**MASTER EN SCIENCES INFORMATIQUES**

# Vers une méthode de conception orientée objet applicable aux développements xBase

Delacharlerie, André

*Award date:*
1993

*Awarding institution:*
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

# Vers une méthode de conception orientée objet applicable aux développements xBase

André Delacharlerie

Mémoire de Licence et Maîtrise en Informatique
Promoteur : N. Habra

Année académique 1992-93

# Sommaire

## Volume 1 : Mémoire

# Volume 2 : Annexes

Les documents donnés en annexes C à E ont été générés directement par le logiciel SACOO d'aide à la conception orientée objet et concernent le projet "Voyage scolaire".

Annexe A : Disquette contenant le logiciel SACOO et le repository ROME correspondant au projet "Voyage scolaire"

Annexe B : Procédure d'installation du logiciel SACOO

Annexe C : Schéma complet des classes du projet.

Annexe D : Documentation textuelle complète du projet.

Annexe E : Listing Clipper+Class(y) de la classe `Participant`.

# 1

# Introduction

Effet de mode ou progrès réel, il n'est plus guère aujourd'hui de langages de programmation, de systèmes d'exploitation ou de gestionnaires de bases de données qui ne se revendiquent d'être - au moins partiellement - "orientés objet". Il est donc légitime de s'interroger sur les raisons qui ont conduit à l'émergence de ce nouveau paradigme.

Historiquement, c'est la programmation impérative qui s'est manifestée comme le paradigme de base pour l'expression des traitements informatiques. En effet, l'ordinateur est par nature un exécutant et il faut donc lui donner des ordres pour qu'il effectue les tâches que nous lui soumettons. On est ainsi naturellement amené à rédiger des "marches à suivre" constituées principalement par une série d'instructions impératives et organisée par quelques structures de contrôle pour exprimer "comment" mener à bien les tâches considérées. De très nombreux langages, tels bien évidement l'assembleur mais aussi Cobol, Fortran, Pascal, C, ... relèvent de ce paradigme et ont servi à écrire la plus grande partie des logiciels aujourd'hui disponibles.

Produire un algorithme "impératif", qui décrive en détail tous les aspects d'une tâche de quelque importance n'est cependant pas chose aisée et les informaticiens ont cherché à réduire cette difficulté en imaginant d'autres langages bâtis sur des paradigmes nouveaux. La programmation fonctionnelle et la programmation logique permettent ainsi (selon des approches bien différentes) de remplacer l'expression d'une "marche à suivre" par une description plus déclarative de ce qui doit être fait mais sans détailler (trop) explicitement "comment" l'ordinateur doit effectuer la tâche.

Avec le paradigme objet, c'est une autre direction qui est explorée. Elle repose sur un déplacement du centre d'intérêt de l'informaticien de la description de la tâche elle-même vers le "paquet" informationnel qui en est ...l'objet. On observe en effet aisément, que chaque objet informationnel détermine, par sa définition, les caractéristiques qui permettent de le décrire et les traitements qui peuvent lui

être appliqués. Chaque objet peut ainsi être étudié indépendamment du système dans lequel il fonctionnera. En conséquence, un système informatique peut être plus facilement construit comme un assemblage de divers objets qui contribuent chacun selon leurs possibilités à l'objectif poursuivi.

Devant cette inexorable percée du "paradigme objet" aucun informaticien ne peut rester indifférent. Au contraire, il est indispensable de s'interroger sur les apports réels de cette nouvelle approche et d'élucider, le cas échéant, les conditions dans lesquelles il est, dès aujourd'hui, possible d'en tirer profit.

Aussi, c'est avec le double regard de l'étudiant de maîtrise et du formateur d'adultes (en particulier d'enseignants) que je suis, que je me propose d'étudier quels bénéfices peuvent être apportés par le paradigme objet lors de la conception et de l'implémentation de systèmes de gestions de données sur micro-ordinateurs (particulièrement mis en oeuvre avec des outils de type Xbase). Je voudrais également tenter, dans ce travail, de mettre en évidence quelques règles méthodologiques susceptibles d'aider des non-spécialistes de l'informatique ou des informaticiens débutants à concevoir des logiciels de gestion d'informations dans une optique orientée objet.

# 2

# Objectifs et contraintes

## 2.1 Paradigme objet

Le premier objectif qui nous paraît devoir être poursuivi consiste à circonscrire le paradigme objet en en présentant les concepts fondamentaux, les apports et les limites. Ce sera aussi l'occasion de fixer le sens que nous attacherons dorénavant aux nombreux mots du vocabulaire "orienté objet". En effet, si de nombreux colloques et ouvrages récents contiennent le terme "objet" dans leur titre, il y a lieu de se demander si ce terme désigne bien le même concept. En particulier, il serait intéressant de rechercher quel sens attacher à des termes comme classe, héritage, encapsulation, polymorphisme, généricité... Nous pourrons d'ailleurs observer que si beaucoup de langages ou de méthodes de conception revendiquent d'être "orientés objet", ils n'intègrent généralement qu'un sous-ensemble plus ou moins grand des concepts constituant le paradigme objet.

Nous consacrerons donc un premier chapitre de ce travail à une présentation aussi didactique que possible de ces concepts. Nous profiterons aussi de ce chapitre pour introduire quelques conventions de notations qui nous aiderons à illustrer nos propos et que l'on retrouvera plus loin dans la partie méthodologique du travail. A l'occasion, nous montrerons aussi quels sont les liens qui existent entre le modèle objet et d'autres modèles plus classiques (Entité - Association, Relationnel, ...)

## 2.2 Conception orientée objet

Pour l'informaticien, les apports les plus évidents du paradigme objet se trouvent concrétisés par l'introduction de nouveaux langages de programmation (Simula, Smalltalk, Eiffel...) ou par l'adjonction à des langages "classiques" de

fonctionnalités nouvelles permettant de créer et d'utiliser des objets (C++, Turbo Pascal orienté Objet, Visual Basic ou encore le couple Clipper 5.0 accompagné de la librairie Class(y) qui nous intéressera plus particulièrement dans la suite). Le défi qu'il appartient aujourd'hui à la communauté informatique de relever ne se situe pas seulement dans la mise en oeuvre de ces langages lors de la phase d'implémentation des logiciels. Il est bien plus important car il vise à intégrer la mentalité orientée objet dans chacune des phases de la conception et de la réalisation des logiciels avec l'objectif de réduire l'investissement de développement et les coûts de maintenance.

Nous tenterons de voir ainsi si le paradigme objet est effectivement porteur d'atouts lors de la conception et nous chercherons à mettre en évidence ses limites et ses écueils éventuels. Tout naturellement nous serons amenés à présenter et à comparer quelques méthodes de conception se revendiquant de l'approche objet (O*, OBLOG, HOOD...).

## 2.3 Propositions méthodologiques

A la lueur des enseignements que l'on aura pu dégager de l'exploration de quelques méthodes de conception orientée objet, nous voudrions retenir et expliciter quelques propositions méthodologiques pour la conception de logiciels de gestion de système d'information. Ces propositions devraient pouvoir être mises en oeuvre par une large catégorie de développeurs incluant à la fois des non-professionnels de l'informatique mais aussi les nombreux concepteurs, familiers de l'analyse fonctionnelle et de la programmation impérative, qui voudraient tirer profit de l'approche objet.

Pourquoi vouloir ainsi, vulgariser les méthodes de conception et courir le risque de voir les inévitables "simplifications" nous faire perdre certains avantages ? La réponse à cette question trouve ses justifications dans mon expérience de formateur au CeFIS (Centre de Formation à l'Informatique pour le Secondaire). En effet, depuis plus de dix ans, le centre apporte une formation à la programmation à de nombreux enseignants qui se préparent à enseigner eux-mêmes l'informatique et/ou à utiliser des outils informatiques dans le cadre de leur enseignement. Or, il est d'emblée apparu qu'une formation sérieuse de ce public ne saurait se limiter à faire apprendre l'utilisation de langages tels que BASIC ou PASCAL ou de systèmes de gestion de données comme dBASE. Il faut également offrir aux enseignants/apprenants des outils intellectuels leur permettant de concevoir et organiser le programme qu'ils projettent de rédiger.

C'est dans cette optique que l'on doit donc situer les propositions méthodologiques que nous voudrions mettre en évidence. S'adressant à des développeurs individuels (à la rigueur à des petits groupes) qui de surcroît n'ont pas toujours derrière eux une lourde formation à l'informatique, ces propositions se doivent donc d'aller à l'essentiel. En un mot, nous désirons rester simple sans être simpliste, mettre en évidence les étapes fondamentales d'une méthode sans pour autant la trahir...

Notre recherche sera également contrainte par la nature des outils de développement qui sont utilisés par le public visé. En effet, on sait qu'une importante majorité des "petits" développeurs ne s'intéressent qu'au matériel de type PC-Compatible et par conséquent aux langages disponibles sur ce type d'ordinateur. En particulier, dans le domaine de la gestion de données, Nantucket Clipper est un langage en plein essor aujourd'hui. Il est d'ailleurs l'héritier de dBASE qui fut lui aussi très populaire il y a quelques années. Or, précisément la version actuelle (5.0) de Clipper introduit timidement des classes d'objets et Computer Associates, qui commercialise le compilateur, annonce des versions ultérieures résolument orientées objet. Enfin, différentes firmes commercialisent aujourd'hui des librairies qui permettent de créer et de gérer des classes d'objets dans la version 5 de Clipper.

C'est donc une occasion à saisir que de réfléchir, dès aujourd'hui, aux méthodes de conception qui seront les mieux adaptées au développement de logiciels dans ce genre d'environnement.

## 2.4 Outil d'aide à la conception

La plupart des auteurs qui présentent une méthode de conception orientée objet proposent également un environnement logiciel qui en permet la mise en oeuvre concrète que ce soit par le biais d'un langage [MEYER] ou d'outil de support [LAI], [SERNADAS].

Un second objectif de ce travail sera donc de développer, au moins sous forme de prototype, un environnement, aussi convivial que possible, qui facilite la mise en oeuvre des propositions méthodologiques élaborées. Par ailleurs, le développement de cet outil sera lui-même un laboratoire de réflexion et d'expérimentation tant sur la méthodologie de conception que sur la mise en oeuvre concrète de la programmation orientée objet dans le cadre du langage Clipper (version 5.01) complété par la bibliothèque Class(y).

La figure 2.1 illustre schématiquement, la démarche adoptée dans le mémoire.

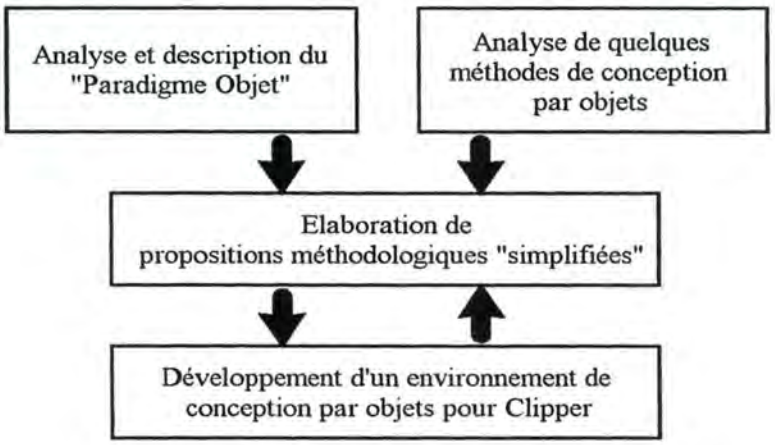



Figure 2.1 : Démarche adoptée dans le mémoire.

# 3

# Le paradigme objet

Dans ce chapitre, nous allons décrire les concepts qui nous apparaissent fondamentaux du paradigme objets[1]. Ce sera l'occasion de nous entendre sur le vocabulaire et sur une série de conventions, de représentations que nous emploierons désormais.

## 3.1 Le concept d'objet

Le concept d'*objet* (au sens informatique) est, tout compte fait, l'un des plus simples et des plus naturels qui soit. Il permet d'isoler et de décrire n'importe quel objet du monde réel (au sens commun), qu'il soit concret (par exemple, une facture, un produit, un véhicule, un employé, ...) ou abstrait[2] (par exemple, un cours, une organisation, une oeuvre littéraire, ...). L'opération fondamentale qui consiste ainsi à représenter informatiquement une "chose" du monde réel est appelée la "réification" [FERBER,35], [AUBERT,2].

Dès que l'on introduit le concept d'objet, le monde peut être représenté comme un système composé d'objets qui entretiennent des "relations" entre-eux et qui peuvent interagir les uns sur les autres. C'est pour cette raison que l'on parlera parfois de "population" d'objets. Un des avantages immédiats de la modélisation par objet réside dans la possibilité de décrire, dans le même modèle, des objets

---

[1] Sans épiloguer sur le sens du mot "paradigme" signalons simplement que nous avons préféré l'expression "paradigme objet" plutôt que "langage (orienté) (à) objet" ou encore "modèle (orienté) objet" pour éviter de mettre inutilement l'accent sur l'aspect langage de programmation ou modèle de représentation et support de conception car le paradigme objet s'intéresse aux deux aspects.

[2] La distinction entre objets concrets et abstraits est ici sans importance pratique. Ce qui compte, c'est qu'il s'agit d'objets auxquels sont attachées des informations que l'on juge utiles dans le contexte considéré.

du système informatique en développement ainsi que des objets de l'environnement réel en relations avec les premiers.

Comme on va le voir, la richesse du concept d'objet est due au fait qu'il intègre à la fois une représentation des *qualités statiques* d'un objet du monde réel mais aussi ses *qualités dynamiques.*

### 3.1.1 Qualités statiques : état de l'objet

A l'instar du concept d'entité dans le modèle Entité-Association, un objet est caractérisé par une série d'informations qui décrivent son *état.* Un objet est d'abord une structure de données qui pourra être souvent relativement complexe en faisant intervenir à la fois des valeurs simples (le titre d'un ouvrage) ou multiples (la liste des numéros de téléphone d'une institution) mais aussi des liens qui pointent vers d'autres objets (l'emprunteur d'un ouvrage).

### 3.1.2 Qualités dynamiques : comportement de l'objet

La grande originalité du paradigme objet consiste à associer à la description statique, une description des *comportements* qui peuvent être attendus de l'objet. On définit ainsi un objet par ce qu'il peut faire[3] en interaction avec les autres objets du système informatique. Par exemple, un objet facture sera décrit en y associant les comportements qui lui permettent de s'imprimer, de s'annuler, de s'acquitter, ...

Un objet est ainsi une entité dynamique capable de modifier son propre état (c'est-à-dire les informations qui le décrivent) et d'effectuer certaines actions (afficher telle information ou calculer telle information dérivée de son état par exemple).

### 3.1.3 Dialogue entre les objets : les messages

Un système informatique bâti avec des objets peut être considéré comme une population d'objets qui interagissent entre eux. Le mécanisme d'interaction est basé sur l'envoi de *messages.* Si par exemple, un objet `Client` désire faire acquitter une certaine facture, il enverra à l'objet `Facture` le message `"Acquitte-toi"`. L'objet `Facture` concerné examinera d'abord son état pour vérifier par exemple, qu'il n'est pas déjà acquitté ou annulé et, si rien ne le lui interdit, il modifiera son état comme demandé.

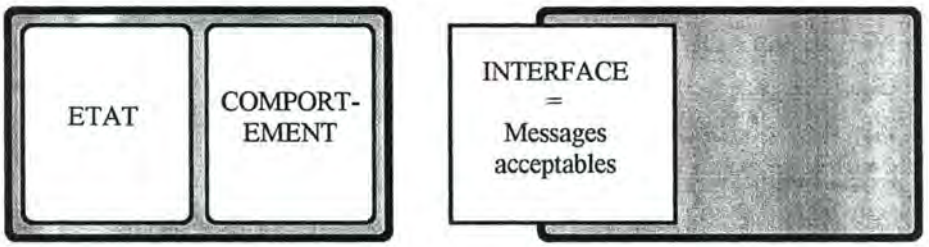
On met ici le doigt sur l'un des principes les plus fondamentaux de paradigme objet : chaque objet doit être le seul à connaître son état et à pouvoir le modifier. Ainsi, lorsqu'un objet `X` désire modifier l'état d'un objet `Y`, il ne peut le faire seul, mais doit le demander en envoyant à `Y` un message approprié. Ce principe

---

[3] L'anthropomorphisme qui consiste comme le rappelle Larousse à "attribuer aux animaux et aux choses des réactions humaines" n'est pas du tout innocent dans le contexte du paradigme objet. C'est peut-être précisément parce que l'on peut voir un programme comme une population d'«êtres» capables d'action, voire d'initiative, que le paradigme objet peut paraître si "naturel" au premier abord.

s'appelle *l'encapsulation* et consiste à voir l'objet selon deux angles *privé* et *public* (figure 3.1).

Sous l'angle privé - c'est-à-dire interne - un objet connaît la description de son état actuel ainsi que les comportements qu'il peut avoir avec les algorithmes à mettre en oeuvre pour produire ces comportements. Il s'agit en fait, de *l'implémentation* de l'objet. Sous l'angle publique - ou externe -, un objet est essentiellement caractérisé par son *interface,* c'est-à-dire par les messages qu'il est censé comprendre et auxquels il peut réagir par une action (activation d'une opération interne) ou en renvoyant une information déduite de son état. L'interface d'un objet n'est donc autre chose que le reflet de sa *spécification.* puisqu'il exprime ce que l'on peut attendre de cet objet.




Vue privée d'un objet

Vue publique d'un objet

Figure 3.1 : Les deux faces d'un objet

### 3.1.4 Apports fondamentaux du concept d'objet

Il est intéressant de souligner dès à présent les divers bénéfices associés à la structuration d'un système informatique en une population d'objets.

- ***Maîtrise de la complexité***

  Le premier avantage semble être de disposer d'un moyen de maîtriser la complexité des systèmes informatiques qu'il faut aujourd'hui concevoir. Partant du principe qu'il convient de "diviser pour régner" et qui aurait déjà été mis en lumière il y a bien longtemps (entre autre par Machiavel ou Descartes), le paradigme objet permet de modulariser un système informatique en entités autonomes qui peuvent être étudiées et développées de façon (presque) indépendantes les unes des autres.

  Cette autonomie est beaucoup plus grande que celle que l'on peut avoir avec la programmation impérative classique car l'objet est à la fois maître de son état ET de son comportement : seules les procédures internes de l'objet ont le droit de manipuler les informations contenues dans l'état de cet objet[4]. Il ne peut donc y avoir aucun effet de bord, ni application abusive d'une procédure à des données étrangères. Par ailleurs, une modification

[4] En pratique, toutefois, de nombreux langages admettent des exceptions à ce principe en laissant plus ou moins de possibilités d'accéder, de l'extérieur, à la description interne d'un objet.

dans la structuration des données n'a pas de répercussion dans l'ensemble du logiciel mais seulement dans l'objet qui en est le dépositaire [MASINI, 18].

De plus, le concept de classe, qui sera décrit au paragraphe suivant, et qui généralise la notion d'objet, facilite sensiblement le développement et particulièrement la maintenance des applications. D'une part en effet, un comportement ne doit être décrit qu'à un seul endroit, appliquant ainsi le principe de non-redondance - bien connu dans le domaine des bases de données - au domaine de l'algorithmique[5]. D'autre part, en cas de problème, il est relativement facile d'identifier l'objet qui ne produit pas le comportement attendu et par conséquent, d'isoler le code qui doit être corrigé.

- ***Réutilisabilité***

  Tout programmeur sait bien qu'un nouveau logiciel à réaliser n'est jamais entièrement "nouveau". Bien des tâches à implanter ont déjà été rencontrées dans d'autres programmes. Pourtant, on sait aussi combien qu'il est malaisé d'extraire une portion de code d'un programme et de la plaquer directement dans un autre. En effet, le contexte d'un programme n'est pas celui d'un autre et la transposition d'une portion de code nécessite alors des modifications parfois mineures mais le plus souvent importantes.

  Ici aussi, le paradigme objet se révèle très intéressant car l'encapsulation du code dans les objets rend ceux-ci indifférents au contexte du programme. Il suffit de faire appel aux services des objets via les messages publiés dans leur interface, leur fonctionnement interne est indépendant du reste du programme.

  Certes, cette idée est déjà largement exploitée dans la programmation impérative classique avec la modularisation du code en procédures et la constitution de librairies de procédures ayant trait à un "domaine" particulier mais ici, dans le paradigme objet, le principe d'encapsulation est appliqué systématiquement et associe intimement les données avec les actions qui leur sont relatives.

  Corollaires importants de cette réutilisabilité, le développement d'un logiciel peut être à la fois plus efficace (on ne développe que ce qui est vraiment nouveau) et plus fiable (les modules réutilisés ont déjà été validés antérieurement et leur robustesse est donc garantie).

- ***Lisibilité***

  En plus de la maîtrise de la complexité, la structuration d'un logiciel en objets offre une meilleure lisibilité car les détails d'implémentations sont cachés tandis que l'interface de l'objet fournit indirectement un mode

---

[5] Le mécanisme de l'héritage qui sera présenté plus loin, contribue aussi largement à réaliser cette non-redondance au niveau algorithmique.

d'emploi précis et détaillé de ses possibilités [MASINI, 20][6]. Le code des objets est lui-même plus lisible car il ne fait appel qu'à des structures de données locales et à des services demandés à d'autres objets.

- ***Adaptabilité***

  Il est très facile de modifier un logiciel pour l'adapter à un nouvel environnement (matériel différent, contraintes changées, ...) car il suffit de corriger ou de remplacer seulement les objets concernés par les modifications sans toucher au reste du logiciel.

Dans les réalisations concrètes, la portée de certains des avantages énumérés ci-dessus peut bien entendu être réduite par les contraintes spécifiques d'un environnement de développement mais aussi, par la plus ou moins grande capacité du développeur à structurer son logiciel dans l'esprit du paradigme objet. Il importera donc, de lui fournir le maximum d'assistance lors de la conception de son projet.

## 3.2 Le concept de classe

De nombreuses situations conduisent à devoir gérer simultanément plusieurs objets ayant des caractéristiques semblables (par exemple toutes les fenêtres d'une interface graphique). Il est alors naturel de rassembler ces objets "semblables" afin, par exemple, de ne décrire qu'une seule fois les comportements qu'ils partagent.

Une *classe* est donc une entité plus générale qui regroupe des objets ayant une structure interne semblable et capables d'avoir les mêmes comportements[7]. Par analogie au modèle Entité-Association, on peut dire qu'une classe est à un objet comme un type d'entité est à une entité. Plusieurs auteurs présentent la classe comme "une sorte de moule à partir duquel sont générés les objets que l'on appelle les *instances* de la classe" [FERBER, 15].

Le mécanisme d'*abstraction* est à la base du concept de classe. En effet, c'est grâce à l'abstraction que nous pouvons construire une classe à partir de plusieurs objets du monde réel en ne retenant que les seuls caractères communs considérés comme essentiels par rapport à l'objectif que l'on poursuit [AUBERT, 24].

Comme le fait remarquer Michel Lai, le concept de classe au sens du paradigme objet est plus large que le concept de classe en mathématique (ensemble muni d'une relation d'équivalence). "En fait, l'ensemble des objets définis à partir d'une classe C forme une classe d'équivalence C' pour la relation suivante : Oi est construit sur le même modèle qu'Oj" [LAI,197].

---

6 Cet avantage sera toutefois réduit lorsque l'objet fera partie d'une hiérarchie d'héritage.

7 Certains langages tels que Ada ne fournissent toutefois pas le concept de classe et obligent donc l'utilisateur à considérer séparément chaque objet.

C'est donc la classe qui est dépositaire de la structure des informations déterminant l'état de chaque objet, de la description des messages compréhensibles par les objets instances ainsi que des algorithmes décrivant les comportements associés. La figure 3.2 montre quels sont les éléments qui sont propres à chaque instance Ob.1...Ob.n et ceux qui sont partagés au sein de la classe C.

Notons encore que le concept de la classe n'est pas équivalent à celui du type tel qu'il se rencontre dans les langages de programmation [DELOBEL, 261]. En effet, une classe peut être vue comme la collection des objets qui sont ses instances et permettre ainsi des manipulations globales des instances alors que cette possibilité n'apparaît jamais pour un type de donnée.

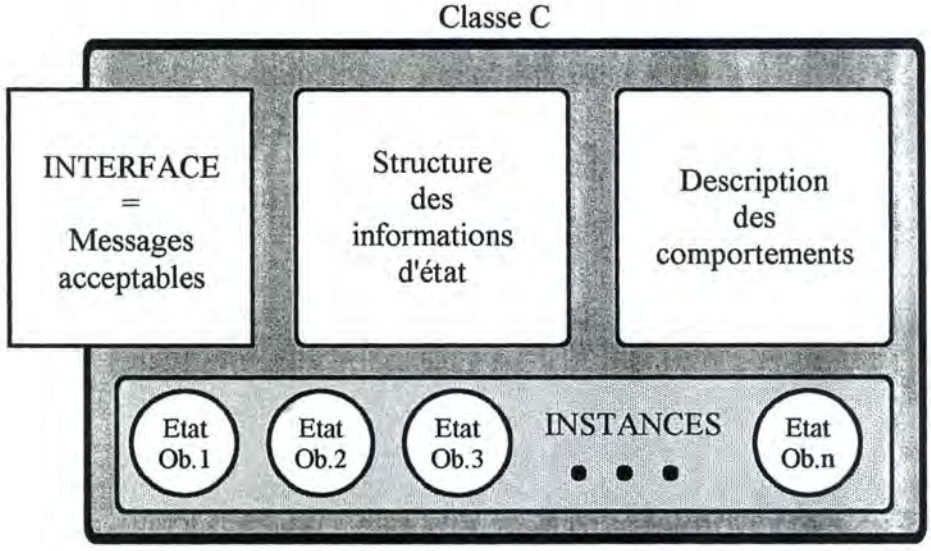



Figure 3.2 : Composants d'une classe

Pour fixer les idées, nous allons illustrer les concepts de classe et d'instance ainsi que ceux qui vont être présentés dans la suite en faisant référence à l'exemple d'un logiciel qui devrait gérer (de façon très limitée !) une bibliothèque.

La première démarche à entreprendre consiste à isoler les classes d'objets qui vont intervenir[8]. Il est naturel de distinguer d'une part les "oeuvres littéraires" que nous appellerons "documents" et d'autre part les exemplaires de ces documents qui sont physiquement présents dans la bibliothèque. On considérera donc séparément la classe `Document`, dont chaque instance correspond à une oeuvre (abstraite) bien particulière et la classe `Exemplaire` dont chaque instance représente un "livre" (concret) de la bibliothèque. Bien entendu, chaque instance de `Document` pourra être associée à zéro, un ou plusieurs instances d'`Exemplaire` représentant ainsi le fait que la bibliothèque possède zéro, un ou plusieurs exemplaires d'un document.

[8] Notre démarche est, à ce point du travail, essentiellement inspirée de la méthode ce conception que nous adoptons habituellement pour créer un schéma Entité-Association.

De façon analogue, il est intéressant de considérer chaque personne ayant collaboré à la rédaction d'un document comme instance d'une classe `Auteur` et par conséquent d'établir un lien entre chaque instance de `Document` et la ou les instance(s) d'`Auteur` qui lui correspond(ent). Par le même raisonnement, on crée une classe `Editeur`.

Par contre, il est raisonnable de considérer que le titre d'un document est intimement lié au document lui-même et que si plusieurs documents partagent (exactement) le même titre, c'est généralement fortuit. Il n'y a donc pas lieu de définir une classe pour les titres mais plutôt de considérer le titre comme une valeur attachée à un document[9] et permettant de le décrire. De même, on peut décider de considérer l'année de publication d'un document comme une autre valeur attachée au document.

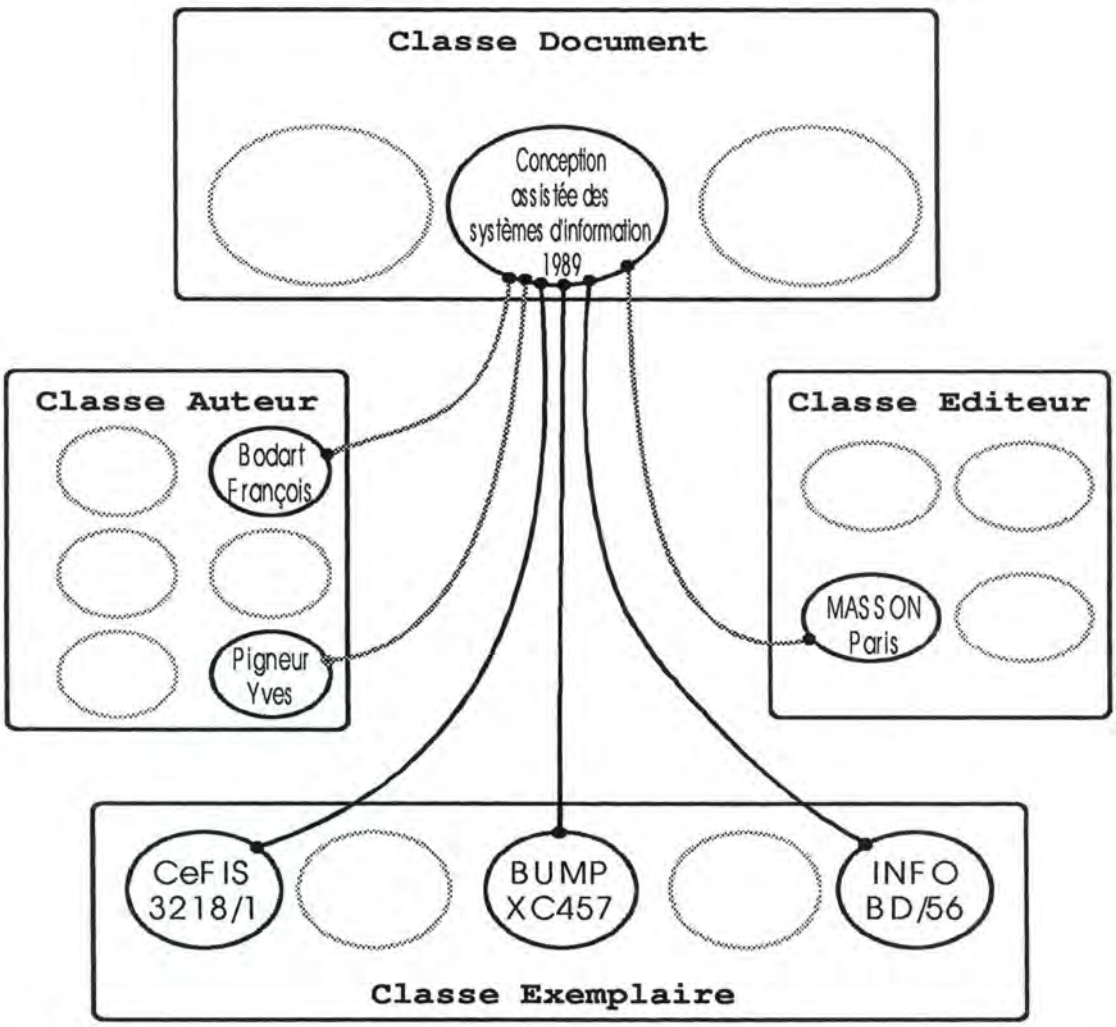



Figure 3.3 : Classes et instances pour la modélisation d'un document.

Considérant, par exemple, le document [BODART] de la bibliographie, la figure 3.3 représente une partie de la population d'objets nécessaires pour représenter ce

[9] Pour être précis, à une instance de la classe `Document`.

document selon les conventions qui viennent d'être décrites (on suppose connaître trois exemplaires de ce document localisés au CeFIS, à la BUMP et à l'Institut d'Informatique). Les classes sont représentées par des rectangles tandis que les instances (objets) sont représentées par des ovales. Signalons déjà que les arcs noirs traduisent des liens de composition tandis que les arcs grisés dénotent des liens de référence. Cette dernière distinction va être explicitée au paragraphe suivant.

### 3.2.1 Attributs

Comme dans beaucoup d'autres modèles de données, la structure des informations mémorisées par les instances d'une classe est décrite en faisant appel à des *attributs* [DELOBEL], [MEYER] parfois aussi appelée *variables d'instance* [MASINI] ou encore *champs* [FERBER].

Un attribut peut bien entendu désigner directement une valeur (un nombre, une chaîne de caractères, une date, ...). Il peut aussi pointer vers un autre objet ou même une collection d'autres objets (éventuellement de la même classe mais souvent d'autres classes).

Ainsi, pour décrire la classe `Document` des documents de notre bibliothèque, on pourrait déclarer :

```
Classe :
     Document
Attributs :
     Liste_auteurs
     Titre
     Edité_par
     Année_Publication
```

Figure 3.4 : Déclaration de la classe Document

Dans cette description l'attribut `Titre` désigne une chaîne de caractères alors que `Année_Publication` qualifie un nombre.

Par contre, les attributs `Auteurs`, `Edité_par` et `Exemplaires` ne contiennent pas, à proprement parler, de valeur, mais plutôt un ou des pointeurs vers d'autres objets, appartenant respectivement aux classes `Auteur`, `Editeur`, et `Exemplaire`.

Remarquons dès à présent que, "vu de l'extérieur", il n'est pas important de savoir si le nom de l'éditeur est directement enregistré sous forme de chaîne de caractères dans l'état des instances de `Document` ou si ce dernier doit le demander à un autre objet. A partir du moment où la classe `Document` offre, dans son interface, un service `Nom_éditeur`, la procédure qu'elle emploie pour trouver ce nom est sans importance pour les utilisateurs de ce service.

Il faut aussi observer que l'on peut facilement distinguer deux catégories de liens unissant un objet à un (ou plusieurs) autre(s).

- ***Composition***

  La relation qui associe objet un document aux objets qui représentent ses exemplaires fournit un bel exemple de lien de *composition* (ou lien "*part-of*" selon [ROLLAND]). En effet, l'existence des objets exemplaires est subordonnée à l'existence du document auquel ils sont associés : les premiers ne peuvent exister en l'absence du second [10].

- ***Référence***

  D'autres relations entre objets telles que celles qui unissent le document à ses auteurs ou à son éditeur ne subissent pas cette contrainte existentielle car chacun de ces objets peuvent exister indépendamment les uns des autres. Nous appellerons *référence* ce type de relation, terme qui nous paraît plus évocateur que association [ROLLAND]. Pour peaufiner encore le modèle de données, il pourrait être intéressant de distinguer les références *permanentes* (une fois établies, elles ne cessent qu'avec la disparition de l'un des objets) des références *temporaires* (dont la durée d'existence n'est pas liée à celle des objets associés). Par exemple, la référence entre un document et ses auteurs est évidement permanente alors que celle qui associe un exemplaire avec un emprunteur peut être qualifiée de temporaire.

On mesure ici la plus grande expressivité du modèle objet par comparaison au modèle Entité-Association. Dans ce dernier en effet, c'est toujours le même concept d'association qui sert à modéliser les différentes relations entre entités. La notion de composition est alors partiellement représentée par la cardinalité minimum associée au type d'association.

Etant donné l'intérêt de spécifier le type des attributs et la nature du lien d'association, nous adopterons dorénavant les conventions suivantes pour la représentation graphique des classes (notamment inspirées du modèle O* [BRUNET]) :

- Une classe est représentée par un rectangle contenant son nom ;
- Les attributs sont énumérés en dessous du rectangle et sont introduits par une petite flèche (indiquant que l'objet est "composé" de cet attribut) ;
- Chaque attribut est désigné par un nom qui permet de l'identifier parmi les autres attributs de la même classe. Des attributs de classes différentes peuvent porter des noms identiques ;
- Le caractère facultatif d'un attribut est indiqué par la mise entre parenthèses du nom de l'attribut ;
- Le type d'un attribut est désigné par :

[10] Certains auteurs dont [ROLLAND] justifient cette propreté en introduisant la notion de cycle de vie d'un objet et en montrant qu'il existe un relation d'inclusion entre les cycles de vie des objets composants et composés.

- le nom d'un type standard écrit en majuscule et précédé du symbole @ lorsque l'attribut désigne une valeur simple (nombre, date, chaîne de caractères, ...),

  ou bien

  - le nom de la classe du ou des objet(s) pointé(s) lorsque l'attribut désigne un pointeur vers un objet ou vers un ensemble d'objets;

- La nature du lien d'association et sa multiplicité sont représenté par différentes sortes de flèches :

| | | |
|---|---|---|
| Composition simple | `A ——> B` | "A est composé d'un B" |
| Composition multiple | `A —>> B` | "A est composé de plusieurs B" |
| Référence simple | `A ----> B` | "A réfère à un B" |
| Référence multiple | `A --->> B` | "A réfère à plusieurs B" |

En reprenant l'exemple de la figure 3.4, on a le schéma de la figure 3.5. Bien entendu, les classes `Auteur`, `Editeur` et `Exemplaire` doivent aussi être décrites pour que le schéma soit complet.

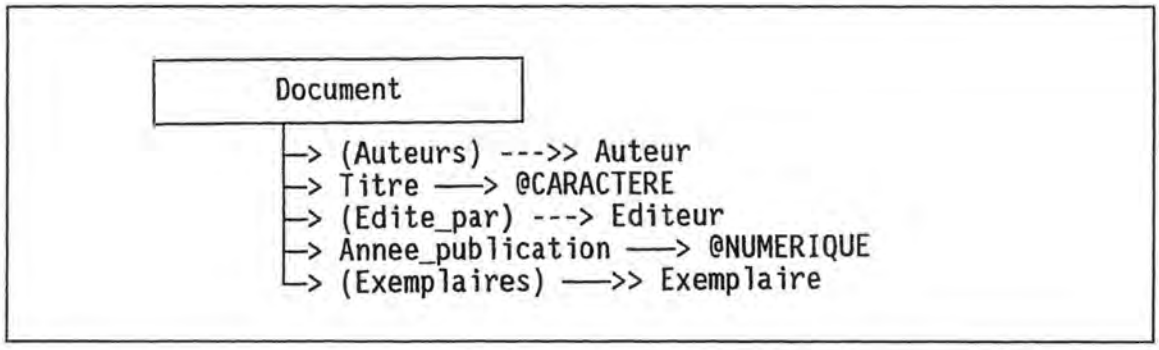



Figure 3.5 : Schéma des attributs de la classe Document

### 3.2.2 Opérations

Les comportements possibles des objets d'une classe sont décrits au moyen d'*opérations* souvent aussi appelées *méthodes*. Chaque opération correspond à un algorithme qui a pour effet de consulter ou de modifier les valeurs des attributs de l'objet et éventuellement de produire une valeur en résultat.

A chaque opération disponible correspond un nom de *message* qui peut être invoqué par tout utilisateur désirant voir s'effectuer l'opération demandée SUR l'un des objets de la classe.

Dans le schéma d'une classe, nous adopterons les conventions suivantes :

- Les opérations sont énumérées en dessous du nom de la classe (et éventuellement à la suite des attributs si ces derniers sont représentés) et sont introduits par une petite barre horizontale.
- Chaque opération est désignée par un nom qui l'identifie parmi les autres opérations de la classe. Le nom est suivi d'une paire de parenthèses renfermant éventuellement les noms des arguments à fournir à l'opération.

- Lorsque l'opération produit un résultat (l'opération est donc une fonction), le type de ce dernier est décrit en suivant les mêmes règles que pour les types des attributs (composition).

En pratique, le schéma de l'objet `Document`, déjà cité, peut être complété comme illustré sur la figure 3.6 où l'on observe trois opérations. `Description()` est une fonction qui retourne une chaîne de caractères, `Disponible()` en est une autre qui retourne une valeur booléenne tandis que `Reserver()` est une procédure qui nécessite un argument nommé `Lecteur`.

Cet exemple illustre clairement que, lorsque les noms des descripteurs sont bien choisis, ils permettent de se faire une idée intuitive relativement correcte de la spécification de chaque opération. Toutefois, ce schéma n'est pas complet à lui seul et doit être accompagné d'une description textuelle plus précise (éventuellement formelle) de la spécification de chaque opération.

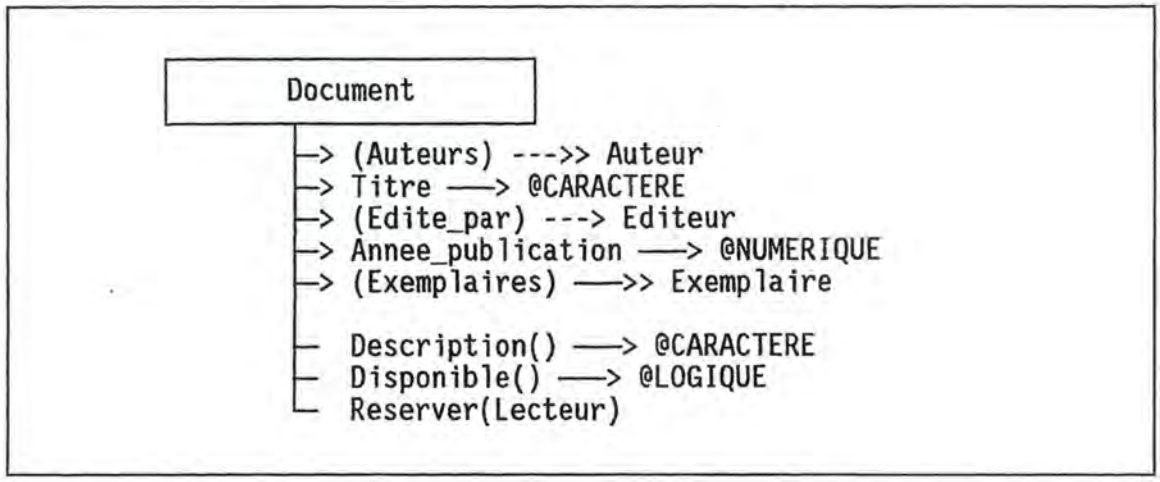



Figure 3.6 : Schéma de la classe Document

### 3.2.3 Messages

Si l'on se réfère au principe d'encapsulation qui est à la base du paradigme objet, les noms des attributs et des opérations ne doivent pas être rendus publiques. Seul, une série de noms de *messages* doivent être cités dans l'interface de la classe. En pratique toutefois, certains auteurs font discrètement l'impasse sur la spécification des messages en admettant implicitement que les noms des messages sont en fait les noms des opérations qui ont été définies (ainsi éventuellement que les noms des attributs). Si cet amalgame peut être toléré au niveau de la phase de conception d'un système, il est nettement plus dangereux lors de l'implémentation.

