MASTER EN SCIENCES INFORMATIQUES

Conception et réalisation d'un environnement de développement d'applications bureautiques

Roberfroid, Eva-Marie; Steinier, Stéphane

Award date:
1991

Awarding institution:
Universite de Namur

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

Rue Grandgagnage, 21
B-5000 NAMUR (Belgium)

Conception et réalisation d'un environnement de développement d'applications bureautiques

Mémoire présenté en vue de l'obtention
du diplôme de Maître en Informatique

Eva-Marie Roberfroid
Stéphane Steinier

Promoteur : J-L. Hainaut

Année académique 1990-1991

Qu'il nous soit permis de remercier vivement notre promoteur, le professeur Jean-Luc Hainaut pour les conseils qu'il nous a apportés tout au long de la réalisation de ce travail.

Nous exprimons également notre reconnaissance au professeur Eric Dubois pour ses précieuses remarques concernant les techniques de spécification.

Enfin, nous remercions toutes les personnes qui ont participé de près ou de loin à l'achèvement de ce travail.

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue de Bruxelles 61, B-5000 NAMUR
Tél. 081-72.41.11
Télex 59222 facnam-b
Téléfax 081-23.03.91

Conception et réalisation d'un environnement de développement d'applications bureautiques

Eva-Marie Roberfroid
Stéphane Steinier

Résumé

Dans les bureaux, la classification et l'accès rapide à l'information reste un problème sans réponse. Le but de ce travail est de développer un modèle permettant de représenter les objets du bureau à savoir les informations, les collections et les objets de rangement et ensuite, construire un environnement de développement permettant de résoudre ce problème. L'analyse de l'environnement de bureau est entièrement réalisée selon l'approche orientée objets. Suite à la phase de conception, une boîte à outils d'objets bureautiques destinée au développement d'applications pour micro-ordinateurs est obtenue. Un langage de programmation orienté objets et un gestionnaire de bases de données de type réseau constituent les supports de la phase d'implémentation. Des techniques de transformations ont dû être utilisées afin de rendre le schéma conceptuel des données conforme aux outils choisis. Finalement, l'interface entre ces deux outils très différents est étudiée en détails.

Abstract

In an office, environment classification and quick access to information still stay an unsolved problem. The aim of the work is to build a model to represent the main office objects (information, collections and storage facilities) and to build a development framework in order to help solving this problem. Analysis of the office environment is undertaken according to the object oriented approach. Design phase finally provides an office automation objects toolbox able to develop applications for small computers. The implementation is supported by an object oriented language and a network data bases system. Transformation technics are used to merge the data model with the implementation tools We also pay attention to the behaviour of the interface between these different tools.

Mémoire de maîtrise en Informatique
Septembre 1991
Promoteur : J-L. Hainaut

Table des matières

Chapitre 3 : Analyse conceptuelle d'un environnement de bureau

Chapitre 6 : Implémentation de la boîte à outils d'objets bureautiques

Introduction

1. Cadre général de travail

"La bureautique désigne l'assistance aux travaux de bureau, procurée par des moyens et des procédures faisant appel aux techniques de l'informatique, des télécommunications et de l'organisation administrative et, de façon générale, à tout ce qui concourt à la logistique du bureau et de son environnement. Plus conceptuellement, la bureautique intéresse le système d'information individuel de toute personne travaillant dans un bureau, sans exiger d'elle d'autres connaissances que celles de son savoir-faire professionnel."[BLAS-82]

L'analyse du champ d'application couvert par la définition qui précède se révèle très vaste. Il nous permet de comprendre pourquoi ces dernières années se caractérisent par l'apparition en masse de l'informatique dans le monde de la bureautique. Les nouveautés technologiques apportées par la bureautique moderne ont un impact économique incontestable dans un domaine resté jusqu'alors peu sujet aux changements. Ce mémoire s'inscrira donc dans une longue lignée de travaux ayant pour cadre le domaine de la bureautique.

Aujourd'hui, l'automatisation du travail de bureau ne concerne plus uniquement les tâches administratives qui y sont accomplies comme par exemple, le calcul de la paie des salariés d'une société, mais s'attaque également à des tâches individuelles telles que la gestion de documents personnalisés sur support informatique. En particulier, la bureautique intègre "les systèmes d'archivage et d'accès aux documents qui constituent l'élément "mémoire" de la bureautique"[BLAS-82]. Ces systèmes ont donc pour but de mémoriser et retrouver les informations à savoir : les dossiers, lettres, contrats,...

Les fonctions principales des bureaux restent donc de produire, d'exploiter ou de distribuer de l'information. Dans ce domaine, une des difficultés qui subsistent est d'organiser l'accès rapide à une masse d'informations de plus en plus colossale. Dès lors, que les supports utilisés pour stocker l'information soient électroniques ou non, la question principale reste identique : "à quel endroit faut-il accéder pour trouver l'information recherchée ?".

C'est dans le cadre de l'automatisation des problèmes relatifs au classement de l'information et à l'accès à l'information rangée que se situe ce mémoire. En recherchant un modèle permettant de représenter les structures et les fonctions associées à l'environnement de rangement d'informations d'un bureau, nous espérons apporter une réponse à cette question. Notre but final est de produire une boîte à outils permettant de construire des applications destinées à gérer des environnements de rangement d'informations.

Comme l'environnement bureautique est mouvant et que les objets et les techniques utilisées pour ranger l'information varient fortement d'une organisation à l'autre, nous voulons accorder une attention particulière à l'adaptabilité de la boîte à outils produite de manière à pouvoir représenter des environnements très divers.

2. Processus et environnement de développement

Dernièrement, une évolution très importante s'est fait sentir dans le domaine de la conception de systèmes informatiques. L'approche purement fonctionnelle de développement de systèmes cède peu à peu la place à une nouvelle méthodologie qui connaît actuellement une vive expansion : le développement orienté objets. Certains concepteurs y voient le remède miracle à tous les problèmes de conception rencontrés à ce jour.

D'une manière plus réaliste, cette méthode de développement semble être beaucoup plus souple que l'approche fonctionnelle. Elle permet le développement aisé de systèmes complexes et peut facilement être couplée avec une approche par prototypage. La maintenance des logiciels construits selon cette approche est aussi grandement simplifiée grâce aux propriétés naturelles qu'elle leur procure. Un des grands mérites de cette méthode est notamment la flexibilité des systèmes produits en l'utilisant. Comme cette dernière caractéristique correspond à l'objectif d'adaptabilité que nous nous sommes fixé, la méthodologie de développement orienté objets sera adoptée.

Le lien entre l'environnement bureautique et l'approche orientée objets est quasi immédiat. En effet, rien qu'en observant distraitement un bureau, les objets et les services qui leur sont associées peuvent aisément être identifiés. Les objets sont, entre autres, la table de travail, une information, une armoire et un type de service rendu serait le rangement de l'information dans l'armoire. La transposition n'est, en fait, pas aussi triviale mais cet exemple a le mérite de montrer qu'il serait intéressant de procéder à l'analyse complète du bureau et de son environnement immédiat en termes objets.

Une fois la méthode de développement choisie, pour pouvoir développer la boîte à outils, il faut trouver une technique permettant de garder une trace persistante des objets d'une exécution à l'autre des applications utilisant celle-ci. Pour ce faire, nous avons opté pour un système de gestion de base de données.

La conception de la boîte à outils consistera à décrire l'environnement de rangement du bureau en termes d'objets bureautiques et ensuite, à concevoir et à implémenter une solution exécutable grâce à un S.G.B.D et un environnement de programmation particulier.

Le développement comprendra la production d'un schéma conceptuel orienté objets constituant la mémoire du système ainsi que la conception d'une architecture de modules spécifiées en termes d'objets et de services rendus par ces objets (voir figure I.1).

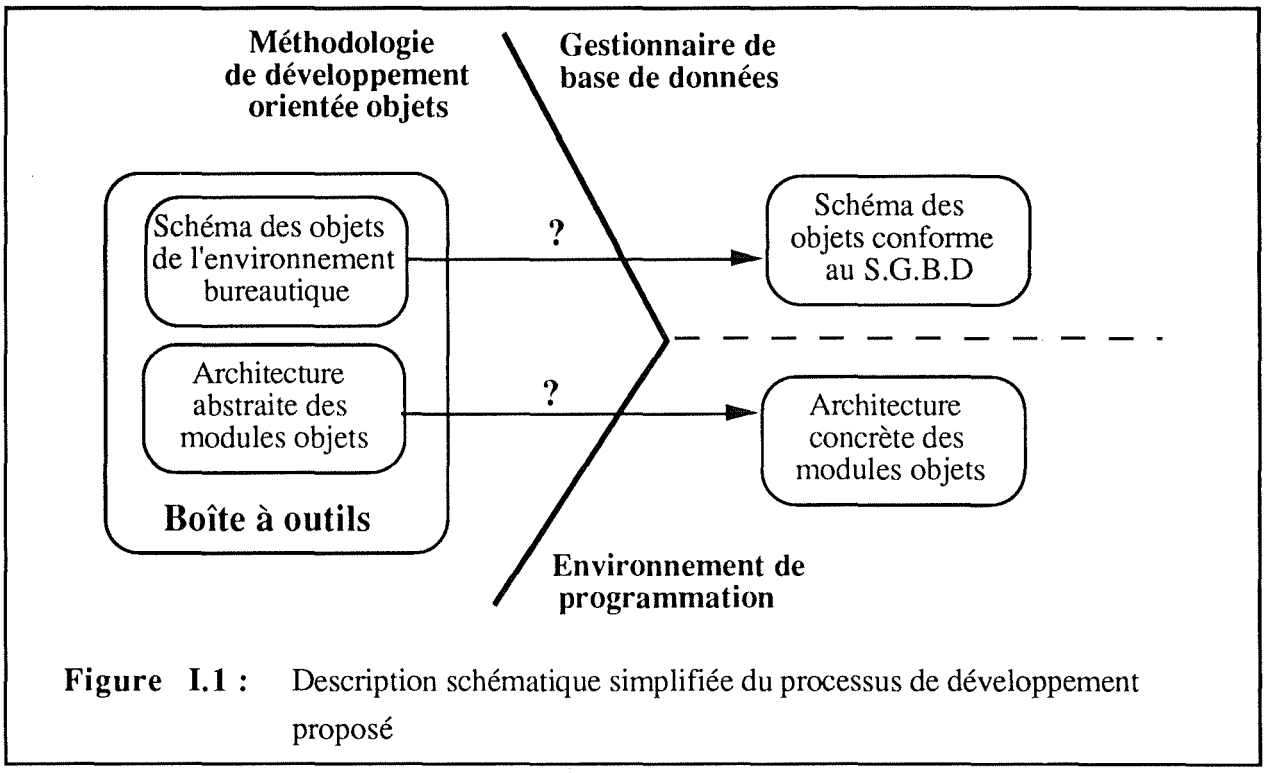

Figure I.1 : Description schématique simplifiée du processus de développement proposé

Une fois l'étude théorique du modèle terminée, il restera à implémenter la solution proposée grâce au gestionnaire de base de données et au langage de programmation cibles. Pour ce faire, des transformations devront être appliquées au schéma des objets et à l'architecture des modules du système initial pour les rendre conformes aux outils choisis. Elles dépendent évidemment des environnements retenus.

3. Originalité de ce travail

Une des originalités de ce mémoire relève du choix des outils utilisés pour implémenter la boîte à outils. Le gestionnaire de base de données choisi est le gestionnaire imposé à savoir, N.D.B.S. Dès lors, le langage de programmation utilisé devait être le PASCAL.

Alors que N.D.B.S. est un gestionnaire de base de données dit de type traditionnel (non orienté objets), le langage choisi sera le TURBO PASCAL version 5.5 qui est une version du langage PASCAL incluant les concepts de la programmation orientée objets (voir figure I.2).

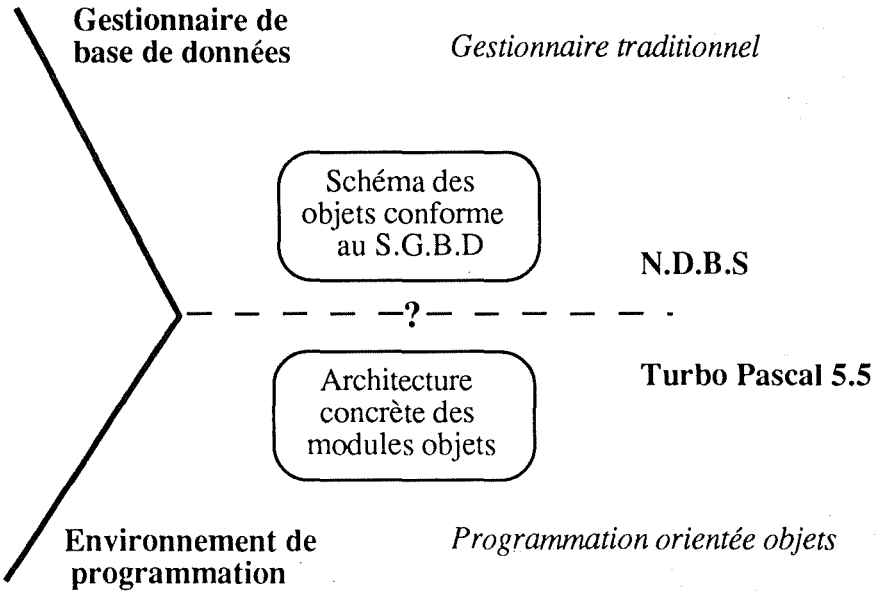

Figure I.2. : Confrontation de la programmation orientée objets et d'un gestionnaire de base de données dit traditionnel

Le choix de ce langage de programmation reste donc dans la ligne de conduite que nous nous sommes tracée quant à l'approche orientée objets

Le principal attrait du choix de ces outils est que nous allons pouvoir étudier la confrontation de deux mondes totalement différents : un langage orienté objets et un gestionnaire base de données classique.

Le premier point intéressant à étudier sera les contraintes que l'emploi d'une base de données traditionnelle va impliquer sur l'implémentation des modules qui se fera conformément aux principes de l'approche orientée objets utilisés durant la phase de développement.

Le deuxième point portera sur l'examen de l'extensibilité du système final suite au mariage de ces deux environnements totalement différents. Sera-t-il aussi aisément extensible que s'il avait été entièrement conçu à l'aide d'outils orientés objets c'est-à-dire en remplaçant le gestionnaire de bases de données traditionnel par un gestionnaire orienté objets.

Un autre attrait important lié à l'utilisation de ces outils est l'étude des transformations qui devront être appliquées au schéma et à l'architecture abstraite pour les rendre conformes aux outils utilisés.

Le PASCAL orienté objets est un langage hybride qui n'inclut pas tous les concepts faisant la puissance et l'originalité des langages orientés objets. L'architecture des modules devra donc être modifiée pour devenir conforme au langage utilisé. Les transformations possibles sont encore mal connues et nous essayerons donc d'apporter quelques éléments de réponse quant aux techniques de transformation applicables.

De même, le schéma des objets de l'environnement bureautique devra être transformé de manière à fournir une version non orientée objets implémentable grâce à N.D.B.S. C'est évidemment à ce niveau que se situera l'essentiel des transformations à réaliser. L'application des techniques de transformation existantes à un exemple concret nous permettra d'obtenir quelques renseignements pertinents sur ces dernières : respect de la richesse sémantique initiale, difficultés de mise en oeuvre, étude comparée des avantages et inconvénients de chacune d'elles.

4. Organisation du travail

Avant de procéder au développement proprement dit de la boîte à outils d'objets bureautiques, il nous semble opportun de décrire les concepts propres à l'approche orientée objets. Ils nous fourniront les outils indispensables pour poser le problème à résoudre et décrire la solution proposée en des termes précis.

Cette description tiendra en deux parties faisant chacune l'objet d'un chapitre. Le premier chapitre expliquera les langages orientés objets. Ils constituent les bases qui, quelques années plus tard, ont donné naissance à la méthodologie de développement de systèmes informatiques orientée objets. Dans un premier temps, les langages orientés objets seront abordés en toute généralité par la présentation de leurs concepts fondamentaux. Ensuite, nous traiterons de l'incorporation de ces concepts dans l'environnement de programmation orienté objets qui sera utilisé lors de la phase d'implémentation, à savoir le TURBO PASCAL version 5.5.

Le deuxième chapitre traitera de la méthodologie de développement orientée objets proprement dite. Tout d'abord, un petit compte rendu des différentes approches de développement de systèmes informatisés sera présenté. Ensuite, la deuxième section du chapitre sera consacrée à la description détaillée de l'approche orientée objets et de ses nombreuses propriétés. Finalement, pour clôturer ce chapitre, la méthodologie utilisée dans le cadre de notre travail sera présentée. Nous montrerons comment la méthode générale de développement orientée objets peut être adaptée au cas du développement d'une boîte à outils d'objets bureautiques et comment le prototypage peut être intégré dans chaque phase du développement.

Après ces deux premiers chapitres introduisant les outils fondamentaux, le développement proprement dit de la boîte à outils pourra commencer. Il sera subdivisé en plusieurs phases qui correspondent chacune à un chapitre distinct. Ainsi, le texte abordera successivement l'analyse d'un environnement de bureau, la conception logique de la boîte à outils, sa conception physique et l'implémentation de celle-ci à l'aide du S.G.B.D. et du langage de programmation choisis.

Pour utiliser pleinement la méthode de prototypage proposée, les phases du développement seront décomposées en plusieurs étapes qui chacune auront un effet rétroactif sur les étapes qui l'ont précédée. Nous espérons que cette démarche nous permettra d'atteindre un système donnant pleinement satisfaction quant au critère que nous nous sommes fixé c'est-à-dire une grande souplesse du modèle orienté objets utilisé pour représenter l'environnement de bureau. Cette souplesse est requise afin de pouvoir facilement étendre ou réutiliser la totalité ou une partie le système construit dans un projet présentant des similitudes avec le travail que nous avons fourni.

Une description plus approfondie du contenu des chapitres relatifs aux phases de développement ne peut être réalisée que si la méthodologie de développement utilisée dans le cadre de ce travail est connue. Il faudra donc attendre la fin du chapitre deux pour la réaliser. Pour tenir compte de cette particularité, chacun des chapitres concernés par cette remarque commencera par une introduction présentant les étapes propres à la phase de développement qu'il aborde.

Afin de couvrir complètement le cycle de vie d'un système informatique, nous dédierons une partie de notre travail à la conception d'une petite application utilisant la boîte à outils d'objets bureautiques. Ce sujet sera abordé dans le chapitre six relatif à la phase d'implémentation. Les buts de cette partie seront d'étudier comment la couche logicielle fournie par la boîte à outils peut être intégrée dans une application de haut niveau, de voir si cette dernière est réellement facile à employer et d'en déduire qu'elle est l'extensibilité et la réutilisabilité réelle du système final.

En conclusion, nous tenterons de répondre aux nombreuses questions qui sous-tendent à la confrontation de deux mondes aussi différents que le développement orienté objets et les gestionnaires de bases de données traditionnels et de donner quelques idées sur les études futures qui pourraient être entreprises afin de compléter notre travail.

Chapitre 1 :
Les langages orientés objets

Introduction

Ce premier chapitre sera destiné à introduire les concepts propres aux langages orientés objets. La description portera principalement sur les notions qui nous seront utiles par la suite afin de poser clairement le problème à résoudre et ensuite, exprimer la solution proposée en des termes corrects.

La première section **Concepts fondamentaux d'un langage orienté objets** abordera les notions propres aux langages orientés objets.

La deuxième section **Présentation de l'environnement de programmation TURBO PASCAL version 5.5** traitera de l'incorporation de ces concepts dans le langage de programmation cible choisi pour implémenter la boîte à outils. A ce stade, l'on pourra déjà se faire une idée du type de transformations que devra subir l'architecture abstraite des modules pour devenir conforme au langage PASCAL orienté objets.

1. Concepts fondamentaux d'un langage orienté objets

Une approche orientée objets est basée sur cinq concepts : l'objet, la classe d'objets, l'héritage, le polymorphisme et la liaison dynamique [MGKO-90]. Les trois premiers concepts apparaissent dès la phase de conception de l'application (haut niveau) et les deux derniers sont liés à une implémentation d'un langage orienté objets (bas niveau).

1.1. L'objet

Les **objets** sont les entités de base des systèmes orientés objets. Tout concept manipulé est représenté sous forme d'un objet. Par exemple, le nombre π peut être représenté par un objet.

A chaque objet est associé un ensemble de procédures, appelées **méthodes**, qui définissent les opérations applicables à l'objet. L'ensemble des méthodes s'appelle l'**interface** de l'objet. Par exemple, pour le nombre π, l'interface associée à l'objet le représentant pourrait se composer des différentes opérations algébriques associables aux nombres réels. Dans chaque objet, il y a donc mariage des données permettant de représenter l'objet et des méthodes assurant la manipulation de ces données. Cette fusion de données et de méthodes est appelée **encapsulation**.

1.2. La classe d'objets

Une **classe d'objets** définit un ensemble possible d'objets. Reconsidérant l'exemple du nombre π, les différentes opérations associées à l'interface de cet objet peuvent également être appliquées aux autres nombres réels, une classe d'objets représentant les nombres réels peut ainsi être créée. Dès lors, de par la définition du concept de classe d'objets, tous les objets faisant partie de la même classe d'objets contiennent une information de même type et ont la même interface.

Chaque occurrence d'une classe d'objets est appelée **instance** de cette classe. Par exemple, le nombre π est une instance de la classe des nombres réels.

1.3. L'héritage

La notion d'**héritage** est un concept riche et prometteur. Il permet de concevoir des applications à partir de systèmes autonomes réutilisables composés de librairies de classes d'objets véhiculant le même concept (comme des librairies de fonctions mathématiques ou des librairies à vocation graphique par exemple) plutôt que de devoir réécrire l'application dans sa totalité.

En effet, la notion d'héritage permet une extensibilité aisée d'un système donné. De nouvelles classes d'objets peuvent être définies à partir de classes d'objets existantes ce qui diminue fortement la quantité de code nécessaire pour définir une nouvelle classe d'objets. De plus, un changement des propriétés d'une classe d'objets se répercute immédiatement sur toutes les classes héritant des propriétés de la classe modifiée. Ceci facilite grandement la maintenance des applications développées à l'aide d'un langage orienté objets.

Pour comprendre exactement les mécanismes des relations d'héritage et pour pouvoir bénéficier pleinement des avantages qu'ils procurent, ces mécanismes seront présentés en détails à l'aide d'exemples.

Quand une classe d'objets Y hérite d'une classe d'objets X (voir figure 1.1), la classe d'objets X est appelée **classe de base** (elle est également nommée super-classe) et la classe d'objets Y est appelée **classe dérivée** (ou encore sous-classe). Dans ce cas précis, la classe d'objets Y peut être divisée en deux parties : une partie dérivée constituée des propriétés **héritées** de la classe d'objets X et une partie incrémentale spécifique à la classe d'objets Y. Cette partie incrémentale constitue l'**enrichissement** des propriétés de la classe d'objets Y. Ce mécanisme porte le nom d'**héritage simple**.

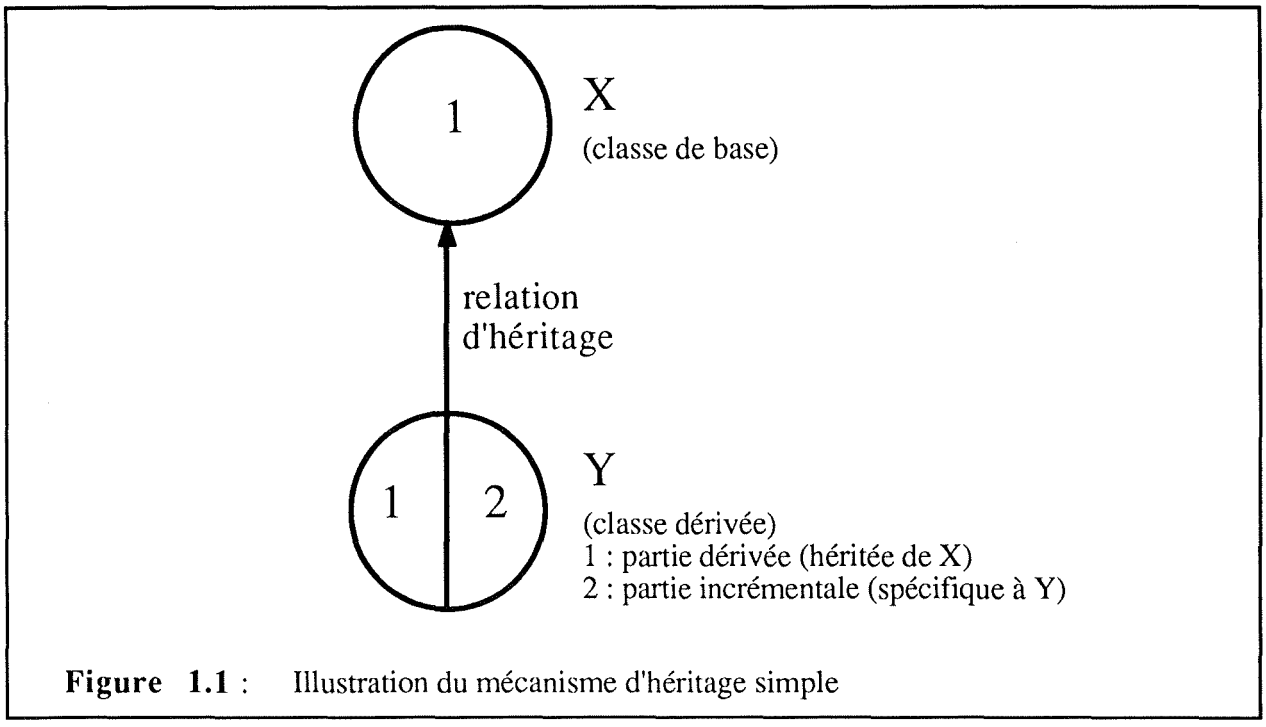

Figure 1.1 : Illustration du mécanisme d'héritage simple

Les relations entre les propriétés propres à la classe d'objets X et celles décrites au niveau de la partie dérivée de la classe d'objets Y dépendent du langage orienté objets utilisé. Cette relation peut

être une simple relation d'identité : dans ce cas, la partie dérivée de la classe d'objets Y est celle de la classe d'objets X. Mais, il va de soi que cette relation peut être plus riche qu'une simple relation d'identité. En général, une propriété de la classe d'objets X peut être réimplémentée, renommée, dupliquée, inhibée ou voir sa portée modifiée.

La relation d'héritage est souvent appelée une relation **is-a**. La raison en est que lorsqu'une classe d'objets Y est dérivée d'une classe d'objets X, cette classe d'objets Y hérite de toutes les propriétés de la classe d'objets X ce qui implique que tout objet O appartenant à la classe d'objets Y est également une instance de la classe d'objets X. Cette relation d'inclusion de la classe d'objets Y dans la classe d'objets X est présentée à la figure 1.2.

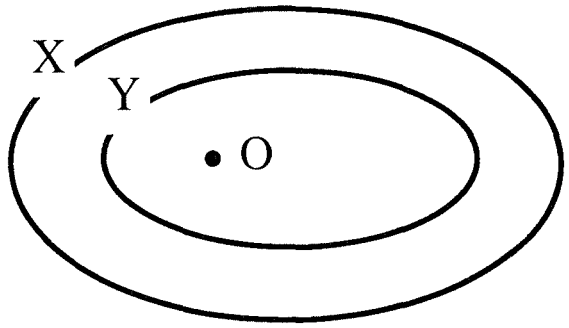

Figure 1.2 : Représentation ensembliste de l'inclusion de classes d'objets dans le cas d'une relation d'héritage simple

Il est évident que le mécanisme d'héritage n'est pas restreint à l'héritage simple, il peut être étendu pour arriver à la notion d'héritage multiple.

Dans l'exemple illustrant la relation d'**héritage multiple** (voir figure 1.3), la classe d'objets Y hérite à la fois des propriétés des classes d'objets X et X'. Comme dans le cas d'une relation d'héritage simple, la classe d'objets héritière peut être divisée en autant de parties dérivées qu'elle ne possède de classes d'objets de base et en une partie qui lui est propre. Dans l'exemple présenté, la classe d'objets Y possède une partie dérivée de la classe d'objets X, une partie dérivée de la classe d'objets X' et une partie spécifique.

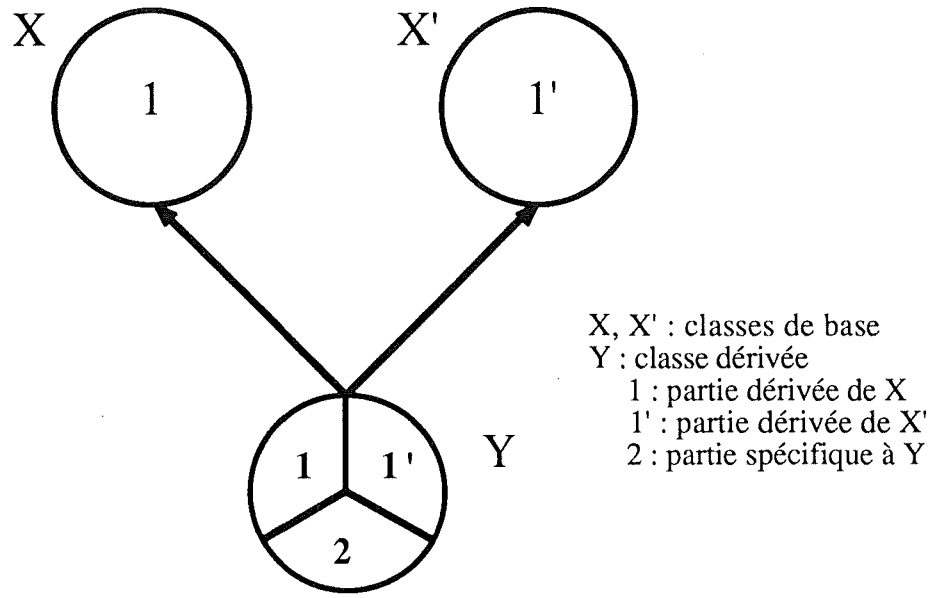

Figure 1.3 : Illustration du mécanisme d'héritage multiple

Si les classes d'objets illustrées par la figure 1.3 sont représentées selon le formalisme ensembliste déjà utilisé, on obtient le schéma de la figure 1.4. Dans ce schéma, la classe d'objets Y ne couvre pas l'entièreté de l'intersection des ensembles représentant les classes d'objets X et X' puisqu'il est aisé d'imaginer des objets héritant des propriétés des classes d'objets X et X' et n'appartenant pas à la classe d'objets Y.

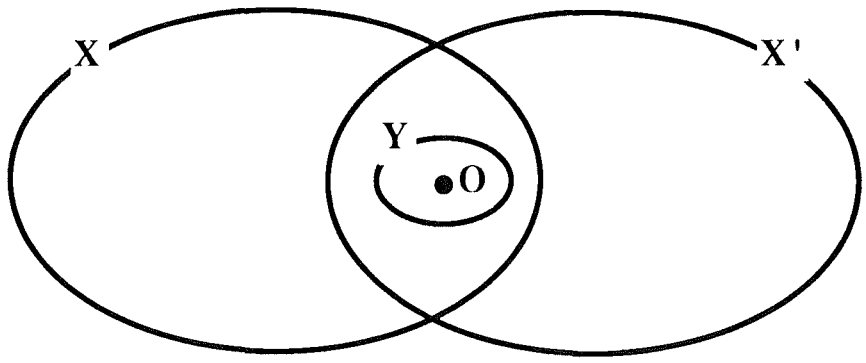

Figure 1.4 : Représentation ensembliste de l'inclusion de classes d'objets dans le cas d'une relation d'héritage multiple

Les caractéristiques qui sont réellement attractives et puissantes dans le mécanisme d'héritage sont que le programmeur peut réutiliser une classe qui est presque mais, pas exactement ce qu'il recherche. Il peut modeler cette classe d'une manière telle qu'elle n'introduise pas des effets de bord

indésirables dans la hiérarchie d'objets qu'il est en train d'implémenter. De plus, le concept d'objet se focalise sur l'identification et l'encapsulation de propriétés communes à différents objets dans des classes d'objets d'un haut niveau d'abstraction. Si ces classes d'un haut niveau d'abstraction sont accumulées, il est alors facile lors du développement d'une nouvelle application de trouver des classes pouvant supporter la construction de cette application.

1.4. Le polymorphisme

Plusieurs définitions de la notion de polymorphisme peuvent être recensées. En général, le terme de **polymorphisme** rend compte de la capacité de quelqu'un ou de quelque chose de prendre plusieurs formes.

Dans un langage orienté objets, une **référence polymorphique** est une référence à un objet dans un programme qui, durant l'exécution de ce programme, peut appartenir à plusieurs classes d'objets différentes. A cause de cette propriété, une référence polymorphique est associée à la fois à un type d'objet statique et à un type d'objet dynamique.

Le **type d'objet dynamique** d'une référence polymorphique peut varier d'une instruction à l'autre durant l'exécution du programme.

Le **type d'objet statique** d'une référence polymorphique est déterminé par la déclaration de l'objet dans le texte du programme. Il est connu au moment de la compilation et détermine l'ensemble des classes d'objets valides que cet objet peut accepter au moment de l'exécution du programme, autrement dit les valeurs que peut prendre le type dynamique de la référence polymorphique. Cette détermination est faite à partir de la hiérarchie des classes d'objets du système.

La relation is-a (relation d'héritage) est fortement couplée avec la notion de polymorphisme. L'idée est que si la classe d'objets Y hérite de la classe d'objets X, toute instance de la classe d'objets Y fait également partie de la classe d'objets X et partout dans le programme où une instance d'un objet de la classe d'objets X est attendue, une instance de la classe d'objets Y est permise.

1.5. La liaison dynamique

Dans un langage orienté objets, le concept de liaison dynamique est fortement lié aux notions d'héritage et de polymorphisme. Ce concept permet de résoudre le problème suivant : quel code faut-il exécuter lors de l'appel d'une procédure associée à une référence dynamique ?

Pour illustrer ce principe, prenons l'exemple d'une référence polymorphique R qui a un type statique X et qui peut être de type dynamique X ou Y. De plus, supposons que la classe d'objets X possède une procédure P. Si cette procédure a été redéfinie et réimplémentée au niveau de la classe d'objets Y, la procédure appelée varie suivant le type dynamique de la référence polymorphique R. Ce choix ne peut se faire qu'à l'exécution et est résolu grâce au concept de liaison dynamique qui permet de déterminer le type dynamique de la référence polymorphique R.

2. Présentation de l'environnement de programmation TURBO PASCAL version 5.5

La présentation des concepts de base étant terminée, il reste à décrire la manière dont ces concepts sont intégrés dans le langage de programmation cible.

L'environnement de programmation TURBO PASCAL version 5.5 est une implémentation du langage PASCAL défini par Jensen et Wirth qui inclut les concepts de la programmation orientée objets. Cette implémentation du PASCAL respecte les règles de la programmation orientée objets présentées ci-dessus bien qu'elle présente certaines particularités, voire certaines restrictions, que nous nous devons d'aborder ici.

2.1. Structure de l'héritage

Le TURBO PASCAL version 5.5 ne permet l'utilisation que de structures d'héritage simple. Cette restriction implique la transformation de toute structure d'héritage multiple à implémenter en une structure d'héritage simple correspondante. Celle-ci est réalisable grâce à la redéfinition des différentes méthodes d'un objet à tout niveau de la hiérarchie de classes d'objets.

2.2. Implémentation de la liaison dynamique [LACH-89]

Le fait que le TURBO PASCAL version 5.5 soit un langage compilé et que le type dynamique d'une référence polymorphique ne puisse être évalué qu'à l'exécution conduit à un problème pour connaître la liaison dynamique de cette référence polymorphique. La solution choisie par les concepteurs de cet environnement de programmation sont les **méthodes virtuelles**. Cette solution consiste à renvoyer tout appel de méthode déclarée virtuelle vers un emplacement déterminé de la mémoire appelé **table des méthodes virtuelles** d'une classe d'objets. Dans cette table qui comprend l'adresse du point d'entrée de toutes les méthodes déclarées virtuelles pour une classe d'objets, quelques instructions supplémentaires permettent d'aller chercher l'adresse de la procédure à appeler et ensuite, de réaliser l'appel de cette procédure.

Du point de vue technique, tout objet possède un champ supplémentaire, caché de l'utilisateur, constitué d'un pointeur vers la table des méthodes virtuelles de la classe dont cet objet est une instance. Il est alors facile de comprendre comment la liaison dynamique est réalisée : la valeur du pointeur est lue et ensuite, l'adresse de la méthode souhaitée est déduite de la table des méthodes virtuelles (voir figure 1.5).

Les tables des méthodes virtuelles des diverses classes d'objets possédant des méthodes virtuelles sont construites au moment de la compilation et l'adresse de ces tables est placée dans le pointeur contenu dans chaque objet lors de son initialisation à l'aide d'une méthode spéciale appelée **constructeur**. Cette méthode est en fait, identique aux autres méthodes de la classe d'objets exception faite que son appel provoque l'initialisation du pointeur de l'objet vers la table des méthodes virtuelles de la classe d'objets. Elle doit être appelée une et une seule fois pour chaque instance de la classe d'objets, avant toute utilisation de celle-ci.

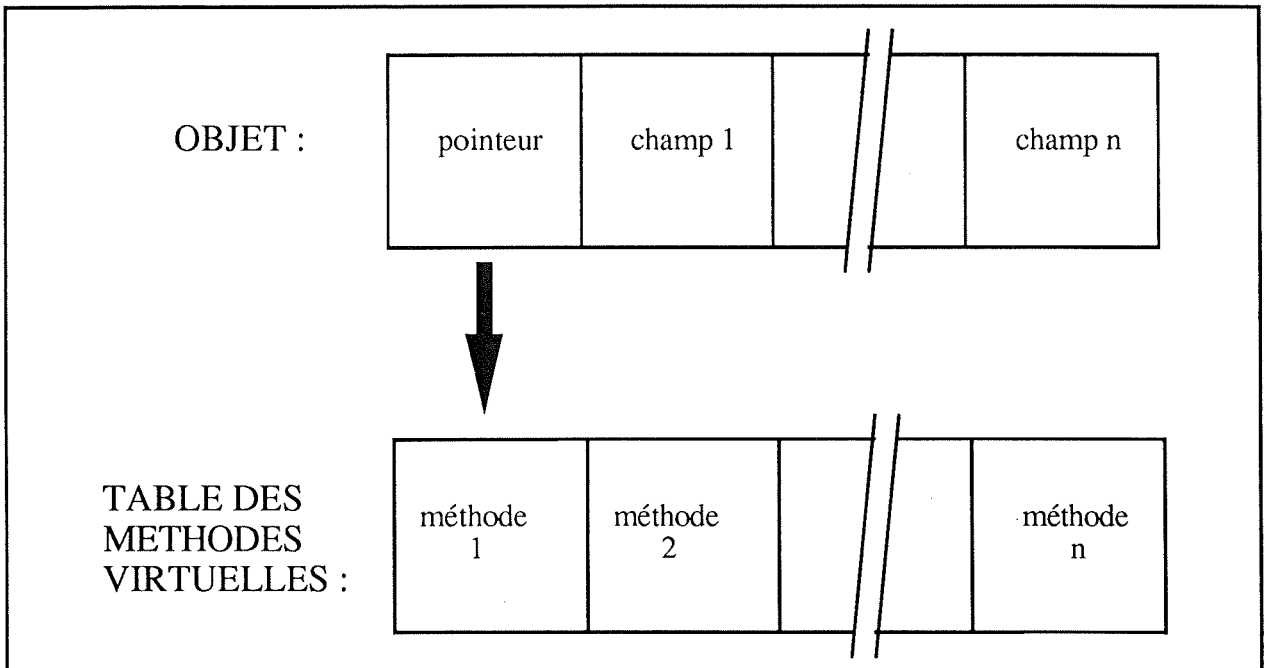

Figure 1.5 : Illustration de la table des méthodes virtuelles associée à une classe d'objets, mécanisme réalisant la liaison dynamique

Comme pendant des méthodes appelées constructeurs, il existe des méthodes nommées **destructeurs**. Ces méthodes n'ont en fait aucune particularité et doivent seulement leur existence à celle des constructeurs. Il est cependant conseillé d'utiliser de telles méthodes pour la libération de l'instance de la classe d'objets et des ressources mémoire qu'elle pourrait occuper.

2.3. Syntaxe du TURBO PASCAL version 5.5

Après avoir introduit les concepts de la programmation orientée objets tels qu'ils sont implémentés dans l'environnement de programmation TURBO PASCAL version 5.5, il est intéressant de présenter la syntaxe utilisée pour la description de ces concepts. Ainsi, les utilisateurs avertis des versions du langage PASCAL non orientées objets pourront aisément savoir comment

utiliser la version orientée objets de PASCAL fournie par la maison d'édition de logiciels BORLAND [BORL-89].

2.3.1. Déclaration d'une classe d'objets

La déclaration d'une classe d'objets en TURBO PASCAL version 5.5 revient à définir un nouveau type de données à l'aide du mot réservé **OBJECT**. Cette déclaration se fait donc dans la section TYPE du programme source. La structure d'une classe d'objets (voir figure 1.6) s'apparente fortement à celle d'un RECORD pour ce qui est de la définition des données de la classe d'objets mais, à cette structure viennent s'ajouter les procédures et fonctions faisant partie de l'interface de l'objet. Les structures d'héritage qui sont simples dans cette implémentation du langage PASCAL, sont construites en indiquant après le mot réservé OBJECT la classe d'objets de base de la classe déclarée. Les déclarations des procédures et des fonctions sont celles définies par la syntaxe du langage PASCAL. Lors de l'implémentation des méthodes d'une classe d'objets, qui se fait en dehors de la déclaration de cette classe d'objets, il faut faire précéder le nom de la méthode par le nom de la classe d'objets à laquelle elle appartient. Il est cependant à remarquer que l'accès aux données de la classe d'objets peut se faire directement dans le code des méthodes faisant partie de l'interface de la classe d'objets.

2.3.2. Procédures virtuelles, constructeurs et destructeurs

Les procédures virtuelles, astuce d'implémentation qui apporte une solution au problème de liaison dynamique, sont déclarées en faisant suivre l'en-tête de la fonction ou de la procédure de la classe d'objets par le mot réservé **VIRTUAL** [SHAM-89].

Les constructeurs et les destructeurs sont des procédures permettant de délimiter le cycle de vie d'une variable objet, instance d'une classe d'objets possédant des méthodes déclarées virtuelles. Les mots réservés utilisés pour déclarer ces procédures sont respectivement **CONSTRUCTOR** et **DESTRUCTOR**. Dans le cas d'un constructeur, l'échec de l'initialisation d'une variable objet doit être reportée par le mot réservé **FAIL** qui signifie au programme que l'initialisation du pointeur vers la table des méthodes virtuelles de la classe d'objets a échoué.

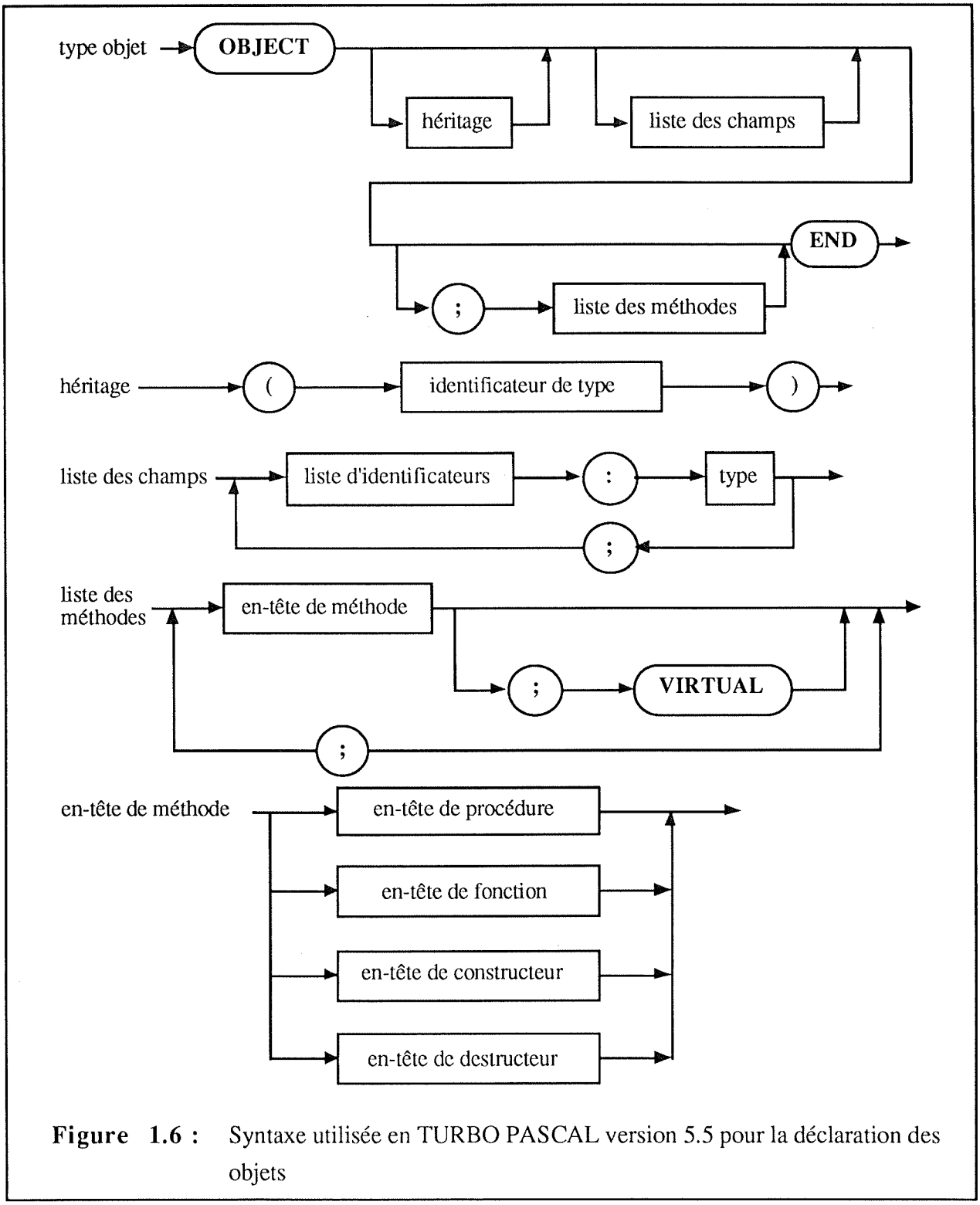

Figure 1.6 : Syntaxe utilisée en TURBO PASCAL version 5.5 pour la déclaration des objets

2.3.3. Déclaration d'une variable objet et appel des méthodes d'une variable objet

La déclaration d'une variable objet se fait tout naturellement dans la section VAR du programme source, chose qui est déjà connue des utilisateurs du langage PASCAL.

L'appel des méthodes d'une variable objet particulière se réalise dans le programme en faisant précéder le nom de la méthode par le nom de la variable objet considérée. Remarquons que cette technique est dérivée directement de celle utilisée pour accéder aux différents champs d'un enregistrement telle qu'elle est décrite dans le langage PASCAL conventionnel.

Chapitre 2 :
Méthodologie de développement de systèmes

Introduction

Les concepts de base propres à une approche orientée objets étant posés, nous pouvons maintenant aborder le problème que constitue le développement de systèmes informatiques.

Dans un premier temps, **l'approche classique et quelques approches alternatives** de développement de systèmes seront abordées.

Une fois ces principes posés, la **méthodologie de développement orientée objets** sera expliquée en détails puisque c'est cette approche qui a été choisie pour construire la boîte à outils travaillant sur l'environnement de bureau. Dans cette partie, l'intégration du prototypage dans la démarche orientée objets sera abordée. Finalement, les propriétés de l'approche orientée objets qui ont fait le succès de cette méthode de développement seront énoncées.

La dernière section de ce chapitre sera consacrée à la présentation de la **méthodologie utilisée dans le cadre de ce travail**. Les principaux points abordés seront les adaptations apportées à la méthodologie générale orientée objets et les étapes spécifiques importantes qu'elle comporte sur lesquelles il est bon dès à présent d'attirer l'attention du lecteur.

1. Approche classique et approches alternatives de développement

L'approche la plus courante de développement de logiciels est l'**approche fonctionnelle** qui utilise la décomposition fonctionnelle pour spécifier les différentes tâches à exécuter afin de produire une solution conforme à un problème précis. Le cycle de vie du système qui lui est associé est abondamment traité dans la littérature et le nombre d'étapes qu'il comporte varie d'un auteur à l'autre.

Nous ne retiendrons de cette approche que le schéma simplifié proposé à la figure 2.1 [HEED-90].

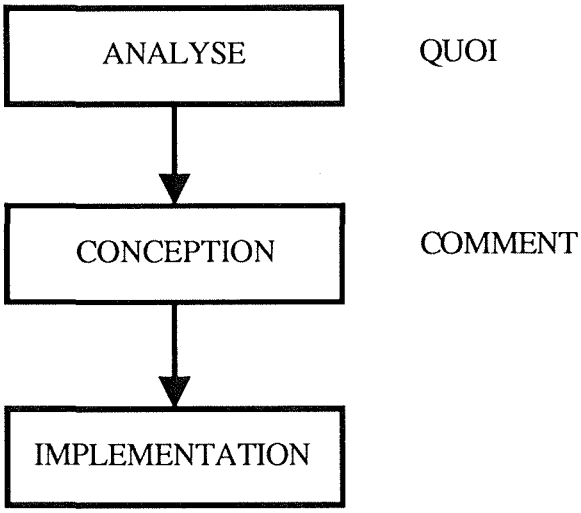

Figure 2.1 : Vue simplifiée du cycle de vie d'un système informatique

La phase d'**analyse** recouvre l'étude de faisabilité et l'analyse des besoins des utilisateurs aussi appelée analyse conceptuelle.

La phase de **conception** regroupe les étapes communément appelées conceptions globale, détaillée, logique et physique.

Quant à l'**implémentation**, elle traite de l'écriture des programmes dans un langage cible donné avec les vérifications, validations, tests, etc qui y sont associés.

La vue du problème purement procédurale se traduit dans le cycle de vie du système par le type de questions que le concepteur se pose. Par exemple, pourquoi le système est-il fait, quelles fonctions doit-il remplir ?

L'approche fonctionnelle est aussi une approche du type top-down qui lors de la recherche d'une solution dans la phase de conception (le comment) va du général vers le particulier.

Cette approche a beaucoup de succès et est celle qui reste, aujourd'hui, de loin la plus utilisée. Mais, le besoin de développer des applications de plus en plus importantes et de plus en plus complexes a mis en évidence les lacunes que comportait une approche purement procédurale et les points critiques devant être pris en compte par toute approche de développement aussi bien dans la phase de conception de l'application que dans sa phase d'implémentation. Ainsi, des approches de conception alternatives sont apparues.

Tout d'abord, l'**approche basée sur les objets**, aussi appelée approche structurée de Jackson, est une approche intermédiaire entre l'approche fonctionnelle et l'approche orientée objets (application des principes des langages orientés objets décrits au chapitre 1 au stade de conception). Dans cette approche, une attention plus grande est portée aux structures de données que dans l'approche fonctionnelle mais, la décomposition fonctionnelle reste toujours employée pour construire l'architecture du système [MGKO-90].

Bien que cette approche tente de prendre en compte les objets, la méthode de conception reste identique à celle utilisée dans l'approche classique.

Comme dans l'approche classique, la principale question reste celle des fonctions que doit remplir le système. Cette question légitime, qui conduit souvent à des systèmes qui remplissent bien leurs fonctionnalités, a pour principal inconvénient de produire une solution bien adaptée au problème à résoudre mais totalement inutilisable pour la résolution d'autres problèmes. De plus, toute modification apportée à l'architecture implique souvent des modifications en cascade (cet effet en cascade est aussi appelé effet "domino") s'étendant à de nombreux autres modules.

La troisième approche, l'**approche orientée objets** tente d'apporter une solution à ces problèmes en allant plus loin en ce sens qu'elle s'intéresse uniquement aux données et aux relations entre ces données qu'elle considère comme une partie fondamentale de l'architecture du système à développer [HEED-90]. Dans ce type d'approche, le système est vu comme une collection de classes d'objets. L'analyse est réalisée en termes des objets et des services que ceux-ci devront rendre au système. L'application est alors construite selon une démarche bottom-up dont les briques de base que sont les classes d'objets fournissent les services nécessaires à la construction des modules de plus haut niveau.

Ces trois approches ne sont évidemment pas rigides et statiques. Les publications les plus récentes proposent des méthodes de conception de systèmes utilisant plusieurs approches différentes durant les phases de cycle de vie du système (voir figure 2.2).

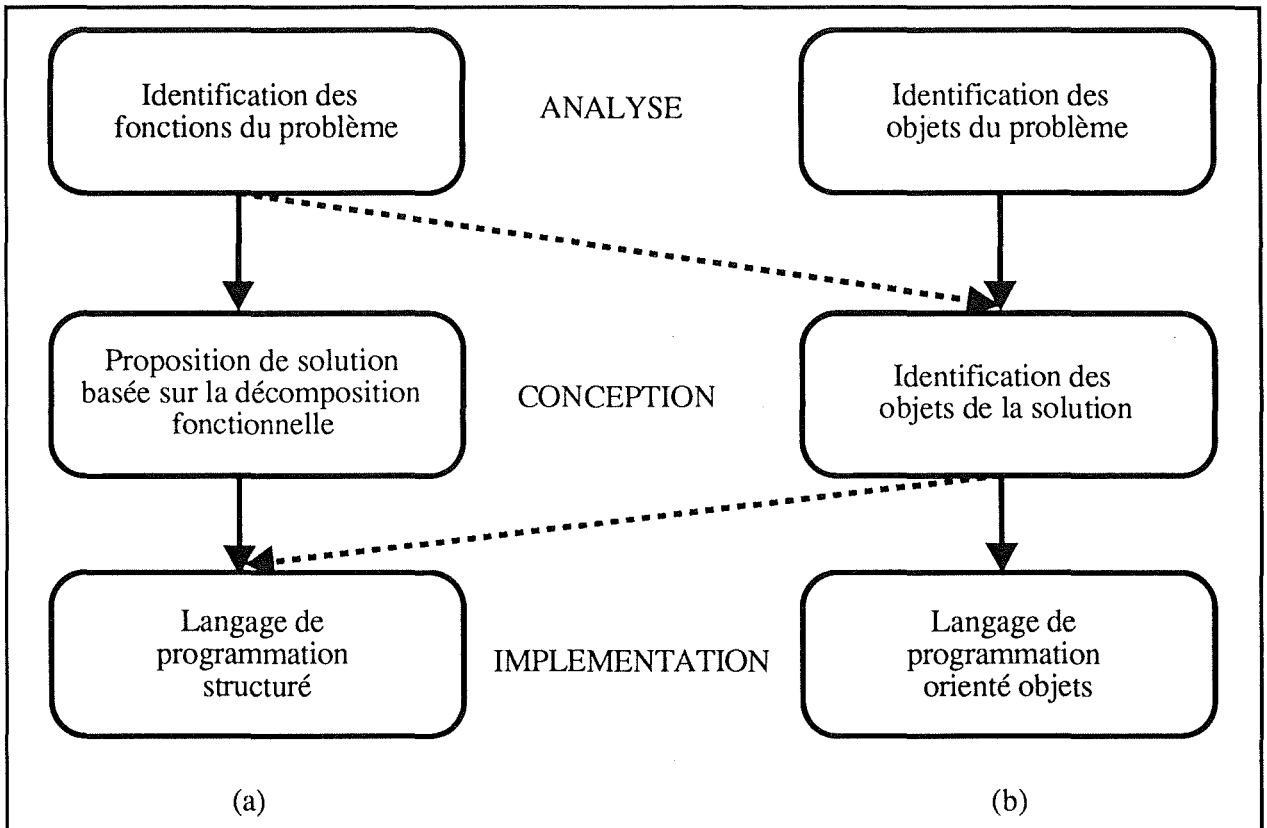

Figure 2.2 : Passage possible entre les approches fonctionnelle (a) et orientée objets (b) pendant les différentes phases du développement

Ainsi, par exemple, le cycle de vie complet d'un logiciel peut comprendre une phase de conception réalisée sous l'angle d'une approche orientée objets et une phase d'implémentation réalisée à l'aide d'un langage structuré ne disposant pas des concepts de la programmation orientée objets.

