Le réseau sémantique universel (3ème partie)

© 1997 EPHITEQ

L'intégration de quelques concepts permet d'étendre considérablement les possibilités de modélisation. Voici la suite - et la fin - de l'analyse de cette intégration.
L'auteur vous renouvelle ici ses excuses pour la densité et la non-exhaustivité de cette étude (et pour les égratignures faites à MERISE), mais il s'agit d'une suite d'articles à considérer comme un condensé – et un résumé – d'une étude plus complète qui sera publiée dans un proche avenir.

Erratum : dans le schéma paru dans LA LETTRE n°27, page 31, la patte "possède" entre **OBJET** et l'association **Posséder** doit être fléchée vers **OBJET** (identification relative de **ATTRIBUT**) (Suite à un déplacement graphique, l'extrémité du tracé s'est retrouvée derrière **OBJET**, masquant ainsi la flèche).

Rappel des parties précédentes

Nous avons vu dans la première partie de cet article (La Lettre n° 26) qu'il était possible d'utiliser un seul et unique symbolisme pour modéliser pratiquement n'importe quel système, ceci ayant été présenté avec sept systèmes : Modèle Conceptuel de Données (MCD) de Merise ; Modèle Conceptuel de Traitements (MCT) de Merise ; Système à Base de Connaissance (SBC) ; Tableur (considéré comme un croisement entre une Base de Données et un SBC) ; Système NeuroMimétique (SNM) ; Modèle de Traitements en Temps Réel (TTR) ; Programmation Orientée Objets (POO).

Bien que le symbolisme utilisé soit - à la base - celui des MCD de Merise, il a été sensiblement élargi, prenant en compte les concepts suivants :

- **Objets** (modèles, populations, occurrences, singularités) ;
- **Attributs** (propriétés simples, faits, tableaux, agrégats, méthodes ; appartenant à des objets, des associations, ou des actions) ;
- **Associations** (modèles, populations, particularités, héritages) ;
- **Actions** (les traitements) ;
- **Prédicats** (attributs contribuant à un héritage par spécialisation ou au déclenchement d'actions) ;
- **Pattes** (reliant entre eux objets, associations, actions, prédicats).

Dans la deuxième partie de cet article (La Lettre n° 27), nous avons bâti un métamodèle - incomplet, mais cependant utilisable - qui permet de définir les entités décrivant un modèle, et ajoute diverses notions comme :

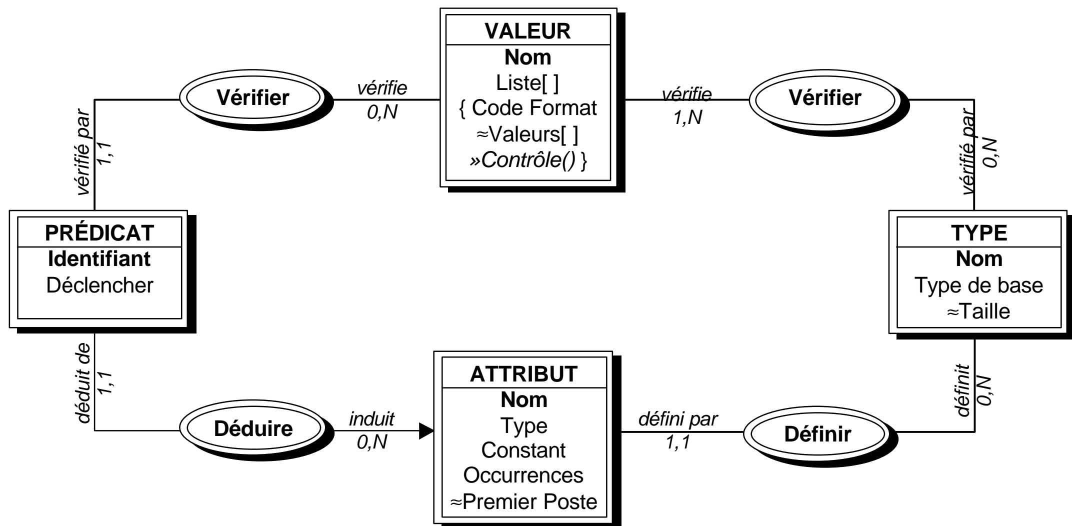
- **Types** (description des attributs-types) ;
- **Valeurs** (contrôle du contenu des attributs et des prédicats) ;
- **Règles** (Actions de type règle de gestion ou de Système à Base de Connaissances) ;
- **Méthodes** (Actions de type méthodes associées à des objets, fonctions, routines...) ;
- **Tâches** (Occurrences de Règles et/ou Méthodes activées pour exécution) ;
- etc.

Nous allons maintenant élaborer un modèle physique qui, implanté en tant que SGBD entité-relation, permettra de concevoir, prototyper et réaliser n'importe quelle application sans programmation.

Ce qui est décrit ci-après correspond à des concepts qui ont été, soit déjà testés dans divers prototypes, soit effectivement mis en œuvre sous des formes approchées dans diverses applications ; il ne s'agit donc pas d'un exercice de style de théorie pure.

Exercice

Avant de plonger dans cette élaboration, un petit rappel. Dans la partie précédente, quand nous avons modélisé les prédicats, nous avions obtenu proposé un exercice[1] concernant le sous-ensemble suivant :



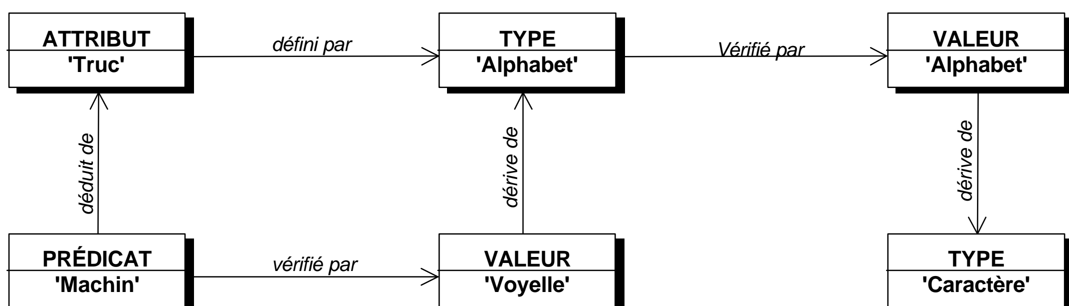
Nous avions dit «*L'association* **Vérifier** *avec* **VALEUR***, qui permet d'associer une liste de valeurs à un prédicat, est sémantiquement légèrement différente de son homonyme avec* **TYPE***, car elle définit en fait un sous-ensemble des valeurs associées au type de l'attribut. Les occurrences de* **VALEUR** *concernées ne sont donc pas identiques, mais il existe entre elles une contrainte, dans le détail de laquelle nous n'entrerons pas maintenant*». Eh bien, voici comment implémenter cette contrainte.