Aussi, la plupart des langages orientés objets offrent explicitement la possibilité de citer quels sont les messages qui sont accessibles à l'extérieur de la classe. C'est par exemple, le rôle de la close `export` du langage Eiffel [MEYER, 244] ou `public` dans C++. Eventuellement, il est possible de spécifier un nom de

message différent du nom de la méthode comme nous le verrons dans le cas de Class(y) [VANSTRAATEN].

### 3.2.4 Métaclasses

En poussant jusqu'au bout le principe de modélisation par objet, on est naturellement amené à vouloir considérer une classe comme... un objet à part entière. Une classe est en effet un objet puisqu'il existe des informations qui la décrive (par exemple son nom, le nombre de ses instances, le nombre de ses attributs et opérations...) et qu'elle est capable de certains comportements (le plus important étant sa capacité à générer des instances).

Or, puisque toutes les classes possèdent, entre autres, un nom, et sont capables de générer des instances, il est normal de considérer qu'il doit exister des classes dont les instances sont elles-mêmes des classes. Ces classes un peu particulières sont appelées des *métaclasses*. Le problème est toutefois circulaire car on peut alors créer des classes de métaclasses et il faut bien admettre qu'il doit exister une classe qui n'est pas un objet ou qui est sa propre métaclasse.

Certains langages orientés objet, dans la lignée de SMALLTALK, proposent un mécanisme permettant la gestion des métaclasses et disposent ainsi d'une très grande souplesse. Cette caractéristique est toutefois beaucoup plus rare dans les langages compilés.

## 3.3 Le mécanisme d'héritage

On a montré à suffisance que le principe d'encapsulation est l'un des atouts du paradigme objet. Le mécanisme d'héritage en est un autre tout aussi important.

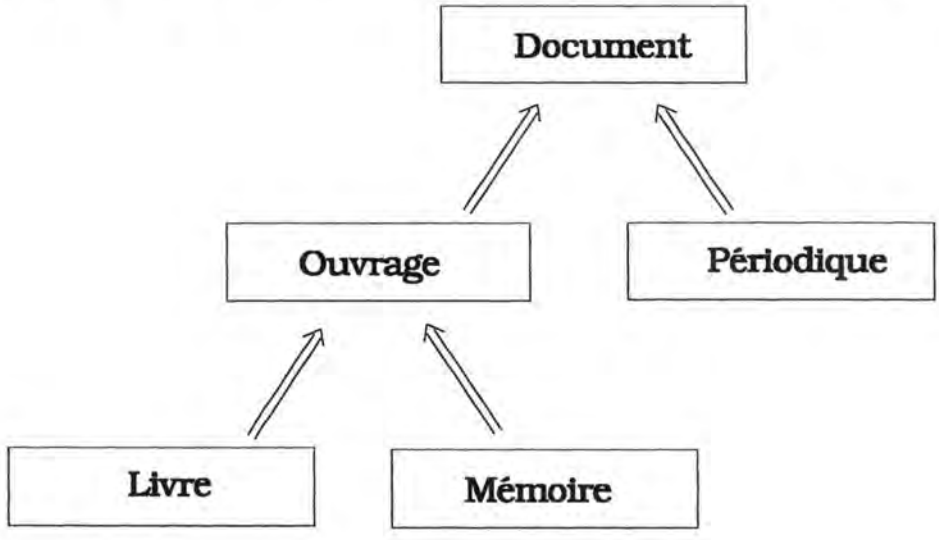



Figure 3.7 : Graphe d'héritage de la classe Document

### 3.3.1 Héritage simple

Si l'on désire poursuivre et raffiner la modélisation de la bibliothèque qui nous a servi d'exemple jusqu'ici, on est rapidement amené à observer que les documents appartiennent en fait à deux catégories : d'une part les ouvrages "isolés" et d'autre part les périodiques. Or, on sait que les descriptions bibliographiques de ces deux catégories de documents ne sont pas rédigées de la même façon. De plus, les exemplaires de ces documents ne sont généralement pas soumis aux mêmes conditions de prêt.

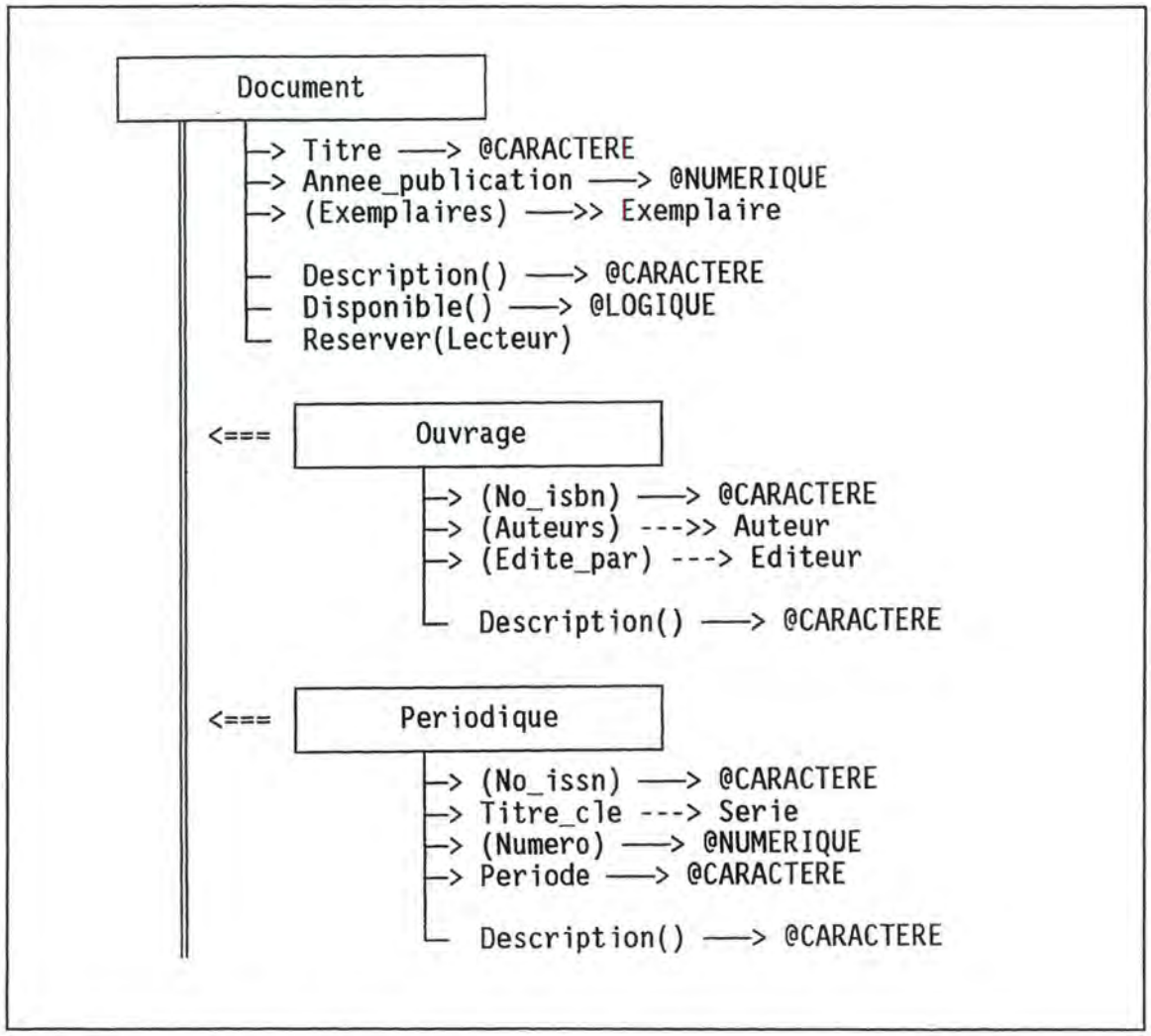



Figure 3.8 : Schéma de classe avec héritage

Le mécanisme d'héritage nous permet de modéliser facilement cette situation en notant qu'un `Ouvrage` *est un* `Document` ayant des caractéristiques particulières et qu'un `Périodique` *est un* `Document` ayant d'autres caractéristiques particulières. On dira que les classes `Ouvrage` et `Périodique` *héritent* de la classe `Document`

et l'on représente cette relation par une flèche double ==>[11] qui peut se lire "est-un" ou "hérite de". L'héritage peut éventuellement se réaliser sur plusieurs niveaux comme l'illustre le *graphe d'héritage* de la figure 3.7.

Dès le moment où l'on affirme que la classe `Ouvrage` hérite de la classe `Document`, on sait aussi que tous les attributs, opérations et messages définis dans la classe `Document` le sont aussi, implicitement, pour la classe `Ouvrage`. Il n'est donc pas nécessaire de réfléchir ou même de redéclarer tous ces éléments. Par contre, il est possible de déclarer de nouvelles caractéristiques propres aux instances des classes héritières. Ainsi, par exemple, si l'on précise la description des classes `Document`, `Ouvrage` et `Périodique`, on peut obtenir le schéma de la figure 3.8.

Fondamentalement, le mécanisme d'héritage trouve sa source dans le principe de *généralisation / spécialisation.* En effet, on peut dire, en se référant à notre exemple, qu'un `document` est une forme généralisée d'un `ouvrage` ou d'un `périodique` et inversement que ces derniers sont des formes spécialisées d'un `document` [BOUGHLAM, OOACONC-21].

### 3.3.2 Polymorphisme et surcharge

On notera, dans le dernier schéma, que l'opération `Description()` est citée aussi bien dans les classes héritières (appelées aussi sous-classes) que dans la classe dont elles héritent (*superclasse* ou sur-classe). En fait, le message `Description()` adressé à un objet de l'une de ces classes doit permettre d'obtenir une description bibliographique. Pour ce faire, une opération décrite au niveau de la classe `Document` est insuffisante car la façon de décrire un ouvrage et un périodique n'est pas identique. Toutefois, si un titre est présent, il devra y figurer dans les deux cas. En conséquence, il convient de déclarer dans les deux classes `Ouvrage` et `Périodique`, une opération ayant le même nom mais n'ayant pas exactement le même comportement.

Ce procédé porte le nom de *polymorphisme* : contrairement aux langages de programmations impératifs classiques l'appel de l'opération `Description()` ne produira pas toujours le même comportement et ce, suivant l'objet qui reçoit le message. Le polymorphisme induit une grande souplesse dans la programmation : ce n'est pas le programmeur qui détermine quel comportement activer (en Pascal, par exemple, il aurait fallu choisir d'activer l'une des procédures `Description_Ouvrage()` ou `Description_périodique()`) mais c'est le système lui-même qui le détermine en parcourant le graphe d'héritage.[12]

---

[11] Beaucoup d'auteurs représentent l'héritage par une flèche simple épaisse et parfois grisée [MASINI], [FERBER], [ROLLAND]. Nous avons préféré içi une flèche double qui est plus facile à dessiner et à reconnaître tant en écriture manuelle qu'avec l'aide de l'ordinateur.

[12] Ce mécanisme d'activation dynamique des opérations est la cause d'une perte de performance non-négligeable car, même pour un langage compilé, l'édition des liens ne peut être faite à priori et doit s'effectuer au coup par coup pendant l'exécution.

La recherche de la méthode à activer quand un objet reçoit le message s'effectue de la façon suivante : si l'objet est un `Ouvrage`, alors, il recherche le message dans l'interface de la classe `Ouvrage` et éventuellement, s'il ne l'y trouve pas, il recherche alors ce message dans l'interface de la superclasse (`Document`) puis éventuellement en continuant à remonter la lignée d'héritage. De cette façon, il n'y a plus d'ambiguïté entre les méthodes `Description()` des classes `Ouvrage` et `Périodique` car, suivant la nature de l'objet concerné, une seule peut être activée. De plus, l'ambiguïté est aussi levée quant au choix de l'opération entre les classes `Ouvrage` et `Document` : si le même message est présent, c'est toujours celui de la sous-classe qui est activé (en l'occurrence dans notre exemple, `Description()` de `Ouvrage`).

Toutefois, il reste possible à une opération de faire appel explicitement à une opération de même nom située plus haut dans la hiérarchie d'héritage. Ceci est réalisé en envoyant directement le message à sa superclasse grâce à une syntaxe spécifique (`super:Description()` dans Class(y)). Dans l'exemple qui nous occupe, il est ainsi possible aux opérations `Description()` des classes `Ouvrage` et `Périodique` d'utiliser les service de l'opération `Description()` de `Document` pour l'impression du titre de l'ouvrage/périodique. On dit alors que l'opération de la sous-classe *surcharge* celle de la superclasse ou que l'opération est *spécialisée*. Suivant les situations, la surcharge permet de compléter, de remplacer ou de rendre totalement inactif un comportement décrit dans une superclasse.

Polymorphisme et surcharge apportent beaucoup de puissance à l'approche objet. Ces deux mécanismes doivent cependant être manipulés avec discernement, sinon, ils risquent d'introduire pas mal de confusion dans les services activés par les messages. Des règles méthodologiques d'utilisation devront donc être définies.

### 3.3.3 Généricité

Nous avons montré au paragraphe 3.1.4 que l'un des intérêts du paradigme objet est sa capacité à faciliter la réutilisation de codes déjà rédigés. La mise en place d'opérations, voire de classes, génériques oeuvre dans le même sens. On pourra toutefois observer que la généricité n'est pas un concept spécifique du paradigme objet, comme le fait remarquer Bertrand Meyer, elle avait déjà été introduite dans le langage ALGOL 68 puis ADA. [MEYER, 470].

- ***Opérations génériques***

Une opération générique est une opération qui détaille un comportement en faisant appel à d'autres opérations, considérées comme élémentaires pour cette opération générique, mais qui ne sont décrites que dans les sous-classes et qui sont bien souvent des opérations polymorphes. Ainsi, l'opération générique a (en quelque sorte) des opérations "paramètres" qui sont "liées" dynamiquement suivant le contexte au moment de l'exécution.

Empruntons un exemple à Jacques Ferber pour illustrer cette notion [FERBER, 30-31]. Supposons que l'on ait décrit une classe `Objet_Graphique` et des sous-classes `Fenêtre`, `Icône` et `Polygone` qui en hérite. On peut alors définir dans la

classe `Objet_Graphique` une opération `Déplacer()` qui permette effectivement de déplacer l'objet sur l'écran. Tout naturellement, l'algorithme de cette opération peut se résumer en 3 envois de messages :

```
opération Déplacer()              // définie dans Objet_graphique
    Effacer()                     // Efface l'objet de l'écran
    Translater_Coordonnées()      // Détermine ses nouvelles coordonnées
    Afficher()                    // Affiche à nouveau l'objet à l'écran
```

Figure 3.9 : Algorithme de l'opération Déplacer

Les opérations `Effacer()` et `Afficher()` ne peuvent pas être décrite directement dans la classe `Objet_Graphique` car elles dépendent fortement du type d'objet considéré. Elles y seront déclarées "*deferred*" (différées) pour indiquer qu'elles doivent absolument être décrites à un niveau inférieur dans le graphe d'héritage. On implantera donc ces opérations dans chacune des trois sous-classes de `Objet_Graphique` et l'opération *générique* `Déplacer()` sera alors complètement décrite.

L'intérêt de l'organisation décrite ci-dessus devient évident lorsque l'on ajoute un nouveau type d'objet graphique : plus besoin de décrire comment le déplacer sur l'écran, il suffit simplement de préciser comment l'effacer et comment l'afficher. On a donc bien une économie d'écriture.

- ***Classes génériques***

Dès l'instant ou une classe contient une opération générique, on peut dire qu'elle est elle-même générique car elle ne peut plus être instanciée directement (certaines de ces opérations ne pouvant s'exécuter correctement). Plus généralement, on a souvent intérêt à décrire des classes auxquelles ne correspondent aucune instance mais qui rassemblent les caractéristiques (attributs et opérations) communes à plusieurs autres classes. Ainsi, déjà les classes `Document` et `Ouvrage` de la figure 3.7 peuvent être considérées comme classes génériques : elles ne font pas partie du schéma pour être instanciées, mais pour servir de cadre à un ensemble cohérent de caractéristiques. Les classes génériques doivent donc toujours être spécialisées en d'autres classes avant d'être utilisées.

### 3.3.4 Héritage multiple

Jusqu'à présent, toute classe faisant partie d'un graphe d'héritage ne possédait qu'au plus, un parent direct : il s'agissait d'héritage simple et le graphe d'héritage était en fait une arborescence. Dans certaines situations, il peut s'avérer intéressant d'élargir le mécanisme d'héritage pour qu'une classe hérite directement de plusieurs autres classes. On parle alors d'héritage multiple.

Revenons à l'exemple du graphe de la figure 3.7. Imaginons que l'on désire ajouter une nouvelle classe `Livre-en-série` qui modélise les livres qui paraissent à intervalles réguliers dans une collection. Dans un contexte où

l'héritage multiple est admis, il serait judicieux de faire hériter la classe `Livre-en-série` à la fois de `Livre` et de `Périodique` (figure 3.10) car, entre autre, un tel document doit être attaché à un (des) auteur(s) et à un titre-clé de collection.

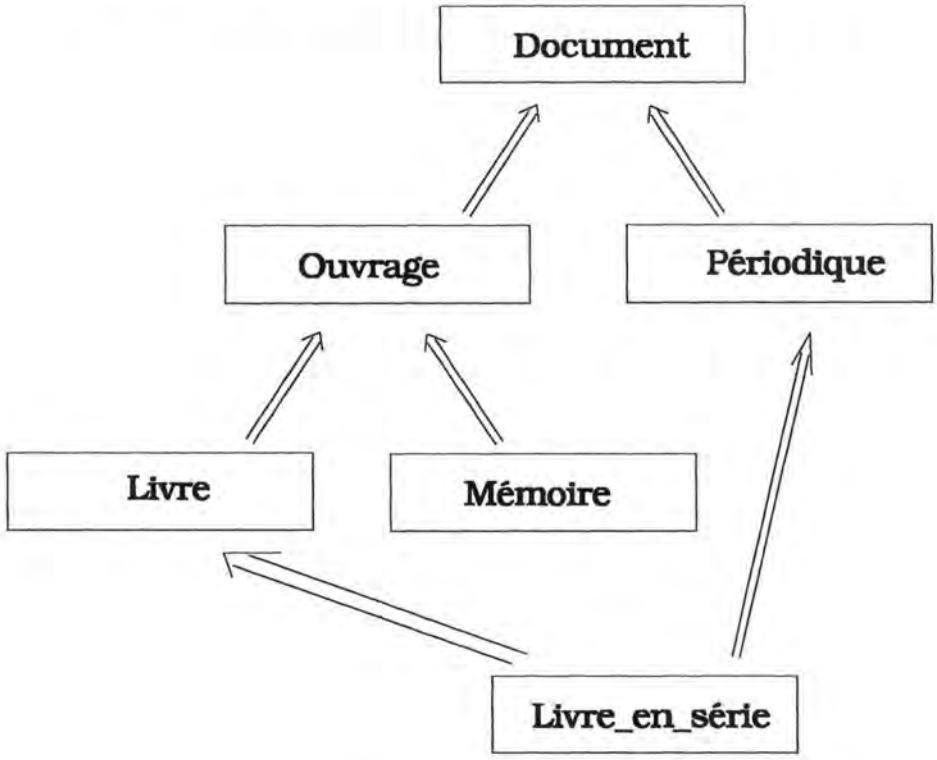



Figure 3.10 : Héritage multiple

L'héritage multiple accroît donc la souplesse de modélisation en donnant vraiment la possibilité de tirer profit de toutes les classes déjà définies à un instant donné. Par contre, il faut observer que la médaille à son revers et que des conflits peuvent apparaître. On se souvient avoir définit des opérations `Description()` spécifiques aux classes `Ouvrage` et `Périodique`. Quelle sera l'opération qui devra être activée pour une instance de `Livre-en-série` ? Aucun algorithme ne peut décider laquelle est la plus appropriée, seul le concepteur peut en décider. [MASINI, 55]. La définition d'une nouvelle opération `Description()` attachée à la classe `Livre-en-série` n'apporte de toute façon qu'une solution boiteuse car cette dernière opération peut elle-même avoir besoin de faire appel aux opérations de ces superclasses.

Les environnements qui offrent l'héritage multiple, intègrent donc nécessairement un mécanisme qui permet au système de lever les conflits. Ce peut être par exemple, en demandant au concepteur de citer explicitement le nom de la classe dans laquelle il faut rechercher le message (C++) ou en imposant de renommer les opérations dont le nom peut être ambigu (Eiffel). Ces contraintes restreignent évidemment les avantages de l'héritage multiple.

Aussi, étant entendu que d'un point de vue didactique, l'héritage multiple n'apporte pratiquement pas d'éléments nouveaux et que de surcroît, la bibliothèque Class(y) que nous utiliserons au niveau implémentation ne l'autorise

pas, nous nous permettrons de négliger, dans la suite de ce travail, cette extension du paradigme objet.

Notons toutefois qu'il est possible de simuler l'héritage multiple grâce à l'héritage simple. Pour ce faire, il convient d'abord de faire hériter directement la nouvelle classe de la classe qui lui apporte le plus d'éléments utiles. Ensuite, les attributs et opérations manquants sont recopiés purement et simplement en provenance de l'autre classe dont on aurait voulu faire hériter la nouvelle classe. Une autre solution, peut être plus élégante car induisant moins de redondance, consiste à construire la nouvelle classe comme une composition de classes [FERBER,70].

## 3.4 Le concept d'événement

Les concepts d'objet, de classe et d'héritage sont reconnus de façon pratiquement unanime par tous les auteurs traitant du paradigme objet, il n'en va pas de même pour le concept d'*événement* dont on ne trouve (pratiquement) pas trace dans plusieurs des documents cités dans la bibliographie et notamment [FERBER], [MEYER], [AUBERT] ou [DELOBEL].

D'un certain point de vue, cette absence est d'autant plus étonnante que l'une des justifications importantes du choix de la programmation orientée objet pour développer des logiciels "modernes" (des interfaces graphiques entre autres) réside précisément dans la possibilité d'adopter une programmation "événementielle". Toutefois, en analysant plus soigneusement ce paradoxe on constate que le concept d'événement relève essentiellement du niveau conceptuel et tend à disparaître dès qu'il est question d'implémentation. En conséquence, si l'on aborde le paradigme objet par le biais des langages de programmation orientés objet, le concept d'événement n'apparaît pas comme pertinent et est donc généralement ignoré.

Pour préciser ce concept d'événement dans le cadre du paradigme objet, nous nous baserons essentiellement sur les travaux de C. Rolland, d'A. Sernadas et de leurs équipes.

### 3.4.1 Notion d'événement

Mis en évidence depuis de nombreuses années déjà dans le cadre de la modélisation de la dynamique des S.I., le concept d'événement se définit comme "un changement d'état qui survient à un moment donné de l'évolution du système d'information et qui correspond à un stimulus auquel ce système doit réagir, principalement par le déclenchement de certains processus" [BODART, 72-73]. Vu dans le cadre du paradigme objet, les événements peuvent être décrits comme des "changements d'état particuliers d'objets qui déclenchent certaines opérations" [ROLLAND c, 78].

En fait, le concept d'événement permet de jeter un nouvel éclairage sur la modélisation de la dynamique dans une population d'objets. Lorsque nous avons écrit au paragraphe 3.1. qu'un objet intègre à la fois une composante statique (son

état) et une composante dynamique (son comportement), nous nous intéressions à la description de l'objet lui-même. Si l'on observe à présent que l'objet est amené à interagir avec d'autres objets de la population en leur envoyant des messages, il est utile d'introduire le concept d'événement pour pouvoir nommer la CAUSE de cette interaction. Citons Joël Brunet : "*Operations express how objects change. Events explain why they undergo changes*" [BRUNET a, 17].

L'équipe de A. Sernadas rejoint globalement cette interprétation mais en présentant plutôt l'événement comme une "action atomique" qui se produit pendant la vie de l'objet et qui peut avoir un effet sur ses attributs [SERNADAS a, 3.21] et [SERNADAS b, 3]. L'aspect intéressant est ici la mise en évidence du caractère atomique des opérations de changement d'état qui sont, de plus, réputées s'exécuter en une durée nulle.

### 3.4.2 Catégories d'événements

Les événements qui peuvent être associés aux objets peuvent être classés en événements internes, externes ou temporels [BRUNET a, 17].

Un *événement interne* correspond à un changement des informations d'état de l'objet qui deviennent telles que la condition d'occurrence de l'événement se trouve remplie. Par exemple, l'abaissement du nombre de produits de stock en dessous d'un certain seuil provoque l'occurrence de l'événement "Demande de réapprovisionnement". Un objet tel qu'il existe au moins un événement qu'il soit capable de déclencher par sa propre initiative est parfois appelé un *objet actif* [SERNADAS a, 3.27][13] Par opposition, les autres objets sont dits passifs.

Un *événement externe* est, quant à lui, généré par l'environnement du système. Une demande d'emprunt d'un ouvrage constitue par exemple un événement externe. Rolland et Cauvet introduisent le concept d'*acteur* pour spécifier plus clairement l'origine des événements externes.

Un *événement temporel*, comme son nom le suggère, est généré par l'écoulement du temps et le passage par des instants particuliers tels que l'heure de clôture d'une certaine tâche ou le dépassement d'un délai (p. ex. restitution d'un ouvrage). Par souci de cohérence, certains introduisent un objet spécial "*clock*" qui n'a d'autre tâche que de déclencher ces événements temporels.

Il nous apparaît toutefois que la distinction entre ces trois catégories d'événements n'est évidemment pas toujours exclusive. Par exemple, l'événement "Demande de fermeture" d'une fenêtre dans une interface graphique peut être à la fois interne (généré par la modification d'un autre objet du système), externe (demandé explicitement par l'utilisateur du logiciel) voire temporel (déclenché après un certain délai).

---

[13] Cette idée se retrouve aussi dans [LAI, 122] bien que formulée de façon assez différente.

### 3.4.3 Cycle de vie

A partir du moment où l'on reconnaît que l'existence d'un objet est jalonnée par les événements qu'il génère et qu'il subit, on est naturellement amené à introduire la notion de cycle de vie. Le cycle de vie d'un objet est constitué par la séquence des événements qu'il subit. Il commence par un événement de naissance (création de l'objet), se poursuit par zéro, un ou plusieurs événements de modification et se termine (éventuellement) par un événement de mort (élimination de l'objet) [SERNADAS a, 3.29].

Entre les instants d'occurrence de ces événements, l'objet se trouve dans des *états remarquables* [ROLLAND b,311] qui peuvent être assez facilement identifiés. Par exemple, un objet exemplaire peut être Disponible, Emprunté, Réservé ou Perdu. Les transitions possibles entre les états remarquables peuvent avantageusement être représentées par un graphe des états et transitions valides. La figure 3.11 donne un exemple d'un tel graphe (emprunté à [ROLLAND b, 311]) selon les conventions du modèle O*. Suivant les auteurs, chaque transition sera associée à l'occurence d'un événement ou éventuellement à l'exécution d'une opération, ce qui revient pratiquement au même puisque la seconde est déclenchée par le premier.

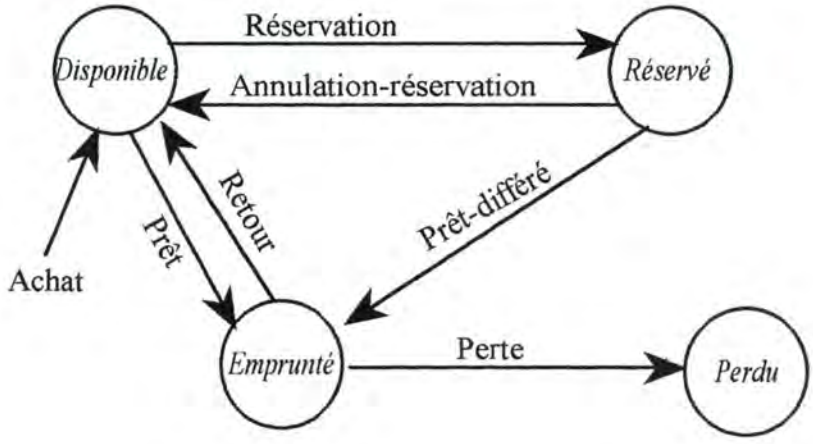



Figure 3.11 : Graphe des états et transitions valides d'un objet exemplaire

### 3.4.4 Exploitation du concept d'événement

Dans le cadre de la conception de logiciels et de S.I., il apparaît donc intéressant de repérer soigneusement les événements qui interviennent et de les caractériser par un nom, par le prédicat qui spécifie leur condition d'occurrence et/ou par le nom des acteurs qui les génèrent ainsi que par la liste des opérations déclenchées.

En effet, le concept d'événement s'avère extrêmement utile pour modéliser la dynamique des objets entre-eux et vis-à-vis de l'environnement. Comme le souligne Colette Rolland, "la notion d'événement permet au niveau conceptuel une spécification plus déclarative du comportement d'un objet; laissant au niveau de la réalisation la possibilité d'implanter la notion d'événement de différentes façons" [ROLLAND b, 312]. C'est finalement pour cette raison que le concept

d'événement n'apparaît pas explicitement dans les langages orientés objets mais s'y trouve "déguisé" sous les conditions d'activation des messages.

## 3.5 Le langage Class(y)

Au terme de cette rapide exploration du paradigme objet, il est intéressant de montrer un exemple concret de langage de programmation implémentant ce paradigme. Les pages qui vont suivre proposent donc une présentation du langage offert par la librairie Class(y) développée par Integrated Development Corporation et qui permet d'étendre le langage du compilateur Nautucket Clipper 5.01 de manière à pouvoir créer et manipuler des classes et des objets. Pour les lecteurs non familier du monde Xbase, nous allons commencer par un très bref rappel concernant dBase et Clipper.

### 3.5.1 Le langage Clipper 5.01

***Un peu d'histoire***

Il y a une bonne dizaine d'année déjà, dBASE II s'est imposé sur les micro-ordinateurs de première génération (à base de Z80) comme un des logiciels de gestion de fichiers parmi les plus novateurs et les plus intéressants. Transféré sur les micro-ordinateurs 16 bits PC et compatibles, sous le nom de dBASE III, le logiciel est devenu pendant plusieurs années, la référence en matière de gestion de bases de données pour micro. Son succès était dû à la fois à sa capacité à gérer simultanément jusqu'à 10 fichiers de données qui pouvaient être interconnectés à la manière des BD relationnelles[14], mais aussi à la richesse de son langage de commande qui autorise aussi bien la manipulation interactive que la rédaction de véritables logiciels qui peuvent alors être utilisés par des personnes n'ayant aucune connaissance du langage.

Aujourd'hui encore, même si dBASE III est largement dépassé par d'autres produits, le format de ses fichiers est presque devenu un standard de fait et de nombreuses firmes ont développé des logiciels ou des compilateurs qui utilisent les fichiers au format dBASE et qui reconnaissent son langage de commande. Ce sont, entre autre, Nantucket Clipper, Fox Pro, dBFast, ... Clipper est donc un héritier de dBASE III même si son langage a été sensiblement étendu afin d'améliorer ses performances et de simplifier le travail du développeur.

***Caractéristiques du langage***

Clipper est un langage structuré faiblement typé rappelant tantôt Cobol (pour ses éléments les plus anciens) et tantôt C (pour les ajouts les plus récents). Comme dBASE, Clipper reconnaît les quatre types de variables : numérique, chaîne de caractères, date et logique. Une originalité du langage réside dans le fait que le

[14] Bien que dBASE III ne puisse aucunement revendiquer le titre de SGBD relationnel

type d'une variable n'est pas nécessairement fixe pendant l'exécution (typage dynamique) : il peut donc varier au gré des affections subies. Avec la version 5 du langage, sont apparus les deux nouveaux types tableau et objet. Les variables objet ne peuvent toutefois être instances que de l'une des quatre classes prédéfinies dans le langage[15] et il n'est pas possible de déclarer de nouvelles classes. Le caractère "orienté objet" de Clipper 5.0 est donc très limité.

Les algorithmes peuvent être modularisés en procédures et en fonctions avec possibilité d'appels récursifs, passage d'arguments par valeur ou par référence, variables locales,... Différentes variantes des structures alternatives et répétitives sont disponibles. Clipper dispose donc de tous les éléments nécessaires à une bonne structuration des programmes selon les usages de la programmation impérative.

De plus, diverses firmes ont, à ce jour, rédigé plus d'une centaine de librairies pouvant être associées à Clipper pour en élargir les possibilités, et Computer Associates, la société qui vient d'acquérir le langage, compte bientôt mettre sur le marché une version entièrement orientée objet et adaptée à l'environnement Windows. Dès lors, on comprendra que ce langage est aujourd'hui fort prisé par les développeurs d'applications de gestion sur micro.

### 3.5.2 Déclaration des classes

Voyons à présent ce qu'apporte la librairie Class(y) qui prétend étendre, dès aujourd'hui, Clipper 5.0 pour en faire un langage orienté objet.

Avec Class(y), chaque classe doit être décrite dans un fichier distinct commençant par la déclaration

```
#include "CLASSY.CH"
```

Le fichier CLASSY.CH assure l'activation automatique des procédures de la librairie CLASSY.LIB nécessaires à la gestion des classes définies par l'utilisateur. CLASSY.CH contient également la description syntaxique du langage et cette dernière peut éventuellement être modifiée par l'utilisateur, ce qui confère une grande souplesse au langage[16].

Chaque fichier de description d'une classe se divise en deux parties distinctes que nous appellerons déclaration et implémentation. La partie déclaration commence par les mots `create class` suivis du nom que l'on veut attribuer à la classe et se termine par les mots `end class`.

La partie déclaration de la classe permet d'annoncer les attributs - appelés `var` - les opérations - appelées `method` - et éventuellement les messages - `message` - renommant certaines opérations. Par défaut, tout nom d'attribut ou d'opération

---

[15] Il s'agit des classes `Error`, `Get`, `TBrowse` et `TBColumn` dont les caractéristiques et les usages sont très spécifiques à l'environnement Clipper.

[16] Nous ne présentons ici qu'une seule expression pour chaque commande même si le langage offre des synonymes et des formes abrégées.

est automatiquement déclaré par Class(y) comme nom de message rendant ainsi généralement superflue la déclaration explicite des messages qui pourront être employés pour dialoguer avec les objets de la classe. Il ne faut déclarer de message spécifique que lorsque le nom que l'on veut donner au message est différent de celui que portera l'opération correspondante[17].

***Protection des messages***

Toutes les manipulations des attributs (lecture de la valeur, affectation) ou appel d'opération se réaliseront toujours par envoi de messages aux objets concernés. Toutefois, pour limiter l'emploi de certains messages, il convient de spécifier le degré de protection (d'encapsulation, si l'on préfère) que l'on désire pour chaque attribut ou opération. A cette fin, on regroupera ces derniers en trois catégories introduites par les mots-clés `hidden:`, `protected:` et `exported:`.

Un attribut ou une opération - c'est-à-dire finalement un message ou la réponse à un message - déclaré `hidden:` est, comme son nom le suggère, totalement caché au monde extérieur, il ne peut être utilisé que dans l'implantation de la classe où il a été défini et dans elle seule. Un message `protected:` peut de plus être invoqué par les classes héritière de la classe où il a été défini tandis qu'un message `exported:` peut être invoqué par toute classe ou par n'importe quelle partie du logiciel où la classe est connue. Les messages déclarés `exported:` (et dans une moindre mesure `protected:`) constituent, à proprement parlé, l'interface de la classe.

Concernant les attributs, il est possible de moduler un peu plus leur protection en les déclarant `readonly`. De cette façon, leurs valeurs peuvent être lues "de l'extérieur" mais ne peuvent pas être modifiées. Cette disposition est fort intéressante car elle permet de déclarer les attributs `exported:` et d'offrir ainsi un message de consultation accessible de partout, tout en empêchant toute mise à jour de la valeur autrement qu'en invoquant une opération spécifiquement définie. La déclaration `readonly` est également utilisable pour les attributs `protected:` mais est évidemment incompatible avec `hidden:`.

***Attributs et opérations de classe***

Le mot clé `class` précédant `var` ou `method` permet de déclarer des attributs de classe et des opérations de classe. Bien que Class(y) ne le signale pas, les `class var` et `class method` peuvent en quelque sorte être considérés comme des attributs et opérations de la métaclasse. En effet, une classe ne mémorise qu'une seule valeur pour chaque attribut de classe contrairement aux attributs "normaux" pour lesquels il existe une valeur associée à chaque instance.

Les attributs de classe permettent donc de stocker des informations "globales" concernant l'ensemble de la classe[18]. Leur valeur peuvent ainsi être partagées par

---

[17] C'est notamment nécessaire lorsque le nom du message est un mot réservé du langage.

[18] Cette possibilité illustre à nouveau la dualité sous-jacente au concept de classe qui est à la fois un modèle (un type, une abstraction) de données et un ensemble d'objets construits sur ce modèle.

```
#include "CLASSY.CH"

create class DOCUMENT

   hidden:
      var Reservations           // Liste des reservations en cours[19]

      class var NbreDocuments    // Nombre total de documents

   protected:
      method Annul_Reserv        // Annulation de la réservation

      class method Increment_NbDoc // Incremente NbreDocuments

   exported:
      var Titre  readonly        // Titre du document : caractères
      var Annee_publ  readonly   // Année de publication : numerique
      var Exemplaires  readonly  // Liste des exemplaires : tableau dynamique

      method Ajouter_exemplaires  // Ajout de nouveaux exemplaires
      method Reserver            // Demande de réserv. pour ce lecteur
      method Description         // Donne une description du document
      method Disponible          // -> Vrai si au-moins 1 exemplaire en rayon

   init class:
      ::NbreDocuments := 0

   end class
```

Figure 3.12 : Déclaration de la classe Document en Class(y)

toutes les instances. Par exemple, un attribut de classe pourrait être employé pour mémoriser la couleur d'objets graphiques si tous les objets d'une même classe ont la même couleur.

Dans le même ordre d'idée, les `class method` sont prévues pour manipuler la classe dans son ensemble, elles n'ont donc accès qu'aux attributs de classe. Leur spécificité est moins évidente car Class(y) accepte que les méthodes "normales" consultent les `class var` (ce qui est très utile comme l'illustre l'exemple précédent) et en modifient la valeur (ce qui est nettement plus dangereux).

Enfin, notons que les variables de classe peuvent être initialisées au moment de la déclaration de la classe en terminant cette dernière par une clause `init class:` suivie des instructions appropriées.

Illustrons cette brève présentation par un exemple (figure 3.12). Les commentaires ajoutés à droite du code // donnent une courte description de chaque attribut ou opération.

---

[19] L'attribution des différents niveaux de protections aux variables et méthodes relève essentiellement du souci d'illustrer les variantes de syntaxe.

### 3.5.3 Implémentation

La partie implémentation comprend le détail des algorithmes à exécuter pour chacune des opérations énumérées dans la déclaration. Elles se présente donc comme une suite de procédures et de fonctions très semblables à du code Clipper 5.0 non orienté objet. Les mots-clés `procedure` et `function` doivent seulement être précédés de `method` pour être associés au message correspondant de la déclaration.

***Constructeur d'instance***

Une procédure très particulière doit cependant être ajoutée : c'est le *constructeur* d'instance qui porte normalement le nom `New()`. Cette procédure doit être annoncée par les mots `constructor New()`. Cette opération sera invoquée chaque fois que l'on voudra créer un nouvel objet c'est-à-dire une nouvelle instance de la classe. Le code de la procédure décrit les actions à poser sur la nouvelle instance pour la rendre utilisable. Il s'agit, le plus souvent d'une séquence d'affectation permettant d'attribuer une valeur initiale à chaque attribut. On en donnera bientôt un exemple dans la figure 3.13.

***Envoi de messages***

Que l'opération soit un constructeur ou pas, toutes les manipulations d'attribut et tous les appels à d'autres opérations se réalisent par envoi de messages en observant les règles de syntaxe suivantes qui sont, du reste, également valables en dehors de la partie implémentation de la classe.

L'envoi d'un message `message` à un objet `objet` s'écrit `objet:message`. Si le message est un appel à une opération ayant des arguments, ces derniers suivent le nom du message et sont encodés par des parenthèses. En voici quelques exemples :

```
oDoc:Reserver(oLecteur)
```

envoie le message `Reserver` à l'objet `oDoc` en lui passant l'argument `oLecteur`.

```
if oDoc:Disponible() ...
```

envoie le message `Disponible()` à l'objet `oDoc` qui retournera une valeur logique.

```
oDoc:NbreDocument:= 0
```

affecte la valeur zéro à l'attribut `NbreDocument` de l'objet `oDoc`.

```
Texte:= oDoc:Editeur:Adresse()
```

affecte à la variable `Texte` le résultat retourné par l'objet `oDoc` auquel on a envoyé le message `Editeur:Adresse()`. Pour ce faire, il envoie lui-même le message `Adresse()` à son attribut `Editeur` qui est lui-même un objet (d'une autre classe).