2. Méthodologie de développement orientée objets

Cette section est consacrée à la présentation détaillée des étapes propres à la démarche de conception des systèmes orientés objets. Le problème du prototypage y est aussi abordé. Finalement, nous verrons qu'elles sont les principales propriétés inhérentes à une approche orientée objets.

2.1. Description générale de la démarche

La première proposition d'une méthodologie de développement orientée objets a été faite par Booch. Bien qu'elle représentait, à l'époque, un grand pas en avant, elle restait fortement orientée vers la phase d'implémentation. La programmation orientée objets apparaissant, alors, comme se suffisant à elle seule, la phase de conception était quelque peu négligée.

Par la suite, cette méthodologie a été enrichie pour devenir une démarche à sept étapes exprimées de manière concise ci-dessous [HEED-90] :

1. Etude des exigences du système en termes d'objets.

2. Identification des objets et des services qu'ils doivent rendre.

3. Recherche des interactions existant entre les objets en termes de services offerts et demandés.

4. Prise en compte des aspects de conception.

5. Utilisation grâce au mécanisme bottom-up de librairies de classes d'objets.

6. Introduction des structures d'héritage entre classes pour construire le système.

7. Agrégation et généralisation des classes.

La première étape se situe à un haut niveau d'analyse. Au cours de celle-ci, l'étude des spécificités devant être respectées par le système se fait en termes d'objets.

L'étape suivante consiste à identifier les objets ou entités et leurs services ce qui se fait souvent par comparaison avec les objets du monde réel.

La recherche des interactions entre les objets qui constitue la troisième étape, est souvent réalisée grâce au formalisme Entité/Association. Les interactions trouvent leur origine dans le type de services que les objets rendent.

La quatrième étape consiste à prendre en compte les aspects internes propres aux objets de manière à se faire une idée sur les possibilités de construction de leurs méthodes (c'est-à-dire les services qu'ils devront rendre).

De là, par le mécanisme bottom-up, les classes élémentaires sont utilisées pour construire des objets plus complexes.

Finalement, l'introduction des relations d'héritage fournit le support nécessaire pour construire les objets du système.

La dernière étape consiste à itérer, sur le système final, les mécanismes d'agrégation et de généralisation déjà utilisés au cours des étapes précédentes pour remodeler les décisions prises au cours des étapes antérieures. En général, le système de classes d'objets produit subira plusieurs généralisations successives. Ainsi, les classes d'objets deviendront suffisamment générales pour constituer des outils utilisables par de nombreuses applications.

A ce stade, la technique de prototypage[1] peut être utilisée et procure aussi les feed-backs nécessaires vers les étapes de spécification pour construire les classes générales.

2.2. Le prototypage dans l'approche orientée objets

Le prototypage, souvent difficile dans une approche de conception classique car il conduit parfois à reconsidérer l'ensemble de l'architecture logicielle, devient naturel dans une approche orientée objets. En effet, l'indépendance entre les classes d'objets fait que les modifications à apporter sont mieux localisées et donc moins lourdes en conséquences sur l'architecture.

Pour pouvoir utiliser la technique de prototypage durant le cycle de vie de développement (comprenant les phases d'analyse, de conception et d'implémentation), Meyer propose d'utiliser le concept de cluster [MEYE-90]. Le **cluster** est un petit ensemble de classes d'objets possédant des interactions entre elles. Lors de l'étape d'analyse, chaque cluster est spécifié. Ensuite, les étapes de conception et d'implémentation sont appliquées à un cluster choisi parmi l'ensemble de ceux disponibles. Finalement, les classes d'objets du cluster peuvent être validées et généralisées (voir figure 2.3).

---

[1] Le prototypage est défini comme une première version d'un système qui en montre les caractéristiques essentielles.

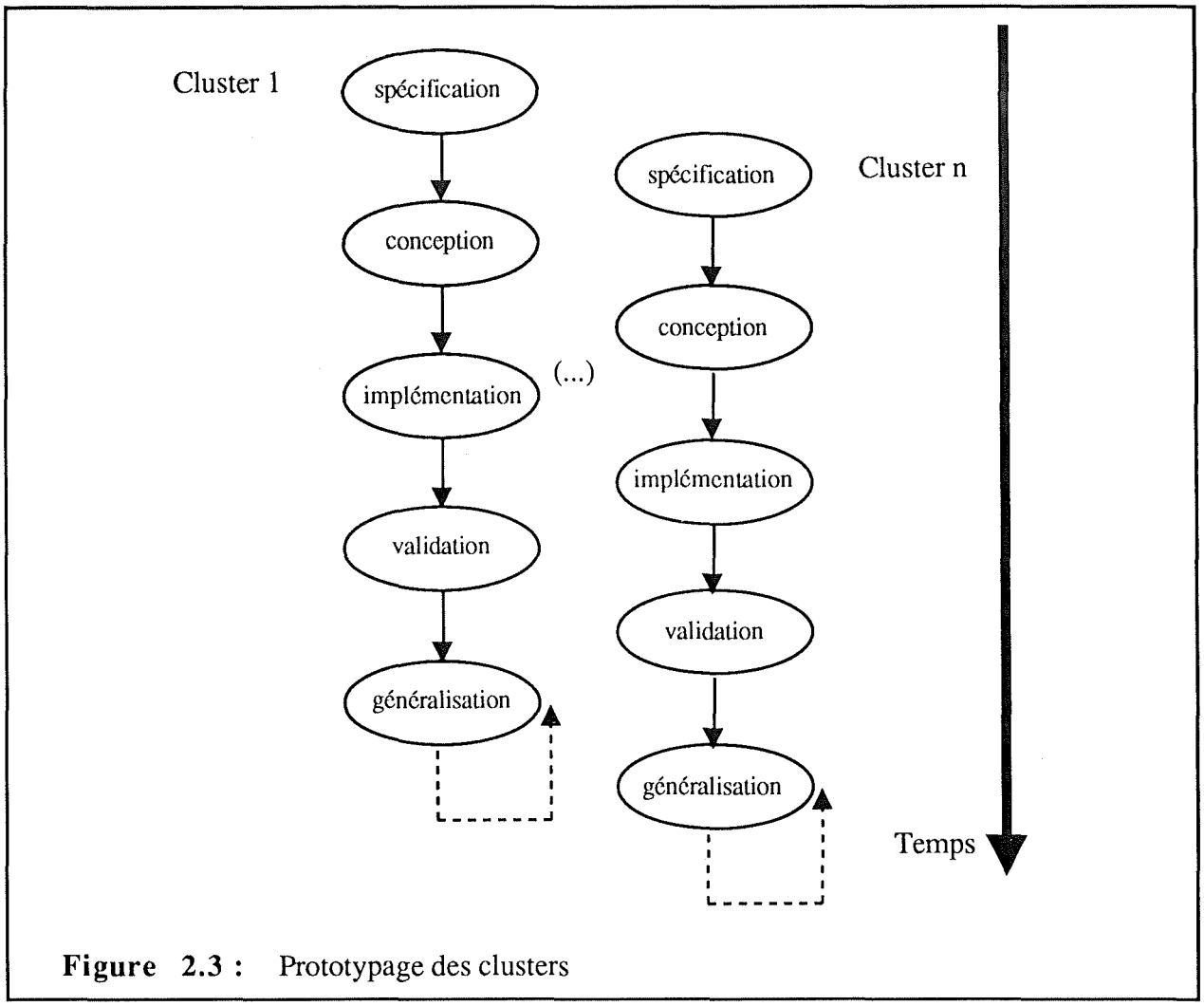

Figure 2.3 : Prototypage des clusters

Cette technique permet notamment de commencer à programmer un sous-système alors que la phase d'analyse n'est pas encore terminée. Par exemple, ceci est très utile pour prendre en compte l'évolution des besoins au fur et à mesure que la compréhension du système par l'utilisateur et le concepteur s'améliore.

Le prototypage peut être vu à deux niveaux : celui du système lui-même et celui des classes d'objets car celles-ci constituent des modules autonomes (voir figure 2.4.). Au niveau des classes d'objets, la démarche de développement suivie est identique à celle du système et le mécanisme de prototypage peut résulter en plusieurs révisions successives de la spécification des classes d'objets.

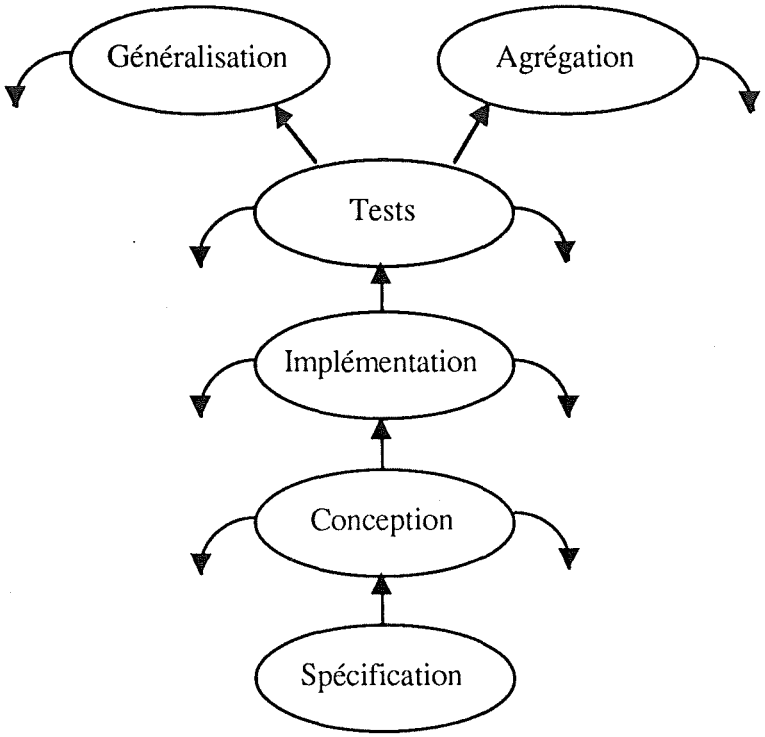

Figure 2.4 : Prototypage au niveau des classes d'objets

Le prototypage implique que les spécifications de l'ensemble du système ne soient pas gelées dès le début de la conception car les modifications qu'il provoque ont un impact local, elles ne remettent donc pas en cause la totalité de l'architecture. Dès lors, les spécifications évoluent dynamiquement au cours de tout le cycle de développement.

2.3. Propriétés d'une approche orientée objets

Les propriétés que présente une application développée grâce à une approche entièrement orientée objets sont les suivantes [MGKO-90] :

- une grande **modularité**,
- l'information contenue dans les objets est invisible à l'utilisateur qui ne peut y accéder que via les méthodes de l'interface de l'objet (principe d'**information hiding**),
- un faible **couplage** entre les différents modules,
- une grande **cohésion** à l'intérieur d'un module,
- une grande **extensibilté** et, finalement,

- une **abstraction** qui peut se présenter à deux niveaux : soit au niveau des spécifications (**abstraction par spécification**), soit au niveau du type des données (**abstraction par paramétrisation**).

2.3.1. La modularité

Un système conçu entièrement selon une approche orientée objets est naturellement décomposé en modules. Dans le paradigme d'une approche orientée objets, les différentes classes d'objets forment les modules de l'application. Ce concept de modularité est non seulement supporté par le processus de conception du système, mais également par la phase d'implémentation à travers la définition des classes d'objets. Le fait que chaque classe d'objets soit définie comme étant un module permet une représentation très fine de la composition du système.

Les modules peuvent également être construits de diverses manières : les **clusters**, **sous-systèmes** et **frameworks**.

Un **cluster**, comme précédemment défini, est constitué par la réunion de classes d'objets qui sont conceptuellement reliées, il résulte donc d'un regroupement naturel [MEYE-89].

Les **sous-systèmes** et les **frameworks** comportent des classes d'objets qui, lorsqu'elles sont regroupées, forment des abstractions. Par exemple, le concept de liste chaînée peut être représenté par trois classes d'objets distinctes : une première classe d'objets représentant la liste elle-même, une deuxième classe d'objets décrivant les liens entre les objets de la liste et une troisième classe représentant les itérations possibles sur la liste. Individuellement, ces trois classes d'objets ne sont pas très utiles mais leur réunion procure une abstraction couramment utilisée.

2.3.2. Le principe d'information hiding

Lors de la construction d'une classe d'objets, il y a séparation entre l'interface de la classe d'objets qui est le seul moyen pour l'utilisateur de manipuler l'information contenue dans l'objet et l'implémentation de la classe d'objets (voir figure 2.5). Dès lors, l'utilisateur n'a aucune idée de la façon dont les données sont représentées à l'intérieur de la classe d'objets et ne se soucie pas de la manière dont les méthodes de l'interface de cette classe d'objets ont été implémentées. Cette séparation permet une maintenance aisée puisque seule l'en-tête de la méthode doit rester inchangé pour que la classe d'objets puisse toujours être utilisée telle qu'elle l'était avant de subir une quelconque modification de son implémentation.

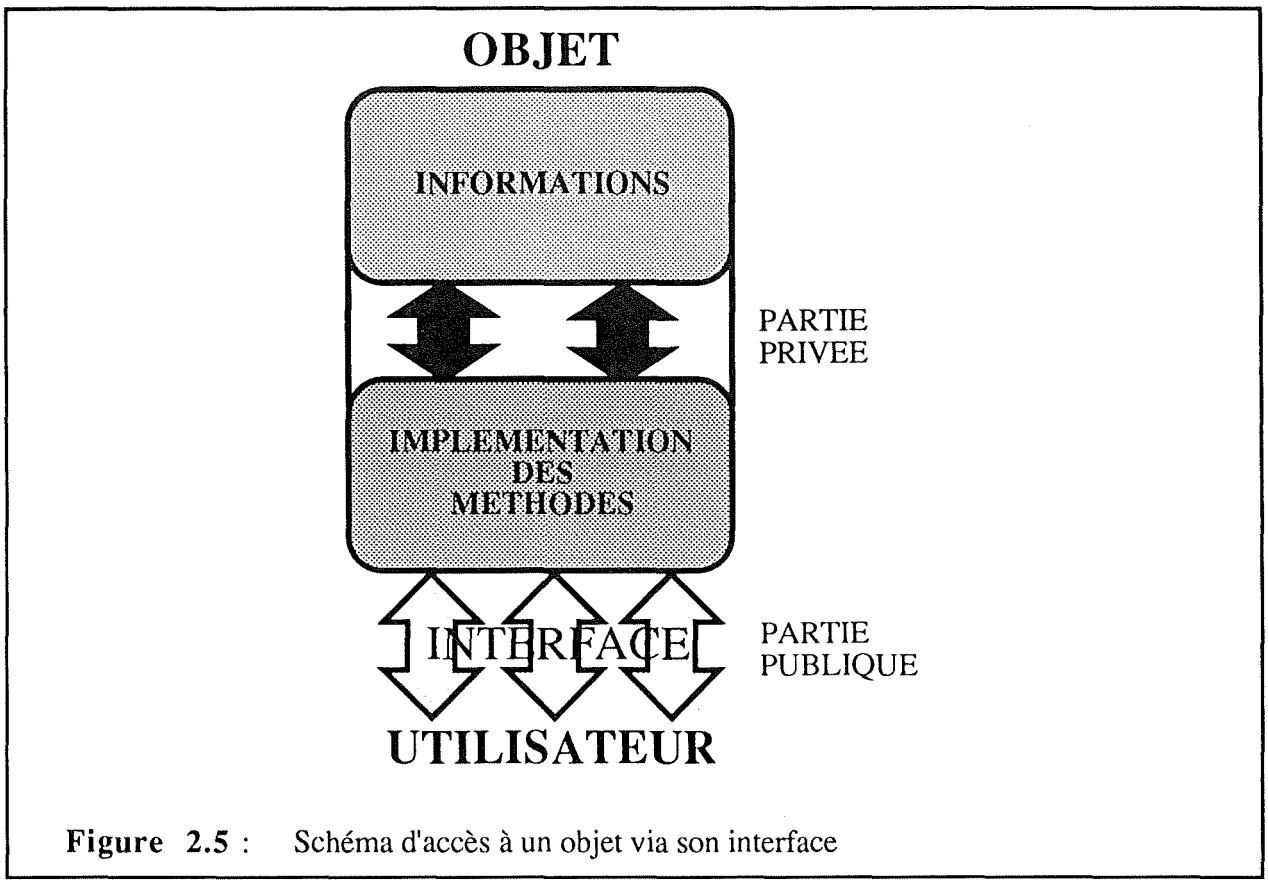

Figure 2.5 : Schéma d'accès à un objet via son interface

2.3.3. Le couplage

Les classes d'objets ont été décrites comme des collections de données caractérisées par un ensemble d'opérations permises sur ces données. De plus, les opérations fournies par l'interface d'une classe d'objets ne sont censées consulter ou modifier que les données internes à la classe d'objets à laquelle elles appartiennent (principe élémentaire de conception), ce qui mène à un très faible couplage entre les classes d'objets.

Les interactions entre les classes d'objets peuvent cependant exister mais l'intérêt est que même si elles existent, les modules communiquent en échangeant peu d'information et le couplage reste faible. Quant à ces interactions, elles ont principalement deux origines.

Premièrement, un certain couplage peut apparaître si une instance d'une classe d'objets A est déclarée comme type de données d'une classe d'objets B. Dès lors, les méthodes de la classe d'objets B peuvent invoquer les méthodes de la classe d'objets A.

Deuxièmement, un couplage peut naître lorsque les méthodes de l'interface d'une classe d'objets possèdent comme paramètre une instance d'une autre classe d'objets. Le couplage provient

alors du fait que ce paramètre apporte soit de l'information à la méthode invoquée, soit que cette instance sera modifiée par l'appel d'une des méthodes de son interface.

2.3.4. La cohésion

Une classe d'objets possède de par son essence même, une forte cohésion. En effet, les méthodes rattachées à l'interface de l'objet peuvent seulement consulter ou modifier les données définies à l'intérieur même de la classe d'objets.

Le mécanisme d'héritage peut être vu comme un facteur qui affaiblit la cohésion d'un module, les données et les méthodes héritées d'une classe d'objets formant un groupe séparé des données et des méthodes ajoutées à la nouvelle classe d'objets. Cependant, le test ultime de cohésion doit montrer que ces différentes pièces, une fois assemblées, représentent bien le concept que l'on a voulu modéliser.

2.3.5. L'abstraction

Il existe deux méthodes d'abstraction distinctes : l'abstraction par spécification et l'abstraction par paramétrisation. Les langages orientés objets les supportent toutes deux à différents degrés.

L'**abstraction par spécification** concerne l'abstraction de la spécification d'une entité par rapport à son implémentation. Ce type d'abstraction est pratiquement supporté par tous les langages orientés objets. L'interface d'une classe d'objets constitue la spécification de cette classe d'objets, elle fournit les opérations légitimes sur les données contenues dans les instances d'une classe d'objets. Dans la plupart des langages orientés objets, l'implémentation de ces méthodes et la représentation des données de la classe d'objets sont inaccessibles, voire invisibles, des utilisateurs de la classe d'objets.

L'**abstraction par paramétrisation** est l'abstraction du type de données à manipuler par rapport aux spécifications qui définissent les manipulations relatives à ces données. Ce type d'abstraction est supporté par la plupart des langages orientés objets au niveau des méthodes de la classe d'objets mais peu le supportent au niveau des classes d'objets.

2.3.6. L'extensibilité

Les applications qui sont développées grâce à une approche orientée objets donnent des architectures facilement extensibles. En effet, le mécanisme d'héritage favorise l'extension de deux manières distinctes.

Tout d'abord, la relation d'héritage facilite la réutilisation de classes d'objets existantes en permettant la définition de nouvelles classes d'objets. Il faut cependant remarquer que plus la structure d'héritage devient profonde, plus la quantité de spécifications et de méthodes déjà implémentées augmente pour une nouvelle classe héritant des propriétés d'une classe d'objets terminale de cette hiérarchie : les efforts restant alors à fournir sont beaucoup moins importants que si l'implémentation de la classe d'objets devait se faire entièrement.

Ensuite, le polymorphisme introduit dans les systèmes orientés objets (voir section 1.4 "Le polymorphisme") permet aussi des extensions aisées. Supposons l'existence d'une hiérarchie d'héritage simple. Supposons, également, la procédure X qui nécessite comme paramètre une instance de la classe d'objets A. Le comportement polymorphique des classes d'objets permet l'utilisation, comme paramètre de cette procédure X, des instances des classes d'objets B, C, D, E ou F. Si la hiérarchie de classes d'objets se trouve étendue comme présentée à la figure 2.6, c'est-à-dire en ajoutant une classe d'objets G héritant des propriétés de la classe d'objets E, les instances de cette classe d'objets peuvent évidemment être utilisées comme paramètre de la procédure X. Donc, une nouvelle classe d'objets a été ajoutée et aucune modification de la procédure X n'a dû être effectuée.

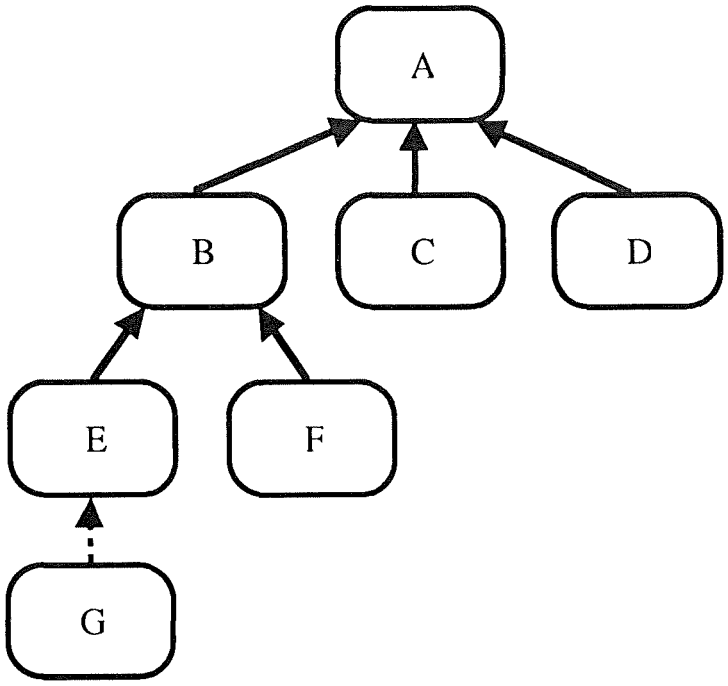

Figure 2.6 : Hiérarchie de classes d'objets illustrant l'extension possible grâce à la relation d'héritage

En conclusion, il apparaît qu'une approche de développement de systèmes informatiques orientée objets si elle est correctement menée, peut procurer naturellement de multiples avantages impliquant des propriétés souvent recherchées lors du développement de logiciels.

3. Méthodologie utilisée dans le cadre de notre travail

Après la présentation des différentes méthodes utilisables lors du développement de systèmes informatiques et l'énoncé des propriétés de l'approche complètement orientée objets, nous allons maintenant expliquer la méthodologie que nous avons utilisée dans le cadre du développement de la boîte à outils permettant la gestion d'un environnement de bureau.

3.1. Adaptation de la méthode orientée objets

Pour construire la boîte à outils de gestion d'un environnement de bureau, nous avons choisi d'appliquer la méthodologie générale de développement orientée objets en sept étapes. En termes de cycle de développement de système, les trois premières étapes relèvent de l'analyse tandis que les quatre étapes suivantes sont relatives à la phase de conception mais aussi à la phase d'implémentation si l'on tient compte des considérations de prototypage abordées à la section précédente.

La technique de prototypage sera appliquée au niveau du système lui-même pour la phase d'analyse et au niveau des clusters pour les phases de conception et d'implémentation et ce, dans le but d'en tirer au plus vite des enseignements.

Comme nous ne voulons pas attendre la fin de la phase d'implémentation du premier sous-système pour nous faire une idée de la valeur la hiérarchie de classes d'objets du système, une étape de validation théorique non existante dans la méthodologie proposée sera introduite.

Pour ce faire, après une première analyse, les étapes 5 et 6 sont appliquées aux classes obtenues de manière à produire une hiérarchie de classes d'objets. La validation consiste ensuite à analyser l'impact de l'introduction de nouveaux objets non prévus dans le système initial sur la hiérarchie. De là, les généralisations et agrégations pertinentes pourront être déduites et une nouvelle analyse pourra être entreprise.

Quand la hiérarchie sera jugée satisfaisante, l'étape 4 de conception détaillée sera enfin entreprise et l'implémentation du premier cluster fournira un premier prototype. Les problèmes survenus lors de cette implémentation conduiront à la révision de la conception qui sera suivie de l'implémentation d'un nouveau prototype auquel nous nous limiterons, faute de temps.

Pour montrer le fonctionnement de la boîte à outils de gestion d'environnement de bureau, une petite application l'employant sera conçue mais cette dernière ne fera pas à proprement parler, partie

de la conception de la boîte à outils. Elle a été développée afin de montrer au lecteur le type d'utilisation qui pourrait être fait de cette boîte à outils.

3.2. Cycle de vie du système d'environnement de bureau

3.2.1. Analyse conceptuelle

L'analyse conceptuelle reprend les trois premières étapes de la méthodologie présentée.

Dans un premier temps, nous analyserons le problème à résoudre en termes d'objets ce qui donnera naissance au modèle d'environnement de bureau classique. Ce modèle initial permettra de poser le problème à résoudre en termes plus précis.

Ensuite, les interactions entre les objets seront étudiées et un premier schéma Entité/Association en résultera. De manière à pouvoir représenter les classes d'objets et leurs interactions correctement, la modélisation Entité/Association [BOPI-83] sera étendue pour prendre en compte les structures d'héritage (relation is-a). De même, les services associés aux classes d'objets fourniront une première description des interfaces de ces classes d'objets. Le langage de spécification utilisé permettra de décrire les mécanismes d'héritage de manière formelle.

De là, la première validation, c'est-à-dire la première tentative d'introduction de nouveaux objets dans le modèle sera réalisée.

Suite à cette validation, le modèle sera généralisé et donnera naissance à un modèle d'environnement de bureau étendu dont les classes plus générales seront plus facilement réutilisables. Un nouveau schéma Entité/Association pourra alors être produit et la description des interfaces des classes d'objets sera complétée par celles des nouvelles classes d'objets introduites lors de la généralisation.

La validation sera à nouveau appliquée au schéma final et ainsi de suite, jusqu'à ce que l'introduction de nouveaux objets dans le modèle se fasse sans difficultés (voir figure 2.7).

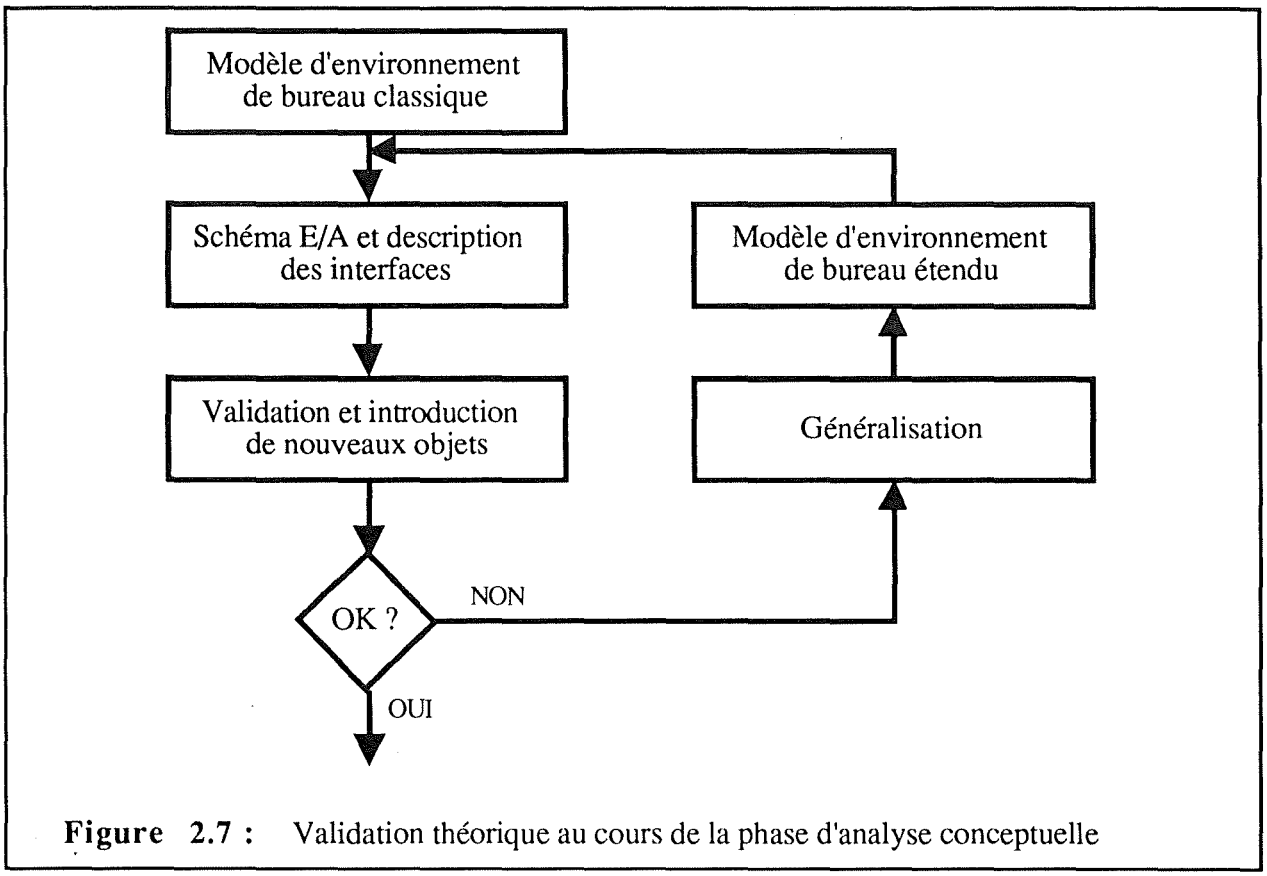

Figure 2.7 : Validation théorique au cours de la phase d'analyse conceptuelle

L'analyse conceptuelle d'un environnement de bureau fera l'objet du chapitre 3 de ce mémoire. Nous y aborderons la description du système finalement obtenu suite aux différentes validations, les modèles intermédiaires n'y seront pas présentés.

3.2.2. Conception et implémentation

Une fois l'analyse conceptuelle terminée, la phase de conception prendra place.

Cette phase sera subdivisée en deux sous-étapes : les étapes de conception logique et de conception physique. Rappelons que la conception logique consiste à produire une solution exécutable sur une machine abstraite et la conception physique, une solution dérivée de la première et exécutable sur une machine concrète [HAIN-86].

C'est lors de ces étapes que prendront place les transformations décrites dans l'introduction.

La conception logique portera sur le choix d'une architecture pour le système d'environnement de bureau et sur les transformations à appliquer au schéma E/A pour obtenir un schéma logique (en l'occurrence le schéma des accès nécessaires) [HAIN-86]. Comme la phase de conception logique a

une influence dynamique sur les spécifications, la spécification détaillée des classes du modèle ne sera réalisée qu'après cette étape.

La conception physique consistera à traduire l'architecture logique en structures conformes à la configuration cible, à savoir le système de gestion de base de données N.D.B.S. et environnement de programmation TURBO-PASCAL version 5.5. Les transformations concernent le schéma des classes d'objets représentant les modules du système et le schéma de la base de données.

Un premier sous-système pourra alors être implémenté. Suite aux tests réalisés, l'architecture sera corrigée et une nouvelle implémentation pourront prendre place (voir figure 2.8).

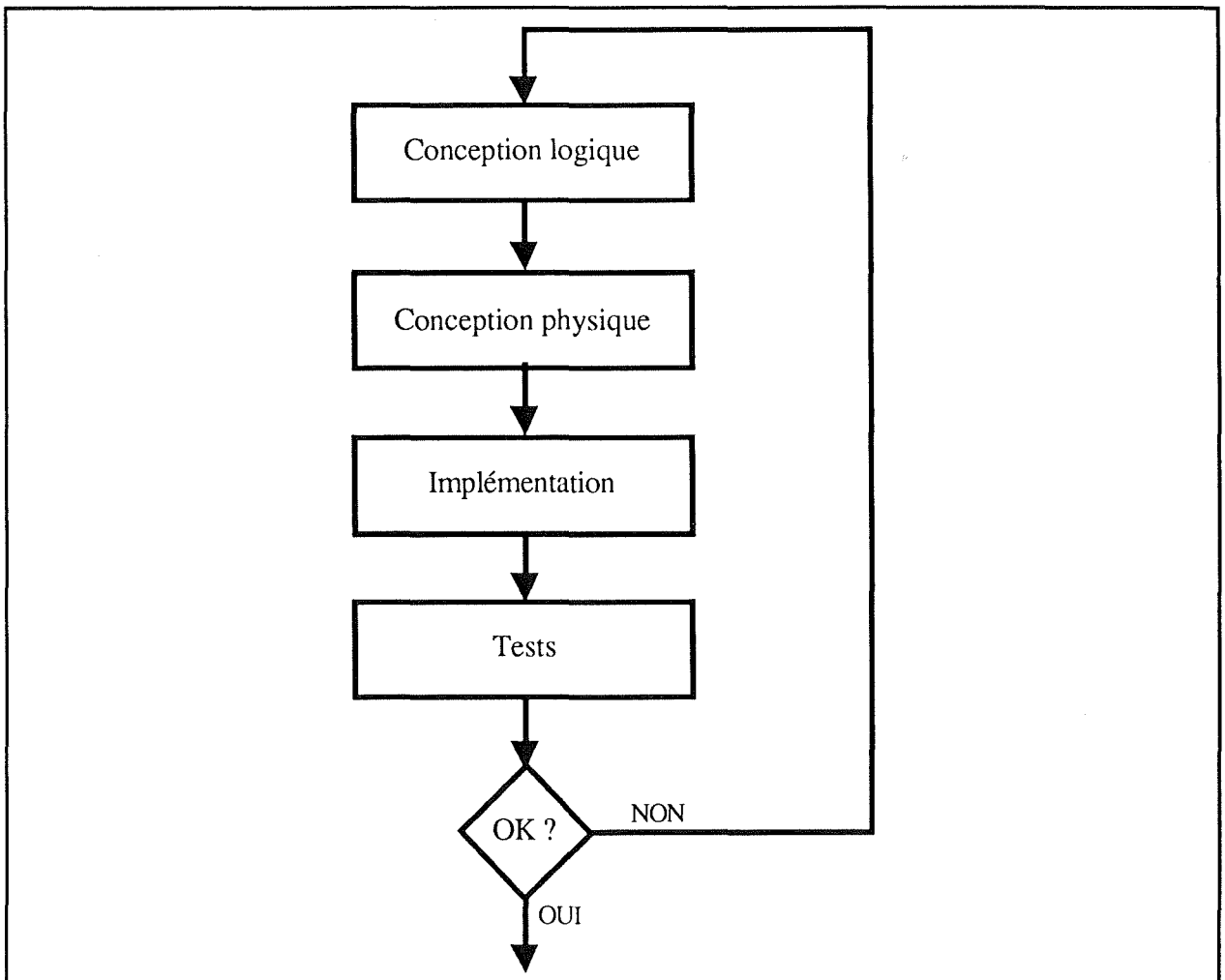

Figure 2.8 : Prototypage au cours des phases de conception et d'implémentation

La description de l'application de cette démarche à l'environnement de bureau étendu fera l'objet des chapitres 4, 5 et 6. Comme pour l'étape d'analyse, c'est la solution finale qui sera présentée. Quand des changements importants seront appliqués au modèle obtenu après la phase

d'analyse, par exemple, suite au prototypage d'un sous-système déterminé, leur motivation sera expliquée en détails.

3.2.3. Fonctionnement

Une petite application permettant de montrer l'utilité des services offerts par la boîte à outils formée des classes d'objets dégagées de l'analyse d'un environnement de bureau sera implémentée. Elle ne se veut pas exhaustive et constitue, dans le cadre de ce travail, une partie minime destinée à montrer la facilité d'emploi de la boîte à outils de gestion d'environnement de bureau.

Chapitre 3 :
Analyse conceptuelle d'un environnement de bureau

Introduction

1. Présentation de l'analyse conceptuelle

Maintenant que les concepts de base sont posés, l'analyse conceptuelle d'un environnement de bureau peut être entreprise. Elle sera subdivisée en plusieurs étapes qui restent cependant fortement couplées. Ces étapes ont été décrites brièvement à la section 2.3 "Méthodologie utilisée dans le cadre de notre travail".

Lors d'une première étape, nous nous attacherons à décrire un **environnement de bureau classique**. Par "classique", nous entendrons un bureau tel qu'il apparaît dans le réel perçu de ses utilisateurs, c'est-à-dire avec ses fonctionnalités et ses structures physiques. Les concepts repris dans le modèle seront simples, ils serviront de support pour construire la hiérarchie de classes d'objets.

Pour obtenir **l'environnement de bureau étendu**, chacun des objets de l'environnement classique sera étudié sur base de sa fonction principale ("sa raison d'être") en faisant abstraction d'éléments de différenciation ayant pour seule origine leur structure physique. Certaines contraintes seront levées dans le but de faciliter les extensions futures du modèle. Cette technique permettra de regrouper les objets de l'environnement de bureau classique en classes élaborées sur base de propriétés logiques communes à plusieurs objets. Par l'application d'une démarche "bottom-up", de nouvelles classes d'objets, inexistantes dans le modèle initial, apparaîtront.

La troisième partie sera consacrée à la construction du **schéma E/A généralisé d'un environnement de bureau étendu**. Le schéma conceptuel des données sera obtenu par dérivation de la hiérarchie d'objets du bureau. Le formalisme E/A étendu, intégrant les concepts d'héritage permettra de traduire la hiérarchie de classes d'objets directement dans le formalisme E/A, par un ou plusieurs types d'entités. Ainsi, la correspondance entre les deux descriptions, schéma des classes et schéma des données s'établira facilement.

Finalement, nous décrirons les **interfaces associées aux classes d'objets**. L'ensemble de leurs méthodes fournira les services offerts par les instances des classes d'objets.

Pour terminer le chapitre, nous passerons à la phase de validation théorique. Autrement dit, nous décrirons comment il est possible de **réutiliser la hiérarchie** existante pour étendre le modèle en définissant de nouvelles classes.

Rappelons que cette démarche de construction d'un modèle par étapes a pour but final de produire un modèle aussi général que possible de manière à ce qu'elle puisse supporter des extensions futures (définitions de nouvelles classes dérivées supportées par la hiérarchie existante).

Mais avant toute chose, il nous semble intéressant de décrire l'objet même qui est le fondement de cette étude : le **bureau**.

2. Qu'est ce qu'un bureau ?

"Un bureau typique est une cellule de l'organisation ayant une activité commune s'apparentant à un service. Sa dimension est réduite dans l'espace (quelques pièces tout au plus) et en nombre de personnes (une douzaine environ). Un même bureau peut comprendre diverses catégories de personnel : cadres, techniciens, agents, employés, secrétaires, dactylos. Leurs postes de travail sont en général voisins. C'est son activité qui fait l'unité du bureau, même si les tâches accomplies par les divers acteurs sont de natures différentes. Dans notre esprit, un bureau dispose d'une relative autonomie pour l'accomplissement de ses activités, qu'elles soient de production, de transformation, de création, ou autres" [BLAS-82].

Comme le montre la description qui précède, un bureau est constitué d'un ensemble de personnes aux fonctions très diverses. Aussi disparates qu'elles semblent être, toutes ses activités dont les bureaux sont le siège convergent vers seulement trois fonctions principales : la production, la distribution et l'exploitation de l'information [MART-82]. Elles constituent le point unificateur, commun à toutes les tâches prenant place dans cet environnement.

Plutôt que de nous attarder sur les fonctions du bureau, nous allons dès à présent tenter de répondre à la question fondamentale qui nous préoccupe à savoir : quels sont les objets qui permettent de remplir ces fonctions ?

La production de l'information concerne l'information sous toutes ses formes : lettres, contrats, comptes rendus de réunion, messages de toutes natures, notes de services, projets,... Il existe donc un nombre impressionnant d'objets variant d'une organisation à l'autre, qu'il nous sera nécessaire de classifier afin de les modéliser. En effet, une telle diversité ne pourrait que nuire au caractère générique du modèle que nous voulons justement extensible.

Les activités de distribution de l'information n'entrent pas dans le cadre strict de ce travail, le but restant de trouver un modèle destiné à automatiser le rangement d'informations.

Par contre, l'exploitation de l'information nous concerne à plus d'un titre : "une exploitation rationnelle de l'information exige de retrouver le ou les documents pertinents dans les meilleurs délais et à moindre coût. Peu d'entreprises sont capables d'afficher un bilan satisfaisant dans ce domaine" [MART-82].

Le problème reste donc le classement de l'information. Celui-ci conduit dans le cadre de la démarche de modélisation orientée objets, à l'identification de deux types d'objets.

Les fichiers comptables, clients, fournisseurs ; les dossiers projets, contacts ; les catalogues de produits, répertoires téléphoniques, d'adresses,... c'est-à-dire tout support créé dans le but de regrouper des informations dont le contenu présente des analogies sont des exemples du premier type d'objets. Nous les appelerons **collections**.

Ensuite, le deuxième type d'objets regroupera l'ensemble des objets d'ameublement : armoires, tables, étagères, compartiments,... qui peuvent être vus comme des supports d'archivage et de classement. Il constitueront les **objets de rangement**.

Comme on le voir, les objets du problème sont nombreux. Afin de produire un modèle correct, les objets de la solution devront être de niveau d'abstraction plus élevé et permettre de représenter la plus grande diversité possible d'environnement de production et d'exploitation de l'information. C'est un des buts de l'analyse qui suit.

1. Présentation d'un environnement de bureau classique

Sur base de la description du bureau, nous allons analyser en détails les objets décrits et les regrouper en catégories. Un noyau d'objets de base va ainsi être obtenu, il nous fournit un modèle initial. Par la suite, les concepts définis dans ce noyau seront enrichis pour produire un modèle plus général (environnement de bureau étendu).

La description aborde avant tout les différentes formes prises par l'information. Ensuite, elle porte sur les méthodes logiques d'organisation utilisées pour structurer les informations en ensembles cohérents, les collections. Finalement, les modes de rangement les plus couramment rencontrés et utilisés à des fins de classement et d'archivage sont étudiés. Les objets concernés sont ceux que nous avons nommés objets de rangement.

1.1. L'information

Une **information** est un ensemble structuré de signes ou de symboles porteurs de connaissance pour celui qui les reçoit. Le modèle développé ici reprend les trois représentations de l'information les plus communément rencontrées, à savoir le **message**, le **formulaire** et le **document**. Chacun de ces concepts possède des caractéristiques spécifiques, décrites ci-dessous [DACH-86].

a) Le message

Le message est une information présentée sous forme écrite. Il est généralement bref et son corps n'obéit à aucune structure particulière. Il s'agit, par exemple, d'un aide-mémoire, d'un post-it, d'une remarque, d'un mémo ou d'une note.

b) Le formulaire

Le formulaire est une information écrite, structurée et ayant une forme standardisée. Il est composé de champs libellés auxquels des valeurs peuvent être attribuées. Les champs sont agencés selon un squelette prédéfini. L'information s'y inscrit lorsque l'on complète ces champs. La dénomination formulaire englobe les fiches, les bons de réquisition, les factures...

c) Le document

C'est une information volumineuse, structurée et soignée, présentée sous forme textuelle ou imagée (graphiques, tableaux, ...). Le document est le type d'information le plus répandu dans une organisation. Les contrats, les lettres, les comptes rendus, ... appartiennent à cette catégorie.

Souvent, le document est décomposable en un en-tête, un corps et une clôture, ces parties pouvant, à leur tour, obéir à une découpe plus fine. Les sections, paragraphes, lignes en sont des exemples. Ainsi, un document apparaît comme un ensemble d'informations organisées selon une structure logique hiérarchique.

De manière générale, toute partie d'un document s'appelle un **fragment**. Un fragment est lui-même une information mais il n'a pas d'existence autonome. Les fragments sont agencés entre eux suivant la structure logique propre au document. Les titres, les graphiques ou tableaux, les notes en bas de page, la signature,...sont des fragments.

Cette notion sera étudiée plus en détails par la suite car elle servira de base pour établir une stratégie de stockage des documents dans la base de données.

1.2. L'organisation de l'information

Généralement, les informations sont structurées en agrégats, appelés **collections** d'informations, en vue de faciliter leur gestion ultérieure : rangement, accès ou traitement. Dans les bureaux, différentes formes de regroupement sont utilisées pour faciliter l'accès rapide aux informations importantes, pour archiver les informations anciennes,...

L'organisation des informations en collections est une activité d'origine intellectuelle. Elle est appelée **rangement logique** de l'information car les critères de regroupement utilisés pour structurer les informations en agrégats dépendent de la manière dont une personne voit ces informations et identifie les liens les unissant, deux personnes différentes ne produisant pas nécessairement les mêmes agrégats.

Trois types de collections ont été retenues : le **fichier**, le **dossier** et la **pile**. Ces notions sont introduites dans [VPMS-87]. Dans le modèle élaboré ici, leur description a été étendue afin de favoriser l'extensibilité du schéma final des classes d'objets.

a) Le fichier

Le fichier est une collection d'informations de même type : documents, messages ou formulaires. Les composants du fichier sont identifiés univoquement et rangés selon un certain ordre. Les composants étant de même type, le nombre de critères sur lesquels une relation d'ordre peut être établie est grand et dépend du type des composants. Par exemple, pour un fichier de formulaires "vendeurs", un classement pourrait être réalisé sur des champs libellés nom, région, catégorie.

b) Le dossier

Le dossier est une collection d'informations de type quelconque dont les composants sont connectés entre eux par une relation logique. La constitution de dossiers a pour but de rassembler des informations dispersées, traitant un même sujet. C'est par exemple, un dossier médical ou un dossier contenant les pièces justificatives de dépenses.

La différence de type entre les composants du dossier limite le nombre de critères sur lesquels une relation d'ordre est définissable aux seules caractéristiques communes à tous les composants. Par exemple, une possibilité serait la date de production des pièces justificatives du dossier de dépenses.

c) La pile

Il s'agit d'une collection d'informations de type quelconque, sans lien logique entre ses composants sauf l'ordre chronologique de leur arrivée sur la pile. C'est par exemple, l'ensemble du courrier de la journée. Les piles sont souvent utilisées pour stocker des informations en attendant un traitement.

Le tableau 3.1 fournit un résumé des critères caractérisant les fichiers, les dossiers et les piles. Ces critères permettent d'identifier sans équivoque le type de collection dont il est question.

Collection	Composant	Relation logique	Ordre
Fichier	un seul type	existe	Nombreux critères liés au type
Dossier	de type quelconque	existe	Critères communs à tous les types
Pile	de type quelconque	aucune	aucun

Tableau 3.1 : Critères d'identification des collections

Pour la suite de la construction du modèle, les fichiers, les dossiers et les piles seront regroupés en un concept général appelé collection. Il faut aussi retenir que les collections transportent une connaissance supplémentaire par rapport aux informations : le critère logique utilisé lors du regroupement des informations en collections. Ce critère est introduit dans le but de faciliter le

classement et le traitement ultérieur des informations. Il apporte une vision globale et générique de celles-ci à l'utilisateur de l'environnement.

1.3. Les objets de rangement

Dans un bureau, toutes les informations sont rangées en un endroit précis. Cet endroit doit être connu de manière à pouvoir retrouver l'information qu'il faut traiter.

Souvent, les informations sont organisées en collections avant d'être classées mais elles peuvent aussi être rangées de manière autonome.

En général, le **rangement physique** de l'information est lié à son rangement logique, les informations appartenant à une même collection recevant des localisations spatiales identiques pour faciliter leur accessibilité. Ainsi, la constitution de collections permet de décrire plus facilement le rangement physique en retenant la localisation de la collection plutôt que la localisation de chacun de ses composants.

Pour compléter la définition de l'environnement de bureau classique, il reste donc à définir les objets de rangement qu'il comprend ainsi que les relations possibles entre ces objets, les informations et les collections d'informations : c'est-à-dire qui contient quoi ? Ces relations constituent les règles de base auxquelles les éléments du bureau doivent obéir lorsqu'ils sont en relation. Certains des objets retenus sont inspirés d'une étude antérieure [ROBE-86], d'autres ont été ajoutés pour élargir la définition de l'environnement de rangement du bureau.

a) L'armoire et l'étagère

L'armoire est un meuble plus ou moins volumineux, composée de tiroirs qui sont destinés à recevoir des informations, des collections ou d'autres objets de rangement. On peut dès à présent remarquer que la structure de classement possède plusieurs niveaux : certains objets peuvent être inclus dans d'autres objets de l'environnement de bureau.

Tout comme l'armoire, l'étagère est décomposable en différents compartiments représentables sous forme de travées et de colonnes. Chaque compartiment peut servir de support à des collections et à des informations.

Lors de notre étude, l'étagère sera assimilée à l'armoire et les tiroirs seront vus comme des compartiments. En effet, il a été précisé dans l'introduction que les fonctions principales des objets seraient étudiées en faisant abstraction de leur structure physique. Il est évident que la fonction

remplie est identique. Dans une armoire, le rangement de l'information ne peut se faire que par l'intermédiaire du compartiment. Cette contrainte est aussi en accord avec la définition de l'étagère.

b) Le classeur

Le classeur est un objet de rangement qui peut contenir des informations, des collections et d'autres classeurs. Cette dernière inclusion permet de modéliser la décomposabilité d'un classeur en parties distinctes. Il peut lui-même être rangé dans un objet de rangement qui serait par exemple un compartiment ou une boîte.

Ces structures d'inclusion sont importantes pour la suite de l'analyse car elles fournissent des relations entre les objets et offrent une possibilité de hiérarchisation des classes d'objets.

c) La boîte

La boîte est un objet de rangement qui est souvent utilisé pour l'archivage. Elle peut contenir les mêmes objets que le classeur, le classeur y compris. Des boîtes peuvent éventuellement contenir d'autres boîtes.

d) Le compartiment et le tiroir

Le compartiment est un objet de rangement qui peut contenir des boîtes, des classeurs, des informations ou des collections. Il sert de structure intermédiaire pour le rangement dans les armoires (et donc les étagères).

e) La poubelle

La poubelle est un objet de rangement particulier : on y range les objets à détruire. Elle peut accueillir tous les objets non volumineux comme les classeurs, les boîtes, les informations et les collections d'informations.

f) La table de travail

La table de travail est l'endroit où l'information est traitée. Seul le plan de travail sera considéré, les éventuels montants composés de tiroirs peuvent être modélisés à l'aide des concepts d'armoire et de compartiment.

La table de travail peut accueillir des classeurs, des boîtes, des informations et des collections. Le plus souvent, il s'agit d'un accueil temporaire dont le but est de pouvoir traiter l'information proprement dite. Dans l'environnement de bureau classique, toutes les informations et les collections consultées à un moment donné se trouvent sur la table de travail.

1.4. Les relations d'inclusion entre les objets de bureau

Les objets de rangement, les informations et les collections constituent l'ensemble des **objets de bureau**.

La figure 3.2 présente les **relations d'inclusion** existant entre les différents objets d'un environnement de bureau classique. Les flèches sont dirigées du contenant vers le contenu. Par souci de clarté, les relations sont décrites par une structure simplifiée ne possédant que deux niveaux. Pour examiner la hiérarchie complète pour un objet donné, il suffit de se reporter successivement aux différents objets que peut lui-même contenir cet objet.

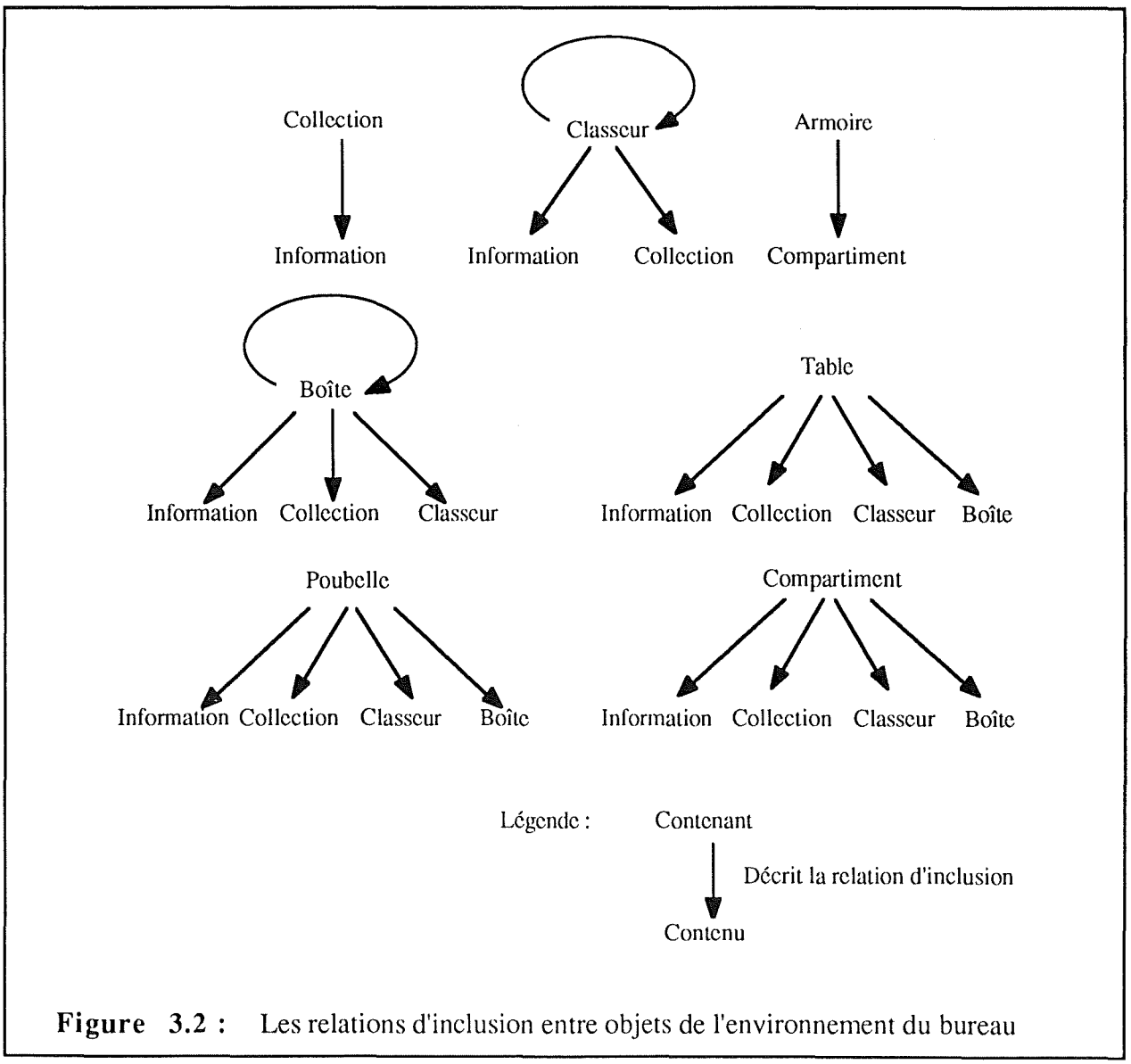

Figure 3.2 : Les relations d'inclusion entre objets de l'environnement du bureau

Nous attirons l'attention sur les objets de type boîte et classeur qui peuvent contenir des objets de leur type. Cette caractéristique résulte en l'existence de contraintes difficiles à gérer telles que la non contenance d'un objet dans lui-même.