C'est en fait très simple. Il suffit que lors de la création de la listes de valeurs associées au prédicat, une action de validation s'assure que toute valeur fournie soit une valeur possible pour l'attribut.

Comment ? Nous sommes dans un métamodèle, donc nous allons associer à chaque individu VALEUR un TYPE qui définit le format et les valeurs autorisées pour l'attribut VALEURS.

Par exemple, si un attribut TRUC est de type ALPHABET, à ce type sera associée une liste de valeurs limitant la saisie aux valeurs de 'A' à 'Z', elle-même associée au type CARACTÈRE (qui lui n'a pas besoin de liste de valeurs, puisque format primaire). Si l'attribut TRUC a un prédicat qui déclenche une action (ou définit un sous-type de l'objet contenant l'attribut) quand sa valeur correspond à une voyelle, ce prédicat sera relié à une liste de valeurs qui sera elle-même associée au type ALPHABET. Lors de la définition du prédicat, il sera donc impossible de saisir comme valeur un caractère qui ne soit pas alphabétique.

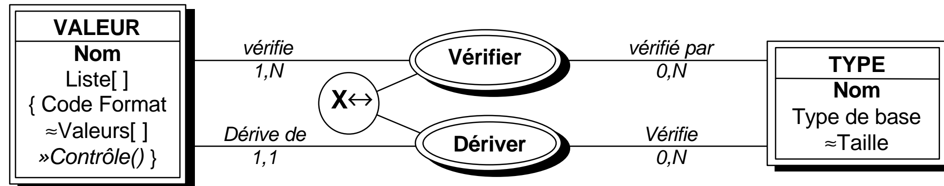
Pour plus de clarté, schématisons cet exemple (sous forme non standard) avec les individus correspondants :



---

[1] *Cf note 17 page 26 de LA LETTRE n°27 :* «Les amateurs pourront essayer de modéliser cette contrainte (objets "objet valeur" associés à des objets "attribut valeur", etc.). Réponse dans la 3ème partie de ce dossier.»

Il nous suffit donc d'enrichir le métamodèle avec une association **Dériver** entre **VALEUR** et **TYPE** , en interdisant les chemins en boucle entre **Dériver** et **Vérifier** :



Certains objecteront que le TYPE n'est ici pas associé à l'objet ATTRIBUT "Valeurs", donc au modèle, mais à l'objet VALEUR, ce qui semble anormal.

En fait, cette "bizarrerie" est associée à un format d'attribut spécial, utilisé dans un certain nombre de L4G et langages objet[2] : le format **VARIANT**. Il s'agit d'un format métamorphe : un attribut de ce type peut effectivement contenir n'importe quel type d'information : son format est celui de son contenu et est résolu non pas à la compilation, mais à l'exécution, et change en fonction de ce qu'on y met.

Bases de raisonnement

Pour élaborer le métamodèle, nous étions partis d'un premier niveau de généralisation (modèle) pour aboutir à un niveau de généralisation supérieur (métamodèle). En principe, nous devrions continuer cette généralisation jusqu'à aboutir à un ensemble suffisamment simplifié pour être réalisé physiquement (à force de paramétrer les paramètres, vient un moment où il faut bien programmer "en dur").

En fait, nous allons suivre maintenant un cheminement inverse : nous allons partir du composant physique le plus simple - ou presque - pour, allant de spécialisation en spécialisation, obtenir une représentation de notre métamodèle. Et, contrairement à ce qui avait été fait pour le métamodèle, c'est-à-dire l'élaborer au fur et à mesure du raisonnement, nous allons ici montrer des ensembles complets et les commenter, la démarche de conception étant de peu d'intérêt.

Notre point de départ sera donc constitué de l'ensemble qu'on retrouve dans tout système manipulant des données : un **Article** (ou enregistrement, segment, record, ligne...) contenant des **Champs** (ou éléments, données, fields, rubriques, colonnes...).

L'article

Qu'est-ce qu'un article ? C'est le plus petit regroupement d'informations accessible au niveau de la base de données physique : un article est toujours physiquement écrit et lu en entier.

Cependant, nous sommes ici dans un environnement complètement objet, et notre base de données est susceptible de contenir les ensembles les plus divers, certains n'existant qu'en un seul exemplaire. Il nous faut donc, pour pouvoir maîtriser ceci, faire au niveau de l'article ce que les SGBD relationnels font au niveau de la table : associer à chaque article un descripteur permettant le décodage de son contenu par une fonction générique.

Chaque article sera donc constitué ainsi :

| Longueur | Type | Numéro | Descripteur | Champs |

Longueur : la taille de l'article, puisqu'elle peut différer pour chacun.

Type et numéro : permettent d'identifier chaque article de façon unique (nous verrons les types plus loin) et de le retrouver directement dans la base sans index intermédiaire.

Descripteur : suite de codes prédéfinis permettant de connaître exactement le contenu de la partie champs, c'est-à-dire pour chaque champ : son type, sa taille (s'il y a lieu). Des codes

---

[2] *Dbase et Clipper (format implicite) ; Visual Basic (type "Object") ; Delphi (type "Variant") ; etc.*

permettent également d'indiquer l'emplacement de champs de longueur nulle, et de savoir pourquoi ils sont absents (valeur inconnue, non renseignée, etc.). De par sa construction, ce descripteur est auto-délimité. La fonction qui lit un article commence donc par décoder ce descripteur, ce qui lui permet de créer une table de description du contenu de la partie champs. En écriture, elle utilise cette table (remplie par une fonction de niveau supérieur) pour assembler le descripteur et constituer l'article. Les codes des formats de base sont "en dur", certains formats élaborés d'usage général peuvent se voir associer un code pour améliorer les performances du système.

Champs : le contenu proprement dit de l'article. Selon son type, un champ pourra correspondre à un attribut, un poste d'index, un pointeur vers un autre article, etc.

Réaliser un programme gérant un fichier constitué d'une telle suite d'articles est très simple. On disposera en fait ici d'une couche logicielle de base, purement technique, qui prendra en charge lecture et écriture des articles, gestion de la configuration de la base, gestion de l'espace disque et des adresses des articles, adressage des sous-ensembles à travers le réseau, partage entre utilisateurs, sécurité et recouvrement, réplication[3]...

Nous n'avons ici qu'un système de gestion de fichiers amélioré. Nous devons donc maintenant, d'une part différencier plusieurs types d'articles de base, qui auront chacun une pré-structure type, et être en mesure de naviguer simplement dans la base, avec un système d'adressage simple. Définissons maintenant ces différents types.