*Messages à l'objet lui-même*

Pour implémenter les opérations d'une classe, il est constamment nécessaire de faire appel aux attributs et aux opérations de cette classe. Or, on ne connaît pas à l'avance les noms que porteront les instances de la classe. Aussi, l'identificateur `self` est-il utilisé pour désigner l'instance sur laquelle est exécutée l'opération. Les envois de message à l'objet lui-même s'écrivent donc, par exemple, `self:Disponible()` ou `self:NbreDocument`. Class(y) autorise une écriture plus compacte en remplaçant `self:` par `::` . Nous adopterons systématiquement cette écriture dans la suite. La figure 3.13, donne à titre d'exemple, le texte de l'implémentation de quelques-unes des opérations déclarées dans la figure 3.12.

```
constructor New()

   // Initialisation des attributs de la nouvelle instance
   ::Titre         := ""           // Chaine vide
   ::Annee_publ    := 0
   ::Exemplaires   := {}           // Tableau dynamique vide
   ::Reservations  := {}           // Tableau dynamique vide
   ::Increment_NbDoc()             // Envoi à soi-même du message Increment_NbDoc
   return


method function Disponible()

   local Trouve,i                  // Déclaration de variables locales non typées

   Trouve := .F.                   // .F. signifie false
   i := 1
   do while i <= len(::Exemplaires) .and. .not. Trouve
     Trouve := ::Exemplaires[i]:Disponible()
       // Envoi du message Disponible() au ième objet de la suite des exemlaires
       // et récupération du résultat dans la variable Trouve
     i := i+1
     enddo
   return Trouve


method procedure Increment_NbDoc()

   ::NbreDocuments := ::NbreDocuments+1
   return
```

Figure 3.13 : Implémentation (partielle) de la classe Document

### 3.5.4 Mise en oeuvre des objets

L'exploitation d'une population d'objets au sein d'un programme est fort simple et consiste essentiellement à envoyer des messages. Cependant, avant de pouvoir

utiliser un objet, il faut d'abord le créer en instanciant une classe et lui attribuer un nom qui permettra de le désigner par la suite. Cette création s'effectue en envoyant le message "constructeur" à la classe avec la syntaxe `Classe():New(arguments_éventuels)`. Par exemple, l'instruction

```
NouvDoc := Document():New()
```

crée une instance de la classe `Document`, l'initialise suivant les directives de constructeur `New()` et place dans la variable `NouvDoc` un pointeur vers cet objet.

Notons que la variable `NouvDoc` ne contient pas l'objet mais seulement un pointeur sur cet objet. De cette façon, l'affectation

```
CopieDoc := NouvDoc
```

n'a pas pour effet de dupliquer l'objet mais seulement de fournir un nouveau nom permettant de le désigner. Class(y) n'offre d'ailleurs pas de procédé simple permettant de dupliquer effectivement un objet alors que d'autres langages tel Eiffel offrent par exemple la primitive `clone` [MEYER, 112].

Notons enfin que deux messages "utilitaires" sont automatiquement prédéfinis par Class(y). Le message `class` permet essentiellement de comparer deux objets pour savoir s'ils appartiennent à la même classe en donnant une instruction telle que :

```
if objet1:class == objet2:class ...
```

Le message `className` retourne quant à lui le nom de la classe sous forme de chaîne de caractères.

### 3.5.5 Gestion de l'héritage

La création d'une classe héritière d'une autre est extrêmement simple : il suffit de compléter la déclaration de son nom par le mot `from` suivi du nom de la superclasse. Par exemple :

```
create class Ouvrage from Document
```

déclare que la classe `Ouvrage` hérite de la classe `Document`. Cela signifie donc qu'elle dispose automatiquement des attributs et opérations `protected:` et `exported:` de sa superclasse (qu'ils soient d'instance ou de classe). La classe héritière peut bien entendu déclarer ses propres attributs, opérations et messages qui peuvent être `hidden:`, `protected:` ou `exported:`.

En application du principe de surcharge, il est même possible de déclarer des opérations de même nom que ceux de la superclasse. Si l'on désire seulement rendre inactive une opération de la superclasse, il suffit de faire suivre son nom - dans la partie déclaration - par le mot `null`. Si par contre, on désire que l'opération de la classe héritière agisse différemment de celle décrite dans la superclasse, il faudra en spécifier complètement l'algorithme dans la partie implémentation. Toutefois, il reste possible de faire explicitement appel aux

services de la superclasse en envoyant les messages appropriés, non plus à `self`, mais à `self:uper` ou simplement `::super`.

Exemple :

```
method procedure Description()
     (...)
     ::super:Description()
     (...)
```

Pour écrire des opérations génériques, on est amené à faire appel dans les algorithmes de la superclasse, à des opérations qui ne sont décrites que dans les classes héritières. Il convient alors de déclarer ces messages dans la superclasse en faisant suivre leur nom du mot-clé `deferred`.

Enfin, il faut remarquer que s'il n'y a pas de raison à surcharger un attribut d'instance, il n'en va pas de même pour un attribut de classe. En effet, si l'on se remémore l'exemple de la couleur des objets graphiques (cf. 3.5.1.), la définition en surcharge, d'un attribut de classe pour une sous-classe de `Objet-graphique` permettrait d'imposer une couleur différente pour les objets de la sous-classe tout en conservant la couleur originale pour les objets de la superclasse.

# 4

# Conception orientée objet

De nombreux auteurs ont, depuis le milieu des années quatre-vingt, travaillé à la mise au point de méthodes de conception qui se revendiquent du paradigme objet. Ce sont, pour n'en citer que quelques uns, Grady BOOCH, Peter COAD, Edward YOURDON, Bertrand MEYER, ... Dans le présent chapitre, nous allons examiner trois de ces méthodes afin de mettre en évidence les principaux concepts utilisés et la démarche proposée. Dans la mesure du possible, nous tenterons de relever les limites et écueils qui leur sont associés.

Il faut toutefois noter, dès à présent, que notre objectif n'est absolument pas de réaliser ici une présentation exhaustive des trois méthodes retenues mais de rechercher, à travers ces trois exemples, les concepts que l'on pourrait considérer comme les plus importants et que nous pourrions intégrer dans notre méthode de conception "simplifiée". Nous nous attacherons donc plus à cerner l'"esprit" de ces méthodes et des modèles de représentations associés que les détails liés par exemple à leur syntaxe.

De plus, il ne faut pas chercher dans le choix des trois méthodes qui vont être examinées une quelconque volonté de couvrir systématiquement les grandes tendances du moment en matière de conception orientée objet. Le choix de O* a été directement influencé par la remarquable présentation qu'en a faite Madame Colette ROLAND dans le cadre de la Chaire Francqui 1992, tandis que celui d'OBLOG me fut conseillé par Messieurs HABRA et DUBOIS de l'Institut d'Informatique. La méthode HOOD, découverte presque par hasard lors de recherches bibliographiques, a été retenue parce qu'elle donne un éclairage assez différent de la conception orientée objet.

***Grille d'analyse***

Pour structurer notre examen des trois méthodes retenues, nous adopterons, autant que faire se peut, une grille d'analyse qui peut être articulée autour d'une série de questions :

- Quel est l'objectif annoncé de la méthode ? Quels domaines d'applications sont visés ?
- Quel modèle de représentation est utilisé ? Quels concepts sont considérés comme essentiels ? Quel formalisme est adopté ?
- Quelles sont les étapes proposées pour la conception ? Existe-t-il des outils d'assistance dans la mise en oeuvre de la démarche ?
- Quel est le résultat de la conception ? La phase d'implémentation est-elle prise en charge par la méthode ?

## 4.1 O* et l'approche en fontaine

Le modèle O* a été conçu dans le cadre du projet "Business Class" du programme Esprit II. Il est décrit dans [BRUNET a] et [ROLLAND a][1]. La méthode de conception associée, dite "en fontaine", est décrite dans [ROLLAND a] et [ROLLAND b] mais est empruntée à Henderson et Edwards[2] [ROLLAND a, 314]. Paradoxalement J. Brunet présente presque exactement les mêmes étapes mais parle de "modèle en spirale" et fait référence aux travaux de Boehm [3] [BRUNET a, 35].

### 4.1.1 Objectifs

Le modèle O* a été conçu pour la conception de vastes applications de gestion faisant intervenir des systèmes distribués et des communications avec des applications traditionnelles. Il repose largement sur les travaux de C. Rolland et C. Cauvet sur la conception des systèmes d'information.

### 4.1.2 Le modèle O*

Toute modélisation O* s'articule autour de la mise en évidence de deux catégories d'entités : les objets d'une part et les acteurs d'autre part. Nous allons voir que les premiers servent à décrire les composants du système d'information lui-même tandis que les seconds permettent de représenter l'environnement dans lequel le S.I. est appelé à fonctionner et avec lequel il doit interagir.

***Objet***

A la base du modèle, se trouve donc le concept d'objet qui est défini comme la projection d'un phénomène réel qui est persistant et d'intérêt pour le monde organisationnel. Un objet est décrit par un triplet (Id, Φ, V) où Id est un

---

[1] Dans ce chapitre, nous nous permettons de ne donner de références explicites que lorsqu'il existe des différences d'interprétation entre plusieurs papiers relatifs à un même modèle.

[2] HENDERSON-SELLER B., EDWARDS J.M. : The Object-oriented Systems Life Cycle, Communications of the ACM, Vol. 33, Nb 9, September 1990.

[3] BOEHM B.W., A Spiral Model of Software Development and Enhancement, Software Engineering Project Management, 1987.

identifiant, Φ un cycle de vie et V un ensemble de valeurs décrivant l'état de l'objet.

Chaque objet O* possède donc, indépendamment de son état, un identifiant permanent, que l'on qualifie d'identifiant interne, et qui permet de le désigner de façon univoque tout au long de son existence. Le modèle O* permet de plus, par analogie avec la notion de clé dans le modèle relationnel, de définir un, voire plusieurs identifiants faisant référence aux valeurs de l'état de l'objet (et que l'on peut alors qualifier d'identifiant externe).

Le cycle de vie d'un objet est constitué par la séquence des occurrences événements qui ont affecté la vie de l'objet (en modifiant, entre autres, les valeurs de l'état de l'objet). Un cycle de vie d'un objet commence nécessairement par un événement de naissance et se termine (éventuellement) par un événement de mort. De plus, nous verrons bientôt que la définition du concept d'événement est telle qu'il est impossible que deux événements se produisent simultanément. Par conséquent, l'ensemble des événements subis par un objet est ordonné strictement par rapport au temps.

***Schéma d'objet***

Dans le modèle O*, les caractéristiques communes à un ensemble d'objets "semblables" sont décrites dans un schéma d'objet (*object scheme*) et les objets de cet ensemble forment l'extension du schéma d'objet.

La notion de schéma d'objet est évidemment fort semblable à celle de classe telle que nous l'avons définie au chapitre précédent mais J. Brunet fait remarquer qu'elle s'en distingue par le fait qu'un schéma d'objets modélise toujours des objets persistants qui ont un correspondant effectif dans le monde organisationnel. De plus, nous allons voir que la description des comportements ne se limite pas à l'énoncé de méthodes mais comprend également le relevé des contraintes subies et des événements dont l'occurrence peut être déclenchée par des objets du schéma. La distinction explicite entre schéma d'objet et extension du schéma d'objet permet, de surcroît, de bien dissocier la définition des objets de leur mise en oeuvre.

La description complète d'un schéma d'objet comprend sept aspects distincts mais complémentaires : les attributs, les références, les contraintes, le graphe des transitions d'état, les opérations, les événements et la hiérarchie d'héritage. Les quatre premiers aspects forment la spécification statique du schéma tandis que les deux suivants constituent sa spécification dynamique.

*a) Attributs*

Les attributs donnent la description structurelle d'un schéma d'objet en faisant uniquement intervenir le lien de composition. Les attributs (composants) d'un objet font donc partie intégrante de l'objet et la création ou le suppression de l'objet composé (dit aussi agrégat) entraîne automatiquement la création ou la suppression des valeurs des attributs composants.

Les valeurs des attributs peuvent être prise dans les domaines suivants :

- domaines pré-définis (ex.: `INTEGER, STRING, ...`)
- domaines utilisateurs :
  - domaines complexes
    (ex.: `compte = {numéro : INTEGER, solde : DECIMAL}`)
  - schémas d'objets
    (ex.: un objet `commande` est composé, entre autre, d'un ensemble d'objets `ligne`)

De plus, O* distingue les attributs simples (contenant une seule valeur) et multiples (contenant un ensemble de valeurs prises dans le même domaine).

*b) Références*

Les références permettent de décrire des liens entre des objets ayant des "existences" bien distinctes (n'étant pas composé ou composant l'un de l'autre). Les références permettent d'associer un même objet avec plusieurs autres. Par exemple, le lien associant une ligne de commande avec le produit commandé est un lien de référence et le produit commandé peut bien entendu être référencé par de nombreuses lignes de commande.

*c) Contraintes*

Un schéma d'objet est généralement le siège d'une série de contraintes qui peuvent être classées en contraintes d'attributs et contraintes d'unicité.

Les contraintes d'attributs permettent de raffiner la spécification des attributs en limitant, par exemple, les plages de valeurs possibles. Les contraintes d'unicité, à l'instar des clés dans le modèle relationnel, définissent un sous-ensemble des attributs tels que les valeurs prises par ces attributs permettent d'identifier un quelconque objet du schéma.

*d) Opérations*

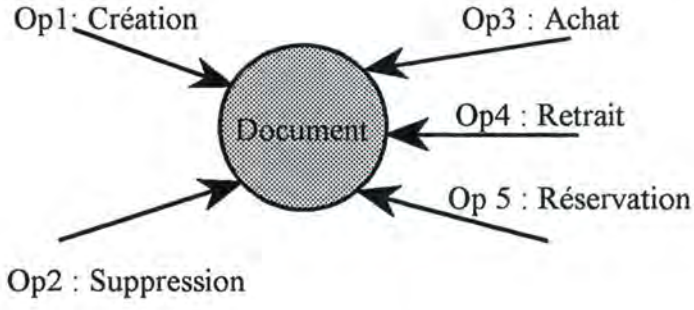



Figure 4.1 : Représentation graphique des opérations du schéma d'objet `Document`

Les opérations associées à un schéma d'objet permettent de créer les objets (instances) de ce schéma, d'en supprimer ou d'en modifier les valeurs d'attributs[4]. Chaque opération est définie sous forme textuelle par un nom et optionnellement [BRUNET a, 16] par un texte spécifiant sa

[4] Les papiers consultés ne précisent pas explicitement si les opérations sont aussi habilitées à modifier les références entre objets.

fonctionnalité et les règles de modification des attributs concernés. La figure 4.1 donne un exemple de représentation graphique des opérations d'un schéma d'objet.

*e) Evénements (internes)*

Le concept d'événement est essentiellement intégré au modèle O* pour représenter la cause du déclenchement des opérations sur les objets. Le modèle distingue trois catégories d'événements : les événements internes, externes et temporels, suivant qu'ils sont produits à l'initiative d'un objet, d'un acteur (cfr plus loin) ou d'une horloge.

La description d'un événement interne comporte un nom (facultatif), un prédicat et une partie déclenchement. Le prédicat consiste en une expression booléenne portant sur les valeurs d'attributs de l'objet avant et après l'occurrence de l'événement (par exemple : `old.quantité > stock_min et new.quantité ≤ stock_min`). La partie déclenchement (*trigger*) spécifie la ou les opération(s) déclenchée(s) ainsi que les objets concernés[5].

*f) Graphe des transitions d'état*

Au-delà des valeurs d'attributs, il est souvent intéressant de caractériser l'évolution d'un objet de façon plus globale en mettant en évidence et en nommant une série d'états (remarquables) par lesquels il peut transiter (par exemple, une facture peut être émise, payée, refusée,...). Il est évident que les transitions d'un état à un autre ne s'effectuent que dans des circonstances bien précises. Le graphe des transitions d'état décrit l'ensemble des états significatifs que peuvent prendre les objets du schéma ainsi que les transitions qui sont permises.

En fait, toute transition d'un état à un autre s'effectue au travers de l'exécution d'une opération de l'objet (elle-même déclenchée par l'occurrence d'un événement). Chaque transition (possible) du graphe est donc étiquetée par le nom de l'opération correspondante. Il faut noter que les états ne correspondent pas obligatoirement à des valeurs d'attributs des objets mais peuvent être obtenus à partir des événements subis par les objets [ROLLAND b, 312][6]. Dans ce cas, les états fournissent donc une représentation de "l'histoire" de l'objet.

Afin que les transitions représentent toujours le passage d'un état à un autre, un état "bidon", noté `So`, est systématiquement introduit dans le graphe pour représenter la non-existence de l'objet comme l'illustre la figure 4.2.

On notera aussi que le graphe des transitions d'état représente les états et transitions *possibles* pour un objet quelconque alors que le cycle de vie d'un

---

[5] [ROLLAND b] distingue de plus les événements locaux qui n'affectent qu'un seul schéma d'objet des événements partagés qui déclenche des opérations dans plusieurs schémas.

[6] Le graphe de transition d'état synthétise ainsi à la fois des aspects statiques de l'objet (les états) et dynamiques (les transitions entre états).

objet décrit les états et transitions effectivement *subis* par un objet particulier.

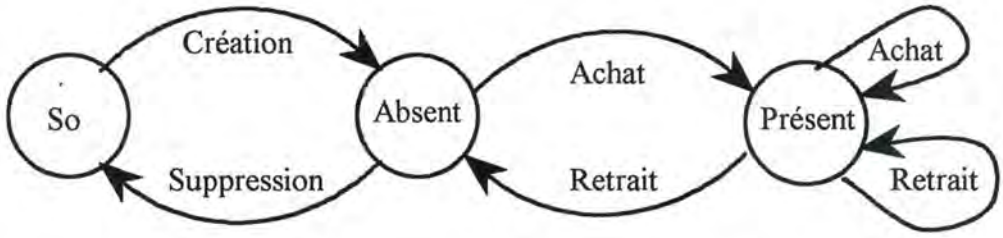



Figure 4.2 : Graphe des transitions d'état du schéma d'objet `Document`

*g) Hiérarchie d'héritage.*

L'héritage est défini entre deux schémas d'objets respectivement appelés schéma spécialisé et schéma généralisé de telle façon que l'on ait les deux inclusions suivantes :

- le schéma généralisé est inclus dans le schéma spécialisé :
  toutes les caractéristiques (attributs, références, contraintes, opérations et événements internes) définies au niveau du schéma généralisé sont également valides et utilisables au niveau du schéma spécialisé[7].
- l'extension du schéma spécialisé est incluse dans l'extension du schéma généralisé :
  pour tout objet spécialisé, il existe un objet généralisé ayant le même identificateur interne.

Trois types de contraintes peuvent être associées à une hiérarchie d'héritage :

- une contrainte de *disjonction* spécifie que les schémas spécialisés sont mutuellement exclusifs : un schéma d'objet généralisé ne peut être spécialisé qu'en au plus une forme spécialisée;
- une contrainte de *couverture* spécifie que tout schéma d'objet généralisé doit être spécialisé dans au moins une forme spécialisée;
- une contrainte de *partition* correspond à la conjonction des deux premiers types de contraintes : à tout schéma d'objet généralisé correspond un et un seul schéma d'objet spécialisé.

Ces contraintes d'héritages permettent par exemple de spécifier que tout objet `Document` est soit un `Livre`, soit un `Périodique` (contrainte de partition) ou encore qu'un objet `Personne` doit être un `Client` ou un `Fournisseur` ou les deux simultanément (contrainte de couverture).

Le modèle O* propose deux formalismes pour représenter les schémas d'objets à la fois sous forme textuelle et graphique. Illustrons ces formalismes en reprenant l'exemple des documents d'une bibliothèque qui a déjà été présenté au chapitre

---

[7] Cette inclusion implique que les opérations ne peuvent être spécialisées que par "augmentation" c'est-à-dire en ajoutant de nouvelles fonctionnalités mais sans en supprimer ou en redéfinir comme le permet la surcharge définie au paragraphe 3.3.2.

précédent. La figure 4.3 donne un extrait de la description textuelle tandis que la figure 4.4 nous donnent une idée du formalisme graphique proposé par O* pour la représentation structurelle .

```
objet DOCUMENT
    attributs
        Titre : STRING
        Année_publ : INTEGER
        Exemplaires : set of (E : EXEMPLAIRE)
    contrainte
        Année_publ ≤ Année_courante[8]
        partition (OUVRAGE, PERIODIQUE)
    opérations
        Création
        Suppression
        Achat (achat d'un nouvel exemplaire)
        Retrait (retrait d'un exemplaire perdu ou abîmé)
        Description (génère un texte descriptif)
        Réservation (inscription d'une nouvelle réservation)
    transitions-états
        (So, création, Absent)[9]
        (Absent, achat, Présent)
        (Présent, achat, Présent)
        (Présent, retrait, Présent)
        (Présent, retrait, Absent)
        (Absent, suppression, So)

objet OUVRAGE est-un DOCUMENT
    attributs
        No-isbn : STRING
    références
        Auteurs : set of (A : AUTEUR)
        Edité-par : EDITEUR
    opérations
        Description (génère un texte descriptif)

objet PERIODIQUE est-un DOCUMENT
    ...

objet EXEMPLAIRE
    ...
```

Figure 4.3 : Description textuelle de schémas d'objets en O*

[8] Année_courante est supposée contenir le millésime de l'année en cours

[9] Chaque triplet comporte le nom de l'état "de départ", le nom de l'opération qui permet la transition et le nom de l'état "d'arrivée".

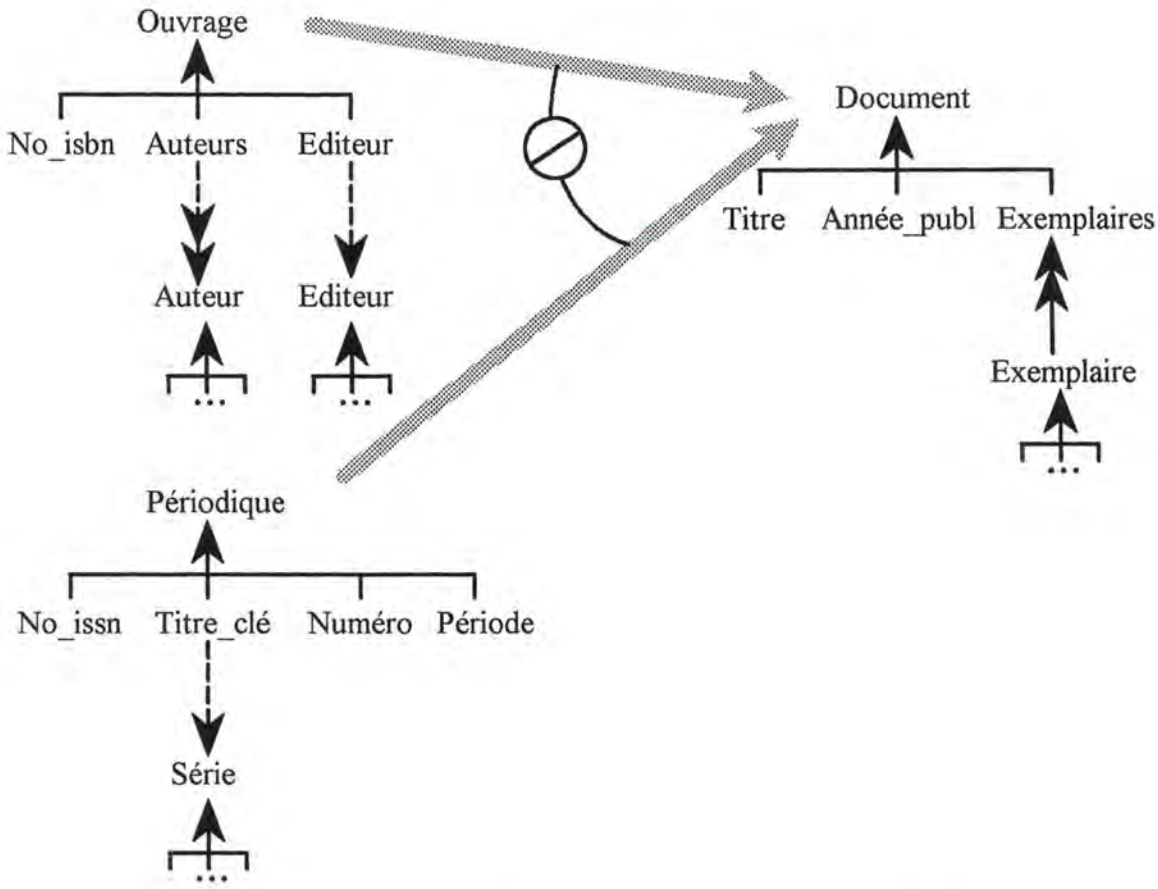



Figure 4.4 : Description graphique structurelle de schémas d'objets[10] en O*

### *Acteur*

Le concept d'acteur permet dans O* de représenter explicitement les êtres humains ou les dispositifs physiques qui ont l'aptitude et la compétence nécessaires pour interagir avec le système considéré. Plus précisément, un acteur peut être vu comme la projection d'une personne (d'un dispositif) assumant un certain *rôle* car, dans la pratique, il est possible qu'une même personne assure plusieurs rôles et ce, tour à tour ou même simultanément.

### *Schéma d'acteur*

Un schéma d'acteur permet de décrire les propriétés communes à une classe d'acteurs assumant le même rôle. Les notions de rôle et de schéma d'acteur sont donc équivalentes.

Les propriétés attachées à un schéma d'acteur sont uniquement dynamiques : ce sont des opérations et des événements.

[10] Pour ne pas alourdir le diagramme, nous n'avons pas repris explicitement la descriptions des schémas d'objet `Auteur`, `Editeur`, `Série` et `Exemplaire`.

*a) Opération*

Une opération sur un acteur modélise le fait que l'acteur reçoit effectivement un message en provenance du système. Le déclenchement par le système d'une opération sur un acteur permet donc de représenter les interactions du système vers son environnement.

*b) Evénement (externe)*

Un événement (qualifié d'externe du point de vue du système) modélise le fait qu'un acteur peut être la cause du déclenchement d'une ou plusieurs opération(s) sur des objets du système. Les événements externes permettent donc de représenter les interactions de l'environnement à destination du système.

*c) Hiérarchie d'héritage*

Les schémas d'auteurs peuvent également être structurés par une relation d'héritage tout à fait semblable à celle qui a été définie sur les schémas d'objets.

Un schéma d'acteur peut être représenté dans O* par une description textuelle (figure 4.5) ou graphique (4.6).

```
acteur BIBLIOTHECAIRE
    opérations
        expédition lettres de rappel
    événements
        catalogage document
            déclenche création sur DOCUMENT
        achat exemplaire
            déclenche achat sur DOCUMENT
            déclenche création sur EXEMPLAIRE
        (...)

acteur LECTEUR
    événements
        demande de réservation
            déclenche réservation sur DOCUMENT
        (...)
```

Figure 4.5 : Exemple de description textuelle de schémas d'acteurs

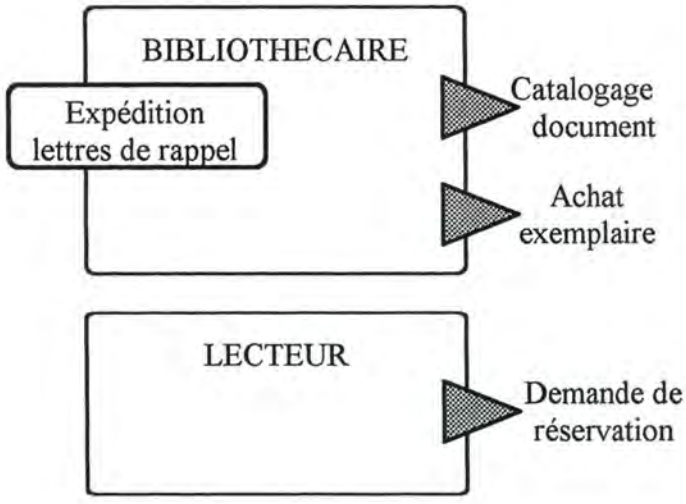



Figure 4.6 : Exemple de description graphique de schémas d'acteurs

Le modèle O* propose de plus de représenter sous forme graphique les liens dynamiques de déclenchement entre les événements (internes, externes ou horloges) et les opérations (sur les objets ou les acteurs). La figure 4.7 donne un exemple d'un tel diagramme.

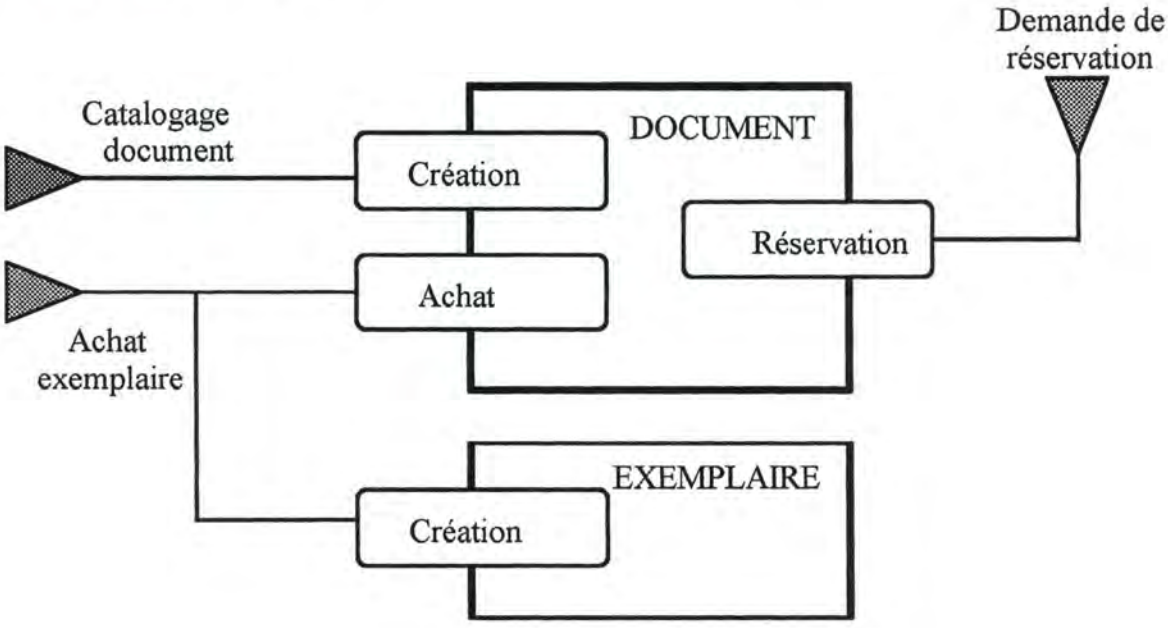



Figure 4.7 : Exemple de schéma de la dynamique

### 4.1.3 Méthode de conception "en fontaine"

La méthode de conception associée au modèle O*, appelée méthode "en fontaine", propose de travailler à la construction du schéma conceptuel de façon progressive et systématique en structurant les activités de conception en 6 étapes successives.

L'originalité de la démarche réside dans le fait qu'elle autorise le passage d'une étape à la suivante sans avoir nécessairement terminé les étapes précédentes. Au contraire, il est prévu d'effectuer régulièrement des retours sur les étapes antérieures (éventuellement jusqu'à la première) pour compléter ou raffiner la description des objets. Cette démarche est illustrée par la figure 4.8 que nous empruntons à [ROLLAND a]. La démarche proposée dans [BRUNET a] est fort semblable même, si le nombre d'étapes est réduit à cinq (les étapes 2 et 3 de [ROLLAND a] sont rassemblée en une seule) et si Joël Brunet parle plutôt de modèle "en spirale" pour insister sur les nécessaires itérations du processus de conception. Voyons à présent quelles sont ces étapes et quelles sont les tâches associées.

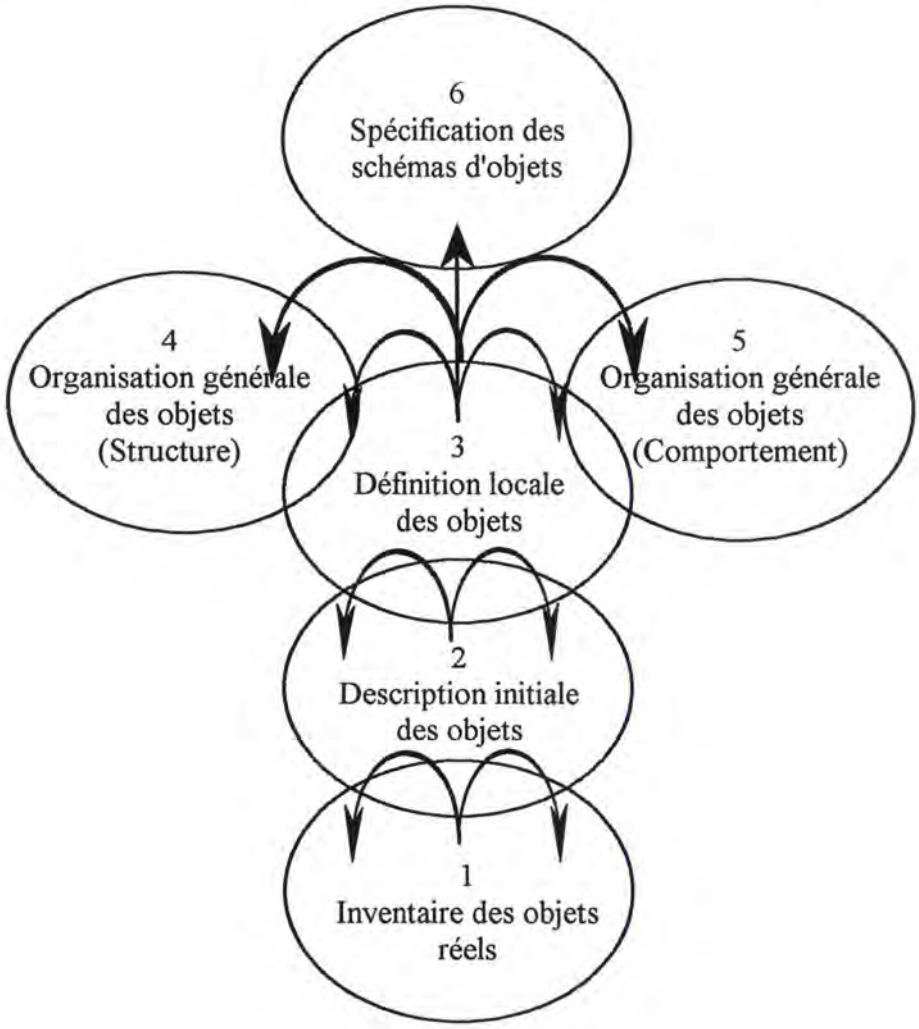



Figure 4.8 : Vue d'ensemble de la méthode "en fontaine"

### *1) Inventaire des objets du monde réel*

Il s'agit, dans une première phase, d'identifier intuitivement les entités, les acteurs et les événements du monde réel que l'on considère comme pertinents dans le domaine considéré. Entités, acteurs et événements peuvent se compléter mutuellement pour faire découvrir de nouveaux objets. Ainsi par exemple, Rolland et Cauvet signalent que "l'identification des acteurs permet de

comprendre un système en terme de «qui (un acteur) agit (un événement) sur quoi (une entité) ?»".

Dans une seconde phase, il convient de classifier les entités et les acteurs pour faire émerger les schémas d'objets et les schémas d'acteurs qui seront retenus. Il faut particulièrement être attentif à éviter de multiplier inutilement le nombre de schémas. Les auteurs conseillent, à ce propos, de regrouper les schémas pour lesquels les cycles de vie sont identiques.

Joël Brunet signale de plus qu'il est judicieux, dès cette étape, de rechercher si certains objets n'ont pas déjà été spécifiés lors d'autres analyses ayant trait à des domaines similaires. La réutilisation complète ou même partielle de spécifications d'objets permettra en effet de faire non seulement des économies de temps au niveau de l'analyse conceptuelle, mais aussi de préparer la réutilisation effective de classes déjà définies au niveau de l'implémentation.

*2) Description initiale des objets*[11]

Cette seconde étape doit permettre d'obtenir une description essentiellement statique des schémas d'objets en mettant en évidence les attributs et les états remarquables qui sont jugés pertinents et utiles à la gestion des objets. Au terme de cette étape, chaque schéma d'objet aura reçu un nom, une liste d'attributs et une liste d'états.

*3) Analyse et définition locale des objets*

La troisième étape consiste d'abord à définir plus précisément la structure du type d'objet en distinguant notamment les attributs qui sont de véritables composants de l'objet (relation de composition) de ceux qui ne sont pas en fait que des références à d'autres objets. De plus, cette étape doit permettre de décrire le comportement des objets en définissant complètement les opérations et les événements du schéma d'objet.

Rolland et Cauvet proposent de commencer par construire le graphe des transitions d'état de façon à mettre en évidence les événements (et les opérations) associées à chaque transition. Les événements rencontrés peuvent bien entendu être internes, externes ou temporels. Ils seront décrits avec mention de leur type et si possible de leur condition d'occurrence ainsi que des opérations déclenchées. Une dernière phase de cette étape consiste à décrire les opérations offertes ou utilisées par les objets. Il s'agit généralement d'une opération de création, d'une opération de destruction et d'une ou plusieurs opérations de mise à jour des attributs. Il faut noter que, si l'on relève souvent un "couplage" évident entre événements et opérations, il est des opérations déclenchées par plusieurs événements distincts et, inversement, on trouve des événements qui déclenchent plusieurs opérations.

---

[11] Colette Rolland parle presque indifféremment d'objet, de type d'objet et de schéma d'objet tout au long de ses articles [ROLLAND a&b]. Le terme objet semble désigner plus particulièrement la ou les entité(s) du monde réel tandis que le type d'objet désigne l'abstraction que l'on fait de ces entités et le schéma d'objet correspondant à la description que l'on en donne.

*4) Organisation globale des objets : structure*

Les différents schémas d'objets, considérés isolément jusqu'ici, sont à présent confrontés et assemblés en une structure globale. Cette étape conduit généralement à reconsidérer la structure de certains schémas d'objets (liens de composition / de référence) et à regrouper plusieurs schémas au sein d'une hiérarchie d'héritage.

*5) Organisation globale des objets : comportement*

Comme dans l'étape 4, il s'agit de compléter et d'enrichir les descriptions des types d'objets par confrontation des schémas déjà décrits et ce dans trois orientations distinctes. Premièrement, il y a lieu de repérer les événements qui sont partagés par plusieurs objets et par conséquent les points de synchronisation dans leurs cycles de vie. Ensuite, il faut compléter le relevé des événements et des opérations en partant du principe qui veut, dans O*, que toute opération soit déclenchée par un événement. Certaines opérations permettent ainsi de découvrir l'événement associé et vice versa. Enfin, il convient de terminer la description des événements en spécifiant très précisément leurs conditions d'occurrence.

*6) Spécification des schémas d'objet et validation*

Cette étape est essentiellement caractérisée par l'effectuation d'une série de contrôles en vue d'assurer la correction et la validation de l'ensemble des schémas. Quatre directions sont proposées pour vérifier :

- la conformité des schémas aux concepts du modèle O*;
- la cohérence des schémas (pas d'ambiguïté, ni de contradiction contenue ou déductible dans les schémas);
- la complétude : tous les éléments référencés sont complètement décrits;
- la fidélité : l'ensemble des schémas donne un reflet fidèle de la réalité et est adéquat aux besoins de gestion.

### 4.1.4 Remarques et commentaires

Au terme de cette rapide exploration du modèle O* et de la démarche de conception qui lui est associée, nous allons nous permettre d'émettre quelques commentaires afin de mettre en lumière certains points forts ou points faibles.

Tout d'abord, il apparaît évident que le couplage entre le modèle et la méthode n'est pas très important : on sent bien que la méthode pourrait s'appliquer avec des modifications mineures à un autre modèle pour autant que celui-ci comprenne les concepts d'objet, de classe et d'événement. Le concept d'acteur semble d'ailleurs peu utilisé (sauf à l'étape initiale). Les concepts de cycle de vie et de graphe des transitions d'état apparaissent par contre comme des leviers importants de la méthode en offrant un outil original pour analyser le comportement des objets.

Une seconde remarque doit être faite au sujet du domaine d'application de la méthodologie. O* trouve son terrain de prédilection dans l'analyse de systèmes

d'informations et des applications de gestion associées. La mise en pratique de ce modèle dans des domaines d'application bien différents (par ex. : la conception d'interfaces graphiques) ne paraît pas évidente quand ce ne serait que par la définition initiale du concept d'objet qui se limite aux entités persistantes du monde réel [BRUNET a, 3].

La richesse et la variété des représentations graphiques, si elles permettent évidement de visualiser et de synthétiser de nombreux concepts, nous paraît un peu lourde et parfois inconsistante. Certains concepts sont ainsi tour à tour représentés sous différentes formes (un objet est une sorte d'arbre dans le schéma structurel, il est un cercle dans le schéma des opérations et devient un rectangle dans le schéma de la dynamique; le cercle représente tantôt un objet, tantôt un état possible de cet objet...). Cette très large utilisation des représentations graphiques induit presque automatiquement la nécessité de disposer d'un environnement de travail offrant un support à la méthode. Il n'est pas fait mention d'un tel outil dans les papiers consultés.

Enfin, le résultat obtenu au terme de l'analyse est un modèle conceptuel du système d'informations et de l'application associée qui répond, en fait, plus à la question "quoi faire (faire) ?" qu'à la question "comment faire (faire)?". En effet, O* n'est pas très exigeant sur la définition des opérations et n'offre aucun support à leur analyse détaillée. La poursuite du développement jusqu'à l'implémentation et la validation du système doivent donc s'effectuer en marge de O*.