Les relations d'inclusion illustrent l'existence de similitudes entre les objets. Les informations se situent toujours au plus bas niveau de la hiérarchie d'inclusion tandis que l'armoire, la table et la poubelle sont toujours au sommet de cette hiérarchie. Pour les boîtes et les classeurs, une structure d'imbrication apparaît. Tout comme le compartiment, ces derniers servent toujours de structures intermédiaires de rangement .

La deuxième étape de l'analyse conceptuelle consiste à étudier plus en détails ces similitudes et les mettre à profit pour généraliser la définition des objets introduits ci-dessus. Le but recherché est d'utiliser ces caractéristiques pour produire une hiérarchie suffisamment générale pour pouvoir supporter des extensions futures. Les généralisations décrites ne sont pas apparues spontanément lors de l'étude de l'environnement de bureau classique, elles résultent, pour la plupart d'entre elles, de plusieurs cycles complets d'analyse.

2. Présentation d'un environnement de bureau étendu

Dans cette partie, une analyse des objets précédemment décrits est réalisée afin de dégager les similitudes qu'ils présentent. Par cette méthode, des classes d'objets sont obtenues, notamment en éliminant les restrictions sur les fonctionnalités des objets qui sont uniquement dues à leur structure physique. En effet, celles-ci n'ont pas lieu d'être ici car elles réduiraient considérablement l'extensibilité du modèle final.

Les classes générales supportant la hiérarchie d'objets sont définies par abstraction des objets du noyau de base. Elles forment de nouvelles classes d'objets non apparentes dans le modèle initial. Les objets de l'environnement de bureau classique sont représentés par des classes d'objets obtenues par spécialisation des classes générales. Elles héritent des caractéristiques de leurs classes de base auxquelles viennent s'ajouter leurs caractéristiques spécifiques (enrichissement de classe).

2.1. Les critères de partitionnement et les classes résultantes

Sur base de la description que nous avons faite d'un environnement de bureau classique, un certain nombre de propriétés communes à tous les objets de l'environnement de bureau peut être défini. C'est par exemple le cas du nom ou de la date de création de ces objets. De même, des actions telles que la création, la suppression, la modification des propriétés précitées sont valables pour tous les objets. Il paraît donc intéressant de regrouper tous les objets définis en une classe générique unique appelée Objet de Bureau (OB). Elle reprend les propriétés communes à tous les objets et les services qu'elle rend définissent son interface.

La classe Objet de Bureau possède plusieurs classes dérivées qui sont mises en évidence à l'aide de propriétés générales déduites de la description précédente :

- la **nature physique** de l'objet : est-il porteur ou non d'information ? Ce premier critère conduit à distinguer la classe Objet Informationnel (O-Info) et la classe Objet de Rangement (O-Rgt).
- son **caractère rangeable** : un objet peut-il être rangé ou non ? Dans le modèle de base, la table, l'armoire et la poubelle sont des objets terminaux quant aux relations d'inclusion définies (ce sont donc des objets non rangeables) tandis que tous les autres objets sont non terminaux (ils font partie des objets rangeables). La partition résultante de ce deuxième critère comprend les classes dérivées Objet Rangeable (O-Rgble) et Objet Non Rangeable (O-Non-Rgble).

Les classes ainsi obtenues sont illustrées à la figure 3.3.

Ces deux propriétés fournissent des critères de partitionnement, c'est-à-dire que les sous-classes engendrées sont couvrantes et disjointes. En effet, il n'existe aucun objet qui soit à la fois rangeable et non rangeable ou encore informationnel et de rangement (caractère disjoint) et tous les objets du modèle possèdent au moins une de ces propriétés (caractère couvrant).

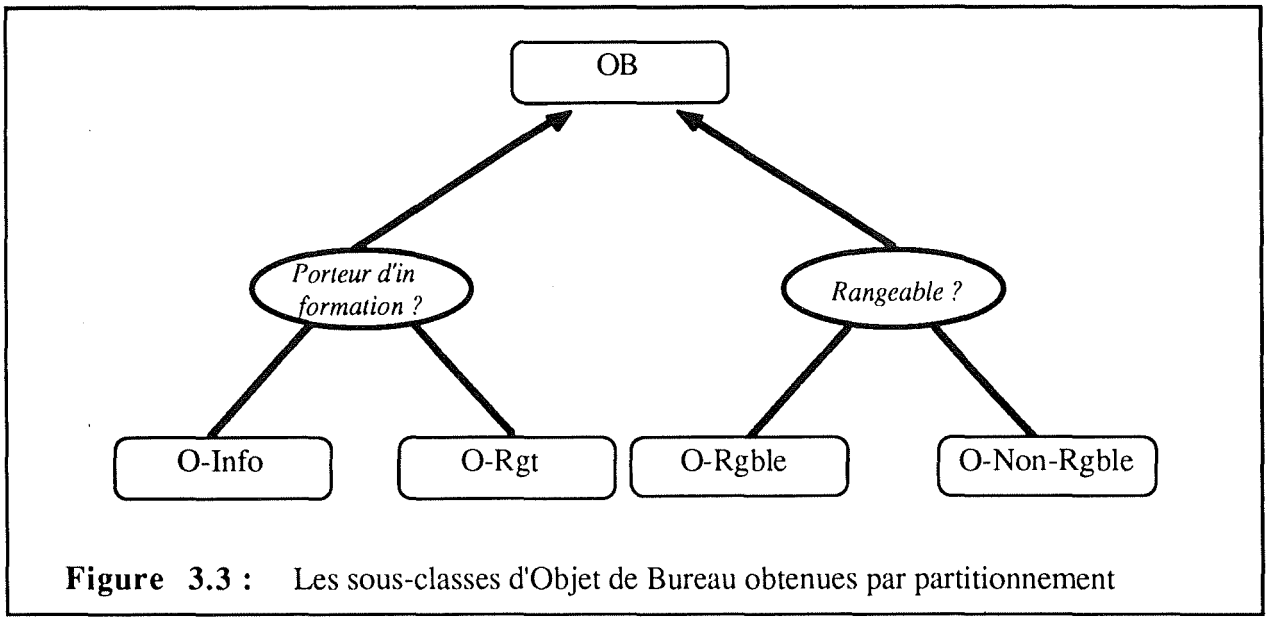

Figure 3.3 : Les sous-classes d'Objet de Bureau obtenues par partitionnement

Les critères utilisés pour le partitionnement ne sont pas uniques, d'autres pourraient y être ajoutés. Cependant, nous nous limiterons à ceux-ci car le but recherché est de modéliser un environnement de rangement des informations. Il suffit donc de pouvoir distinguer la nature des objets et de savoir s'ils sont rangeables ou non.

Une fois ces sous-classes définies, il reste à étudier la nature des différents objets décrits précédemment pour les classifier. Pour ce faire, les mécanismes d'héritage simple et multiple, présentés au chapitre précédent, sont disponibles.

2.2. La classe Information

Cette classe comprend les objets de bureau qui sont les supports de l'information, à savoir les messages, les documents et les formulaires.

La classe Information (Info) obéit, bien évidemment, au critère de partitionnement "porteur d'information". De plus, toute information est rangeable. Donc, par le mécanisme d'héritage multiple, elle hérite des propriétés des classes Objet Informationnel et Objet Rangeable.

Les trois formes prises par l'information deviennent des classes dérivées de la classe Information : les classes Formulaire, Message et Document. La classe Information reprend toutes les propriétés et les méthodes communes à tous les formulaires, messages et documents. Par exemple, le concept de copie d'information est défini à ce niveau.

En plus des propriétés et méthodes de la classe Information dont elles héritent naturellement, les trois sous-classes d'Information peuvent posséder des propriétés et des méthodes qui leur sont propres.

La hiérarchie finalement obtenue est présentée à la figure 3.4.

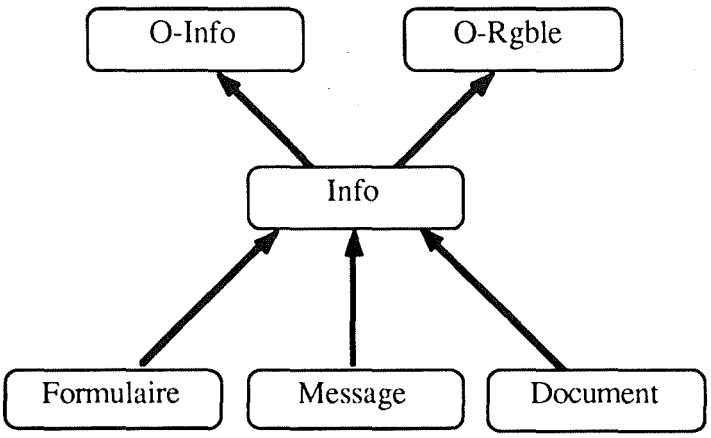

Figure 3.4 : Les super-classes et sous-classes de la classe Information

2.3. La classe Collection

La classe Collection reprend les fichiers, piles et dossiers précédemment définis. Comme pour la classe Information, des méthodes spécifiques à ces objets existent. Cependant, une analyse plus approfondie montre qu'il serait intéressant d'étendre les fonctions remplies par ces objets afin de leur associer des méthodes plus générales, le principal objectif de cette partie étant de généraliser le modèle de base pour définir une hiérarchie d'objets extensible.

Le critère d'organisation d'informations en collections permet d'avoir une vision plus générale d'informations apparemment dispersées : il introduit des liens logiques supplémentaires entre les objets et transporte une information qui lui est propre (la signification du lien logique). Dès lors, la classe Collection sera considérée comme une sous-classe de la classe Objet Informationnel. Elle

dérive aussi de la classe Objet Rangeable puisque toute collection est rangeable. La hiérarchie résultante est illustrée à la figure 3.5.

La classe Collection telle que nous l'avons définie peut encore être enrichie. En effet, rien n'interdit la définition de collections de collections. Ainsi, un dossier ou un fichier peut être composé d'autres dossiers, une pile peut accueillir des dossiers... Même si certaines constructions telle qu'une pile de fichiers peuvent heurter l'esprit car nous y associons notre réel perçu, elles ne présentent en elles aucune difficulté de modélisation. Il paraît donc intéressant de laisser à l'utilisateur la liberté de choisir les associations permises dans un environnement de bureau donné. Il ne faut pas être restrictif lors de la construction du modèle car ce serait perdre de vue les principes fondamentaux d'extensibilité et de généralité qui caractérisent l'approche orientée objet [MEYE-88].

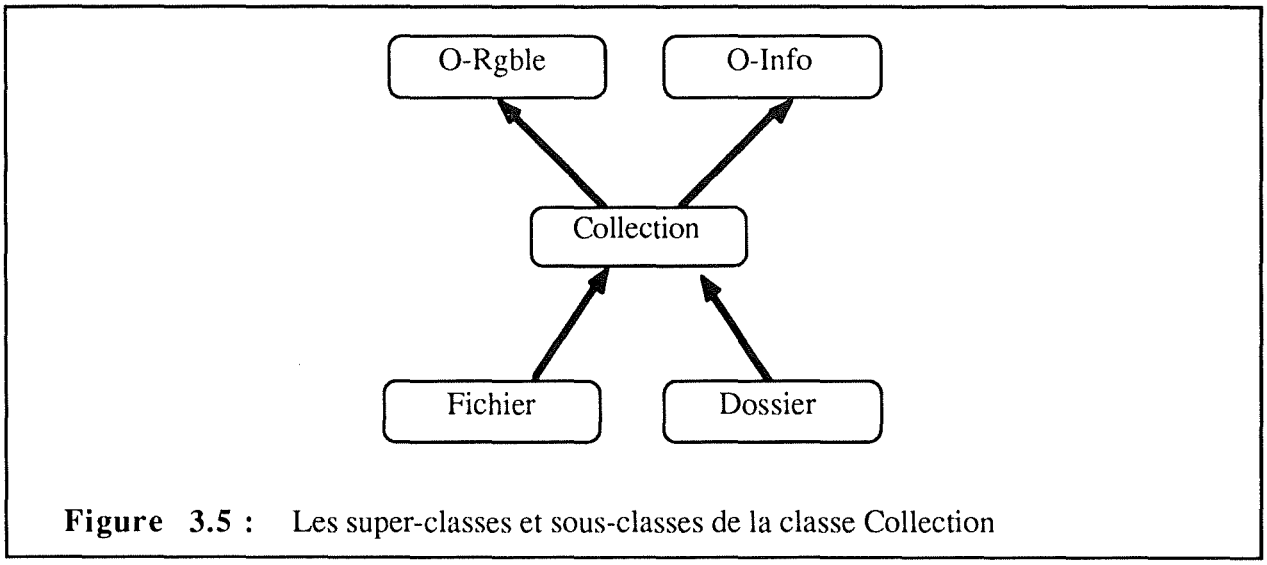

Figure 3.5 : Les super-classes et sous-classes de la classe Collection

Le concept de collection est donc étendu par le biais des types permis pour ses composants (pour rappel, se reporter à la figure 3.1). La notion de type reprise dans le tableau englobe désormais les types fichier, dossier et pile en plus des types document, message et formulaire. Si l'on raisonne en fonction des classes déjà définies, une collection a donc pour composant tout objet informationnel.

De plus, d'après le tableau introduit lors de la présentation d'un environnement de bureau classique, le dossier et la pile n'offrent pas de différences fondamentales si ce n'est la relation logique de classement propre au dossier. Le dossier peut donc être vu comme un cas particulier de la pile (il en est en fait une définition plus précise). Par la suite, ces deux notions seront agrégées en une seule : la classe Dossier. Pour ce faire, les méthodes associées à la pile seront tout d'abord étendues. La notion d'accès à un élément d'une pile est remplacée par l'accès direct à un élément de

celle-ci. Cette extension la différencie d'une pile classique pour laquelle les éléments sont uniquement accessibles dans l'ordre inverse de leur arrivée. Ensuite, l'ajout d'un élément à la pile devient un opérateur non structuré correspondant à l'insertion d'un élément à un endroit quelconque de la pile. Ce changement n'est pas gênant puisque l'ordre chronologique inverse existant dans les piles est lié à l'insertion forcée au sommet de la pile. Ainsi, les méthodes qui seront définies dans l'interface de la classe Dossier seront aussi applicables à la pile.

Toutes les extensions proposées pour généraliser la classe Collection se résument comme suit : toute collection est un objet informationnel rangeable qui est lui-même composé d'objets informationnels (sans aucune restriction sur leur type). Cette structure décrit, par sa récursivité, l'imbrication possible des collections. La hiérarchie introduite dans la première partie est modifiée pour devenir celle illustrée à la figure 3.6.a ou en utilisant les classes définies, celle de la figure 3.6.b.

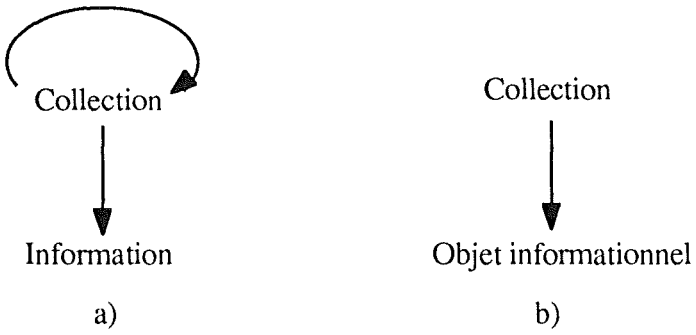

Figure 3.6 : Les imbrications de collections :
a) description par les relations d'inclusion ,
b) description basée sur les classes définies

2.4. La classe Objet de Rangement Structurant

La classe Objet de Rangement Structurant (O-Struct) est obtenue par héritage multiple sur les classes Objet de Rangement et Objet Rangeable (voir figure 3.7). Elle reprend tous les objets de rangement qui sont eux-mêmes rangeables, à savoir la boîte, le classeur et le compartiment (pour rappel, se reporter à la figure 3.2 relatives aux relations d'inclusion). La différence existant entre ces trois objets est d'ordre purement physique, elle n'a pas lieu d'être dans un environnement d'objets bureautiques informatisés. Les fonctionnalités des objets de cette classe sont identiques : ils représentent une structure de rangement intermédiaire permettant le regroupement d'objets informationnels. De plus, les méthodes attachées à ces objets sont identiques : ajouter ou supprimer un composant, lister leur contenu...

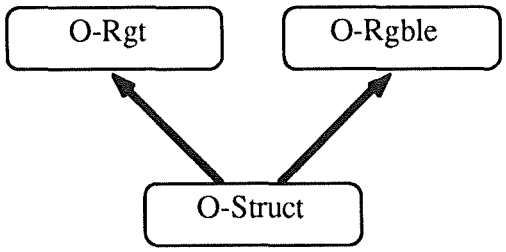

Figure 3.7 : La classe Objet de Rangement Structurant

Pour rester tout à fait général, une structure d'imbrication est définie sur ces objets. Elle permet notamment de modéliser l'appartenance d'une boîte à une autre boîte rencontrée dans l'environnement de bureau classique.

A ce stade, une généralisation concernant le rangement des instances de la classe O-Info peut être introduite. Dans la première partie, nous avons supposé que les informations organisées en collections possèdent des localisations physiques similaires, le rangement logique étant alors confondu avec le rangement physique. Dans un environnement de bureau étendu, la définition de la classe Collection s'appuie sur un rangement de type purement logique et on peut très bien imaginer que ses composants, qu'ils soient informations ou collections, possèdent des localisations très diverses dans l'environnement de rangement. Nous doterons donc ces objets d'une possibilité de rangement autonome sans tenir compte de leur regroupement en collections. Dès lors, un même objet informationnnel peut appartenir à plusieurs collections différentes. Cette extension pose des problèmes concernant la suppression des collections dans l'environnement. Différents cas sont à envisager : supprimer la collection et tous ses composants (et récursivement si un des composants est une collection) et supprimer uniquement la collection, supprimer la collection avec les composants qui n'appartiennent qu'à cette collection.

2.5. La classe Objet de Rangement Terminal

Les caractéristiques principales des objets de rangement terminaux (armoire, table et poubelle) sont qu'ils ne peuvent être accueillis par aucun autre objet de l'environnement de bureau et qu'ils peuvent accueillir tous les objets instances de la classe Objet Rangeable. La classe Objet de Rangement Terminal (O-Rgt-Term) sera donc construite par héritage multiple sur les classes Objet Non Rangeable et Objet de Rangement

Cette classe a pour classes dérivées les classes Poubelle, Table et Armoire (voir figure 3.8). Contrairement aux classes Boîte, Classeur et Compartiment, elles ne peuvent être agrégées car

elles possèdent chacune des caractéristiques qui leur sont propres : ajouter un objet de rangement structurant à l'armoire, consulter une information sur la table, vider la poubelle...

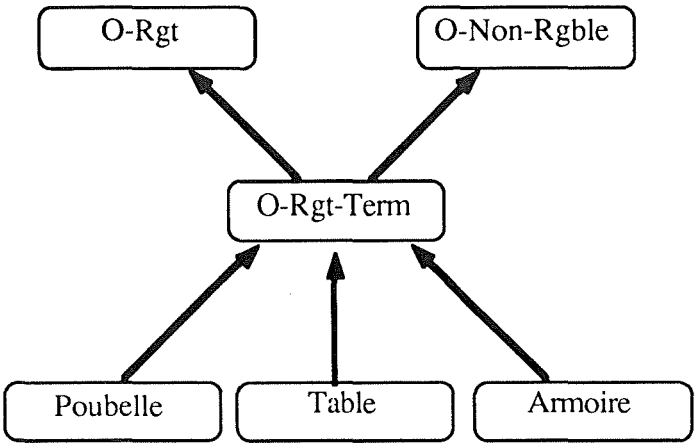

Figure 3.8 : Les classes de base et les classes dérivées de la classe O-Rgt-Term

La dernière extension de la hiérarchie d'inclusion d'un environnement de bureau classique portera sur les armoires. Ces dernières ne pouvaient contenir des informations et des collections que par l'intermédiaire des tiroirs. Nous levons cette dernière restriction à nouveau par souci d'extensibilité du modèle final. La description la plus générale consiste à dire que toute armoire peut contenir directement toute instance de la classe Objet Rangeable.

2.6. Les relations d'inclusion entre les objets de bureau

La hiérarchie d'objets qui vient d'être construite résulte de la généralisation du schéma d'un environnement de bureau classique. Les extensions introduites ont pour principale origine la suppression de restrictions sur les fonctionnalités des objets de bureau liées à la structure physique de ceux-ci. Il nous a semblé utile de fournir une nouvelle description des relations d'inclusion obtenues suite à l'élimination de ces contraintes (voir figure 3.9).

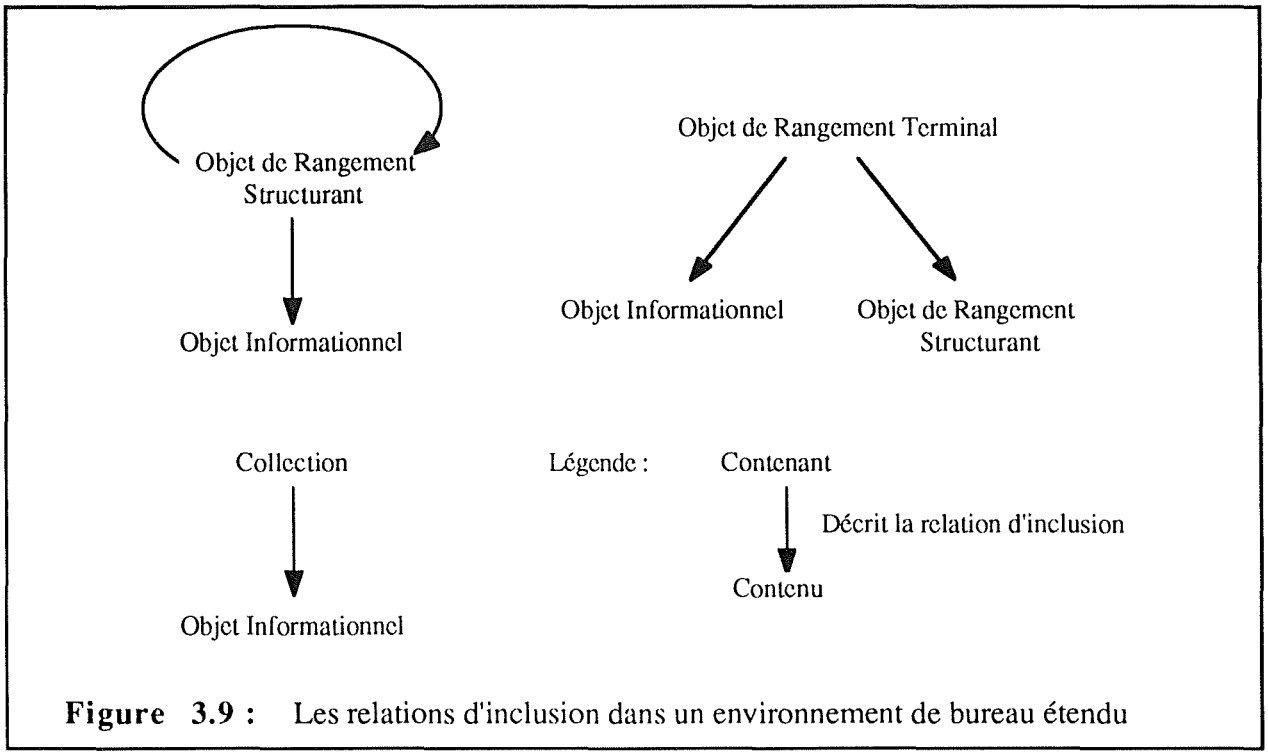

Figure 3.9 : Les relations d'inclusion dans un environnement de bureau étendu

Les relations d'inclusion sont beaucoup plus simples que celles du schéma initial et sont supportées par une hiérarchie de classes d'objets suffisamment générale pour pouvoir être étendue et enrichie par la suite. De plus, ces relations peuvent être décrites de manière encore plus concise à l'aide des deux propositions suivantes :

- **tout objet rangeable est contenu dans un objet de rangement** (description du rangement physique). En effet, tous les objets des classes O-Info et O-Struct sont aussi des objets rangeables,
- **les objets informationnels peuvent être assemblés pour constituer des collections** (description du rangement logique).

A elles seules, ces deux propositions suffisent pour décrire l'ensemble des inclusions d'objets dans un environnement de bureau. Ce sont ces deux relations qui sont retenues pour la représentation des classes dans le formalisme E/A, comme nous le verrons à la section suivante.

2.7. La structure des documents : la classe Fragment

Pour rendre compte de la structure des documents décrite à la partie 3.1.1 "L'information", la classe Fragment a été créée. Le fragment peut être vu comme un "morceau" de document. La structure du document est modélisée par la décomposition de ses fragments en fragments de plus en

plus fins. Ainsi, un livre se découpe en parties, chapitres, sections, paragraphes... L'intérêt de cette structure est de pouvoir accéder individuellement à des morceaux de document.

De manière plus précise, un document est modélisé par une structure arborescente dont chacun des noeuds est un fragment. L'ensemble de cette structure est appelée **arbre des fragments.**

La racine de l'arbre est le fragment **origine**, il réalise la liaison entre le document propriétaire des fragments et son arbre des fragments.

Un fragment est **terminal** dans l'arbre des fragments s'il n'est pas décomposé en d'autres fragments. Le plus souvent, l'indécomposabilité est liée à la nature du fragment. Si celui-ci possède un mode de représentation unique (par exemple : texte, graphique,...) alors l'accès à son contenu est possible par l'activation d'un seul processeur (par exemple : un traitement de texte) et le fragment est terminal. Cependant, un fragment terminal n'ayant pas un mode de représentation unique pourrait aussi être envisagé. En fait, cette décomposition est arbitraire et relève d'un choix personnel réalisé par l'utilisateur du document. Tout dépend de la précision avec laquelle il veut décrire le contenu du document. En particulier, un document ne possédant aucun fragment peut être envisagé si l'on ne s'intéresse qu'à son rangement dans l'environnement.

Tous les fragments de l'arbre non origine ou non terminaux sont dits **intermédiaires**.

Un document peut avoir un nombre quelconque de **niveaux de décomposition**, c'est-à-dire d'étages dans son arbre des fragments. A chaque niveau, les fragments reçoivent un **numéro de séquence** de manière à pouvoir reconstituer l'ordre logique de ceux-ci dans l'arborescence. Le déplacement d'un fragment dans le document correspond au déplacement de l'entièreté de la sous-arborescence dont ce fragment est la racine.

Pour mieux comprendre les notions introduites, celles-ci sont illustrées à l'aide de l'exemple suivant : un document de type classique, c'est-à-dire obéissant à une structure découpée en sous-chapitres telle qu'elle est représentée à la figure 3.10.

Le fragment origine de l'arbre est le document lui-même. Les valeurs attribuées aux propriétés de ce fragment fournissent les caractéristiques propres au document. En particulier, un document peut ne pas être caractérisé.

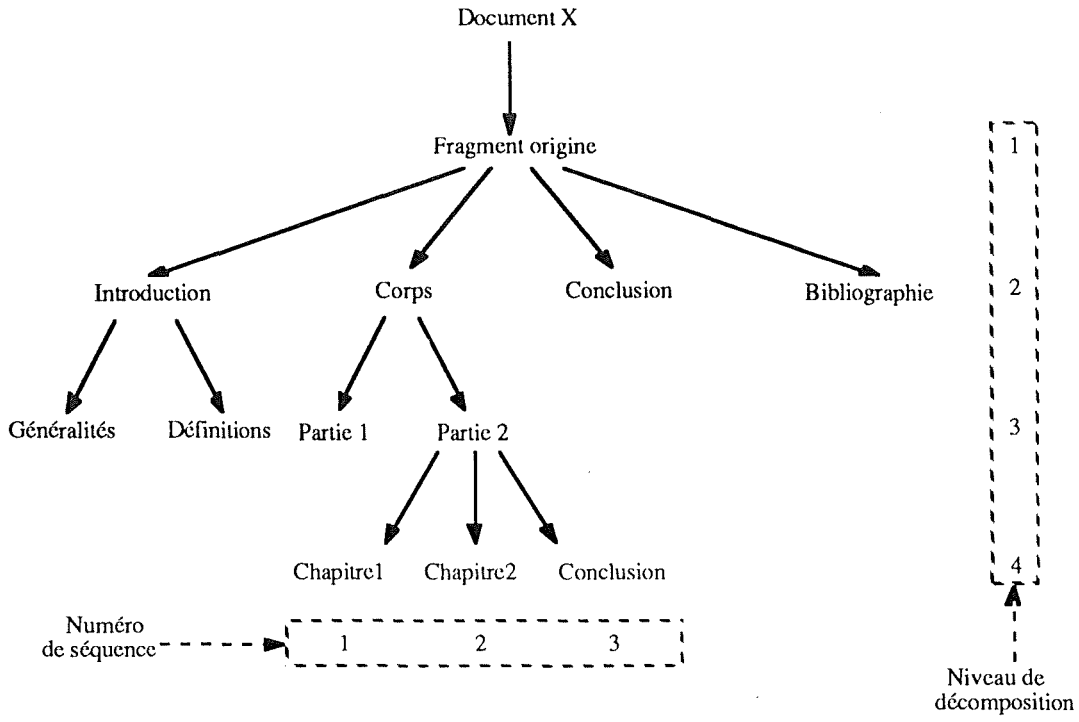

Figure 3.10 : Illustration de la découpe en fragments d'un document

Comme illustré par cette figure, le document est divisé en 4 niveaux de décomposition. Les noeuds de l'arbre prénommés "Généralités" ou "Partie 1" constituent des fragments terminaux. Ils sont, par exemple, activables à l'aide d'un processeur de traitement de texte. Par contre, "Corps" est un fragment intermédiaire et ne peut être directement activé. Les composants de "Corps" sont décrits par l'arborescence dont le fragment "Corps" est la racine. Au niveau 4, les numéros de séquence attribués aux fragments ont été indiqués. Ainsi, leur ordre d'apparition dans la "Partie 2" est connue.

Grâce à cette modélisation, n'importe quelle structure de document est représentable car les fragments peuvent être agencés de manière quelconque. Cette méthode de représentation permet d'introduire une modélisation simplifiée de la structure d'hypertexte. Pour ce faire, la description est complétée par l'introduction du concept de **référenciation** : n'importe quel fragment peut pointer sur n'importe quel autre fragment. Ainsi, des liens peuvent être créés entre les fragments d'un même document mais aussi, avec des fragments appartenant à des documents séparés.

Dans l'exemple, la partie bibliographie ou encore une note en bas de page dans la "Partie 1" pourraient faire référence à d'autres documents en pointant sur le fragment principal de ceux-ci ou à une partie de ce document en pointant sur "Introduction", etc.

Le fragment ne rentre pas dans la classification des objets de bureau. Il n'a aucun caractère rangeable et il n'est pas non plus directement manipulable dans l'environnement de rangement : toute action sur le fragment étant contingente au document auquel il appartient. Pour ces raisons, les fragments constituent une classe autonome. Ce choix implique la redéfinition de certaines méthodes et propriétés au niveau du fragment mais il permet de faciliter la gestion des objets de bureau.

L'indépendance de la classe Fragment évite notamment la définition de contraintes supplémentaires sur l'environnement, par exemple, celles relatives à l'interdiction de ranger un fragment indépendamment de son document propriétaire. De plus, les services offerts par cette classe d'objets tels que la création ou la suppression d'un fragment n'ont pas la même signification que la création d'un objet de bureau car au niveau de l'environnement de bureau, elles correspondent à une modification d'un document (objet déjà existant).

Les avantages retirés de l'indépendance de cette classe apparaîtront clairement lors de la description du schéma E/A qui en sera simplifié et lors de la présentation des méthodes associées aux classes définies.

2.8. Schéma de la hiérarchie de classes

L'introduction de la classe Objet de Bureau et des classes dérivées, descendantes directes de celle-ci (O-Info, O-Rgt, O-Non-Rgble et O-Rgble) fournit un support à la définition de nouvelles classes d'objets non présentes dans le modèle initial. Celles-ci peuvent être ajoutées en se servant des classes existantes qui leur apportent un ensemble de méthodes prédéfinies et réutilisables par le mécanisme d'héritage. Les objets du modèle initial sont maintenant représentés par les classes obtenues par spécialisation et enrichissement des classes O-Info, O-Rgt, O-Non-Rgble et O-Rgble. Le schéma des classes finalement obtenu est illustré à la figure 3.11.

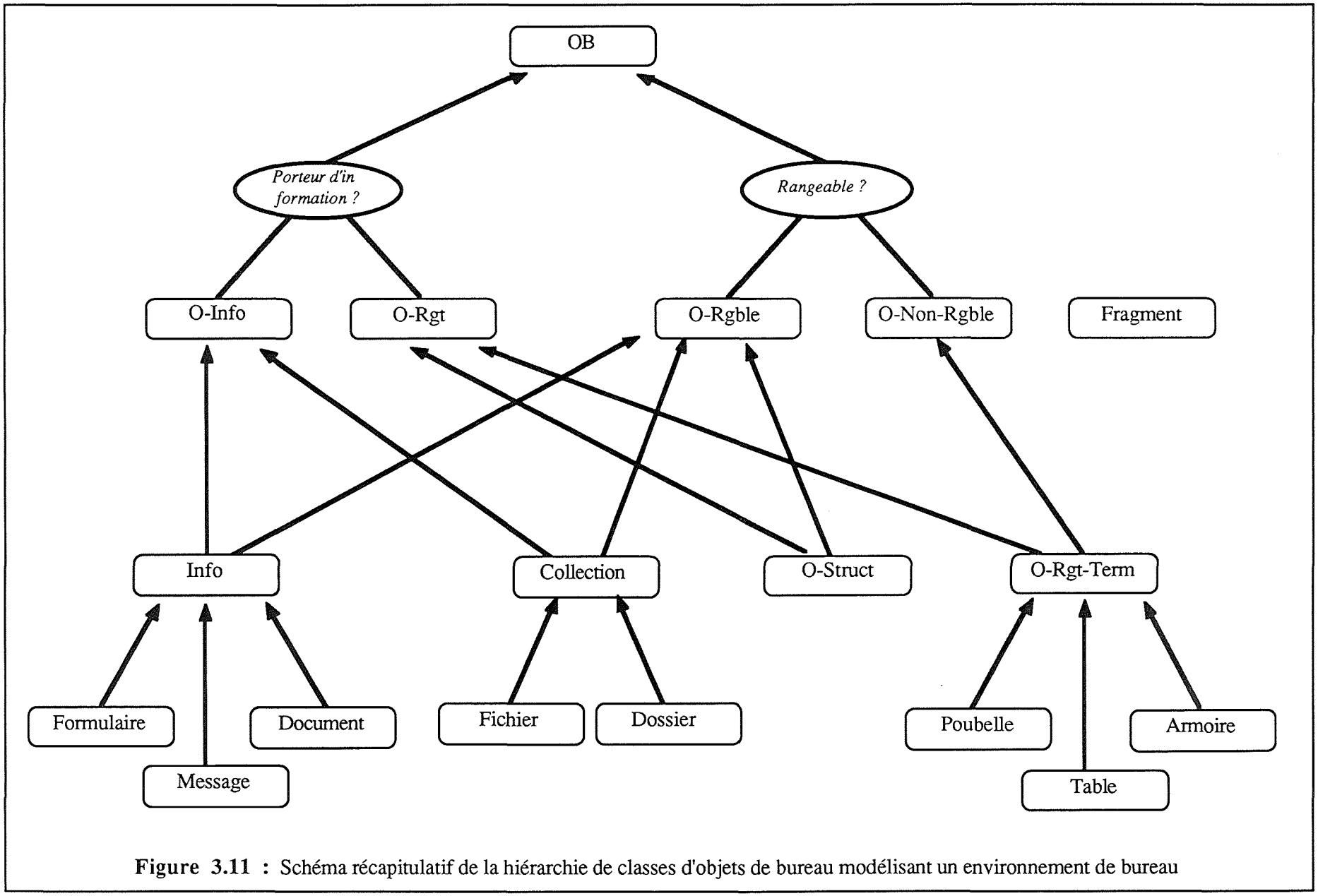

Figure 3.11 : Schéma récapitulatif de la hiérarchie de classes d'objets de bureau modélisant un environnement de bureau

3. Schéma E/A de l'environnement de bureau étendu

La section 3.1 "Présentation d'un environnement de bureau classique" était consacrée à la description générale des objets de bureau d'un environnement de rangement d'informations. L'environnement de bureau étendu reprenait ces mêmes objets et en introduisant un niveau d'abstraction supplémentaire, les agrège en classes d'objets suffisamment générales pour supporter des extensions ultérieures. La généralisation portait sur l'élimination des contraintes liées à la nature physique des objets (contenance maximale, lieu de rangement unique pour les collections). Il est à noter que ces contraintes peuvent être réintégrées dans le modèle grâce à la création de nouvelles classes d'objets spécifiques, héritant des classes définies, et possédant des méthodes qui filtreraient les méthodes générales impliquant ainsi le respect de ces contraintes.

Une fois la hiérarchie de classes d'objets fixée, il nous reste à définir sa représentation sous forme E/A puis les interfaces des classes d'objets afin de terminer l'analyse conceptuelle. Le schéma de la base de données obtenu fournit le support nécessaire au stockage permanent des objets, instances des classes définies. Pour ce faire, un formalisme E/A étendu décrit dans [HAIN-89] et qui intègre, dans sa description des données, le mécanisme d'héritage grâce à la relation **is-a** est utilisé. Cette relation (notée $\uparrow$) implique que tout type d'entité B jouant le rôle spécifique dans une relation is-a définie sur un type d'entité A hérite de toutes les propriétés propres à la description de A. Le type d'entité B est alors décrit par l'ensemble des propriétés des types d'entités A et B, à savoir leurs attributs et les associations auxquels ils participent.

Les classes définies précédemment sont matérialisées par des types d'entités de même nom. Les relations is-a rendent compte de la structure d'héritage. Par exemple, un objet appartenant à la classe Objet Informationnel est stocké dans la base de données par une occurrence du type d'entité OB et une occurrence du type d'entité O-Info. L'existence d'une relation is-a entre ces deux types d'entités indique qu'elles décrivent le même objet.

Grâce à l'extension au modèle E/A fournie par les relations is-a, les types d'entités sont obtenus par traduction directe des classes d'objets de l'environnement de bureau étendu. Le schéma E/A final est doté d'une structure d'héritage similaire à celle du schéma des classes d'objets de manière à prévoir l'extensibilité future de la base de données. En calquant le schéma des données sur le schéma de classes, on espère faciliter les modifications futures, toute introduction d'une nouvelle classe d'objet pouvant être traduite au niveau base de données par l'ajout d'un nouveau type d'entité.

Cette partie est consacrée à la traduction du schéma des classes d'objets en schéma E/A. Les types d'entités sont présentés dans l'ordre d'apparition des objets dans la hiérarchie de

classes. Leurs attributs traduisent, dans le formalisme E/A, les propriétés des objets. Ils seront accessibles via les méthodes de l'interface de chaque classe d'objets.

Pour plus de clarté, le schéma E/A est décomposé en deux parties :

- la première partie décrit les objets de bureau de la hiérarchie (figure 3.12), ce schéma découle du schéma de classes de l'environnement de bureau étendu,
- la deuxième partie est spécifique à la classe Document et à sa découpe en fragments (voir infra figure 3.13), le fragment étant le concept choisi pour modéliser leur structure.

3.1. Le sommet de la hiérarchie : Objet de Bureau

La classe racine de la hiérarchie d'objets est la classe OB. Elle est traduite dans le formalisme E/A par le type d'entité OB qui reprend tous les attributs communs à tous les objets de bureau.

a) Définition

L'entité **Objet de Bureau** (OB) représente tout objet de bureau appartenant à l'environnement de bureau (figure 3.12). Il s'agit soit d'informations telles que des documents, soit de collections telle que le fichier ou la pile, soit d'objets de rangement tels que la table de travail.

Toutes les propriétés des objets ont été regroupées dans la classe la plus générale pour laquelle elles avaient un sens. Au niveau schéma conceptuel, cela signifie que tous les attributs correspondant sont repris dans le type d'entité le plus général matérialisant la classe concernée. Les attributs repris ci-dessous ont donc une signification pour tous les objets de la hiérarchie.

La description des attributs reprend, pour chacun d'eux, leur définition, leurs propriétés, leur représentation et, quand cela est utile, quelques exemples de valeurs envisageables.

b) Attributs

- **Nom :**

Définition : cet attribut désigne le nom de l'objet de bureau. Il est déterminé à la création de l'objet.
Propriétés : attribut simple, élémentaire, obligatoire et identifiant.
Représentation : type chaîne de caractères.

- **Description :**

Définition : c'est un commentaire libre et personnel, réalisé par le créateur de l'objet.
Propriétés : attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type chaîne de caractères.

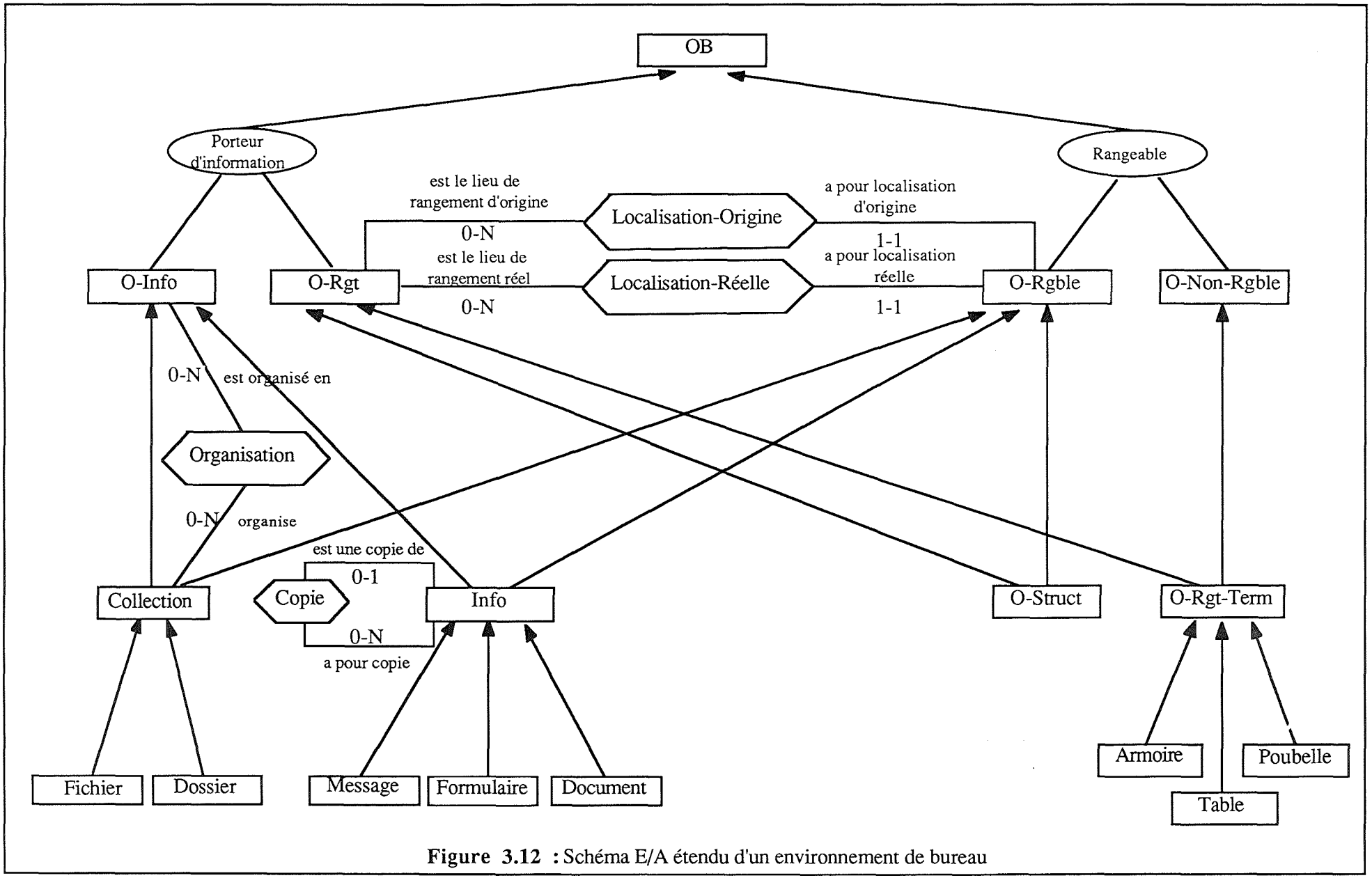

Figure 3.12 : Schéma E/A étendu d'un environnement de bureau

Exemples de valeurs : "pièce justificative", "rapport complémentaire".

- **Créateur :**

Définition : cet attribut fournit le nom du créateur ou du responsable de l'objet de bureau.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

- **Date-création :**

Définition : la date de création détermine la date à laquelle un objet de bureau a été créé. Pour un objet de rangement terminal, elle correspond à la date d'entrée de l'objet dans le bureau.
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date (type décomposable en jour, mois et année).

- **Propriétaire :**

Définition : cet attribut détermine le nom du propriétaire de l'objet de bureau. Ce nom peut être distinct du nom du créateur.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

- **Date-der-cons :**

Définition : cet attribut fournit la date à laquelle un objet a été manipulé pour la dernière fois. Le traitement peut consister soit en des consultations des propriétés de l'objet soit en des manipulations réelles de celui-ci : déplacement, nouveau classement, mise à jour...
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date (défini ci-dessus).

- **Date-der-maj :**

Définition : cet attribut fournit la date de modification des caractéristiques de l'objet de bureau la plus récente. La mise à jour de sa valeur se fait automatiquement, par exemple, lors de la modification des attributs d'un objet de bureau.
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date.
Remarque : toute mise à jour d'un objet entraîne la modification des attributs date-der-maj et date-der-cons.

3.2. Les classes résultant du partitionnement

Les classes résultant de l'application des critères de partitionnement définis dans la partie 2.2.1 "Les critères de partitionnement et les classes résultantes" sont les classes O-Info, O-Rgt, O-Rgble et O-Non-Rgble. Chacune d'entre elles se traduit par un nouveau type d'entité dans le schéma E/A qui est relié au type d'entité OB par un type d'association particulier exprimant le partitionnement. Il s'agit de deux types d'associations multidomaines dont le rôle générique est joué par une occurrence du type d'entité OB et le rôle spécifique est soit joué par une occurrence du type d'entité O-Info, soit par une occurrence du type d'entité O-Rgt (pour le critère de partitionnement "porteur d'information") et soit joué par une occurrence du type d'entité O-Rgble, soit par une occurrence du type d'entité O-Non-Rgble (pour le critère de partitionnement "rangeable").

Rappelons que, par le mécanisme d'héritage, chacune des instances de ces classes est décrite par l'ensemble des attributs de l'entité spécifique le représentant ainsi que par les attributs de l'occurrence du type d'entité OB.

3.2.1. Entité Objet Informationnel

a) Définition

Un **Objet Informationnel** (O-Info) est un type d'objet de bureau porteur d'information (premier critère de partitionnement). Dans le modèle d'environnement de bureau étendu, les classes Collection et Information sont des classes dérivées de la classe Objet Informationnel, elles se traduiront par deux nouvelles relations is-a.

b) Attributs

- **Mots-clés :**

Définition : cet attribut décrit le contenu de l'objet informationnel sous forme de mots clés ayant un caractère thématique.

Propriétés : attribut répétitif, élémentaire, facultatif et non identifiant.

Représentation : type chaîne de caractères.

Exemples de valeurs : un fichier "médical", un formulaire de "réception marchandises", un formulaire "inscription".

3.2.2. Entité Objet de Rangement

a) Définition

Le type d'entité **O-Rgt** provient de la classe Objet de Rangement et représente les objets de bureau non porteurs d'information, ces objets servent au rangement des objets de bureau de la sous-classe Objet Rangeable.

b) Attributs

La classe d'objets Objet de Rangement héritant de tous les attributs de la classe OB, aucun attribut propre à la classe d'objet O-Rgt n'a été défini. La non existence d'attributs n'est pas, dans ce cas précis, un signe que le type d'entité O-Rgt devrait disparaître car ce type d'entité est le support de plusieurs autres classes descendantes et, la classe Objet de Rangement possède plusieurs méthodes qui lui sont propres. En particulier, l'existence de deux types d'associations dans lesquelles ce type d'entité est impliqué différencie fortement les instances de la classe d'objets Objet de Rangement des instances de la classe d'objets OB.

3.2.3. Entité Objet Rangeable

a) Définition

Les Objets Rangeables résultent, tout comme les Objets de Rangement, du deuxième critère de partitionnement des objets de bureau. Ces objets représentent les objets qui sont rangeables dans d'autres objets de l'environnement de bureau. Ils sont représentés dans le formalisme E/A par le type d'entité **O-Rgble**. Les structures de rangement auxquelles ils peuvent participer sont matérialisées par les associations Localisation-Réelle et Localisation-Origine.

b) Attributs

Aucun attribut spécifique à la classe Objet Rangeable n'a été défini. Cette sous-classe hérite déjà des propriétés définies pour la classe Objet de Bureau.

3.2.4. Associations entre Objet Rangeable et Objet de Rangement

L'étude d'un environnement de bureau classique a montré qu'il existait des relations d'inclusion entre les objets d'un tel environnement. Par la suite, ces relations ont été généralisées pour aboutir à une relation unique décrivant l'imbrication des objets de bureau. Cette relation est définie sur les classes d'objets O-Rgt et O-Rgble et est traduite, dans le schéma E/A, par deux types d'associations appelées Localisation-Réelle et Localisation-Origine.

A ce niveau, on peut remarquer que les classes O-Rgt et O-Rgble sont reliées conceptuellement et apparaissent comme un cluster (voir section 2.2 "Méthodologie de développement orientée objets"). Ceci est visible, en particulier, dans le schéma E/A qui introduit deux associations entre elles. Déjà, au niveau de la construction du schéma de classes, cette interrelation entre les deux classes était apparue clairement lors de la présentation des relations d'inclusion. L'énoncé de la première proposition : "tout objet rangeable est contenu dans un objet de rangement" en était la confirmation (voir partie 3.2.6 "Les relations d'inclusion entre les objets de bureau).

3.2.4.1. L'association Localisation-Origine

a) Description

Cet type d'association fournit l'objet de rangement qui est la localisation d'origine d'un objet rangeable. Elle doit prendre une valeur lors de la création de l'objet de bureau si celui-ci est rangeable et peut être modifiée par la suite.

b) Rôles

– "est le lieu de rangement d'origine" joué par le type d'entité O-Rgt est de connectivité 0-N. Un objet de rangement peut, en effet, être vide ou contenir un nombre quelconque d'objets rangeables. Rappelons que la notion de capacité maximale d'un objet de rangement n'a aucun sens dans le modèle d'environnement de bureau étendu.

– "a pour localisation d'origine" joué par le type d'entité O-Rgble est de connectivité 1-1. Tout objet rangeable possède une et une seule localisation d'origine.

3.2.4.2. L'association Localisation-Réelle

a) Description

Un objet rangeable peut avoir, momentanément, une localisation réelle différente de sa localisation d'origine. Il peut, par exemple, être situé sur la table de travail en vue de subir un

traitement. Cette association permet de garder une trace du dernier endroit où un objet rangeable est rangé.

b) Rôles

– "est le lieu de rangement réel" joué par la classe Objet de Rangement est de connectivité 0-N. Un objet de rangement peut être vide ou contenir un nombre quelconque d'objets rangeables.

– "a pour localisation réelle" joué par la classe Objet Rangeable est de connectivité 1-1. Tout objet rangeable possède un et un seul lieu de rangement réel.

3.2.5. Entité Objet Non Rangeable

La dernière classe obtenue par application des critères de partitionnement est la classe O-Non-Rgble. Cette classe est traduite, sans difficulté particulière, par le type d'entité **O-Non-Rgble**.

a) Définition

Les objets de la classe Objet Non Rangeable représentent les objets qui ne peuvent être rangés dans aucun autre objet de l'environnement de bureau, à savoir tous les objets appartenant aux classes Armoire, Poubelle et Table.

b) Attributs

Pour les mêmes raisons que pour les classes Objet de Rangement et Objet Rangeable, aucun attribut spécifique n'a été défini pour la classe Objet Non Rangeable.

3.3. La classe Information et ses classes dérivées

Les classes étudiées dans cette section sont les classes obtenues par héritage multiple sur les classes O-Info et O-Rgble, à savoir la classe Info et, ses classes dérivées, les classes Formulaire, Message et Document.

3.3.1. Entité Information

a) Définition

Une information est un type particulier d'objet de bureau contenant directement l'"information". Cette classe hérite des propriétés des classes Objet Informationnel et Objet Rangeable ce qui définit deux nouvelles relations is-a dans le schéma E/A. Ainsi, toute occurrence

du type d'entité Information participe aux rôles "a pour localisation d'origine" et "a pour localisation réelle".

b) Attributs

- **Présence :**

Définition : la présence de l'information dans l'environnement de bureau est connue grâce à cet attribut. Elle peut, par exemple, avoir été empruntée par un employé et est alors, momentanément, indisponible.

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type booléen.

- **Confidentialité :**

Définition : la confidentialité permet de savoir si l'accès à l'information est réglementé. L'attribut Confidentialité peut être seulement modifié par les personnes ayant la connaissance de la valeur de l'attribut Mot-de-passe de l'information.

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type booléen.

- **Mot-de-passe :**

Définition : cet attribut donne la valeur de la clé d'accès à connaître par l'utilisateur pour pouvoir accéder au contenu de l'information.