Types d'articles

Commençons par un bref inventaire :

- Tout accès à des données d'un SGBD entité-relation passe par un ou plusieurs objets. Le premier type d'article est donc l'**ENTITÉ** (appelé ainsi pour ne pas faire de confusion avec la classe OBJET, qui définit ceux-ci, mais un article ENTITÉ est bien l'équivalent d'un individu-objet).
- Pour accéder à un objet donné, il faut pouvoir le retrouver en fonction de son identifiant. Nous utiliserons pour ce faire des articles INDEX D'ENTITÉ (que nous appellerons **INDEX-E**).
- Dans certains cas, des objets auront plusieurs identifiants physiques. Nous disposerons donc d'articles **ALIAS-E** et **ALIAS-R** (parce que certaines relations, ayant également un identifiant physique, pourront également avoir des alias).
- Un objet a - généralement - des attributs autres que son identifiant. Ils seront stockés dans des articles **DONNÉES**.
- Certains attributs, de par leur taille et leur nature ne seront pas groupés, mais stockés dans des articles spécifiques : les articles **BLOB**[4].
- Les objets sont reliés entre eux avec des associations. Ceci implique de disposer d'articles **RELATION** (appelés ainsi pour ne pas faire de confusion avec les ASSOCIATIONS, qui sont physiquement constituées de RELATIONS et d'INDEX-R, mais chaque article RELATION est bien l'équivalent d'une occurrence d'association).
- À partir d'un objet, il faut pouvoir retrouver une association ou en obtenir la liste. On ira de l'un aux autres à l'aide d'articles INDEX DE RELATION (que nous appellerons **INDEX-R**).

D'autres types d'articles sont définis, notamment pour le paramétrage et la gestion technique du SGBD, mais ils ne sont pas importants dans la description condensée du présent document.

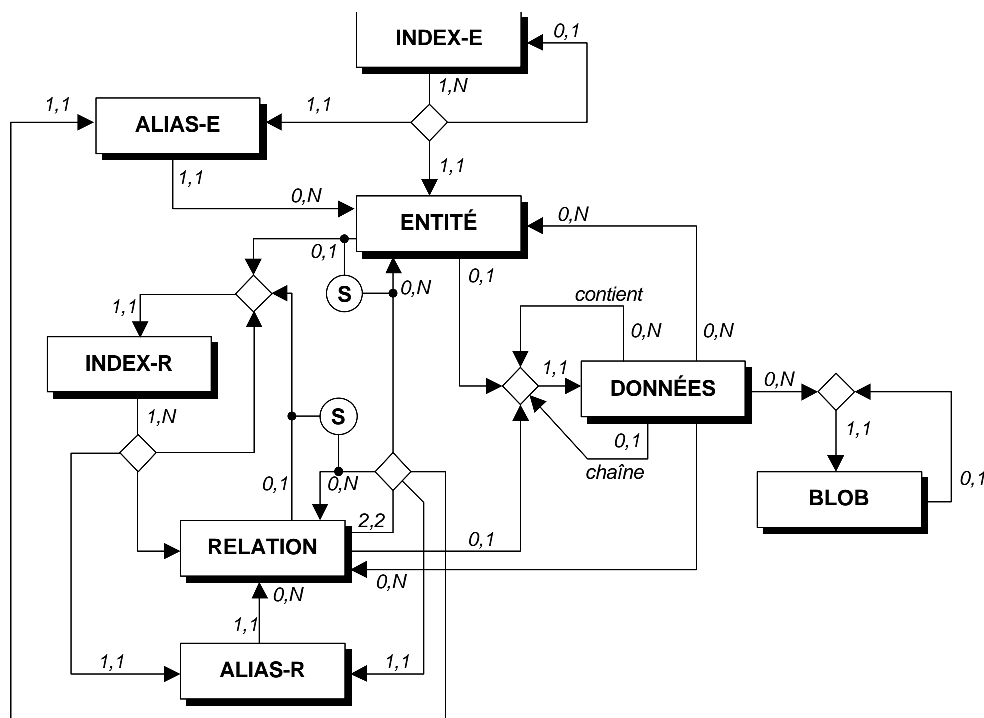
Avant de détailler chacun de ces types d'articles, un Modèle Logique de Données expliquera plus sûrement que deux pages de baratin comment on peut naviguer entre eux. Rappelons simplement que les losanges correspondent à des "pattes" alternatives : on peut ainsi voir qu'un article **DONNÉES** est

---

[3] Il faut savoir que la simple description complète de cette couche et de la suivante fait l'objet d'un dossier EPHITEQ d'une centaine de pages. Ce qui est présenté ici est donc considérablement abrégé… et simplifié.

[4] BLOB = Binary Large OBject, pouvant être un texte, un son, une image, etc.

pointé, soit par un article **ENTITÉ**, soit par un article **RELATION**, soit par un article **DONNÉES (contient)**, soit par un article **DONNÉES (chaîne)**, et par un et un seul d'entre eux.



Les flèches indiquent le sens de parcours, et chacun pourra s'assurer qu'à trois articles-type près (exceptions qui seront explicitées plus loin), toutes les liaisons ont une cardinalité de départ maximale de 1 ou 2, et correspondent physiquement à des pointeurs stockés dans les articles de départ.

Tout pointeur contient simplement l'identifiant de l'article vers lequel il pointe.

Vous avez bien sûr remarqué la présence de deux contraintes de simultanéité. Explication : une entité (resp. relation) qui pointe sur un index de relation a donc des associations. Dans ce cas il existe obligatoirement au moins une relation qui pointe vers cette entité (resp. relation) pour gérer le parcours inverse. CQFD.

Détaillons maintenant chaque type d'article avec ses liaisons.

L'article ENTITÉ

Tout article ENTITÉ a la structure suivante :

| Lg clé | Niveau | Classe | Langue | Champ(s) clé | Ptr→index-R | Ptr→données | Données |

Lg clé : la taille de l'identifiant, puisqu'elle n'est pas la même pour toutes les entités.

Niveau : un code d'état, permettant de différencier des niveaux et/ou versions différentes d'une même entité, sans duplication superflue, et de protéger des domaines. Par exemple, un niveau "*Système*" est attribué aux entités définissant le métamodèle de base, le protégeant de toute altération intempestive ; un autre niveau "*Test système*" sera vu comme l'union du contenu de "*Système*" et des changements effectués, invisibles quand on utilise directement "*Système*".

Classe : un code permettant d'identifier l'appartenance d'une entité, c'est-à-dire l'objet-type. On aura une classe définie pour chaque type d'objet au sens Merise, le nom de la classe étant associé au code.

Langue : un code permettant d'identifier à quelle langue est associé l'identifiant. Par exemple, "OBJET" sera associé au français, "OBJECT" à l'anglais, les deux étant synonymes (par alias). Une valeur spécifique existe évidemment pour spécifier "toutes langues" (l'individu "DUPONT Marcel" ne change pas d'identifiant si on travaille en anglais…).

Champ(s) clé : le ou les attributs constituant l'identifiant "normal" de l'entité.