## 4.2 OBLOG

OBLOG (*OBject oriented LOGic*) est un langage de conception orienté objet développé depuis 1987 par Amilcar Sernadas et son équipe (INESC, Lisbonne). Ce langage, purement textuel au départ mais supportant une notation graphique spécifique depuis 1989, est aussi le langage de base pour un environnement d'assistance au développement (dit *CASE workbench*) dont la réalisation a commencé en 1990. Une première version commerciale de l'outil CASE devrait être mise sur le marché dans les prochains mois. Cette version supportera un sous-ensemble du langage OBLOG appelé OBLOGlight.

Compte tenu de l'objectif de simplification que nous nous sommes fixés pour le présent travail, il nous a paru intéressant de nous intéresser spécialement à OBLOGlight (décrit dans les documents [SERNADAS c] et [SERNADAS d]) en ne faisant référence à OBLOG lui-même [SERNADAS a] qu'à titre de complément pour certains éléments et spécialement pour toutes les notations graphiques[12].

---

[12] Tous les documents consultés étant rédigés en langue anglaise, nous nous somme permis de choisir un terme français pour traduire chacun des concepts évoqués. Lors de la première évocation du terme, nous donnerons, entre parenthèses le vocable original correspondant.

### 4.2.1 Objectifs

OBLOG a l'ambition d'offrir un langage orienté objet adapté à de très grands projets et capable de soutenir le processus de développement depuis la phase d'analyse conceptuelle jusqu'à celle de l'implémentation même si cette dernière phase doit s'effectuer avec des "outils" non orientés objet tels que le langage C ou les bases de données relationnelles (SQL). L'outil de support doit comprendre un générateur de code qui soit capable de transformer automatiquement une spécification objet en un texte de programme directement compilable.

### 4.2.2 Le modèle OBLOG

Le langage OBLOG (plus précisément OBLOGlight) permet de structurer le système informatique en objets regroupés dans des classes d'objets et caractérisés principalement par des attributs, des actions, des règles de comportement et des règles d'affectations de valeur aux attributs. Etudions pas à pas chacun de ces concepts.

***Objets***

Pour OBLOG, un objet (*object*) est une unité significative d'informations qui comprend à la fois des données et des règles comportementales. L'objet lui-même est seul à être habilité à modifier les données qu'il renferme et cela, soit de sa propre initiative, soit à la demande d'un autre objet du système.

Un objet peut être plus ou moins persistant et être actif ou passif. Un objet est actif s'il est capable d'initiative, c'est-à-dire si certains de ses comportements peuvent être déclenchés sans intervention de requête extérieure mais simplement par observation de l'état de l'objet. Un objet passif n'agit par contre que sur base de demandes qui lui sont adressées par d'autres objets du système.

A chaque objet peut être associé un "patron" ou un "modèle" (*template*) qui décrit exactement toutes les propriétés statiques et dynamiques de l'objet. Les objets qui partagent un même patron (ou au moins une même portion de patron) sont regroupés et décrits dans une même "classe". Tous les objets gérés par OBLOG sont nécessairement instances d'au moins une classe et ce, même si la sémantique impose qu'il n'existe qu'une seule instance pour cette classe.

***Classes***

Une classe (*class*) est décrite en OBLOG par un nom, par la spécification des caractéristiques partagées par toutes les instances, mais aussi, et c'est plus original, par un mécanisme d'identification qui doit permettre d'identifier les instances (potentielles) de la classe. A l'instar du modèle relationnel, ce mécanisme est constitué par la spécification d'un groupe-clé (*key group*) d'attributs dont les valeurs permettent d'identifier n'importe quelle instance de la classe. La définition d'un groupe-clé, qui peut bien entendu ne comprendre qu'un seul attribut, est toutefois optionnelle (mais recommandée) car OBLOG attribue d'office un identificateur système à tout objet. Le fait de disposer d'un mécanisme d'identification par clé permet de désigner un objet sans disposer d'un pointeur explicite (identifiant système) sur cet objet.

La déclaration d'une nouvelle classe est automatiquement couplée à la définition d'un nouveau type de donnée (*data type*) qui porte le même nom et qui est lui-même un héritier du type générique "OBJECT". Une classe peut ainsi être vue comme l'ensemble des objets qui relèvent du même type.

OBLOGlight admet l'héritage simple mais préfère parler de spécialisation de classes. Les objets spécialisés sont appelés des aspects[13] (*aspect*) de la classe ancêtre. Un aspect hérite de toutes les caractéristiques définies pour son ou ses ancêtre(s). Il doit préserver l'interface exportée par ses ancêtres (c'est-à-dire la liste des actions activables de l'extérieur) mais peut éventuellement modifier l'implémentation des éléments hérités en raffinant les comportements associés aux actions héritées. L'interface peut évidement être étendue en déclarant de nouveaux attributs et/ou actions. Les clés héritées doivent être maintenues avec possibilité d'en définir de nouvelles, spécifiques à l'aspect considéré.

Si l'on considère le type de donnée associé à la classe, on peut donc affirmer qu'un sous-type (*sub-type*) associé à un aspect est un type "compatible" avec tous ses super-types (*super-types*) associés aux classes dont il hérite directement ou indirectement et ce, grâce à la préservation de l'interface exportée des ancêtres.

Une classe est représentée graphiquement par un (grand) cercle contenant le nom de la classe écrit en lettres majuscules. Les caractéristiques associées à une classe sont essentiellement ses attributs qui décrivent l'état de ses instances et les actions qui permettent de modifier cet état. Les classes qui ne sont destinées qu'à être instanciée une seule fois sont repérées par un "1" sous leur nom.

***Attributs***

Les attributs (*object attribute*) constituent les éléments statiques de description des objets d'une classe. L'ensemble des valeurs, à un instant donné, des attributs d'un objet déterminent l'état de cet objet. Chaque attribut est défini par un nom, par un type de "codomaine" (*codomain data type*) et éventuellement par une contrainte de valeur.

Le codomaine détermine le type des valeurs qui sont acceptables pour l'attribut. Le codomaine peut être aussi bien l'un des nombreux types simples acceptés par OBLOG (*integer, natural, string, boolean, date, ...*) qu'un type défini par l'utilisateur (à l'aide de constructeurs tels que *enumeration, range, list, array, ...*) ou encore un type objet défini via une autre classe. Cette dernière possibilité permet de créer des objets complexes constitués par l'agrégation de plusieurs objets plus simples appelés composants de l'objet complexe. Les objets composants peuvent toutefois faire partie, simultanément; de plusieurs agrégats.

Une contrainte limitant les valeurs acceptables pour l'attribut peut être annexée à chaque attribut. Cette contrainte peut être statique (restriction pure et simple du codomaine) ou dynamique (relative aux valeurs d'autres attributs de la même classe).

---

[13] Ce terme n'est cependant évoqué que dans [SERNADAS c]

Les attributs d'une classe peuvent, de plus, être caractérisés selon quatre critères :

- Variable ou constant selon que la valeur peut être modifiée ou non pendant la vie de l'objet ;
- Partiel ou total selon que l'on admet ou non l'absence de valeur (un attribut total peut aussi être qualifié de *not null* [SERNADAS c, 8];
- Public ou privé selon que l'attribut est consultable ou non par d'autres objets ;
- Interface ou normal selon que la valeur peut être, ou non, directement modifiée par l'utilisateur sans faire appel à aucune action.

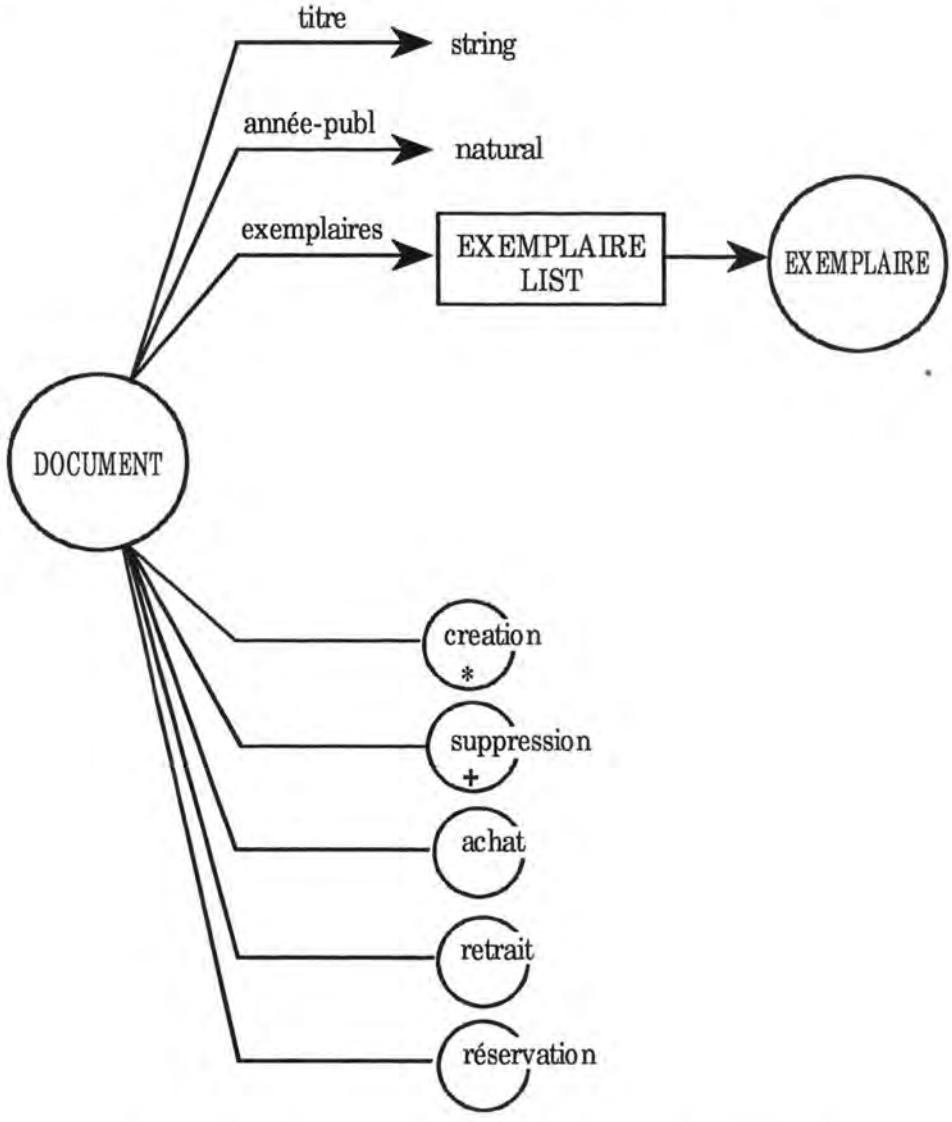



Figure 4.9 : Diagramme OBLOG de la classe Document

Les attributs participant à un groupe-clé doivent nécessairement être totaux et publics (mais pas nécessairement constants !).

Enfin, OBLOG admet la définition d'attributs dérivés (*derived attribute*) dont la valeur est déduite d'autres attributs de la même classe ou même d'autres classes (en faisant appel à des "observations"). Ces attributs dérivés peuvent faciliter la

gestion mais introduisent une certaine forme de redondance dans la population d'objets.

Comme illustré dans la figure 4.9 (moitié supérieure), les attributs sont représentés graphiquement par des flèches pointant vers leur codomaine. Le nom de l'attribut est noté à proximité du trait de la flèche de même que l'éventuelle contrainte de valeur.

***Actions***

Les opérations qui modifient l'état des objets sont appelées actions (*actions*) par OBLOG. L'occurrence d'une action est par définition atomique et de durée nulle[14]. Ce sont les actions d'une classe et elles seules qui sont habilitées à modifier les valeurs d'attributs des objets de la classe.

Une action est spécifiée par un nom, elle appartient à l'une des trois catégories suivantes :

- action de naissance (*birth action*) qui crée une nouvelle instance de la classe et affecte une valeur initiale aux attributs non nuls;
- action de mort (*death action*) qui détruit l'instance désignée de la classe ;
- action de mise à jour (*update action*) qui modifie l'état de l'objet désigné suivant les indications d'une "évaluation d'attribut" (*object attribute valuation*).

Le nom d'une action de naissance est précédé du symbole "*" et celui d'une action de mort est préfixé par "+". Une action peut aussi être douée d'initiative si elle peut se déclencher d'elle-même c'est-à-dire sans intervention des autres objets du système. Une action douée d'initiative est repérée par un point d'exclamation "!" en préfixe de son nom. Une action peut aussi être définie comme une variante d'une autre action déjà décrite : il s'agit alors d'une action dérivée (*derived action*).

Enfin, il est souvent utile de compléter la spécification d'une action par une liste d'attributs d'action (*action attributes*) qui décrivent en fait les arguments qui doivent être passés à l'action pour qu'elle puisse avoir lieu (par exemple, le `montant` à retirer pour une action `retrait` à déclencher sur une instance donnée de la classe `compte_bancaire`). Par analogie avec la définition des attributs de classe, OBLOG distingue les attributs d'action publics qui peuvent recevoir une valeur de l'extérieur, des attributs d'actions privés qui ne peuvent être évalués que par l'action elle-même. Ces deux catégories correspondent simplement, si l'on voit l'action comme une procédure, à des paramètres permettant respectivement de fournir de l'information à la procédure et de retourner des informations résultat de la procédure.

---

[14] Jusqu'au début de cette année 1993, le concept d'action portait le nom d'événement mettant ainsi l'accent sur le caractère atomique (non décomposable) et ponctuel dans le temps des actions. Cette terminologie introduisait cependant une certaine ambiguïté, surtout pour le néophyte, en masquant le fait que tout "événement" a un effet modificateur sur l'état de l'objet.

Chaque attribut d'action est caractérisé par un nom, un codomaine (un type) et sa catégorie publique ou privée. Graphiquement, une action est représentée par un petit cercle contenant le nom de l'action et relié à sa classe par un trait.

***Observations***

Une observation (*observation*) est une fonction définie sur une classe et qui fournit des informations sur l'état des objets de cette classe ou de classes associées. Une observation peut accéder directement (en lecture uniquement) aux attributs des objets de sa classe et faire appel à des observations définies sur d'autres classes pour construire l'information qu'elle fournira en retour.

***Requêtes***

Le déclenchement d'une action ou d'une observation sur un objet déterminé nécessite de pouvoir désigner explicitement l'objet qui est considéré. Comme dans tous les langages orientés objet, cette désignation peut s'effectuer en utilisant l'identifiant système qui pointe sur l'objet. Toutefois, cette méthode de désignation n'est pas toujours pratique car il arrive que l'on ne dispose pas, à priori, de pointeur sur l'objet que l'on veut manipuler ou interroger. OBLOG offre le mécanisme des requêtes (*queries*) pour isoler, parmi toutes les instances d'une classe, celle que l'on recherche. Il suffit, à la manière d'une sélection dans une base de données relationnelle, de fournir une condition portant sur les valeurs d'attribut des objets de la classe. Les requêtes peuvent également être utilisées pour sélectionner un sous-ensemble des instances d'une classe ou encore pour tester l'existence d'une instance déterminée.

***Comportement des objets***

La vie d'un objet (appartenant à une classe donnée) commence avec l'occurrence d'une action de naissance, se poursuit par un certain nombre d'actions de mise à jour et se termine éventuellement par une action de mort en transitant ainsi par un certain nombre de situations (*situations*) remarquables.

La spécification du comportement d'une classe d'objets dresse la liste des règles qui régissent les transitions (*transitions*) entre les diverses situations possibles. Chaque transition admissible est ainsi décrite par les situations de départ et d'arrivée, par une action et par une éventuelle condition à satisfaire avant d'effectuer la transition. Eventuellement, l'action peut être remplacée par un motif (*pattern*) que l'on peut voir comme une macro-action elle-même constituée de plusieurs actions (élémentaires).

Le comportement (potentiel) d'une classe d'objets (*object behavior*) peut être représenté graphiquement par un diagramme de comportement où les actions sont représentées par de petits cercles et les transitions par des barres horizontales. Les arcs joignant les cercles aux barres peuvent être surchargés par l'énoncé de la condition à satisfaire. Un exemple de diagramme de comportement est illustré à la figure 4.10.

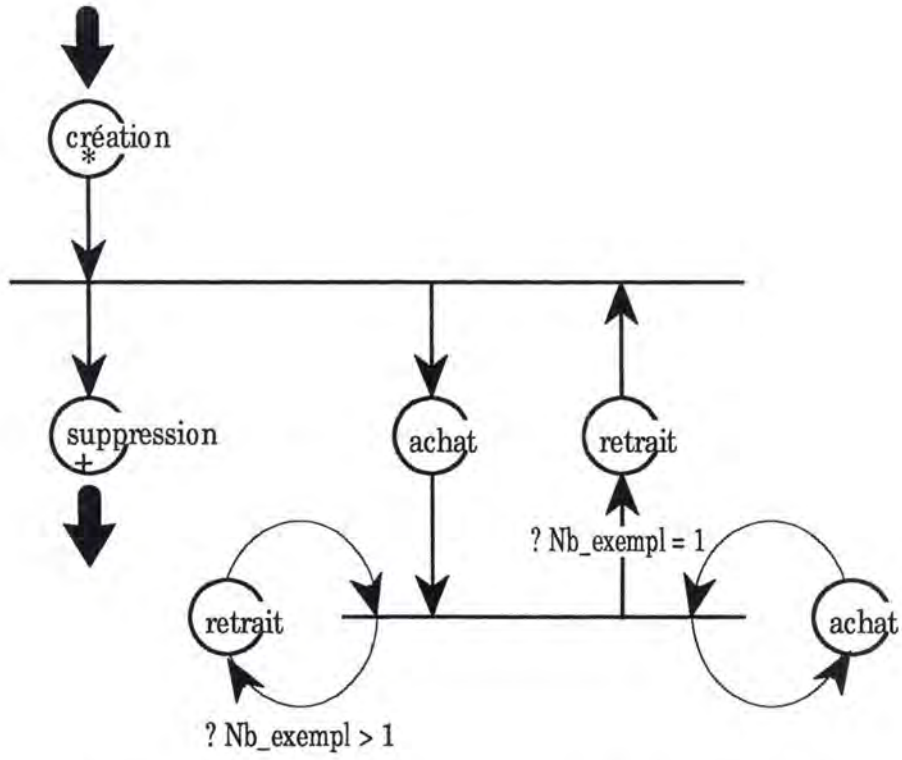



Figure 4.10 : Diagramme de comportement de la classe DOCUMENT.

***Evaluation des attributs***

Toutes les règles d'affectation de valeurs, tant aux attributs des objets, qu'à ceux des actions (arguments) sont spécifiées via un mécanisme unique appelé "évaluation d'attribut" (*object/action attribut valuation*). Chaque évaluation est associée à une action précise et décrit la procédure de calcul pour affecter la valeur voulue aux attributs concernés. Les expressions peuvent faire appel à des constantes, aux valeurs des attributs de l'objet, aux attributs de l'action ou à des observations sur d'autres objets. Toutes les opérations compatibles avec les types des valeurs peuvent être employées et des expressions conditionnelles peuvent être introduites.

***Interaction entre objets***

Les objets peuvent évidemment interagir entre eux. Ils le font grâce au mécanisme d'appel d'action (*action calling*). Ainsi, par exemple, l'action `retrait` d'un objet `Document` peut appeler l'action `suppression` de l'objet `Exemplaire` qui doit être retiré. Le mécanisme d'appel d'action permet ainsi de synchroniser des événements (actions) qui doivent se produire simultanément sur des objets distincts. L'appel d'une action peut bien entendu s'accompagner de transfert d'informations de l'appelant vers l'appelé grâce aux attributs des actions.

***Autres concepts***

Outre les concepts fondamentaux qui viennent d'être rapidement présentés, OBLOG introduit une série d'autres notions qui permettent de raffiner encore la description du système. Ainsi, par exemple, il est également possible de définir des contraintes d'intégrités plus globales portant sur plusieurs objets, voire sur

plusieurs classes. De même, la notion d'action peut elle-même être abstraite pour donner naissance à un véritable objet (appelé *expanded action*) permettant par exemple de représenter plus commodément des transactions entre objets.

### 4.2.3 Méthode de conception

Comme nous l'avons déjà souligné, OBLOG est avant tout un langage d'expression pour la modélisation orientée objet d'un système informatique en cours de développement. Il n'y a pas, à ce jour, de véritable méthode de conception qui soit directement associée au modèle OBLOG. Toutefois, on peut trouver quelques éléments méthodologiques disséminés dans le document [SERNADAS a]. Nous allons nous efforcer de les rassembler dans ce paragraphe.

Le développement d'un logiciel peut être décomposé, selon l'équipe d'Amilcar Sernadas, en 4 phases distinctes (dont on ne dit pas si elles sont strictement successives !). Ce sont :

1. Perception

   Cette phase permet la représentation de l'"univers du discours" par l'identification et la description des objets jugés pertinents du monde réel, qu'ils soient actifs ou passifs. La description des objets doit envisager à la fois les aspects statique et dynamique.

2. Conception

   La phase de conception a pour objectif la description précise du système informatique en cours de développement en incluant la spécification des objets informationnels qui seront utilisés. Il s'agit à la fois d'objets passifs (messages et enregistrements) qui seront traités, mémorisés et échangés dans le système ainsi que des objets actifs (processus) qui permettront ces manipulations.

3. Implémentation logique

   Il s'agit cette fois de la construction de la solution effective en utilisant le / les langage(s) de programmation / de description des bases de données qui sont prévus pour l'implémentation. Aucune optimisation n'est prévue à ce stade.

4. Implémentation physique

   Cette dernière phase permet de raffiner l'implémentation en prenant les dispositions nécessaires à l'optimisation des processus les plus critiques.

OBLOG se propose d'offrir un support à chacune de ces phases. Le support consiste à fournir, pour les deux premières phases, un environnement de travail qui permette de décrire, d'enregistrer et de structurer les classes nécessaires à la modélisation du problème d'abord, de sa solution ensuite. Les deux dernières phases sont ensuite entièrement prises en charge par le *workbench* qui génère le code sur base de la description conceptuelle de la solution.

Pour les phases de perception et de conception, les concepteurs d'OBLOG proposent de procéder à la description des objets retenus comme pertinents en passant par trois étapes :

a) Définition des mécanismes d'identification des instances

Cette étape doit donc mettre en évidence les classes retenues ainsi que les relations entre celles-ci (héritage, composition, référence, ...). Les attributs permettant l'identification des objets sont également précisés[15].

b) Définition du "patron" des instances

Il s'agit cette fois de décrire plus précisément chaque classe en énumérant d'abord les attributs et les actions associés, en précisant ensuite le comportement des objets (diagramme des situations et transitions) et enfin en spécifiant exactement les effets de chaque action sur les attributs.

c) Description des interactions entre objets

Cette dernière étape permet de définir le comportement global des objets qui interagissent entre eux notamment par déclenchement d'actions (*action calling*). Cette étape peut remettre en cause ou raffiner la structuration des classes d'objets.

### 4.2.4 Remarques et commentaires

Le modèle OBLOG se distingue par la simplicité des concepts de base qui sont mis en oeuvre (objet, classe, attribut et action) et qui sont agencés entre eux de manière fort ingénieuse afin de donner une grande expressivité au modèle. La sémantique très précise des différents concepts permet à la fois une modélisation minutieuse de la solution conçue et surtout, autorise sa traduction automatique en un code source directement compilable.

On doit cependant regretter que la méthode associée au modèle OBLOG est fort peu explicitée. Quelques pistes sont seulement esquissées dans les documents consultés. La démarche de développement relève en fait de l'approche transformationnelle en proposant de construire pas à pas le logiciel par transformation successive des solutions intermédiaires (spécification, conception, implémentation). Ces différentes phases rappellent d'ailleurs de façon étonnante les étapes classiques de la conception de bases de données (schéma conceptuel ≈ modélisation de l'univers du discours, schéma logique de BD ≈ spécification des objets informationnels, schéma physique BD ≈ programme implémenté).

[15] Cette étape doit probablement être rapprochée de celle qui permet, dans une modélisation de type "Entité-Association", le relevé des types d'entités et des types d'associations et leur structuration.

## 4.3 HOOD

HOOD (*Hierarchical Object Oriented Design*) est une méthode de conception qui a vu le jour en 1987 à l'initiative de l'Agence Spatiale Européenne (ESA) mais qui a été mise au point par un consortium formé par les sociétés CISI Ingénierie, MATRA Espace et CRI. C'est au travers de l'ouvrage [LAI] de Michel Lai que nous allons découvrir les traits essentiels de la version 3.1 de la méthode HOOD.

### 4.3.1 Objectifs

La méthode HOOD est destinée à supporter la conception de très grands projets spécialement dans le domaine de l'aéronautique spatiale, avec Ada comme langage cible préférentiel (mais non obligatoire). HOOD, à l'instar du langage Ada est aussi remarquable par sa capacité à gérer la concurrence entre processus. Cette méthode est donc tout à fait adaptée au développement de logiciels industriels en temps réel ou de logiciels répartis sur réseau. Une autre caractéristique essentielle de la méthode HOOD est son approche hiérarchique de la conception qui permet au travers d'une démarche descendante de raffiner progressivement les objets qui sont mis en évidence.

### 4.3.2 Le modèle HOOD

Contrairement à la plupart des modèles orientés objets, le concept principal du modèle HOOD n'est pas la classe, mais l'objet. De plus, la notion d'héritage n'est pas présente dans ce modèle.

***Objets***

Le manuel de référence de HOOD, cité par [LAI, 194] nous apprend que "*An object is a model of a real world entry, which combines both data and operations on that data*". Il s'agit là d'une définition tout à fait traditionnelle d'un objet. Plus formellement, un objet est un triplet (N, Ty, Op) où N est un nom (unique dans l'univers considéré), Ty est un ensemble de types de données[16] et Op est un ensemble non vide d'opérations (procédures ou fonctions).

Dans le jargon de HOOD, on dit que les types appartenant à Ty et opérations appartenant à Op sont "produits" (*provided*) par l'objet N et qu'ils constituent l'interface de l'objet (partie *provided*). Une seconde partie de l'objet, dite partie *internal* contient la description algorithmique de l'objet, c'est-à-dire son implémentation. En pratique, toutefois, on constate que HOOD met essentiellement l'accent sur les opérations produites par les objets au détriment des données associées aux objets à tel point que Michel Lai peut résumer sa définition d'un objet HOOD en affirmant qu'il est défini par un nom, un ensemble

[16] Les types d'un objet HOOD correspondent à peu près à ce que nous appelons plus couramment les attributs d'un objet. M. Lai utilise d'ailleurs ce terme d'attribut dans un exemple illustratif [LAI, 113].

d'opérations et deux arborescences d'objets et d'opérations dont nous allons parler un peu plus loin [LAI, 50]. HOOD admet d'ailleurs l'existence d'objets triviaux dont l'ensemble des types est vide et l'ensemble des opérations est un singleton : une opération peut donc être un objet à elle seule.

Dans l'optique de la modélisation objet de processus concurrents et des applications "temps réel", HOOD distingue plusieurs types d'objets[17] dont les plus importants sont les objets actifs et les objets passifs. Un objet est dit "actif" s'il comporte au moins une opération "contrainte", c'est-à-dire dont l'exécution est sujette à une contrainte de synchronisation avec d'autres opérations. Un objet passif ne comporte, par contre, que des opérations non-contraintes. Le transfert du contrôle entre objet appellant et appelé se fait alors de façon purement séquentielle.

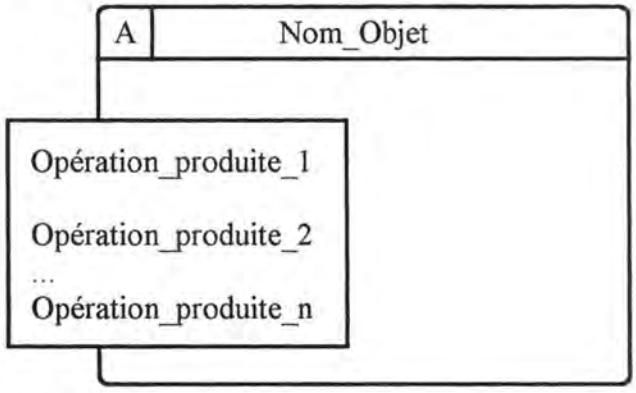



Figure 4.11 : Diagramme d'un objet HOOD

Un objet HOOD est spécifié par une description textuelle reprenant toutes ses caractéristiques en un texte indenté et structuré par des mots clés spécifiques à HOOD (cf. syntaxe partielle dans [LAI, 77-80]). Une représentation graphique des objets est également prévue. Comme le montre la figure 4.11, un objet est représenté par une rectangle à coins arrondis portant dans la partie supérieure le nom de l'objet (ainsi que la lettre A si l'objet est actif). Un second rectangle à angles vifs surcharge le bord gauche de l'objet et contient la liste des noms des opérations produites tandis que le reste de la surface de l'objet peu être utilisé pour décrire les objets internes (Cf. ci-dessous). Il est à noter que la représentation graphique ne fait pas mention des types de données produits par l'objet.

***Relations entre objets***

Les objets HOOD sont structurés par deux relations très importantes : la relation d'inclusion et la relation d'utilisation.

La relation d'inclusion (*include*) est la relation sur laquelle est basée le caractère hiérarchique de la conception. Lors d'une phase quelconque de la conception, un

[17] Le mot "catégorie d'objet" eut été plus judicieux pour ne pas entraîner de confusion avec les types (attributs) produits par l'objet.

objet (dit objet père) est ainsi décomposé en des objets fils qui, ensemble, produisent la même fonctionnalité que l'objet père[18]. A chaque opération produite par l'objet père correspond donc une opération (et une seule !) produite par l'un de ses objets fils. Par souci de simplification, l'opération garde généralement le même nom. Les objets fils peuvent toutefois "produire" d'autres opérations qui seront par exemple utilisées par les autres objets fils (dit objets frères). Un objet est dit "terminal" lorsque toutes ses opérations sont elles-mêmes terminales, c'est-à-dire ne font pas appel à des opérations produites par d'autres objets. Une opération terminale peut toutefois faire appel à des opérations internes (ou locales) au même objet.

La relation d'utilisation (*use*) est, par contre, définie sur des objets qui se trouvent, en principe, au même niveau de parenté. On dira qu'un objet X utilise un autre objet Y dès qu'une opération de l'objet X fait appel à une opération de l'objet Y.

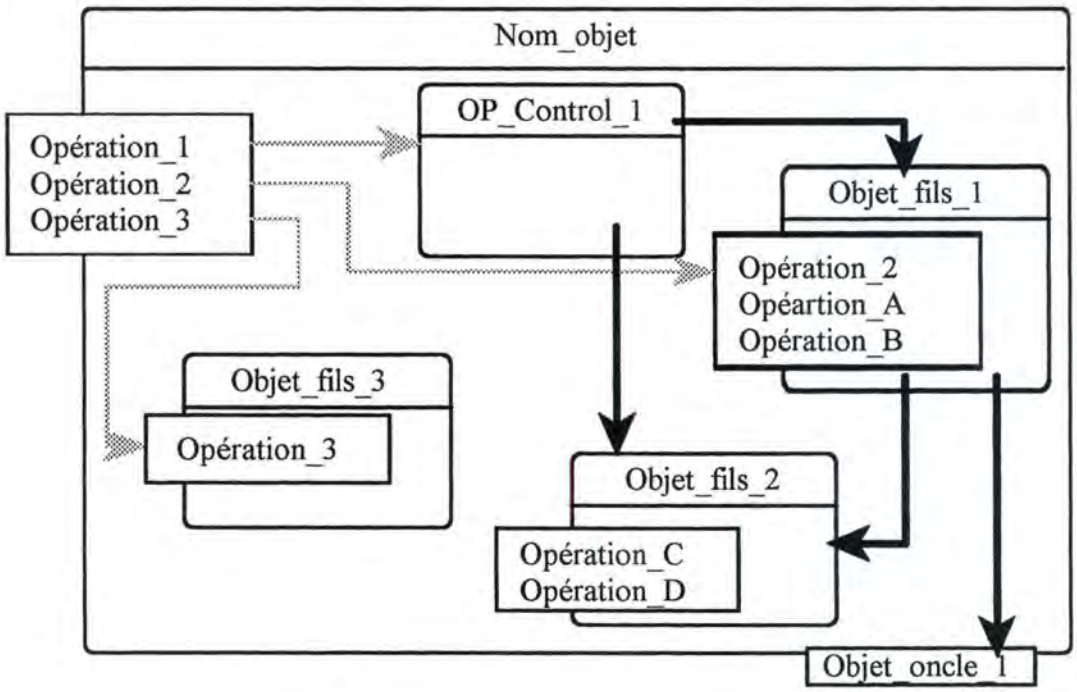



Figure 4.12 : Diagramme d'un objet père avec ses objets fils

HOOD pose deux restrictions importantes à la relation *use* sur les objets en imposant, d'une part, qu'un objet ne peut pas s'utiliser lui-même (à l'exception des appels récursifs d'une opération et des appels entre opérations terminales) et d'autre part, qu'il ne peut y avoir de cycle dans le graphe de la relation "utilise". Ces règles ont bien entendu pour objectif de limiter au maximum le "couplage" entre les objets et par conséquent, de simplifier leur développement et leur réutilisation.

[18] Au-delà de la terminologe père-fils, cette relation d'inclusion n'a donc aucun rapport avec une quelconque notion d'héritage.

En principe, la relation *use* ne peut donc associer que des objets frères entre eux. Cette règle est cependant trop restrictive et HOOD admet que, occasionnellement, un objet utilise un autre objet qui n'est pas l'un de ses frères et qui sera appelé "objet oncle".

La description des deux relations *include* et *use* est introduite dans la spécification textuelle par des mots-clés appropriés et la liste des noms des objets internes d'une part, et par celle des noms des objets oncles d'autre part. Cette dernière constitue ce que HOOD appelle l'interface requise (*required_interface*) de l'objet qu'il ne faut pas confondre avec son interface produite (*provided_interface*).

Graphiquement, les objets fils sont représentés à l'intérieur du cadre de l'objet père avec seulement leur interface produite. Des flèches pointillées associent les opérations de l'objet père avec celles qui leur correspondent dans l'un des objets fils. Les relations *use* entre les objets frère sont représentées par des flèches en trait plein. De plus, si des objets oncles sont utilisés, leur nom apparaît en bas du cadre de l'objet et une flèche émanant des objets qui les utilisent pointe sur eux (figure 4.12).

Les flots de données, qui sont en fait les paramètres formels des opérations peuvent aussi être indiqués en surcharge du diagramme de l'objet.

***Objets d'environnement et objets classe***

Les objets d'environnement permettent de décrire de façon explicite l'interface des objets qui utilisent le système à concevoir (ou qui sont utilisés par lui) sans en faire partie. Ces objets d'environnement permettent donc de représenter le contexte dans lequel le système devra fonctionner (système d'exploitation, réseau, interface graphique...). Les objets d'environnements peuvent faire partie de la relation "utilise" en tant qu'objets oncle.

Les objets classe permettent quant à eux, de décrire des objets réutilisables par d'autres systèmes ou des objets fort similaires dans une même application. Dans tous les cas, l'objet classe ne peut être utilisé directement mais doit être instancié afin de créer un "véritable" objet qui pourra alors être associé à d'autres via la relation "utilise" (en tant qu'objet oncle). Lors de l'instanciation, la description de l'objet peut être raffinée en spécialisant des paramètres formels génériques déclarés au niveau de l'objet classe.

***Description des opérations***

Il nous paraît intéressant de terminer ce survol du modèle HOOD en décrivant les informations qui sont attachées aux opérations.

Tout d'abord, il faut savoir que les opérations des objets non-terminaux sont en fait des coquilles vides : elles ne contiennent pas de code, mais permettent de raffiner la description (en texte informel) de l'opération et de préciser plus finement l'objet qui est chargé de la mettre en oeuvre. Au niveau terminal, par compte, la spécification d'une opération peut comporter, outre le texte informel de description, une liste d'opérations utilisées, un texte de pseudo-code et/ou de code spécifiant l'algorithme de l'opération, la liste des paramètres formels et de

leurs types ainsi que les listes d'exception qui peuvent être levées ou traitées par l'opération.

### 4.3.3 La méthode HOOD

Le principe de la méthode HOOD est implicitement décrit dans le modèle de représentation que nous venons d'évoquer. La relation d'inclusion sur les objets permet en effet de procéder à l'analyse et à la conception du système par un processus itératif de raffinement qui rappelle nettement l'approche *top-down* classique.

L'ensemble du système est tout d'abord décrit comme un seul objet (dit objet racine) qui doit offrir un certain nombre de fonctionnalités (ses opérations), puis, dans un second temps, cet objet racine est lui-même décomposé en objets fils qui seront agencés de telle façon qu'ils fournissent globalement les mêmes fonctionnalités que leur objet père. Le même processus de décomposition peut alors être répété autant de fois que nécessaire et ne s'arrête que lorsque tous les objets fils générés sont des objets terminaux. La méthode exige toutefois que la conception d'un niveau (fils) ne puisse commencer que lorsque la conception du niveau précédent (père) est entièrement terminée. On a donc une exploration "en largeur d'abord" de l'arbre de conception qui permet une certaine forme de structuration en couche de l'application développée.

Au-delà de ce grand principe de base, la méthode HOOD comprend plus d'une centaine de règles formelles qui fournissent entre autres (nous illustrons chaque fois par un exemple) :

- les définitions de base et les propriétés des types d'objets :
  "G-37 : les types d'objets ENVIRONMENT et CLASS sont mutuellement exclusifs".
- les règles relatives à la relation *use* :
  "U-3 : la relation *use* ne doit pas induire de cycle entre des objets passifs"
- les règles relatives à la relation *include* :
  "I-5 : un objet ne peut avoir plus d'un objet parent"
- les règles relatives aux opérations :
  "O-1 : une opération peut être dans une interface externe ou dans la partie interne d'un objet"
- les règles de visibilité :
  "V-4 : une entité qui est déclarée dans la partie requise (*required_interface*) d'un objet est visible de n'importe quel endroit de cet objet"
- les règles de consistance et de complétude :
  "C-8 : la déclaration de type, dans une interface produite d'un objet non terminal, doit être consistante avec la déclaration de type dans l'interface produite par l'objet fils qui implémente ce type" [LAI, 85-102].

On le voit, ces multiples règles précisent bien ce qui peut être fait et ce qui doit être évité pendant la conception. Toutefois, "l'identification des objets n'est ni

imposée ni même guidée par HOOD et reste donc du ressort exclusif du concepteur" [AUBERT, 87].

Différents outils ont été développés pour offrir un support au processus de conception en assurant notamment le contrôle automatique des règles de décomposition et de structuration des objets.

### 4.3.4 Remarques et commentaires

La première remarque qui vient à l'esprit concernant la méthode HOOD consiste à s'interroger sur le caractère véritablement orienté objet de cette méthode. En effet, on a pu voir que le modèle n'intègre que peu des concepts qui constituent traditionnellement le paradigme objet, (utilisation très limitée des classes, pas d'héritage, pas d'événement, ...). De plus, on ne peut manquer d'être frappé par l'insistance portée sur la structuration des opérations et donc des fonctionnalités offertes. La méthode HOOD ne serait elle donc qu'une méthode de conception fonctionnelle "aromatisée à la sauce objet" ? Ce n'est en tout cas pas l'opinion de Michel Lai qui affirme que cette perception provient des difficultés qu'éprouvent les concepteurs à se défaire de leurs habitudes d'analyse fonctionnelle. En fait, la démarche HOOD n'est pas appliquée à une opération mais à l'objet dans son ensemble. Il s'agit donc bien de décomposer des objets en d'autres objets avant de décrire la correspondance entre les opérations des objets père et fils. [LAI, 140]

Une caractéristique très remarquable de HOOD réside dans la séparation très claire entre la phase de spécification externe de l'objet et la phase de spécification interne. Cette distinction permet ainsi d'éviter (autant que possible) les influences parasites de l'analyse du "comment" sur la spécification du "quoi".

Il faut également noter l'intérêt de la spécification de ce que HOOD appelle l'interface requise et qui rassemble la description des éléments externes nécessaires à la mise en oeuvre d'un objet déterminé. Ces informations peuvent être de la plus haute importance en cas de réutilisation et lors de la maintenance d'une application.

Si l'on envisage à présent l'emploi de la méthode HOOD, on doit bien reconnaître qu'en dépit de la volonté affichée d'adaptation à de nombreux langages, c'est Ada qui est le langage cible privilégié de la méthode et que de nombreuses particularités semblent d'abord se justifier par une volonté de correspondance avec des mécanismes de ce langage. D'un autre point de vue, on est aussi frappé par le caractère méticuleux (et rébarbatif !) de toutes les règles que le concepteur doit respecter mais il s'agit peut-être du prix à payer pour garantir une extrême fiabilité au logiciel ainsi développé[19].

[19] S'agissant de logiciel destiné à l'aéronautique spatiale, on comprend que cette exigence soit tout à fait cruciale.

## 4.4 Quelques réflexions de synthèse

Au terme de ce bref examen, nécessairement rapide et incomplet, de ces trois méthodes de conception, il est intéressant de porter un regard "transversal" afin de relever quelques éléments de concordance et de discordance entre ces méthodes et leurs modèles associés.