Propriétés: attribut simple, élémentaire, facultatif et non identifiant.

Représentation : type chaîne de caractères.

Remarque : si l'information est classée confidentielle alors l'attribut Mot-de-passe doit nécessairement prendre une valeur.

- **Support :**

Définition : cet attribut donne la nature du support sur lequel l'information est stockée. Elle peut, par exemple, être stockée sur papier, support magnétique...

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type chaîne de caractères.

3.3.2. Association Copie

a) Description

Lors de l'étude d'un environnement de bureau étendu, il a été question d'une méthode permettant de copier l'information. Le stockage des copies d'informations est réalisé de la même

manière que le stockage de l'information, le lien avec l'original étant conservé grâce au type d'association récursif Copie.

b) Rôles

– "est une copie de" joué par le type d'entité Info est de connectivité 0-1 car une Information est la duplication d'au plus une autre information.

– "a pour copie" joué par le type d'entité Info est de connectivité O-N. Une information peut être copiée un nombre quelconque de fois.

3.3.3. Entité Document

a) Définition

La classe **Document** est définie comme une sous-classe de l'entité **Information**, l'entité Document sera donc reliée, en tant que type spécifique, par une relation is-a à l'entité Information.

b) Attributs

Etant donnée la décomposition du document en fragments, tous les attributs propres au type d'entité Document sont reportés au niveau du type d'entité Fragment. Les caractéristiques du document seront celles fournies par le fragment origine de son arbre des fragments. La description des documents sous forme de fragments fait l'objet d'un paragraphe spécifique.

3.3.4. Entité Message

a) Définition

Le **Message** est une information représentée sous forme écrite. Il est généralement bref et sans structure particulière. Ce type d'entité joue le rôle spécifique dans une relation is-a avec le type d'entité Info. Des attributs spécifiques viennent enrichir la définition de la classe, ils sont décrits ci-dessous.

b) Attributs

- **Mode-repr-mess :**

Définition : cet attribut mémorise le mode de représentation du message. Il peut être écrit, graphique, imagé, sonore...
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

- **Stockage-mess :**

Définition : cet attribut détermine si le contenu du message se trouve dans la base de données.
Propriétés: attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type booléen.

- **Contenu-mess :**

Définition : le contenu représente le corps proprement dit du message.
Propriétés : attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type texte.
Remarque : le contenu du message ne peut être stocké dans la base de données que si l'attribut Mode-repr-mess prend la valeur texte et l'attribut Stockage-mess prend la valeur vrai.

- **Lieu-stockage-mess :**

Définition : cet attribut détermine le lieu où l'on peut trouver le message si celui-ci n'est pas effectivement stocké dans la base de données.
Propriétés : attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type chaîne de caractères.
Remarque : cet attribut prend une valeur seulement si l'attribut Stockage-mess possède la valeur faux.

3.3.5. Entité Formulaire

a) Définition

Le **Formulaire** est une information écrite, structurée et composée de champs obéissant à une structure prédéfinie. Les attributs de ce type d'entité sont similaires à ceux définis pour le type d'entité Message. Cette similitude pourrait être un signe qu'une agrégation des deux classes est possible. Cependant, ces classes ne peuvent être agrégées à ce niveau car elles décrivent des objets différents qui, par conséquent, peuvent posséder des méthodes spécifiques. Par exemple, la classe Formulaire pourrait posséder des méthodes relatives à la manipulation de son contenu qui exploiteraient sa structure squelettique, inexistante dans le document et le message.

b) Attributs

- **Mode-repr-form :**

Définition : cet attribut donne le mode de représentation du formulaire. Il peut être écrit, graphique, sur papier, électronique, sur micro-films...
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

• **Stockage-form :**

Définition : cet attribut détermine si le contenu du formulaire se trouve dans la base de données.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type booléen.

• **Contenu-form :**

Définition : comme pour l'entité précédente, le contenu du formulaire donne le corps du formulaire.
Propriétés : attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type texte.

• **Lieu-stockage-form :**

Définition : cet attribut détermine le lieu où l'on peut trouver le formulaire si celui-ci n'est pas effectivement stocké dans la base de données.
Propriétés : attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type chaîne de caractères.
Remarque : les contraintes prises par les attributs le type d'entité Formulaire sont identiques à celles définies pour le type d'entité Message.

3.4. La classe Collection et ses classes dérivées

3.4.1. Entité Collection

a) Définition

La définition du concept de **Collection** telle que nous l'entendons dans ce travail a été donnée dans le chapitre 2 "Présentation d'un environnement de bureau étendu". Il s'agit d'un concept étendu dérivé lors de la définition de l'objet réel mais généralisé dans le but de supporter les extensions futures du modèle.

b) Attributs

• **Critère-classement :**

Définition : cet attribut définit le critère de classement applicable aux composants d'une collection, c'est-à-dire le critère sur lequel la relation d'ordre est établie.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.
Remarque : la valeur aucun est obligatoire pour la pile.

• **Ordre-classement :**

Définition : cet attribut caractérise la méthode de classement des composants d'une collection. On distingue traditionnellement quatre méthodes de classement qui sont : classement alphabétique, numérique, chronologique et idéologique [MART-82]. Ces méthodes de classement sont complétées par le mode de classement manuel qui correspond à un ordre d'insertion quelconque déterminé par l'utilisateur.

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type énuméré dont le domaine de valeurs est le suivant : [alphabétique, alphabétique inverse, numérique croissant, numérique décroissant, chronologique, anti-chronologique, idéologique, manuel, aucun].

Exemple : pour une collection telle qu'un fichier de dossiers médicaux, les critères suivants constituent un exemple de valeurs envisageables :

Ordre-classement	**Critère-classement**
alphabétique	nom du patient
numérique	numéro du dossier
chronologique	date de création d'un dossier
idéologique	par maladies
manuel	—

3.4.2. Description de l'organisation des informations en collections

a) Description

Ce type d'association décrit l'organisation d'objets informationnels en collections. Il est tout-à fait général et, par le fait qu'il porte sur les objets informationnels, modélise aussi bien les collections de collections que les collections d'informations. L'organisation des informations en collections n'implique aucune contrainte sur les structures de rangement physique des objets concernés (voir figure 3.12).

b) Rôles

– "est organisé en" joué par le type d'entité O-Info est de connectivité 0-N. Du fait de la nature purement logique d'une collection, un même objet informationnel peut appartenir à un nombre quelconque de collections. La connectivité minimale s'explique par le fait qu'un objet informationnel peut ne faire partie d'aucune collection et, est alors rangé directement dans un objet de rangement de l'environnement de bureau.

– "organise" joué par le type d'entité Collection est de connectivité 0-N. Une collection peut être vide, elle peut aussi contenir un nombre quelconque d'objets informationnels.

3.4.3. Entité Fichier

a) Définition

Le **fichier** est une Collection de composants de même type appartenant aux classes Information et Collection ou, plus généralement, de la classe Objet Informationnel.

b) Attributs

• **Type-composant :**

Définition : cet attribut donne le type des composants de la collection.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

3.4.4. Entité Dossier

a) Définition

Le **Dossier** est une collection d'objets informationnels reliés entre eux par un lien logique. Quant à la **Pile**, elle est définie comme une collection d'objets informationnels qui n'ont pour lien que leur ordre d'arrivée dans la pile. Rappelons que, dans le schéma E/A, les notions de pile et de dossier sont agrégées.

b) Attributs

Les attributs du type d'entité Dossier ont déjà été développés lors du traitement des types d'entités OB, O-Info et Collection puisque la classe Dossier hérite des propriétés des classes OB, Objet Informationnel et Collection.

3.5. La classe Objet de Rangement Structurant

L'ensemble des objets de rangement intermédiaires a été agrégé en une classe unique appelée O-Struct. Remarquons que, par le mécanisme d'héritage, le type d'entité O-Struct participe à tous les rôles des associations Localisation-Réelle et Localisation-Origine, ce qui traduit l'imbrication possible des objets de rangement structurants les uns dans les autres.

a) Définition

La classe **Objet de Rangement Structurant** (O-Struct) hérite des propriétés des classes Objet de Rangement et Objet Rangeable. Elle comprend, entre autres, le compartiment, le tiroir, la boîte et le classeur décrits dans l'environnement de bureau classique mais elle peut aisément être étendue à d'autres objets du même type.

Un objet de rangement structurant étant un objet de rangement rangeable, son existence dans la base de données se traduit par une occurrence des entités O-Rgt et O-Rgble, ce qui implique l'existence d'une contrainte d'intégrité interdisant qu'un objet de rangement structurant ne soit son propre lieu de rangement, ceci directement aussi bien qu'indirectement, via un ou plusieurs autres objets de rangement structurants.

b) Attributs

- **Critère-regroup :**

Définition : comme pour la classe Collection, le critère de rangement des objets de rangement structurants définit le critère sur lequel ces objets seront rangés.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

- **Ordre-regroup :**

Définition : cet attribut détermine la méthode de classement des composants d'un objet de rangement.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : les valeurs définies pour cet attribut sont similaires à celles définies pour l'attribut Ordre-classement du type d'entité Collection.

- **Etiquette :**

Définition : cet attribut permet à l'utilisateur d'apposer une étiquette sur l'objet de rangement structurant. Elle peut, par exemple, le renseigner sur l'ordre de regroupement caractéristique de l'objet de rangement structurant ou tout simplement lui donner un renseignement quelconque sur le contenu de l'objet de rangement structurant.
Propriétés : Attribut simple, élémentaire, facultatif et non identifiant.
Représentation : type chaîne de caractères.
Exemples de valeurs : "de A à H", "1980", "Dossiers médicaux",...

3.6. La classe Objet de Rangement Terminal et ses classes dérivées

La traduction de ces classes d'objets ne présente pas de difficultés particulières et est similaire à celles effectuées pour les classes Information, Collection et leur sous-classes.

La classe **Objet de Rangement Terminal** (O-Rgt-Term) reprend les objets de rangement non rangeables. Elle comprend les classes Armoire, Table et Poubelle. Cette classe héritant des attributs définis pour les classes O-Rgt et O-Rgble, elle se trouve déjà complètement décrite à ce niveau de raffinement.

L'**Armoire** est composée de divers objets de rangement structurants (compartiments ou les tiroirs) qui sont destinés à recevoir des objets rangeables Elle peut aussi contenir directement les objets informationnels.

La **Table** est un endroit où l'on peut ranger des objets uniquement en vue de leur traitement ultérieur. Elle constitue un lieu de passage temporaire.

La **Poubelle** est l'endroit où l'on range des objets en vue de leur destruction. Nous supposerons qu'un objet placé dans la poubelle peut y être récupéré tant que celle-ci n'a pas été vidée.

Ces trois types d'entités héritant des attributs de l'entité Objet de Rangement Terminal, leurs attributs se trouvent déjà décrits à ce niveau de développement.

3.7. Description de la structure des documents

La structure des documents se matérialise par un arbre des fragments. Pour modéliser cette structure sous forme E/A, le type d'entité Fragment est utilisé pour stocker les instances de la classe Fragment. La découpe des fragments en fragments de plus en plus fins est modélisée par le type d'association Décomposition (voir figure 3.13) tandis que les références à d'autres fragment sont traduits par le type d'association Référenciation.

Aucune structure de généralisation/spécialisation n'est définie sur le type d'entité Fragment car il provient d'une classe autonome.

Dans le schéma des classes d'objets, l'existence de cette classe autonome a pour conséquence de simplifier la définition de l'interface des objets (notamment les méthodes de mise à jour). Dans le schéma E/A, l'avantage procuré est similaire, l'autonomie de cette classe simplifiant fortement la

gestion des contraintes d'intégrité. En effet, un exemple simple suffit pour montrer les avantages apportés par ce choix de conception : si un fragment avait été une information, il aurait été un objet rangeable et aurait participer aux associations de localisation mais, cette dernière caractéristique n'a, dans leur cas, aucune signification. Une remarque identique s'impose pour l'association Organisation, etc...

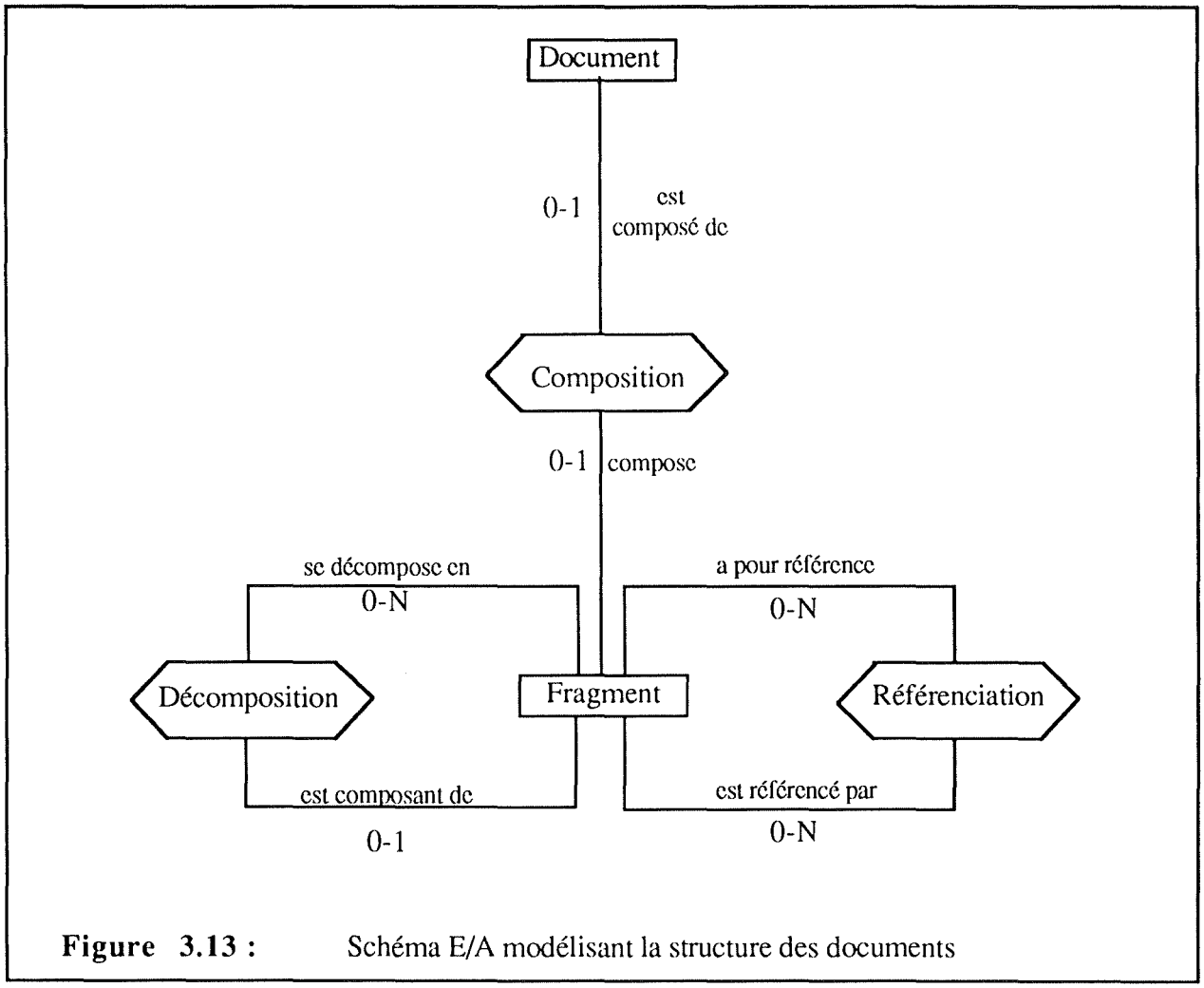

Figure 3.13 : Schéma E/A modélisant la structure des documents

3.7.1. Entité Fragment

a) Définition

Une définition complète de la notion de **Fragment** (voir figure 3.13) est donnée dans la section 3.2 "Présentation d'un environnement de bureau étendu". Comme les fragments forment une classe d'objets autonome, certains attributs déjà rencontrés sont redéfinis.

b) Attributs

- **Nom-frag:**

Définition : cet attribut donne le nom du fragment. Il est déterminé lors de la création du fragment.
Propriétés : attribut simple, élémentaire, obligatoire et identifiant local à un document.
Représentation : type chaîne de caractères.

- **Date-création-frag :**

Définition : la date de création détermine la date à laquelle un fragment a été créé. Cette date doit bien évidemment être postérieure à la date de création du document auquel le fragment appartient.
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date.

- **Date-der-cons-frag :**

Définition : cet attribut fournit la date à laquelle un fragment a été manipulé pour la dernière fois. Il est automatiquement mis à jour lors de toute consultation ou modification des attributs d'une instance de la classe Fragment. Il est également à noter que toute manipulation d'un fragment entraîne la mise à jour de l'attribut Date-der-cons du Document auquel le fragment appartient.
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date.

- **Date-der-maj-frag :**

Définition : cet attribut fournit la date la plus récente de modification des caractéristiques du fragment. La mise à jour de sa valeur est automatique, par exemple, lors de la modification des attributs du fragment.
Propriétés : attribut simple, décomposable, obligatoire et non identifiant.
Représentation : type date.

- **Support-frag :**

Définition : cet attribut détermine la nature du support sur lequel le fragment est stocké. Il peut, par exemple, être stocké sur papier, support magnétique. Il va de soi que deux fragments différents appartenant au même document peuvent être stockés sur des supports de nature différente.
Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.
Représentation : type chaîne de caractères.

- **Mode-repr-frag :**

Définition : cet attribut donne le mode de représentation du fragment. Il peut être écrit, graphique, imagé, sonore, mixte...

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type chaîne de caractères.

- **Stockage-frag :**

Définition : cet attribut permet de savoir si le contenu du fragment se trouve dans la base de données.

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type booléen.

- **Contenu-frag :**

Définition : le contenu représente le corps proprement dit du fragment.

Propriétés : attribut simple, élémentaire, facultatif et non identifiant.

Représentation : type texte.

Remarque : le contenu du fragment ne peut être stocké dans la base de données que si l'attribut mode-repr-frag possède la valeur texte et si l'attribut Stockage-frag prend la valeur vrai.

- **Lieu-stockage-frag :**

Définition : cet attribut fournit le lieu où l'on peut trouver le fragment si celui-ci n'est pas effectivement stocké dans la base de données.

Propriétés : attribut simple, élémentaire, facultatif et non identifiant.

Représentation : type chaîne de caractères.

Remarque : cet attribut prend une valeur seulement si l'attribut Stockage-frag prend la valeur faux.

- **Type-frag :**

Définition : cet attribut détermine le type du processeur à activer pour avoir accès au contenu du fragment et pouvoir le modifier. Pour les fragments de type texte par exemple, le processeur à activer est un traitement de textes.

Propriétés : attribut simple, élémentaire, facultatif et non identifiant.

Représentation : type chaîne de caractères.

3.7.2. Construction de l'arbre des fragments

3.7.2.1. L'association Composition

a) Description

Ce type d'association fournit le lien entre le fragment racine de l'arbre des fragments d'un document et le document dont l'arbre des fragments représente la structure (figure 3.13).

b) Rôles

– "est composé de" joué par le type d'entité Document est de connectivité 0-1. Un document peut très bien ne posséder aucun fragment, soit parce qu'il est vide, soit parce qu'aucune information concernant sa structure n'a été répertoriée. Dans ce cas précis, on s'attache uniquement à stocker dans la base de données son lieu de rangement.

– "compose" joué par le type d'entité Fragment est de connectivité 0-1. Seul le fragment racine de l'arbre est relié au document propriétaire. Le lien entre les autres fragments et le document s'obtient en remontant du fragment considéré vers la racine de l'arbre auquel il appartient, le document propriétaire de ce dernier étant connu.

3.7.2.2. L'association Décomposition

a) Description

Ce type d'association modélise, pour chaque fragment de l'arbre, la décomposition de ce fragment en ses fragment fils dans l'arbre.

b) Rôles

– "se décompose en" joué par le type d'entité Fragment est de connectivité 0-N. Un fragment peut avoir un nombre quelconque de fils, il peut aussi être terminal dans l'arbre.

– "est composant de" joué par le type d'entité Fragment est de connectivité 0-1. La connectivité 0 s'applique dans le cas où le fragment est une racine d'arbre, 1 sera la connectivité de tous les fragments descendants (conséquence directe de la définition d'un arbre).

c) Attributs

• **Num-seq :**

Définition : cet attribut donne le numéro d'ordre d'un fragment parmi l'ensemble des fragments appartenant à un même niveau de décomposition. Il permet de reconstituer la séquence des fragments appartenant à un même niveau de décomposition.

Propriétés : attribut simple, élémentaire, obligatoire et non identifiant.

Représentation : type entier.

3.7.3. Introduction du concept de Référenciation

a) Description

Ce type d'association permet de relier entre eux les fragments ayant un rapport quant à la sémantique de leur contenu. Par exemple, deux fragments seront reliés s'ils traitent de sujets complémentaires. Les fragments impliqués dans cette association peuvent appartenir à des documents différents.

b) Rôles

– "est référencé par" joué par le type d'entité Fragment a pour connectivité 0-N. Un fragment peut ne pas être référencé par un autre fragment, il peut également l'être par plusieurs autres fragments.

– "référence" joué par le type d'entité Fragment a pour connectivité 0-N. Un fragment peut ne référencer aucun fragment, il peut , de même, en référencer plusieurs.

3.8. Contraintes d'intégrité

Les contraintes sur les attributs seront gérées, de manière interne, par la classe d'objets correspondant au type d'entité sur lequel les contraintes portent. Quant aux contraintes d'intégrité, elles seront gérées par la (ou les) classe(s) concernée(s) (s'il s'agit d'un cluster).

L'avantage essentiel lié à la gestion locale des contraintes est que, en cas de transformation du schéma conceptuel, les modifications sont localisées à la classe concernée. Or, dans un tel schéma, les modifications de contraintes seront plus fréquentes que celles des types d'entités qui représentent des objets stables. D'où, par cette technique, on espère minimiser l'impact d'éventuelles modifications.

Dans le texte qui suit, les contraintes d'intégrité seront présentées en terme de contraintes sur des classes d'objets plutôt qu'en termes de contraintes sur des types d'entités. Elles sont d'application pour toutes les classes descendantes de la classe concernée par chaque énoncé (principe d'héritage).

3.3.1. Contraintes d'intégrité sur le schéma général

Toute instance de la classe Objet Informationnel appartenant à la sous-classe Collection ne peut être directement ou indirectement impliquée dans une association Organisation avec elle-même[2].

Toute instance de la classe Objet de Rangement Structurant ne peut être directement ou indirectement impliquée dans une association Localisation-Origine ou Localisation-Réelle avec elle-même[3]

Les instances de la classe Information impliquées dans l'association Copie doivent appartenir à la même sous-classe, c'est-à-dire à l'une des sous-classes Document, Message ou Formulaire.

3.3.2. Contraintes d'intégrité relatives aux fragments

Une instance de la classe Fragment ne peut être impliquée dans une association Référence avec elle-même.

Une instance de la classe Fragment ne peut être impliquée directement ou indirectement dans une association Décomposition avec elle-même[4].

Une instance de la classe Fragment joue soit le rôle "est composant de" de l'association Décomposition, soit le rôle "compose" de l'association Composition (contrainte d'exclusion de rôles).

3.3.3. Contraintes sur les attributs

Toutes les Date-der-cons et Date-der-maj doivent être postérieures à la Date-création de l'objet de bureau.

Toute modification de la Date-der-maj d'un objet de bureau implique la modification simultanée de sa Date-der-cons.

---

[2] Si l'on considère l'association Organisation comme étant une relation définissant les arcs d'un graphe dont les sommets seraient les objets informationnels impliqués dans cette association, alors cette contrainte revient à dire que le graphe doit être sans circuits.

[3] Ici aussi les graphes dont les sommets sont les instances de la classe Objet rangeable et les relations les associations Localisation-Réelle et Localisation-Origine doivent être sans circuits.

[4] Le graphe représentant la structure hiérarchique des fragments est bien un arbre donc sans circuits.

Toute modification de la Date-der-cons-frag ou Date-der-maj-frag d'un fragment implique la modification des attributs correspondant du document auquel le fragment appartient.

Toute mise à jour de la Date-der-maj-frag d'un fragment implique la mise à jour de la Date-der-cons-frag.

Si une information est classée confidentielle alors l'attribut Mot-de-passe doit nécessairement prendre une valeur.

Si l'attribut Stockage (pour Message, Formulaire et Fragment) prend la valeur vrai alors l'attribut Contenu doit obligatoirement prendre une valeur. Dans le cas contraire, l'attribut Lieu-stockage doit prendre une valeur et l'attribut Contenu n'entre pas en ligne de compte.

4. Description des interfaces des classes d'objets du modèle d'environnement de bureau étendu

4.1. Méthode générale de construction d'une interface de classe d'objets

L'analyse effectuée jusqu'à présent a permis de développer deux éléments : le schéma conceptuel des données dans lequel les objets sont décrits sous forme de types d'entités et la structuration de ces mêmes objets sous forme d'une hiérarchie de classes d'objets.

L'étape suivante de l'analyse conceptuelle consiste à définir des ensembles regroupant les primitives d'accès au schéma des données relatives à une même classe d'objets et à les attribuer à cette classe d'objets comme méthodes de son interface. Ainsi, chaque classe d'objets du schéma de classes possède des services permettant de manipuler ses instances stockées dans la base de données. La manière dont les informations privées à une classe d'objets sont stockées (schéma conceptuel) ne doit pas être connue des utilisateurs de cette classe d'objets.

Les méthodes sont les primitives d'accès classiques, rencontrées lors de l'élaboration de tout module d'accès à une base de données. Il s'agit des primitives de consultation des données (accès direct à des entités et accès à des entités cibles via un type d'association) et des primitives de mise à jour : création, suppression et modification des données (entités et associations). La particularité est que les primitives sont vues en termes de manipulation d'instances d'une classe d'objets stockées dans le schéma des données et non en termes de manipulation d'occurrences de types d'entités dans le schéma de la base de données.

L'intérêt de calquer le schéma conceptuel des données sur le schéma de classes apparaît aussi au niveau de la définition des primitives l'accès à la base de données. De cette manière, les méthodes associées aux classes d'objets générales de la hiérarchie sont utilisées par les classes d'objets dérivées qui les enrichissent si leur définition dans la base de données est plus complète. En plus de ces méthodes héritées, les classes dérivées peuvent, évidemment, posséder des méthodes qui leurs sont propres.

Un exemple de ce mécanisme est fourni par la méthode Lire-Att, définie au niveau de la classe OB, qui permet de lire les informations relatives à cette classe. La classe O-Rgble , dérivée de la classe OB, hérite de cette méthode sans qu'aucun enrichissement ne soit nécessaire. Par contre, la classe O-Struct, dérivée de la classe O-Rgble, doit enrichir la méthode Lire-Att pour renvoyer les propriétés qui lui sont propres (Etiquette, Critère-classement, etc).

Dans cette partie, les interfaces des classes d'objets sont décrites de manière sommaire. Nous renvoyons le lecteur à l'annexe 2 "Spécification des modules d'accès à la base de données" pour obtenir des informations complètes sur les interfaces des classes d'objets, les méthodes y sont spécifiées par leur signature. En principe, la signature des méthodes est suffisante pour décrire une classe d'objets à ce stade de développement. Une spécification complète des classes de l'environnement de bureau étendu sera réalisée dès que l'architecture logique des modules sera connue. Nous aborderons alors la méthode de spécification des classes d'objets plus en détails (voir section 4.2 "Spécification formelle des méthodes").

L'étude de l'interface des classes d'objets réalisées ici a pour unique objectif de fournir au lecteur une vue d'ensemble des services offerts par chaque classe.

Les interfaces de chacune des classes du modèle d'environnement de bureau seront présentées dans leur ordre d'apparition dans la hiérarchie d'objets.

4.2. La classe OB et les classes obtenues par partitionnement

La classe OB, support de toute la hiérarchie, possède dans son interface les méthodes qui ont un sens pour tous les objets de bureau à savoir Créer, Supprimer et Accès-Direct qui permettent la création, la suppression et l'accès direct à une instance de cette classe d'objets et Lire-Nom, Lire-Att et Modifier-Att pour lire, respectivement, l'identifiant et les attributs ainsi que pour modifier ces derniers.

Les classes O-Rgt et O-Rgble héritent de toutes ces méthodes. A celles-ci viennent s'ajouter les méthodes relatives à la création et à la consultation des associations Localisation-Réelle et Localisation-Origine définies sur les types d'entités matérialisant ces objets dans la base de données. Ainsi, la classe O-Rgt possède des méthodes permettant de lire séquentiellement les objets rangeables que ses instances contiennent : Premier-O-Rgble-Loc-Réelle, O-Rgble-Loc-Réelle-Suivant et Premier-O-Rgble-Loc-Origine, O-Rgble-Loc-Origine-Suivant. Elles sont complétées par deux méthodes,Tester-Localisation-Réelle et Tester-Localisation-Origine, servant à tester si les objets de rangement ne sont le lieu de rangement d'aucun objet rangeable (connectivité minimale égale à zéro).

L'interface de la classe O-Rgble comprend, en sus des méthodes héritées, deux méthodes de consultation, destinées à accéder à l'objet de rangement qui les contient : Loc-Origine et Loc-Réelle et trois méthodes de modification relatives aux déplacements des objets rangeables dans l'environnement : Déplacer (modification du type d'association Localisation-Réelle), Changer-

Origine (modification du type d'association Localisation-Origine) et Replacer (remettre un objet à sa place d'origine).

Des liens sémantiques entre objets rangeables et objets de rangement sont visibles au niveau de leur interface. Rappelons que ces liens entre les deux classes étaient déjà apparus lors de la définition du schéma conceptuel des données.

L'interface de la classe O-Non-Rgble est entièrement définie, sans enrichissement, par l'héritage des méthodes provenant de la classe OB.

La classe O-Info est enrichie au niveau de son information par l'ajout de la propriété Mots-clés fournissant l'ensemble des mots clés associés à chaque instance de la classe. Cette propriété est accessible et modifiable par l'utilisateur grâce aux méthodes publiques suivantes : Ajouter-Mot-Clé et Supprimer-Mot-Clé pour les mises à jour et Lister-Mots-Clés pour les consultations.

4.3. Les classes obtenues par héritage multiple

Un niveau plus bas dans la hiérarchie des objets se trouvent les classes Info, Collection, O-Struct et O-Rgt-Term. Ces classes possèdent déjà une interface riche grâce aux classes de base sur lesquelles elles sont fondées par le mécanisme d'héritage multiple. Les classes Info et Collection possèdent toutes les méthodes permettant de manipuler des objets informationnels rangeables, la classe O-Struct toutes les méthodes relatives à la manipulation des objets de rangement rangeables et la classe O-Rgt-Term celles relatives à la manipulation des objets de rangement non rangeables. L'interface de chacune de ces classes est complétée par des méthodes spécifiques, décrites ci-dessus.

Si on l'analyse sous l'angle du schéma conceptuel des données, la classe Info est enrichie par des attributs définis dans le type d'entité Info et par le type d'association récursive Copie. Au niveau interface de l'objet, les nouvelles opérations définies sont la méthode Copier qui permet de dupliquer des informations et les méthodes Lire-Première-Copie et Lire-Copie-Suivante ajoutées pour réaliser la consultation des copies d'informations. Finalement, la méthode A-Pour-Original sert à obtenir l'information original d'une information copiée.

Les méthodes Emprunter et Restituer sont relatives à la gestion des informations qui peuvent sortir de l'environnement de bureau.

L'interface de la classe Info reprend aussi des méthodes de mise à jour des propriétés importantes, propres à la classe Information à savoir Changer-Conf et Changer-Mot-De-Passe qui

servent à modifier la confidentialité et le mot de passe de l'information. Les autres propriétés seront manipulables grâce aux méthodes Lire-Att et Modifier-Att, enrichies à ce niveau.

Deux méthodes de consultation Mot-De-Passe-Valide et Présence permettent à l'utilisateur de savoir si l'accès au contenu de l'information est possible (l'utilisateur est bien détenteur du bon mot de passe) et si l'information à consulter est présente dans l'environnement.

La classe Collection est enrichie par les propriétés Critère-classement et Ordre-classement. Elles sont modifiables grâce à la méthode Changer-Classement. Les autres méthodes de l'interface de la classe Collection concernent l'organisation des objets informationnels en collections, c'est-à-dire les propriétés de la classe présentées dans le schéma conceptuel par l'association Organisation :

- Vide teste si la collection ne contient aucun objet informationnel,
- Ajouter-Composant et Supprimer-Composant sont destinées à ajouter et à supprimer un objet informationnel d'une collection,
- Premier-Composant et Composant-Suivant fournissent, en combinaison, la liste des objets informationnels contenus dans une collection.

La troisième classe définie par héritage multiple est la classe O-Struct. Elle est complétée par les propriétés Critère-regroup, Ordre-regroup et Etiquette. Les méthodes Etiqueter, Lire-Etiquette et Changer-Regroupement permettent de consulter et de modifier ces propriétés.

La classe O-Struct hérite des méthodes propres à la classe d'objets O-Rgble. Cependant, comme les O-Struct sont aussi des objets de rangement, toutes les méthodes héritées des objets rangeables devront être redéfinies de manière à pouvoir déplacer, replacer ou de changer la localisation de ces objets sans risquer de les inclure dans eux-même soit directement, soit indirectement, via un autre objet de rangement structurant.

Finalement, la classe O-Rgt-Term hérite telle quelle de l'interface de ses classes de base, les classes O-Rgt et O-Non-Rgble.

4.4. Les classes d'objets terminales dans le modèle

Examinons maintenant les interfaces des classes d'objets terminales dans la hiérarchie (c'est-à-dire celles ne possédant pas de classes d'objets dérivées).

La classe O-Rgt-Term et ses descendantes directes ne sont pas enrichies en méthodes mais leur définition reste nécessaire pour le cas ou l'utilisateur voudrait étendre ces classes.

Pour les classes dérivées de la classe Collection un seul changement est à constater. Au niveau de la classe Fichier, la propriété Type-composant, privée à l'objet, est utilisée pour restreindre la méthode Ajouter-Composant à la seule insertion d'objets informationnels possédant le bon type (celui stocké dans la base de données pour l'instance concernée par l'application de la méthode). Au niveau de la classe Dossier aucun enrichissement n'est nécessaire.

Les interfaces des classes Formulaire et Message sont complétées par la méthode Maj-Contenu qui , comme son nom l'indique, permet de modifier le contenu d'un formulaire ou d'un message si celui-ci est stocké dans la base de données. Pour la classe Document, aucun enrichissement n'est nécessaire car cette dernière méthode est définie au niveau de la classe Fragment.

4.5. La classe autonome Fragment

Pour compléter la description des interfaces des classes du schéma de classes d'un environnement de bureau étendu, il reste à décrire l'interface de la classe Fragment. Ses méthodes peuvent être regroupées en trois catégories :

- celles qui agissent sur un fragment, par exemple en modifiant ou en observant ses propriétés. Elles ont une action locale,
- celles qui fournissent des informations sur l'arbre des fragments ou qui permettent de le modifier. Elles ont un sens global au niveau de la structure du document,
- celles qui observent et modifient les références entre des fragments. Elles permettent de lier des documents ou des parties de documents entre eux.

La première méthode de l'interface de la classe d'objets Fragment est la méthode Existe. Elle permet de savoir si un fragment donné est présent dans la base de données. L'identifiant est composé puisqu'il est formé du nom du fragment et du nom de son document propriétaire.

Les méthodes de la première catégorie, c'est-à-dire celles agissant sur un fragment particulier, sont principalement relatives à la consultation et la modification des propriétés du fragment. Les consultations et modifications sont réalisées par l'intermédiaire des méthodes Modifier-Att et Lire-Att. A celles-ci viennent s'ajouter deux méthodes plus ciblées : Localiser-Stockage qui permet de savoir si le contenu d'un fragment est stocké dans la base de données et, si tel est le cas, la méthode Activer-Processeur peut être utilisée pour consulter et mettre à jour son contenu.

Les méthodes ayant un sens global au niveau de la structure du document sont de deux types. Les unes permettent de modifier l'arbre des fragments. Il s'agit de Créer qui permet

d'ajouter un nouveau fragment à l'arbre, de Supprimer qui détruit un fragment ainsi que toute l'arborescence dont ce fragment est la racine et de Changer qui déplace un fragment et toute sa sous-arborescence d'un endroit à un autre de l'arbre. Les autres méthodes de la classe Fragment sont relatives à la consultation de l'arbre. La figure 3.15 fournit une illustration graphique de l'effet de chacune de ces méthodes. Par rapport à un fragment courant, l'action de Père, Fils, Précédent, Suivant et Lister-Sous-Arbre est de renvoyer le ou les fragments encadré(s) en vis-à-vis de leur nom.

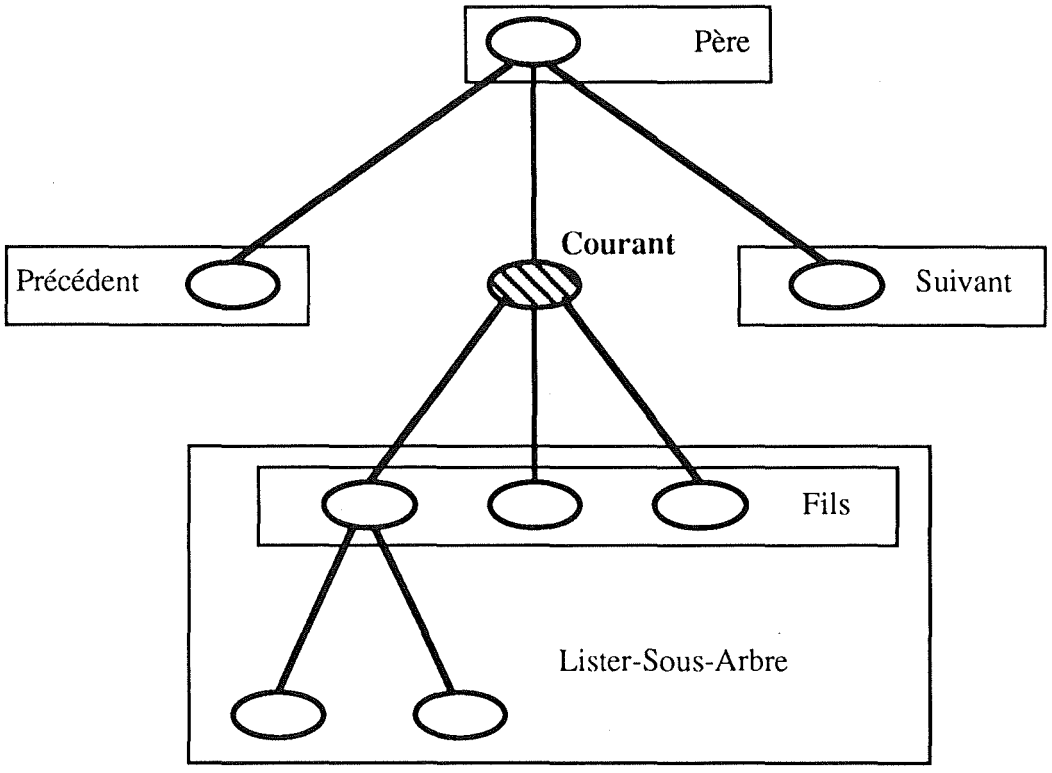

Figure 3.15 : Illustration graphique de l'effet des méthodes de consultation relatives à la structure du document

Les dernières méthodes de l'interface de la classe Fragment sont celles relatives aux références. Au niveau du schéma conceptuel des données, ces méthodes ont un effet sur le type d'association Référenciation. La méthode Ajouter-Référence permet d'ajouter, pour une instance de Fragment donnée, une référence vers un autre fragment tandis que Supprimer-Référence a l'effet inverse. Finalement, la méthode Références fournit l'ensemble des fragments référencés par l'instance pour laquelle la méthode est activée.

5. Réutilisabilité de la hiérarchie étendue d'objets de bureau

Le modèle d'environnement de bureau développé ci-dessus veut respecter les principes fondamentaux pour lesquels la conception orientée objets est reconnue. Le modèle a donc été construit en ne perdant pas de vue les principes d'extensibilité et de réutilisabilité. Dans un modèle à objets, une des façons de dégager les modules du système est de considérer les classes d'objets elles-mêmes. L'introduction de nouvelles classes et les changements de spécifications doivent, en principe, n'avoir d'effets que sur un nombre limité de classes, une seule idéalement.

Pour ce faire une idée de la valeur du modèle construit, il serait intéressant d'introduire de nouveaux objets et de voir quel est le prix de leur inclusion dans le modèle existant. Si ces objets s'intègrent directement dans la classification, leur introduction se fera en tirant au maximum profit des propriétés des classes prédéfinies. C'est un gage d'extensibilité et de réutilisabilité.

Deux types d'extensions sont envisageables : celles qui introduisent de nouveaux objets et celles qui définissent des restrictions sur des classes existantes.

5.1. Prise en compte de restrictions sur les classes du modèle

5.1.1. Exemples introductifs

Un exemple de restriction est la réintroduction des contraintes du modèle d'environnement classique éliminées lors de la construction du modèle étendu (voir section 1.1 "Présentation d'un environnement de bureau classique"). Prenons la contrainte de capacité de contenance maximale des objets de rangement terminaux. Pour la réintroduire, il suffit de créer une nouvelle classe d'objets fondée sur la classe O-Rgt-Term. Elle hérite directement de toutes les méthodes non concernées par l'introduction de cette nouvelle contrainte. Les autres méthodes sont redéfinies par cette nouvelle classe en introduisant les filtres nécessaires sur les méthodes générales imposant ainsi, le respect des contraintes. Le schéma conceptuel des données ne subit aucune modification puisque les propriétés de la classe de base ne sont pas enrichies, seule son interface est modifiée.

De même, un utilisateur peut trouver la définition de la classe Collection trop générale. Par exemple, il aimerait imposer que les objets informationnels ne fassent partie que d'une seule collection, forcer le lieu de rangement physique de tous les composants d'une collection à la même valeur, interdire la définition de piles de fichiers, etc. Pour réaliser ces modifications, il lui suffit de

créer une classe dérivée de la classe Collection dans laquelle il spécifie les restrictions sur l'utilisation des méthodes qui ne respectent pas les contraintes qu'il veut gérer.

5.1.2. Démarche théorique d'extension

En toute généralité, les modifications relatives à des restrictions sur des classes d'objets existantes ne modifient pas le schéma conceptuel des données. Pour les réaliser, il suffit d'introduire de nouvelles classes d'objets dans le schéma de classes.

Comme ces classes sont souvent des classes dérivées d'objets terminaux de la hiérarchie, la réutilisabilité du modèle est grande. Ainsi, un grand nombre de méthodes d'accès à la base de données seront héritées telles quelles (voir figure 3.15). D'autres méthodes, affectées par les restrictions à apporter, sont redéfinies moyennant l'introduction d'un filtre sur les méthodes héritées. Dans ce cas, la réutilisabilité est aussi présente car le créateur des nouvelles classes utilise les méthodes existantes pour exprimer les restrictions. Il ne travaille pas directement sur le schéma conceptuel des données. Finalement, il peut exister des méthodes nouvelles, non prévues dans le modèle initial. Pour ces dernières, il sera éventuellement nécessaire de réaliser des accès au schéma conceptuel des données sans passer par les méthodes existantes. Il va de soi que cette dernière catégorie de modifications est la plus difficile à apporter.

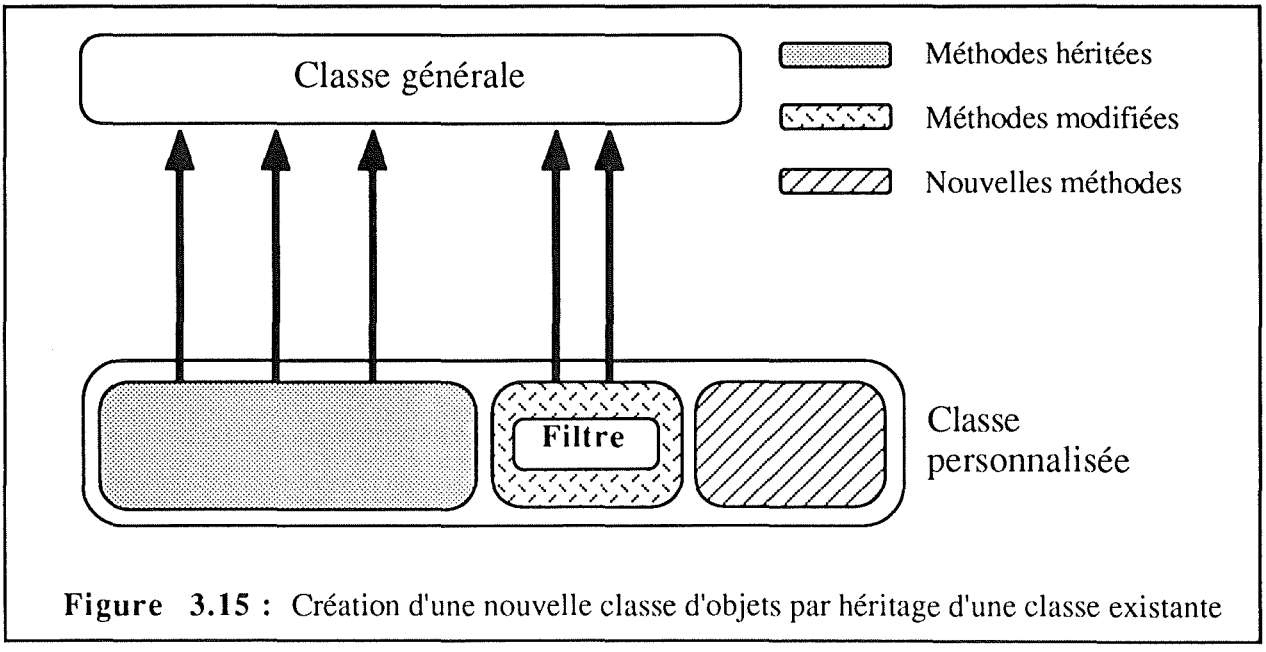

Figure 3.15 : Création d'une nouvelle classe d'objets par héritage d'une classe existante

Mis à part ce dernier type d'enrichissement, il reste que l'introduction d'une nouvelle classe se fait facilement et si la classe de base qui la supporte est choisie judicieusement, la réutilisabilité du schéma initial est grande.

5.2. Introduction de nouveaux objets

Après avoir présenté ce premier type d'extension, nous abordons maintenant l'introduction de nouveaux objets dans le modèle. La prise en compte d'un nouvel objet peut impliquer des modifications très diverses suivant la nature de l'objet considéré. Dans de telles circonstances, il est très difficile de fournir une démarche générale d'extension. Nous nous contenterons donc d'aborder le problème au travers d'exemples.

5.2.1. La photocopieuse

Imaginons le pire des cas, l'introduction d'un objet n'ayant en apparence aucun point commun avec les classes d'objets existantes. Malgré tout, il donnera naissance à une classe dérivée de la classe Objet de Bureau. C'est par exemple le cas de la photocopieuse. Son introduction nécessitera la création d'une nouvelle classe d'objets nommée Objet de Traitement. Dans ce cas précis, la réutilisabilité du modèle, bien qu'elle existe, est réduite à l'utilisation de la classe OB.

Si par abstraction, l'aspect de traitement pur est écarté et que seuls les critères de partitionnement "rangeable" et "porteur d'information" sont retenus, alors la photocopieuse à la même nature qu'une table : c'est un objet de rangement terminal qui accueille temporairement des informations en vue d'un traitement. La photocopieuse peut alors être intégrée dans le modèle par une nouvelle classe dérivée de la classe O-Rgt-Term (voir figure 3.16). Elle hérite de toutes ses propriétés et méthodes et les enrichit. Par exemple, une méthode telle que Dupliquer-Document pourrait faire partie de son interface.

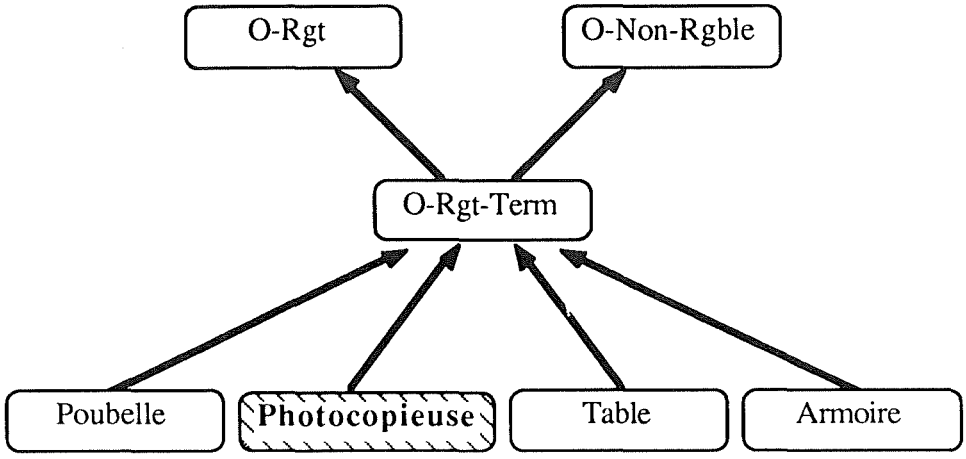

Figure 3.16 : Insertion de la classe Photocopieuse dans le modèle d'environnement de bureau étendu

Il apparaît que suivant le point de vue adopté, l'introduction de la classe Photocopieuse peut se faire à différents endroits dans la hiérarchie. La deuxième solution est ici préférable car elle tire mieux profit de la hiérarchie existante.

5.2.2. Les micro-fiches

Un autre exemple d'extension possible du modèle est l'introduction des micro-fiches, informations dont le support est un film.

Des informations de tout type sont susceptibles d'être photographiées, la micro-fiche est donc intégrable dans le modèle en tant que classe dérivée de la classe Information pour laquelle l'attribut Support serait forcé à la valeur "film". Aucun enrichissement des attributs de la classe Information n'étant nécessaire, le schéma E/A n'est pas modifié. Cette nouvelle classe, appelée Micro-Fiche, hérite de toutes les méthodes propres à la classe Information en particulier la copie et l'organisation des objets informationnels en collections.

L'apparition de la classe Micro-Fiche nécessite l'introduction de filtres sur les méthodes générales Ajouter-Composant et Supprimer-Composant de la classe Collection. Ceci est possible par la création d'une classe dérivée de la classe Collection, la classe Collection-MF, qui modifie les méthodes selon le principe général décrit ci-dessus. La classe Collection-MF hérite de toutes les autres méthodes de la classe Collection sans qu'aucune modification ne soit nécessaire. La figure 3.17 apporte une illustration de la modification réalisée sur le schéma de classes initial.

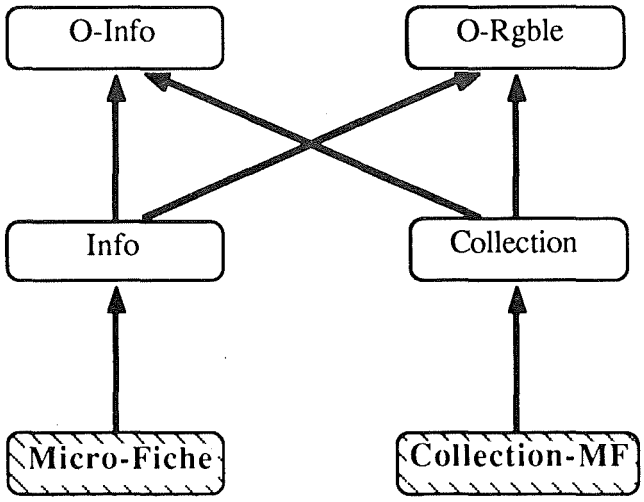

Figure 3.17 : Nouvelles classes résultant de l'introduction des micro-fiches dans le modèle d'environnement de bureau étendu

Pour la classe Micro-fiche, il faut aussi définir un filtre qui force le lieu de rangement physique à la même valeur pour tous les composants d'une collection-MF (car les fiches sont sur le même film). Une redéfinition de certaines des méthodes héritées de la classe Objet Rangeable s'impose pour tenir compte de cette nouvelle contrainte, elle sera réalisée au niveau de la classe Micro-Fiche.

5.2.3. L'outil informatique

Le dernier exemple d'enrichissement du schéma de classes de l'environnement de bureau étendu portera sur l'introduction de l'outil informatique dans le modèle. L'exemple restera assez sommaire et l'introduction envisagée n'est qu'une possibilité parmi d'autres, celle qui nous a semblé tirer le mieux profit du modèle d'environnement de bureau étendu. Le statut attribué à chacun des objets introduits peut évidemment être discuté.

Les fichiers informatiques (fichiers d'enregistrements) sont intégrables dans la hiérarchie en tant que fichiers de formulaires. Ceci ne signifie pas que chaque enregistrement du fichier doit être décrit, on peut ne s'intéresser qu'à la localisation du fichier dans l'environnement, par exemple.le disque sur lequel il est situé.

Les disques optiques, les bandes magnétiques et les disquettes sont des objets destinés au rangement des fichiers informatiques. La manière la plus simple de les intégrer est d'introduire la classe O-Struct-Infor, dérivée de la classe O-Struct. Ces objets peuvent en effet être rangés dans d'autres objets comme des armoires, boîtes, etc et servent eux-mêmes de lieu de rangement pour les

fichiers. Si la capacité maximale de contenance des objets de la classe O-Struct-Infor doit être prise en compte, ceci se fait par la redéfinition des méthodes de cette nouvelle classe dérivée (comme décrit précédemment). Comme les objets de la classe O-Struct-Infor n'ont pas tous la même capacité de contenance si, pour un objet donné, la valeur prise par celle-ci doit être conservée, il faut la stocker dans la base de données. Dans ce cas précis, le schéma conceptuel des données devra subir des modifications. Pour les O-Struct-Infor, le schéma conceptuel deviendrait celui présenté à la figure 3.18. Le nouveau type d'entité O-Struct-Infor possède pour attribut Capacité. Il est associé à son type d'entité générique par une relation is-a.