Ptr→index-R : un pointeur (facultatif) vers l'index répertoriant les associations de l'entité.

Ptr→données : un pointeur (facultatif) vers les attributs de l'entité.

Données : si l'entité a peu d'attributs, ils peuvent être stockés directement dans l'article entité au lieu d'être dans un article séparé, ce qui améliore les performances. Dans ce cas, **Ptr→données** est nul.

Navigation : On peut retrouver un article entité, soit à partir d'un (et un seul) article index-e, soit à partir d'un ou plusieurs articles alias et/ou relations et/ou données (ce dernier lien n'est pas évident a priori, il sera explicité plus loin).

Note : Les champs Niveau et Langue - effectivement implémentés dans les prototypes déjà mis en œuvre - ont été décrits ici pour montrer les possibilités d'un tel système. Par manque de place et leur non-fondamentalité dans les principes présentés, nous ne décrirons pas leur mise en œuvre dans cet article.

L'article INDEX-E

Nous n'allons pas détailler ici ce type d'article. Disons simplement que ces articles constituent une arborescence organisée pour optimiser la recherche d'entités, classées dans l'ordre de leurs identifiants.

Ajoutons que si une entité, un alias ou un index-e est adressé par un et un seul index-e, il existe un (et un seul) article index-e qui n'est pas adressé ainsi, mais dont l'adresse fait partie de la configuration : il s'agit de l'"index-maître", point d'entrée de toute la base.

L'article DONNÉES

Un article DONNÉES est l'article le plus standard. Il suffit de savoir que quand on a affaire à un tableau ou un agrégat, son contenu fait l'objet d'un article distinct, et que l'article le contenant possède à l'emplacement de cet ensemble un pointeur (DONNÉE contient DONNÉE). Il en est de même si l'attribut est un BLOB.

Dernier cas particulier, un objet ou une relation peut être vu comme un attribut d'un autre objet et/ou relation. Pour ce faire, il existe dans un article DONNÉES un champ qui est un pointeur vers l'entité ou relation correspondante (nous verrons plus loin comment retrouver l'attribut à partir de l'objet ou de la relation). Bien sûr, on pourrait gérer ceci au moyen d'une association "normale" entre entités et/ou relations, mais ce raccourci correspond au sens quasi-exclusif de la navigation ; donc, autant simplifier celle-ci.

Un ensemble d'attributs pouvant contenir plusieurs agrégats, tableaux, etc., il peut donc exister plusieurs pointeurs à partir d'un même article, d'où les cardinalités 0,N des liens vers DONNÉES, BLOB et ENTITÉ.

Sachant qu'un objet, une relation ou un agrégat peut contenir un grand nombre d'attributs, le pointeur DONNÉE chaîne DONNÉE permet de répartir les attributs sur plusieurs articles.

Navigation : On peut retrouver un article données à partir d'un (et un seul) article entité ou relation ou données.

L'article BLOB

Ce type d'article est très simple, puisque ne contenant que tout ou partie d'un seul attribut. L'ensemble de cet attribut est contenu, soit dans un seul article s'il est taille suffisamment petite, soit réparti dans plusieurs articles chaînés.

Navigation : Un BLOB contenant un attribut, on accède à celui-ci exclusivement à partir d'un (et un seul) article DONNÉES (ou BLOB en cas de chaînage pour un gros attribut).

L'article RELATION

Tout article RELATION a la structure suivante :

| Lg clé | TypeG | TypeD | Champ(s) clé | Ptr→entitéG | Ptr→entitéD | Ptr→index-R | Ptr→données | Données |

Lg clé : la taille de la clé de la relation. Cette clé est constituée du type (TypeG ou TypeD, selon le chemin d'arrivée) et d'un certain nombre de champs permettant de différencier une occurrence spécifique parmi d'autres et/ou de les trier selon une séquence définie.

TypeG : un code permettant d'identifier le type de la relation par rapport à la première entité (ou relation) reliée, c'est-à-dire la patte gauche.

TypeD : un code permettant d'identifier le type de la relation par rapport à la seconde entité (ou relation) reliée, c'est-à-dire la patte droite.

Champ(s) clé : le ou les attributs constituant la clé de la relation. Dans le cas où on a affaire à une relation qui permet de retrouver un attribut pointant une entité, cette clé contient, soit le numéro de l'attribut de l'entité concernée, soit l'ensemble des numéros permettant de localiser cet attribut en cas d'agrégat(s) et/ou tableau(x).

Ptr→entitéG : un pointeur vers la première entité (ou relation) reliée.

Ptr→entitéD : un pointeur vers la seconde entité (ou relation) reliée.

Ptr→index-R : un pointeur (facultatif) vers l'index répertoriant les associations de cette association.

Ptr→données : un pointeur (facultatif) vers les attributs de l'association.

Données : si l'association a peu d'attributs, ils peuvent être stockés directement dans l'article relation au lieu d'être dans un article séparé, ce qui améliore les performances. Dans ce cas, **Ptr→données** est nul.

Navigation : On peut retrouver un article relation, soit à partir d'un (et un seul) article index-r, soit à partir d'un ou plusieurs articles relation dans le cas d'associations d'associations.

L'article INDEX-R

Même usage que les articles index-e, sauf que l'index-maître est obligatoirement adressé par une (et une seule) entité ou relation.

Signalons simplement qu'à chaque pointeur vers une relation est associé un indicateur permettant de savoir par quelle patte on arrive (gauche ou droite).

Les articles ALIAS-E et ALIAS-R

Il s'agit de variantes des articles ENTITÉ et RELATION, qui ont pour particularité de n'avoir ni données ni pointeurs, à l'exception d'un pointeur sur l'entité ou la relation dont ils sont l'alias :

Navigation : On peut retrouver un article alias, soit à partir d'un (et un seul) article index-e ou index-r, soit à partir d'un (et un seul) article relation (dont le code type identifie un lien vers un alias) appartenant à l'entité pointée par l'alias.