### 4.4.1 Concepts mis en évidence

Tout d'abord, nous allons tenter de rassembler en un tableau synthétique les concepts les plus importants mis en évidence dans chaque modèle (fig 4.13). Bien entendu, les correspondances établies sont toujours plus où moins approximatives mais permettent de dégager une vue d'ensemble des différents modèles.

| Concepts | O* | OBLOG | HOOD |
|---|---|---|---|
| objet | objet | objet | objet |
| classe | schéma d'objet | classe | (classe) |
| attribut | attribut/référence | attribut | type |
| contrainte | contrainte | contrainte | - |
| opération | opération | évaluation d'attribut | opération |
| paramètre d'opération | - | attribut d'événement | flots de données |
| événement | événement | événement | - |
| héritage | héritage simple | héritage simple | - |
| comportement | graphe de transition d'état | comportement d'objet | - |

Figure 4.13 : Tableau synthétique des concepts retenus par les différents modèles

On observe ainsi un parallélisme assez marqué entre O* et OBLOG qui proposent pratiquement les mêmes concepts alors que HOOD reste largement en retrait. Si l'on analyse plus attentivement, on observe que O* s'attache à décrire

avec un peu plus de finesse la structure de données modélisées par l'objet alors qu'OBLOG se distingue par une description plus précise des opérations pouvant agir sur les objets. Rien d'étonnant à cela si l'on se rappelle que O* a été mis au point par des chercheurs spécialisés en systèmes d'informations alors qu'OBLOG vise avant tout la génération assistée de logiciel.

L'interprétation du concept d'événement peut, à priori, sembler différente pour ces deux modèles. En effet, O* distingue nettement les concepts d'opération et d'événement et alors qu'OBLOG les fusionne pratiquement (en prétendant qu'un événement est une action atomique qui modifie un objet) mais introduit les règles d'évaluation d'attribut pour exprimer comment s'effectue la modification de (l'état de) l'objet. Bref, dans les deux cas, on cherche d'une part à appréhender globalement les actions (ce qui revient souvent à en exprimer le "pourquoi") et d'autre part à exprimer le détail du fonctionnement (le "comment"). Il s'agit donc plutôt de deux niveaux distincts d'abstraction que l'on retrouve aussi, dans une certaine mesure, au travers des niveaux hiérarchiques de spécification des opérations dans le modèle HOOD.

Les modèles O* et OBLOG proposent tous deux de compléter la spécification d'une classe par un ensemble de contraintes portant sur les valeurs des attributs. On retrouve là le concept de contrainte d'intégrité bien connu dans le monde des systèmes d'information et qui est aussi exprimé par Bertrand Meyer sous le nom d'"invariant de classe" [MEYER, 155]. La notion de contrainte dans HOOD est par contre tout à fait différente et s'applique aux opérations.

### 4.4.2 Méthodes de conception

Si l'on s'intéresse plus particulièrement aux aspects méthodologiques, on est de prime abord frappé par la différence très importante d'approche entre HOOD qui crée des objets nouveaux à chaque niveau de sa hiérarchie (c'est-à-dire à chaque niveau d'abstraction) et O* ou OBLOG qui évite, tout au contraire, de multiplier ainsi les objets mais insiste (spécialement dans l'approche "en fontaine" ou "en spirale") sur les nécessaires itérations du processus de raffinement de la spécification des objets. Par ailleurs, ne peut-on voir dans la relation *include* entre les objets père et fils de HOOD un concept fort semblable à celui de composition (éventuellement de référence) qui permet de rassembler plusieurs objets O* en un objet agrégé ?

Bien entendu, il existe une distinction importante entre, d'une part, HOOD qui propose une démarche strictement descendante pour analyser, et d'autre part, O* et OBLOG qui sont plutôt orientés vers une démarche ascendante (ou mixte) en décrivant d'abord les objets perçus du monde réel puis en les agrégeant et en les combinant pour construire peu à peu l'application recherchée. Concernant cette distinction entre approche ascendante et descendante, il nous paraît intéressant de citer Bertrand Meyer :

> "Quoique l'on pourrait imaginer des techniques de conception descendante par objets, la conception ascendante s'allie mieux encore à notre approche.
>
> "Descendant" et "ascendant" sont, bien entendu, des caractéristiques extrêmes. On ne se met jamais à construire un système sans considérer

l'ensemble des composants disponibles; de la même façon, on ne construit jamais des composants logiciels sans une idée préconçue de leur futur utilisation. La différence, c'est que le concepteur descendant se concentre sur son problème spécifique, alors que le concepteur ascendant essaie autant que possible d'utiliser les composants existants et, lorsqu'il découvre qu'un nouveau composant doit tout de même être produit, essaie de lui donner un caractère général, afin que des futurs développements puissent le réutiliser"[MEYER, 386].

Enfin, pour terminer cette rapide mise en perspective de ces trois méthodes, il nous apparaît que l'on peut probablement qualifier la méthode HOOD de méthode d'architecture de logiciel alors que O* semble plus apte à l'analyse qui doit précéder l'architecture. OBLOG, quant à elle, est manifestement conçue pour jouer successivement ces deux rôles.

# 5

# Propositions méthodologiques

Les précédents chapitres nous ont permis de découvrir le paradigme objet et de cerner ses apports pour le développement de logiciel tant du côté de la conception, par le biais des diverses méthodes de conception, que de l'implémentation par l'apport des langages de programmation orientés objet. La plupart des recherches en ces domaines aboutissent aujourd'hui avec la mise sur le marché d'outils CASE (Computer Aided Software Engineering) très prometteurs mais... peu accessibles aux développeurs individuels, aux non-professionnels, aux enseignants...

Notre projet est donc de tenter d'extraire de ces multiples travaux (à tout le moins de quelques-uns d'entre eux) les "clefs" essentielles du développement orienté objet et de les synthétiser sous forme d'une série de recommandations méthodologiques à l'intention du public que nous venons d'évoquer, mais aussi de tous ceux qui sont aujourd'hui familier du développement de logiciel selon l'approche fonctionnelle traditionnelle et qui voudraient intégrer progressivement la dimension "objet" dans leurs développements.

## 5.1 Objectifs

Pour être cohérentes et pratiques, nos propositions doivent s'articuler autour de trois axes complémentaires : un modèle (ou langage) de représentation, une méthode de conception et un outil support.

Le **modèle** de représentation doit définir quels concepts sont retenus, comment ces derniers s'articulent les uns avec les autres et quels formalismes sont adoptés pour les représenter.

Le choix des concepts est toutefois contraint par divers impératifs. Tout d'abord nous chercherons clairement à favoriser, lorsque cela est possible, la cohérence avec d'autres modèles plus classiques et particulièrement avec le modèle Entité-

Association qui est certainement l'un des plus populaires aujourd'hui. D'autre part, nous essayerons de limiter le nombre de concepts qui seront introduits de façon à faciliter l'apprentissage du modèle et de la méthode associée. Il faut néanmoins éviter de tomber dans le piège d'une trop grande simplification qui conduirait à limiter l'expressivité du modèle. Enfin, nous veillerons également à prévoir, pour les concepts les plus importants, des notations graphiques à la fois simples, claires, intuitives et pratiques à mettre en oeuvre tant lors de l'élaboration manuscrite d'un schéma que lors de sa description à l'aide d'un logiciel d'édition ou de traitement de texte.

La **méthode**, ou plus modestement l'ensemble de propositions méthodologiques que nous voudrions proposer, devrait allier simplicité et souplesse pour pouvoir être mise en oeuvre par de nombreuses catégories de développeurs. Elle devrait ainsi permettre un déroulement progressif de la conception et de l'implémentation du logiciel en projet, tout en laissant l'opportunité de remettre en cause, de compléter ou de corriger des éléments qui ont déjà été spécifiés. Une des exigences importantes nous semble être la nécessité d'embrasser le processus de développement dans son ensemble, c'est-à-dire depuis la spécification jusqu'à l'implémentation en n'omettant pas les aspects système d'informations et interface utilisateur.

Enfin, modèle et méthode devraient être complétés par un **outil** logiciel qui offre une assistance et un support pour leur mise en oeuvre en prenant notamment en charge certaines tâches fastidieuses et en permettant au concepteur de "naviguer" facilement entre tous les éléments de son projet. La description du prototype d'outil que nous proposons sera réalisée dans le chapitre suivant.

Auparavant, nous allons présenter dans ce cinquième chapitre le modèle et la méthode qui constituent le coeur de ce mémoire. Certes, tous les objectifs qui viennent d'être évoqués ne pourront pas être atteints complètement (certains étant d'ailleurs contradictoires) mais nous avons voulu les énumérer ici pour montrer les préoccupations qui nous animent au moment d'élaborer nos propositions méthodologiques.

Notre modèle est, tout naturellement, l'héritier des trois modèles qui ont été présentés dans le chapitre 4. Aussi, pour ne pas alourdir inutilement la présentation, nous nous permettrons de ne donner de références explicites que pour les emprunts qui n'ont pas encore été signalés dans les chapitres précédents.

## 5.2 Modèle orienté objet

### 5.2.1 Objet

Par définition, un système orienté objet est un système qui manipule des objets. S'agissant d'un système informatique, les objets manipulés sont des objets informatiques qui peuvent avoir un correspondant plus ou moins concret dans le monde réel. Ce sera ainsi souvent le cas pour les objets manipulés par un

système de gestion de base de données : les objets informatiques `Facture`, `Client`, `Ouvrage`, `Prêt`, ... gérés par le système informatique ont en effet un correspondant tangible dans le monde réel. Tous les objets informatiques n'ont cependant pas toujours de correspondant dans le monde réel.

D'une façon générale, on peut admettre de considérer comme objet informatique toute entité utilisée dans le système informatique qui possède une mémoire (c'est-à-dire à laquelle est affectée une ou plusieurs informations qui la décrive) et qui est susceptible de produire des comportements.

> *C'est volontairement que nous ne donnons pas une définition plus précise du concept d'objet afin de ne pas en limiter le champ d'application. Ainsi, même si cette notion d'objet s'applique sans difficulté à tous les objets ayant un correspondant évident dans le monde réel, elle permet aussi de considérer comme objet toute structure de donnée pour laquelle on définit un ensemble de comportements acceptables. Le choix des objets qui seront considérés comme pertinents et utiles dans le cadre du projet relève de toute façon de la méthodologie.*

### 5.2.2 Classe

Le concept de classe est véritablement le concept le plus important pour notre travail de modélisation. La description d'un système logiciel orienté objet est en effet essentiellement concentrée dans la description des classes d'objets qui pourront interagir au sein du logiciel.

> *Il peut paraître, de prime abord, étonnant que le concept principal d'un modèle orienté objet ne soit pas le concept d'objet. C'est pourtant tout à fait normal si l'on se rappelle que nous nous concentrons ici sur le processus de conception d'un logiciel. Les objets qui seront effectivement manipulés par le logiciel lorsqu'il sera mis en service n'existent pas au moment de la conception du logiciel et les caractéristiques spécifiques de chacun d'eux sont sans importance pour nous. Ce qui nous importe, c'est de détecter combien de catégories distinctes d'objets seront nécessaires et quelles structures d'information sont les plus aptes à représenter (plus tard) ces objets. Ce qui nous intéresse, c'est donc de décrire les classes auxquelles seront attachés ces objets et non les objets eux-mêmes.*

Une classe permet donc de spécifier les caractéristiques partagées par une série d'objets informatiques. Bien souvent cependant, nous serons amenés à décrire des classes qui ne seront destinées qu'à "contenir", à un instant donné, un seul objet informatique. Ainsi, par exemple un logiciel de comptabilité peut être

conçu pour qu'à un instant donné, il ne "contienne" qu'un seul objet `Client`, un seul objet `Facture` ainsi qu'un seul objet `Imprimante`, mais plusieurs objets `Produit`.

> *Cette situation fort courante est sans doute à l'origine de la constante confusion de langage qui nous fait volontiers parler d'objet plutôt que de classe. Si le logiciel ne considère qu'une facture à la fois, il est en effet naturel de parler de l'objet `Facture` plutôt que de l'objet actuellement manipulé appartenant à la classe `Facture`.*

En pratique, nous décrirons chaque classe en lui associant :

- un ***nom*** (unique parmi tous les noms de classe);
- une ***définition***, c'est-à-dire une courte description textuelle de forme libre;
- les listes de ses ***attributs*** et de ses ***opérations***;
- la liste (éventuellement vide) de ses ***contraintes***.

Les concepts d'attribut, opération et contrainte vont être présentés en détail dans les paragraphes suivants.

Quant à la représentation graphique d'une classe, elle est régie par les règles suivantes :

- Une classe est représentée par un rectangle contenant son nom ;
- Les attributs et les opérations sont énumérés en dessous du rectangle suivant des règles qui seront présentées plus loin.

### 5.2.3 Attributs

Les attributs d'une classe permettent de décrire la structure des informations qui sont attachées à chacun des objets de la classe. Nous distinguerons deux catégories bien distinctes d'attributs : les attributs composants d'une part et les attributs références d'autre part.

Comme cela a déjà été souligné au chapitre 3, les ***attributs composants*** désignent des informations (simples ou complexes) qui sont "intimement" attachées aux objets de la classe décrite. Par exemple, les attributs `Nom` et `Date_de_naissance` d'une classe `Personne` seront considérés comme composants de la classe car ces informations appartiennent véritablement aux objets de la classe : c'est grâce à ces informations que l'objet peut être décrit et identifié. Inversément, ces informations perdent pratiquement tout sens dès qu'elles ne sont plus attachées à "leur" objet.

Notons qu'un attribut composant peut désigner une valeur (un nombre, une chaîne de caractères, une date, ...) ou un ensemble de valeurs de même type (une série de date par exemple) mais aussi un autre objet, voire un ensemble d'autres objets (d'une même classe). Ainsi par exemple, une classe `Client` peut être décrite entre autre par un attribut `AdresseClient` destiné à contenir un objet de la classe `Adresse` ainsi que par un attribut `Factures` capable de contenir un ensemble d'objets de la classe `Facture`. Une facture n'a en effet aucun sens si

elle n'est pas "attachée" à un client et elle ne peut pas "changer" de client pendant son existence.

Par opposition aux attributs composants, on peut définir les ***attributs références*** d'une classe comme la désignation de pointeurs vers d'autres objets qui sont "momentanément" associés à cette classe. La notion essentielle est ici celle de partage d'une information (complexe) par plusieurs objets. Par exemple, l'attribut `Produit_commandé` d'une classe `Commande` est nécessairement un attribut référence car l'objet (de la classe) `Produit` ainsi désigné ne peut être la propriété d'une seule commande mais doit au contraire être partageable par de nombreuses commandes. On voit donc qu'un attribut référence désigne toujours un (éventuellement plusieurs) objet qui a son existence propre et qui n'est donc que peu ou pas affecté par la naissance ou la disparition de l'objet qui le référence. Il en découle que les attributs références ne peuvent désigner que des objets et pas de "simples" informations (valeurs simples ou multiples).

> Une remarque doit être faite sur le partage de valeurs. On pourrait en effet imaginer de dire que la valeur "DUPOND" est un nom partagé par toutes les personnes qui s'appellent DUPOND. Ce partage n'est cependant que purement formel car, du point de vue sémantique, il n'apporte rien de plus que la connaissance de la séquence des caractères qui forment ce nom. Deux objets `Client` qui partageraient cette même valeur de `Nom` désignent-elles la même personne ou des personnes qui sont apparentées : rien ne permet de l'affirmer. Par contre, si deux objets `Ouvrage` partagent le même objet `Auteur` alors, non seulement le nom de l'auteur est connu (et identique) mais bien d'autres informations peuvent en être inférées (par exemple que les deux ouvrages n'ont certainement pas été écrits à plus de cent ans d'intervalle).

Chaque attribut d'une classe doit être caractérisé par :

- un ***nom*** qui l'identifie parmi les attributs de la classe (le nom d'un attribut d'une classe ne peut cependant être égal au nom d'une opération de la même classe).
- une ***définition*** qui précise dans une courte description textuelle le rôle de l'attribut.
- une ***nature*** : composant ou référence.
- une ***obligation*** qui détermine si la présence d'une "vraie" valeur (au moins) est obligatoire ou facultative.

> La représentation de l'absence d'une information n'est pas toujours simple. Souvent, on représente l'absence de valeur par une chaîne vide pour une information alphanumérique et par le valeur zéro pour une information numérique mais ces représentations sont inadéquates

lorsque ces valeurs peuvent aussi être des valeurs valides dans le contexte considéré. Dans le cas d'un attribut facultatif, on conviendra donc d'ajouter une constante spéciale `NIL` à l'ensemble des valeurs possibles de son type et qui servira, le cas échéant, à représenter l'absence de "vraie" valeur.

- un ***type*** qui précise la nature des valeurs qui peuvent être prise par la valeur de l'attribut (ou les valeurs s'il s'agit d'un attribut multiple). Le type d'un attribut peut être soit un type que nous qualifierons de "standard" s'il s'agit de valeurs simples (nombre, date, chaîne de caractères, booléen, ...), soit le nom d'une classe définie par ailleurs, si l'attribut désigne un ou plusieurs objets.

  Les types standards reconnus sont les suivants :

  `@NUMERIQUE` : nombre entier ou réel;
  `@CARACTERE` : un seul caractère alphanumérique;
  `@CHAINE` : chaîne de caractères de longueur quelconque mais ne contenant pas de "retour à la ligne" (paire CR/LF);
  `@TEXTE` : chaîne de caractère, généralement assez longue et structurée en "texte" par des retours à la ligne.;
  `@LOGIQUE` : valeur booléenne : `.T.` = vrai et `.F.` = faux;
  `@DATE` : toute date valide;

  Par souci de simplicité, nous n'avons pas retenu ici de types complexes qui pourraient être définis par l'utilisateur tels que des types énumérés, intervalles, ... car il existe au moins deux façons de contourner cette limitation : soit en décrivant une classe dont les objets seront du type demandé, soit en ajoutant une contrainte sur les valeurs admissibles pour l'attribut dont on définit le type. D'autres types pourraient toutefois être introduits sans remettre en cause le modèle.

- une ***multiplicité*** qui indique si l'attribut est simple (n'admet qu'au plus une valeur à tout instant) ou multiple (admet éventuellement plusieurs valeurs simultanées).

  Implicitement, on admettra que les valeurs d'un attribut multiple sont ordonnées. Cela permet de considérer que les valeurs sont rangées dans un ordre précis et qu'une même valeur peut être répétée. Ces propriétés ne seraient pas vraies dans le cas d'une structure d'ensemble. L'implémentation peut se réaliser en utilisant une liste dynamique.

Les conventions de représentation des attributs d'une classe sous forme de schéma ont déjà été présentées et illustrées au chapitre 3 et sont brièvement rappelées ci-dessous :

- Les attributs sont énumérés en dessous du rectangle de la classe et sont introduits par une petite flèche (indiquant que l'objet est "composé" de cet attribut) ;
- Le caractère facultatif d'un attribut est indiqué par la mise entre parenthèses du nom de l'attribut ;
- Le type d'un attribut est désigné par le nom d'un type standard écrit en majuscule et préfixé par le symbole @ ou par le nom de la classe du ou des objet(s) pointé(s) ;
- La nature du lien d'association et sa multiplicité sont représentés par différentes sortes de flèches :

| | | |
|---|---|---|
| Composition simple | `A ——> B` | "A est composé d'un B" |
| Composition multiple | `A —>> B` | "A est composé de plusieurs B" |
| Référence simple | `A ----> B` | "A réfère à un B" |
| Référence multiple | `A --->> B` | "A réfère à plusieurs B" |

La figure 5.1 nous donne un exemple illustratif de cette notation.

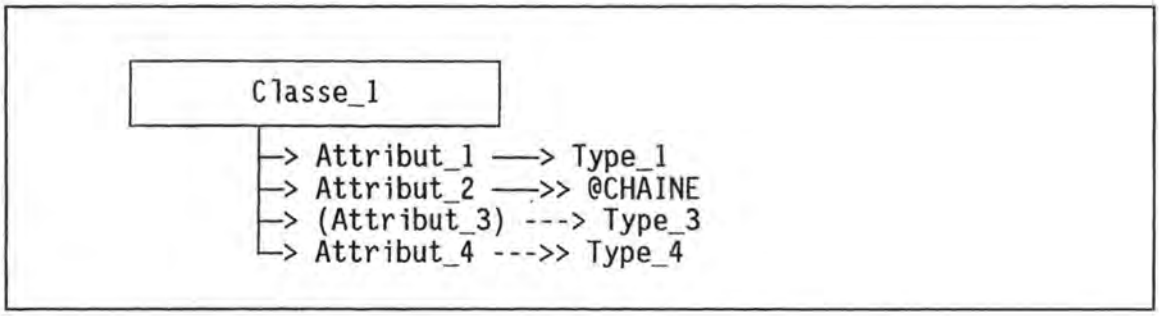



Figure 5.1 : Exemple de représentations des attributs d'une classe

### 5.2.4 Opérations

Les opérations d'une classe permettent de décrire l'ensemble des comportements que l'on peut attendre de la part des objets de cette classe.

> Même si par commodité et par cohérence avec le vocabulaire couramment employé en software engineering, nous avons concervé le terme d'opération pour désigner les modules décrivant le comportement des classes, il nous apparaît que le terme de "service" serait peut-être plus adéquat. Nous montrerons en effet, lorsque nous présenterons la méthode de conception, que les classes sont le plus souvent créées afin que les objets de ces classes puissent offrir un certain nombre de services à des objets de plus haut niveau.

Idéalement, les opérations attachées à une classe devraient pouvoir être rangées dans deux catégories disjointes : d'une part, les opérations qui modifient les attributs de la classe et qui ne fournissent pas de résultat (communément

appelées procédures dans les langages de programmation) et d'autre part les opérations qui fournissent un résultat et qui dans ce cas ne sont pas autorisées à modifier les valeurs des attributs de la classe (appelées communément fonctions) [MEYER, 390].

Pour des raisons de commodité, nous admettrons cependant, dans le langage, que certaines opérations puissent appartenir aux deux catégories c'est-à-dire effectuer des modifications sur l'objet et retourner un résultat mais dans ce cas, il faut que le résultat retourné soit en lien direct avec les mises à jour réalisées. Typiquement, il s'agit ici d'autoriser, par exemple, une opération à retourner une information booléenne ou numérique qui indique dans quelle mesure la modification demandée a pu être effectuée avec succès.

> En fait, on est ici tout près du mécanisme d'exception [MEYER 179-192] qui permet aussi de détecter puis de traiter convenablement les situations anormales et d'en informer, le cas échéant, le reste du système.

Pratiquement une opération est décrite par :

- un ***nom*** qui l'identifie parmi les opérations de sa classe et ne peut pas être égal à un nom d'attribut de cette même classe.

  > Pour le choix du nom d'une opération, il est conseillé de prendre un verbe à l'infinitif ou une expression verbale lorsque l'opération est une procédure (par exemple : `Ouvrir`, `Aller_Début`, ...) et de prendre un substantif lorsque l'opération retourne un résultat (par exemple : `Solde`, `Description`, ...).

- une ***définition*** qui précise en quelques mots le service qui est fourni par l'opération. La définition est donc une sorte de spécification simplifiée de l'opération dont la forme est laissée à l'appréciation de l'utilisateur.
- un ***type*** et une ***multiplicité*** du résultat lorsque l'opération en produit un. Comme pour les attributs, le type peut être le nom d'un type "standard" ou celui d'une classe tandis que la multiplicité peut être simple ou multiple.
- une liste d'***arguments*** qui peuvent ou doivent être passés à l'opération pour qu'elle puisse effectuer le service attendu. La description du concept d'argument est présentée dans le paragraphe suivant.
- un ***algorithme*** qui détaille la marche à suivre pour effectuer le service prévu. La rédaction de l'algorithme, si elle est nécessaire au terme de la conception pour que l'implémentation puisse s'effectuer dans les meilleures conditions, ne doit cependant pas être réalisée dès la déclaration de l'opération comme nous le verrons dans la méthode. Dans une phase intermédiaire, l'algorithme proprement dit peut être remplacé par une expression moins formelle de la marche à suivre que nous appellerons le ***script*** de l'opération.

Comme nous l'avons déjà annoncé au chapitre 3, la représentation schématique des opérations d'une classe est assez semblable à la représentation des attributs aux différences suivantes près :

- le nom de l'opération est introduit par un trait horizontal (au lieu d'une flèche).
- le nom est suivi d'un couple de parenthèses renfermant le cas échéant les noms des différents arguments.
- si l'opération retourne un résultat, le type en est précisé comme pour un attribut composant sinon l'indication de type est omise.

La figure 5.2 illustre ces conventions :

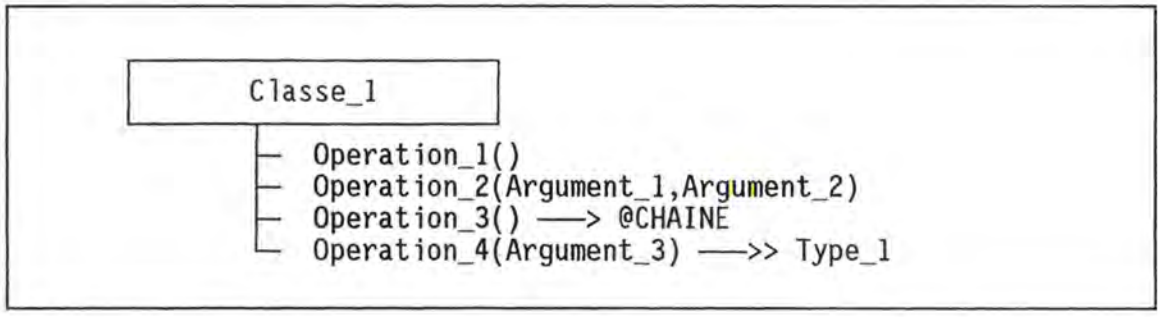



Figure 5.2 : Exemple de représentation des opérations d'une classe

### 5.2.5 Arguments

Concept moins fondamental que ceux qui ont été présentés jusqu'ici, le concept d'argument est néanmoins fort utile pour spécifier quelles informations doivent être fournies à une opération d'une classe pour qu'elle puisse rendre le service que l'on attend. C'est en fait la même notion que celle de "paramètre formel" qui est bien connue en programmation. Ainsi, si l'on considère un objet `Fenêtre` dont les dimensions sont supposées définies par ses attributs et que l'on désire lui demander de s'ouvrir à un endroit spécifique de l'écran, on peut imaginer de le doter d'une opération `Ouvrir` ayant deux arguments `Ligne` et `Colonne` qui permettront de passer à l'objet `Fenêtre` la position désirée pour son sommet supérieur gauche.

S'agissant de la désignation d'une information, la spécification d'un argument obéit à des règles fort semblables à celles qui ont été définies pour les attributs. Ainsi, tout argument porte un ***nom*** qui doit l'identifier parmi les autres arguments de l'opération, éventuellement une ***définition*** qui décrit sommairement son rôle, ainsi qu'un ***type*** (qui peut désigner un type standard ou une classe définie par ailleurs) et une ***multiplicité***. De plus, un argument peut être obligatoire ou facultatif. Dans ce dernier cas, il y a un lieu alors de préciser la ***valeur par défaut*** qui doit être donnée à l'argument si aucune valeur n'est passée lors de l'envoi du message demandant l'exécution de cette opération.

Il va sans dire, qu'un argument ne doit jamais être utilisé pour passer des informations qui peuvent être directement trouvées par l'opération en consultant les attributs de l'objet auquel est adressé le message.

### 5.2.6 Contraintes

A l'instar des contraintes d'intégrité bien connues dans les systèmes d'informations, il s'avère utile de compléter la description d'une classe par l'énoncé de contraintes diverses qui précisent les conditions qui doivent être vérifiées en permanence par l'état des objets de la classe. Il s'agit globalement de la même notion que celle d'invariant de classe qui est définie dans [MEYER, 155-160].

En pratique, les contraintes permettent le plus souvent d'apporter des restrictions aux domaines de valeur des attributs ou d'exprimer des relations entre les valeurs de plusieurs attributs. De plus, les contraintes associées à une classe peuvent aussi contenir des relations exprimant des restrictions sur le comportement dynamique des objets de la classe. Ainsi par exemple, une contrainte peut exprimer que la valeur d'un attribut ne peut que croître.

Pratiquement une contrainte est décrite par :

- un ***nom*** qui l'identifie au sein des contraintes de la classe ainsi que des contraintes des classes de sa lignée d'héritage;
- une ***définition*** exprimant, en langage courant, le sens de la contrainte;
- un ***énoncé*** exprimant la contrainte de manière (plus) formelle.

### 5.2.7 Evénements, Acteurs et Générateurs

Les concepts d'événement, d'acteur et de générateur constituent trois concepts mineurs - tout au moins au stade actuel de réflexion sur le modèle et la méthode - qui apportent essentiellement une aide au niveau de la spécification initiale du système à développer.

> Il n'entre pas dans les prétentions de notre langage de permettre une modélisation complète de l'univers considéré comme le proposent les plus récentes méthodes de spécification OO. Notre objectif est seulement de pouvoir représenter, aussi simplement que possible, les objets de l'environnement qui sont appelés à interagir avec le système en cours de développement. C'est la raison essentielle de l'introduction des concepts d'acteur et corollairement d'événement.

Partant du principe que les événements explicitent les causes des modifications des objets [CAUVET, 17] le concept d'***événement*** est introduit pour modéliser la cause du déclenchement d'une opération lorsque ce déclenchement ne résulte pas simplement et directement de l'envoi d'un message par un autre objet du système. Il en résulte que les événements ont pour rôle de modéliser les interactions de l'environnement sur le système.

L'occurrence d'un événement ne se produit pas spontanément et est due à l'initiative soit d'un ***acteur*** c'est-à-dire d'une entité extérieure au système tel que, typiquement, l'utilisateur du logiciel, soit d'une entité interne au système qui détecte qu'une condition est remplie et qu'un événement doit donc être généré. Le concept de ***générateur*** permet alors de modéliser cette entité responsable de l'occurence d'un événement.

> Si les concepts d'acteur et d'événements sont assez intuitifs et faciles à manipuler, il faut bien reconnaître qu'il n'en est pas de même pour le concept de générateur. Ce dernier a toutefois été intégré au modèle pour élargir le domaine d'application à des systèmes où l'initiative n'est pas l'apanage exclusif de l'utilisateur (interactions via des "triggers" dans la base de données par exemple).

Comme tous les autres concepts déjà évoqués, les événements, acteurs et générateurs doivent être décrits pour un ***nom*** qui permet de les identifier et par une ***définition*** qui les spécifie succinctement. De plus, un événement est associé à la fois aux acteurs et générateurs qui en provoquent l'occurrence ainsi qu'aux opérations qui sont déclenchées. Un générateur comprend de plus l'***énoncé*** de sa condition d'occurence qui peut être évaluée par observation de différents attributs appartenant éventuellement à des objets de plusieurs classes.

> A ce point de la présentation de notre modèle, on s'étonnera peut-être de ne retrouver qu'une utilisation si modeste du concept d'événement ainsi que de l'absence totale des notions d'états et de transitions qui sont pourtant importantes, sinon essentielles, dans d'autres modèles ([BRUNET], [SERNADAS]). Il ne s'agit pas, bien entendu, de contester l'intérêt de ces concepts mais tout simplement de faire un choix de simplification dans le but d'offrir un modèle et une méthode qui respectent fondamentalement les principes du paradigme objet, qui en conservent le maximum d'avantages mais qui soient aussi facilement accessibles à un maximum d'utilisateurs et particulièrement à ceux qui cherchent à migrer "en douceur" de l'approche "analyse fonctionnelle" vers l'approche objet. On notera toutefois que la possibilité de spécifier des contraintes dynamiques permet précisément de spécifier quels sont les changements d'état qui sont admissibles et ceux qui ne le sont pas. Tous les avantages de la modélisation des états et transition ne sont donc pas perdus.

### 5.2.8 Héritage

Notre modèle se doit évidement d'inclure le concept d'héritage mais nous le limiterons toutefois à l'héritage simple. Toute classe A peut donc être définie

comme héritière d'une autre classe B. De cette façon, la classe A, appelée sous-classe de la classe B, est pourvue automatiquement de tous les attributs, opérations et contraintes définis pour la classe B qui est appelée la superclasse de la classe A. Selon le principe de l'héritage simple, une classe ne peut hériter que d'une seule autre classe (avoir au plus une superclasse) mais une classe peut bien entendu avoir plusieurs sous-classes distinctes.

Ainsi par exemple, il est possible de définir une classe `Personnel` puis plusieurs classes héritières `Ouvrier`, `Employé`, `Directeur` qui héritent de la classe `Personnel`. Si l'on représente le lien d'héritage par une flèche à double ligne orientée de la sous-classe vers la superclasse, on peut schématiser la situation par la figure 5.3.

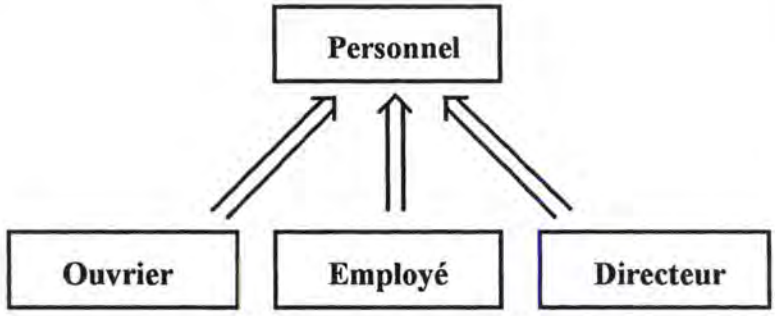



Figure 5.3 : Héritage

Tous les attributs et toutes le opérations décrits pour la classe `Personnel` sont donc automatiquement définis pour les classes `Ouvrier`, `Employé` et `Directeur`. De plus, toutes les contraintes énoncées dans la classe `Personnel` doivent aussi être respectées par toutes les instances des classes héritières [MEYER 304-305]. Cela postule que les contraintes appliquées à une sous-classe ne peuvent jamais être "plus faibles" que les contraintes supportées par la superclasse et il est donc exclu de définir une sous-classe pour "assouplir" des contraintes imposées par la superclasse.

Pour prendre un exemple facile à comprendre (mais relativement stupide) considérons que la classe `Personnel` possède un attribut `Age` et une contrainte qui spécifie que `16 ≤ Age ≤ 65`. Dans ce cas, il est interdit de définir une contrainte sur la classe `Directeur` qui spécifie que `16 ≤ Age ≤ 80` car on pourrait trouver des instances de la classe `Directeur` qui ne soient pas aussi des instances de la classe `Personnel`.

Par ailleurs, nous admettons aussi une autre restriction sur le mécanisme d'héritage qui est relativement classique, même si elle n'est pas postulée dans le modèle O* [BRUNET a, 21], et qui simplifie l'implémentation. Cette restriction veut que les sous-classes soient mutuellement exclusives c'est-à-dire, pour l'illustrer en faisant référence à l'exemple de la figure 5.3, qu'un objet appartenant à la classe `Ouvrier` ne peut pas être aussi `Employé` ou `Directeur`. Il existe donc seulement quatre catégories possibles d'instances dans ce schéma : les instances de `Personnel` qui ne sont ni `Ouvrier`, ni `Employé`, ni `Directeur`, les instances d'`Ouvrier` qui sont aussi (automatiquement) instance de `Personnel`,

les instances d'Employé et de Personnel et les instances de Directeur et de Personnel.

Dans les schémas de classes, l'héritage sera réprésenté par des flèches et des lignes doubles comme illustré à la figure 5.4 (Classe_3 hérite de Classe_2 qui hérite elle-même de Classe_1).

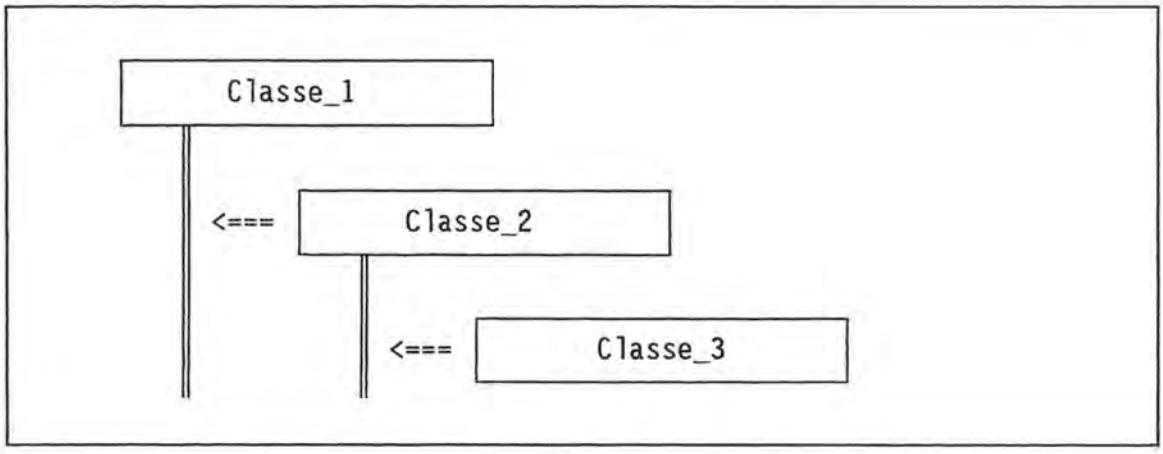



Figure 5.4 : Représentation de l'héritage en cascade.

### 5.2.9 Synthèse

En guise de synthèse et d'illustration de cette présentation de notre proposition de modèle orienté objet on trouvera à la figure 5.5, un schéma respectant la syntaxe que nous proposons et décrivant le méta-modèle de notre modèle. L'oeil est évidemment tout de suite attiré par les noms des huit concepts de base du modèle qui sont (dans le méta-modèle) représentés comme des classes. Chaque concept peut en effet être décrit par les informations qui le qualifient (ses attributs) et par une série de contraintes propres (qui ne sont pas représentées ici). Le schéma comporte cependant trois autres classes qui méritent un mot d'explication.

Tout d'abord, on observe que toutes les classes décrivant un concept héritent de la classe Concept. On a en effet pu se rendre compte que tout objet manipulé lors de la modélisation (c'est-à-dire toute instance d'un concept) doit être décrit par un nom et par une brève définition textuelle. La classe Concept peut être considérée comme une classe générique car elle n'est pas destinée à être instanciée directement.

La classe Objet permet elle aussi de généraliser les trois concepts d'Attribut, d'Opération et d'Argument car ils ont en commun la possibilité de contenir ou de produire une information dont il convient de préciser le type, la nature et la multiplicité. La dernière classe, nommée TypeObjet, a précisément été créée pour agréger en une seule entité les trois notions de type, de nature et de multiplicité qui forment naturellement un tout qu'il ne serait pas heureux de dissocier.

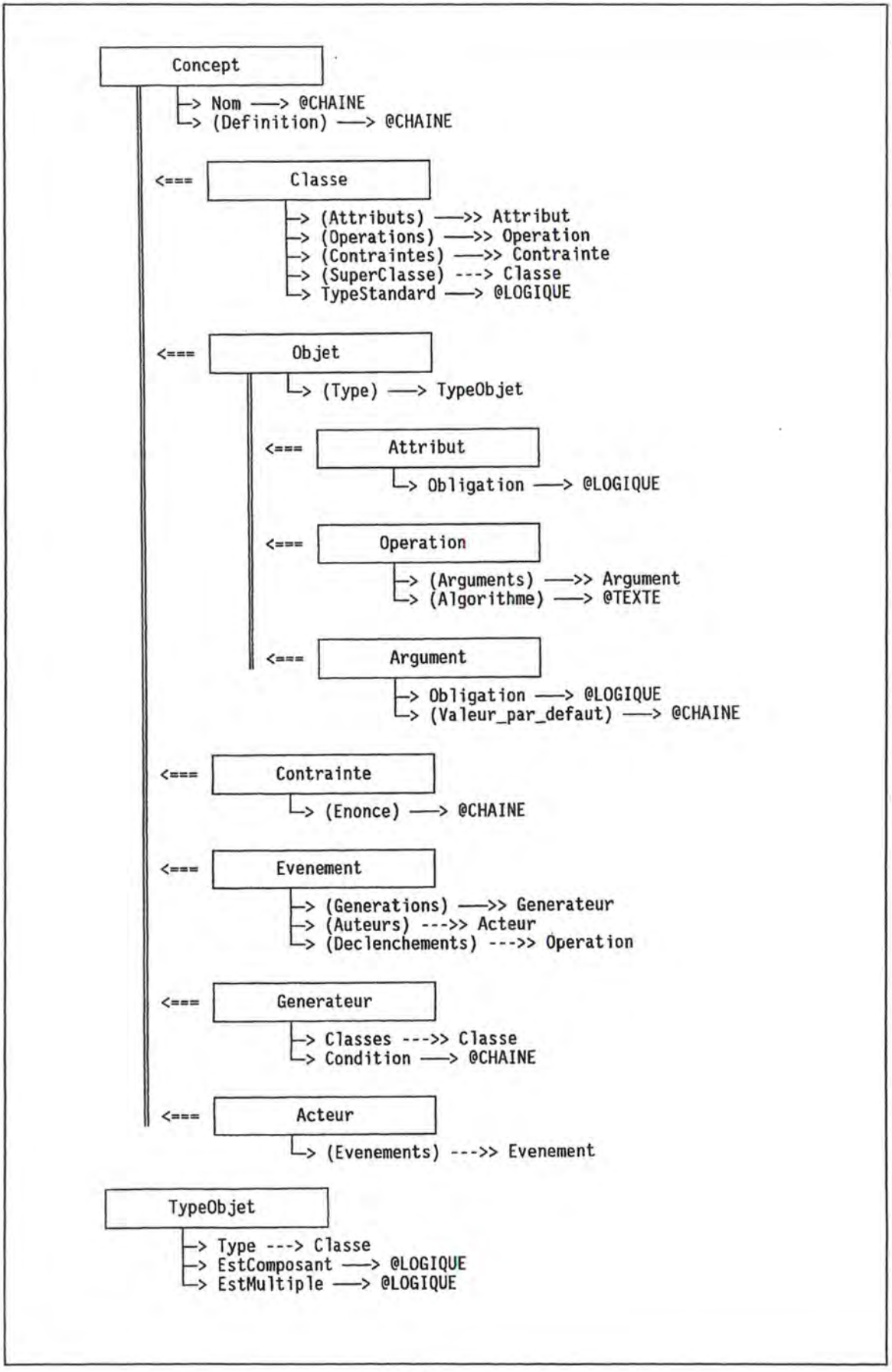



Figure 5.5 : Schéma du méta-modèle

Pour être complet, il faut encore noter que contrairement à ce qui a été dit au paragraphe 5.2.3, le type d'un objet désigne toujours une classe et non pas tantôt une classe, tantôt un type "standard". Cette légère modification s'explique dès que l'on admet que le concept de classe peut aussi bien représenter de "vraies" classes, définies par le concepteur, que les "fausses" classes que sont les types standards. L'attribut booléen `TypeStandard` de la classe `Classe` permet de distinguer précisément les "vraies" classes des "fausses".

