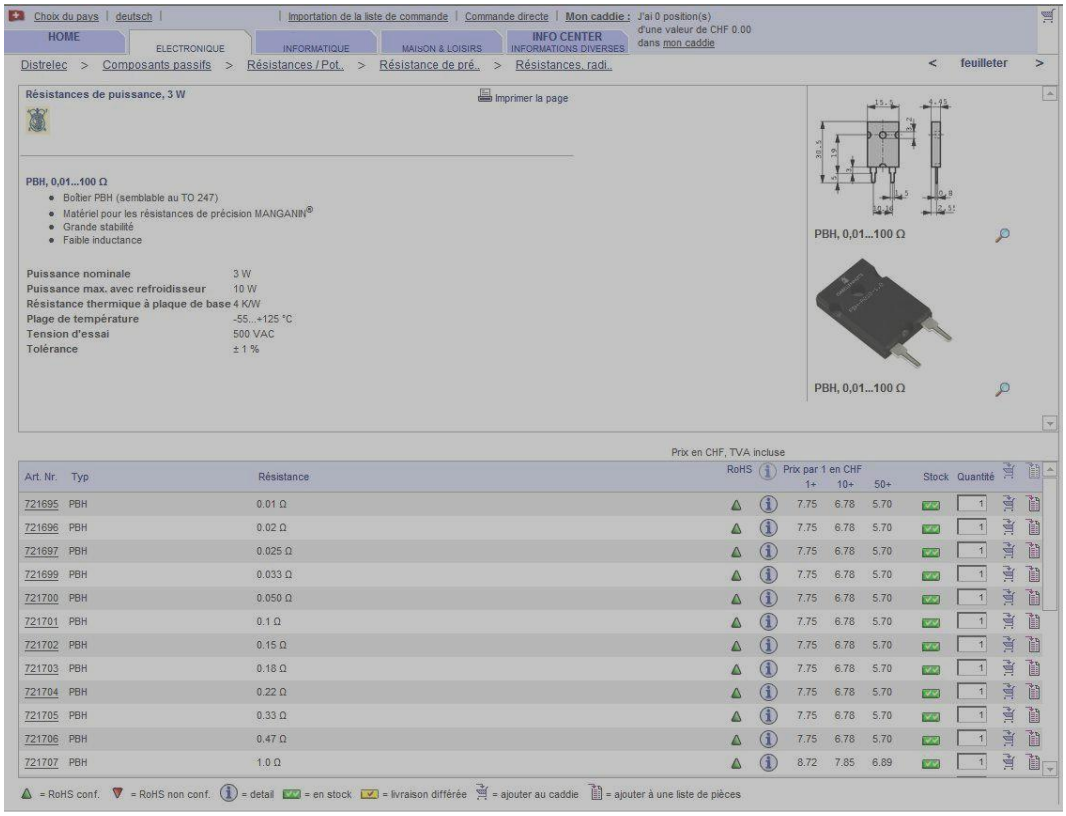
Il s'agit ici de la spécification d'un catalogue d'articles pour un système de vente, par exemple, sur papier ou Internet. Outre les indications habituelles telles que le numéro d'article, le libellé, la catégorie et le prix, le catalogue doit également contenir des données techniques telles que les dimensions, la couleur, le poids et d'autres valeurs dépendant de l'article. Le problème est que les informations à enregistrer peuvent être différentes pour chaque catégorie d'article.

Pour une résistance électrique, il faudra par exemple fournir la résistance, la puissance maximale et la précision. Pour un condensateur, ce sera la capacité et la tension maximale. Pour un transistor, les tensions applicables. Pour une diode, le courant maximal, etc. Voir illustration ci-dessous

Bien sûr il existe des produits de gestion du contenu permettant de créer des catalogues sur Internet. Et le contenu statique de pages telles que celle qui est illustrée ici peut toujours être saisi au moyen d'outils de bureautique. Le problème se pose lorsqu'il faut être capable d'effectuer des traitements avec les valeurs des paramètres de ces articles.

Comment résoudre ce problème?

La première solution est de simplement prévoir pour chaque article une dizaine de champs vides destinés à recevoir ces paramètres. Le problème sera alors de savoir ce que contiennent ces champs pour un article précis, de manière à pouvoir produire des pages de catalogue semblables à celle qui est illustrée. Il faudra donc créer une table des entêtes décrivant que, pour une résistance par exemple, le premier champ désigne la résistance, le second la puissance maximale, le troisième la précision.




Entête (*#catégorie article*, #-propriété, entête)

La deuxième solution consisterait à créer une table des propriétés entièrement dynamique.

Propriété (*#-article*, #-propriété, valeur)

On aurait donc dans cette table une ligne par propriété et par article.

Une table des entêtes est toujours nécessaire.

La troisième solution consisterait à appliquer la notion d'agrégation telle qu'étudiée dans ce chapitre.

Article (#-article, *#-catégorie*, libellé, prix, …

Type résistance (*#-article*, résistance, puissance max, précision, …

Type condensateur (*#-article*, capacité, tension max, …

Etc.

Les tables correspondant aux différents types d'articles se formeront au moyen de définitions de vues comme précédemment.

Condensateur = Jointure (article, type condensateur)

Ici également, une table de définitions des entêtes reste nécessaire.

## Conclusion

L'apport de l'orientation objets se situe principalement au niveau de la conception du schéma, les notions de spécialisation et d'agrégation étant sémantiquement plus précises que la notion d'association.

Au niveau de la réalisation, par contre, on reste toujours limité à la notion d'association décrite par une clé étrangère.

La définition de tables "types" telles que décrites dans les exemples ci-dessus permet, conjointement à la création de vues, d'implémenter de façon intéressante la notion d'agrégation et de résoudre ainsi certains problèmes qui donneraient autrement du fil à retordre.