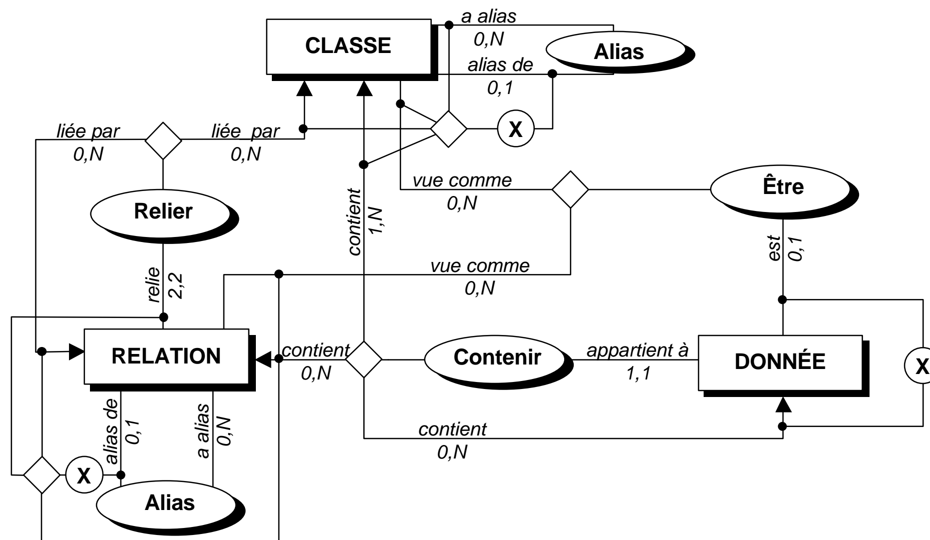
Synthèse

La gestion de ces différents articles est faite par un ensemble de fonctions qui permettent la consultation, navigation, ajout/insertion, modification, suppression de ces différents articles. Au-dessus de cette deuxième couche, on ne voit plus ni pointeurs ni articles en tant que tels, mais uniquement des objets, des attributs et des associations, cependant sous une forme encore très technique (pas de mnémoniques mais des codes binaires, données en format interne, etc.).

Nous disposons maintenant d'un substrat que nous allons commencer à alimenter avec des informations qui nous permettront ensuite d'y définir le métamodèle élaboré dans LA LETTRE n°27.

Du modèle physique au méta-2-modèle[5]

Nous allons maintenant définir dans le système décrit un modèle qui sera la conceptualisation de ce système, c'est-à-dire le MCD suivant :



La correspondance avec le modèle physique est simple : nous pouvons ignorer les index, qui sont ici "implicites", et considérer chaque article DONNÉES comme un regroupement physique d'objets DONNÉE, un BLOB étant lui-même une donnée ; un alias est une entité ou une relation n'ayant qu'une et une seule association ; la classe correspond à un champ de l'identifiant de chaque article entité.

Nous avons par ailleurs utilisé le mot CLASSE à la place d'ENTITÉ, car chaque individu CLASSE est une ENTITÉ-modèle. Vous allez comprendre tout de suite, car nous allons maintenant commencer à alimenter la base.

Toute la difficulté (et pas seulement à expliquer ou à comprendre) vient du fait qu'on dispose d'un contenant qu'on va remplir avec un contenu qui va se décrire lui-même[6], en plus de faire les correspondances entre le niveau physique et le méta-2-modèle. Nous n'allons donc pas entrer ici dans le détail (la simple liste des articles à créer pour définir ce métamodèle, en prenant en compte les notions de niveau et de langue, occuperait plusieurs dizaines de pages, à raison d'une ou deux lignes par article…), mais seulement montrer quelques exemples simples afin que vous vous rendiez compte que "ça peut marcher…"[7].

On utilisera une "classe" spéciale (le n°0) pour distinguer les entités et relations contenant les codes binaires des entités et relations contenant les mnémoniques.

Nous avons ci-dessus 3 entités et 9 relations. Nous allons donc créer les entités correspondantes (le caractère "•" sépare des données différentes) :

- Le premier groupe est | 1 | CLASSE | , | 1 | RELATION | et | 1 | DONNÉE |, avec respectivement comme alias : | 0 | 1•1 |, | 0 | 1•2 | et | 0 | 1•3 | (le premier chiffre est la zone **Classe** des articles Entité). Ceci signifie simplement que la classe n°1 s'appelle **Classe**, la classe n°2 s'appelle **Relation** et la classe n°3 s'appelle **Donnée**.
  (Pour éclaircir, l'individu **Classe Relation** décrit le modèle de la *classe* **Relation**, dont chaque individu décrit le modèle d'une *relation*…)

---

5 Ou méta-métamodèle, c'est-à-dire modèle du métamodèle.

6 C'est pourquoi, si le MCD ci-dessus représente des individus (cadre simple), on aurait pu tout aussi bien y représenter des modèles (cadre double), les deux coexistant (le modèle d'un objet est un individu d'une classe de modèles).

7 Et je puis vous certifier que ça marche vraiment !

• Le second groupe contient, entre autres, | 2 | Contenir |, qui a avec | 1 | CLASSE | la relation | 3 | 3 | contient•gauche | (ayant elle-même l'alias | 1 | 2 | 0•1•4•3 |), et avec | 3 | DONNÉE | la relation | 3 | 3 | appartient à•droite| (qui a l'alias | 1 | 1 | 0•3•1•1 |). Ce qui signifie que l'*entité* **Relation "Classe Contenir Donnée"** décrit la relation du même nom ; que la patte qui part de **Classe** s'appelle **contient** et a le type 4 (alias "0.1.4.3" = de classe 1, 4ème relation, vers classe 3), celle qui part de **Donnée** s'appelle **appartient à** et a le type 1. On a utilisé pour cela deux relations **"Classe Relier Relation"** qui sont de type 3 depuis Classe (**liée par**) et de type 3 depuis Relation (**relie**).
Pour mieux comprendre, avec des commandes mnémoniques on aurait (par exemple) :

```
Créer_entité Relation Nom="Contenir" ;
Créer_relation Classe "Classe" liée_à Classe "Relation" Nom="contient" Gauche=oui
    alias=(Nom=x'00' & x'01' & x'04' & x'03') ;
Créer_relation Classe "Donnée" liée_à Classe "Relation" Nom="appartient_à" Gauche=non
    alias=(Nom=x'00' & x'03' & x'01' & x'01') ;
```

Le même processus permettra de définir les données, leur format (qui est lui-même une donnée des entités de la classe Donnée…), leur appartenance, leur numéro relatif par rapport à cette appartenance... Dans un système opérationnel, d'autres classes de base sont bien sûr indispensables (langue, niveau, mots-clés, paramètres système…).

Bref, les fonctions qui gèrent ceci sont désormais capables de convertir le nom de classe **Relation** en code classe **2** (et vice-versa), savent que si on cherche les données appartenant à une entité ABC (on cherche la *relation* **Classe contient Donnée**) il faut partir de l'*entité* **Classe ABC** et lister les relations ayant le code gauche **2**, etc.[8]

À partir de cet ensemble, on utilisera désormais un ensemble de fonctions qui s'occuperont elles-mêmes d'allouer des numéros aux diverses entités, relations et données. Quand on demandera "Créer Classe Prédicat", que le système lui alloue le n°25 ou bien le n°36453 nous est complètement indifférent…

Implémentation du métamodèle

L'intérêt de ce système de métamodèles réflexifs, c'est que :

a) On définit une information de la même manière qu'on l'alimente (à la différence du système relationnel dans lequel création et modification des tables n'utilisent pas les mêmes commandes que leur mise à jour) ;
b) Toute information définie est immédiatement utilisable, comme vous pourrez le constater dans les exemples ci-après.