## 5.3 Méthode de conception "en tire-bouchon".

Comme nous l'annoncions au début de ce chapitre, la méthode de conception que nous allons maintenant présenter a l'ambition d'embrasser une très large portion du processus de développement de logiciel et pourrait sans doute être qualifiée de méthode de développement. Par contre, le terme de méthode est probablement un peu usurpé car les quelques principes qui vont être présentés n'ont pas la prétention de fournir les moyens nécessaires pour résoudre toutes les difficultés qui peuvent se présenter lors du développement de logiciel de gestion d'un système d'information dans un environnement micro-informatique.

Le qualificatif "en tire-bouchon" a été choisi pour synthétiser en une image les principes fondamentaux de la méthode proposée. En effet, nous allons voir qu'à l'instar de la méthode HOOD, notre démarche sera globalement descendante comme le tire-bouchon qui pénètre progressivement dans le bouchon. Telle la vrille d'un tire-bouchon, nous ne chercherons cependant pas à "descendre" trop rapidement dans notre analyse et nous explorerons soigneusement notre projet niveau par niveau. Enfin, lorsque toute l'analyse nous aura fait découvrir l'ensemble des classes nécessaires et des services qui devront être offerts, il ne restera plus qu'à peaufiner l'analyse des classes et à en assurer l'implémentation en partant cette fois des niveaux les plus bas vers le niveau le plus élevé un peu à la manière de la vrille qui est extraite du bouchon.

La méthode peut ainsi être structurée en cinq phases (en principe) successives :

1. Analyse des besoins avec notamment le relevé des événements
2. Conception de la base de données
3. Elaboration du scénario de l'interface
4. Conception proprement dite de l'application
5. Rédaction des algorithmes et vérification de la cohérence.

> Les différentes phases de la conception devraient, à priori, se dérouler successivement (à l'exception des phases 2 et 3 qui peuvent être inversées sans aucun problème). En pratique, il n'en est jamais ainsi, car, comme l'a bien suggéré C. Rolland avec la métaphore de la fontaine, le concepteur est régulièrement amené à faire apparaître des éléments qui n'avaient pas été perçus au départ et qui l'oblige à compléter ou raffiner les descriptions qui avaient été faites jusque là.

*Nous admettrons donc que les différentes phases doivent être conduites en essayant de minimiser les retours en arrière.*

Avant d'étudier plus avant chacune de ces phases, présentons un exemple qui mous permettra de donner, pas à pas, une illustration de chaque phase. Il s'agit d'un problème réel et relativement simple qui me fut proposé voici quelques années par un enseignant du secondaire.

Signalons dès à présent, que la documentation complète de l'analyse de ce projet, générée à l'aide du logiciel SACOO d'aide à la conception que nous présenterons au chapitre suivant, est fournie en annexe du mémoire.

***Enoncé de l'exemple illustratif "Voyage scolaire".***

Il s'agit de concevoir un logiciel à faire fonctionner sur PC (mono-poste) et qui permette d'automatiser certaines des tâches afférentes à la gestion, par un enseignant, d'un voyage scolaire qui est organisé annuellement pour des élèves de rhétorique d'une école donnée mais pour lequel il est toutefois possible que quelques élèves d'autres classes et quelques professeurs ne faisant pas partie de l'équipe des accompagnateurs puissent également participer au voyage.

Les tâches demandées concernent essentiellement la gestion des inscriptions, l'édition de listes (essentiellement à destination des institutions douanières et des pensions où les participants doivent résider) et surtout la gestion financière des payements effectués par les parents des élèves. Sans entrer dans trop de détails, il faut savoir que le prix à payer peut varier suivant que le participant est élève ou accompagnateur, qu'il possède ou non un billet de service SNCB, qu'il est déjà affilié à une assurance voyage, ... Par ailleurs, l'organisateur accepte que le paiement soit effectué en plusieurs versements échelonnés. Il en résulte que des écarts tant positifs que négatifs sont régulièrement observés entre les montants dus et les montants effectivement payés.

Aussi, le logiciel à rédiger devrait permettre d'enregistrer pas à pas les payements effectués (sur base des extraits de banque reçus) et de fournir en temps opportun, des extraits de compte pour chaque participant ou tout au moins pour ceux pour lesquels il existe un écart significatif entre les montants dus et payés. Enfin, il serait souhaitable que le logiciel fournisse, à la demande du gestionnaire, une balance globale des payements attendus et déjà reçus.

### 5.3.1 Analyse des besoins

Même si notre approche se revendique du paradigme objet, il nous semble indispensable d'initier le processus de développement par une phase d'analyse ... fonctionnelle. On ne rédige pas du logiciel sans une idée précise de la tâche à automatiser ou alors on risque très fort de s'engager dans des développements qui sont non seulement inadéquats par rapport au projet mais qui n'ont pas plus de "chance" pour autant d'être "réutilisés" dans d'autres circonstances.

Dès lors, la première phase du processus de conception doit être celle de la recherche des fonctionnalités à assurer. Dans l'esprit du modèle orienté objet qui vient d'être présenté, il s'agit donc de mettre en évidence les acteurs qui interviennent, de déterminer pour chacun d'eux les événements qui leur sont associés et de les décrire au moins sommairement. Pour s'aider à découvrir les événements signalons qu'ils seront généralement décrits par une expression de la forme "Demande de <action ou résultat >".

Dans l'exemple qui nous occupe, il apparaît que le seul acteur à intervenir est le professeur `Gestionnaire` du voyage. Parmi les événements, on peut relever (la liste n'est pas exhaustive) :

- `D_Inscription_Part`[1] : Demande d'enregistrement de l'inscription d'un participant (en principe nouveau c'est-à-dire non encore enregistré).
- `D_Enregistre_Paye` : Demande d'enregistrement d'un payement effectué par un participant (ou son responsable légal) ou éventuellement d'un remboursement effectué par le gestionnaire du voyage (suivant le signe du montant).
- `D_Extraits_Cpte` : Demande d'impression des extraits de compte relatifs aux participants (la sélection des participants concernés doit être effectuée via un écran de dialogue).
- `D_MAJ_Tarifs` : Demande de mise à jour du tarif (par défaut) des frais de voyage. Cet événement n'a lieu, en principe, qu'une fois par an au moment de commencer la gestion d'un nouveau voyage.
- (...)

### 5.3.2 Conception de la base de données

La tâche la plus délicate de la conception orientée objet réside dans la mise en évidence et le choix des "bonnes" classes pour structurer l'application. Comme le souligne Bertrand Meyer, il n'existe pas de technique universelle et infaillible. Aussi "le talent et l'expérience sont des parties inévitables du succès de la conception "[MEYER, 387]. Certains principes généraux peuvent toutefois aider considérablement à progresser efficacement dans la recherche des "bonnes" classes.

Une "recette" particulièrement intéressante nous paraît être la conception de la base de données associée à l'application. Cette conception nous permettra en effet d'isoler et de décrire les entités du monde réel qui doivent être considérées dans le projet que l'on analyse.

La méthode de conception que nous retiendrons ici et qui est elle-même structurée en plusieurs étapes successives est largement inspirée de la méthode que nous avons élaborée dans le cadre des formations du CeFIS et qui est présentée dans [DELACHARLERIE a&b].

---

[1] Les noms des événements (comme de tous les autres concepts) sont arbitrairement limités à 20 caractères dans le logiciel SACOO.

La méthode de conception proposée au CeFIS est fondamentalement basée sur des formes un peu simplifiées du modèle conceptuel Entité-Association et du modèle Relationnel. Ce dernier permet en effet d'obtenir une description de la base de données qui soit facilement implémentable via un système de fichiers au format dBASE. Or le public que nous visons est souvent familier de ces deux modèles; nous pensons donc qu'une approche ayant une certaine cohérence avec ces modèles a toute chance de séduire plus facilement le public de développeur qui nous intéresse. Par ailleurs, il ne faut pas perdre de vue que nous désirons concevoir "orienté objet" du logiciel qui devra cependant, au niveau de l'implémentation, utiliser en une base de données relationnelles ou "orientée" fichiers telle que dBASE.

L'objectif que nous poursuivons lors de la conception de la base de données est donc double : il faudra à la fois imaginer une structure de fichiers ou de tables relationnelles qui soit bien adaptée au projet étudié et à la fois décrire un ensemble structuré de classes qui constitueront une bonne base pour le développement de l'application, mais qui permettront un "mapping" aisé entre les structures "objet" et les structures "relationnelles".

Pour cette phase de conception de la base de données, nous nous permettons de donner une illustration de la méthode employée sur un exemple un peu plus complexe que celui du voyage scolaire et ce afin de présenter un plus grand nombre de situations typiques qui pourraient être rencontrées. L'exemple que l'on considérera a toujours trait au monde scolaire et consiste à modéliser de façon, certes un peu simpliste, la structure d'une école composée de professeurs qui donnent des cours selon un certain horaire à des élèves qui sont regroupés en classes et qui peuvent également s'inscrire à des activités extra-scolaires.

Rappelons que notre but, dans cette phase de la conception est double : d'une part mettre en évidence la structure de la base de données qui sera associée au logiciel en cours de conception et d'autre part préparer cette même conception en relevant déjà une certain nombre de classes d'objets qui seront très probabalement indispensables par la suite. Toutefois, même si l'on peut déjà imaginer certaines des opérations qui devront être associées à ces classes, nous ne les décrirons pas ici et nous nous concentrerons essentiellement sur la description statique des classes mises en évidence.

### *a. Graphe des éléments informationnels*

La première étape de construction du schéma de la base de données consiste, comme dans pratiquement toute méthode, à faire un relevé aussi soigneux et complet que possible des informations qui sont considérées comme significatives et utiles pour la gestion. Ce relevé s'effectue au travers de l'analyse des documents écrits de l'organisation et des interviews des gestionnaires. Il consiste à détecter les entités et les informations qui devront (probablement) être intégrées au schéma. De façon très intuitive, il s'agit de répondre à la question "De quoi est-il question ?".

Afin de préparer déjà la structuration des éléments mis en évidence, il est judicieux de les organiser immédiatement en un graphe (plus ou moins informel) où les sommets sont constitués par les entités ou informations repérées et les arêtes relient les éléments qui sont manifestement associés (quelle que soit la nature du lien d'association) comme illustré à la figure figure 5.6.

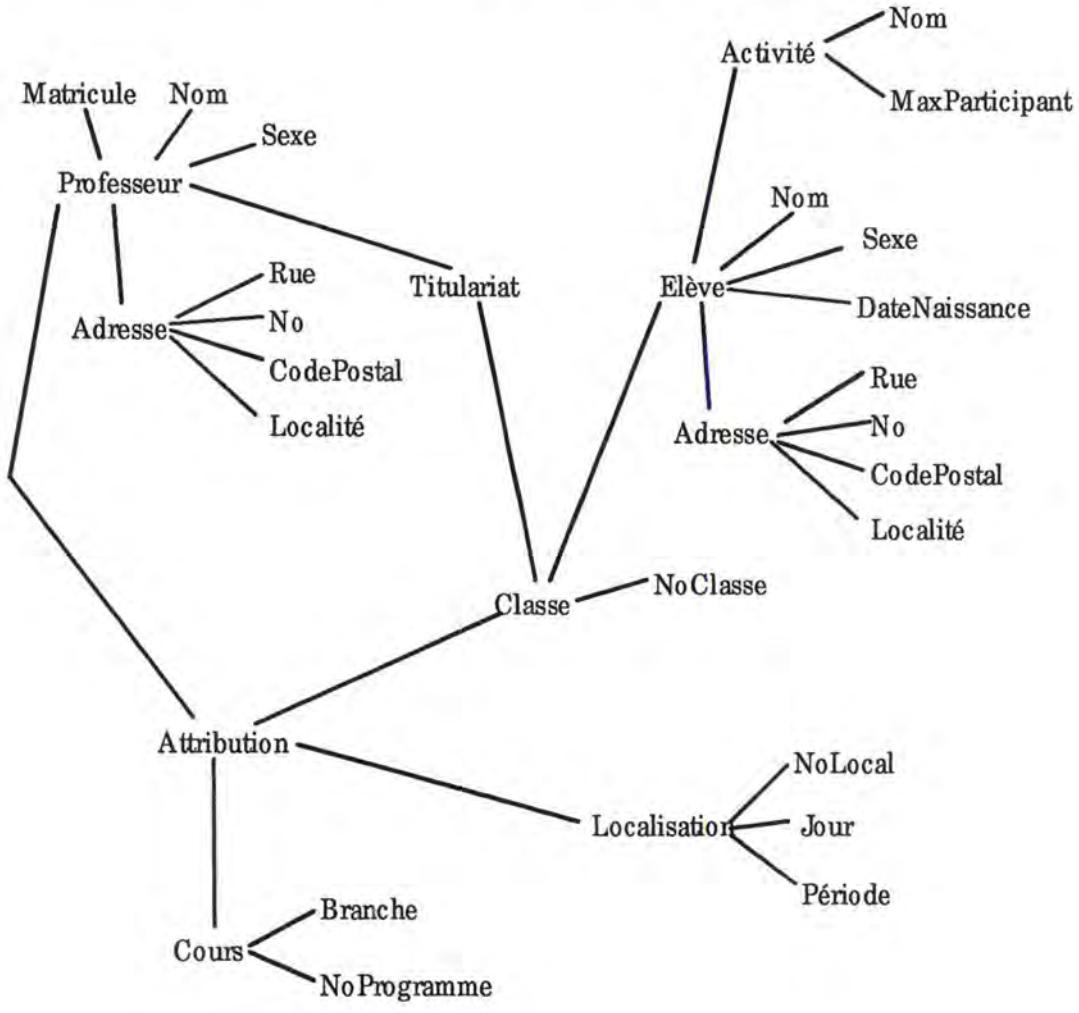



Figure 5.6 : Graphe des éléments informationnels

Il est intéressant d'observer qu'à ce stade la distinction entre entité et information n'est pas toujours évidente. Ainsi, dans notre

exemple, nous associerons une information Adresse à chaque élève puis nous décrirons l'Adresse comme associée à une Rue ou une Localité. Alors Adresse désigne-t-elle une information ou une entité ? Aussi, nous préfèrerons parler, à ce stade, d'élément informationnel.

Pour que le graphe soit utile à la suite de la conception, il est toutefois indispensable de respecter une règle de non-redondance : un même élément informationnel ne peut apparaître qu'une seule fois dans le graphe et doit donc être relié par des arêtes à tous les éléments qui lui sont associés.

Si l'on voulait être plus formel et précis, il conviendrait de donner à chaque information "distincte" un nom distinct. Ainsi par exemple il faudrait noter Nom_Professeur, Nom_Elève ou encore Nom_Activité et ainsi de suite pour tous les éléments d'information repérés. La règle de non-redondance s'exprimerait alors facilement en énonçant qu'un même nom d'élément informationnel ne peut se trouver deux fois dans le graphe.

Il existe pourtant au moins deux raisons de ne pas obliger le concepteur à attribuer des noms "entièrement" identifiants aux informations détectées. La première raison est celle de la simplicité et de l'évidence : si Nom est associé à Professeur, il est évident qu'il s'agit du nom du professeur. Cette raison est d'ailleurs cohérente avec le principe de l'orientation objet qui veut que les identifiants soient locaux à la classe considérée. La seconde raison, plus déterminante encore est qu'il ne faut pas entraver le processus de généralisation qui doit permettre de dégager des classes plus génériques. Si tous les éléments informationnels portent des noms distincts, plus aucune structure générique ne peut apparaître.

### *b. Schéma global Classe - Héritage - Association*

La seconde étape consiste bien entendu à raffiner le graphe des éléments informationnels dans le but de mettre en évidence les objets (ou plus exactement les classes d'objets) avec leurs attributs ainsi que les liens d'héritage et d'association. Pour ce faire, nous allons à nouveau mettre en évidence quelques règles simples qui peuvent aider considérablement le concepteur et que l'on peut considérer comme autant de sous étapes.

#### Mise en évidence des classes

Tout d'abord, il s'agit de considérer que tous les sommets du graphe qui ne sont reliés au reste du graphe que par une seule arête sont (vraisemblablement) des attributs et qu'à l'inverse, tout sommet qui forme un "noeud" du graphe, c'est-à-dire qui est relié à plusieurs autres et qui comporte au moins un attribut peut être

considéré comme un bon candidat pour devenir une classe du futur schéma. De façon très intuitive, il convient donc de surcharger le graphe des éléments informationnels en entourant les classes repérées et leurs attributs (lignes grises de la figure 5.7).

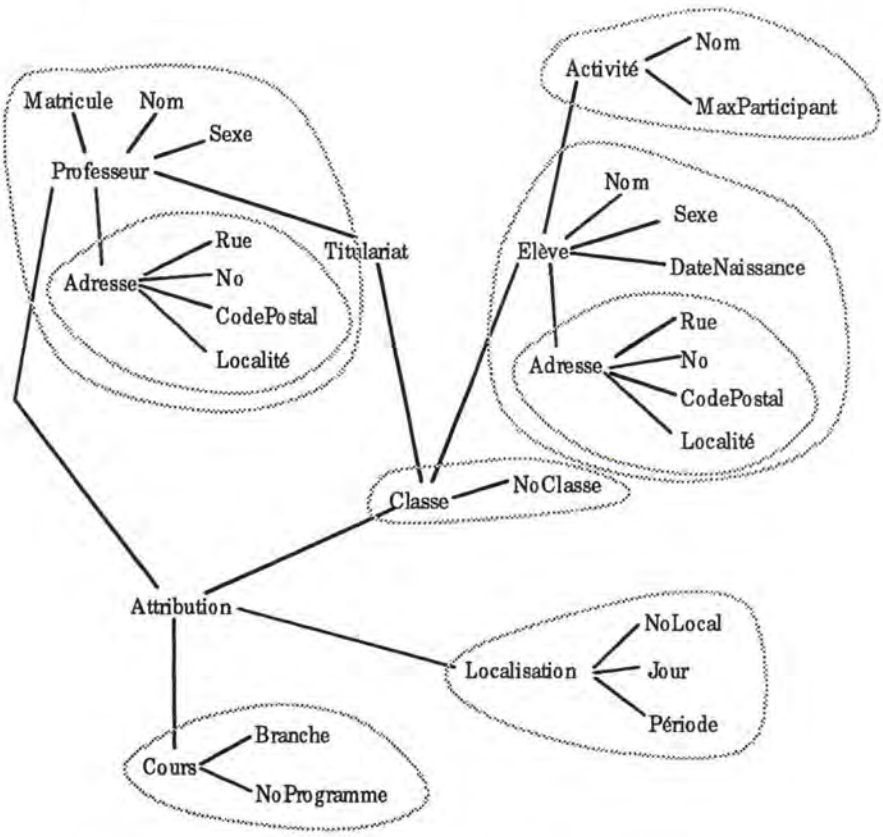



Figure 5.7 : Relevé des classes dans le graphe des éléments informationnels

**Mise en évidence des structures d'héritage**

Ensuite, il faut analyser le graphe ainsi surchargé pour y rechercher des structures identiques ou fort semblables afin de les généraliser. Ainsi, dans notre exemple, on retrouve évidemment le groupe `Adresse` qui est repris deux fois de façon identique et qui sera certainement généralisé en une classe autonome. De même, on repère également que les groupes `Professeur` et `Elève` sont très semblables : il y a donc lieu de les généraliser en une nouvelle classe que nous nommerons `Personne` et qui sera caractérisé par un `Nom`, un `Sexe`, et une `Adresse`. Les classes `Professeur` et `Elève` pourront ainsi hériter de cette classe `Personne` tout en s'en distinguant par au moins un attribut spécifique mais aussi par des associations différentes.

**Représentation des classes**

Une première version du schéma des classes peut à ce moment être élaboré en représentant les classes repérées avec leurs attributs ainsi que les éventuelles relations d'héritage. Pour bien mettre en évidence les sommets ou arêtes qui n'ont pas encore été pris en compte, il peut s'avérer pratique de barrer d'une croix

tous les éléments (aussi bien sommets qu'arêtes !) qui sont déjà représentés dans notre ébauche de schéma.

**Mise en évidence et représentation des "liens" entre classes**

Il reste dès lors à analyser la nature "profonde" de chacun des éléments restant et à découvrir s'ils doivent être représentés par un nouvel objet, (cas de `Attribution`), par un nouvel attribut avec lien de composition simple ou multiple (cas de l'arête `Attribution-Localisation`) ou par une "association" (cas de `Titulariat` ou de l'arête `Classe-Elève`).

> Le schéma Classe-Héritage-Association est en fait une représentation quelque peu hybride car elle tient à la fois du modèle orienté objet (notions de classe et d'héritage) mais aussi du classique modèle Entité-Association auquel elle emprunte le concept d'association pour représenter le fait que deux classes sont liées par une certaine relation et qu'elles peuvent donc se référencer l'une l'autre. Le concept d'association ne peut toutefois pas être utilisé pour représenter le lien de composition entre classes.

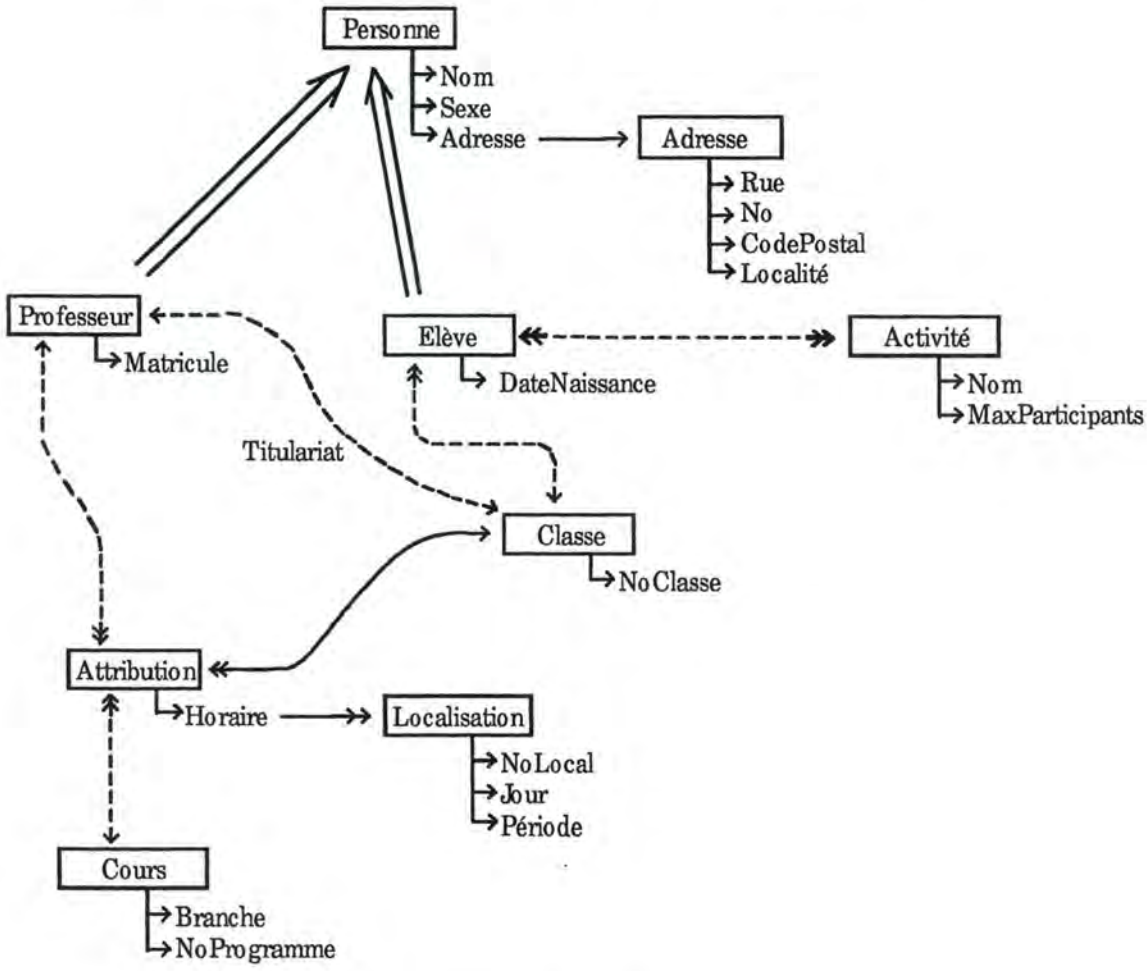



Figure 5.8 : Schéma Classe-Héritage-Association

Attention toutefois : il est très important de ne pas introduire, à ce stade, d'attributs références, mais de les représenter par des arcs d'association dont chaque extrémité comporte une simple ou double flèche suivant la multiplicité de l'association.

Par référence au modèle Entité-Association, signalons que la simple flèche est équivalente a une connectivité 0-1 ou 1-1 et que la double flèche représente une connectivité 0-N ou 1-N.

Le schéma obtenu est alors, pour notre exemple celui de la figure 5.8.

> Il va de soi que l'élaboration de ce schéma ne s'effectue pas toujours de façon aussi "linéaire" que nous l'avons montré ici et qu'il est bien souvent nécessaire de modifier et d'améliorer le schéma par petites touches progressives jusqu'à ce que la représentation qu'il donne de la réalité soit jugée satisfaisante. De toute façon, il existe pratiquement toujours plusieurs représentations possibles pour la même réalité qui ne sont peut être pas strictement équivalentes mais qui permettent de mieux mettre en lumière ou de gérer plus facilement telle ou telle portion du schéma. Comme toujours, il convient, a priori, de préférer les solutions les plus "génériques" même si, plus loin dans la conception il pourra être décidé de se rabattre sur une solution moins générique, mais plus facile à implémenter ou plus efficace.

### *c. Schémas individuels des classes*

Sur base du schéma global qui vient d'être élaboré, il devient extrêmement simple , dans cette troisième étape, de déduire les schémas individuels de chaque classe. Il suffit en effet de reprendre chacune des classes qui ont été isolées et d'en raffiner et compléter la description.

Les attributs déjà mis en évidence sont évidemment repris et sont complétés par l'indication de leur type ainsi que de leur caractère obligatoire ou facultatif. Les associations auxquels ils participent sont représentées par des attributs références pour lesquels il faut créer un nom, mais dont le type et la multiplicité sont immédiatement déduits du schéma global. Le cas échéant, la relation d'héritage est conservée. Pour notre exemple, on aboutit ainsi au schéma de la figure 5.9.

### *d. Schéma relationnel de la base de données*

N'ayant pas à notre disposition de SGBD qui accepte directement une description orientée objet de notre système d'informations, il est nécessaire d'effectuer encore une transformation pour obtenir un schéma de base de données implémentables sur un SGBD existant. Nous nous intéresserons au modèle relationnel puisque celui-ci tend aujourd'hui à devenir très populaire et que le "mapping" vers un système de fichiers tel que celui employé par dBase est pratiquement immédiat.

Les objectifs que l'on poursuit avec la modélisation objet d'une part et la construction de la base de données, d'autre part, sont en fait assez contradictoires. En effet, selon le point de vue OO, il convient de distinguer un maximum d'entités significatives et de les décrire en tant que classe, quitte à les combiner par la suite à l'aide des mécanismes d'héritage et de composition. Ainsi dans notre exemple, le concept de `Professeur` est finalement éclaté dans les trois classes `Adresse`, `Personne` et `Professeur`. Du point de vue BD par contre, il importe finalement de minimiser les accès physiques pour restituer une information demandée. Au contraire d'un éclatement, il importe donc ici de regrouper au maximum les informations qui peuvent l'être et de minimiser ainsi le nombre de fichiers consultés. Nous devons donc imaginer une procédure de dérivation d'un schéma relationnel (ou fichiers) tel que ce dernier permette un mapping aisé avec les schémas individuels des classes.

Les règles que nous proposons sont les suivantes. Elles nécessitent absolument de repartir du schéma global Classe-Héritage-Association et doivent être conduites successivement comme autant de sous étapes dans la démarche.

On peut en effet montrer que la transformation vers les schémas individuels des classes s'effectue avec une (légère ?) perte d'information. Lorsque les associations sont représentées par des attributs références : la multiplicité de la classe référencée vers la classe référençante est en effet perdue. Par exemple, dans la classe `Activité`, on ne peut plus dire si un `Elève` est lié à une ou plusieurs `Activité(s)`.

1. Commencer par regrouper les classes qui sont attribut composant simple avec la classe "composée" qui les contient.
   Dans notre exemple, il y a donc lieu de regrouper les classes `Personne + Adresse` en une seule entité.
2. Regrouper ensuite les classes (ou groupes) faisant partie d'une hiérarchie d'héritage.
   Toujours dans notre exemple, il convient donc de regrouper d'une part, `Professeur + (Personne + Adresse)` et `Elève + (Personne + Adresse)`.
3. Décrire alors chaque groupe ou chaque classe ne faisant pas partie d'un autre groupe comme une table relationnelle en "mettant à plat" tous les attributs.
   A ce stade, par exemple, on obtient entre autres les deux tables suivantes :

```
Professeur (Matricule, Nom, Sexe, Rue, No, Code Postal, Localité)
Cours (Branche, NoProgramme)
```

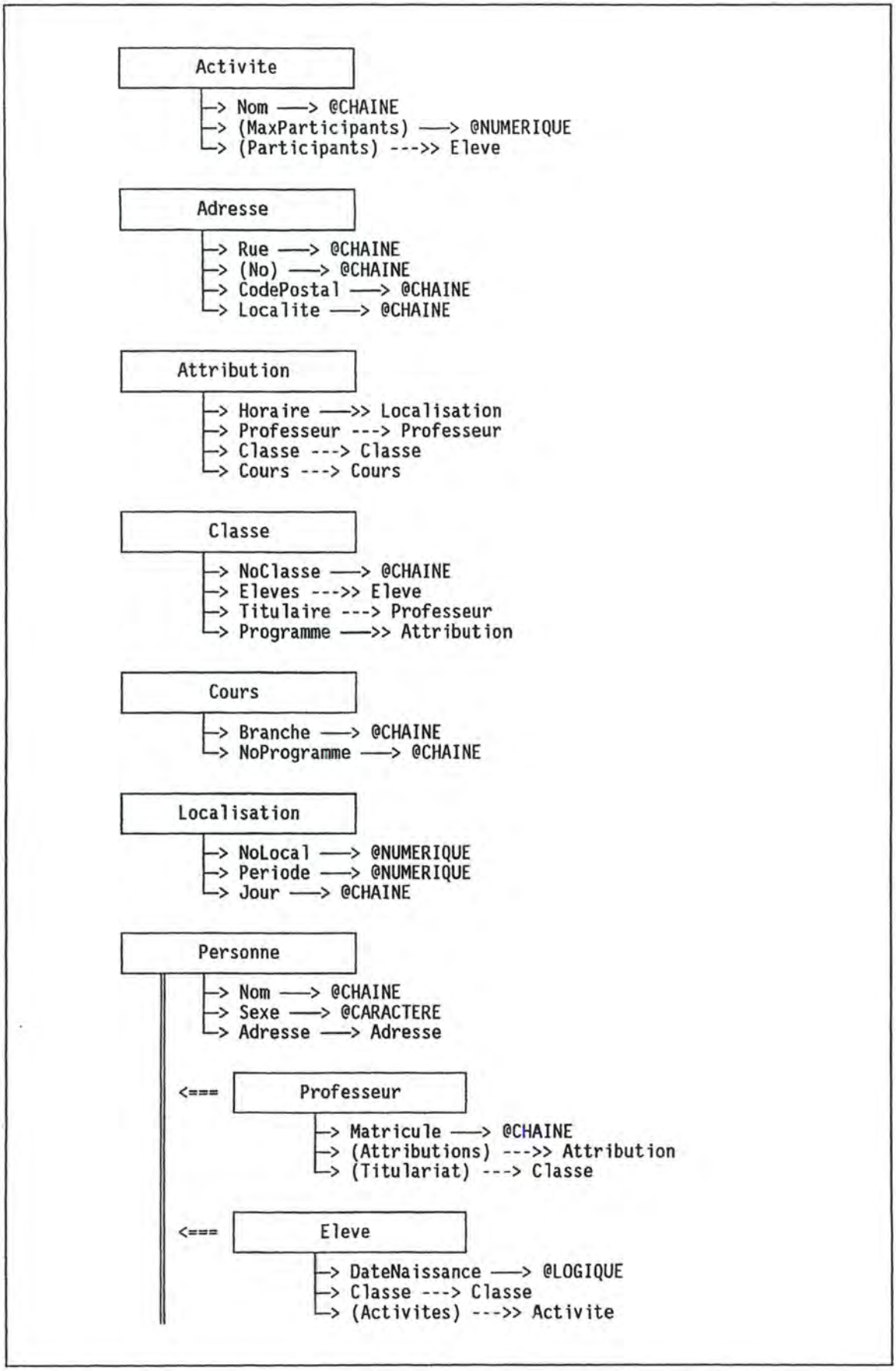



Figure 5.9 : Schémas individuels des classes

4. Ajouter alors d'office à chaque table un attribut technique qui jouera le rôle de clef primaire. Par convention, cet attribut sera dénommé `Identif`.

   *C'est en effet une pratique très courante en BD OO que d'affecter d'autorité un identifiant à chaque objet manipulé à l'inverse du modèle relationnel qui se base sur les valeurs des attributs "effectifs" pour construire la clef primaire. Les inconvénients liés à l'utilisation d'informations significatives pour l'identification des tuples d'une table sont suffisamment connus pour ne pas être rappelés ici.*

   *L'ajout de l'identifiant technique ne doit pas pour autant occulter le fait qu'il doit continuer à exister une clef primaire "naturelle" à chaque table et qu'il convient donc de relever au moins une contrainte d'identification sur chaque table. Ainsi, il convient de relever la contrainte qui affirme par exemple qu'il ne peut exister deux tuples de `Professeur` avec la même valeur de `Matricule`.*

5. Représenter alors les attributs composants multiples en ajoutant dans la table de la classe composante un nouvel attribut destiné à recevoir l'identifiant du tuple correspondant de la table représentant la classe composée.
   Dans notre exemple toujours, il y a donc lieu d'ajouter à la table `Localisation` un nouvel attribut `Id_Attribution` qui désignera l'attribution à laquelle se rapportera la localisation.
6. Enfin, il reste à représenter les associations et il convient de distinguer trois cas.

   1er cas : association de type `A <---->> B` (1 - N)

   On est en fait dans une situation très similaire à celle de la représentation des attributs composants multiples et le même type de solution doit être appliqué. Il faut donc ajouter à la table B un attribut qui contiendra les identifiants des tuple correspondants de la table A.

   Dans notre exemple, il convient, entre autres, d'ajouter dans la table `Elève` un attribut `Id_Classe`.

   2e cas : association de type `A <-----> B` (1 - 1)

   On a ici une version simplifiée du 1er cas et l'on peut choisir soit d'ajouter du attribut `Id_B` dans la table A, soit `Id_A` dans la table B, soit encore de combiner les deux solutions mais en veillant alors à maintenir l'intégrité du système.

   Dans notre exemple, c'est le cas de l'association `Titulariat` que l'on représentera par un attribut `Id_Titulaire` dans la table `Classe`. Cette solution est ici préférée car on sait qu'il existe un professeur titulaire pour chaque classe alors que tout professeur n'est pas nécessairement titulaire d'une classe.

   3ème cas : association de type `A <<--->> B` (N - M)

Impossible cette fois d'ajouter simplement un attribut dans l'une ou l'autre table. Il faut donc créer une nouvelle table spécifique avec deux attributs `Id_A` et `Id_B` pour enregistrer toutes les références nécessaires. Dans notre exemple, c'est le cas de l'association qui lie `Elève` et `Activité`. Elle sera donc représentée par une nouvelle table :
`Participation (Id_Elève, Id_Activité)`

**Professeur** (Identif, Matricule, Nom, Sexe, Rue, No, CodePostal, Localité)

**Elève** (Identif, DateNaissance, Nom, Sexe, Rue, No, CodePostal, Localité, *Id_Classe*)

**Classe** (Identif, NoClasse, *Id_Titulaire*)

**Cours** (Identif, Branche, NoProgramme)

**Localisation** (Identif, Jour, Période, Local, *Id_Attribution*)

**Attribution** (Identif, *Id_Cours*, *Id_Professeur*, *Id_Classe*)

**Activité** (Identif, Nom, MaxParticipants)

**Participation** (*Id_Elève*, *Id_Activité*)

Figure 5.10 : Schéma relationnel

Les six règles ci-dessus peuvent bien évidemment être complétées par quelques "réglages" plus fins pour améliorer encore le schéma relationnel obtenu. Une telle "astuce", que nous avons mise en pratique dans la base de données associée à l'outil d'assistance à la conception que nous présenterons au chapitre suivant, consiste à rassembler toutes les tables représentant des associations N-M (lorsqu'il y en a plusieurs) en une seule et à ajouter, bien entendu, un nouvel attribut pour distinguer à quelle table "logique" appartient un tuple donné. Cette astuce permet de limiter ainsi le nombre de fichiers nécessaires mais aussi celui des index.

Le schéma relationnel final correspondant à notre exemple est donc celui de la figure 5.10:

> On observe facilement en comparant le schéma relationnel avec l'ensemble des schémas individuels des classes qu'il n'y a pas de correspondance un à un entre eux, tantôt une table correspond à plusieurs classes, tantôt c'est au contraire une seule classe qui est représentée par plusieurs tables. Toutefois, les règles de correspondance sont assez évidentes : il suffira de les implémenter au travers de la conception de l'application associée.

Pour retrouver notre exemple du voyage scolaire, la figure 5.11 donne les schémas individuels des classes qui ont été mis en évidence.

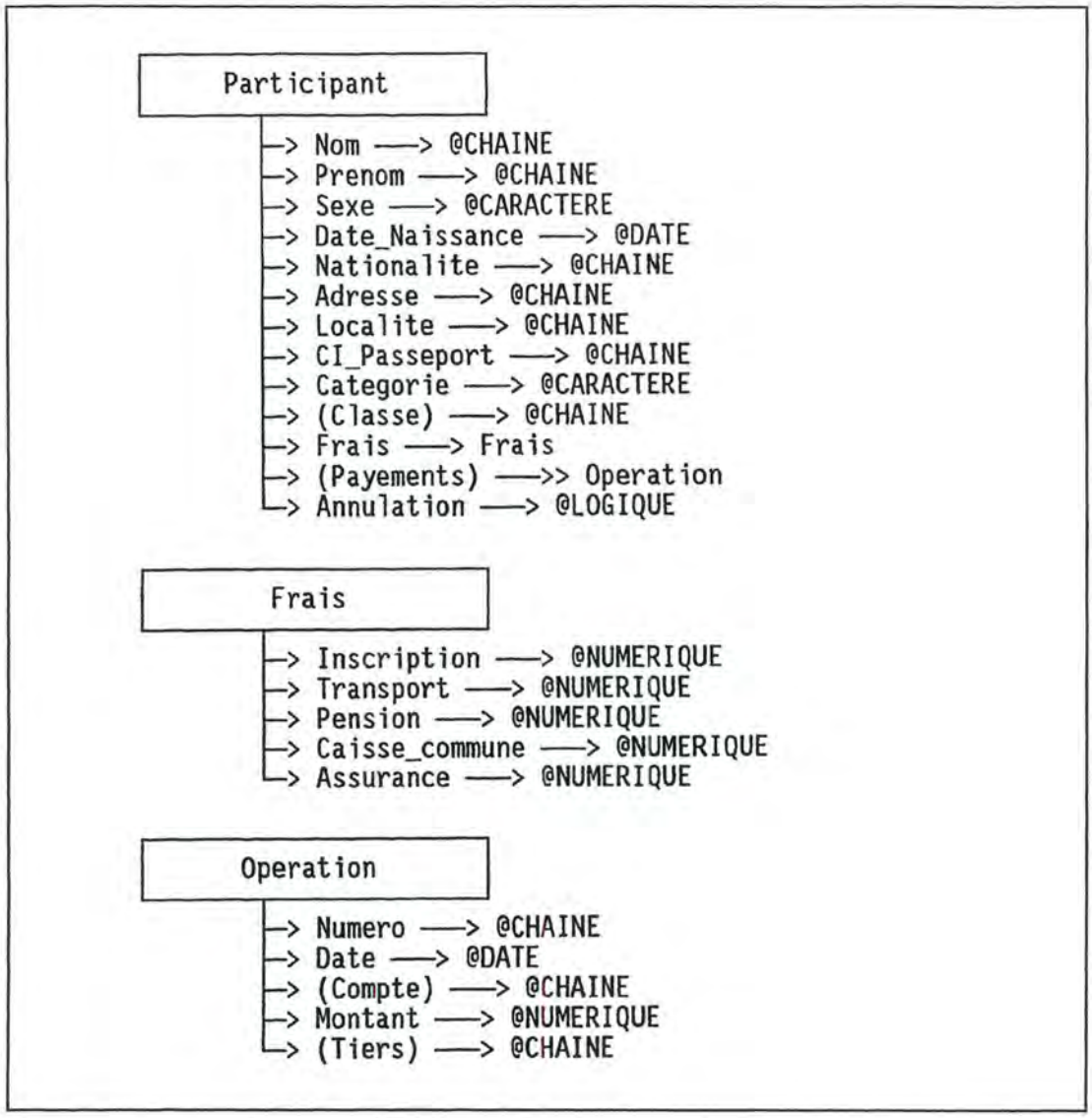



Figure 5.11 : Schémas individuels des classes de la BD "Voyage scolaire"

### 5.3.3 Elaboration du scénario de l'interface

*Signalons d'emblée que la présente phase du processus de conception peut éventuellement être conduite en parallèle avec la phase de conception de la base de données voire avant cette dernière car les interactions entre ces deux phases sont nulles ou négligeables.*

Cette phase de la conception ne doit surtout pas être négligée car elle conditionne considérablement la convivialité de produit final. Or précisément si, il y a 5-10 ou 15 ans le programmeur se préoccupait peu de l'interface qu'il offrirait à l'utilisateur (ce dernier subissant généralement la logique de l'application), il est aujourd'hui indispensable d'offrir une interface soignée à un utilisateur qui devient (et il a parfaitement raison) de plus en plus exigeant. Nous ne nous

étendrons pas ici sur les avantages d'une interface de qualité que sont l'apprentissage plus rapide, le gain de productivité, la diminution des erreurs...