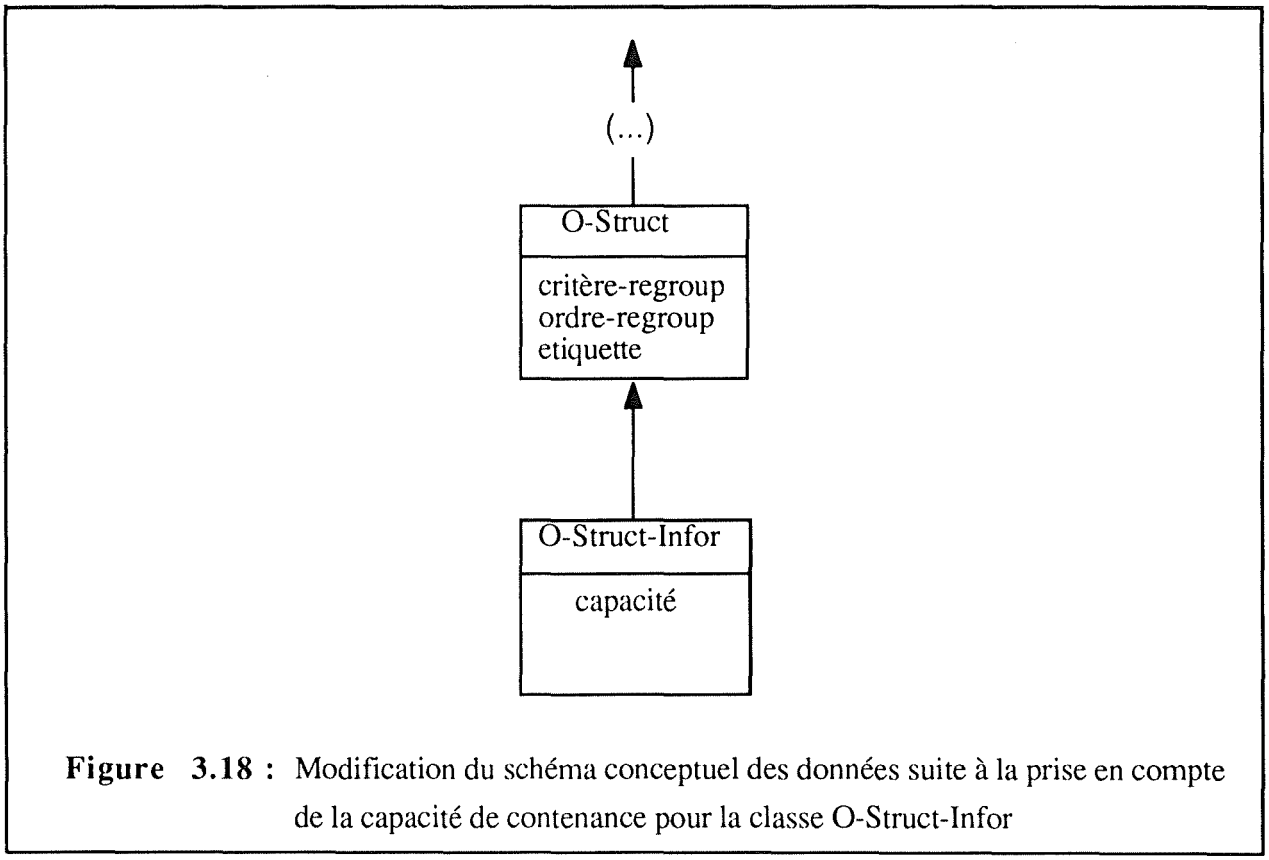

Figure 3.18 : Modification du schéma conceptuel des données suite à la prise en compte de la capacité de contenance pour la classe O-Struct-Infor

De manière générale, dès que la création d'une nouvelle classe d'objets nécessite l'introduction de nouvelles propriétés et donc de nouveaux attributs pour pouvoir conserver une trace de leurs valeurs, le schéma conceptuel des données est modifié par introduction de nouveaux types d'entités (et non pas de nouveaux attributs dans un type d'entité existant car plusieurs classes d'objets risqueraient d'être affectées par des modifications).

Les objets de la classe O-Struct-Infor ne pourront évidemment être utilisés comme objets de rangement que pour des fichiers informatiques. Cette restriction s'exprime, comme précédemment, par l'introduction de filtres sur les méthodes générales de l'interface de O-Struct.

Les lecteurs de disques et de disquettes, les dérouleurs de bandes sont des objets de rangement terminaux puisqu'ils accueillent temporairement les objets décrits ci-dessus dans un but précis : accès à certaines informations en vue de leur traitement. Leur prise en considération dans le modèle peut se faire, comme pour la photocopieuse, par l'introduction d'une classe dérivée de la classe O-Rgt-Term, la classe O-Term-Infor.

Les bandes magnétiques, les cassettes ou les disques sont eux-mêmes rangés dans des armoires spécialement conçues : les armoires à disques ou à bandes et les chargeurs. Ces derniers sont compartimentables. Dès lors, ils s'intègrent parfaitement dans le modèle initial en tant qu'objets de la classe Armoire-Infor, dérivée de la classe Armoire.

Des filtres sur les méthodes générales sont définis au niveau des méthodes des interfaces des classes O-Term-Infor et Armoire-Infor. Ils concernent notamment le type des objets rangeables pouvant être accueillis par ces objets de rangement.

Le schéma final, résultant de l'introduction de l'outil informatique dans le modèle, est présenté à la figure 3.19. La partie du modèle initial, non représentée dans la figure, n'est pas modifiée par l'introduction de l'outil informatique.

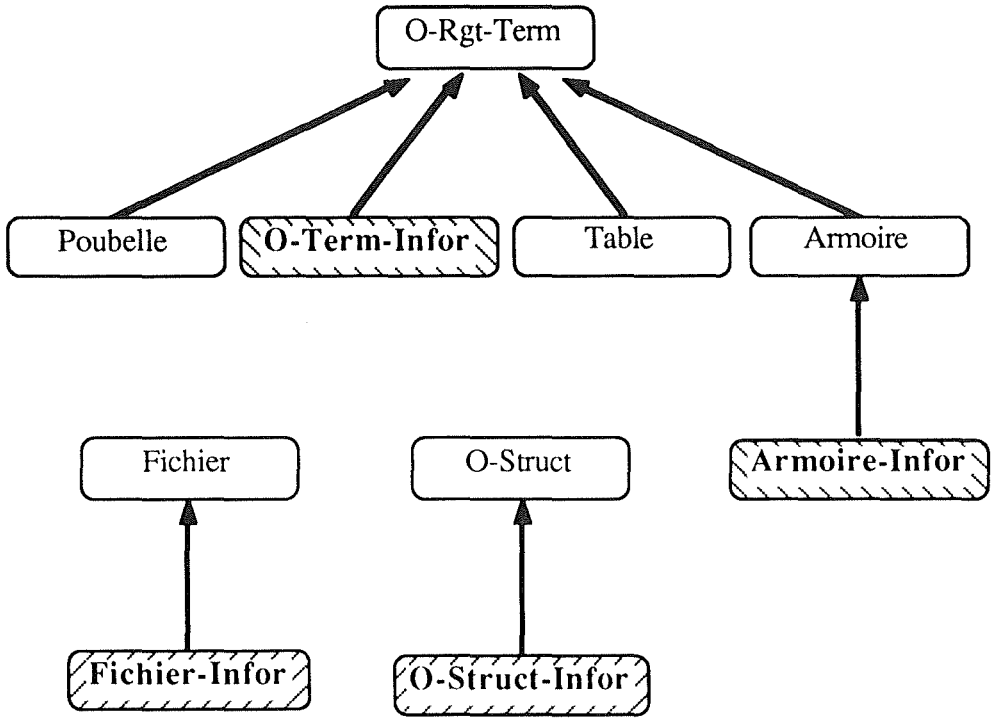

Figure 3.19 : Nouvelles classes résultant de l'introduction de l'outil informatique dans le modèle

5.2.4. En conclusion

L'exemple de l'intégration de l'outil informatique, plus complet que les précédents, montre que l'introduction de nouveaux objets dans la hiérarchie des objets de bureau peut faire apparaître plusieurs classes d'objets. Une étude attentive de l'endroit d'introduction des nouveaux objets dans le modèle d'environnement de bureau étendu doit être réalisée de manière à tirer, au maximum, profit de la hiérarchie existante.

En particulier, lors de la prise en compte d'un nouvel objet, il ne faut pas essayer de l'intégrer en recourant à une seule classe d'objets. Souvent, plusieurs classes d'objets fournissent un meilleur support et augmentent la réutilisabilité du modèle de base.

Pour les différents exemples envisagés, l'introduction des objets s'est toujours traduite par la création de nouvelles classes d'objets terminales dans la hiérarchie. En fait, plusieurs modèles d'environnement de bureau ont été successivement construits jusqu'à ce qu'une telle introduction soit possible pour tous les exemples envisagés dans cette partie. Grâce à la technique de validation théorique introduite au cours de la phase d'analyse, le schéma finalement obtenu est très

général C'est ainsi que le schéma final résulte de plusieurs abstractions successives réalisées dans le but de fournir des classes suffisamment générales pour supporter les extensions présentées. Même si le prix à payer est parfois élevé, il reste bien inférieur à l'effort que l'introduction de nouveaux objets de bureau aurait nécessité dans un modèle non orienté objets. Il apparaît donc que la validation théorique qui consiste à introduire de nouveaux objets dans le modèle est un bon test qui tend à améliorer fortement la qualité finale de la solution produite.

Chapitre 4 :
Conception logique de la boîte à outils d'objets bureautiques

Introduction

Une fois l'analyse conceptuelle terminée, nous pouvons passer à la première partie de la phase de conception : la conception logique.

Dans un premier temps, nous construirons une **architecture des modules d'accès à la base de données**. Dans cette partie, nous verrons comment le prototypage a influencé la construction de cette architecture. Les décisions de conception prises dans cette partie ont pour principal but d'améliorer les performances de la boîte à outils.

Cette partie sera suivie par la **spécification formelle des modules**. Dans une approche orientée objets, les spécifications évoluent dynamiquement au cours du cycle de vie du développement. Nous avons donc attendu qu'elles soient fixées pour les présenter.

Le schéma E/A étendu est indépendant d'un S.G.B.D. cible. Pour produire un schéma conforme au S.G.B.D. cible N.D.B.S., il faut passer par plusieurs phases de transformation successives. Les trois premières phases de cette transformation prennent place au niveau logique car la solution finalement obtenue doit être exécutable sur un S.G.B.D. abstrait Les transformations feront l'objet des trois dernières sections de ce chapitre. La première section "**Construction d'un schéma E/A de base**" abordera l'élimination de toutes les structures d'héritage présentes dans le schéma étendu. En effet, le schéma E/A initial est orienté objets. Comme nous allons travailler avec un S.G.B.D. traditionnel, la première phase de transformation consiste à éliminer du schéma toute structure liée à la méthodologie de développement orientée objets choisie, ce qui est le cas des structures de généralisation/spécialisation du schéma E/A. Ensuite, la **conception logique du schéma de base de données** proprement dit prendra place, elle portera sur la production des schémas des accès possibles et des accès nécessaires.

Comme les techniques de transformation de schémas relatives aux structures de généralisation/spécialisation sont encore mal connues. Nous terminerons ce chapitre par **un essai de définition d'une méthodologie de transformation de schéma**. Cette section aura pour unique but de fournir au lecteur désireux d'employer les techniques de transformation quelques conseils pratiques sur leur utilisation. Il nous a en effet semblé intéressant de tirer quelques enseignements généraux du cas que nous avons eu à traiter.

1. Architecture des modules d'accès à la base de données

1.1. Apports de l'approche orientée objets

Rappelons que la construction de l'architecture logique des modules consiste à produire une solution qui soit exécutable sur une machine abstraite donc, indépendante d'un S.G.B.D. cible et d'un environnement (système, langage de programmation, etc) cible [HAIN-86].

Dans une approche orientée objets, une possibilité de dégager les modules d'accès à la base de données est de baser la construction de l'architecture des modules sur le schéma de la hiérarchie de classes d'objets. Dans cette architecture, les classes d'objets représenteraient les modules et les structures d'héritage définies sur ces classes, les liaisons entre les modules.

L'architecture obtenue par cette technique a toutes les chances de respecter les principes élémentaires inhérents à toute bonne architecture orientée objets : modularité, faible couplage, forte cohésion, etc.

Comme nous l'avons montré au chapitre deux, ces principes sont naturels pour une architecture orientée objets correctement développée. De plus, lors de la phase d'analyse, l'introduction de nouveaux objets dans le modèle (voir section 3.5 "Réutilisabilité de la hiérarchie étendue d'objets de bureau") a permis de prouver que la hiérarchie d'objets de bureau était extensible et réutilisable. Les modules obtenus de cette manière ont donc toutes les chances de l'être également.

Cependant, bien que ces concepts soient importants, ils ne suffissent pas pour constituer une architecture qui sera source d'une boîte à outils de qualité. Il faut aussi considérer les critères de performance et la facilité d'emploi de la boîte à outils pour l'utilisateur.

L'architecture fournie par dérivation du schéma des classes d'objets constituera donc un modèle de base que nous essayerons d'améliorer. Les améliorations apportées prendront place à deux moments.

Premièrement, avant implémentation, des choix de conception élémentaires, relatifs à l'amélioration des performances d'accès seront réalisés. Ils concerneront la gestion de l'accès aux objets stockés dans la base de données. Ces choix auront surtout un impact sur la partie privée des objets. Ils feront l'objet de la seconde partie de cette section : **Gestion en mémoire centrale des objets**.

Deuxièmement, suite à l'implémentation d'un premier sous-système, la conception par prototypage nous amènera à revoir l'architecture de la boîte à outils de manière à prendre en considération le problème des mises à jour simultanées sur les objets. Cette révision résultera en la duplication de la hiérarchie des modules et l'introduction d'une table résidente en mémoire centrale gérée par un contrôleur d'accès aux objets. Les modifications apportées concerneront aussi l'introduction de la notion de cycle de vie d'un objet. Nous verrons comment il facilite le travail du programmeur d'application. Les transformations de la hiérarchie des modules feront l'objet des troisième et quatrième parties de cette section : **Cycle de vie d'un objet** et **Duplication de la hiérarchie : contrôle d'accès aux objets**.

1.2. Gestion en mémoire centrale des objets

Si l'on travaille directement sur la base de données, toute consultation des caractéristiques d'une instance d'une classe d'objets ou toute modification de celle-ci se traduit par de nombreux accès logiques à la base de données.

Par exemple, si la méthode Lire-Att est utilisée pour lire les informations relatives à une instance de la classe Info, il faut accéder à cet objet. De même, si la méthode Modifier-Att est utilisée, cet accès doit à nouveau être réalisé. Pire, si la localisation d'origine de cette information doit être obtenue, il faut, en plus de l'accès logique, accéder à l'objet rangeable correspondant (puisque toute information est un objet rangeable), puis tester la valeur de l'objet de rangement cible de l'association Localisation-Origine. Cet état de fait n'entraîne évidemment une grande efficacité.

De ces quelques considérations résulte une première décision de conception :

> Toute instance d'une classe d'objets utilisée par un programme d'application sera chargée en mémoire centrale.

Ainsi, une fois le chargement réalisé, les accès suivants se feront en mémoire centrale. De cette manière, le programme d'application (P.A.) ne travaille pas continuellement sur la base de données. Cette manière de procéder permet un accroissement des performances du logiciel assez appréciable.

Ce choix de conception implique qu'une structure de données adéquate, permettant de stocker toutes les informations relatives à un objet doit être trouvée. Nous aborderons ce sujet plus en détails lors de la spécification complète des objets (voir section suivante, 4.2 "Spécification formelle des classes d'objets").

Un P.A. commence l'utilisation d'une variable objet par le chargement de cet objet en mémoire centrale et la termine en la recopiant dans la base de données. Ceci implique l'apparition d'une nouvelle méthode dans l'interface des objets, la méthode Fin-Utilisation qui assure la mise à jour dans la base de données de l'instance de la classe d'objets correspondant à la variable objet lorsque le P.A. n'a plus besoin de cette instance.

Le chargement des objets en mémoire centrale est réalisé de deux manières, soit par la création d'une nouvelle instance en utilisant la méthode Créer, soit par l'accès-direct à une instance déjà stockée dans la base de données grâce à la méthode Accès-Direct (voit figure 4.1).

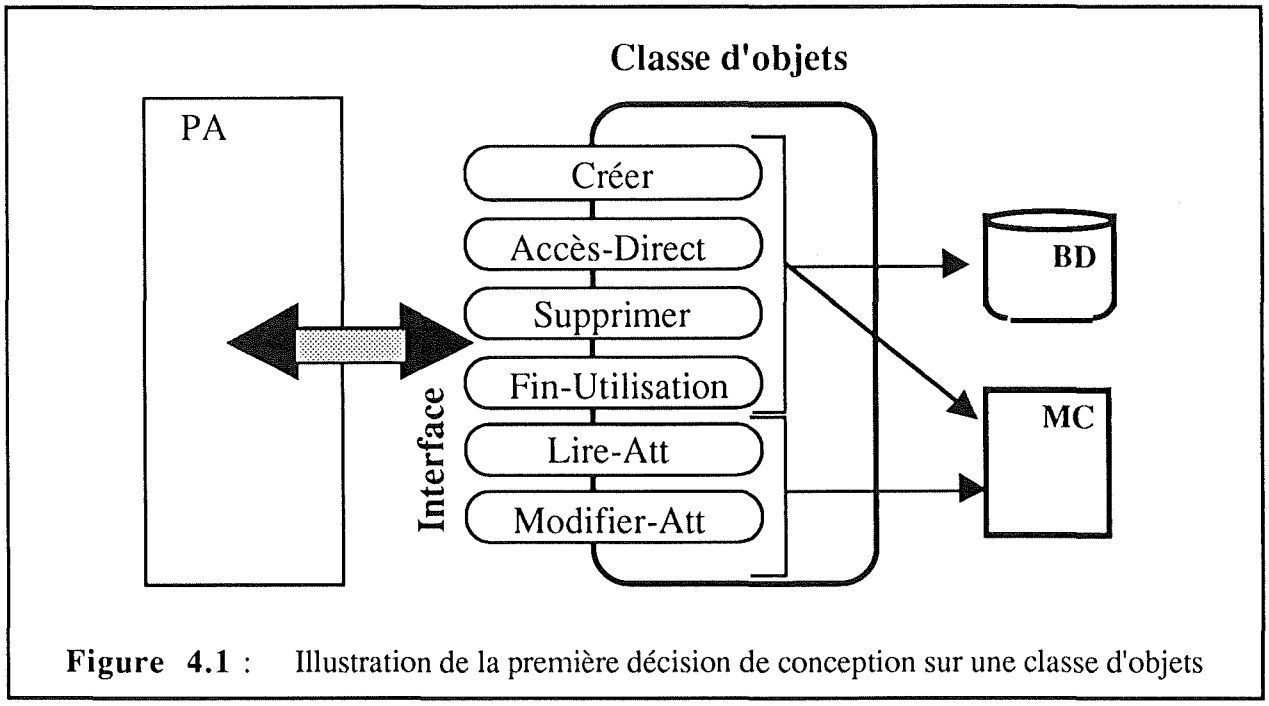

Figure 4.1 : Illustration de la première décision de conception sur une classe d'objets

La mise à jour de la base de données suite aux modifications réalisées sur les objets chargés en mémoire centrale se fait en utilisant la méthode Fin-Utilisation. Il est à noter que la méthode Supprimer a un effet similaire, elle agit en effet sur la base de données en ôtant toute trace de l'instance de la classe d'objets concernée par la suppression.

Seules ces quatre méthodes travailleront directement sur la base de données. Les autres agissent sur les objets chargés en mémoire centrale (voir figure 4.1). Grâce à la diminution du nombre d'accès logiques réalisés au schéma des données, les performances de la boîte à outils sont améliorées.

Du point de vue de l'utilisateur, la seule modification visible est l'apparition de la méthode Fin-Utilisation qui vient s'ajouter aux méthodes précédemment définies dans l'interface des classes

d'objets décrites à la section 3.4 "Description des interfaces des classes d'objets du modèle d'environnement de bureau étendu".

1.3. Cycle de vie d'un objet

La variable objet manipulée par le P.A. peut être analysée en termes de cycle de vie. Celui-ci débute avec le chargement de l'objet depuis la base de données vers la mémoire centrale et prend fin avec la recopie de l'objet de la mémoire centrale dans la base de données. Le mécanisme régissant le cycle de vie d'une variable objet et les méthodes qui y sont associées sont présentés à la figure 4.2.

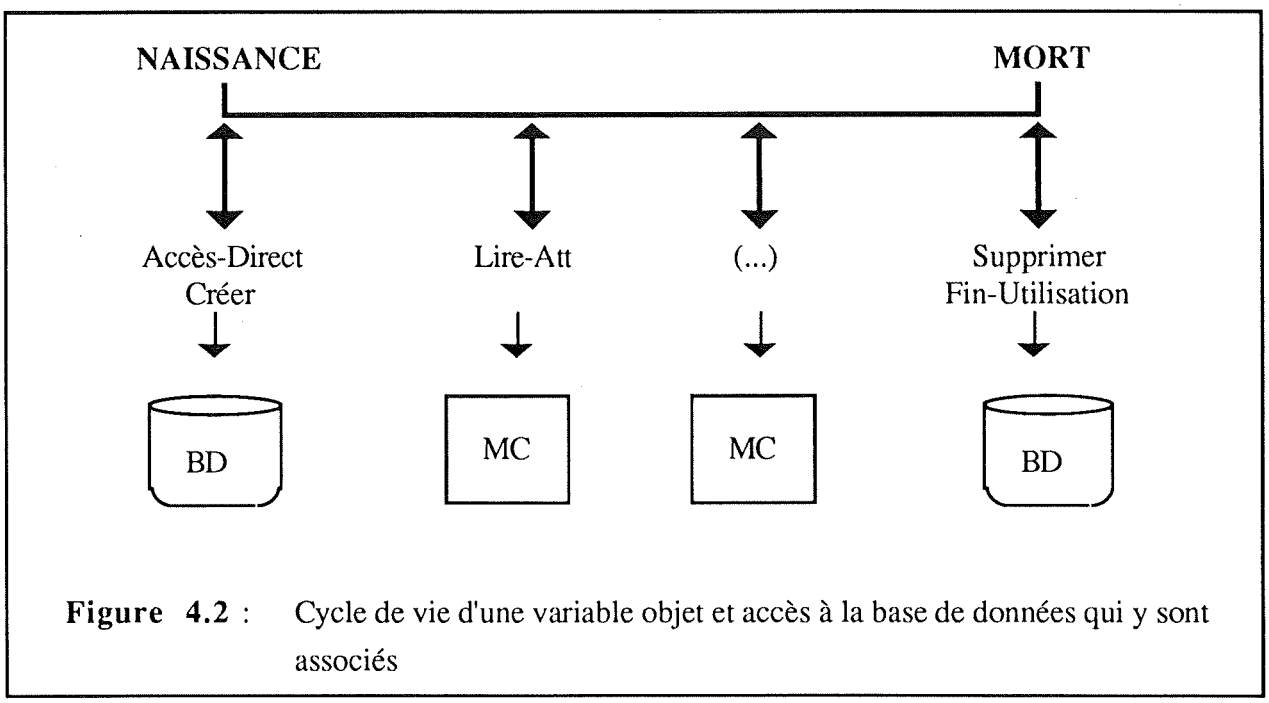

Figure 4.2 : Cycle de vie d'une variable objet et accès à la base de données qui y sont associés

Mais cette extension de la variable objet ne suffit pas. Il apparaît suite au premier prototypage qu'elle est de peu d'utilité. En effet, une variable du P.A. doit pouvoir successivement désigner plusieurs objets de bureau pour pouvoir être utilisée en toute généralité. C'est par exemple le cas lors de l'écriture d'une boucle dans un programme devant permettre d'accéder à tout les objets informationnels contenus dans une collection, dans un objet de rangement, etc.

Cette constatation résulte en un deuxième choix de conception, énoncé ci-dessous :

Une variable objet manipulée par le P.A. peut, en suivant le mécanisme du cycle de vie, désigner successivement plusieurs objets de bureau.

La délimitation de la durée de vie de la variable objet du P.A. se réalise grâce à deux nouvelles méthodes qui s'appellent Init et Fini. Durant sa vie, cette variable peut être liée à plusieurs cycles de vie d'objets de bureau (voir figure 4.3). Les variables déclarées dans le P.A. et utilisées selon le mécanisme du cycle de vie permettent de construire des algorithmes de tout type.

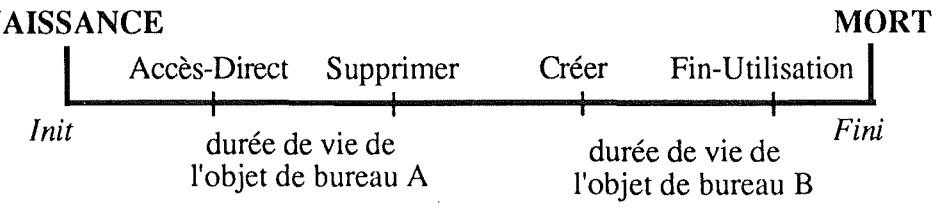

Figure 4.3 : Mécanisme du cycle de vie des objets de bureau tel qu'il est vu par le P.A.

1.3. Duplication de la hiérarchie : contrôle d'accès aux objets

1.3.1. Problèmes rencontrés lors du prototypage

Suite à l'implémentation du premier prototype, il est apparu que le cycle de vie des objets chargés en mémoire centrale pouvait conduire à des problèmes de cohérence de la base de données dans le cas où une même instance d'objets était utilisée plusieurs fois simultanément par le programme d'application.

La figure 4.4 illustre les deux types de problèmes qui peuvent survenir :

- si un programme d'application (ici, P.A.2) doit accéder à un objet de bureau alors que celui-ci est déjà utilisé et modifié par un autre programme d'application (ici, P.A.1) alors l'objet chargé depuis la base de données fournit une perception ancienne et erronée de celui-ci[1],
- de même, si le cycle de vie d'un objet se termine pour un programme d'application (P.A.1) l'objet est recopié dans la base de données. Si, comme dans le cas précédent, un programme P.A.2 utilise ce même objet, dès qu'il le recopiera dans la base de données, les modifications réalisées par P.A.1 seront perdues.

[1] La distinction entre programmes d'application peut concerner le cas tout à fait général de deux applications différentes mais aussi le cas particulier de deux modules différents dans une même application, ce qui sera le cas le plus fréquent.

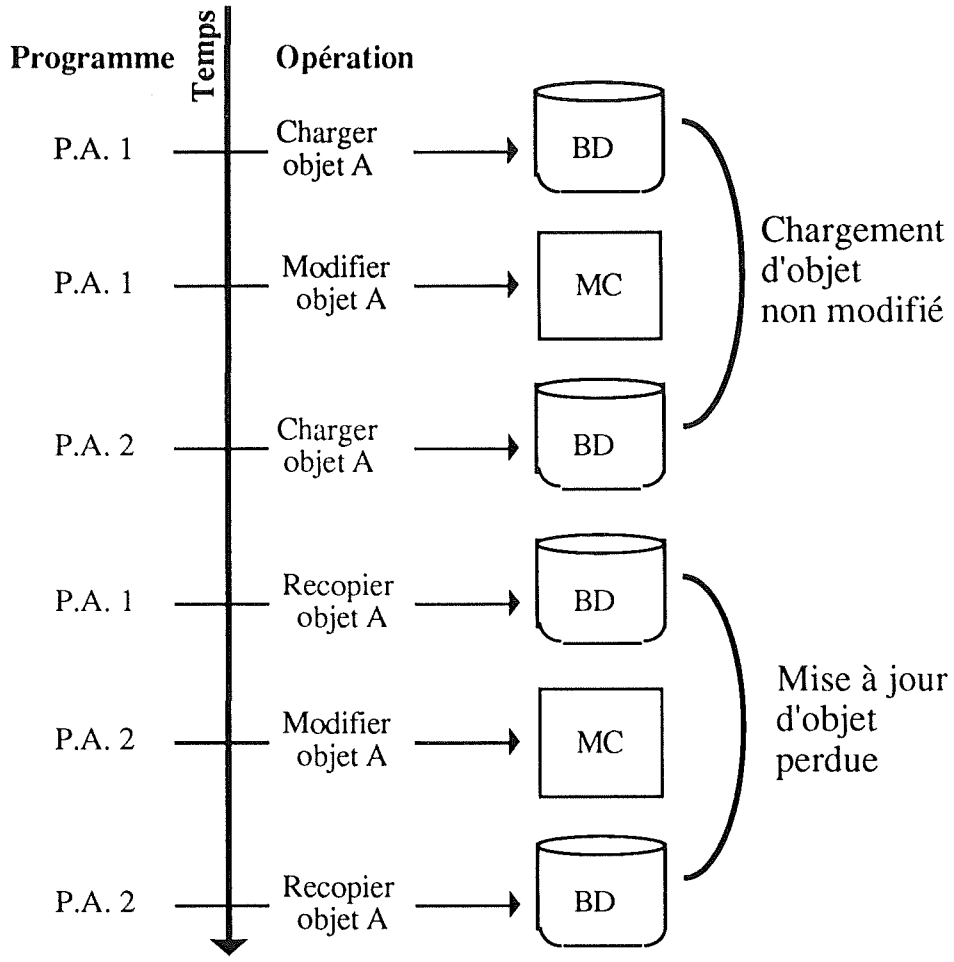

Figure 4.4 : Problèmes liés au chargement des objets en mémoire centrale

1.3.2. Table résidente et contrôleur d'accès aux objets

Ce type de problème n'est pas propre à notre application et se rencontre dans de nombreux cas où le parallélisme d'accès à des objets est permis. Plusieurs solutions existent à ce problème qui est notamment abordé par [ELIO-90].

Il existe deux façons de remédier à ce problème.

La première solution consiste à interdire l'accès à tout objet déjà chargé en mémoire centrale. De cette manière, P.A.2 n'accéderait pas à l'objet A et ainsi, n'aurait pas une perception erronée des données. Cette solution est implémentable en conservant en mémoire centrale une table contenant les noms de tous les objets en cours de cycle de vie. Les accès aux objets de la base de

données et la gestion de la table seraient assurés par un module spécial appelé contrôleur d'accès. Mais l'inconvénient de cette solution est qu'elle peut conduire à des temps d'attente longs dans le cas où un P.A. doit accéder à un objet qui est déjà en cours de vie et dont il a besoin pour continuer son exécution normale.

Une deuxième solution plus élaborée consiste à accepter l'accès à un objet déjà présent dans la table. Cependant, au lieu de le charger depuis la base de données, le contrôleur d'accès va chercher l'objet correspondant en mémoire centrale.

C'est cette deuxième solution, beaucoup plus satisfaisante que la première, que nous avons choisie. Elle s'exprime par deux choix de conception énoncés ci-dessous :

> Une table résidente en mémoire centrale contient tous les objets en cours de cycle de vie. Toute entrée de la table est assignée à une et une seule variable objet du P.A. Si deux objets identiques sont en cours de cycle de vie, les entrées correspondantes dans la table désigne les mêmes informations en mémoire centrale.

> Un contrôleur d'accès à la base de données gère l'accès aux objets de bureau. Quand le chargement d'un objet est demandé, si cet objet n'est pas déjà présent dans la table alors un défaut d'objet apparaît et le contrôleur d'accès accède à la base de données pour charger l'objet concerné en mémoire centrale.

D'un point de vue pratique, si l'exemple précédent est repris (voir figure 4.4), les variables objets des programmes d'application 1 et 2 correspondent à deux entrées différentes dans la table. Quand P.A.2 demande un cycle de vie pour l'objet A, le contrôleur d'accès constate que cet objet correspond déjà à une autre entrée dans la table résidente. Dès lors, l'entrée de P.A.2 est mise à la même valeur que l'entrée de P.A.1. Donc, le programme P.A.2 a une perception correcte de l'objet de bureau. De même, lors de la recopie dans la base de données, comme toutes les modifications réalisées par les P.A. ont porté simultanément sur le même objet en mémoire centrale, le mise à jour effectuée par P.A.1 n'est dès lors pas perdue.

Pour se donner une idée de la manière dont le contrôleur d'accès travaille, examinons le cas d'une variable objet d'un programme d'application à laquelle un seul cycle de vie d'objet de bureau est associé.

Dans un premier temps, l'utilisation de la méthode Init, propre à la variable objet du programme d'application, avertit le contrôleur d'accès qu'une nouvelle entrée doit être ajoutée à la

table résidente. Par opposition, la méthode Fini conduit à la suppression de cette entrée de la table résidente.

Lorsqu'un nouveau cycle de vie d'objet débute, soit par l'appel de la méthode Accès-Direct, soit par celui de la méthode Créer, le contrôleur d'accès réalise les séquences d'actions décrites au tableau 4.5 de manière à charger la version la plus récente de l'objet concerné par la demande. Si le cycle débute par l'appel de la méthode Créer, aucune vérification préalable de la présence de l'objet dans la table résidente n'est nécessaire, hormis celle due à la contrainte d'identifiant global.

Quand le programme d'application termine l'utilisation de l'objet de bureau, il le signifie en utilisant l'une des deux méthodes Fin-Utilisation ou Supprimer auxquelles le contrôleur d'accès associe les séquences d'actions décrites dans le tableau.

CHARGEMENT DES OBJETS : DEBUT DE CYCLE DE VIE	
Accès-Direct si Objet ∈ table résidente alors Recopier Objet → Entrée table sinon Accéder Objet en BD Copier Objet → Entrée table	**Créer** Créer Objet en BD Copier Objet → Entrée table
RECOPIE DES OBJETS : FIN DE CYCLE DE VIE	
Fin-Utilisation Recopier Entrée table dans BD Effacer Contenu de Entrée table	**Supprimer** Détruire Objet dans BD Pour toute Entrée table ∋ Objet Effacer Contenu de Entrée table

Tableau 4.5 : Description des actions réalisées par le contrôleur d'accès pour les méthodes régissant le cycle de vie des objets

Comme on le voit, seule la méthode Supprimer présente quelques difficultés lorsque l'objet concerné par la suppression est manipulé par plusieurs P.A. simultanément.

La possibilité de trouver en mémoire centrale des entrées vides (suite à l'utilisation de la méthode Supprimer) implique que le contrôleur d'accès doit vérifier à chaque nouvelle opération réalisée sur une variable objet si l'instance de classe d'objets qu'elle représente n'a pas été détruite par l'activation de la méthode Supprimer demandée par une autre variable objet. Cette séquence de contrôle sera donc associée à toute les méthodes d'objets qui travaillent sur les structures de données

chargées en mémoire centrale. Cela donne par exemple, le contrôle suivant pour la méthode Modifier-Att :

Modifier-Att

si Vide Entrée table
alors Opération illicite
sinon Modifier Entrée table

Tous les modules développés jusqu'à présent le sont selon l'approche orientée objets. La table résidente introduite dans cette partie constitue un module autonome formé d'une nouvelle classe d'objets : la classe Table-Res. Son interface reprendra toutes les méthodes permettant de construire de nouvelles entrées dans la table résidente, de lire et de modifier les valeurs prises par les entrées. Ces méthodes découlent directement des actions nécessaires au contrôleur pour qu'il puisse remplir sa fonction. Voici quelques exemples de services rendus par l'interface de la classe Table-Res :

- Allouer-Place est utilisée pour ajouter une nouvelle entrée à une table résidente,
- Garnir-Place sert à associer un objet à une entrée,
- Recopier-Place permet d'associer une entrée à une autre entrée, etc.

Toutes les méthodes associées à la classe Table-Res sont décrites dans l'annexe 4 "Spécifications des objets complémentaires". La décision qui consiste à faire de la classe Table un module autonome permet de profiter au maximum du principe d'information hiding (voir partie 2.3.2 "Cycle de vie du système d'environnement de bureau"). Ainsi, l'implémentation des méthodes de la classe Table-Res et toutes les modifications de la structure de données choisie pour la représenter en mémoire (par exemple, pour optimiser les performances de la boîte à outils) restent locales à ce module.

L'approche orientée objets utilisée lors de la conception de cette architecture la rend transparente pour l'utilisateur qui ne sait pas qu'il existe un niveau d'indirection supplémentaire lors de l'accès aux objets. Comme l'utilisateur ne peut accéder aux objets que via leur interface (principe d'information hiding), cette approche apporte une sûreté supplémentaire au niveau de l'accès à l'information. Ceci augmente également la robustesse[2] des modules d'accès à la base de données.

[2] La robustesse est la capacité d'un logiciel à fonctionner même dans des conditions anormales [MEYE-88].

1.3.3. Architecture de la boîte à outils

Nous allons maintenant décrire sommairement l'architecture du système. Cette architecture repose sur le principe décrit précédemment.

De manière à dissocier l'activité du contrôleur d'accès des activités propres à la gestion de l'environnement de bureau, la hiérarchie d'objets a été dupliquée.

D'un côté se trouvent les modules d'accès constitués par les classes d'objets du modèle d'environnement de bureau étendu. Leurs interfaces sont celles déjà décrites. Lorsque ces méthodes sont utilisées par un P.A., le contrôleur d'accès impose un filtre préalable à toute utilisation portant sur la vérification de la présence des objets en mémoire centrale. L'architecture en résultant est présentée à la figure 4.6.

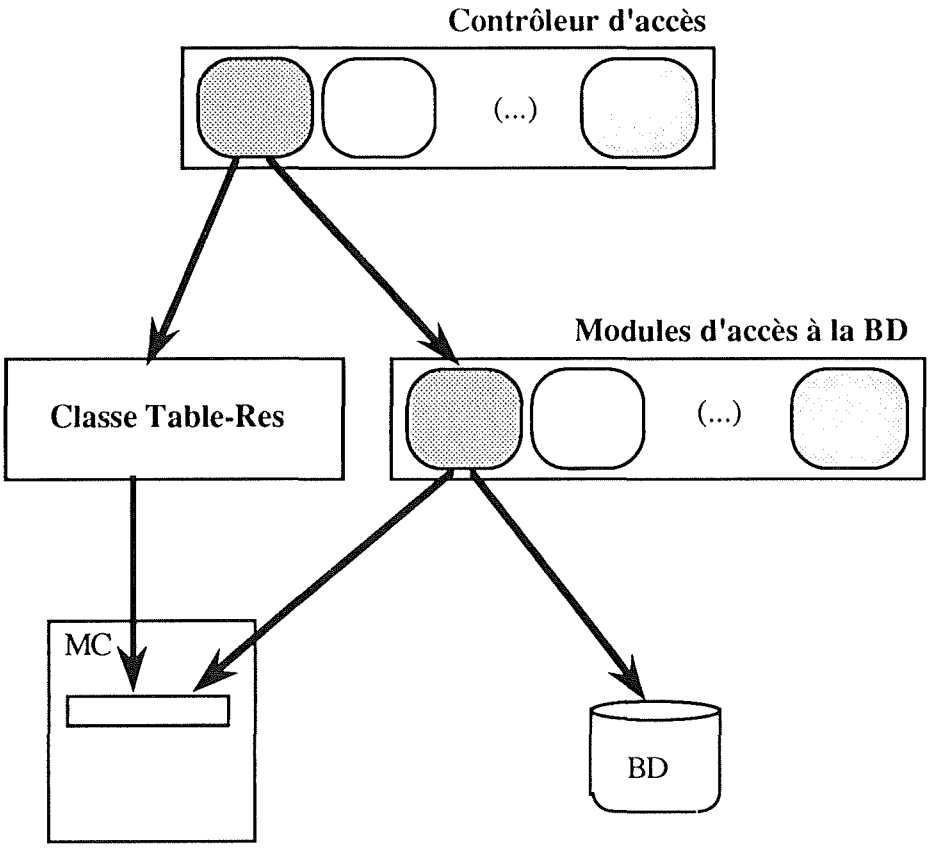

Figure 4.6 : Architecture des modules de la boîte à outils de gestion d'un environnement de bureau

Le contrôleur d'accès est lui aussi constitué d'un ensemble de modules calqués sur la hiérarchie des classes d'un environnement de bureau. Cette duplication se traduit par l'existence, pour chaque classe d'objets du modèle d'environnement de bureau, d'une classe équivalente au niveau contrôleur d'accès. Son activité est de gérer l'accès à la table pour les méthodes propres à la classe d'objets de bureau dont la classe associée au contrôleur est le pendant. Les principales utilités de cette duplication sont :

- de pouvoir profiter au niveau du contrôleur d'accès du mécanisme d'héritage et donc de rendre les modules du contrôleur d'accès facilement extensibles par opposition à l'extensibilité plus faible des modules d'accès à la base de données qui est limitée par l'extensibilité du schéma conceptuel des données,
- de fournir à l'utilisateur une interface pour chaque variable objet manipulée par le programme d'application identique à celle de la classe d'objet de bureau correspondante et ce malgré l'existence d'un niveau d'indirection supplémentaire dû au contrôleur,
- d'isoler dans des modules distincts des activités très différentes ce qui facilite la maintenance,
- d'un point de vue réutilisabilité, l'isolation des aspects BD et du contrôle d'accès implique que toute l'architecture ne devra pas être modifiée si la gestion de la BD ou des objets en mémoire doit subir des modifications,
- si la base de données change, seuls les modules d'accès à la base de données doivent être réécrits. De même, si la stratégie du contrôleur d'accès est modifiée, seuls les modules du contrôleur d'accès doivent subir des modifications.

Le contrôleur d'accès manipule la table grâce aux méthodes offertes par la classe Table-Res. Cette table étant conçue également selon une approche orientée objets, toutes les opérations qu'elle effectue sont invisibles pour le contrôleur d'accès.

2. Spécification formelle des classes d'objets

2.1. Description du principe de la spécification

Il existe deux méthodes de spécification d'une classe d'objets :

- la **spécification par structures de données** : dans ce cas, la définition préliminaire des structures de données assignées aux classes d'objets implique que la spécification de l'interface des objets est influencée par le type de données choisi pour les définir. Ainsi, une structure mal adaptée peut rendre les spécifications complexes,
- la **spécification par type abstrait** : par opposition à la spécification par structures de données, cette technique possède l'avantage de pouvoir spécifier des objets de manière complète sans devoir leur associer une structure de données et donc sans risque d'effectuer un mauvais choix de représentation. Le choix final d'une représentation se fait seulement au moment de l'implémentation.

Dans notre cas, la spécification par structures de données semble être mieux adaptée que la spécification par type abstrait car le schéma de classes est étroitement lié au schéma conceptuel des données. Ainsi, les structures étant définies et fixées à l'avance, nous possédons une bonne base pour réaliser la spécification des classes d'objets. Une première spécification formelle basée sur les équations algébriques nous a permis de confirmer cette intuition.

Pour réaliser la spécification, les informations associées à la partie privée de chaque classe d'objets sont définies à l'aide du produit cartésien (CP). Les classes dérivées sont spécifiées par enrichissement progressif des spécifications de l'interface de leurs classes de base en utilisant les mécanismes d'héritage simple et multiple. Le langage de spécification utilisé est celui décrit au cours de Méthodologie de développement de logiciels [DUBO-89].

Pour se fixer les idées, voyons la structure de données associée à la classe OB :

OB = CP[nom : STRING, description : STRING, créateur : STRING, propriétaire : STRING, date-création : DATE, date-der-cons : DATE, date-der-maj : DATE]

Cette structure de données reprend toutes les informations relatives aux objets de bureau qui sont stockés dans la base de données (pour rappel, se référer à la section 3.3 "Schéma E/A de l'environnement de bureau étendu"). L'objet chargé en mémoire centrale contiendra donc toutes les caractéristiques nécessaires pour la construction de ses méthodes sans qu'aucun accès ultérieur à la base de données ne soit nécessaire.

Le mécanisme d'héritage intervient pour la représentation des structures de données des classes descendantes. Ainsi, pour la classe O-Rgt, la structure de données associée est décrite par :

O-Rgt **is a** O B
with CP[contient-origine : Liste-Objets, contient-réelle: Liste-Objets]

La relation *is-a* traduit le fait que tout O-Rgt est un OB et possède donc la même structure de données que tout objet de la classe OB. Cette structure est enrichie par l'ensemble des informations propres à la classe O-Rgt qui suit le *with* dans la définition de l'objet.

La structure Liste-Objets mérite une remarque.

Cette structure de données est une structure particulière que nous avons créée afin de pouvoir y associer des objets de bureau de tout type. Elle est inspirée de la structure ensembliste proposée par [BRIS-91] et nous permettra de décrire tous les types d'associations présents dans le schéma conceptuel des données. Cette modélisation consiste à représenter tous les objets cibles d'un type d'association par un attribut de l'objet origine reprenant sous la forme d'un séquence tous les objets cibles. Ce type de structure fait encore l'objet de recherches à l'heure actuelle, parfois très controversées. Nous ne nous étendrons pas plus sur ce problème qui ne fait pas partie de l'objet de ce mémoire.

Pour la propriété contient-réelle, elle fournit l'ensemble de tous les objets rangeables qui participent à une association Localisation-Origine avec l'objet de rangement à décrire.

La caractérisation complète d'une classe d'objet comprend en plus de la définition de sa structure de données, ses invariants et la spécification des méthodes de leur interface.

L'héritage porte sur la structure de données, les invariants et les méthodes.

Les invariants sont ceux présentés dans le schéma conceptuel des données mais, cette fois exprimés en terme de la structure de données associées aux classes d'objets. Par exemple, pour la classe OB, la contrainte d'identification associée au nom des objets de bureau s'exprime de la manière suivante :

$$\forall\ o, o' : o \neq o' \Rightarrow nom(o) \neq nom(o')$$

où o, o' sont des instances de la classe OB.

Pour réaliser les spécifications, les méthodes ont été divisées en trois catégories [MEYE-88] :

- les **constructeurs** qui permettent de créer de nouvelles instances d'une classe d'objets,
- les **modificateurs** qui servent à modifier les propriétés d'objets existants,
- les **observateurs** qui servent à consulter les informations contenues dans la mémoire privée des objets.

Dans un premier temps, les méthodes sont décrites par leur signature. Elle est composée des paramètres données et des paramètres résultats. Cette description est celle dont nous avons déjà parlé dans la partie 3.4 "Description des interfaces des classes d'objets". Par exemple, pour la méthode Modifier-Att de la classe OB cela donne :

Modifier-Att : OB x liste-att-OB → OB

où les paramètres données sont ceux situés à gauche de → et les paramètres résultats ceux à droite.

Suite à la construction d'une architecture des modules et au choix d'une structure de données à associer à chaque classe d'objets, les méthodes peuvent être décrites en termes de précondition et de postcondition.

2.2. Spécification des classes d'objets de l'architecture

La description complète des méthodes des classes d'objets se trouve en annexe. Chaque annexe reprend dans l'ordre, pour chaque classe d'objets y étant décrite : la structure de données associée à la classe, les invariants définis sur cette classe, la description complète de l'interface de l'objet en terme de signature et finalement, la spécification complète de toutes les méthodes en considérant d'abord les constructeurs puis les modificateurs et les observateurs.

L'annexe 2 "Spécifications des modules d'accès" est consacrée à la spécification de la hiérarchie des objets de bureau. Pour plus de facilité, la base de données a été décrite comme étant un état[3] du système auquel on accède par des primitives types qui travaillent sur la base de données et

[3] Dans le langage de spécification utilisé, un état est une variable globale pour l'application et persistante d'une utilisation à l'autre.

qui renvoient aux méthodes des classes d'objets les structures de données par lesquelles elles sont représentées. Par exemple, la primitive d'accès à un objet de bureau dans la base de données s'écrit :

Charger-OB(nom) = o

En entrée, elle reçoit le nom identifiant l'objet qui est stocké dans la base de données. En sortie, elle renvoit un objet de bureau possédant la structure de données (telle que définie plus haut). Tous les accès nécessaires aux types d'entités et aux types d'associations pour construire cette structure sont réalisés dans les primitives de l'état BD.

L'annexe 3 "Spécifications du contrôleur d'accès" reprend toutes les classes calquées sur les classes de la hiérarchie d'un environnement de bureau étendu. Par le niveau d'indirection supplémentaire qu'il procure, le contrôleur d'accès procure une meilleure sécurité des données et toute tentative d'accès aux objets autrement que par l'utilisation de leurs méthodes se traduira par un échec car le programmeur d'application n'a aucun moyen de connaître la localisation de l'objet de bureau en mémoire centrale.

L'annexe 4 "Spécifications des objets complémentaires" est consacrée à la description des classes d'objets Table-Res et Liste-Objets.

3. Construction d'un schéma E/A de base

Nous allons maintenant débuter la phase de transformation du schéma E/A étendu de la hiérarchie de classes d'objets. Le but final est de le rendre conforme au S.G.B.D. choisi pour l'implémentation, à savoir N.D.B.S. La première transformation à réaliser consiste à mettre ce schéma sous une forme non orientée objets, c'est-à-dire à en éliminer les structures de généralisation/spécialisation.

3.1. Transformation du schéma E/A étendu

Par définition, un schéma E/A de base est un schéma E/A dont toutes les structures de généralisation/spécialisation (relations is-a) ont été éliminées. Le schéma E/A de l'environnement de bureau devra subir plusieurs modifications pour être mis sous cette forme. A partir de ce schéma E/A de base, un schéma MAG pourra être produit.

D'emblée, une remarque importante s'impose, les transformations qui seront apportées au schéma conceptuel des données sont évidemment sans influence sur la définition du schéma de classes d'objets.

Pour transformer un schéma E/A étendu en un schéma E/A de base, trois techniques sont disponibles [HAIN-89], [COJO-88] .La description de ces techniques est donnée dans l'annexe 5 "Les techniques de transformation de schéma" :

- la matérialisation des relations is-a (M.R.is-a),
- la représentation des types d'entités spécifiques (R.T.E.S.),
- la représentation des types d'entités génériques (R.T.E.G.).

Des avantages et inconvénients multiples, difficiles à cerner, sont liés à l'utilisation de chacune de ces trois techniques. Ainsi, la transformation du schéma a dû être réalisée par essais successifs jusqu'à l'obtention d'un schéma final satisfaisant du point de vue performances d'accès et nombre de contraintes d'intégrité à gérer. Pour obtenir plus de détails sur les principes d'utilisation de ces techniques de transformation, nous renvoyons le lecteur aux ouvrages précités.

Les transformations appliquées sur le schéma étendu seront expliquées en subdivisant le schéma E/A en trois parties : le sommet comprenant le type d'entité OB, le bas reprenant les types d'entités Collection, Info, O-Rgt-Term et leurs descendants et le niveau intermédiaire qui reprend les descendants directs de OB.

3.1.1. Transformation du sommet de la hiérarchie

Le type d'entité OB est éliminé par la technique de R.T.E.S. Les attributs du type d'entité générique sont reportés dans ses types spécifiques, c'est-à-dire O-Info et O-Rgt. La technique utilisée est illustrée à la figure 4.7.

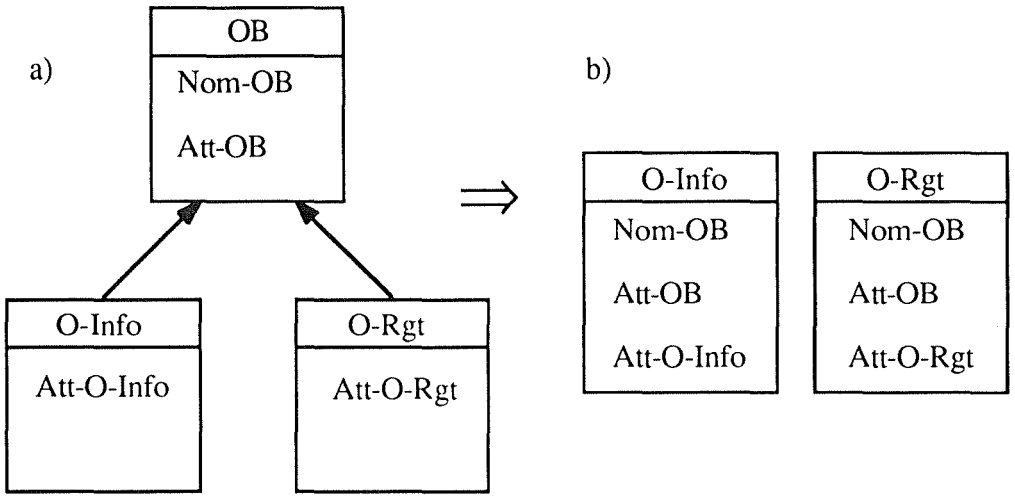

Figure 4.7 : Transformation par R.T.E.S. pour le type d'entité OB :
a) situation initiale b) situation finale

Les classes O-Info et O-Rgt avaient été obtenues par application du critère de partitionnement "porteur d'information" (voir section 3.2 "Présentation d'un environnement de bureau étendu"), ce qui résulte en une double conséquence lors de la transformation. Primo, comme la structure d'inclusion est couvrante[4], il n'est pas nécessaire d'ajouter à la structure transformée un type d'entité spécifique destiné à la représentation d'éventuelles occurrences du type d'entité OB qui ne seraient ni des O-Info, ni des O-Rgt. Secundo, cette structure est aussi disjointe[5]. Il n'est donc pas nécessaire de créer un type d'entité supplémentaire qui regrouperait les occurrences d'OB qui appartiendraient simultanément aux deux sous-types.

Ces remarques étant faites, la transformation du schéma E/A se résume à la représentation des deux types d'entités O-Info et O-Rgt avec report des attributs génériques du type d'entité OB. Il

---

[4] Rappelons que tout objet de bureau est au moins un objet informationnel ou un objet de rangement, d'où le caractère couvrant de la structure d'inclusion.

[5] De même, tout objet de bureau est soit un objet informationnel soit un objet de rangement d'où le caractère disjoint des classes O-Info et O-Rgt.

faudra gérer une **contrainte d'identifiant global** sur l'attribut Nom (voir figure 4.7.b) commun aux types d'entités O-Info et O-Rgt. La transformation choisie est satisfaisante car elle respecte le principe de représentation unique des objets du réel perçu.

Notons, finalement, que le report des attributs du type d'entité OB vers ses deux autres types d'entités spécifiques O-Rgble et O-Non-Rgble n'est pas nécessaire. En effet, comme ces deux classes ont pour origine le deuxième critère de partitionnement définis sur les objets de bureau, les OB appartenant à l'une de ces deux classes dérivées sont aussi soit des objets informationnels, soit des objets de rangement et possèdent donc déjà une description dans les types d'entités O-Info et O-Rgt.

Une fois le type d'entité OB transformé, la transformation du reste du schéma E/A étendu en schéma E/A de base consiste à rechercher une représentation conforme pour les autres types d'entités du schéma E/A étendu. Comme les types d'entités descendant directement du type d'entité OB sont des types d'entité de base pour les types d'entités terminaux de la hiérarchie, nous commencerons par la transformation de ces types d'entités terminaux.

3.1.2. Transformation du bas de la hiérarchie

Les types d'entités du bas de la hiérarchie sont ceux provenant de l'héritage multiple à savoir les types d'entités Collection, Info, O-Struct et O-Rgt-Term. Dans cette partie, nous considérons la transformation de leurs sous-types, c'est-à-dire les types d'entités matérialisant, dans le formalisme E/A, les classes d'objets terminales du schéma de classes.

3.1.2.1. L'entité Collection

Les types d'entités spécifiques du type d'entité Collection sont supprimés par la technique de R.T.E.G. Le résultat de la transformation est illustrée à la figure 4.8.

Un méta-attribut Type-collection est créé. C'est un attribut simple et obligatoire. Simple car une collection est soit un dossier, soit un fichier mais jamais les deux simultanément. Obligatoire car une collection est forcément d'un de ces deux types. Les caractéristiques de cet attribut traduisent la structure de partition initialement définie sur les types d'entités spécifiques. Cet attribut a pour domaine de valeurs les noms des types d'entités spécifiques, c'est-à-dire {Fichier, Collection}. L'attribut Type-composant (présent à la figure 4.8.b) provient de l'héritage ascendant de l'attribut du type d'entité Fichier et devient dès lors un attribut facultatif.

La transformation du schéma introduit une contrainte de structure supplémentaire soit :

Pour toute occurrence du type d'entité Collection, si l'attribut Type-collection prend la valeur "Fichier" alors l'attribut Type-composant doit prendre une valeur.