Avec le méta-2-modèle ci-dessous, qui nous permet désormais de travailler avec des mnémoniques, nous allons enrichir la base pour définir le métamodèle élaboré dans LA LETTRE n°27.

Pour vous simplifier la compréhension, des mots-clés seront utilisés avant d'être définis (nous les mettrons en italiques dans les commandes). D'autre part, l'implémentation de base des classes, données et relations étant "en dur", il est possible d'utiliser un langage de commande[9] plus puissant : on pourra ainsi dire :

```
Créer_entité  Relation  "Objet Identifier Attribut"  Gauche=(Nom="identifié par"  Cardinalités=(1,1))
    Droite=(Nom="identifie" Cardinalités=(0,)) ;
```

Ce qui créera l'entité | 2 | Identifier | et ses relations | 5 | 7 | identifié par•gauche | 1•1 | (avec **Classe "Objet"**) et | 5 | 7 | identifie•droite | 0 | (avec **Classe "Attribut"**).

---

[8] *Si on cherche par "Donnée appartient_à Classe", la recherche se ferait en partant de l'entité* Donnée XYZ *sur les relations ayant le code droit à 1. Vous suivez ?*

[9] *Ceux qui verraient une troublante ressemblance entre le langage utilisé ici et celui du DB/DC Data Dictionary d'IBM n'auraient pas tort : je m'en suis fortement inspiré - tout en l'enrichissant fortement.*

Pour commencer, les types **Objet**, **Attribut** et **Association** du métamodèle correspondent évidemment aux classes **Classe**, **Donnée** et **Relation**. Définissons-les :

```
Créer_alias Classe "Objet" alias_de Classe "Classe" ;
Créer_alias Classe "Attribut" alias_de Classe "Donnée" ;
Créer_alias Classe "Association" alias_de Classe "Relation" ;
```

Le type **Objet** a des attributs :

- *Unique* : unicité des identifiants (booléen)[10] ;
- *Alias* : peut avoir des alias (booléen) ;
- *Type* : type d'objet (singularité, occurrence, population) ;
- etc.

Nous allons les créer et les relier (avec des commandes élaborées[11]) :

```
Créer_entité Attribut "Objet"."Unique" ;
Créer_entité Attribut "Objet"."Alias" ;
Créer_entité Attribut "Objet"."Type" ;
```

… puis les renseigner :

```
Modifier_entité Classe "Objet" Unique=oui Alias=oui Type=population ;
Modifier_entité Classe "Attribut" Unique=non Alias=oui Type=population ;
Modifier_entité Classe "Association" Unique=oui Alias=oui Type=population ;
```

Un Objet a aussi des associations, dont certaines ont des attributs. Par exemple :

```
Créer_entité  Association  "Objet Hériter Objet"  Gauche=(Nom="hérite de"  Cardinalités=(0,N))
     Droite=(Nom="parent de" Cardinalités=(0,N)) ;
Créer_entité Attribut "Objet Hériter Objet"."Type héritage" ;
```

Pour pouvoir définir un attribut, l'objet correspondant a lui-même des attributs. Définissons les principaux :

```
Créer_entité Attribut "Attribut"."Type" ;
Créer_entité Attribut "Attribut"."Constant" ;
Créer_entité Attribut "Attribut"."Occurrences" ;
Créer_entité Attribut "Attribut"."Premier poste" ;
```

À partir d'ici, nous supposerons déjà définis - et renseignés - les types d'objets **TYPE** et **VALEURS** (sinon nous y sommes encore dans quinze pages…). Renseignons donc les attributs que nous avons déjà définis pour les entités **Attribut** existantes :

```
Modifier_entité Attribut "Objet"."Unique" Type=champ Constant=non ;
Créer_association Attribut "Objet"."Unique" défini_par Type "Booléen" ;
Modifier_entité Attribut "Objet"."Alias" Type=champ Constant=non ;
Créer_association Attribut "Objet"."Alias" défini_par Type "Booléen" ;
Modifier_entité Attribut "Objet"."Type" Type=champ Constant=non ;
Créer_association Attribut "Objet"."Type" défini_par Type "Type objet" ;
Modifier_entité Attribut "Objet hériter Objet"."Type héritage" Type=champ Constant=non ;
Créer_association Attribut "Objet hériter Objet"."Type héritage" défini_par Type "Type héritage" ;
Modifier_entité Attribut "Attribut"."Type" Type=champ Constant=non ;
Créer_association Attribut "Attribut"."Type" défini_par Type "Type attribut" ;
Modifier_entité Attribut "Attribut"."Constant" Type=champ Constant=non ;
Créer_association Attribut "Attribut"."Constant" défini_par Type "Booléen" ;
Modifier_entité Attribut "Attribut"."Occurrences" Type=tableau Constant=non Occurrences=2 Premier_poste=1 ;
Créer_association Attribut "Attribut"."Occurrences" défini_par Type "Entier" ;
Modifier_entité Attribut "Attribut"."Premier poste" Type=champ Constant=non ;
Créer_association Attribut "Attribut"."Premier poste" défini_par Type "Entier relatif" ;
```

---

[10] *Indispensable, notamment pour les identifiants relatifs.*

[11] *Qui créent l'entité* Attribut "xxxx" *et le relient à* Classe "yyyy" *avec la relation* "Classe contient Attribut".

On définit également d'autres types d'objets. Par exemple :

```
Créer_entité Classe "Action" Unique=oui Alias=oui Type=occurrence ;
Créer_entité Attribut "Action"."Nom" Type=champ Constant=oui ;
Créer_association Attribut "Action"."Nom" défini_par Type "Chaîne" ;
Créer_entité Attribut "Action"."Code" Type=champ Constant=non ;
Créer_association Attribut "Action"."Code" défini_par Type "Code source" ;
…
Créer_entité Action "Créer tâche" Code="…code source…" …
…
```

…et ainsi de suite.

Bref, le métamodèle élaboré dans LA LETTRE n°27 contenant 14 types d'objets, 25 types d'associations, plus de 30 types d'attributs, des prédicats… - et il est très incomplet -, on procède successivement à la définition d'objets, associations, attributs, de leurs associations internes, puis on enrichit les entités et associations - nouvelles et existantes - avec les nouveaux attributs et associations…

En résumé : on définit, on utilise pour définir et enrichir, etc.

Il reste là-dessus à programmer "en dur" les méthodes et règles de base qui permettront d'exploiter les données - sans oublier l'action qui se chargera d'exécuter les actions définies par les entités du même nom.

Quant aux objets et/ou attributs qui sont des capteurs ou des effecteurs (c'est-à-dire qui correspondent aux entrées-sorties), ils sont associés à des actions de type "pilote"[12] vers/depuis les périphériques connectés (par exemple, pour les accès en saisie et/ou affichage sur votre écran, les pilotes sont tout simplement les fonctions idoines de l'A.P.I.[13] de Windows - ou d'Unix).