Bref, il nous paraît inconcevable, aujourd'hui, de se lancer dans la rédaction d'une application sans une idée relativement précise de l'organisation de l'interface utilisateur que l'on compte proposer.

> Cette phase de conception de l'interface peut aussi être mise à profit (au même titre que la phase précédente) pour effectuer un contrôle auprès du client du projet afin de vérifier l'adéquation de la solution imaginée avec le "problème" posé. Le cas échéant, des ajustements peuvent être effectués avant que la phase la plus lourde de conception proprement dite ne soit entamée.

Ayant avant tout concentré notre effort, dans le cadre de ce mémoire, sur la conception des traitements et de la BD, nous ne pouvons guère donner ici de "recettes" (miracles !?) pour organiser la phase d'élaboration du scénario de l'interface.

Cependant, le terme même de scénario est important. En effet, plus que de décrire minutieusement le layout des différents écrans menus, boîtes de dialogues..., il nous semble crucial de réfléchir à la dynamique de succession de ces différents éléments de dialogue avec l'utilisateur en ayant constamment à l'esprit l'objectif d'offrir un maximum de souplesse.

Il s'agit donc, dans une optique orientée objet, de mettre en évidence les différents éléments de dialogue nécessaires et de construire un graphe orienté dont les sommets sont constitués par les éléments de dialogue et dont les arcs traduisent un enchaînement possible.

> Insistons bien ici sur le fait qu'il s'agit de décrire l'enchaînement "visuel" potentiel des éléments de dialogue et non pas un quelconque mécanisme d'appel d'un élément par un autre ou encore de décomposition d'un élément en sous-éléments.

Faute d'une réflexion approfondie sur le sujet, nous ne nous aventurerons cependant pas à proposer un formalisme précis pour modéliser l'interface utilisateur, c'est-à-dire les éléments de dialogue et leur dynamique d'enchaînement.

> Un exemple informel de scénario d'interface pour la fonctionnalité de gestion des payements de notre projet illustratif du voyage scolaire est cependant donné à la figure 5.12.

Au terme de l'élaboration du scénario, il convient alors d'examiner celui-ci en cherchant cette fois à faire apparaître les objets qui seront nécessaires pour "incarner" les éléments de dialogue, à les regrouper en classes et, le cas échéant, à détecter si certaines classes peuvent être généralisées pour faire apparaître des superclasses. Aux rangs de celles-ci, on a toute chance de retrouver les grands

classiques du dialogue homme-machine que sont les classes `Fenêtre`, `Menu`, `Champ_saisie`... voir, si l'on compte mettre en place une interface graphique de type "Windows", `Bouton`, `Bouton_radio`, `Barre_défilement`, `Icône`...

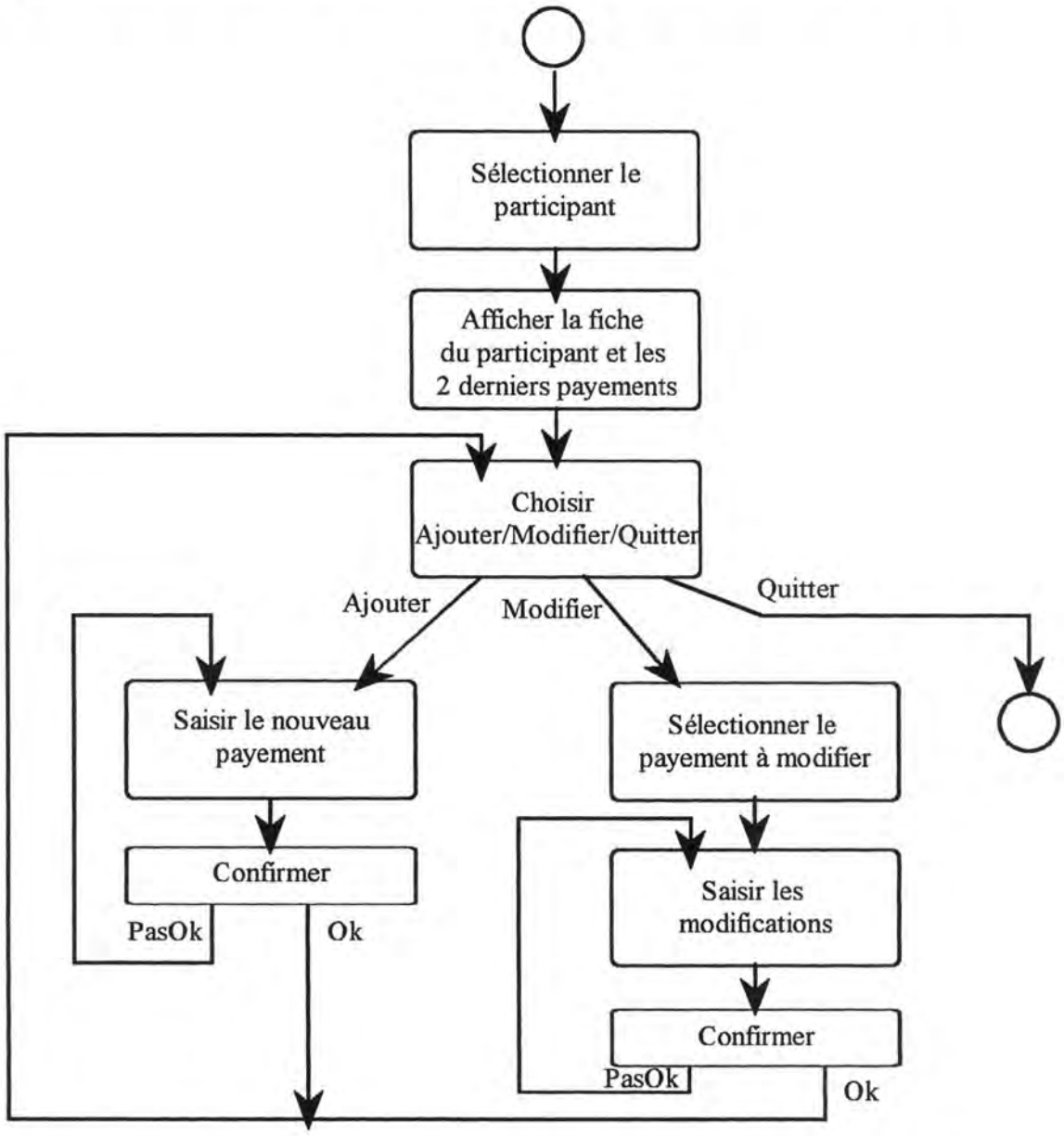



Figure 5.12 : Scénario d'interface de la fonctionnalité "Gérer les payements"

Fort heureusement pour le concepteur, beaucoup de ces classes pourront avantageusement être "réutilisées" d'un projet à l'autre et ne devront donc être analysées et décrites qu'une seule fois ou même pourront être directement "achetées" à l'extérieur. En effet, si dans les cinq dernières années, on a pu voir de très nombreuses firmes mettre sur le marché des bibliothèques de fonctions destinées à prolonger les bibliothèques standard comme celles du compilateur Clipper (et plus largement des compilateurs C, Pascal, ...), il est

hautement probable que ces bibliothèques seront progressivement remplacées dans les années à venir par des bibliothèques de classes prêtes à être instancées ou héritées.

### 5.3.4 Conception de l'application

Le moment est venu à présent de nous attaquer à la conception de l'application proprement dite c'est-à-dire de mettre en évidence les classes d'objets nécessaires et de décrire tant les informations qui doivent être mémorisées (attributs) que les services qui doivent être fournis (opérations). Pour guider le concepteur dans ce travail, nous allons d'abord présenter en détail le principe de la méthode proposée puis nous donnerons une piste pour structurer l'architecture du logiciel à concevoir et enfin, nous terminerons par une série de "tuyaux" c'est-à-dire d'heuristiques ou de conseils divers pour faciliter le travail de conception.

***a. Principe de la méthode***

Rappelons-nous tout d'abord que l'objectif même du processus de développement consiste à élaborer une solution en passant progressivement de l'expression des besoins vers un programme qui répond à ces besoins. On sait toutefois que ce processus n'est pas strictement linéaire et qu'il est tout à fait normal de devoir effectuer des retours en arrière pour corriger ou compléter des parties de solution déjà élaborées. Aussi, il est très important de pouvoir imaginer et construire les éléments de la solution sans devoir en rédiger directement le code. Il faut seulement dans un premier temps décrire la nature du service attendu (spécification) : c'est ce que nous ferons dans la définition des opérations. Dès lors, toute opération ainsi spécifiée existe déjà virtuellement et peut être utilisée par d'autres opérations. Dans un second temps, il faudra étudier chaque opération définie et imaginer une stratégie de solution qui, très probablement nécessitera l'appel d'autres opérations offertes par la même classe ou par d'autres, et qui devront le cas échéant être définies. Cette stratégie de solution sera décrite dans le script de l'opération.

> La notion de script est informelle. Le script permet de donner les grandes lignes de l'algorithme qu'il faudra rédiger en mettant tout spécialement en évidence les appels à d'autres opérations. On peut donc voir le script comme une version plus abstraite de l'algorithme qui ne s'intéresse pas aux "détails" que sont, entre autres, le choix précis des structures (boucles par exemple) ou la définition des variables locales nécessaires.

Enfin, plus tard, lors de la phase d'implémentation, il restera à rédiger effectivement les algorithmes des opérations. Dans le cas où l'algorithme d'une opération est manifestement simple, on peut admettre que l'étape de rédaction du script soit escamotée et que l'algorithme soit directement rédigé.

> A bien y regarder on retrouve ici, derrière les trois étapes définition - script - algorithme, les trois jalons de base de toute démarche de

programmation, que Charles Duchâteau illustre par les questions "Quoi faire ?", "Comment faire ?" et "Comment faire faire ?" [DUCHATEAU].

L'idée générale de la méthode est donc d'adopter une démarche globalement descendante qui parte donc des fonctions à assurer vers les objets qui offriront les services correspondant à ces fonctions.

La conception doit donc s'effectuer par paliers successifs englobant l'analyse d'une ou plusieurs classes. Chaque palier de conception peut être décomposé comme suit :

1. Analyse de la spécification des services attendus.
2. Recherche des classes nécessaires pour assurer chaque service attendu. Pour ce faire, il faut chercher :
   - à utiliser des services déjà offerts par des classes existantes (éventuellement décrites dans d'autres projets);
   - à ajouter des services nouveaux à des classes existantes;
   - à spécialiser (éventuellement généraliser) des classes existantes et à redéfinir certains services;
   - à créer de nouvelles classes avec leurs services propres.

   L'ordre des quatre propositions ci-dessus n'est bien entendu pas sans importance : il vise naturellement à promouvoir au maximum la réutilisation.
3. Rédaction du script de chaque opération (service) en précisant clairement les appels à d'autres opérations de la même classe ou d'autres classes.

Bien entendu, au tout premier niveau, les services attendus de l'application sont tout simplement les événements qui peuvent se produire soit par l'initiative (externe) d'un acteur soit éventuellement par génération interne.

Notons encore que, méthodologiquement, il semble préférable d'effectuer complètement la conception des classes d'un palier avant de s'engager dans l'analyse du palier suivant de façon à mieux raffiner les spécifications des classes/opérations du niveau inférieur. Ce sera surtout utile lorsqu'un élément est utilisé par plusieurs autres et doit donc être conçu pour s'adapter à différentes situations.

***b. Architecture de l'application***

La conception et surtout la maintenance et l'évolution d'une application peuvent être sérieusement facilitées si l'ensemble des classes constituantes a été convenablement architecturé. Ainsi, il nous semble utile de regrouper les classes mises en évidence en plusieurs "modules" ayant chacun une certaine cohérence interne. Une solution relativement classique consiste à baser l'architecture des applications sur un système de couches. Nous proposons plutôt une architecture avec (au moins) quatre modules correspondant à quatre types distincts de

problèmes à résoudre (figure 5.13). Chaque module regroupera un certain nombre de classes.

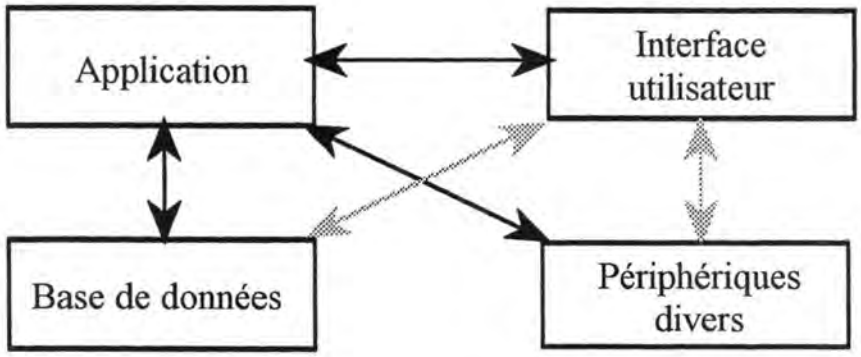



Figure 5.13 : Architecture générique proposée

- Module **Application** :
  Ce module regroupera les classes vraiment spécifiques de l'application. C'est de ce module que partiront la plupart des appels à des services "techniques" vers les autres modules.
- Module **Base de données** :
  Ce module contiendra la où les classes ayant en charge d'assurer le stockage des informations.
- Module **Interface utilisateur** :
  Ce module comprendra les classes chargées d'assurer l'affichage des informations à l'écran et la saisie des interactions émanant de l'utilisateur.
- Module **Périphériques divers** :
  Ce module permettra entre autres de regrouper les classes chargées d'assurer l'impression de documents, la transmission d'informations via réseau, ...

L'intérêt du regroupement des classes en plusieurs modules ayant chacun un objectif précis n'est pas seulement académique, il apporte des avantages qui ne peuvent être passés sous silence. Entre autres, il apporte une aide à la construction des classes et à la définition des services à fournir en donnant un critère de répartition de ces services. Dans une optique de réutilisation, il est judicieux en effet de ne pas "mélanger" dans une même classe des services relatifs à la gestion des informations sur disque avec ceux relatifs à leur présentation à l'écran. Le critère de base du paradigme objet énonçant qu'il faut rassembler dans une même classe les services ayant trait à une même structure de données n'est donc pas toujours suffisant et doit parfois être complété en ajoutant qu'il faut éviter de rassembler dans une même classe des services faisant appel à des technologies trop différentes.

Idéalement, la structuration des classes dans les différentes modules de l'architecture devrait être telle que le changement d'une technologie (par exemple de base de données) n'ait de répercussions que dans le module en charge de cette technologie et soit totalement transparent du reste de l'application. Bien entendu, ces principes ne peuvent pas toujours être respectés à la lettre tout spécialement

avec des projets de petite taille (comme l'exemple du voyage scolaire) car ils conduiraient à une multiplication abusive du nombre de classes.

Il convient donc de rechercher en permanence le meilleur équilibre entre la rigueur de l'architecture (offrant plus de possibilités de réutilisation et une maintenance plus aisée) et une conception plus pragmatique qui limitera la profusion des objets, offrira peut-être de meilleures performances à l'exécution et accélérera sans doute le processus de conception.

### *c. Quelques heuristiques*

Les deux grands principes évoqués ci-dessus, alliés aux propositions de classes qui résultent de la conception de la base de données et du scénario de l'interface devraient permettre de dégager progressivement l'ensemble des classes et des services associés qui constitueront l'application. Néanmoins, il nous paraît judicieux de compléter la panoplie de stratégies de conception en énumérant ici quelques conseils supplémentaires qui peuvent faciliter la tâche de conception.

1. **Réutiliser**

   Il ne faut pas seulement concevoir des classes pour qu'elles soient réutilisables, il faut aussi penser très souvent à rechercher dans les projets antérieurs s'il n'existe pas de classe qui puisse être réutilisée directement ou après quelques modifications.

   > C'est le propre d'un bon environnement de développement que de proposer, en plus d'un outil d'aide à la conception, une bibliothèque bien garnie en classes génériques modélisant les concepts que l'on retrouve régulièrement dans bien des logiciels tels que liste, pile, queue, fichier, arbre binaire ou encore menu, fenêtres, boîte de dialogue... Bien du travail restera donc à faire au terme de ce mémoire pour étoffer cette panoplie

2. **Réifier**

   Il ne faut pas craindre de réifier une structure de données en un objet informatique. Si la classe ainsi détectée s'avère profitable, tant mieux; sinon il sera encore temps de la supprimer ou de l'agréger à une autre par la suite. Un bon critère de choix est le suivant : "Une notion ne doit être établie comme classe que si elle décrit un ensemble d'objets caractérisés par des opérations intéressantes et des propriétés significatives" [MEYER, 389].

3. **Hériter**

   L'héritage est un mécanisme plein de ressources pour le concepteur et doit être utilisé chaque fois que cela est possible. Entre autres, l'héritage permet de rassembler dans une classe générique les caractéristiques et comportements identiques d'un ensemble d'objets semblables. De même, l'héritage permet de distinguer un concept général et facilement réutilisable de caractéristiques locales spécifiques à sa mise en oeuvre dans une application donnée.

4. **Limiter les arguments**

   Le fait de devoir passer un grand nombre d'arguments à une opération est souvent l'indice d'une classe qui n'a pas été détectée et qui gagnerait probablement à être mise en évidence.

5. **Proposer des services**

   Il ne faut pas nécessairement attendre qu'un service soit requis par un composant de l'application pour le proposer dans une classe. Au contraire, si l'on en croit les tenants de la conception ascendante, il est conseillé de doter une classe d'un maximum de services (pour autant que ceux-ci aient une sémantique bien définie et une parenté logique avec le concept représenté par la classe).

6. **Rester "local"**

   Le principe de localité est fondamental dans le paradigme objet. Aussi, il faut éviter de rédiger des opérations qui doivent connaître trop de caractéristiques d'autres objets, ou pire, qui modifient directement ces caractéristiques. Au contraire, il faut toujours préférer adresser un message à un objet pour lui demander d'inspecter / de modifier lui-même ses propres caractéristiques.

7. **Regrouper les opérations par argument**

   Lorsqu'une opération concerne plusieurs classes, on hésite parfois pour déterminer dans quelle classe il convient de décrire l'opération.

   Prenons un exemple en gestion bibliothécaire pour illustrer ce problème. Soit une opération (fonction) nommée `Disponible` qui doit retourner un exemplaire d'un document déterminé qui soit disponible pour le prêt. Faut-il décrire cette opération dans la classe `Document` ou dans la classe `Exemplaire` ? En pratique c'est à la classe `Document` qu'il convient de rattacher l'opération `Disponible` car l'objet `Document` est bien connu et c'est à lui qui l'on peut facilement adresser le "message" "Donne-moi un exemplaire disponible". Décrire l'opération `Disponible` dans la classe `Exemplaire` n'a, par contre, pratiquement pas de sens car on ne sait pas, à priori, à quel objet `Exemplaire` il convient d'adresser la requête.

   En conséquence, il convient de rattacher une opération à la classe qui en est l'argument (implicite) c'est-à-dire à la classe dont un objet est manipulé ou consulté et non à la classe qui décrit l'objet qui est retourné en résultat.

*d. Application au projet "Voyage scolaire"*

Voyons à présent comment ces diverses stratégies peuvent être mises en oeuvre dans notre exemple illustratif. Bien entendu, il ne saurait être question de présenter ici le détail de toute la conception de ce projet qui, malgré sa taille modeste, fait apparaître la plupart des problèmes types qui peuvent être rencontrés. Nous nous bornerons donc ici à présenter quelques éléments clefs de cette conception[2]. Rappelons toutefois que l'ensemble de la

[2] Il faut noter que cet exemple de conception n'est pas seulement une illustration de la méthode mais nous a réellement servi à la mise au point de la méthode. Aussi, on voudra bien nous

documentation relative à ce projet, générée par l'outil SACOO d'aide à la conception, est disponible dans les annexes.

L'analyse des besoins (événements) conjuguée avec les informations qui avaient pu être mise en évidence lors de conception de la base de données et de l'interface utilisateur nous a conduit à définir 11 classes d'objets qui sont structurées comme l'illustre le schéma de la figure 5.14. Les lignes grises de cette figure regroupent les classes en quatre modules suivant l'architecture que nous proposons.

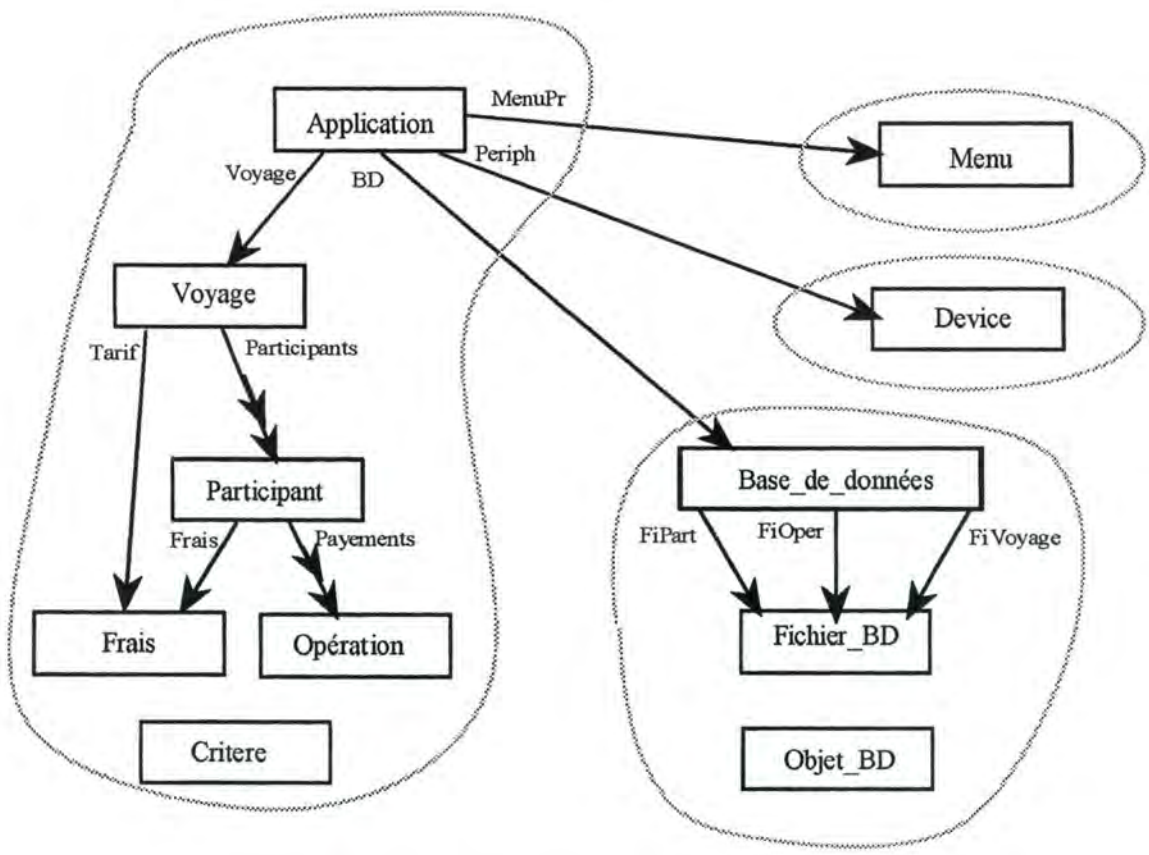



Figure 5.14 : Architecture globale des classes

Présentons brièvement chacune de ces classes :

- **`Application`** a essentiellement été créée pour contenir l'opération `Programme` qui constitue le véritable "programme principal" de l'application. Elle assure aussi le lien avec les autres modules mais sa justification est essentiellement d'ordre technique. Il ne peut bien entendu y avoir qu'une seule instance pour cette classe `Application`.
- La classe **`Voyage`**, pour laquelle il n'existera aussi qu'une seule instance, correspond par contre à une réalité beaucoup plus objective. Elle modélise le voyage dont on assure la gestion. Ce voyage est

excuser si l'une ou l'autre recommandation proposée ci-dessus n'est pas respectée à la lettre dans cet exemple.

principalement caractérisé par le `Tarif` standard des frais et par l'ensemble des `Participants` qui se sont inscrits. Le voyage est aussi caractérisé par quelques informations simples que sont sa `Destination` et la `Période` où il aura lieu. Ces dernières informations pourront être utiles lors de l'impression des listes et extraits de compte. C'est aussi à la classe `Voyage` que seront attachées toutes les opérations responsables des fonctionnalités demandées par l'acteur `Gestionnaire` telles que `Lister_Participant()`, `Inscrire_Part()`, `Etablir_Situation()`, ...

- La classe **`Participant`** contiendra bien entendu les objets représentant chacun un participant. Ses attributs sont bien évidemment ceux qui ont été définis lors de la conception de la base de données et ses opérations correspondent aux services qui sont susceptibles d'être demandés à chaque participant pris isolément. On y trouve entre autre `Ajouter_Payement()` qui gérera l'ajout d'un nouveau payement ainsi que plusieurs opérations retournant des informations au sujet du participant. `Total_paye()` et `Solde()` donnent ainsi une information globalisée sur l'état du compte d'un participant, `Extrait()` et `Description()` fournissent des chaînes de caractères qui pourront être directement intégrées dans les rapports demandés.
- Les classes **`Opération`** (financière) et **`Frais`** sont introduites pour modéliser les différents objets composants de `Participant`. Ils permettent également de rassembler les opérations spécifiques au traitement de chacun de ces concepts.
- La classe **`Critère`** est plus originale car elle ne correspond pas directement à une structure de données gérée par la BD. Cette classe a été introduite pour modéliser le concept de critère permettant de sélectionner une partie des participants dans un but quelconque. On sera amené à y faire appel lors des impressions de listes et extraits de compte.
- La classe **`Base_de_données`** assure la gestion globale des fichiers contenant les informations permanentes. Ses trois attributs `FiPart`, `FiOper` et `FiVoyage` contiennent la description des fichiers effectivement manipulés. Outre l'ouverture et la fermeture de la BD, cette classe offre aussi des services plus spécifiques à l'application tels que la recherche des payements associée à un participant ou la liste des identifiants de participants correspondant à un nom donné.
- La classe **`Fichier_BD`** est un bel exemple de classe réutilisable. Elle offre les services de base que l'on peut être amené à demander à un fichier dBase quelconque : ouverture, fermeture, choix d'un index, recherche d'une information via un index, ...
- La classe **`Objet_BD`** est une classe (réutilisable) dont héritent `Participant` et `Opération`. Son seul rôle consiste uniquement à déclarer un attribut `Identif` qui permet l'enregistrement d'un identifiant pour tous les objets permanents. `Objet_BD` ajoute donc une "clé technique" à tous les objets qui sont sauvegardés sur fichier.

- La classe **Menu** permet de gérer des menus "popup" relativement simples. Elle sera utilisée par le menu principal mais aussi lors de la sélection des participants et des payements à modifier.
- La classe **Device** permet enfin, de modéliser le périphérique de sortie qui sera employé pour "imprimer" les listes et extrait de comptes. Tout comme Menu, la classe Device est aussi susceptible d'être facilement réutilisée.

```
Ajouter_Payement        OPERATION [14][3]

   Définition :
      Inscrition d'un payement nouveau pour ce participant

   Opération de : Participant

   Algorithme :
      /**** Script ****
      Afficher le dernier payement enregistré  Op:Afficher(Ligne)
      Créer un payement vide
      Répéter
        Saisir le nouveau payement             Op:Saisir(Ligne)
        Verifier cohérence                     Op:VerifieCI()
        jusqu'a ce que confirmation ou abandon
      Si Confirmation alors
        Affecter un identifiant au payement Ap:BD:FiOper:NouvId()
        Enregistrer le payement                Op:Ecrire()
        Ajouter cette opération aux Payements
      */

   Opération(s) utilisée(s) :
      Ecrire / Operation
      Afficher / Operation
      Saisir / Operation
      VerifieCI / Operation
      NouvId / Fichier_BD

   Est utilisée par :
      Gerer_Payement / Voyage

   Attribut(s) modifié(s) :
      Payements

   Evenement(s) déclencheur(s) :
      D_Enregistre_Payem.
```

Figure 5.15 : Description d'une opération générée par SACOO.

[3] Ce numéro est l'identifiant de l'opération dans le repository du logiciel SACOO. Il est donné a titre purement informatif.

A côté du relevé des classes nécessaires à l'application et de la description de leurs attributs, un autre composant-clef de la conception réside dans la définition des opérations et de leur dynamique interne décrite par leur script (ou éventuellement par leur algorithme).

La figure 5.15 donne, à titre d'exemple, la description de l'une des 72 opérations qui ont été définies dans ce projet. Ce texte de description est tel qu'il est produit par le générateur de rapports de l'outil SACOO que nous avons mis au point pour supporter notre méthode. Le script proprement dit est un texte rédigé selon une syntaxe libre (pseudo-code) qui donne les grandes lignes de l'algorithme qui sera nécessaire et met en exergue les appels qu'il faudra effectuer à d'autres opérations.

### 5.3.5 Implémentation et vérifications

Nous arrivons à la dernière phase du processus de développement. Si les phases précédentes ont été conduites avec soin, il ne reste pratiquement plus que des détails mineurs à régler lors de la "traduction" des scripts en algorithmes exprimés dans le langage de programmation choisi (Clipper dans notre cas). Cette phase est aussi l'occasion de vérifier la cohérence mutuelle de tous les éléments manipulés :classes, attributs, opérations et arguments. De même, il faut vérifier que tout événement correspond bien au déclenchement d'une opération. Enfin, il convient de s'assurer que le respect de toutes les contraintes qui ont été énoncées est bien garanti par les opérations qui modifient des valeurs des attributs.

## 5.4 Remarques et commentaires

Avec la fin de la présentation de notre modèle et de notre méthode vient l'heure d'un premier bilan. Nous évoquerons tout d'abord quelques extensions qui pourraient compléter harmonieusement le travail déjà réalisé. Nous émettrons ensuite quelques réflexions d'ordre plus général et nous terminerons par un rapide retour sur nos objectifs.

### 5.4.1 Extensions

Parmi les extensions possibles, nous avons déjà évoqué la possibilité de considérer des types de données plus sophistiquées telles que l'énumération et l'intervalle qui sont très souvent utilisés.

Une amélioration plus fondamentale consisterait à introduire, tant pour les attributs que les opérations, un degré de protection qui permette de distinguer les éléments qui sont publics (accessibles librement de l'extérieur de la classe) et ceux qui sont privés (invisibles de l'extérieur). Cette distinction aurait l'avantage, outre sa compatibilité avec les catégories `exported:` et `hidden:` de Class(y) et de beaucoups de langages à objets, de distinguer clairement les éléments qui servent à la "cuisine interne" d'une classe de ceux qui sont mis à la disposition des autres classes.

### 5.4.2 Quelques réflexions

Enfin, il est évident qu'un travail d'approfondissement doit être réalisé du côté de l'élaboration du scénario de l'interface afin de définir un formalisme plus précis et si possible de décrire des règles de dérivation de façon à mettre en évidence des classes, des attributs ou des opérations à partir de ce scénario.

Si l'on prend un peu plus de hauteur vis-à-vis du travail, on constate tout d'abord qu'il ne peut renier sa filiation par rapport aux trois méthodes qui ont été analysées au chapitre 4. La description statique des classes : attribut, composition, références, ... sont largement inspirés du modèle O* tandis que plusieurs éléments de la description dynamique (opérations, arguments, ...) sont plutôt héritiers du modèle OBLOG. Enfin, la méthode HOOD se retrouve certainement dans l'approche globalement descendante que nous proposons pour la conception.

D'aucuns pourraient objecter que la méthode que nous proposons n'est pas spécifiquement orientée objet mais qu'il s'agit d'une approche fonctionnelle déguisée OO. Par avance, nous répondons par les deux arguments suivants : tout d'abord, il faut rappeler qu'une étape primordiale de la démarche consiste à rechercher les classes d'objets qui sont nécessaires pour offrir les services demandés et que ce n'est qu'ensuite que l'on procède à la définition des services offerts par ces classes. Le second argument est celui du réalisme ou du pragmatisme. Tout projet informatique est centré sur l'automatisation de tâches, il serait donc vain de vouloir ignorer que la recherche de la solution est nécessairement guidée par les tâches à automatiser c'est-à-dire, par les fonctions à offrir. Une approche n'incluant pas une dimension "fonctionnelle" n'est donc pas possible.

Une dernière réflexion que nous voudrions formuler concerne l'atomisation du "programme" obtenu. Il est frappant, en effet, de constater que l'approche orientée objet nous conduit à découper le programme en de très nombreuses petites "procédures" alors qu'une approche plus traditionnelle est nettement plus économe. Ainsi par exemple, le projet "voyage scolaire" nous a amené à définir plus de 70 opérations alors que le même programme, rédigé de façon classique il y a quelques années, nécessitait moins de 30 procédures. Cette atomisation est intrinsèquement porteuse d'avantages : elle permet une meilleure décomposition des tâches et une meilleure répartition sur les objets qui en sont responsables. La réutilisabilité et la maintenance doivent en être favorablement influencées. Toutefois, il faut bien reconnaître que cette atomisation transforme le programme en un vaste puzzle dans lequel le concepteur a parfois bien du mal à trouver la "pièce" qu'il recherche. La nécessité d'un outil d'assistance à la conception, assurant au minimum le management de ce "puzzle", est donc tout à fait évidente.

### 5.4.3 Evaluation

Si pour conclure, on se reporte aux objectifs que nous nous étions assignés, on peut être raisonnablement satisfait car le modèle défini est effectivement assez

simple : une dizaine de concepts dont quatre seulement sont vraiment fondamentaux (classe, attribut, opération et héritage). De plus, la cohérence avec les modèles classiques a été largement illustrée.

Du côté de la méthode, nous avons réussi à poser des jalons solides dans chacune des étapes du développement et de nombreuses stratégies très pratiques ont pu être mises en évidence.

# 6

# Le logiciel SACOO

L'outil SACOO (Système d'Aide à la Conception Orientée Objet) est le complément naturel du modèle et de la méthode que nous venons de présenter. Il résulte en fait d'une constatation relativement anodine qui s'imposa dès les premières réflexions sur les propositions méthodologiques. En effet, pour mettre au point une méthode, il nous a semblé indispensable de l'expérimenter. Or il est apparu d'emblée nécessaire de disposer d'un "super éditeur" (ou "super bloc-notes") pour passer rapidement de la description d'une classe à celle de ses attributs ou de ses opérations, consulter la liste des événements, vérifier l'existence d'une opération dans une autre classe, décrire une contrainte, retrouver les opérations qui modifient la valeur d'un certain attribut... Bref l'utilisation d'un programme éditeur classique ou même d'un traitement de texte ne pouvait être satisfaisante. C'est donc tout naturellement qu'a germé le projet de développer un outil d'assistance qui "colle" de très près au modèle et à la méthode de conception qui allait être développée.

Nous nous trouvions cependant face au traditionnel problème dit "de la poule et de l'oeuf" : l'outil doit implémenter la méthode et la méthode est nécessaire pour construire l'outil. Aussi, comme de surcroît nous ne disposions que d'une expérience minime en programmation orientée objet, il fut décidé de commencer par développer un prototype d'outil qui offre un support à la modélisation orientée objet d'une application. Une première version du modèle qui devait être pris en compte avait bien entendu été définie préalablement et assez peu de changements furent nécessaires pour aboutir au modèle qui a été décrit au chapitre 5.

Ce travail de développement a ainsi permis d'amasser des connaissances sur trois plans complémentaires :

1. Apprentissage et perfectionnement de la technique de programmation orientée objet avec le couple Clipper + Class(y).
2. Mise à l'épreuve et raffinement des concepts retenus dans le modèle orienté objet pour la conception.

3. Réflexion et mise en oeuvre d'une démarche de conception orientée objet qui intègre tous les aspects d'une application moderne de gestion.

Pour des raisons très pragmatiques d'économie du temps de développement, il a d'emblée été accepté de ne pas chercher à écrire le logiciel avec 100 % de code orienté objet, mais de s'autoriser à employer une écriture plus "classique" soit lorsque des modules pouvaient être simplement récupérés d'autres applications (réutilisation !), soit encore lorsque l'implémentation de certaines fonctionnalités semblaient (à ce moment-là en tous cas) malaisée à formuler selon le paradigme objet. Si l'on compare les volumes des codes sources, on constate en effet que SACOO est écrit avec plus de 80 % de code orienté objet et un peu moins de 20 % de code "classique". Nous montrerons plus loin que cette organisation, si elle est moins "pure", est probablement plus raisonnable et devrait sans doute être conseillée.

La présentation minutieuse de tout le processus de développement du programme SACOO serait certainement très fastidieuse sans pour autant ajouter beaucoup à la réflexion que nous avons voulu mener dans ce mémoire. Nous nous permettons donc dans ce chapitre de n'en brosser qu'un portrait assez rapide en mettant l'accent sur les étapes et les décisions clés qui ont émaillé le développement. Nous ne pouvons également que conseiller au lecteur d'installer la copie du logiciel qui est jointe en annexe et d'expérimenter par lui-même cet outil d'aide à la conception.

> Le lecteur sera peut-être surpris de ne pas trouver dans ce dernier chapitre la mise en oeuvre de tous les principes de conception orientée objet qui ont été décrits dans le chapitre précédent. En fait, il faut savoir que l'ordre de présentation du mémoire est exactement inverse à celui de la recherche qui a précédé sa rédaction. En effet, c'est le développement de SACOO qui à fait émerger un certain nombre de principes de conception. Ces derniers ont ensuite été mis à l'épreuve, corrigés et complétés lors de la conception, à l'aide de SACOO, du projet "Voyage scolaire". Enfin, la rédaction proprement dite du chapitre 5 a encore permis de peaufiner les propositions méthodologiques qui y sont présentées.

## 6.1 Spécifications

Décrire a posteriori les spécifications d'un logiciel est, par nature, une démarche un peu hypocrite et ce, a fortiori, dans le cas où le logiciel est lui-même un prototype permettant d'affiner et de corriger ces spécifications. Il ne faut donc pas voir dans la liste de spécifications qui va être présentée un énoncé rigide et strict des fonctionnalités qui aurait été entièrement défini avant même de commencer l'écriture du logiciel mais plutôt comme le relevé des fonctionnalités qui se sont progressivement révélées indispensables pour le support du modèle et

de la méthode associée. La plupart des spécifications sont d'ores et déjà implémentées dans la version de SACOO fournie en annexe et le logiciel a été utilisé avec succès pour supporter la conception du projet "Voyage scolaire" présenté au chapitre 5. Les quelques fonctionnalités qui n'ont pas encore pu être implémentées sont signalées en regard de leur description.

Les fonctionnalités attendues du logiciel SACOO sont les suivantes :

- Assurer l'enregistrement et la restitution des informations, pour la plupart textuelles, de description des concepts du modèle (classes, attributs, événements, ...) mis en évidence dans le projet à l'étude. Pour SACOO, tous ces concepts seront des **objets** qu'il manipulera et qui seront stockés dans une base de données spécialisée que nous appellerons le "repository" pour sacrifier au jargon des outils CASE.
- Offrir toutes facilités pour ajouter des objets dans le repository, en modifier la description, les renommer voire les supprimer.
- Permettre d'importer dans un projet des objets (essentiellement des classes) qui ont été décrits dans un autre projet (= support à la réutilisation).
- Offrir plusieurs méthodes pour retrouver facilement un objet qui a déjà été défini et visualiser tous les éléments de sa description (cf. métamodèle au paragraphe 5.2.9).
- Offrir la possibilité d'afficher de nombreuses "références croisées" entre les objets du repository de façon à permettre au concepteur d'avoir une vue aussi complète que possible des objets qu'il manipule.

  En particulier, il faut pouvoir visualiser :

  Pour une classe :
  - la chaîne de ses superclasses (héritage amont);
  - la liste de ses sous-classes directes (héritage aval);
  - les listes des attributs, opérations et contraintes hérités avec indication de la superclasse propriétaire et de la surcharge éventuelle par un objet de même nom (essentiellement pour les opérations).

  Pour un attribut :
  - la liste des contraintes subies,
  - la liste des opérations habilitées à modifier la valeur de cet attribut.

  Pour une opération :
  - la liste des attributs modifiés (ou plutôt modifiables) par cette opération,
  - la liste des opérations qui sont utilisées pour son implémentation,
  - la liste des opérations qui font appel à cette opération,
  - la liste des événements qui déclenchent cette opération.

  Pour une contrainte :
  - la liste des attributs contraints.
- Etablir des rapports de description du projet ou de certaines de ses parties et notamment :

- les schémas d'une ou plusieurs classes selon la syntaxe graphique qui a été définie et présentant, à la demande, les attributs, les opérations et les relations d'héritage.
- Des rapports textuels reprenant la description détaillée d'un objet, d'une catégorie d'objets ou de tous les objets du projet.