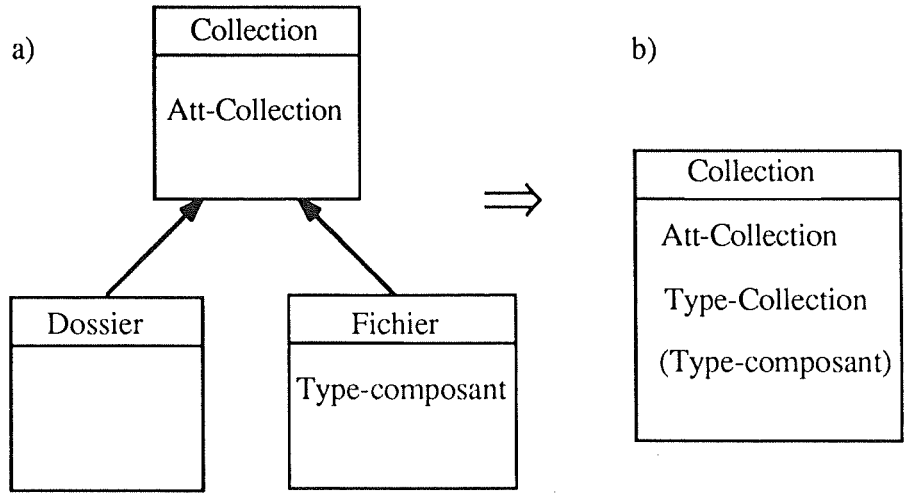

Figure 4.8 : Transformation par R.T.E.G. pour le type d'entité Collection : a) situation initiale b) situation finale

3.1.2.2. L'entité Information

Les types d'entités spécifiques du type d'entité Info (à savoir : Message, Formulaire et Document) sont éliminés par la technique de R.T.E.G. Le résultat de la transformation est représenté à la figure 4.9.b. La méthode utilisée est identique à celle employée pour transformer le type d'entité Collection.

Lors de la transformation, il y a création d'un méta-attribut Type-info. Cet attribut est simple et obligatoire car les types spécifiques reposent sur une structure de partition[6]. Son domaine de valeurs est {Message, Formulaire, Document}. Les attributs provenant de l'héritage ascendant des types d'entités Message et Formulaire deviennent des attributs facultatifs du type d'entité Info. Il est à remarquer que le report des seuls attributs du type d'entité Message suffit car les deux types d'entités Message et Formulaire possédaient des attributs identiques (voir section 3.3 "Schéma E/A de l'environnement de bureau étendu"). Ainsi, la transformation par R.T.E.G. permet d'éliminer

[6] L'attribut est simple car une information est soit un document, soit un message, soit un formulaire. Il est obligatoire car une information est forcément d'un de ces trois types.

une structure redondante. Les nouveaux attributs ont pour noms : Mode-repr, Contenu, Stockage et Lieu-stockage (repris sous le nom générique att-MF à la figure 4.9.b).

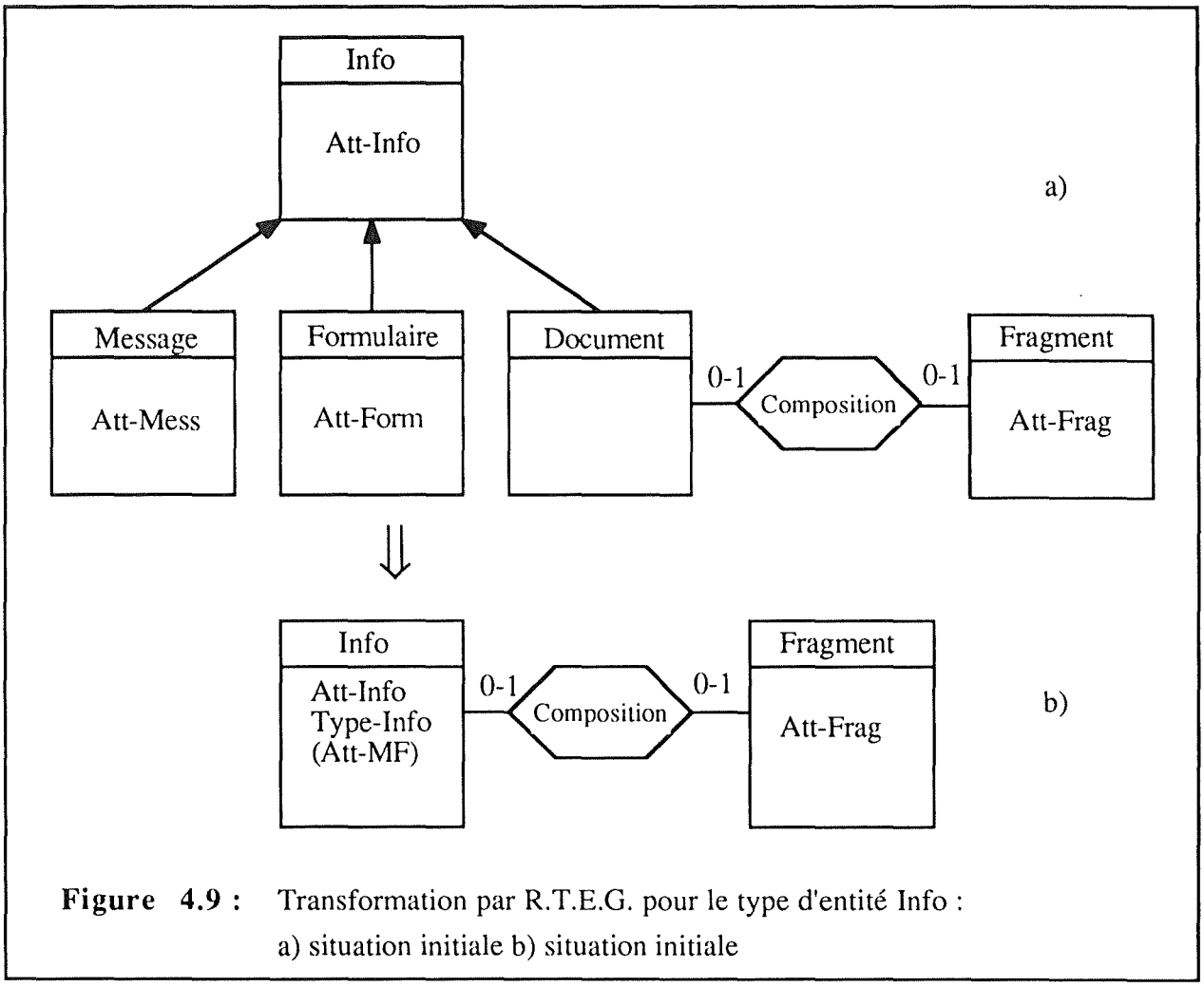

Figure 4.9 : Transformation par R.T.E.G. pour le type d'entité Info : a) situation initiale b) situation initiale

Le type d'association Composition défini sur le type d'entité Document est reporté au niveau du type d'entité Info (voir figure 4.9.b). Comme elle n'a de sens que pour les Documents, une contrainte de structure supplémentaire est introduite, soit :

> Pour toute occurrence du type d'entité Info, si l'attribut Type-info prend la valeur "Document", alors cette occurrence peut participer à l'association Composition (elle ne doit pas car le rôle était initialement facultatif).

De plus, la contrainte d'intégrité définie précédemment sur le type d'association Copie voit son expression transformée en :

> Les occurrences du type d'entité Info impliquées dans une association Copie doivent avoir même valeur pour l'attribut Type-info.

3.1.2.3. L'entité O-Rgt-Term

Le type d'entité O-Rgt-Term ne possède pas d'attributs qui lui sont propres, ni d'attributs hérités du type d'entité O-Non-Rgble (voir figure 4.10.a). Par contre, il possède trois types d'entités spécifiques : Armoire, Table et Poubelle. Ces derniers sont éliminés par la technique de représentation des types d'entités génériques (voir figure 4.10.b).

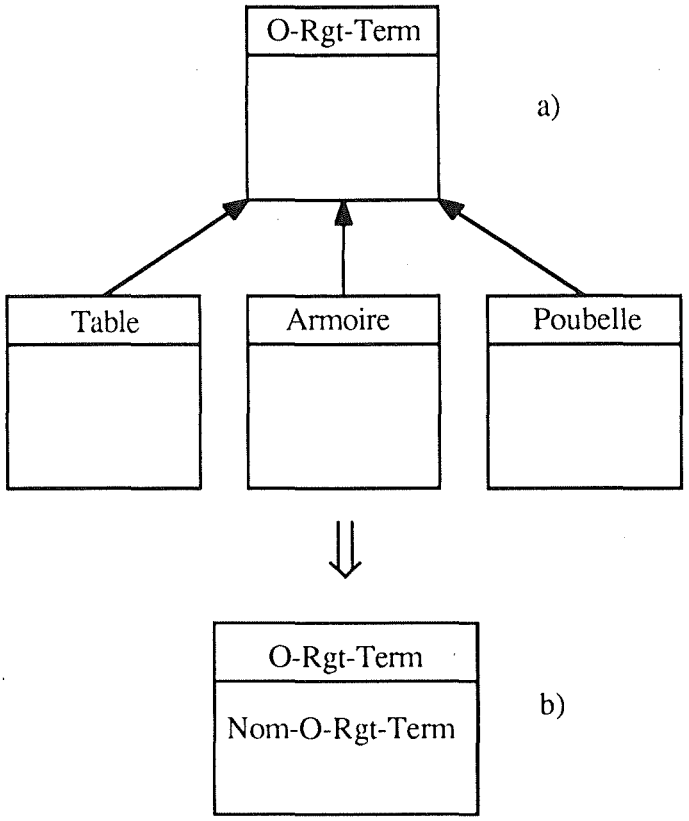

Figure 4.10 : Transformation par R.T.E.G. pour le type d'entité O-Rgt-Term : a) situation initiale b) situation finale

Comme les types d'entités Armoire, Table et Poubelle ne possèdent pas d'attributs, il suffit d'introduire, au niveau du type d'entité O-Rgt-Term, un méta-attribut Nom-O-Rgt-Term ayant pour domaine de valeurs les noms des types d'entités spécifiques ce qui donne {Armoire, Table, Poubelle}. Ainsi, la connaissance de la classe dont l'objet créé fait partie est conservée dans la base

de données via l'attribut du type d'entité O-Rgt-Term. La structure de partition définie sur les trois sous-types d'O-Rgt-Term rend cet attribut simple et obligatoire[7].

3.1.2.4. L'entité Fragment

La classe Fragment a été définie comme une classe autonome. Dès lors, le type d'entité la représentant est aussi autonome. Ceci se remarque dans le schéma E/A des fragments par l'absence de structures d'héritage définies sur le type d'entité Fragment (pour rappel, se référer à la figure 3.13 présentant la structure des documents). Ainsi, le schéma initial se trouve déjà sous forme E/A de base.

3.1.3. Transformation du niveau intermédiaire de la hiérarchie

La transformation des types d'entités représentant les classes d'objets intermédiaires du schéma présente des difficultés car ces classes reposent sur des structures d'héritage multiple. Par exemple, la classe Info est héritière des classes de base O-Info et O-Rgble et donc le type d'entité Info est impliqué dans deux relations is-a avec les types d'entités O-Info et O-Rgble. Ainsi, lors de la transformation, les modifications apportées se répercutent sur deux types d'entités. Il faut donc veiller à ce que la transformation, avantageuse pour un type d'entité, ne se fasse pas au détriment d'un autre.

Les types d'entités à transformer seront examinés dans leur ordre d'apparition de la gauche vers la droite dans le schéma général.

3.1.3.1. L'entité O-Info

La structure d'héritage du type d'entité générique O-Info vers les types d'entités spécifiques Info et Collection (voir figure 4.11.a) est transformée en une structure conforme au schéma E/A de base par la technique de M.R.is-a (voir figure 4.11.b). C'est cette technique qui offre le moins de perturbations par rapport au schéma E/A initial.

Le type d'association is-a obtenu possède, comme rôle spécifique, un rôle multidomaine défini sur les types d'entités Collection et Info (voir figure 4.11.b). Ce rôle multidomaine signifie qu'une entité de O-Info participe à une association is-a soit avec une entité de type Info soit avec une entité de

[7]L'attribut est simple car un objet de rangement terminal est soit une armoire, soit une table, soit une poubelle. Il est obligatoire car un objet de rangement terminal est forcément d'un de ces trois types.

type Collection, mais pas avec les deux simultanément[8]. Inversement, toutes les occurrences des types d'entités Info et Collection sont reliées à une entité du type O-Info (connectivité maximale). Les connectivités du rôle générique s'expliquent par la structure de partition définie sur O-Info : la connectivité minimale (valant un) rend compte de la couverture et la connectivité maximale (valant un aussi) rend compte de la disjonction existant entre les classes Info et Collection.

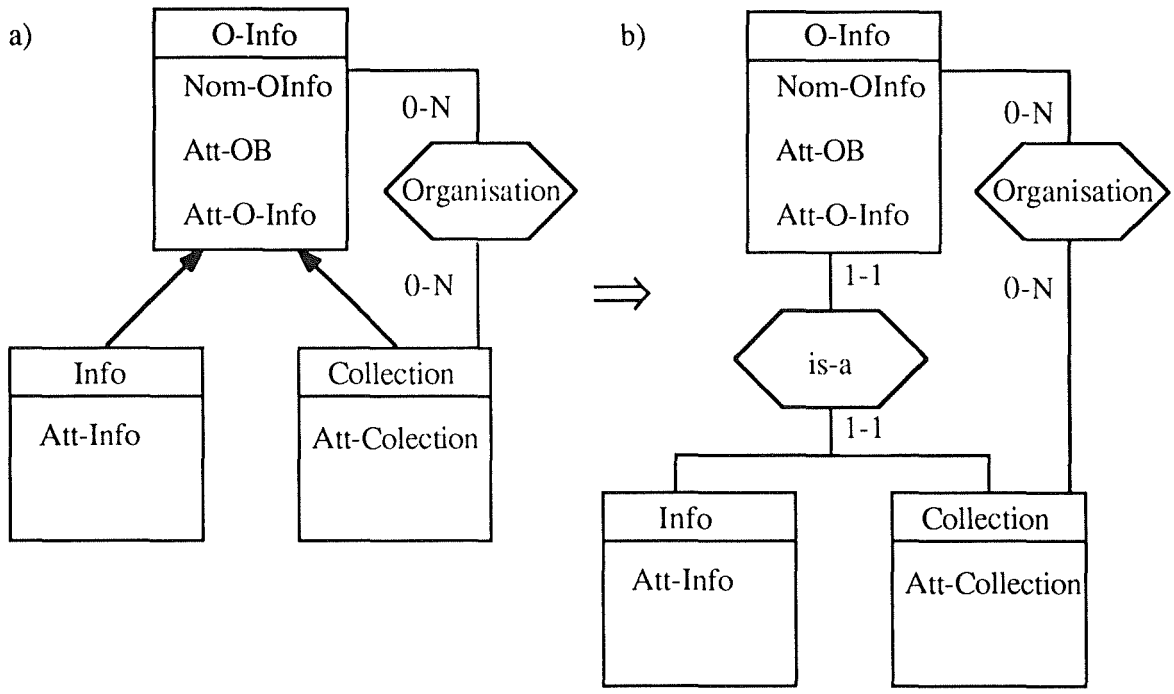

Figure 4.11 : M.R.is-a pour la structure d'héritage définie sur le type d'entité O-Info : a) situation initiale b) situation finale

Ces contraintes de connectivité constituent les seules contraintes supplémentaires définies sur le schéma transformé.

Lors de la transformation par M.R.is-a, le type d'association Organisation ne subit aucune modification car tout objet informationnel peut y participer (voir figure 4.11.b).

Suite à l'application de cette transformation, il apparaît que tout objet du réel perçu qui appartient soit à la classe Info soit à la classe Collection est représenté dans la base de données par trois occurrences :

---

[8] Rappelons que la définition de rôle multidomaine revient à exprimer l'existence d'une contrainte d'exclusion de rôles définies sur les types d'entités impliqués dans le rôle multidomaine.

- une occurrence du type d'entité O-Info,
- une occurrence du type d'entité O-Rgble et,
- une occurrence du type d'entité spécifique auquel il appartient.

Ainsi, il y a non respect du principe de représentation unique des objets du réel perçu.

Si les contraintes relatives à ces occurrences multiples sont mal gérées, de graves problèmes de cohérence de la base de données peuvent survenir. L'obligation pour un programmeur d'application de manipuler les objets en passant par leur interface rend le processus totalement transparent. Le programmeur ignore la manière dont l'objet est stocké dans la base de données[9] et donc ne peut travailler autrement qu'avec les outils qui lui sont fournis. La cohérence est ainsi associée à une correcte implémentation de ces opérations.

3.1.3.2. L'entité O-Rgble

La technique de M.R.is-a est aussi appliquée pour matérialiser l'héritage descendant des attributs du type d'entité O-Rgble vers ses types d'entités spécifiques Info, O-Struct et Collection (voir figure 4.12).

[9]Ceci est conforme au principe d'"information hiding" souvent invoqué en approche orientée objets (voir chapitre 2).

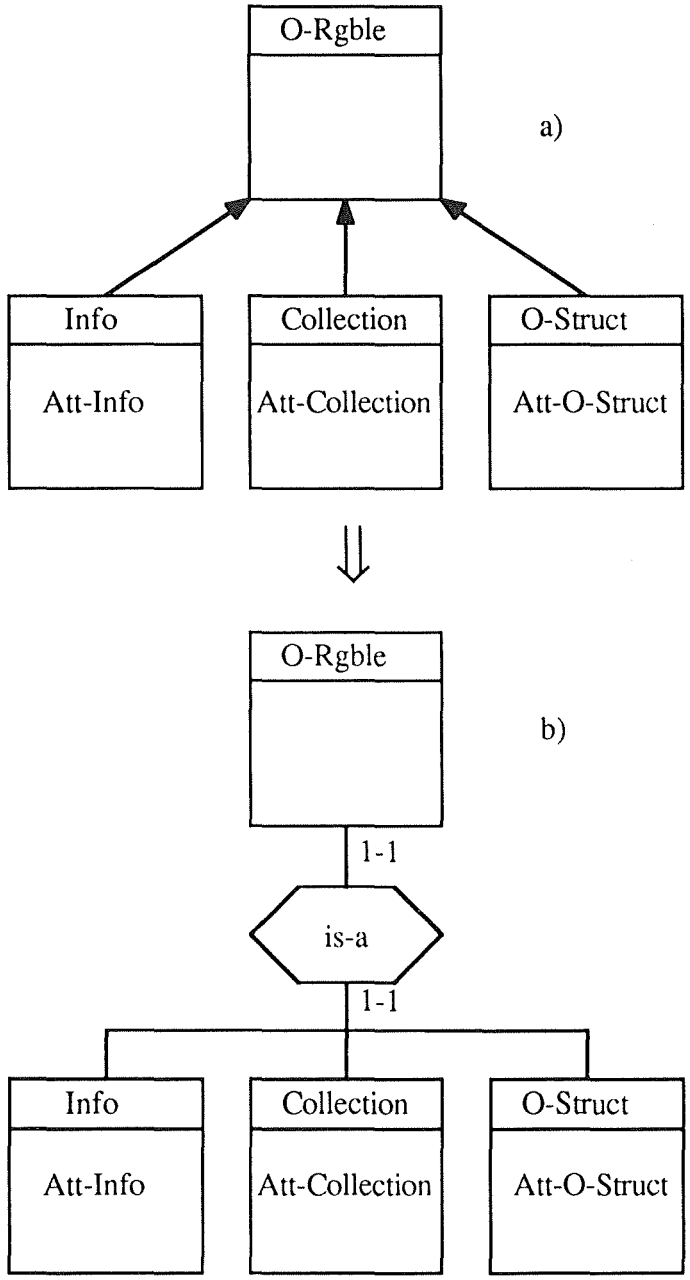

Figure 4.12 : Transformation par M.R.is-a pour le type d'entité O-Rgble :
a) situation initiale b) situation finale

L'explication des valeurs des connectivités du rôle générique et du rôle multidomaine du type d'association is-a résultant de la structure d'héritage de O-Info vers Info et Collection est également valable ici[10]. Il n'y a pas de contrainte d'intégrité supplémentaire à gérer.

Comme on le voit, la technique de M.R.is-a est génératrice de moins de contraintes que les techniques de transformation par R.T.E.G. et R.T.E.S. Cependant, la consultation d'une instance des classes définies de cette manière (par exemple, une instance de la classe O-Struct) nécessitera un accès logique de plus à la base de données.

3.1.3.3. L'entité O-Rgt

Le type d'entité O-Rgt a pour types d'entités spécifiques O-Struct et O-Rgt-Term. Nous avons choisi pour la modification de cette partie du schéma, une méthode de transformation mixte. En effet, cette méthode conduit à un schéma transformé plus simple.

En choisissant la technique de R.T.E.G. pour la structure de généralisation/spécialisation définie entre le type d'entité O-Rgt et les type d'entités O-Struct et O-Rgt-Term (voir figure 4.13.a), nous éliminons aussi ces dernières.

Cependant, cette technique n'est appliquée qu'au type d'entité spécifique O-Rgt-Term et pas à O-Struct à cause de la structure d'héritage multiple définie sur le type d'entité O-Struct en commun avec le type d'entité O-Rgble. Si les attributs du type d'entité O-Struct avaient été rapatriés dans le type d'entité O-Rgt, on aurait obtenu une représentation non consistante par rapport à la matérialisation is-a choisie précédemment pour les autres types d'entités spécifiques de O-Rgble (voir figure 4.12.b). La transformation mixte pour les objets de rangement résulte en :

- la disparition du type d'entité O-Rgt-Term, comme illustré par la figure 4.13.b et,
- la création d'un type d'association is-a entre les types d'entités O-Rgt et O-Struct ainsi qu'entre les types d'entités O-Rgble et O-Struct.

L'existence d'un méta-attribut nommé Type-O-Rgt appartenant au type d'entité O-Rgt permet de reconnaître la nature de tout objet de rangement. Cet attribut est simple et obligatoire et a pour domaine de valeur {O-Rgt-Term, O-Struct}[11].

---

[10]En particulier, la connectivité 1-1 du rôle multidomaine traduit autrement l'existence d'une contrainte d'exclusion entre les types d'entités Info, Collection et O-Struct.

[11]En effet, une structure de partition est définie sur les classes d'objets dérivées de la classe O-Rgt.

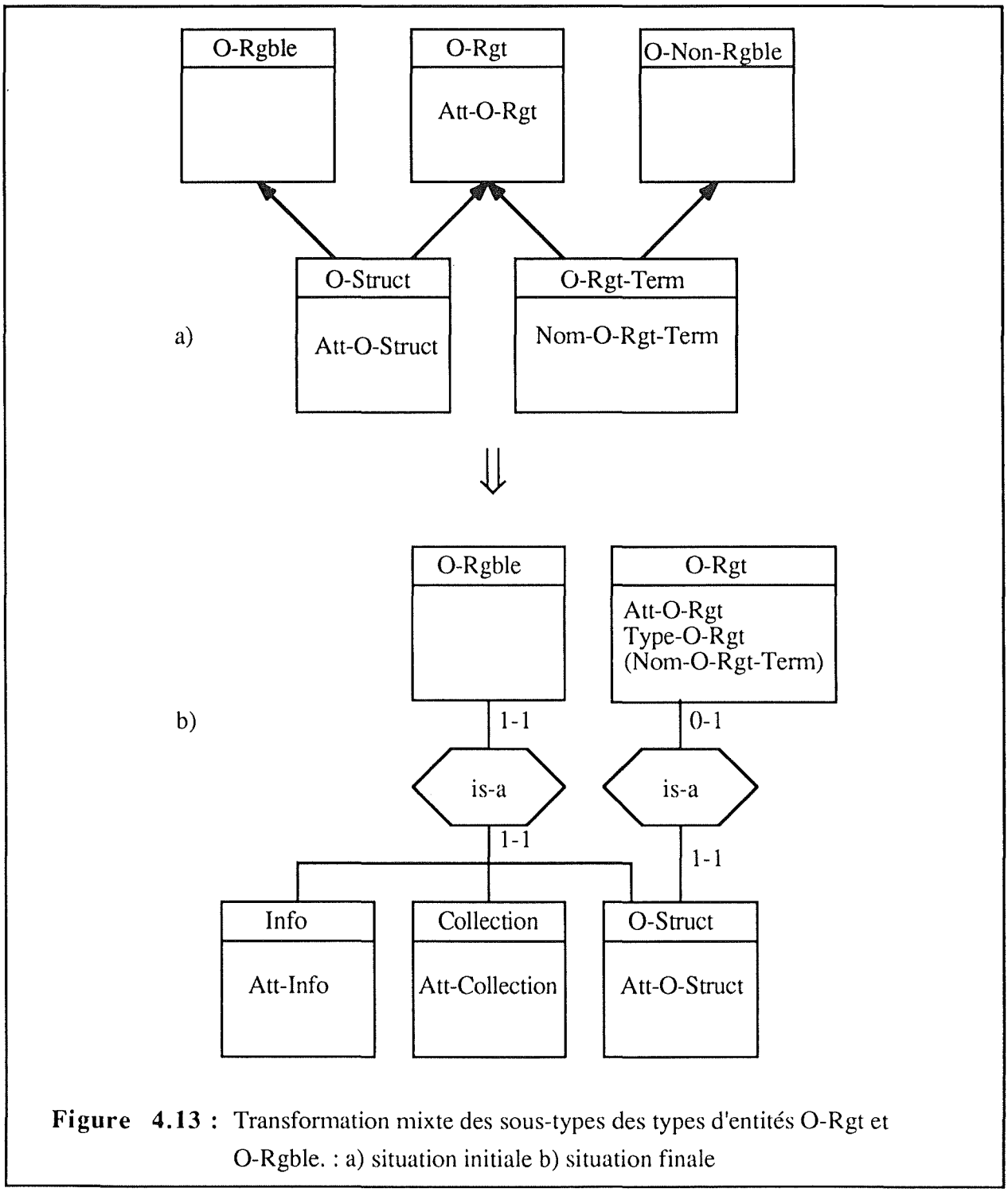

Figure 4.13 : Transformation mixte des sous-types des types d'entités O-Rgt et O-Rgble. : a) situation initiale b) situation finale

Ce type de transformation mixte, à première vue compliqué, apporte une triple simplification.

Premièrement, il existe une représentation unifiée, unique pour tous les sous-types du type d'entité O-Rgble. En effet, les trois classes Info, Collection et O-Struct apparaissent comme des

types d'entités explicites dans la base de données, reliés chacun à leur type d'entité générique par un type d'association matérialisant la relation is-a (voir figure 4.13.b).

Deuxièmement, les objets de rangement terminaux et leurs descendants sont représentés par une occurrence unique du type d'entité O-Rgt dans la base de données. Ceci implique une gestion moins lourde des contraintes et diminue le nombre d'accès logiques nécessaires pour obtenir toutes les informations relatives aux objets des classes Armoire, Table et Poubelle dont les instances sont maintenant accessibles en un seul accès logique.

Troisièmement, le type d'entité O-Non-Rgble reste sans attributs et sans lien avec les autres types d'entités : il peut donc être éliminé du schéma. Malgré la disparition de ce type d'entité, la classe d'objet O-Non-Rgble définie dans le schéma de classes, reste connue dans la base de données via le méta-attribut Type-O-Rgt du type d'entité O-Rgt.

L'attribut Nom-O-Rgt-Term provient de l'héritage ascendant, il est simple comme précédemment mais il devient facultatif car il est sans signification pour les objets de la classe O-Struct. Le rôle générique du type d'association is-a a une connectivité minimale de zéro car tout objet de rangement n'est pas forcément un objet de rangement structurant.

La valeur prise par l'attribut Type-O-Rgt détermine l'existence possible de chacun des attributs et des rôles définis sur le type d'entité O-Rgt. Pour cela, il suffit d'imposer la contrainte de structure suivante :

> Pour toute occurrence du type d'entité O-Rgt, si la valeur de l'attribut Type-O-Rgt est "O-Struct", alors l'objet n'est pas un objet de rangement terminal, l'attribut Nom-O-Rgt-Term ne prend pas de valeur et il existe une association is-a entre les types d'entités O-Rgt et O-Struct.

3.2. Schéma E/A de base de l'environnement de bureau

3.2.1. Redéfinition des attributs

Le schéma E/A finalement obtenu suite aux transformations réalisées est présenté à la figure 4.14. Les attributs des types d'entités du schéma sont repris ci-dessous. Quand leur description n'est pas explicitée, c'est-à-dire quand un nom général du type att-OB ou att-O-Info est utilisé, il s'agit des attributs du schéma conceptuel initial, repris sans autres modifications.

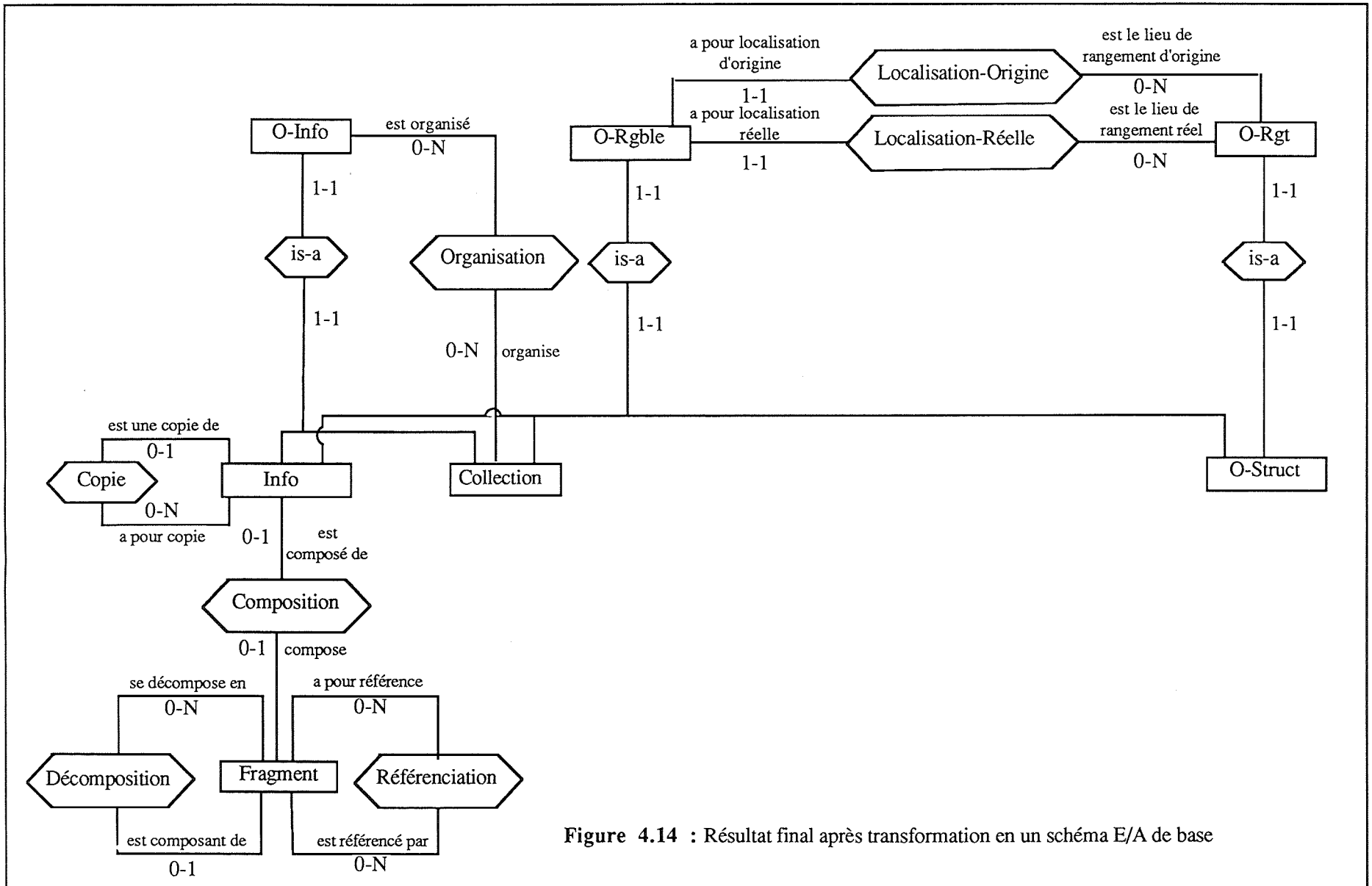

Figure 4.14 : Résultat final après transformation en un schéma E/A de base

• **Entité O-Info :**

- Nom-O-Info

  Propriétés : attribut simple, identifiant, élémentaire et obligatoire.
- Att-OB (attributs non modifiés provenant du type d'entité OB)
- Att-O-Info (attributs non modifiés du type d'entité O-Info)

• **Entité O-Rgble :**

- Aucun attribut n'a été défini pour cette entité.

• **Entité O-Rgt :**

- Nom-O-Rgt

  Propriétés : attribut simple, identifiant, élémentaire et obligatoire.
- Att-OB (attributs non modifiés provenant du type d'entité OB)
- Att-O-Rgt (attributs non modifiés du type d'entité O-Info)
- Type-O-Rgt

  Propriétés : méta-attribut simple, non identifiant, élémentaire et facultatif.

  Domaine de valeurs = {O-Rgt-Term, O-Struct}.
- Nom-O-Rgt-Term

  Propriétés : méta-attribut simple, non identifiant, élémentaire et obligatoire.

  Domaine de valeurs = {Armoire, Table, Poubelle}.

• **Entité Info :**

- Att-Info (attributs non modifiés du type d'entité Info)
- Type-Info

  Propriétés : méta-attribut simple, non identifiant, élémentaire et obligatoire.

  Domaine de valeurs = {Message, Formulaire, Document}.
- Stockage:

  Propriétés: attribut simple, non identifiant, élémentaire et facultatif.
- Lieu-Stockage

  Propriétés: attribut simple, non identifiant, élémentaire et facultatif.
- Contenu

  Propriétés : attribut simple, non identifiant, élémentaire et facultatif.
- Mode-Repr

  Propriétés : attribut simple, non identifiant, élémentaire et facultatif.

• **Entité Collection :**

- Att-Collection (attributs non modifiés du type d'entité Collection)
- Type-Collection

  Propriétés : méta-attribut simple, non identifiant, élémentaire et obligatoire.

Domaine de valeurs = {Fichier, Dossier}

- Type-Composant

Propriétés : attribut simple, non identifiant, élémentaire et facultatif.

- **Entité O-Struct :**

- Att-O-Struct (attributs non modifiés du type d'entité O-Struct)

- **Entité Fragment :**

- Att-Frag (attributs non modifiés du type d'entité Fragment)

Les types d'entités OB, Message, Formulaire, Document, Fichier, Dossier, O-Rgt-Term, Armoire, Table et Poubelle n'apparaissent plus dans le schéma E/A transformé.

3.2.2. Contraintes d'intégrité supplémentaires

La transformation du schéma E/A résulte en l'apparition de nouvelles contraintes d'intégrité appelées contraintes de structure ainsi qu'en la modification de certaines des contraintes définies sur le schéma initial. Elles sont présentées ci-dessous.

3.2.2.1. Contrainte d'identification

Les attributs compatibles Nom-O-Info et Nom-O-Rgt des types d'entités O-Info et O-Rgt constituent un identifiant global.

3.2.2.2. Contraintes de structure

Pour toute occurrence du type d'entité O-Rgt, si la valeur de l'attribut Type-O-Rgt est "O-Struct" (l'objet n'est pas un objet de rangement terminal), alors l'attribut Nom-O-Rgt-Term ne prend pas de valeur et il existe une association is-a entre les occurrences des types d'entités O-Rgt et O-Struct (désignant ainsi qu'elles représentent un même objet).

Pour toute occurrence du type d'entité Collection, si l'attribut Type-collection de la collection a la valeur "Fichier" alors l'attribut Type-composant prend une signification.

Pour toute occurrence du type d'entité Info, si l'attribut Type-info de la collection a la valeur "Document" alors cette occurrence participe à l'association Composition et seulement dans ce cas.

3.2.2.3. Contraintes d'intégrité dont l'expression a changé

Toutes les informations impliquées dans le type d'association Copie doivent avoir même valeur pour l'attribut Type-info.

4. Essai de définition d'une méthodologique de transformation de schémas

Pour transformer un schéma conceptuel des données en un schéma E/A de base, la littérature étudiée ne nous fournissait aucun critère de choix nous permettant de trancher en faveur de l'utilisation d'un type de transformation dans un contexte donné. Nous avons donc réalisé plusieurs essais successifs de transformation du schéma jusqu'à trouver la technique la plus adéquate à appliquer à chaque structure de généralisation/spécialisation. Les observations de l'utilité de ces transformations se situent à deux niveaux : tout d'abord celles relatives à la structure de généralisation/spécialisation concernée par la transformation, ensuite celles relatives aux modifications que cette transformation implique sur la totalité du schéma.

Les techniques de transformation ayant été appliquées à un exemple complet, elles nous ont permis de retirer quelques enseignements généraux relatifs à leur utilisation que nous présentons ci-dessous.

4.1. Critères généraux de sélection

Lors de la transformation d'un schéma E/A, deux critères de choix pertinents, permettant de trancher en faveur de l'application de l'une des techniques, sont rapidement apparus :

1. Si un objet du réel perçu doit être représenté dans la base de données par un type d'entité unique de manière à respecter le **principe d'unicité du réel perçu** mais aussi pour **minimiser le nombre d'accès logiques** nécessaires pour obtenir toutes les informations relatives à un objet. Il est en effet évident que plus le nombre d'entités représentant un objet est grand plus le nombre d'accès logique pour y accéder croîtra. Sur ce point, les techniques de transformation par R.T.E.G. et par R.T.E.S. semblent être les mieux adaptées.

2. Si le but est de **minimiser la multiplication des contraintes** de structures, des méta-attributs et des attributs facultatifs. La plupart de ces contraintes doivent en effet être gérées explicitement par le programmeur des modules d'accès à la base de données et certaines sont difficiles à vérifier. Dans ce cas précis, c'est la technique de M.R.is-a qui est la plus adéquate.

Malheureusement, ces deux critères de choix sont incompatibles : le choix de l'une des deux méthodes ne permettant pas de bénéficier des avantages de l'autre.

L'exemple présenté à la figure 4.15 l'illustre bien.

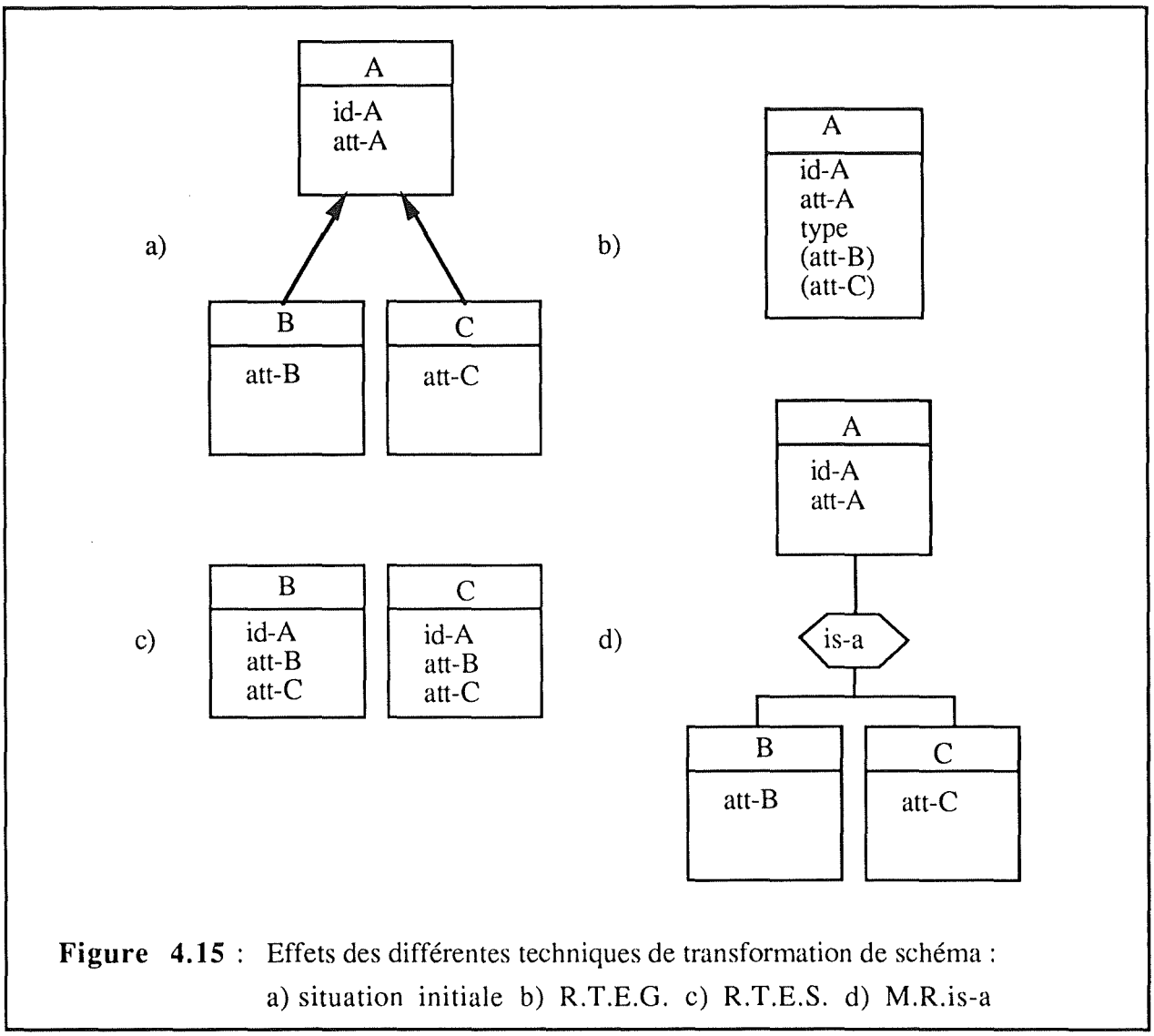

Figure 4.15 : Effets des différentes techniques de transformation de schéma : a) situation initiale b) R.T.E.G. c) R.T.E.S. d) M.R.is-a

Les techniques de transformation par R.T.E.G. (voir figure 4.15.b) et des R.T.E.S. (voir figure 4.15.c) réduisent le nombre d'accès nécessaires pour obtenir les informations relatives à B et à C à un seul accès logique alors qu'il était de deux initialement. Cependant, la technique de R.T.E.G. a pour inconvénient majeur d'introduire un méta-attribut Type ayant pour domaine de valeurs {B, C} et de rendre les attributs des sous-types initiaux B et C facultatifs. Quant à la technique par R.T.E.S., elle dédouble les attributs de A et introduit une redondance dans le schéma. De plus, si au moins un des attributs de A est identifiant, une contrainte d'identification globale devra être définie sur B et C pour cet attribut. Toutes ces contraintes devront être gérées explicitement par le programmeur qui, en contrepartie, y gagnera en temps d'accès. Par opposition, la technique par M.R.is-a n'ajoute pas de nouvelles contraintes mais augmente le nombre d'accès nécessaires pour obtenir toutes les informations relatives à B et à C (voir figure 4.15.d) car les

informations représentant une instance d'une classe d'objets sont dispersées sur plusieurs entités dans le schéma transformé.

4.2. Hiérarchie à plusieurs étages

Dans la mesure où il est possible de combiner les avantages apportés par chacune de ces trois techniques de transformation de schémas, il apparaît une tendance générale de transformation qui consiste à **réduire le nombre de niveaux dans la hiérarchie d'objets** en éliminant certains types d'entités.

Cette tendance se traduit comme suit :

1. La technique de **transformation par R.T.E.S.** est la mieux adaptée pour les types d'entités racines dans la hiérarchie. Elle permet de diminuer le nombre d'occurrences représentant un même objet dans la base de données tout en introduisant peu ou pas de contraintes supplémentaires. Ceci est dû au fait que l'héritage des attributs du type d'entité générique se fait de manière naturelle. Remarquons que la structure résultant de la transformation ne sera pas plus simple si des associations sont définies sur les objets génériques car elles devront être dédoublées sur les types d'entités spécifiques (voir figure 4.16). Dans ce cas précis, la transformation par M.R.is-a est beaucoup plus adéquate.

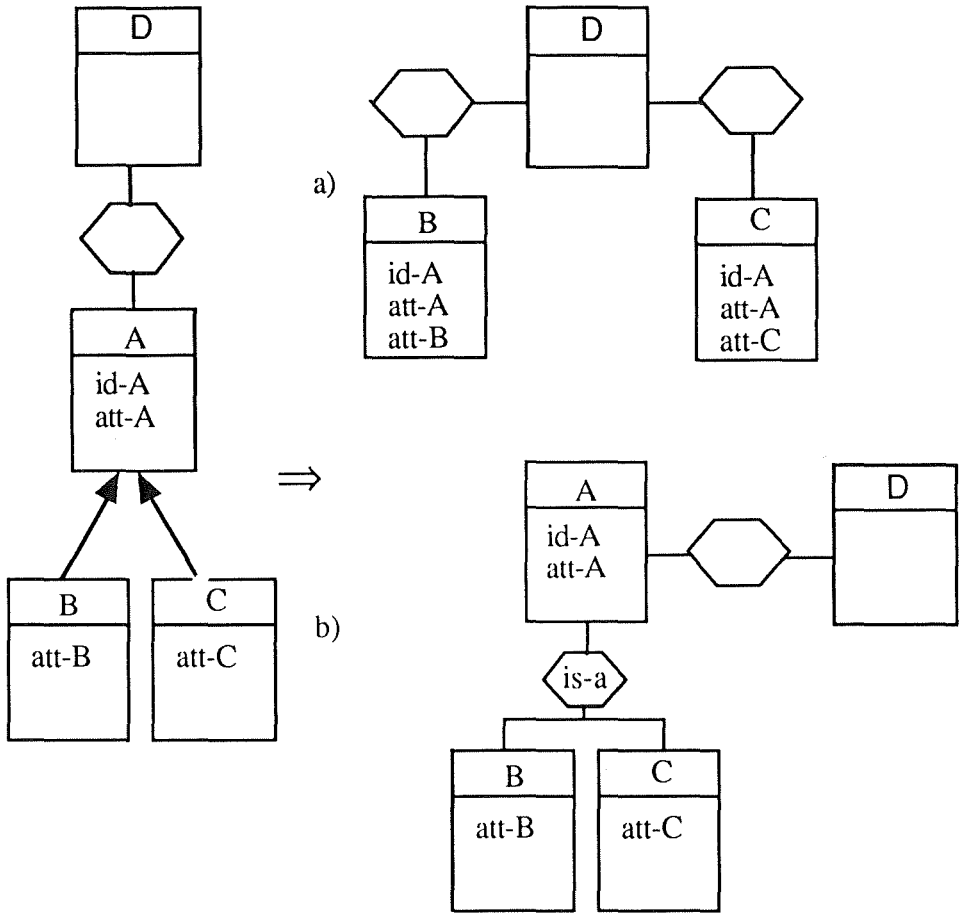

Figure 4.16 : La transformation du haut d'une hiérarchie dans le cas où un type d'association existe : a) la R.T.E.S. b) M.R.is-a : bonne transformation de remplacement

2. Pour les types d'entités spécifiques du bas du schéma (classes d'objets terminales dans la hiérarchie), la technique de **transformation par R.T.E.G**. possède l'avantage de diminuer le nombre de niveaux dans la hiérarchie et donc, d'augmenter les performances d'accès. Mais, son application doit se faire avec prudence car il faut veiller à ne pas multiplier le nombre d'attributs facultatifs dans les types d'entités génériques ainsi que le nombre de contraintes de structure. Bien que, dans l'approche orientée objets, la gestion des contraintes est facilitée par la structure modulaire des classes d'objets auxquelles sont associées les méthodes d'accès à la base de données (de là, bien localisées), tous les problèmes ne sont pas résolus pour autant et il faut faire attention lors de la conception des algorithmes des méthodes des classes d'objets ainsi transformées. De plus, l'apparition d'attributs facultatifs en trop grand nombre est bien un signe que la décomposition devait être maintenue.

Si le nombre d'attributs est faible, on a tout intérêt à appliquer cette méthode afin diminuer le nombre d'accès logiques nécessaires pour obtenir les informations relatives à un objet et ne pas porter une atteinte sérieuse aux performances d'accès à la base de données.

Par contre, si le nombre d'attributs est élevé, il faut maintenir la décomposition pour ne pas augmenter exagérément le volume de la base de données dans le cas où le S.G.B.D. ne gère pas les attributs facultatifs ou encore pour faciliter la gestion des valeurs nulles.

3. Pour les niveaux intermédiaires dans la hiérarchie, les structures de généralisation/spécialisation semblent plus faciles à modifier par la **technique de la M.R.is-a** car elle est celle des trois techniques qui introduit le moins de modifications par rapport à la structure E/A initiale des types d'entités concernés. Il faut en effet se rendre compte que toute modification de la représentation d'un type d'entité se rapporte à deux niveaux : les types d'entités pères de ce type d'entité et ses types d'entités fils. Quand les types entités concernés par la transformation sont obtenus par héritage multiple, cette technique est la seule méthode satisfaisante pour ne pas multiplier exagérément les contraintes de structure. De plus, les types d'entités de niveau intermédiaire sont souvent impliqués dans des associations et, dans ce cas, l'application d'une autre méthode que la M.R.is-a se traduirait par l'apparition de contraintes lourdes à gérer (voir figure 4.17).

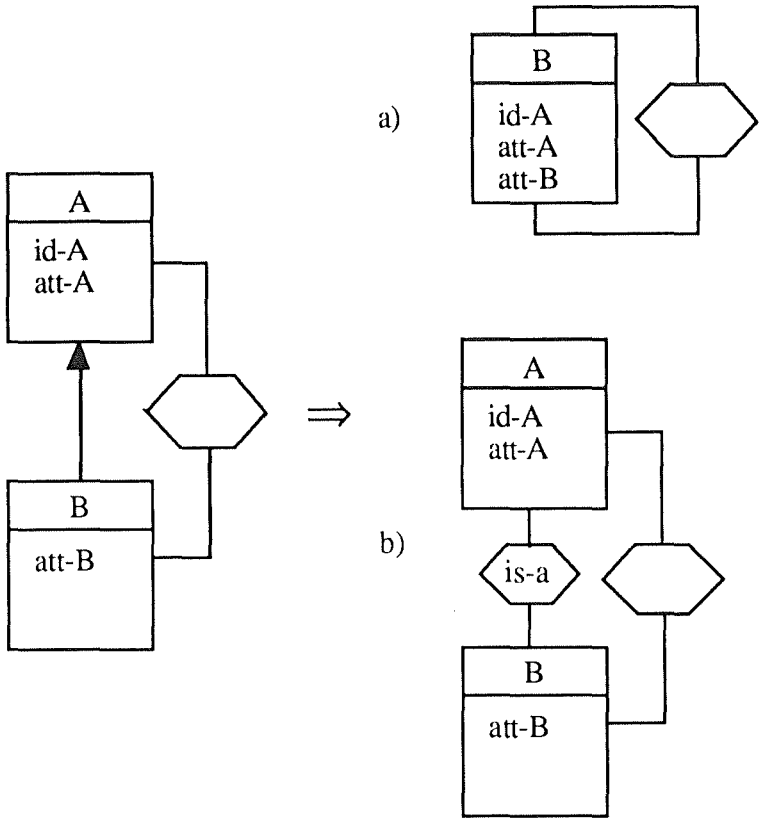

Figure 4.17 : Transformation de types d'entités de niveau intermédiaire dans le schéma : a) exemple de mauvaise transformation b) bonne transformation par M.R.is-a

4.3. Exemple d'une mauvaise transformation

Pour montrer, sur un exemple concret, les effets de l'application d'un mauvais type de transformation, nous pouvons utiliser le schéma d'un environnement de bureau. Si l'on ne tient pas compte du fait que certains objets de rangement sont aussi des objets non rangeables (ce qui complique encore les choses), la technique de transformation par R.T.E.G. appliquée à la structure d'héritage entre OB, O-Rgble et O-Rgt fournit le schéma présenté à la figure 4.18.

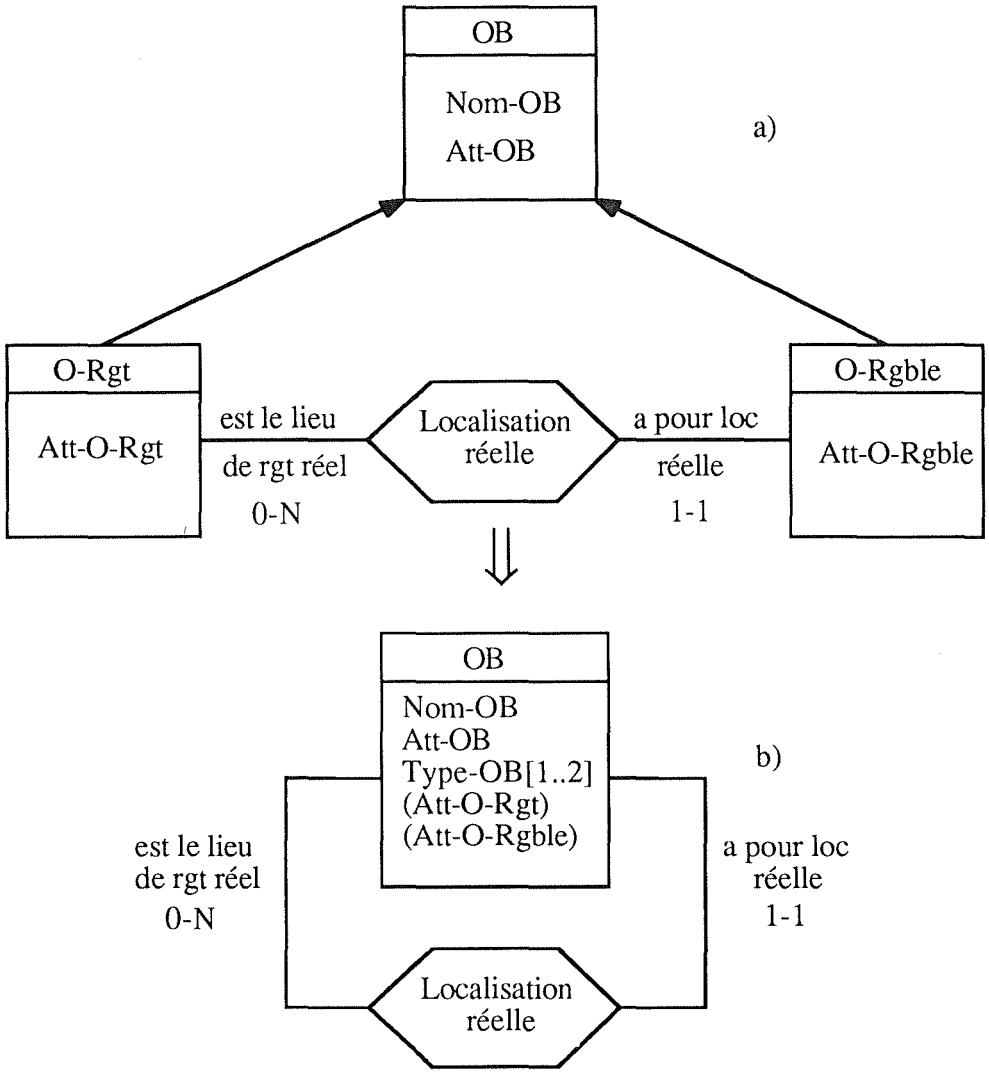

Figure 4.18 : Transformation par R.T.E.G. appliquée à la structure d'héritage définie sur les classes d'objets OB, O-Rgt et O-Rgble :
a) situation initiale b) situation finale

L'application de cette technique introduit :

- un méta-attribut Type-OB répétitif (de répétitivité maximale 2) et obligatoire ayant pour domaine de valeurs {O-Rgble,O-Rgt}[12],

[12]La répétitivité 2 est nécessaire pour pouvoir représenter les Objets de Rangement rangeables.