Commentaires

Résumons-nous. Dans LA LETTRE n°26, nous avons montré qu'on pouvait modéliser n'importe quel type d'application avec un symbolisme unique. Dans LA LETTRE n°27, nous avons élaboré un métamodèle permettant de définir n'importe quel modèle d'application. Et nous avons maintenant élaboré un SGBD qui est capable d'exploiter directement ces modèles pour générer et utiliser des applications.

Obtenir une application opérationnelle directement en élaborant son modèle conceptuel, ça c'est du R.A.D[14] !

D'accord, d'accord, vous avez plein de questions et de critiques : comment définit-on un identifiant relatif, une contrainte d'intégrité, un prédicat ? Où sont la sécurité, le réseau ? Comment être portable ?[15] Je l'ai dit : les presque cinquante pages - réparties sur 3 LETTRES - que vous avez pu lire ne sont qu'un résumé très succinct, et beaucoup de choses y ont été omises. Et inutile de me contacter pour faire des remarques sur des points de détail, je vous ai présenté un principe dans ses grandes lignes.

En-dehors de la rapidité de mise en œuvre d'une application (y compris le temps réel, je le rappelle), voyons quelques avantages d'un système entité-relation.

---

[12] "Driver" pour les anglophiles.

[13] Application Programming Interface, ou routines d'interface entre l'application et le système d'exploitation.

[14] Rapid Application Development.

[15] Grâce à un langage idoine pour le code source des actions, ça a été dit dans LA LETTRE n°27.

La non-redondance

Ce qui permet à un SGBD entité-relation comme celui-ci d'être performant malgré sa complexité interne (sans rapport avec la simplicité - pour ne pas dire l'indigence - du modèle relationnel), c'est qu'on ne duplique **jamais** un identifiant : si Mlle DUPONT devient Mme DURAND, on change son nom dans son entité, et tout ce qui fait référence à cette personne la connaît immédiatement sous son nouveau nom, puisque tout se fait par des pointeurs. On a donc une excellente garantie d'intégrité du système.

La réflexivité du langage vs l'orientation objet

Si le langage utilisé dans cet article pour définir et renseigner la base de données n'est qu'un exemple, vous aurez pu constater qu'il est à la fois très simple et très riche, puisqu'il s'enrichit lui-même au fur et à mesure des définitions, permettant à des choses au départ explicites de devenir ensuite implicites.

Prenons un exemple : si on définit un type d'attribut Description et un type d'objet Mot Clé, on peut y associer une action qui scannerait toute description créée ou mise à jour pour relier l'entité contenant cette description aux entités Mot Clé correspondant aux termes trouvés. Par la suite, il suffit, lors de la définition d'une classe, d'y associer un attribut de type Description pour disposer automatiquement d'un thésaurus.

Autre exemple : on a définit un agrégat Adresse (contenant donc nom du site, n°, rue, code postal, localité…) et on a défini une action qui s'assure de la cohérence entre code postal et localité. Quelque soit par la suite le type d'objet utilisé, s'il a quelque part une Adresse, la saisie et la mise à jour de celle-ci sera contrôlée.

Bref, on dispose d'une réutilisabilité complète.

De plus - ce qui ne gâte rien - on est débarrassés une fois pour toute des programmes à recompiler dès qu'on altère un tant soit peu la structure d'une table relationnelle : toutes les actions utilisent automatiquement les attributs dans leur format réel (il suffit de gérer un contrôle d'intégrité pour éviter les incompatibilités lors de l'altération du format).

La gestion des requêtes

L'une des grandes forces des SGBD relationnels actuels est le langage SQL, qui permet d'effectuer des requêtes parfois très complexes. Or, que fait une requête SQL, sinon une navigation pour identifier les différents objets qui répondent aux critères de recherche ?

Cette force est en même temps une faiblesse, car certaines requêtes complexes sont consommatrices de temps machine, et sont à refaire entièrement à chaque fois.

Le système entité-relation que nous avons décrit permet de conserver cet avantage en en évitant les inconvénients. Il suffit pour cela de disposer d'une classe Requête, dont chaque occurrence sera reliée aux objets répondant aux critères de la recherche correspondante. Une requête contiendra, tant par attributs que par associations, prédicats et actions, le descriptif du traitement.

Quand une requête sera créée, elle sera exécutée. Par la suite, toute nouvel accès à cette requête donnera immédiatement le résultat de la recherche.

Mais s'il y a eu des changement dans la base ? demanderez-vous. Pas de problème, puisque les actions permettant de relier les objets à la requête seront activées lors d'une mise à jour d'un objet/association/attribut concerné et maintiendront automatiquement les liens à jour.

Une autre question vous brûle les lèvres : n'y a-t-il pas risque d'inflation à force de créer des requêtes de toutes sortes ? Effectivement. C'est pourquoi il faut également prévoir une action chargée de supprimer les requêtes qui n'auraient pas été utilisées depuis un certain temps.

Objectif à court et moyen terme

Résumé historique

Le système présenté au cours de ces trois articles a considérablement évolué au cours du temps :

- La version 1, élaborée entre 1977 et 1984, était conçue pour gros système IBM (Système d'exploitation MVS, langages assembleur IBM/370 et PL/1), et n'a existé - hors papier - que sous forme de sous-ensembles séparés, destinés à vérifier la faisabilité et les performances.
- La version 2, élaborée entre 1985 et 1991, était conçue pour PC (MS/DOS, langages turbo-assembleur 1 et turbo-pascal 5.5). Le prototype réalisé était capable de gérer des entités avec leurs index, ainsi qu'une partie des fonctionnalités données et relations.
- La version 3 (1992-1994), conçue multi-plates-formes (MS/DOS avec XMS, Windows, OS/2) a été - toujours partiellement - réalisée en turbo-assembleur 3 et Borland C++ 3.1.
- La version 4 (1995), conçue pour Windows 95, a intégré le multitâche et l'utilisation en réseau, mais n'a pas dépassé le stade du papier.
- La version 5, dont vous venez de lire un bref résumé du concept, sera conçue pour Windows 95 et NT, et le prototype réalisé en Delphi 3.

Le projet

Une commission «***Logiciel d'évaluation des AGL***» a été créée au sein de l'ADELI fin 1996. Le prototype de cette application sera réalisé avec une implantation physique proche de la description ci-dessus - mais très simplifiée, certaines fonctions étant (temporairement) codées "en dur".

Le SGBD sera implémenté, non pas de toutes pièces, mais sur un substrat relationnel (20 champs, 4 tables et 5 index suffisent). Évidemment, ceci se fera au détriment des performances (accès aux articles par clé interne et index au lieu de pointeurs directs), mais accélérera considérablement la réalisation.