♦ Générer des statistiques relatives au projet et en particulier au degré de couplage entre les objets (implémenté de façon rudimentaire dans la version actuelle).

♦ Vérifier la cohérence / la complétude des spécifications (non implémenté dans la version actuelle).

♦ Générer automatiquement des listings Clipper + Class(y) directement compilables et intégrant tous les éléments connus de la description des classes.

De plus, l'ensemble de ces fonctionnalités, directement liées à la conduite du processus de conception, doivent être fournies dans un environnement qui respecte les contraintes suivantes :

♦ Offrir une interface utilisateur aussi conviviale et moderne que possible (menus déroulants, souris, multifenêtrage,...).

♦ Donner de nombreuses possibilités de paramétrage telles que :
- choix des éléments affichés pour chaque catégorie d'objet (par ex. : afficher les contraintes subies par un attribut, mais pas ses opérations modificatrices),
- choix des tailles par défaut des fenêtres,
- choix des couleurs d'affichage (implémenté en partie seulement),
- choix des périphériques de sortie pour les rapports (écran, imprimante fichier disque) et de plusieurs standards de mise en forme (pilotes d'imprimantes).

Enfin, aucune concurrence n'est prévue dans l'accès au repository car le logiciel (ou au moins sa version prototype) ne doit fonctionner que dans un environnement mono-poste.

## 6.2 Organisation du repository

La première tâche à laquelle il a fallu faire face était la conception de la base de données (ou repository) qui permettrait le stockage de tous les éléments informationnels nécessaires. C'est bien entendu le métamodèle qui a servi de base à cette réflexion. Il a cependant été nécessaire de raffiner ce schéma afin d'offrir un maximum de souplesse et de mémoriser toutes les associations supplémentaires demandées dans la spécification (par ex. : la relation d'utilisation entre opérations).

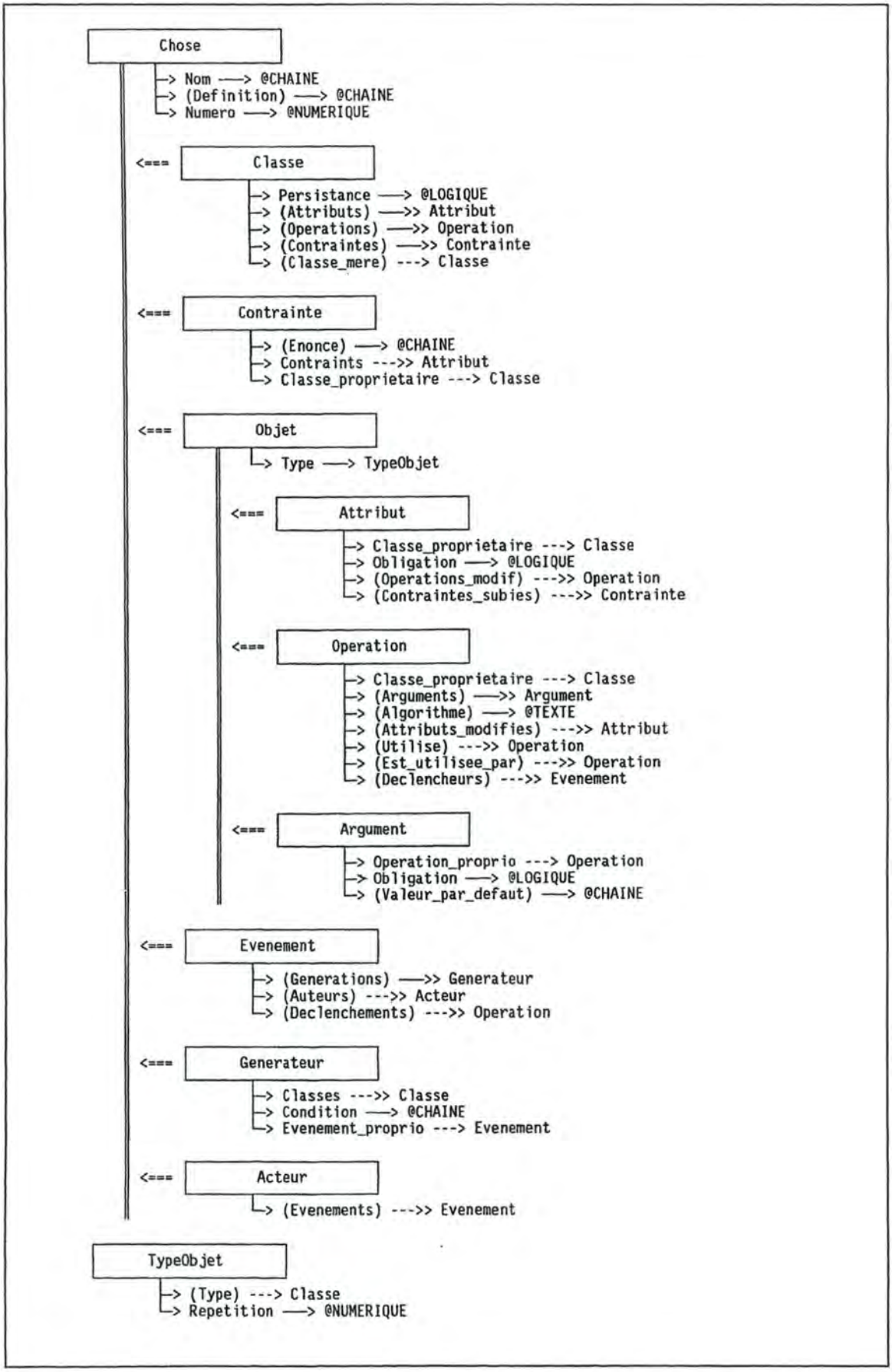



Figure 6.1 : Schémas individuels des classes du repository

### 6.2.1 Schémas individuels des classes

L'ensemble des schémas individuels des classes nécessaires à la modélisation des divers objets manipulés par le logiciel est proposé à la figure 6.1. Commentons rapidement ce schéma en épinglant les différences par rapport au métamodèle qui avait été présenté au paragraphe 5.2.9.

Tout d'abord la classe `Concept` avait initialement été nommée `Chose` et ce nom est resté dans le logiciel SACOO. Ensuite, on observe que toute `Chose` est qualifiée par un numéro qui permet de l'identifier. C'est ce numéro qui sera utilisé dans les fichiers au titre de clef primaire.

Dans la description de la classe `Classe`, on note que l'attribut `TypeStandard` n'est pas repris car un type standard est aussi dénoté par le caractère `@` en préfixe de son nom.

Par contre, un attribut `Persistance` avait été introduit pour repérer les classes décrivant des objets persistants. L'utilité pratique de cette qualification n'ayant pas été démontrée, la `Persistance` n'est plus reprise dans le métamodèle actuel mais subsiste dans le logiciel SACOO. Enfin, on notera que l'attribut `SuperClasse` du métamodèle est renommé dans SACOO en `Classe_mere`.

La description de la classe `Attribut` comporte plusieurs éléments nouveaux. L'attribut référence `Classe_propriétaire` permet "d'inverser" la relation de composition entre `Classe` et `Attribut` en fournissant un pointeur vers la classe dont l'attribut est composant. Le même procédé est utilisé pour `Opération`, `Argument`, `Contrainte` et `Générateur` qui sont tous trois composants d'une autre `Chose`.

Les attributs références `Opérations_modif` et `Contraintes_subies` permettent quant à eux de stocker les "références croisées" qui ont été demandées dans la spécification. Les mêmes commentaires peuvent être faits pour la classe `Opération`.

Enfin, il faut encore noter, au sujet de la classe `TypeObjet` que les attributs `EstComposant` et `EstMultiple` ont été agrégés en un seul attribut `Répétition`. En effet, avec un attribut numérique, il était plus facile de faire face à une augmentation éventuelle du nombre de distinctions que l'on aurait pu faire sur les liens entre attribut et type. Cela aurait été le cas si l'on avait retenu la distinction entre références temporaires et permanentes comme suggéré au paragraphe 3.2.1

### 6.2.2 Schéma relationnel

Le schéma relationnel dérivé du schéma précédent est relativement simple et méthodique. A chacun des huit "véritables" concepts correspond une table

relationnelle et six tables supplémentaires sont nécessaires pour représenter les six associations N-M du schéma (11 références multiples[1]).

Le schéma relationnel est illustrée par la figure 5.2.

**Classe** (Numéro, Nom, Définition, Persistance, *Classe_Mere*)

**Attribut** (Numéro, Nom, Définition, Obligation, Répétition, *Type*, *Classe*)

**Opération** (Numéro, Nom, Répétition, *Type*, Algorithme, *Classe*)

**Argument** (Numéro, Nom, Définition, Obligation, Val_Défaut, Répétition, *Type*, *Opération*)

**Contrainte** (Numéro, Nom, Définition, Enoncé, *Classe*)

**Evénement** (Numéro, Nom, Définition)

**Générateurs** (Numéro, Nom, Définition, Condition, Evénement)

**Acteur** (Numéro, Nom, Définition)

**Modifier** (*Opération*, *Attribut*)

**Contraindre** (*Contrainte*, *Attribut*)

**Utiliser** (*Op-appelante*, *Op-appelée*)

**Générer** (*Génération*, *Classe*)

**Déclencher** (*Evénement*, *Opération*)

**Activer** (*Action*, *Evénement*)

Figure 6.2 : Schéma relationnel du repository

### 6.2.3 Schéma physique (dBASE)

Le passage du schéma relationnel au schéma physique des fichiers dBase est en principe immédiat, il suffit de définir un fichier pour chaque table relationnelle et de préciser le type et la longueur de chaque champ. Tous les numéros d'identification sont représentés sur cinq chiffres ce qui est plus que suffisant. Les noms des objets sont arbitrairement limités à 20 caractères. Cette contrainte peut paraître trop limitative, mais permettra d'éviter d'avoir des noms trop longs à afficher sur l'écran spécialement lorsqu'ils seront composés (par ex. : `Ajoute_Paiement / Participant`).

Les champs `Définition`, `Enoncé` (de `Contrainte`) et `Condition` (de `Génération`) seront par contre étendus jusqu'à 254 caractères qui est la longueur maximum d'un champ en dBase. Cela correspond à 4 (petites) lignes de texte et nous paraît suffisant dans la grande majorité des situations.

---

[1] Une douzième référence multiple a été omise dans le schéma. Elle aurait associé les classes avec les générateurs auxquels elles participent.

Pour le stockage des algorithmes, il faut par contre faire appel au type de champ "mémo" qui permet d'enregistrer des textes de longueur quelconque, mais nécessite un fichier annexe supplémentaire.

Pour uniformiser le mécanisme d'accès par clé aux différents fichiers, chacun sera doté d'un index sur le `Numéro`, un sur le `Nom` et éventuellement un sur le champ qui désigne le "propriétaire" des objets composants. La base de donnée compte ainsi déjà 30 fichiers distincts.

Si l'on ne prend pas de disposition spéciale, chaque table représentant une association N-M nécessitera aussi un fichier auquel il faudra ajouter deux index pour permettre les recherches dans les deux sens, soit un total de 18 fichiers supplémentaires. Pour éviter cette escalade du nombre de fichiers, nous avons décidé de regrouper les six tables en une seule, mais en ajoutant un attribut supplémentaire qui contiendra un code désignant la table "logique" à laquelle appartient un tuple donné. C'est donc le fichier `LIENS` qui combine les six tables logiques et sa structure est la suivante :

`Liens (`*Association*`, `*Origine*`, `*Cible*`)`

Grâce à l'approche objet, la gestion de ce fichier est parfaitement transparente pour les utilisateurs car elle est confiée à la classe `FLiens` qui est seule à connaître la représentation exacte des diverses associations. Pour l'extérieur, elle fournit des services ayant des noms (presque) explicites. Par exemple, l'opération (fonction) `AttrDeCont()`[2] retourne la liste des identificateurs des attributs concernés par une contrainte. Il en va de même pour `Utiliser()` ou encore pour `OperModiAt()` et ainsi de suite pour les 12 références multiples qui avaient été mises en évidence.

## 6.3 Interface utilisateur

Une seconde facette du travail qui a dû être considérée très tôt résidait dans l'élaboration d'un scénario d'interface qui offre un maximum de souplesse. La tâche se trouvait sensiblement compliquée (ou peut-être simplifiée ?!) par le fait que le langage Clipper n'offre qu'une interface en mode texte et qu'il n'existe aucune primitive de gestion de la souris. Ces deux limitations peuvent éventuellement être outrepassées par l'adjonction de librairies spécialisées. Décision fut prise de s'accommoder de l'écran texte, mais de recourir à la librairie "Funcky" pour disposer des quelques primitives de gestion de la souris qui seraient nécessaires.

La solution, apparemment la plus élégante pour maximiser l'ergonomie de l'outil, consiste à proposer, un peu à l'image de Windows, des fenêtres superposables et redimentionnables qui représentent chacune un objet ainsi qu'un système de

---

[2] C'est pour satisfaire les contraintes de Clipper que nous limitons ici les noms des procédures et fonctions à 10 caractères.

menus déroulants qui permettent de sélectionner les actions à poser sur les objets présentés dans la fenêtre active (c'est-à-dire celle qui est au-dessus de la pile des fenêtres).

Chaque fenêtre de présentation d'un objet du repository sera construite sur un même "pattern" (et pourra donc être gérée en partie par une classe générique) dont un exemple est donné à la figure 6.3.

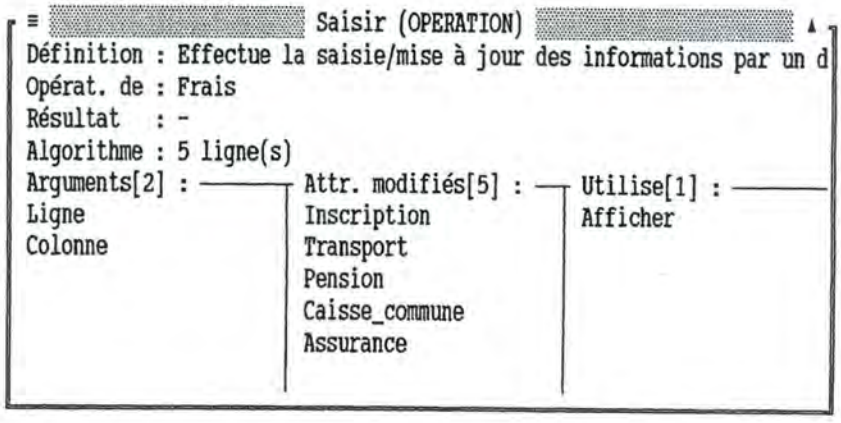



Figure 6.3 : Modèle de fenêtre de présentation d'une opération

Toute fenêtre d'objet est entourée d'un cadre qui est double pour la fenêtre active et simple pour toutes les autres. La barre horizontale supérieure du cadre est surchargée par une "case de fermeture" à l'extrémité gauche, le nom et la classe de l'objet ou centre est une "case de maximisation" à l'extrémité droite.

L'intérieur de la fenêtre est divisé en deux parties superposées. La partie supérieure est destinée à contenir les informations qui se prêtent à une description sous la forme d'une ligne de texte (définition, énoncé, chaîne d'héritage, obligation...). La partie inférieure est, par contre, divisée en colonnes afin de présenter les informations qui sont structurées en listes (liste des attributs d'une classe, des opérations utilisées, des événements déclencheurs....). Le nombre de lignes et de colonnes est automatiquement déterminé en fonction des paramètres de visualisation définis par l'utilisateur. Ces paramètres peuvent être modifiés via les menus de façon à ajouter ou supprimer dynamiquement certaines lignes ou colonnes dans les fenêtres. Bien entendu, le nombre de lignes de la partie inférieure ainsi que la largeur des colonnes sont automatiquement ajustés en cas de redimensionnement d'une fenêtre.

Les menus utilisés sont de plusieurs types : barre de menu, menu "popup", menu "pulldown", et menus mixtes. La barre de menu est évidemment un menu horizontal affiché en permanence en haut de l'écran. Un menu "popup" est un menu vertical qui se superpose sur l'écran lorsqu'il est activé puis qui disparaît dès que la sélection est opérée. Un menu "pulldown" est un cas particulier du précédent qui est directement attaché à un item de la barre de menu. Enfin, un menu mixte est un menu dont les items peuvent être répartis à des endroits quelconques de l'écran. Ils seront utilisés dans les fenêtres pour sélectionner la ligne ou la colonne sur laquelle l'utilisateur veut agir.

Figure 6.4 : Arborescence des menus de SACOO

Dans la fenêtre illustrée à la figure 6.3 par exemple, les textes `Définition`, `Algorithme`, `Argument`, `Attr. modifiés` et `Utilise` doivent pouvoir être considérés comme les items d'un menu "mixte" et être ainsi capables d'activer soit l'édition de l'information correspondante, soit la manipulation de la colonne associée.

Pour information, nous donnons à la figure 6.4 la structure hiérarchique des menus (barre, pulldown et éventuellement popup). Le menu `Visualisation` est un peu particulier car il est contextuel, c'est-à-dire que l'ensemble d'items qu'il affiche est différent suivant la catégorie de l'objet qui est présenté par la fenêtre active. Ainsi donc, si c'est une fenêtre présentant une opération qui est active, ce sera le menu `<si opération>` qui sera proposé à ce moment-là. Si aucune fenêtre n'est ouverte, c'est le menu `<si classe>` qui est affiché.

Notons encore que deux catégories d'items de noms doivent être distinguées : les items qui déclenchent une opération quelconque (`Enregistrer`, `Renommer`, `Statistiques`...) et ceux qui gèrent un commutateur et sont repérés dans la figure par un petit carré (`Définition`, `Héritage amont`, `Barre de statut`,...) Enfin, il faut signaler que plusieurs items activent des opérations dont l'exécution débute par l'ouverture d'une boîte de sélection dont le contenu est défini dynamiquement suivant les circonstances (`Ouvrir`, `Afficher...`).

## 6.4 Conception et architecture

### 6.4.1 Modules orientés objet

Il ne saurait être question de décrire ici par le menu la structure exacte des 46 classes qui ont été nécessaires pour réaliser le logiciel car cela nous conduirait à détailler plus de 150 attributs et 400 opérations ! Par contre, il nous paraît tout à fait intéressant de décrire sommairement la plupart des classes, de présenter leur organisation hiérarchique (via la relation d'héritage) et surtout de montrer le lien étroit qui existe entre ces classes et les "objets" que nous avons décrits aux paragraphes 6.2 et 6.3.

La figure 6.4 reprend les noms des 46 classes avec, les cas échéant, les relations d'héritage entre elles. Les différentes classes y sont regroupées en trois modules comme proposé dans notre architecture générique (paragraphe 5.3.4). On notera toutefois l'absence du module "Périphériques divers". En fait, c'est précisément en analysant l'architecture du logiciel SACOO qu'il nous est apparu intéressant de prévoir ce quatrième module pour regrouper les informations et les services associés aux périphériques tels que l'imprimante ou la souris. Dans SACOO, ces services sont décrits dans les modules "utilitaires" tels que `PRN_UTIL`.

***a) Module Application***

Le module "Application" comprend la hiérarchie des classes qui ont été mises en évidence lors de la conception du repository. Les classes qui en font partie déterminent, par leurs attributs, les informations qui seront retenues des `Choses`

manipulées par le concepteur et offrent des services "de base" pour manipuler ces informations telles que la création d'une nouvelle `Chose`, sa lecture ou son écriture dans un fichier ainsi que la production de diverses formes de description de la `Chose` (schéma, description complète ou simplifiée, ...)

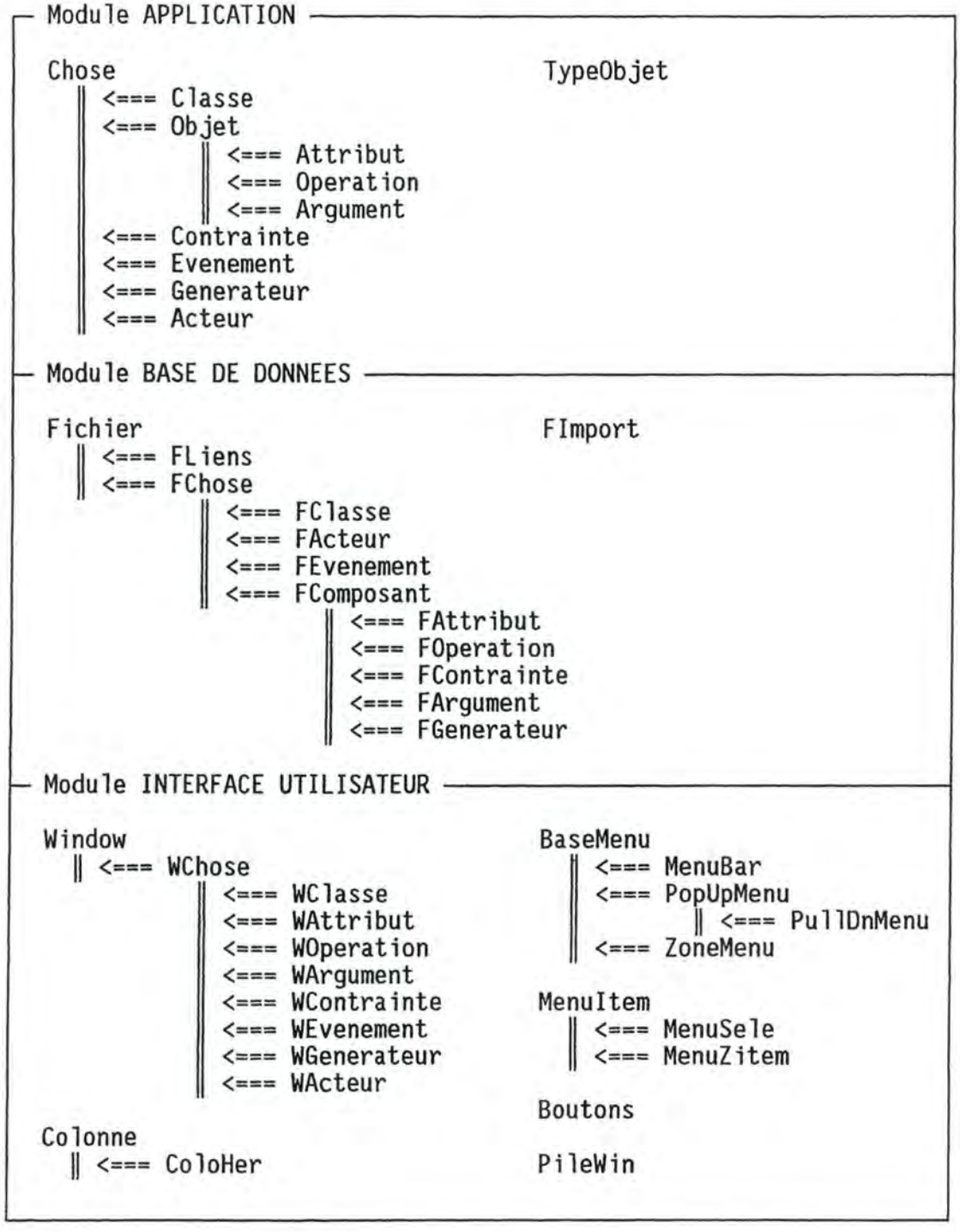



Figure 6.4 : Architecture de l'application SACOO

### *b) Module Base de données*

Le module "Base de données" a pour objectif principal la gestion des fichiers constituant le repository. Toutes les classes gérant un fichier au format dBASE

héritent de la classe `Fichier` qui propose a peu près les mêmes services génériques que ceux définis pour la classe `Fichier_BD` dans le projet "Voyage scolaire". Nous avons déjà évoqué le rôle très particulier de la classe `FLiens` pour gérer toutes les références "croisées" entre les objets stockés dans le repository.

La classe `FChose` offre toute une gamme de services qui sont valables pour tous les fichiers stockant des `Choses`. Il s'agit entre autres d'opérations qui retournent la `Chose` portant un numéro donné ou un nom donné ou encore qui impriment la liste des descriptions de toutes les choses contenues dans un même fichier. Beaucoup parmi ces opérations sont des opérations génériques car elles font appel à d'autres opérations polymorphes définies dans d'autres classes.

Une classe spécifique a été définie pour chaque fichier de données du repository et offre des services qui sont dépendants de la catégorie de la `Chose` manipulée et qui nécessitent des accès à plusieurs de ces `Choses`. Ainsi par exemple, l'opération `Lire` doit connaître la classe de la chose à lire pour renvoyer un objet au format désiré. De même, une opération telle que `AnceDeClas()` qui retourne la liste des classes ancêtres d'une classe donnée n'a de sens que dans le contexte du fichier des classes.

La classe `FComposant` a été construite pour regrouper quelques services partagés par toutes les classes qui gèrent un fichier dont les objets sont des attributs composants d'autres objets. Enfin, la classe `FImport` est tout à fait distincte des précédentes et centralise les services associés à l'importation d'une classe provenant d'un autre projet. Le fichier intermédiaire servant à l'import/export d'une classe n'étant pas au format dBase, il est normal que cette classe ne soit pas héritière de la classe `Fichier`.

*c) Module Interface Utilisateur*

Considérons à présent les classes rassemblées dans le module "Interface Utilisateur". Comme on pouvait s'y attendre à la lecture du paragraphe 6.3, la plupart des classes concernent soit la gestion des fenêtres, soit celle des menus.

La classe `Window` est une classe qui était livrée en tant qu'exemple avec la librairie Class(y). Il était donc tentant de réutiliser cette classe, puis de la spécialiser pour obtenir les différentes variantes de fenêtre qui allaient être nécessaires pour présenter les objets manipulés par le concepteur. Cette spécialisation comporte une "étape" intermédiaire, la classe `WChose`, qui rassemble tous les services génériques qui peuvent être partagés par les fenêtres ayant en charge la visualisation d'une `Chose`.

Dans les faits cependant, la réutilisation de la classe `Window` ne s'est pas effectuée sans difficulté et il a été nécessaire de repenser et de réécrire de nombreuses opérations. La classe livrée dans la librairie Class(y) effectue, en effet, un amalgame entre les notions de fenêtre et de pile de fenêtres qui en complique la mise en oeuvre. Nous avons au contraire choisi de bien séparer ces deux notions en les décrivant d'une part avec les classes qui héritent de `Window` et d'autre part avec la classe `PileWin` qui s'occupe de toutes les manipulations relatives à l'ensemble des fenêtres actuellement ouvertes.

La classe `Colonne` a été introduite pour gérer les différentes colonnes présentées dans la partie inférieure des fenêtres de visualisation des choses. `ColoHeri` est une héritière de `Colonne` spécialisée dans la gestion des colonnes contenant éventuellement des objets hérités. Cette classe est utilisée par `WClasse` pour les colonnes décrivant ses attributs, opérations et contraintes qui peuvent soit être propres à la classe visualisée, soit être hérités et même surchargés.

Les classes `BaseMenu`, `MenuBar`, `PopUpMenu`, `PullDnMenu` et `MenuItem` ont également été empruntées aux exemples fournis avec la librairie Class(y). Tout comme pour la classe Window, il s'est avéré indispensable de les réorganiser et de les améliorer assez sensiblement afin, entre autres, de permettre la commande des menus aussi bien avec la souris qu'avec le clavier. Certaines de ces classes ont de plus été spécialisées pour permettre la gestion des items "commutateurs" (`MenuSele`) et des menus "mixtes" (`MenuZone` et `MenuZitem`).

Enfin la classe `Boutons` a été conçue pour assurer la gestion de toutes les zones de l'écran qui sont sensibles à un clic de la souris. Pour l'utilisateur, les services offerts par cette classe sont fort simples et transparents. Les boutons sont organisés en pile et chacun est identifié par un numéro qui est déterminé par la classe elle-même. Les principales opérations exportées sont les suivantes :

`AddBouton(Top,Left,Bottom,Right) ——> @NUMERIQUE`
Ajoute un nouveau bouton couvrant la zone donnée

`RemBouton(Numéro)`
Supprime le bouton dont on donne le numéro

`LireBoutons() ——> @NUMERIQUE`
Attend qu'un clic de souris survienne et retourne le numéro du bouton sélectionné. Si le clic a lieu à l'intersection des zones couvertes par plusieurs boutons, c'est le numéro du bouton situé le plus "haut" qui est retourné.

`ToTop(Numéro)`
Déplace le bouton désigné en haut de la pile.

### 6.4.2 Modules non-orientés objet

La portion du code qui n'a pas été rédigée avec une orientation objet explicite peut être divisée en trois modules distincts.

***a) Programme principal et fonctionnalités de base.***

Le fichier source `COO.PRG` contient le programme principal de l'application. Son rôle consiste d'abord à organiser la phase de lancement en coordonnant la déclaration des variables publiques, l'ouverture des fichiers, l'instantiation et l'initialisation de nombreux objets qui resteront utilisables tout au long de l'exécution : menus, pile des fenêtres, pile des boutons, fichiers du repository, ... La seconde partie du programme principal est une boucle dans laquelle on trouve essentiellement un appel où l'objet `Boutons` pour qu'il retourne le numéro du bouton "pressé" puis, en fonction de ce numéro, programme principal passe la main au système de menus ou à la fenêtre qui a été "cliquée".

Dans le même module, on trouve également une vingtaine de procédures qui correspondent aux fonctionnalités de base du logiciel et qui sont activées par l'intermédiaire des menus. Ce sont entre autres `NouveauProjet()`, `OuvrirProjet()`, `OuvrirObj()`, `AffichObj()`, `Statistiques()` ou encore `Quitter_Tout()`. Si l'on avait voulu programmer "objet" jusqu'au bout, il aurait suffi de décrire une classe `Sacoo` modélisant le programme lui-même, dont les attributs avaient été les variables publiques et ayant pour opérations les procédures ci-dessus. Un programme principal, même très réduit, resterait cependant nécessaire pour instancier la classe `Sacoo` et passer la main à cet objet.

*b) Initialisation et paramètrage*

Les deux fichiers `MENUS.PRG` et `GESTPARA.PRG` contiennent quelques procédures purement séquentielles (à trois structures alternatives près) qui assurent la déclaration et l'initialisation des items des menus et des nombreux paramètres modifiables.

*c) Utilitaires divers*

Trois fichiers rassemblent une trentaine de procédures et fonctions utilitaires. Une dizaine sont rangées dans les fichiers `MSE_UTIL.PRG` et `PRN_UTIL.PRG` et correspondent respectivement à des utilitaires de gestion de la souris et de l'imprimante. Il suffirait de peu de chose pour les transformer en deux classes `Souris` et `Imprimante` qui prendraient ainsi place dans le module "Périphériques divers" de l'architecture générique que nous proposions au chapitre précédent.

Le cas des utilitaires présents dans le dernier fichier, baptisé `COO_UTIL.PRG` est par contre plus intéressant. En effet, on peut globalement classer les procédures et fonctions que l'on y trouve en deux catégories. La première rassemble une série de fonctions extrêmement simples mais pourtant fort utiles et que l'on peut considérer comme des "macro instructions". C'est le cas, par exemple de la fonction `OuiNon(Valeur_logique)` qui retourne la chaîne "Oui" ou "Non" suivant la valeur de l'argument booléen qui lui est passé.

La seconde catégorie regroupe des procédures ou fonctions qui permettent la saisie, l'édition ou la manipulation d'un type bien précis d'information. La fonction `EditText(Texte, Titre, Ligne, Cal, Hauteur, Largeur)` en est un bon exemple. Elle permet en effet, d'éditer un texte de longueur quelconque dans une fenêtre qui est ouverte à l'endroit désigné et surmonté du titre donné. On pourrait être tenté de définir une classe `Texte` et de lui associer cette unique opération mais cette manière de travailler se révèle assez lourde d'autant plus que chaque utilitaire manipule un type de données différentes et qu'il faudrait ainsi créer de multiples classes ayant finalement une seule opération. Aussi, dans le cas de tous ces utilitaires qui visent en fait à améliorer ou prolonger les primitives du langage, il nous paraît sage de continuer à les rassembler dans une petite bibliothèque qui pourra facilement être réutilisée d'un programme à l'autre.

## 6.5 Extension et améliorations proposées

La version prototype de SACOO constitue dès aujourd'hui un outil puissant pour la conception d'un logiciel en adoptant une approche orientée objet. Sa capacité à visualiser sous de multiples angles les différentes classes qui composent le logiciel en cours de conception et la possibilité de générer automatiquement un listing directement compilable tel que celui présenté en l'annexe (classe `Participant` du projet "Voyage scolaire") devrait d'ores et déjà séduire plus d'un développeur.

Néanmoins, ce logiciel pourrait encore être amélioré dans plusieurs directions que nous allons rapidement évoquer.

### 6.5.1 Améliorations de l'interface

Tout d'abord, il est clair que l'interface uniquement textuelle (et semi-graphique pour les schémas) est très limitative et pourrait être sensiblement améliorée par l'adaptation d'une véritable interface graphique du type "Windows". La prochaine version de Clipper, annoncée pour l'automne de cette année, doit "tourner" sous Windows et offrira une véritable gestion des objets; elle semble tout indiquée pour réaliser cette amélioration. La structuration actuelle du logiciel en classes d'objets devrait largement faciliter cette migration puisqu'en principe, seul le module "Interface Utilisateur" de notre architecture devrait subir des modifications conséquentes.

De nombreuses améliorations de détails permettraient aussi d'augmenter la convivialité du logiciel et par là même, de limiter les risque d'erreur. Ainsi, par exemple, la sélection d'une opération (opération utilisée, utilisatrice ou modificatrice d'attribut) devient vite fastidieuse car SACOO propose invariablement la liste de toutes les opérations de toutes les classes. Un accès hiérarchisé proposant d'abord les opérations locales de la classe puis, à la demande, celles d'une autre classe, serait probablement de nature à faciliter la sélection.

### 6.5.2 Ajout de fonctionnalités

Il serait souhaitable également d'étoffer les outils de statistique et de vérification de cohérence du repository. L'intégration plus explicite du concept d'association entre classes permettrait de réaliser assez facilement un outil qui génère automatiquement la structure des fichiers dBASE nécessaires à la base de données. L'optimisation de cette structure des fichiers serait toutefois laissée à l'utilisateur-concepteur.

Le logiciel SACOO pourrait aussi permettre de distinguer les attributs et opérations qui sont exportés de ceux qui sont internes et servent uniquement d'auxiliaire à l'implémentation des services offerts. Une suggestion pratique consisterait à afficher les attributs et opérations de ces deux catégories dans des couleurs différentes (choisies par l'utilisateur). Dans les schémas, les attributs et

opérations internes pourraient, à la demande de l'utilisateur, être affichés entre crochets ou ne pas être affichés.

Bref, il reste encore du pain sur la planche...

# 7

# Conclusion

Au terme de ce travail, il serait prétentieux de résumer et de figer la réflexion qui a été menée pour en dégager des conclusions définitives. Tout au contraire, il nous apparaît que ce travail devrait être prolongé, par exemple dans le cadre des formations du CeFIS, et que les modèles et outils proposés devraient être expérimentés plus largement pour vérifier leur adéquation aux besoins ainsi que leur "ergonomie". Bien plus qu'une conclusion, nous voudrions donc que cette étape soit au contraire un nouveau départ pour le perfectionnement de la méthodologie que nous avons voulu mettre en place et qu'elle contribue modestement à la vulgarisation du paradigme objet.

Nous voudrions toutefois mettre en avant quelques réflexions qui nous paraissent importantes. La première consiste à insister sur la nécessité d'une approche pragmatique du développement de logiciel. Nous avons largement montré les apports du paradigme objet comme support des processus de spécification déjà, de conception et de programmation plus certainement encore. Il serait cependant vain, à notre avis, de vouloir tout modéliser avec des objets. Le programme principal et les procédures de gestion des grandes fonctionnalités d'un logiciel ne se prêtent souvent qu'artificiellement à la modélisation par objet. Il en va de même pour un certain nombre d'utilitaires comme nous l'avons montré dans la présentation du logiciel SACOO. Aussi, comme en bien d'autres domaines c'est probablement dans le compromis que se cache la meilleure solution.

Dans le même ordre d'idées nous voudrions émettre une proposition concernant la didactique de la programmation. Si nous pensons que le paradigme objet tend à devenir presque incontournable pour le développement de projets de grande envergure, il faut bien reconnaître qu'il est assez mal adapté pour l'écriture de petits programmes du fait de l'atomisation de l'algorithme qu'il provoque. Or, on observe aussi que la programmation orientée objet nécessite de toute façon la maîtrise de l'algorithmique impérative "classique" (il faut bien "finalement" rédiger des algorithmes à l'intérieur des opérations !).

Il nous apparaît donc qu'il reste souhaitable d'aborder l'étude de la programmation (au sens le plus large) par l'apprentissage de la programmation impérative "classique" avant de s'intéresser explicitement au paradigme objet qui serait introduit progressivement afin d'offrir un moyen de structurer des projets plus complexes. Cette option n'exclut pas d'avoir, dès le début, une "mentalité orientée objet" qui cherche à faire apparaître les "classes" ou catégories d'objets manipulés par un programme et à mettre en évidence les procédures et fonctions qui leur sont associés.

Notre troisième réflexion concerne les limites du modèle que nous avons proposé. Nous cherchions à mettre en place un modèle orienté objet qui soit aussi simple et intuitif que possible afin de s'adresser à un large public. Petit à petit, nous avons voulu doter ce modèle de mécanismes puissants qui permettent entre autre de vérifier la cohérence des spécifications ou la génération automatique de la base de données et des codes compilables des programmes. Cette seconde exigence n'est cependant réalisable convenablement que si le modèle possède une définition formelle précise, ce qui tend à s'opposer aux objectifs initiaux de simplicité et d'intuitivité. Sans renier les choix qui ont été faits, il nous apparaît cependant clairement que tout développement futur du modèle devra se baser sur une redéfinition plus formelle des concepts et de leurs interrelations.

Enfin, nous ne pouvons clôturer ce travail sans adresser nos plus vifs remerciements à de nombreuses personnes qui, directement ou indirectement ont contribué à notre réflexion. Il s'agit tout spécialement de Monsieur Naji Habra, directeur de ce mémoire et de Monsieur Charles Duchâteau, directeur du CeFIS ainsi que de l'ensemble du corps professoral de l'Institut d'Informatique et des "étudiants" du CeFIS.

# Bibliographie


[AUBERT]
AUBERT J.-P., DIXNEUF P.
Conception et programmation par objets : Techniques, outils et applications
Masson, Paris, 1991.

[BODART]
BODART F., PIGNEUR Y.
Conception assistée des systèmes d'information : Méthodes, modèles, outils.
2e édition, Masson,Paris, 1989.

[BOUGHLAM]
BOUGHLAM A.
Analyse Orientée Objet
IRIN-IUT de Nantes, Nantes, 1992.

[BRUNET a]
BRUNET J.
A model for object-oriented analysis : Esprit II report n° BC.R.TS.T31.1
chapter XI
Business Class Project n°5311

[BRUNET b]
BRUNET J., CAUVET C., LASOUDRIS L.
Why using events in a high-level specification
in Entity-Relationship Approch : the case of conceptual modelling
Elsevier, North-Holland, 1991

[DELACHARLERIE a]
DELACHARLERIE A.
Introduction aux bases de données : 1. Conception d'une base relationnelle.
CeFIS, Facultés Universitaires N-D de la Paix, Namur, 1988.

[DELACHARLERIE b]
DELACHARLERIE A.
Apprentissage de la conception des bases de données : Une méthodologie de la méthode
in Actes du Colloque francophone sur la didactique de l'informatique.
EPI, Paris, 1988.

[DELOBEL]
DELOBEL C., LECLUSE C., RICHARD P.
Bases de données : des systèmes relationnels aux systèmes à objets
InterEditions, Paris, 1991.

[DUCHATEAU]
DUCHATEAU Ch.
Images pour programmer : Apprendre les concepts de base.
De Boeck-Wesmael, Bruxelles, 1990.

[FERBER]
FERBER J.
Conception et programmation par objets.
Hermes Publishing, Paris, 1991.

[LAI]
LAI M.
Conception orientée objet : Pratique de la méthode HOOD
Dunod, Paris, 1991.

[MASINI]
MASINI G., NAPOLI A., COLNET D., LEONARD D., TOMBRE K.
Les langages à objets : Langages de classes, langages de frames, langages d'acteurs.
InterEditions, Paris, 1991.

[MEYER]
MEYER B.
Conception et programmation par objets : Pour du logiciel de qualité
InterEditions, Paris, 1990.

[ROLLAND a]
ROLLAND C.
Conception des systèmes d'information (Notes de cours)
Chaire FRANQUI, Namur, 1992.

[ROLLAND b]
ROLLAND C., CAUVET C.
Modélisation conceptuelle orientée objet
INRIA, BD3, Lyon, 1991.

[ROLLAND c]
ROLLAND C., FLORY A.
La conception des systèmes d'information : état de l'art et nouvelles perspectives.
20ème anniversaire des MIAGE, INFORSID, 1990.

[SERNADAS a]
SERNADAS A., SERNADAS C., GOUVEIA P, RESENDE P., GOUVEIA J.
OBLOG : OBject-oriented LOGic, An informal introduction.
INESC, Lisboa, 1991.

[SERNADAS b]
SERNADAS C., GOUVEIA P., GOUVEIA J., SERNADAS A., RESENDE P.
The reification dimension in object-oriented data base design
in [SERNADASa].

[SERNADAS c]
SERNADAS et al.
Towards OBLOGLight : draft language reference
internal report 30-11-92.

[VANSTRAATEN]
VAN STRAATEN A.
Class(Y) : User Manuel
Integrated Development Corporation, Hampsead, 1991.