- des contraintes sur les attributs facultatifs telles que : si l'attribut Type-OB prend la valeur "O-Rgble" alors les attributs provenant d'O-Rgble (att-O-Rgble) doivent nécessairement prendre une valeur, etc,
- une association récursive sur laquelle sont définies trois contraintes :

  Une entité du type d'entité OB ne peut être impliquée dans une association Localisation-Réelle avec elle-même.
  Une entité du type d'entité OB ne peut jouer le rôle "est le lieu de rgt réel" que si son attribut Type-OB prend pour valeur "O-Rgt".
  Un entité du type d'entité OB ne peut jouer le rôle "a pour loc réelle" que si son attribut Type-OB prend la valeur "O-Rgble".

Les contraintes à gérer sont lourdes par rapport au résultat qui avait été obtenu par la technique de représentation des types d'entités spécifiques. L'existence d'un type d'association récursive possédant de telles contraintes traduit l'existence d'une représentation du réel perçu définie sur un objet qui serait un agrégat de trop haut niveau. Par contre, son expression non récursive sur les objets descendants permet de gérer naturellement les contraintes définies ci-dessus en les rendant inhérentes à la nature des objets.

Lors de la transformation d'un schéma E/A, il apparaît qu'il n'existe pas de critères clés pour choisir la transformation adéquate. Les transformations appliquées dépendent des objectifs que s'est fixé le concepteur.

5. Conception logique du schéma de base de données

Après avoir traduit le schéma conceptuel des données d'un environnement de bureau en schéma E/A de base, il reste à en dériver le **schéma des accès possibles** et le **schéma des accès nécessaires**, qui sont deux phases importantes de la conception logique du logiciel.

Le **schéma des accès possibles** traduit le schéma E/A de base dans le modèle d'accès généralisé (MAG). Les règles utilisées pour effectuer cette traduction sont décrites dans [HAIN-86]. Ce schéma ne tient évidemment aucun compte des performances d'accès que l'utilisateur est en droit d'attendre de la boîte à outils. Il permet simplement de faire un pas vers la conception physique en procurant les structures nécessaires à l'étape suivant de la phase de conception logique, à savoir le schéma des accès nécessaires.

Le **schéma des accès nécessaires** est construit à partir du schéma des accès possibles et des méthodes offertes à l'utilisateur afin qu'il puisse manipuler les classes d'objets définies lors de l'analyse conceptuelle d'un environnement de bureau. Ces méthodes ont été présentées lors de description des interfaces des classes d'objets (voir section 3.5 "Réutilisabilité de la hiérarchie étendue d'objets de bureau"). Chaque méthode définie permet de dégager du schéma des accès possibles les structures de données nécessaires pour son bon déroulement. Le rassemblement des structures de données nécessaires à toutes les méthodes fournit le schéma des accès nécessaires. A ce niveau, les structures spécifiques permettant d'améliorer les performances d'accès aux données (comme la redondance d'une certaine partie du schéma par exemple) n'interviennent pas encore.

5.1. Production du schéma des accès possibles

La traduction du schéma E/A de base en structures conformes au MAG ne pose guère de problèmes. En effet, ce schéma E/A de base est déjà, en quelque sorte, une normalisation du schéma conceptuel d'un environnement de bureau puisqu'il ne présente plus de structures de généralisation/spécialisation. L'absence de relations d'héritage facilite la traduction en structures MAG car le concept de relation is-a n'est pas représentable en termes de structures du MAG.

Un seul endroit du schéma doit subir une transformation pour être conforme aux structures du MAG : c'est le type d'association Décomposition qui possède un attribut appelé Num-séq. Pour résoudre ce problème, ce type d'association est supprimé en faisant migrer ses attributs vers le type d'entité Fragment (voir figure 4.19). La technique employée n'est pas la technique la plus générale qui consiste à transformer le type d'association en un type d'entité mais elle est particulièrement indiquée dans ce cas précis vu les connectivités du rôle "est composant de" car elle n'introduit pas un

type d'entité supplémentaire. De ce fait, l'attribut Num-séq devient un attribut du type d'entité Fragment. C'est un attribut simple (car la connectivité maximale du rôle "est composant de" est 1), élémentaire, facultatif (car la connectivité minimale du même rôle est nulle) et non identifiant.

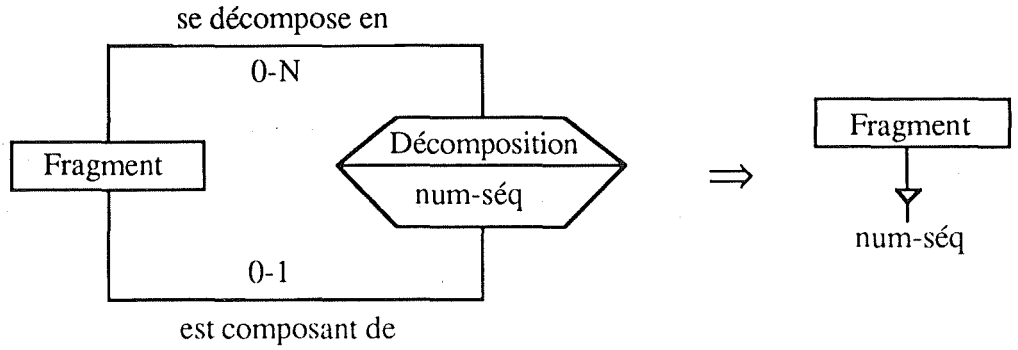

Figure 4.19 : Transformation du type d'association Décomposition en une structure conforme au MAG

Le reste du schéma se traduit aisément en structures du MAG. On note cependant l'attribut Mot-clé du type d'entité O-Info qui se démarque des autres attributs présents dans le schéma par sa répétitivité. Il semble bon de noter cette propriété car elle sera à la base d'une transformation importante qui fera suite à une décision d'implémentation lors de la phase de la conception physique. Le schéma des accès possibles est présenté aux figures 4.20.a et 4.20.b.

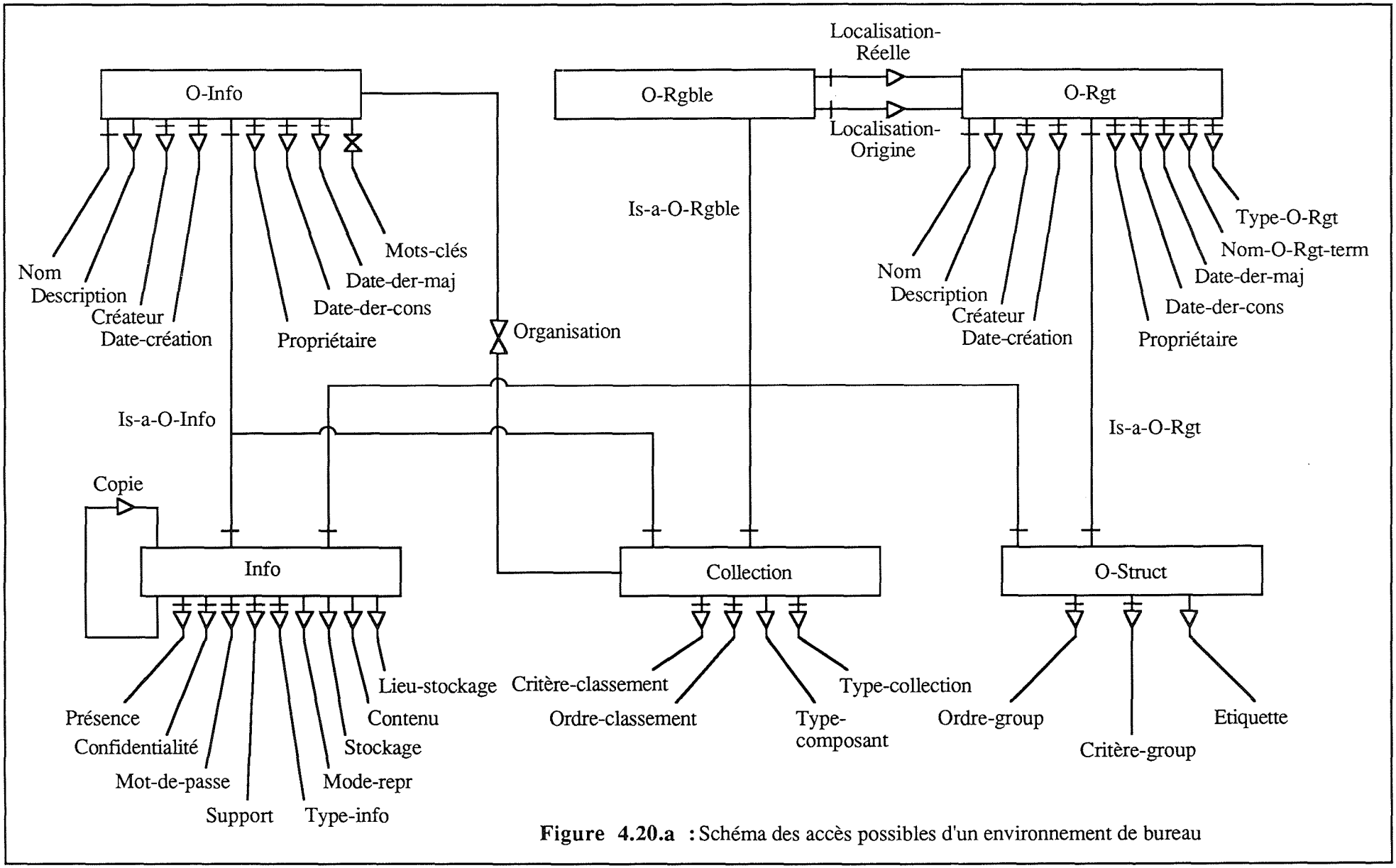

Figure 4.20.a : Schéma des accès possibles d'un environnement de bureau

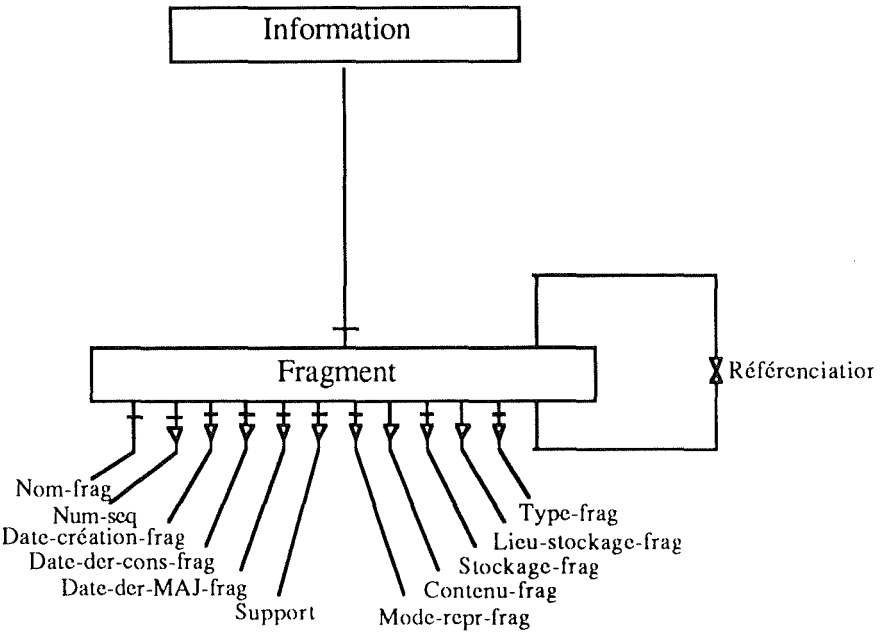

Figure 4.20.b : Schéma des accès possibles pour la découpe en fragments

5.2. Production du schéma des accès nécessaires

Lors des spécifications des primitives définies pour les différentes classes d'objets de l'environnement de bureau, les parties du schéma des données et les accès nécessaires à l'exécution de chaque méthode ont pu être établis. Une fois rassemblés pour toutes les méthodes de toutes les classes d'objets, ils fournissent le schéma des accès nécessaires.

Le schéma des accès nécessaires est présenté aux figures 4.21.a et 4.21.b. On notera que les flèches vers les différents attributs des diverses types d'entités sont implicites. On notera également les clés d'accès suivantes :

- Nom-O-Info et Mot-clé pour le type d'article O-Info,
- Nom-O-Rgt pour le type d'article O-Rgt,
- Nom-Frag pour le type d'article Fragment.

Il est également intéressant de remarquer que tous les chemins d'accès sont dotés d'un chemin inverse.

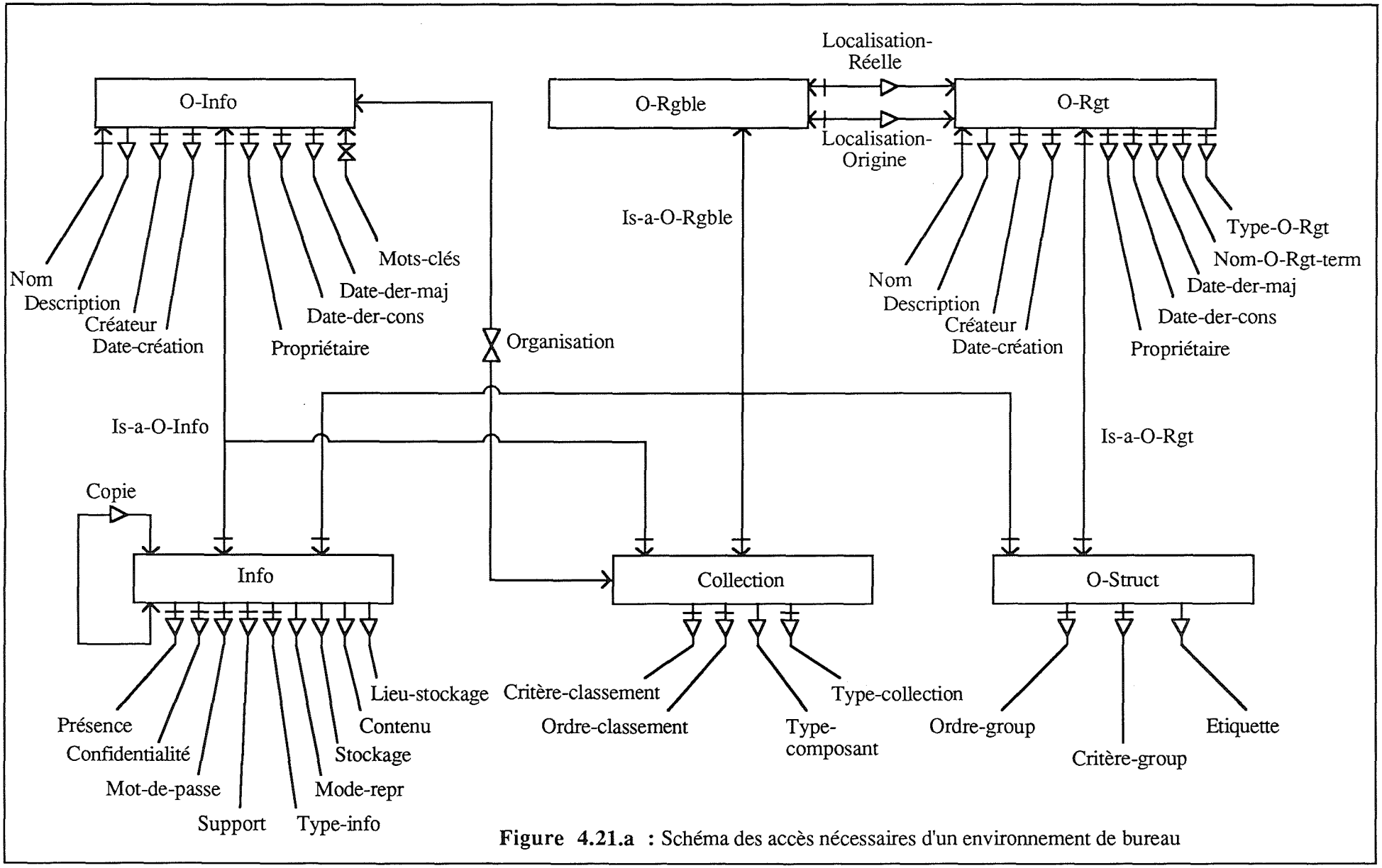

Figure 4.21.a : Schéma des accès nécessaires d'un environnement de bureau

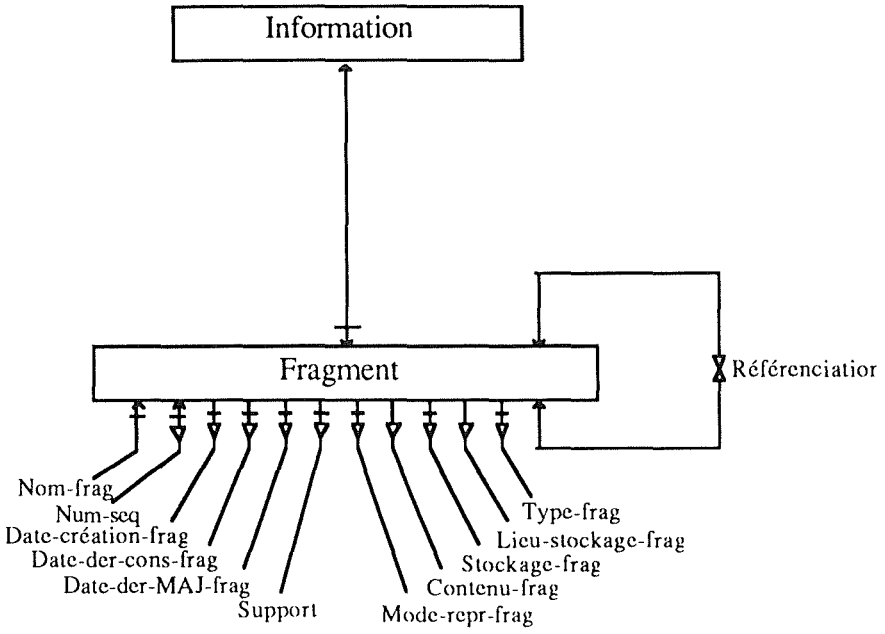

Figure 4.21.b : Schéma des accès nécessaires pour la découpe en fragments

Chapitre 5 :
Conception physique de la boîte à outils d'objets bureautiques

Introduction

Après la phase de conception logique de la boîte à outils d'objets bureautiques, il reste à réaliser sa **conception physique**. Le but de la phase de conception logique était de produire une solution correcte qui soit exécutable sur une machine abstraite. Au cours de cette phase, les principales décisions de conception étaient prises, elles étaient donc indépendantes de toute décision d'implémentation, à savoir le système de gestion de base de données (S.G.B.D.) qui va devoir supporter la base de données et l'environnement de programmation sur lequel la solution tournera.

Lors de la phase de conception physique, une solution tournant sur une machine concrète avec un S.G.B.D. donné va être produite. De même, des décisions permettant d'adapter la solution abstraite à un environnement de programmation donné seront prises. Un regard sera également porté du côté des performances : diverses solutions visant à diminuer le nombre d'accès physiques seront envisagées et celle étant la plus efficace sera conservée.

Pour adapter la solution logique présentée au chapitre 4 "Conception logique de la boîte à outils d'objets bureautiques" à l'environnement de programmation mis à notre disposition, à savoir N.D.B.S. et TURBO PASCAL version 5.5, deux types de transformation de cette solution logique seront effectuées. Premièrement, le schéma des accès nécessaires sera modifié afin de produire un schéma conforme à N.D.B.S, ce sera l'objet de la section **Transformation du schéma de la base de données**. Deuxièmement, le schéma des classes d'objets proposé à la section 2.2 "Présentation d'un environnement de bureau étendu" comportait des structures d'héritage multiple. Il doit donc subir des transformations afin d'obtenir une hiérarchie de classes conforme à TURBO PASCAL 5.5 qui ne supporte que les structures d'héritage simple. Ces transformations seront décrites dans la deuxième section **Transformation du schéma de la hiérarchie d'objets**.

Enfin, suite aux transformations que nous avons dû effectuer sur le schéma de classes d'objets bureautiques, nous essayerons de dégager une **méthodologie de transformation de hiérarchies de classes d'objets** permettant la conversion des structures d'héritage multiple non représentées dans certains langages de programmation orientés objets en structures d'héritage simple.

1. Transformation du schéma de la base de données

1.1. Règles de conformité d'un schéma MAG à N.D.B.S.

N.D.B.S. est un système de gestion de base de données de type réseau qui permet de développer des applications en langage Pascal sur des micro-ordinateurs de type IBM Personnal Computer et machines compatibles avec ce standard [HAIN-87].

N.D.B.S. ne permet pas d'implémenter toutes les structures supportées par le MAG. Pour pouvoir être décrit et implémenté grâce à ce S.G.B.D., un schéma MAG doit respecter certaines règles de conformité qui sont les suivantes :

1) un seul identifiant par type d'article,

2) toute clé d'accès est identifiante,

3) les contraintes d'existence ne sont pas gérées par le S.G.B.D., elles doivent donc être prises en charge par le programmeur qui doit les implémenter explicitement,

4) les items facultatifs sont interdits,

5) les chemins N-N sont interdits,

6) tout chemin est doté de son inverse.

Il est également intéressant de noter que les chemins 1-1, 1-N et N-1 sont gérés, de même que les attributs décomposables et les attributs répétitifs. De plus, les chemins récursifs et les chemins multidomaines sont admis. Ces deux dernières caractéristiques qui sont utilisées abondamment dans notre schéma MAG, sont d'un grand intérêt pour notre application qui ne devra dès lors, subir que des transformations de conformité mineures.

1.2. Production d'un schéma conforme à N.D.B.S.

1.2.1. Justification des transformations

Tous les items facultatifs décrits dans le schéma des accès nécessaires (voir partie 4.5.2 "Production du schéma des accès nécessaires") deviennent des items obligatoires. C'est notamment

le cas des attributs devenus facultatifs lors de la transformation en un schéma E/A de base. Pour garder la souplesse du modèle initial, on utilisera des valeurs interprétées comme des valeurs nulles pour pouvoir gérer le caractère facultatif initial des items.

L'élimination des chemins de classe fonctionnelle N-N est effectuée par application de la technique de création d'un type d'article [HAIN-86]. La transformation du chemin Organisation est illustrée à la figure 5.1.

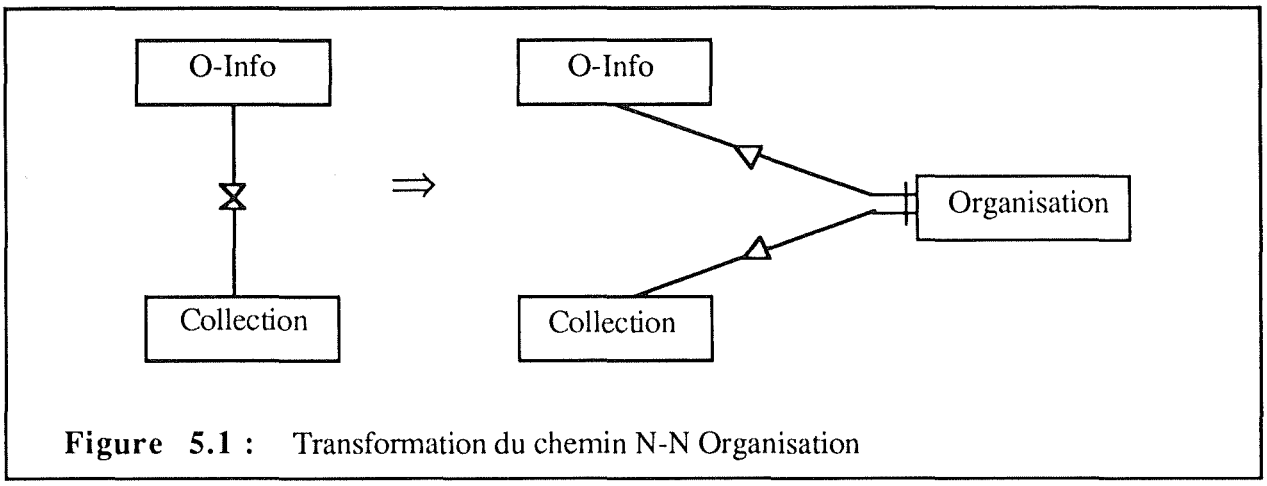

Figure 5.1 : Transformation du chemin N-N Organisation

Le chemin Référenciation possédant également une connectivité N-N, une transformation similaire à la transformation précédente lui est appliquée (voir figure 5.2).

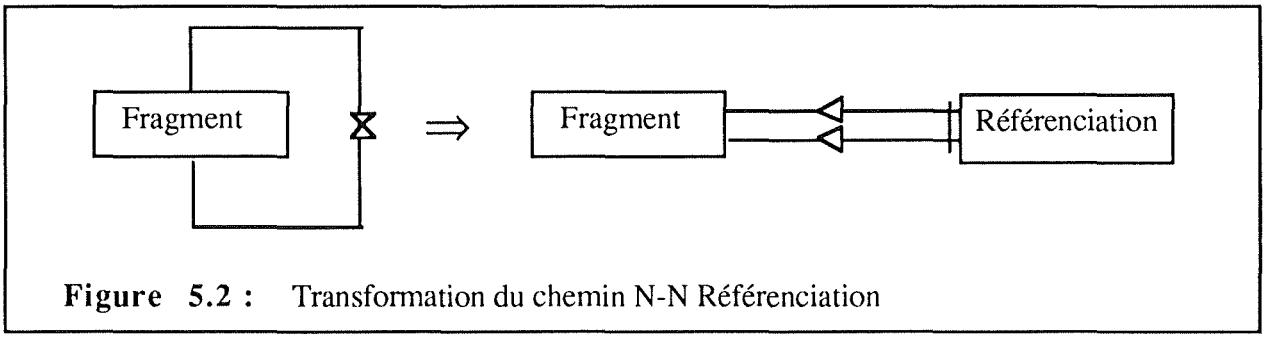

Figure 5.2 : Transformation du chemin N-N Référenciation

Les autres chemins du schéma, à savoir les chemins Localisation-Réelle et Localisation-Origine ne nécessitent aucune transformation pour être rendus conformes à N.D.B.S. Le chemin récursif Copie n'est pas modifié non plus puisque cette classe fonctionnelle est acceptée par N.D.B.S.

Bien que les items répétitifs soient admis par N.D.B.S., l'item Mots-clés a été transformé de manière à pouvoir constituer un thesaurus de mots clés. Cette transformation est utile pour faciliter l'implémentation des méthodes relatives aux objets informationnels impliquant des accès vers les mots clés de ces objets informationnels.

Le résultat final des transformations subies par l'item Mots-clés est illustré à la figure 5.3. Deux étapes sont nécessaires pour l'atteindre : transformation d'un type d'item en un type d'article, suivi de la transformation du chemin N-N pour être conforme à N.D.B.S.

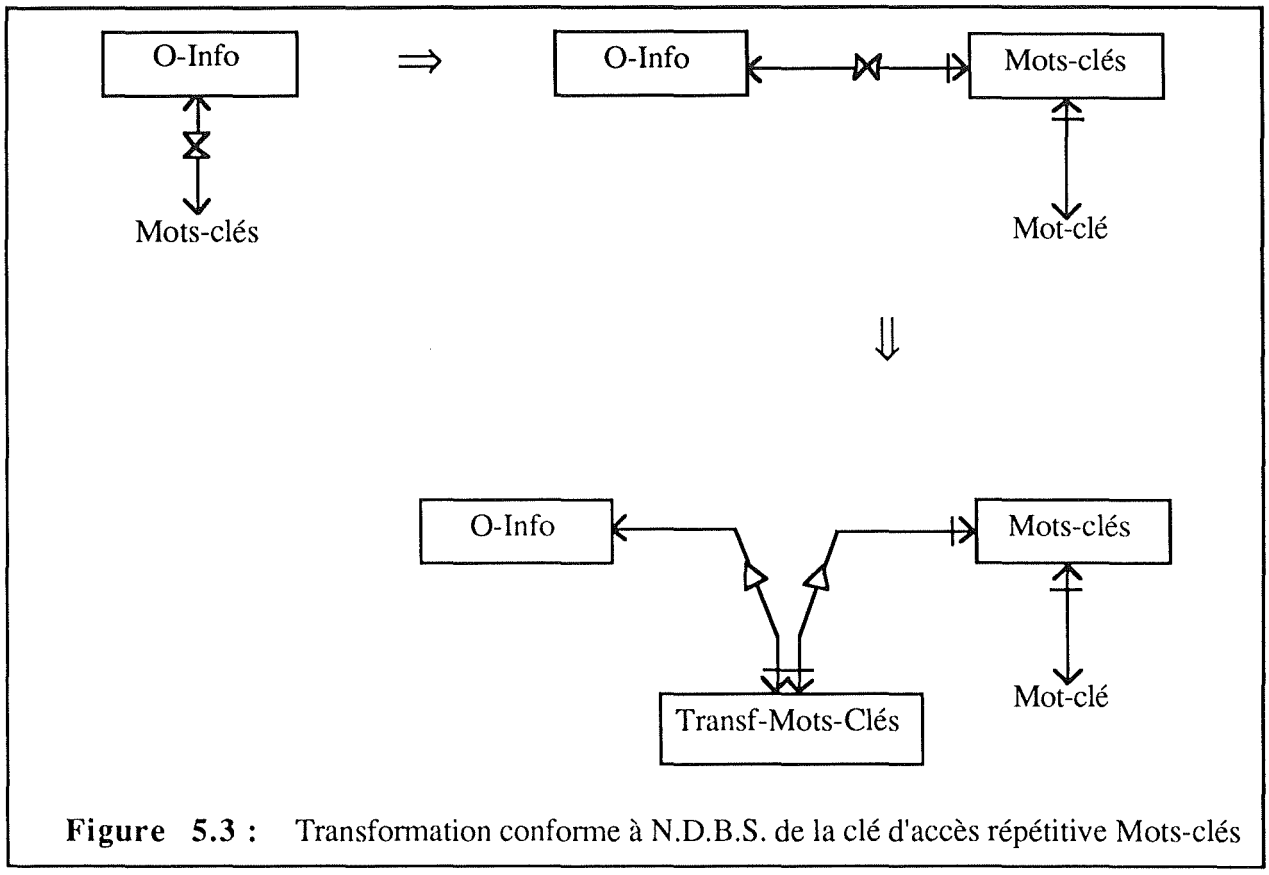

Figure 5.3 : Transformation conforme à N.D.B.S. de la clé d'accès répétitive Mots-clés

L'item Mot-clé identifie le type d'article Mots-Clés et peut donc être utilisé comme clé d'accès à ce type d'article. De ce fait, via le type d'article Trans-Mots-Clés résultant de cette transformation, il est facile d'accéder à toutes les instances de la classe O-Info correspondant à un mot clé particulier.

Finalement, toutes les contraintes d'existence devront être gérées explicitement par les méthodes des objets (modules d'accès à la base de données).

1.2.2. Présentation du schéma

Le schéma final obtenu est représenté à la figure 5.4. Le formalisme graphique utilisé est propre à N.D.B.S. mais proche de celui employé par le modèle Entité/Association. Chaque type d'article est représenté par son nom et les noms de ses items, les identifiants sont soulignés et les chemins sont représentés par des arcs reliant les boîtes représentant les types d'articles.

Lors de la phase de conception physique d'un logiciel sont prises les décisions permettant, entre autres, d'améliorer les performances de l'application. Les choix d'implémentation qui ont été faits pour l'application que nous devons développer sont les suivants :

1) Le chemin multidomaine représentant la relation de généralisation/spécialisation entre le type d'article O-Rgble et ses types spécifiques (se référer à la figure 4.21.a) est éliminé et remplacé par la duplication de la totalité des attributs de l'objet concerné dans le type d'article O-Rgble. Ainsi, la réalisation de deux accès logiques supplémentaires nécessaires pour obtenir les noms des objets rangeables lors des accès du type d'article O-Rgt vers le type d'entité O-Rgble via les chemins Localisation-Réelle et Localisation-Origine est évitée. De plus, il ne faut pas perdre de vue les concepts de réutilisabilité et d'extensibilité que doit posséder le logiciel final. La duplication de l'entièreté des attributs entraîne une réutilisabilité quasi totale des méthodes définies pour la classe d'objets O-Rgble dans le cas où l'utilisateur final du logiciel déciderait l'implémentation d'une nouvelle classe d'objets qui hériterait uniquement des propriétés de la classe d'objets O-Rgble.

2) Un attribut Niveau-frag est ajouté au type d'article Fragment. Sa description est la suivante :

Définition : cet attribut fournit le numéro du niveau de décomposition auquel appartient un fragment dans l'arbre des fragments d'un document.
Propriétés : attribut simple, élémentaire, obligatoire, non identifiant.
Représentation : entier.

Cet attribut permet de construire beaucoup plus facilement les algorithmes de toutes les méthodes de construction et de modification définies sur les fragments car il élimine un nombre considérable d'accès via les chemins Est-organisé-en et Organise ayant pour unique but d'obtenir ce renseignement sur les fragments.

Les contraintes d'existence à gérer explicitement s'énoncent comme suit :

- toute occurrence du type d'article Information ou du type d'article Collection est associée à une occurrence du type d'article O-Info via le chemin Is-a-O-Info et inversement;
- toute occurrence du type d'article Rep-O-Mot-Clé est associée à une occurrence du type d'article O-Info via le chemin O-I-Rep;
- toute occurrence du type d'article Rep-O-Mot-Clé est associée à une occurrence du type d'article Mots-Clés via le chemin Rep-M-O et inversement;
- toute occurrence du type d'article O-Struct est associée à une occurrence du type d'article O-Rgt via le chemin Is-a-O-Rgt. Notons que la contrainte de structure sur les objets de

rangement impose aussi que l'occurrence du type d'article O-Rgt concernée possède pour valeur de l'item Type-O-Rgt la valeur "O-Struct".

- toute occurrence du type d'article Organisation est associée à une occurrence du type d'article O-Info via le chemin Est-organisé-en et à une occurrence du type d'article Collection via le chemin Organise.

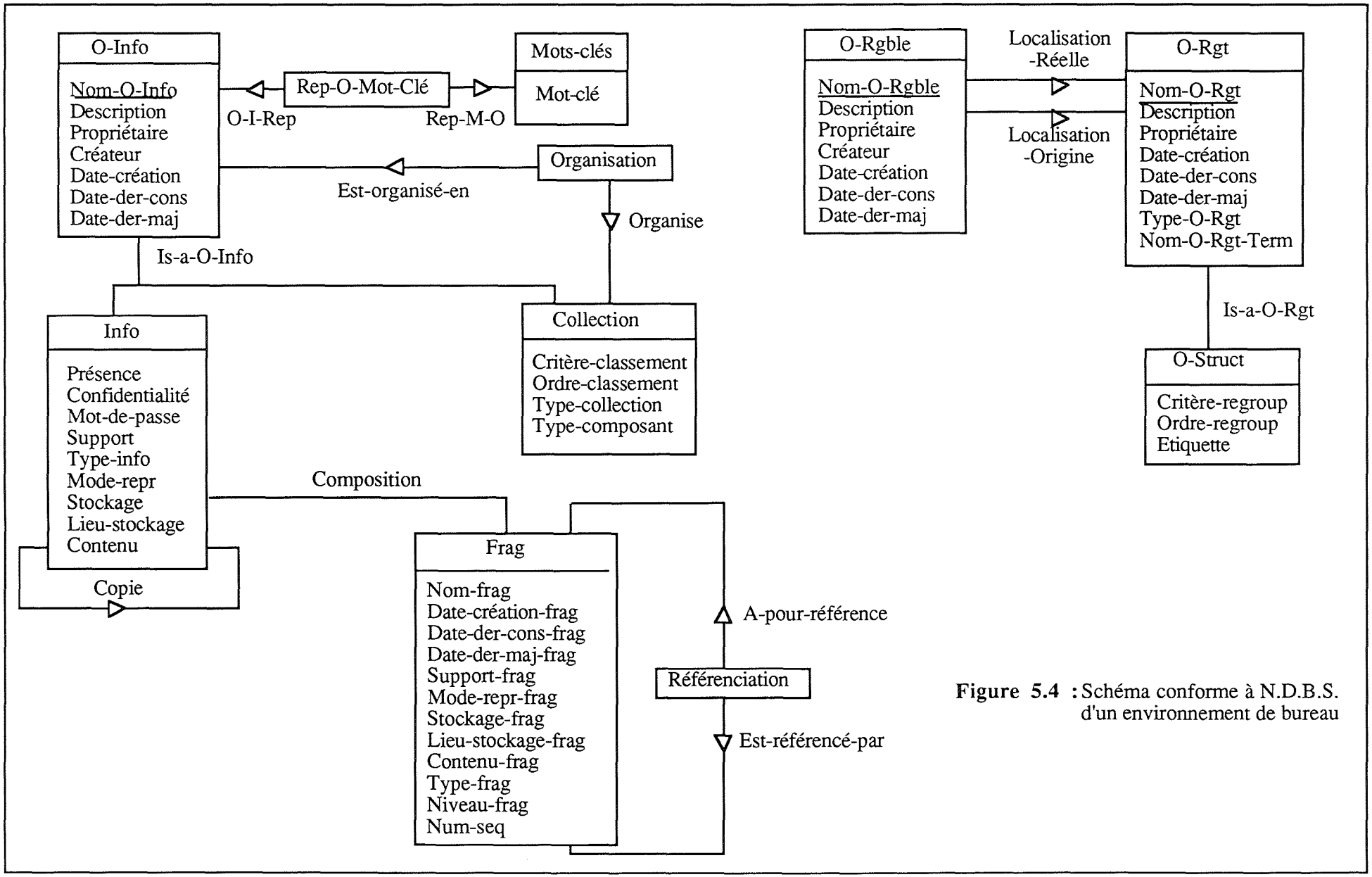

Figure 5.4 : Schéma conforme à N.D.B.S. d'un environnement de bureau

2. Transformation du schéma de la hiérarchie d'objets

La transformation de la hiérarchie d'objets en une hiérarchie conforme à TURBO PASCAL version 5.5 se résume essentiellement à la traduction de toutes les structures d'héritage multiple utilisées pour la description du schéma des classes d'objets d'un environnement de bureau en des structures d'héritage simple, puisque ces structures d'héritage multiple ne sont pas supportées par TURBO PASCAL version 5.5. Ces modifications portent sur les classes dérivées de la classe OB issues des critères de partitionnement qui ont été définis.

Les sous-classes héritant directement des propriétés de la classe OB (à savoir les classes résultant directement des critères de partitionnement : O-Info, O-Rgt, O-Rgble et O-Non-Rgble) doivent être transformées en une hiérarchie à plusieurs niveaux comportant uniquement des structures d'héritage simple construites de manière à fournir à leurs classes dérivées toutes les propriétés qu'elles leur apportaient initialement grâce aux structures d'héritage multiple. Pour effectuer ces transformations, il n'existe aucune théorie décrite dans la littérature : les transformations les plus adéquates ont donc été appliquées en essayant de respecter les principes de réutilisabilité et d'extensibilité que se doit de posséder l'application finale.

Dans un premier temps, il est à remarquer que toutes les classes d'objets héritières de la classe d'objets O-Info c'est-à-dire, les classes d'objets Info et Collection, sont aussi héritières de la classe d'objets O-Rgble, cette dernière peut donc devenir une super-classe de la classe O-Info (voir figure 5.5).

Cette transformation introduit une restriction sur la richesse sémantique du schéma initial dans lequel on aurait pu créer une sous-classe de la classe O-Info qui aurait été composée d'objets informationnels non rangeables. Cette création n'est pourtant pas impossible mais, pour cela, il faudra recourir à l'inhibition des propriétés de la classe O-Rgble au niveau de la classe d'objets nouvellement créée ce qui alourdit quelque peu le travail à fournir.

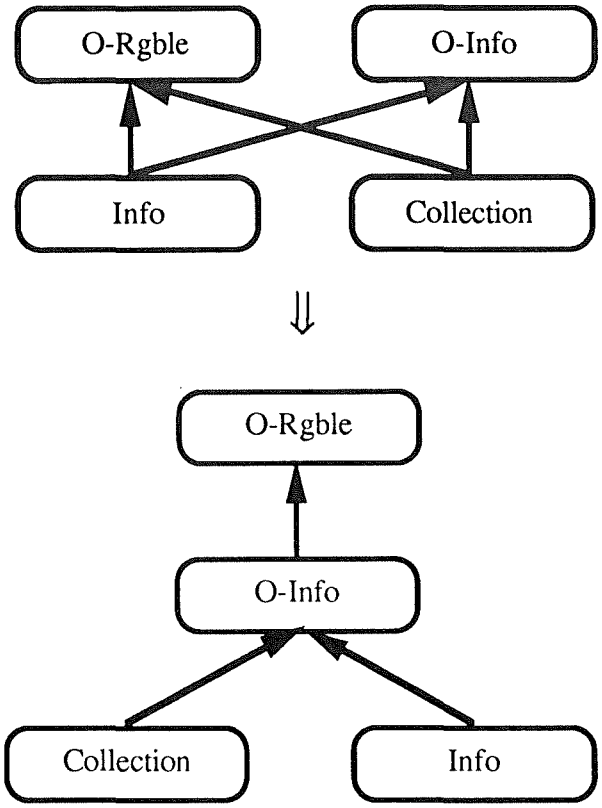

Figure 5.5 : Illustration du passage à une hiérarchie d'héritage simple pour la classe d'objets O-Info

De même, les classes dérivées de la classe O-Non-Rgble ne subissent aucune modification si l'on fait abstraction de leur héritage provenant de la classe O-Rgt. Pour les mêmes raisons que précédemment, ceci implique que la classe O-Rgt doit devenir une super-classe de la classe O-Non-Rgble (voir figure 5.6).

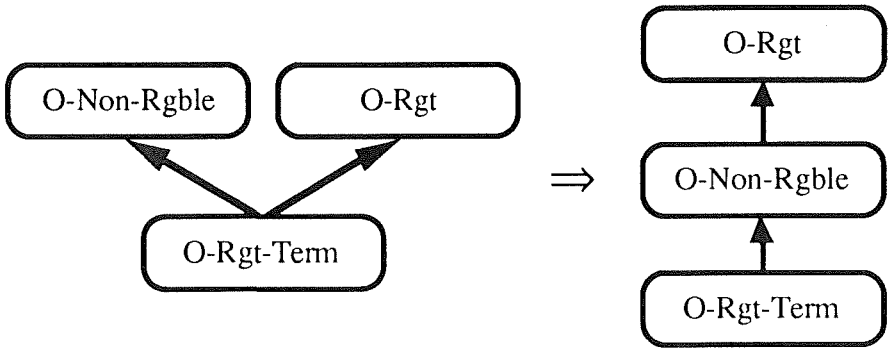

Figure 5.6 : Illustration du passage à une hiérarchie d'héritage simple pour la classe d'objets O-Non-Rgble.

Pour la suite de la traduction, plusieurs solutions sont envisageables et aucune d'entre elles ne paraît entièrement satisfaisante. Du fait qu'un objet de rangement est soit rangeable, soit non rangeable, on pourrait faire de cette classe une super-classe des classes O-Rgble et O-Non-Rgble. Cependant, la difficulté provient du fait que les objets informationnels qui sont eux-mêmes des objets rangeables ne sont pas des objets de rangement (structure de partition). Il faut aussi prendre en compte les objets de rangement structurants qui sont une sous-classe des classes O-Rgble et O-Rgt.

La solution la plus logique semble être de faire de la classe O-Rgt une super-classe des classes O-Rgble et O-Non-Rgble en réalisant une inhibition des propriétés de la classe O-Rgt au niveau de la classe O-Rgble. Ainsi, l'introduction de la classe O-Struct en tant que sous-classe de la classe O-Rgble est immédiate et il suffit de réactiver les propriétés de la classe O-Rgt pour que cette classe possède bien toutes les propriétés propres aux objets de rangement rangeables. Finalement, la classe OB devient une super-classe de la classe O-Rgt (voir figure 5.7).

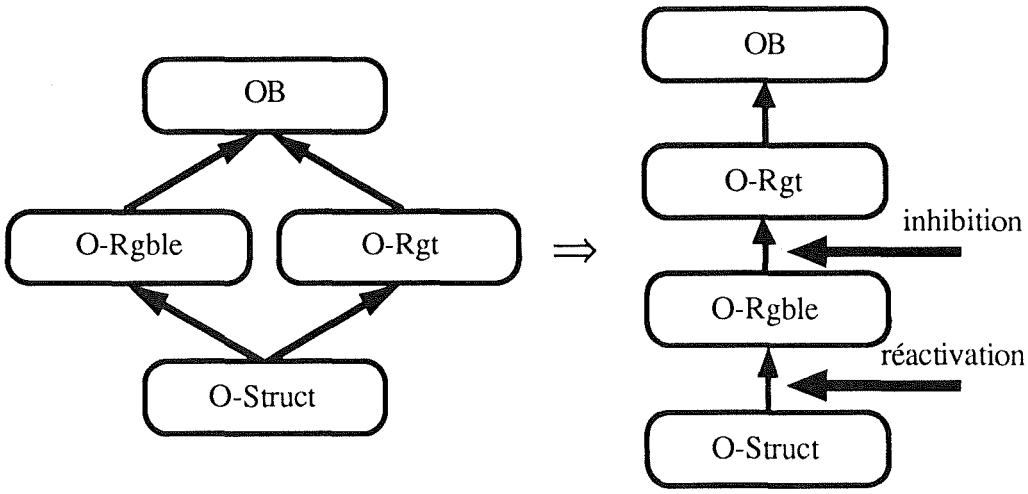

Figure 5.7 : Illustration du passage à une hiérarchie d'héritage simple pour la classe d'objets O-Non-Rgble.

Pour terminer la transformation de la hiérarchie des classes d'objets d'un environnement de bureau, il nous reste à traduire la classe Fragment. Comme celle-ci est indépendante de la hiérarchie des objets de bureau, sa traduction est immédiate.

Le schéma final est présenté à la figure 5.8, c'est ce schéma qui est implémentable en Turbo Pascal.

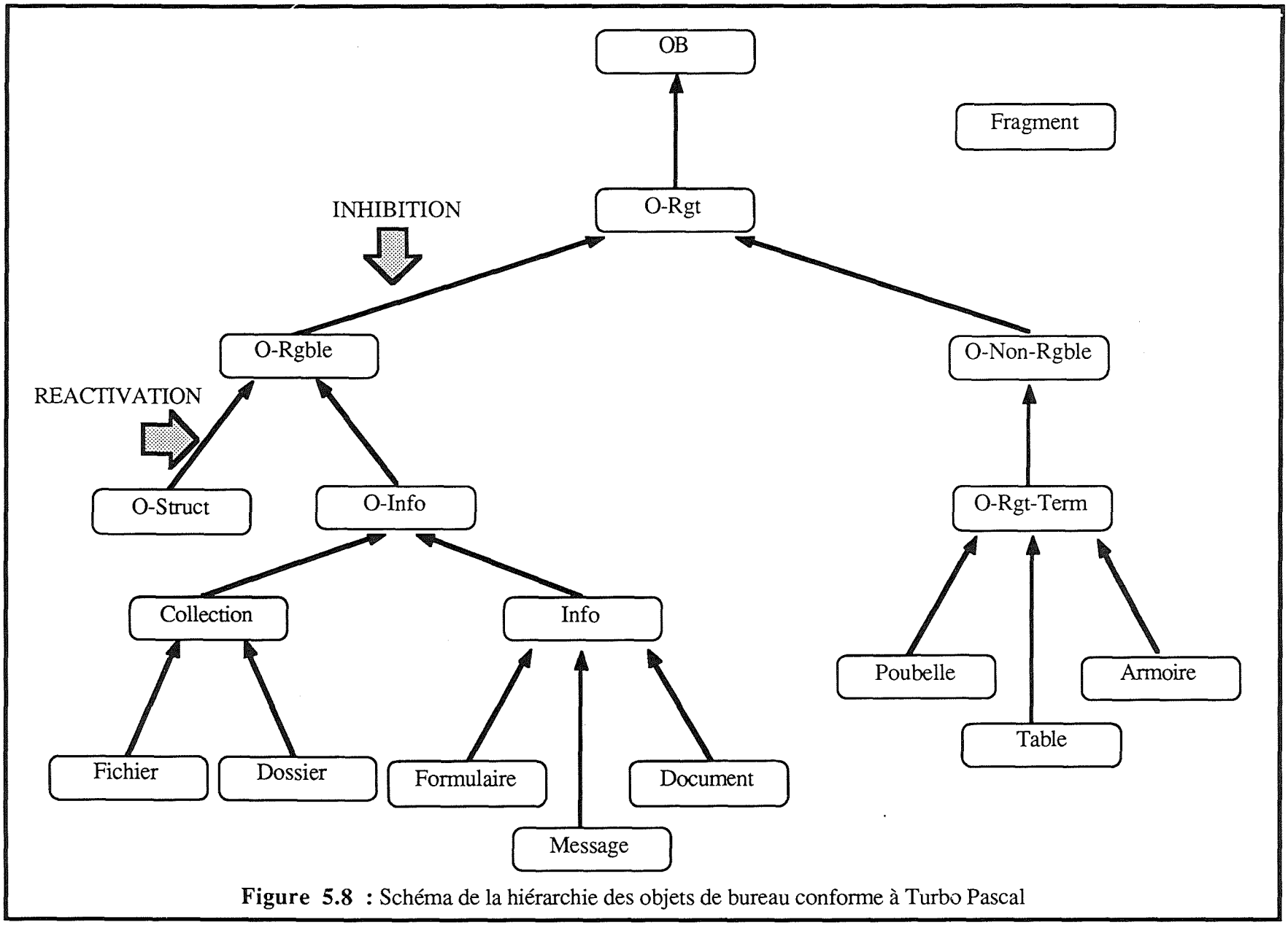

Figure 5.8 : Schéma de la hiérarchie des objets de bureau conforme à Turbo Pascal

3. Méthodologie de transformation de hiérarchies de classes d'objets

Après la transformation de la hiérarchie de classes d'objets bureautiques comportant des structures d'héritage multiple en une hiérarchie de classes d'objets définie entièrement sur des structures d'héritage simple, il semble intéressant de retirer de ces transformations une méthodologie de transformation de hiérarchies de classes d'objets permettant l'élimination de structures d'héritage multiple.

3.1. Premier cas de transformation

La première transformation que l'on peut présenter est illustrée à la figure 5.8. Dans cet exemple, les classes d'objets A et B sont tout à fait quelconques. Deux solutions symétriques sont alors possibles. Dans la première solution, la classe A devient une classe de base de la classe B : une inhibition des propriétés de la classe A au niveau de la classe B doit alors être réalisée et une réactivation de ces propriétés doit se faire au niveau de la classe C. La deuxième solution se justifie exactement de la même manière que la première. Le choix entre ces deux solutions dépend alors de l'environnement de ces classes dans la totalité de la hiérarchie.

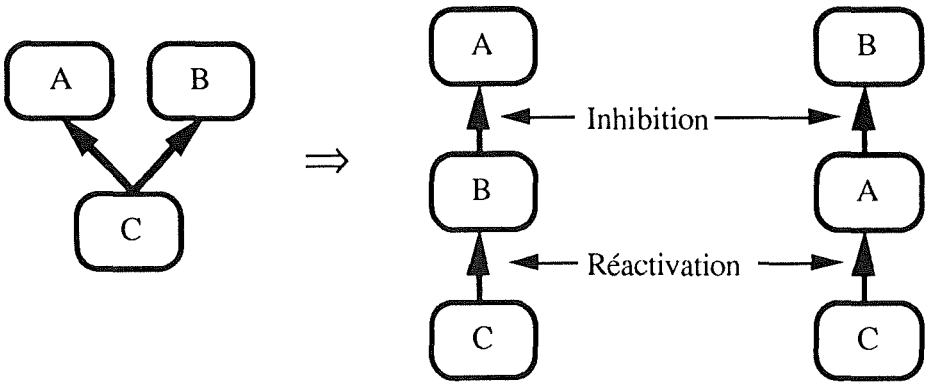

Figure 5.8 : Illustration du premier exemple de transformation

3.2. Deuxième cas de transformation

Le deuxième type de transformation que nous avons pu étudier lors de la transformation du schéma de classes d'objets bureautiques en hiérarchie de classes d'objets conforme au langage de programmation TURBO PASCAL version 5.5 est présenté à la figure 5.9. Dans ce schéma de

classes d'objets qui est en fait un cas particulier du premier exemple de transformation, toutes les instances de la classe d'objets B sont également des instances de la classe d'objets A. Ceci peut être le cas lorsque les classes A et B résultent de deux critères de partitionnement différents d'une seule et même classe d'objets. Pour une transformation correcte, il suffit de faire de la classe A une classe de base de la classe B. Ainsi, la classe d'objets C possède bien les propriétés des classes d'objets A et B.

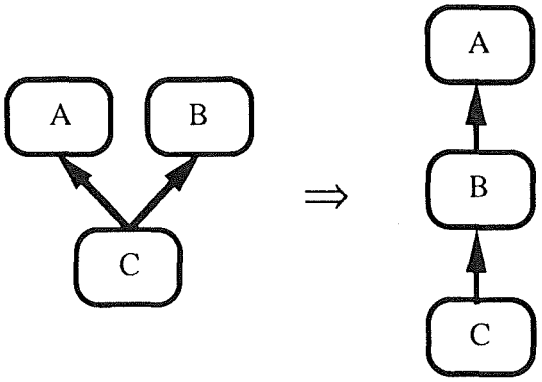

Figure 5.9 : Illustration du deuxième exemple de transformation

Chapitre 6 :
Implémentation de la boîte à outils d'objets bureautiques

Introduction

Suite à l'analyse conceptuelle d'un environnement de bureau, nous avions élaboré une solution correcte exécutable sur une machine abstraite lors de la phase de conception logique (voir chapitre 4 "Conception logique de la boîte à outils d'objets bureautiques").

Ensuite, cette solution a été adaptée à l'environnement de programmation cible de façon à obtenir une solution tournant sur une machine concrète. Cette transformation faisait l'objet du chapitre 5 "Conception physique de la boîte à outils d'objets bureautiques").

Après la phase de conception physique vient la phase d'implémentation proprement dite. Nous allons donc, dans ce chapitre, développer les points qui nous ont paru intéressants durant cette phase de notre travail.