Le MCD du logiciel d'évaluation étant lui-même très simple, il pourra très facilement être implémenté sur ce support. À l'heure où vous lisez ces lignes, les travaux ont déjà commencé.

Par la suite, les versions suivantes de l'applicatif (et d'autres produits à faire) profiteront d'un niveau d'interfaçage toujours plus important, jusqu'à parvenir à une implémentation complètement paramétrée.

Perspectives pour demain…

Le SGBD que nous avons conçu tout au long de ces trois articles est basé sur les sept systèmes cités au début, et permet donc de les implémenter tous[16]. Mais bien d'autres possibilités sont envisageables.

Traitement de texte

Tout document issu d'un traitement de texte est structuré de manière arborescente : sections et/ou chapitres, cadres et tableaux, paragraphes, texte, chacun ayant ses propres attributs :

- pour les chapitres et sections : format de page, en-tête et pied de page…
- pour les tableaux : colonnage, en-tête (et parfois en-pied), bordures…
- pour les cadres : dimensions, positionnement, chaînage…
- pour les paragraphes : style, espacement, alignement, puce ou numérotation…
- pour le texte : police, taille, graisse, décalage, couleur…

…avec la possibilité d'insérer d'autres objets : images, etc.

Il serait donc très simple de définir un type d'objet DOCUMENT, avec une arborescence d'attributs (agrégats et tableaux) qui contiendraient les différentes caractéristiques de chaque type de composante,

[16] *Y compris toutes les formes de SGBD "classiques" (hiérarchique, relationnel, codasyl…). Si, si, essayez…*

associées au contenu (les attributs pointant directement sur d'autres objets permettant l'incorporation d'autres objets par liaison[17]). Un document pourrait donc contenir d'autres documents de tous types.

Hypertexte

Un attribut qui pointe sur un autre objet, ou une information (texte, image…) à laquelle on associe un lien (signet, URL ou autre), et voilà, on surfe dans le SGBD. Fini, les formats spécifiques de fichiers d'aide, HTML et autres…

Graphisme

Tout comme un document issu d'un texteur, une image vectorielle (qu'elle soit 2D ou 3D) se compose de plans, dans lesquels se superposent des objets et groupes d'objets, constitués de lignes, courbes, surfaces, texte…

Une image bitmap constitue également un fichier structuré, surtout quand elle est compactée.

Alors, qui me proposera une structure d'objet "graphisme" renvoyant aux oubliettes les formats BMP, GIF, JPG, PNG, CDR, DXF, CGM, WMF, etc, etc, etc. ?

Multimédia

Qu'est-ce qu'un document multimédia (diapositives, animation…) sinon un ensemble d'objets associés dans un sur-ensemble structuré ? Alors, en avant pour remplacer les formats AVI, MOV, GIF animé, MPEG, PPT et consorts…

Mémoire de masse

Qu'il s'agisse d'Unix, Ms-Dos, Windows NT ou beaucoup d'autres, les données manipulées par un système d'exploitation sont stockées sous forme de fichiers, regroupés dans des répertoires… arborescents. Ici aussi, notre SGBD pourrait, plutôt qu'être implémenté comme un fichier ou un ensemble de fichiers, directement remplacer systèmes FAT, NTFS…[18]

…et après

Finalement, au vu de ce qui précède, au lieu d'énumérer ce qu'un SGBD comme celui que nous avons décrit peut faire, il serait peut-être plus simple de chercher ce qu'il ne peut pas faire (le café ? Pourquoi pas, avec les interfaces domotiques et robotiques idoines connectés à des capteurs et effecteurs…).

À long terme, il est certain que se standardisera au niveau mondial un système intégré dans lequel chaque utlisateur aura son espace - sans distinction pour lui entre mémoire vive et mémoire de masse - qu'il pourra consulter et utiliser comme une banque de données et/ou un "Datawarehouse", dans lequel il pourra naviguer (dans la limite des autorisations des différents domaines), piocher, etc.

Peut-être en ai-je décrit les premières bases, au cours de toutes ces pages ?

Et si un "grand" du logiciel décide de créer ou de faire évoluer son produit en s'inspirant des idées publiées ici, S.V.P., contactez-moi plutôt que de vous servir directement : ce serait plus sympa… et je n'ai pas tout écrit ici, loin s'en faut ! ▲

© 1997 EPHITEQ

Jean-Luc Blary

---

[17] *Comme tout objet est géré par les actions qui lui sont associées, indépendamment de toute liaison externe, le petit parfum d'OLE que nous avons là rend ce dernier primitif et obsolète (sorry, Bill).*

[18] *Par exemple, les fichiers "basiques" peuvent être manipulés comme un seul objet (1 enregistrement = 1 attribut), avec l'avantage de pouvoir accepter des insertions/modifications/suppressions d'enregistrements intermédiaires sans réécriture de tout le fichier…*

Sources documentaires

MERISE, support de cours
ABOUHAIR G.
IBSI, Paris 1988, 1991, 1992

EPHIBASE, SGBD entité-relation orienté objet, dossiers de conception et prototype
BLARY J-L.
EPHITEQ, Caëstre 1989-1990, 1993, 1995, 1996

ODESYS, Outil d'aide à l'évolution du Système d'Information, dossiers de conception
BLARY J-L., THELLIEZ Ph.
EPHITEQ, Caëstre 1993-1996

MERISE & OBJETS, support de cours
BLARY J-L.
EPHITEQ, Caëstre 1995

MERISE, support de cours
BLARY J-L.
EPHITEQ, Caëstre 1996

Le Réseau Sémantique Universel (1ère partie)
BLARY J-L.
La Lettre de l'ADELI n°26, janvier 1997

Le Réseau Sémantique Universel (2ème partie)
BLARY J-L.
La Lettre de l'ADELI n°27, avril 1997

De la modélisation systémique au réseau sémantique
BRES P-A., ROCHFELD A., TABOURIER Y., SIBERTIN-BLANC C.
Conférence-débat de l'AFCET, Paris 15/11/1990

SGBD avancés : bases de données objet, déductives, réparties
GARDARIN G. VALDURIEZ P.
Eyrolles, Paris 1990

Merise vers une modélisation orientée objet
MOREJON J.
Les Éditions d'Organisation, Paris 1994

Réseaux de neurones
NADAL J-P.
Armand Colin, Paris 1993

Le langage C++
STROUSTRUP B.
Addison-Wesley, Paris 1992

Moteurs de systèmes experts
VOYER R.
Eyrolles, Paris 1987

☛*Intéressés, critiques, puristes, adversaires, partisans... pour en savoir plus ou participer :*

☎ *0.660.602.702*
0.328.402.702
jlblary@nordnet.fr
✉ *EPHITEQ - Château Vallée - 59190 CAËSTRE*