Dans la première section, nous parlerons des différents **sous-systèmes** développés. Il est en effet intéressant de voir comment ceux-ci nous permettront d'appréhender les différents problèmes rencontrés et comment certaines décisions pourront influencer les étapes d'analyse antérieures.

L'implémentation de la **table résidente** fera l'objet de la deuxième section de ce chapitre. Nous expliquerons comment cette table résidente est conçue et certains mécanismes de son fonctionnement.

L'étude de l'environnement de programmation sera abordée dans la troisième section. Nous y discuterons également l'interface entre le langage de programmation et le S.G.B.D. utilisés.

Dans la section 3.5 "Réutilisabilité de la hiérarchie étendue d'objets de bureau", nous avions étudié la manière dont le schéma de classes résultant de l'analyse conceptuelle d'un environnement de bureau pouvait être adapté lors de l'introduction d'une nouvelle classe d'objets. La quatrième section étudiera l'**extensibilité de la boîte à outils d'objets bureautiques** d'un point de vue pratique la démarche à suivre par le programmeur d'application pour adjoindre une nouvelle classe d'objets aux classes existantes.

La gestion des erreurs, qui doit être extrêmement facile à mettre en oeuvre, fera l'objet de la cinquième section de ce chapitre. Nous verrons les différentes possibilités de gestion des erreurs applicables et nous expliquerons en détails la politique implémentée.

L'implémentation de l'application que nous présenterons dans la sixième section visera à montrer la facilité de mise en oeuvre et la puissance de la boîte à outils d'objets bureautiques que nous avons développée.

1. Sous-systèmes développés

1.1. Cadre de la démarche de prototypage

Comme nous l'avons déjà mentionné au chapitre 2 "Méthodologie de développement de systèmes", l'approche orientée objets inclut la notion de prototypage. Cette démarche de prototypage intervient à tous les niveaux. Ainsi, lors de la phase d'analyse conceptuelle, nous nous sommes basés sur l'environnement de bureau classique pour construire une hiérarchie de classes d'objets qui était validée et ensuite généralisée jusqu'à obtenir un schéma de classes d'objets satisfaisant .

Dans la partie 2.2.2 "Le prototypage dans l'approche orientée objets", nous avons mentionné que des **clusters**, qui représentent en fait des ensembles de classes d'objets en interaction s'articulant pour fournir un concept cohérent, pouvaient faire l'objet de cycles de développement indépendants et parallèles. C'est cette démarche que nous avons suivie lors de la phase d'implémentation. Nous avons donc successivement couvert le cycle de développement des clusters suivants :

- le **cluster 1** composé des classes d'objets OB, O-Rgt et O-Rgble,
- le **cluster 2** composé du cluster 1 et de la classe O-Struct,
- le **cluster 3**, reprenant les classes précédemment définies dans le cluster 2 et y ajoutant les classes O-Info, Info et Collection,
- le **cluster 4**, dernier cluster que nous avons développé, composé des classes du cluster 3 et des classes O-Non-Rgble, O-Rgt-Term, Armoire, Table et Poubelle.

Ces quatre clusters, qui sont présentés en détails ci-dessous, forment donc l'ensemble des systèmes implémentés. Le cluster 4 constitue dès lors l'ensemble de la boîte à outils développée. L'étude n'a pas pu être poussée plus loin faute de temps.

1.2. Présentation des clusters

1.2.1. Le cluster 1

Le premier cluster que nous avons développé (voir figure 6.1) est composé des classes OB, O-Rgble et O-Rgt. Il doit nous permettre de nous familiariser avec l'environnement de programmation sous lequel nous développons, à savoir le langage orienté objets TURBO PASCAL version 5.5 et le système de gestion de bases de données N.D.B.S.

Nous avons choisi ce premier sous-système parce qu'il est simple et cohérent. Sa simplicité (il contient seulement trois classes d'objets) nous a permis de choisir une stratégie pour l'implémentation des classes d'objets dans le langage utilisé. De plus, ce sous-système est fortement cohérent : en effet, les trois classes implémentées présentent de fortes interactions puisque les classes O-Rgt et O-Rgble sont des spécialisations de la classe OB et que ces deux classes sont reliées par deux types d'associations.

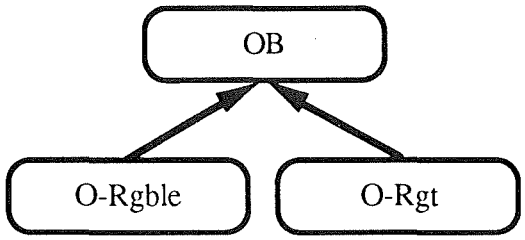

Figure 6.1 : Classes d'objets formant le cluster 1

1.2.2. Le cluster 2

Le deuxième cluster que nous avons choisi d'implémenter (voir figure 6.2) est composé des classes d'objets définies dans le premier cluster auxquelles nous avons ajouté la classe d'objets O-Struct. Cette classe est particulièrement importante puisqu'elle résulte d'une structure d'héritage multiple définie sur les classes O-Rgt et O-Rgble. Cette forte interaction entre ces trois classes d'objets nous a également permis de revoir certaines décisions que nous avions prises lors du développement du premier cluster.

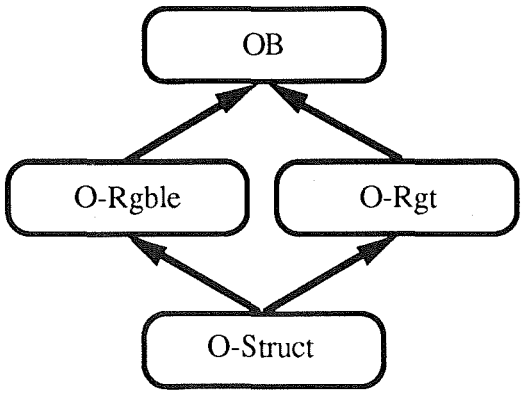

Figure 6.2. : Classes d'objets formant le cluster 2

Un autre intérêt de ce sous-système est d'étudier la manière dont les méthodes des classes d'objets de base doivent être redéfinies pour éviter les problèmes de cohérence. Par exemple, la méthode qui consiste à déplacer un objet rangeable dans un objet de rangement doit être redéfinie de manière à interdire le rangement d'un objet de rangement rangeable dans lui-même. En effet, vu de la classe d'objets O-Struct, le type d'association défini sur les classes O-Rgt et O-Rgble est récursif.

1.2.3. Le cluster 3

Le développement du deuxième cluster nous a permis de mettre en oeuvre des stratégies d'implémentation d'une structure d'héritage multiple. Le troisième cluster (voir figure 6.3) nous permet d'implémenter une des classes résultant des deux critères de partitionnement que nous avons choisis (voir partie 3.2.1 "Les critères de partitionnement et les classes résultantes"), à savoir la classe O-Info. Les classes Collection et Info qui font également partie de ce cluster, nous permettent d'exploiter les connaissances acquises lors de l'implémentation des structures d'héritage multiple.

Ce cluster est intéressant à plus d'un titre. En effet, les classes ainsi implémentées appartiennent à un niveau de la hiérarchie où la spécialisation est très importante. Ainsi, les classes héritant des propriétés des classes Info et Collection possèdent déjà un grand nombre de méthodes de leur interface implémentées. Deux autres points intéressants sont également à mentionner : le type d'association récursif Copie et le type d'association Organisation définis sur des types d'entités appartenant à deux niveaux de la hiérarchie différents. Nous pourrons ainsi améliorer notre connaissance de la gestion des interrelations entre classes d'objets.

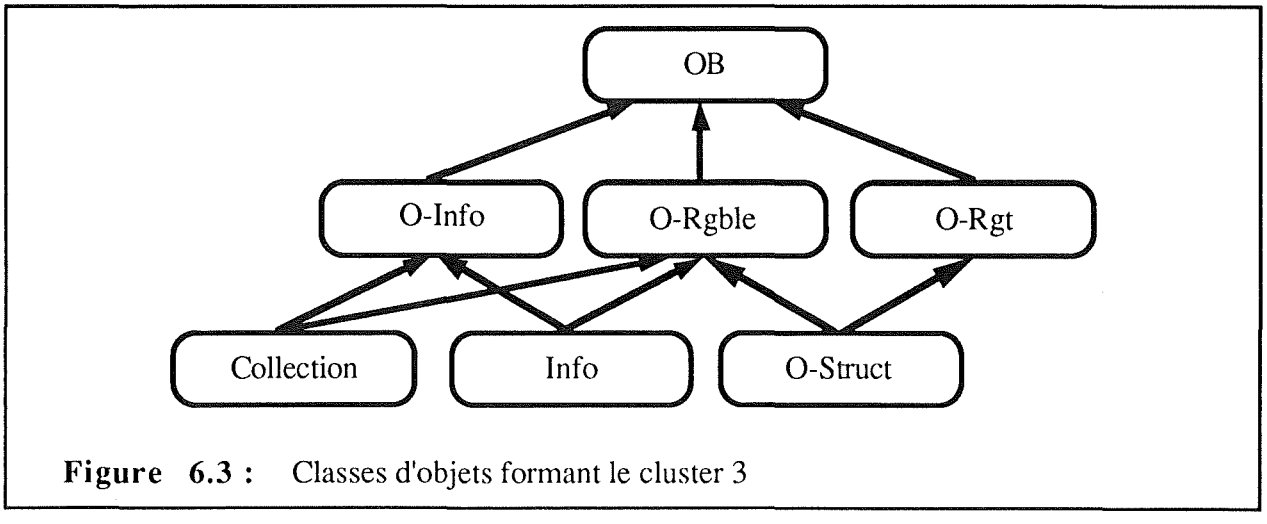

Figure 6.3 : Classes d'objets formant le cluster 3

1.2.4. Le cluster 4

Le cluster 4 (voir figure 6.4), basé sur le cluster 3, intègre la classe O-Non-Rgble et les classes représentant les différents objets de rangement de la hiérarchie, à savoir les classes O-Rgt-Term, Armoire, Table et Poubelle. Ces classes sont très faciles à implémenter car elles héritent en grande partie des propriétés définies au niveau de la classe O-Rgt.

C'est à ce niveau que l'on constate pleinement les avantages apportés par l'approche de développement orientée objets.

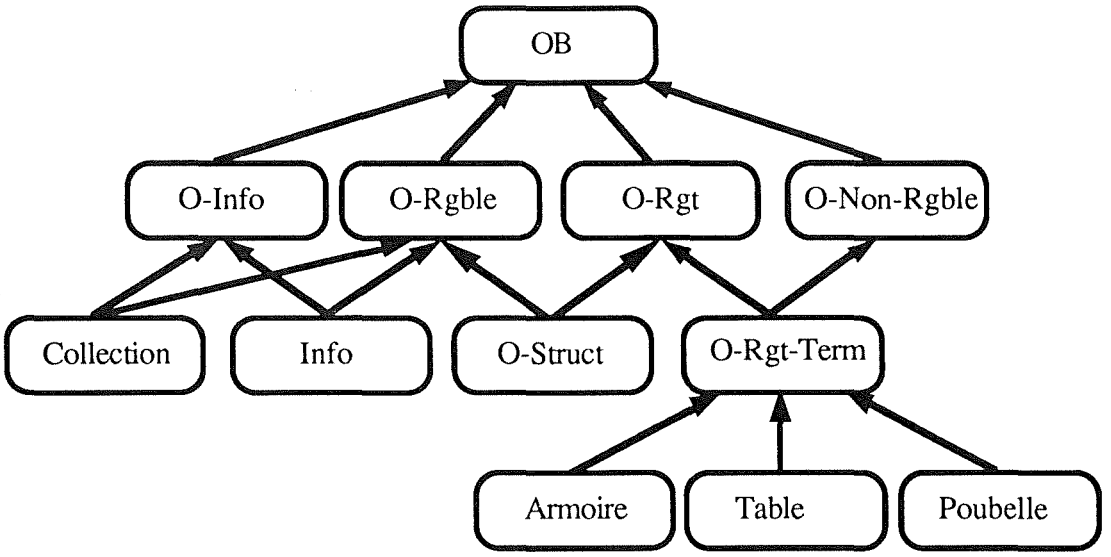

Figure 6.4 : Classes d'objets formant le cluster 4

1.3. Rétroactions liées à la méthode de prototypage

Suite à l'implémentation de chacun des clusters, des modifications et améliorations ont pu être apportées au modèle initial. Nous tenons dès lors à attirer l'attention sur quelques unes d'entre elles.

L'implémentation du cluster 1 a eu le plus d'impact sur le modèle initial. La principale modification a été celle de l'architecture du système résultant en l'introduction de la table résidente. Une autre amélioration non négligeable porte sur l'introduction d'une redondance dans le schéma conforme au S.G.B.D. : les attributs du type d'entité O-Rgt sont dupliqués et attribués au type d'entité O-Rgble. Cette redondance a pour unique but de faciliter l'introduction de nouvelles classes d'objets dérivées uniquement de la classe d'objets O-Rgble.

L'implémentation du cluster 2 nous permet d'élaborer une stratégie d'intégration des classes dérivées par héritage multiple. En effet, l'implémentation de la classe d'objets O-Struct nous a permis de tester les mécanismes d'inhibition et de réactivation des propriétés d'une classe de base. Ces techniques nous permettent d'implémenter les clusters 3 et 4 sans rencontrer de problèmes majeurs.

2. La table résidente

La table résidente qui résulte de la décision prise durant la phase de conception logique de garder toute instance d'une classe d'objets chargée par un programme d'application en mémoire centrale, permet de diminuer grandement le nombre d'accès logiques et de ce fait le nombre d'accès physiques nécessaires à la gestion du cycle de vie d'un objet.

La table résidente, tout comme le reste de l'application, a été implémentée en utilisant les concepts de la programmation orientée objets. Cette façon de faire permet de garder transparente à l'utilisateur toute la technique nécessaire au bon fonctionnement de cette table car il n'y accède que par l'intermédiaire des méthodes qui sont mises à sa disposition.

Les objets sont représentés dans le tableau par des variables dynamiques. Ainsi, des références multiples vers un même objet se traduisent par plusieurs entrées dans la table résidente pointant vers une occurrence unique de l'objet en mémoire centrale. Cette méthode permet une résolution aisée des problèmes de mises à jour simultanées de plusieurs variables du programme représentant un seul et même objet puisque des variables distinctes au niveau du programme d'application travaillent en fait sur la même occurrence de l'objet en mémoire centrale (voir figure 6.5). Cette solution permet également de minimiser l'espace mémoire utilisé pour la représentation des objets au prix de quelques méthodes permettant la gestion des entrées de la table pointant vers le même espace de données.

Finalement, un attribut Modification, ajouté au stade de l'implémentation et permettant de savoir si un objet a été modifié au cours de son cycle de vie assure la minimisation des réécritures dans la base de données à la fin du cycle de vie d'un objet. Les méthodes relatives à la gestion de ce nouvel attribut sont expliquées dans la partie "Spécifications d'implémentation de la classe Table-Res" de l'annexe 4 "Spécification des objets complémentaires".

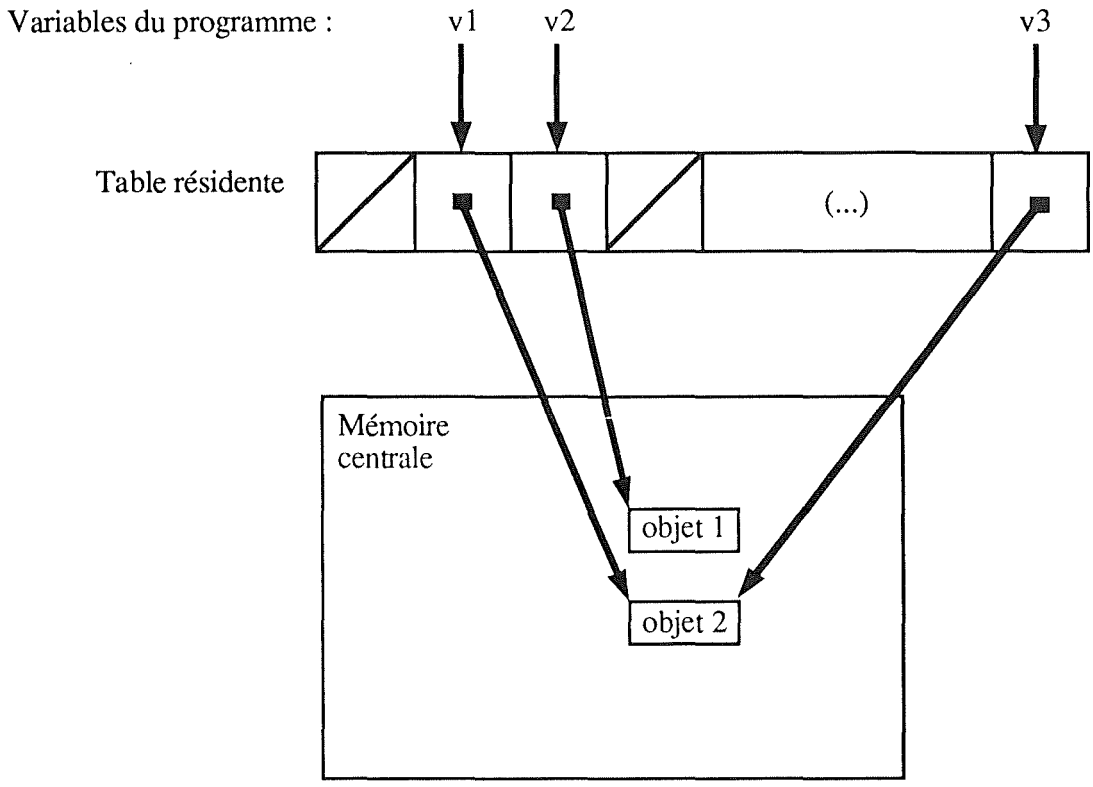

Figure 6.5 : Mécanisme de gestion des occurrences multiples de la table résidente

3. Etude de l'environnement de programmation

Un des aspects dignes d'attention de la phase d'implémentation de la boîte à outils d'objets bureautiques est le mariage d'un langage de programmation orienté objets, le TURBO PASCAL version 5.5 et d'un S.G.B.D. traditionnel, N.D.B.S. Il semble donc intéressant d'analyser le comportement de ces deux outils de programmation afin d'en retirer des enseignements pouvant être utiles aux personnes devant faire face aux problèmes auxquels nous avons été confronté durant cette phase d'implémentation.

3.1. Le système de gestion de bases de données

La phase d'analyse conceptuelle nous avait fourni le schéma E/A étendu d'un environnement de bureau. Pour pouvoir être représenté à l'aide d'un S.G.B.D. traditionnel, ce schéma avait, durant la phase de conception logique, été traduit en schéma E/A de base par l'élimination des structures d'héritage (voir section 4.3 "Construction d'un schéma E/A de base"). Après cette transformation, il restait à représenter le schéma E/A de base dans le formalisme du S.G.B.D. utilisé lors de la phase d'implémentation, à savoir N.D.B.S. (voir section 5.1 "Transformation du schéma de la base de données").

Cette transformation implique une violation du principe d'unicité du réel perçu. En effet, certaines classes d'objets de la hiérarchie sont représentées par plusieurs types d'entités dans la base de données. Par exemple, une instance de la classe d'objets Information est représentée par une occurrence du type d'entité O-Info, une occurrence du type d'entité O-Rgble, une occurrence du type d'entité Info et une occurrence du type d'association Is-a-O-Info.

Cet état de fait entraîne plusieurs conséquences. Premièrement, les performances d'accès s'en ressentent puisqu'un accès logique à une instance de classe d'objets correspond à plusieurs accès physiques à la base de données. Ensuite, si l'on prend deux classes reliées par une relation d'héritage, leur représentation est différente dans la base de données, ce qui entraîne une totale réécriture des modules d'accès pour chaque classe d'objets de la hiérarchie. Cette dernière conséquence entraîne une duplication de la hiérarchie de classes d'objets au niveau du langage de programmation afin d'isoler au maximum les modules d'accès à la base de données et de bénéficier pleinement des avantages apportés par l'utilisation d'un langage orienté objets.

3.2. Le langage de programmation

Nous avons vu dans la partie précédente que la réécriture des modules d'accès à la base de données entraînait une duplication de la hiérarchie de classes d'objets au niveau du langage de programmation. Nous allons maintenant présenter et étudier le fonctionnement de cette double hiérarchie des classes d'objets (voir figure 6.6).

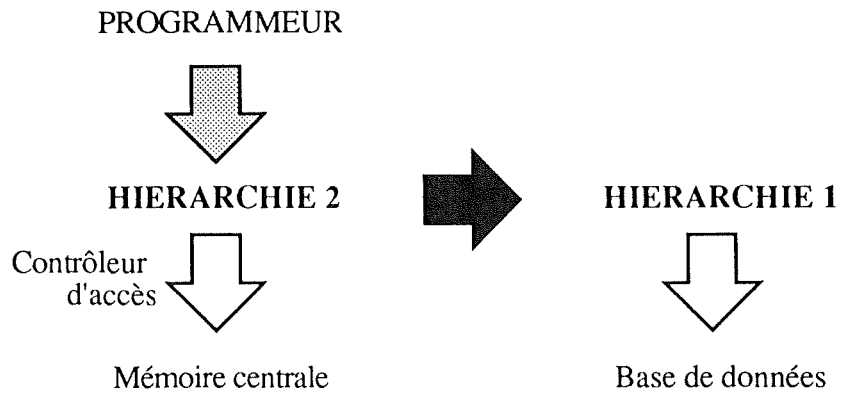

Figure 6.6 : Illustration de la duplication de la hiérarchie des classes d'objets bureautiques

La première hiérarchie de classes d'objets est la hiérarchie correspondant aux modules d'accès à la base de données. Ces modules étant réécrits entièrement à chaque niveau de la hiérarchie, les bénéfices apportés par l'utilisation d'un langage orienté objets sont donc rarement exploités. Ceci met en exergue les limitations du mariage entre un S.G.B.D. traditionnel et un langage de programmation orienté objets. En effet, à cause de la staticité d'un tel S.G.B.D., tous les avantages procurés par la programmation orientée objets sont pratiquement annihilés.

La deuxième hiérarchie de classes d'objets, qui représente le contrôleur d'accès, gère l'accès aux instances de classes d'objets en mémoire centrale. Cependant cette hiérarchie de classes d'objets réalise des appels aux méthodes de la première hiérarchie de classes gérant les propriétés des objets, notamment pour la gestion de leur cycle de vie. Une fois le cycle de vie d'un objet entamé, toute la gestion de celui-ci se fait en mémoire centrale. Ceci nous permet alors de profiter pleinement des avantages offerts par la programmation orientée objets, notamment des relations d'héritage.

3.3. Conclusion

Nous avons vu que l'utilisation d'un S.G.B.D. traditionnel, du fait de sa staticité, impliquait une quasi totale réécriture des modules d'accès à la base de données. Cependant, la décision de conserver toute instance d'une classe d'objets en mémoire centrale et la duplication de la hiérarchie de classes d'objets en vue d'isoler les modules d'accès permettent de profiter des avantages de la programmation orientée objets au niveau de la hiérarchie d'objets contrôlant l'accès aux instances des classes d'objets présentes en mémoire centrale.

4. Extensibilité de la boîte à outils d'objets bureautiques

Dans la section 3.5 "Réutilisabilité de la hiérarchie étendue d'objets de bureau", nous avons étudié l'extensibilité de la hiérarchie de classes d'objets qui avait été produite lors de la phase d'analyse de l'environnement de bureau étendu. Dans cette section, nous allons reprendre un exemple que nous avons déjà développé et étudier les conséquences de cette extension sur la boîte à outils d'objets bureautiques.

4.1. Démarche pratique d'extension

Afin d'introduire une nouvelle classe d'objets dans la hiérarchie existante, il faut avant tout réfléchir à l'endroit adéquat de la hiérarchie où elle doit être introduite. Cet endroit doit être choisi avec soin pour bénéficier au maximum des propriétés déjà existantes à ce niveau de la hiérarchie.

Une fois l'endroit d'insertion de la nouvelle classe d'objets déterminé, il reste à adapter la double hiérarchie de classes d'objets afin de satisfaire aux propriétés de cette nouvelle classe.

Premièrement, si la nouvelle classe d'objets possède des attributs non déjà définis, il faut déterminer un nouveau schéma de la base de données. Ce nouveau schéma est produit à partir du schéma E/A étendu résultant de l'introduction de la nouvelle classe d'objets.

Ensuite, tous les modules d'accès nécessaires pour la nouvelle classe d'objets doivent être réécrits et introduits dans la hiérarchie de classe. Cette étape est très lourde et fastidieuse, du fait qu'un S.G.B.D. traditionnel ne permet pas de bénéficier des avantages de la programmation orientée objets.

Troisièmement, la hiérarchie de classes gérant le cycle complet de l'objet doit être adaptée. Cette étape est beaucoup plus simple puisque seules les nouvelles propriétés ou les propriétés modifiées doivent être implémentées.

4.2. Introduction de la classe d'objets Photocopieuse

L'introduction de la classe d'objets Photocopieuse dans la hiérarchie de classes d'objets bureautiques a déjà été discutée d'un point de vue théorique dans la partie 5.5.2 "Introduction de nouveaux objets". Pour rappel, nous avions alors préconisé l'introduction de cette classe comme classe dérivée de la classe d'objets O-Term.

Afin de pouvoir représenter cette classe d'objets dans la base de données, une seule modification doit être effectuée sur le schéma de la base de données : la valeur "Photocopieuse" doit être ajoutée au domaine de valeurs de l'attribut Nom-O-Rgt-Term.

Ensuite, les primitives d'accès nécessaires à la gestion de la classe d'objets Photocopieuse doivent être réécrites puisque l'emploi d'un S.G.B.D. traditionnel ne permet pas de bénéficier des relations d'héritage entre classes d'objets disponibles dans un langage de programmation orienté objets.

Ensuite, la hiérarchie de classes gérant le cycle de vie de l'objet doit être adaptée afin de satisfaire l'introduction de la nouvelle classe d'objets. Cette adaptation est très aisée car dans ce cas, presque toutes les procédures peuvent être héritées telles quelles des classes de base.

5. La gestion des erreurs

Pour compléter la description de l'implémentation de la boîte à outils d'objets bureautiques, il reste à parler de la **gestion des erreurs**.

Il y a plusieurs politiques possibles pour gérer les erreurs qui peuvent survenir lors de l'appel d'une méthode d'un objet. Une première politique serait, par exemple, d'utiliser des variables globales qui sont testables par le programmeur afin de connaître le résultat de la requête qu'il vient de formuler. Une autre politique serait de passer une variable en paramètre à la méthode appelée dans laquelle on recueillerait un code d'erreur à tester à la sortie de cette méthode.

La politique que nous avons personnellement choisie de suivre est la deuxième. L'avantage majeur que nous y avons vu est que le programmeur sait toujours la variable qu'il doit tester et la manière dont il doit la tester. Avec l'emploi d'un type énuméré, les erreurs renvoyées peuvent devenir hautement significatives pour l'utilisateur. De plus, le choix de cette procédure de gestion des erreurs collait bien avec le reste de notre programme dans lequel toutes les variables sont locales pour procurer plus de facilités au programmeur.

6. Développement d'une application

Afin de montrer la facilité d'utilisation et la puissance de la boîte à outils d'objets bureautiques que nous avons développée, nous avons implémenté une petite application mettant cette boîte à outils en oeuvre.

Cette application ne se veut pas sophistiquée en prenant compte de toutes les possibilités mises au service du programmeur d'application, elle met simplement en avant les facilités de développement d'un logiciel à l'aide des services que cette boîte à outils peut rendre. C'est pour cette raison que l'application que nous avons développée est simple et introduit quelques restrictions quant à l'ampleur des fonctionnalités dont nous pouvions tenir compte.

L'écran de cette application se divise en plusieurs parties.

Une barre de menu principal permet à l'utilisateur de déclencher les actions permettant de commencer le cycle de vie d'un objet, à savoir la création d'un objet et l'accès direct à un objet. Il lui permet également de quitter l'application en sauvegardant les différents paramètres de l'environnement.

L'écran principal de l'application représente la table de travail de l'environnement de bureau. Elle est unique et possède un rôle différent que celui que nous lui avons donné dans la section 1.3 "Présentation d'un environnement de bureau classique". En effet, pour pouvoir travailler sur un objet rangeable, l'utilisateur doit d'abord placer celui-ci sur la table de travail.

Dans l'écran principal se situent les objets rangeables qui sont actuellement sur la table de travail. Pour activer les méthodes de ces objets, il suffit de les sélectionner en cliquant dessus pour voir apparaître la liste des primitives que l'on peut actionner pour un objet particulier.

A côté de la table de travail se trouvent une armoire et une poubelle. Ces deux objets sont également uniques et sont simplement là pour compléter l'environnement et donner une idée des méthodes définies pour ces deux classes d'objets terminales.

Le lecteur qui voudrait examiner le code de cette application et de la boîte à outils d'objets bureautiques peut en faire la demande au secrétariat général des étudiants.

Conclusion et perspectives d'avenir

1. Critique de l'environnement de programmation utilisé

La première partie de la conclusion de ce travail est consacrée à la critique de l'environnement de programmation qui a été utilisé lors de la phase d'implémentation de la boîte à outils d'objets bureautiques. Pour rappel, le langage de programmation mis à notre disposition est un langage incluant les concepts de la programmation orientée objets, en l'occurrence le TURBO PASCAL dans sa version 5.5. Du côté du S.G.B.D., nous nous sommes reposés sur un S.G.B.D. dit traditionnel, à savoir N.D.B.S., qui s'interface avec le langage PASCAL et permet la gestion de base de données de type réseau.

La première chose à remarquer est que ces deux outils de développement d'applications proviennent de deux mondes véhiculant des concepts tout à fait différents. Le TURBO PASCAL version 5.5 offre les concepts de la programmation orientée objets et est donc compatible (du moins partiellement) avec la phase d'analyse de l'environnement de bureau entièrement réalisée selon une approche orientée objets. Quant à N.D.B.S., il permet la gestion de bases de données directement dérivées de schémas décrits dans le formalisme E/A de base et n'inclut donc pas les structures d'héritage.

Dans la première partie de cette conclusion, nous allons analyser le développement de la boîte à outils d'objets bureautiques selon trois angles différents mais néanmoins complémentaires : celui du langage de programmation, celui du S.G.B.D et celui de l'interface entre ces deux outils de développement.

1.1. Le langage de programmation

Le TURBO PASCAL version 5.5 semble être un langage idéal pour la phase d'implémentation puisqu'il supporte les concepts orientés objets.

Cependant, une limitation importante de ce langage apparaît au niveau de l'implémentation car le TURBO PASCAL version 5.5 ne tolère que les structures d'héritage simple. Ceci a été pour nous un facteur extrêmement contraignant, puisque les classes intermédiaires de la hiérarchie reposaient sur des structures d'héritage multiple.

Cette contrainte a nécessité la transformation du schéma de classes d'objets en une structure d'héritage simple. Elle nous a conduits à inhiber des propriétés à différents niveaux de la hiérarchie de classes d'objets pour les réactiver au niveau suivant. Si le langage de programmation orienté objets utilisé pour implémenter l'application avait supporté les structures d'héritage multiple, les

phases de conception physique et d'implémentation auraient été simplifiées car le schéma de classes d'objets produit lors de l'analyse conceptuelle de l'environnement de bureau n'aurait dû subir que des transformations mineures.

1.2. Le S.G.B.D.

La décision d'utiliser un S.G.B.D. dit traditionnel peut sembler étrange à l'heure où le nombre de gestionnaires de base de données orientés objets est en pleine expansion. Il faut cependant noter que de tels gestionnaires de bases de données ne peuvent être mis en oeuvre que sur des configurations extrêmement performantes, tandis que les S.G.B.D. traditionnels se contentent de ressources techniques beaucoup moins importantes. La motivation principale de ce choix était d'étudier le comportement d'un tel S.G.B.D. alors que tout le développement de la boîte à outils d'objets bureautiques avait été réalisé selon l'approche orientée objets.

Pour être conforme au S.G.B.D., le schéma E/A étendu a dû subir de nombreuses transformations. Celles-ci introduisent certaines rigidités dans le schéma final.

Le principal reproche que l'on puisse faire à un S.G.B.D. traditionnel utilisé dans le contexte de la programmation orientée objets est sa staticité. En effet, une fois le schéma de la base de données défini, celui-ci reste immuable jusqu'à une prochaine redéfinition. Cette staticité entrave fortement l'extensibilité de la boîte à outils. Si un utilisateur de cette boîte à outils prend la décision d'ajouter une nouvelle classe à celles proposées, l'introduction de cette classe d'objets ne pose effectivement aucun problème au niveau du langage de programmation, mais du point de vue du S.G.B.D., cet ajout entraîne une redéfinition du schéma de la base de données qui peut, de ce fait, être fort différent du schéma précédemment défini.

1.3. L'interface entre le langage de programmation et le S.G.B.D.

Après s'être attardés successivement sur les problèmes liés au langage de programmation et au S.G.B.D. utilisés, il nous reste à parler de l'interface entre ces deux outils.

Lors de l'implémentation, nous avons été obligés de dédoubler la hiérarchie de classes d'objets. Cette décision, associée à celle de gérer le cycle de vie des objets en mémoire centrale, nous a permis de bénéficier au maximum des avantages que procure la programmation orientée objets. En effet, au niveau du contrôleur d'accès (qui s'occupe de la gestion des objets en mémoire centrale), la réutilisabilité des méthodes définies est quasi totale. Cependant, au niveau des modules

d'accès à la base données, toutes les primitives travaillant sur la base de données, c'est-à-dire celles relatives à la gestion du cycle de vie des objets doivent être réécrites pour chaque classe d'objets.

2. Apports de ce travail

Les quatre principaux apports de ce mémoire concernent le formalisme E/A étendu utilisé lors de la phase d'analyse, la technique de prototypage, les techniques de transformation de schéma et la confrontation des S.G.B.D traditionnels et de la programmation orientée objets.

Ce dernier point a déjà été abordé à la section précédente. Les trois autres sont développés ci-dessous.

2.1. Le formalisme E/A étendu

Lors de la phase d'analyse conceptuelle, le schéma conceptuel des objets a été décrit à l'aide d'un formalisme E/A étendu intégrant les structures d'héritage.

Le modèle E/A traditionnel constitue une technique idéale pour modéliser les structures de données à mémoriser dans une approche de développement traditionnelle. Suite à notre étude, il semblerait que le modèle E/A étendu possède les mêmes mérites vis-à-vis d'une approche orientée objets. L'enrichissement sémantique apporté par l'introduction des relations d'héritage dans le modèle E/A suffit pour modéliser les objets.

Grâce au modèle E/A étendu, nous avons pu calquer le schéma de la base de données sur le schéma de classes fournissant l'architecture des modules d'accès à la base de données. Ceci facilite la gestion de la base de données. En effet, une fois que les contraintes d'intégrité sont gérées par une classe d'objets, elles le sont automatiquement pour toutes ses classes dérivées grâce au mécanisme d'héritage. De plus, il semblerait que la modification de la représentation d'un objet dans la base de données n'ait qu'une influence locale au niveau des modules d'accès limitée à la classe d'objets concernée par la modification.

La symétrie entre les deux schémas facilite aussi l'extensibilité finale de la boîte à outils car de nouveaux objets viennent se greffer aux mêmes endroits dans les deux schémas. Cette caractéristique diminue le travail à fournir par le programmeur d'application pour introduire de nouvelles classes.

Comme on le voit, le modèle E/A étendu fournit donc un support idéal au développement orienté objets de la mémoire d'un système à objets.

2.2. La technique de prototypage

Des conclusions intéressantes sont aussi à retirer de la technique de prototypage utilisée. L'étape de validation théorique introduite lors de la phase d'analyse nous a permis d'appliquer la technique de prototypage à cette phase sans devoir attendre l'implémentation d'un premier sous-système.

L'application de cette technique s'est révélée très enrichissante. Elle est notamment utile pour agréger les objets en des classes d'objets plus générales. C'est grâce à cette démarche que le modèle d'environnement de bureau étendu fournit des services suffisamment généraux pour permettre l'implémentation d'applications gérant des environnements de rangement très divers.

De plus, l'approche orientée objets et l'approche par prototypage sont deux techniques de développement qui se marient très bien. Les principales raisons en sont la définition des clusters qui constituent des sous-systèmes idéaux pour le prototypage et le faible couplage existant entre les modules objets qui limite les modifications décidées suite au prototypage d'un cluster à ce seul cluster.

2.3. Les techniques de transformation de schéma

Un des apports principaux du mémoire concerne l'emploi des techniques de transformation de schémas. Rappelons que ces transformations concernaient deux structures différentes : le schéma conceptuel des objets qu'il fallait adapter à N.D.B.S et le schéma des classes d'objets qu'il fallait adapter à TURBO PASCAL version 5.5.

Pour le schéma conceptuel des données, les techniques de transformation utilisées ont pour but d'éliminer les structures de généralisation/spécialisation incluses dans la modélisation E/A étendue. Sur ce point, nous pensons avoir apporté quelques éléments de réponse intéressants sur les critères permettant d'opter en faveur de l'une ou l'autre des trois techniques de transformation proposées.

Pour le schéma des classes d'objets, la technique de transformation utilisée concernait l'élimination des structures d'héritage multiple. Nous avons montré l'utilité de cette technique mais aussi les lacunes qu'elle comportait quant aux limitations de la richesse sémantique et de la réutilisabilité du schéma initial qu'elle introduit.

De manière générale, il est évident que le grand défaut de toutes ces techniques de transformation est de réduire les possibilités d'extensibilité attribuées au modèle initial. Ces

transformations limitent donc les effets des propriétés propres à l'approche de développement orientée objets.

3. Perspectives d'avenir

3.1. Implémentation des classes terminales

Un travail futur concernant l'analyse et la conception d'une boîte à outils d'objets bureautiques serait de se baser sur ce travail pour implémenter les classes d'objets terminales que nous n'avons pu développer faute de temps.

Il s'agit des classes Message, Formulaire et Document pour les objets de la hiérarchie et de la classe Fragment utilisée pour modéliser la structure des documents. Cette dernière classe permettra d'étendre le cadre de ce travail en complétant les structures de rangement d'informations proposées par le traitement du contenu de ces informations.

3.2. Etude de l'interface entre S.G.B.D. traditionnel et programmation orientée objets

Nous avons déjà présenté les problèmes que comportaient l'interfaçage de N.D.B.S. et de TURBO PASCAL version 5.5. Les problèmes rencontrés dans ce cas sont nombreux et sont liés au fait que PASCAL est un langage hybride, intégrant les concepts d'une approche orientée objets par la force des choses et que N.D.B.S. est un gestionnaire de type réseau dont les schémas sont difficilement extensibles.

Plusieurs propositions qui s'inscrivent dans la ligne de conduite du travail que nous avons fourni peuvent être formulées.

Nous proposons l'utilisation de C++ et d'un gestionnaire de bases de données de type S.Q.L. (voir figure 1). Nous préconisons ce type de gestionnaire de bases de données, plutôt qu'un S.G.B.D. orienté objets qui, comme nous l'avons déjà mentionné, demande une configuration matérielle fort performante.

Les avantages que devraient comporter l'utilisation de ces outils sont relatifs au nombre de transformations à apporter au système résultant de la phase d'analyse.

C++ intègre les structures d'héritage multiple, le schéma de classes représentant les modules d'accès à la base de données ne devra subir aucune transformation.

Un gestionnaire de base de données de type S.Q.L. devrait posséder de nombreux avantages sur N.D.B.S. du point de vue de l'extensibilité finale de la boîte à outils. En effet, l'ajout d'un nouvel objet se traduisant par l'ajout d'une nouvelle table, le schéma ne devrait pas être recompilé à chaque extension. De plus, les classes d'objets dérivées de plusieurs classes de base par le mécanisme d'héritage multiple devraient facilement être représentables à l'aide de vues joignant les tables représentant les classes de base,... Cependant, il faut mentionner que de telles requêtes sont particulièrement peu efficaces.

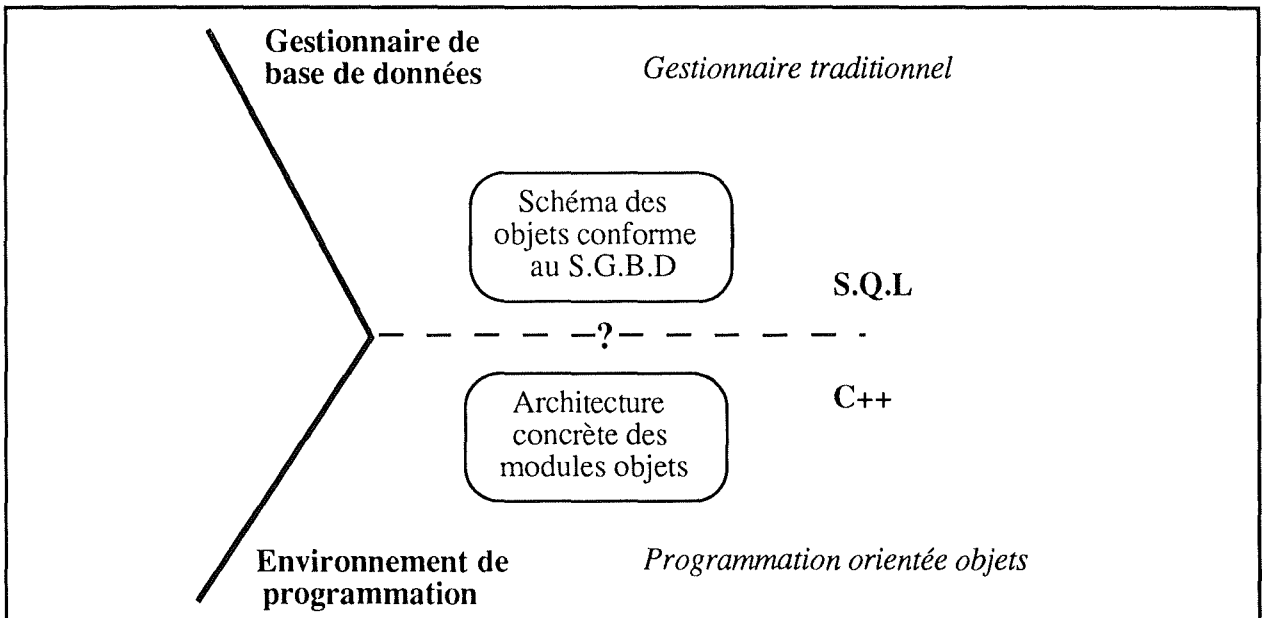

Figure 1 : Idée d'implémentation de l'environnement de rangement d'informations

Nous pensons que cette seconde implémentation de la boîte à outils fournirait une étude complémentaire à la nôtre et que la confrontation des résultats obtenus par chacun des outils serait d'un grand intérêt.

Références bibliographiques

[BLAS-82] J-P. De Blasis, 1982, *La bureautique : outils et applications*, pages 21, 23 et 28, Les éditions d'organisation.

[BOPI-83] F. Bodart, Y. Pigneur, 1983, *Conception assistée des applications informatiques*, Tome 1, Masson, PUF.

[BORL-89] *Object Oriented Programming Guide*, Turbo Pascal 5.5, Borland.

[BRIS-91] F. Bussaud, 1991, *Des associations pour un modèle à objets*, Actes des quatrièmes journées LIANA, Pratique des méthodes et outils logiciel d'aide à la conception des S.I.

[COJO-88] N. Collart, M. Joris, 1988, *Etude pratique et théorique d'extensions du modèle Entité/Association*, Institut d'Informatique, F.N.D.P., Namur.

[DACH-86] N. Dachouffe, 1986, *Spécification et implémentation d'un modèle d'information du bureau*, Institut d'Informatique, F.N.D.P., Namur.

[DUBO-90] Dubois, 1990, *Cours de méthodologie de développement de logiciels*, seconde licence en informatique, F.N.D.P., Namur.

[ELIO-90] J. Eliot, B. Moss, 1990, *Design of the Mneme Persistent Object Store,* A.C.M. Transactions on Information Systems, vol. 8, n° 2, pages 103-139

[HAIN-86] J-L. Hainaut, 1986, *Conception assistée des applications informatiques*, Tome 2, Masson, PUF.

[HAIN-87] J-L. Hainaut, 1987, "N.D.B.S., *A simple data base system for small computers* et *Complement of N.D.B.S. manual for version 2.0.*, Institut d'informatique, F.N.D.P., Namur.

[HAIN-89] J-L. Hainaut , 1989, *Bases de données et bases de connaissance en gestion des organisations*, Cinquième Ecole d'Automne des B.D., Institut d'informatique, F.N.D.P., Namur.

[HEED-90] B. Henderson-Sellers, J. M. Edwards, 1990, *The object-oriented systems life cycle*, Communications of the ACM.

[LACH-89] T. Lachand-Robert, 1989, *Programmation orientée objet en Turbo Pascal*, Editions Sybex.

[MART-82] J. Martineau, 1982, *La bureautique*, pages 23, 175 et 177 à 179, Editions McGraw-Hill.

[MEYE-88] B. Meyer, 1988, *Object Oriented Software Construction*, Editions Prentice-Hall, Internationnal Series on Computer Science.

[MEYE-89] B. Meyer, 1989, *The new culture of software development : Reflections on the practice of object-oriented design*, Proceedings of the First International Conference on the Technology of Object -Oriented Languages and Systems.

[MEYE-90] B. Meyer,1990, *Tools for the new culture : lessons from the design of the Eiffel Libriries*, Communication of the ACM, Vol. 33, N° 9.

[MGKO-90] J. Mc Gregor, T. Korson, 1990, *Object Oriented Design*, Communication of the ACM, Vol. 33, N° 9.

[ROBE-86] S. Robert, 1986, *Analyse et implémentation d'un environnement de rangement d'information*, Institut d'informatique, F.N.D.P., Namur.

[SHAM-89] N. Shammas, 1989, *Object oriented programming with Turbo Pascal*, Editions J. Wiley and Sons.

[VPMS-87] O. Van Pevenaege, M. Simoens, 1987, *Animation graphique de tâches de bureau*, Institut d'informatique, F.N.D.P., Namur.

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

Rue Grandgagnage, 21
B-5000 NAMUR (Belgium)

Annexes au mémoire : Conception et réalisation d'un environnement de développement d'applications bureautiques

Mémoire présenté en vue de l'obtention
du diplôme de Maître en Informatique

Eva-Marie Roberfroid
Stéphane Steinier

Promoteur : J-L. Hainaut

Année académique 1990-1991

Table des matières

Annexe 1 : Le langage de spécification

Annexe 2 : Spécification des modules d'accès aux objets de la base de données

Annexe 3 : Spécification du contrôleur d'accès

Annexe 4 : Spécification des objets complémentaires

Annexe 5 : Les techniques de transformation de schémas

Annexe 1 :
Le langage de spécification

Dans cette annexe, sont décrites les principales particularités du langage de spécification que nous avons utilisé. Ce langage possède plusieurs **types de données** prédéfinis (DATE, BOOL, CHAR, STRING, INTEGER) sur lesquels sont applicables les opérations habituelles. De plus, il met également à disposition des **constructeurs de types** et la possibilité de travailler sur des **états** permettant de garder des informations persistantes entre deux exécutions d'une même fonction ou entre l'exécution de deux fonctions différentes. Les constructeurs de types et les états sont expliqués ci-dessous.

1. Les constructeurs de types

Les constructeurs de type utilisés lors de nos spécifications sont les suivants : la **séquence** (SEQ), l'**ensemble** (SET), le **produit cartésien** (CP) et la **table**.

1.1. La séquence

La séquence permet la représentation d'une suite de données de même type où l'ordre a de l'importance. Pour faire une analogie avec les langages de programmation, elle ressemble quelque peu à un tableau. Cependant, il n'y a pas de limites quant aux bornes d'une séquence.

Les différentes opérations disponibles sur la séquence et appliquées à l'exemple d'une séquence d'entiers s sont :

- la construction d'une séquence : s = [1, 2, 3],
- l'appartenance d'un élément à une séquence : (3 ∈ s) qui donne la valeur TRUE,
- le test d'une séquence vide : Empty(s) qui donne la valeur FALSE,
- l'obtention de la longueur d'une séquence : Length(s) = 3,
- l'accès au premier élément d'une séquence : First(s) = 1,
- l'accès au dernier élément d'une séquence : Last(s) = 3,
- l'extraction du début d'une liste : Head(s) = [1, 2],
- l'extraction de la fin d'une liste : Tail(s) = [2, 3],
- l'extraction d'une sous-liste d'une liste à partir d'un élément donné : Subseq(s, 1, 2) = [1, 2],
- la concaténation de deux séquences grâce à l'opérateur +,
- l'accès au $i^{ème}$ élément d'une séquence : $s_1 = 1$,
- l'ajout d'un élément à la fin d'une séquence : Append(s, 4) = [1, 2, 3, 4],
- l'ajout d'un élément en $i^{ème}$ position d'une séquence : Add-ith(s, 3, 4) = [1, 2, 4, 3],
- la suppression d'un élément en $i^{ème}$ position d'une séquence : Remove-ith(s, 1) = [2, 3],

- le filtrage, c'est-à-dire l'extraction d'un sous-suite vérifiant certaines propriété : Filter(s, s_i = 1) = [1].

D'autres opérations plus compliquées, comme l'itération associative par exemple, peuvent être également mises en oeuvre, mais comme nous ne les utilisons pas dans nos spécifications, elles ne seront pas détaillées ici.

1.2. L'ensemble

L'ensemble permet la représentation d'un ensemble de données de même type où l'ordre n'a pas d'importance.

Les différentes opérations disponibles sur les ensembles et appliquées à l'exemple d'un ensemble d'entiers e sont :

- la construction d'un ensemble : e = {1, 2, 3},
- l'application des opérateurs ensemblistes courants comme la concaténation, la soustraction, la réunion, l'intersection, l'inclusion, ...,
- l'appartenance d'un élément à un ensemble : (3 ∈ e) qui donne la valeur TRUE,
- le test d'un ensemble vide : Empty(e) qui donne la valeur FALSE,
- l'obtention du nombre d'éléments d'un ensemble : Card(e) = 3,
- l'ajout d'un élément à un ensemble : Add(e, 4) = {1, 2, 3, 4},
- la suppression d'un élément d'un ensemble : Remove(e, 1) = {2, 3}.

D'autres opérations plus compliquées, comme l'itération associative par exemple, peuvent être également mises en oeuvre, mais comme nous ne les utilisons pas dans nos spécifications, elles ne seront pas détaillées ici.

1.3. Le produit cartésien

Ce concept permet la définition de t-uples d'éléments de types différents. Il peut bien sûr être combiné avec les autres constructeurs de types.

Un exemple de produit cartésien peut être : p = CP[nom : STRING, âge : INTEGER].

Les opérations permises sur les produits cartésiens sont les suivantes :

- la création d'un objet du type : p = <"Steinier", 23>,
- l'accès à un des champs du t-uple : nom(p) = "Steinier",
- la comparaison de deux t-uples avec les opérateurs = et ≠,

- la modification d'un champ du t-uple : p' = (p, age→24).

1.4. La table

La table est uns structure de données plus élaborée construite à partir du produit cartésien et de l'ensemble.

Un exemple de table pourrait être : clients : [INTEGER→STRING], qui reviendrait à SET[PC[INTEGER, STRING]].

Les opérations disponibles sur les tables sont les suivantes :

- l'insertion d'un enregistrement dans la table : clients' = Insert(clients, 1, "Roberfroid"),
- la recherche des identifiants de la table : In-Dom(clients) = {1},
- l'accès au codomaine de la table : In-Codom(clients) = {"Roberfroid"},
- la suppression d'un élément de la table : clients' = Delete(clients, 1),
- la modification d'un élément de la table : clients' = Modify(clients, 1, "Steinier"),
- l'accès au $i^{ème}$ élément de la table : clients[1] = "Steinier".

2. Les états

Un état agit un peu comme une base de données, en ce sens qu'il permet de garder une trace de renseignements devant subsister entre l'exécution de deux fonctions quelconques, il constitue donc la mémoire du système. Dans le cas qui nous intéresse, l'état utilisé sera décrit grâce au formalisme E/A.

Si l'on prend l'exemple de la figure A.1.1, les requêtes suivantes peuvent par exemple être formulées :

- In(A(var-id-a)) permet de tester l'existence d'une occurrence du type d'entité A identifié par var-id-A,
- att-A(A(var-id-A)) permet l'obtention des attributs de l'occurrence du type d'entité A identifié par var-id-A,
- RA(A(var-id-A)) donne l'ensemble des identifiants des occurrences des types d'entités B reliés à l'occurrence du type d'entité A identifié par var-id-A par l'association AB,
- In(AB(var-id-A, var-id-B)) teste l'existence d'une association AB entre l'occurrence du type d'entité A identifié par var-id-A et l'occurrence du type d'entité B identifié par var-id-B.

Il va de soi que ces requêtes peuvent être imbriquées de manière à former des requêtes beaucoup plus compliquées.

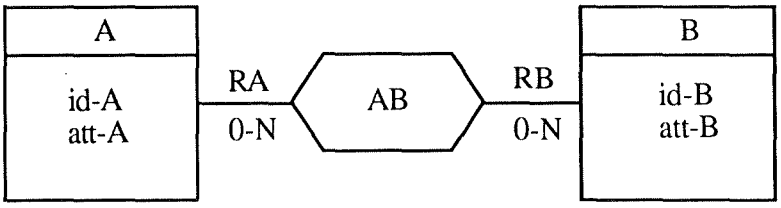

Figure A.1.1 : Etat exprimé selon le formalisme E/A

Annexe 2 :
Spécification des modules d'accès aux objets de la base de données

Introduction

Cette annexe est consacrée à la spécification des interfaces des classes d'objets de la hiérarchie décrivant un environnement de bureau étendu. Ces classes d'objets constituent les modules d'accès aux objets stockés dans la base de données.

Les méthodes à décrire sont de deux types. Premièrement, celles qui réalisent le chargement des objets depuis la BD et la mise à jour des objets modifiés dans la BD une fois leur utilisation terminée. Rappelons que ces méthodes sont celles qui délimitent le cycle de vie des objets à savoir les méthodes Accès-Direct, Créer, Supprimer et Fin-Utilisation. Elles assurent le stockage permanent des objets sur un support non volatile. Pour les spécifier, un état appelé BD présenté en fin d'annexe est utilisé. Quant au deuxième type de méthodes, il est constitué des méthodes qui travaillent sur les objets chargés en mémoire centrale. Comme ces dernières accèdent et modifient les structures représentant les objets chargés depuis la BD, elles n'utilisent pas l'état BD.

Pour réaliser la spécification des interfaces, les principes décrits à la section 4.2 "Spécification formelle des objets" ont été utilisés. Rappelons que l'héritage porte sur les structures de données, invariants et méthodes. Par exemple, la méthode Créer-OB (définie pour la classe OB) travaille sur l'état BD grâce à la primitive Enregistrer-OB(nom,att). Au niveau de la définition de l'interface de la classe O-Rgt, elle est héritée telle quelle. L'enrichissement n'apparaît que si l'on se réfère à la primitive Enregistrer-O-Rgt(nom,att) spécifiée dans l'état BD. Une transposition identique doit être réalisée pour toutes les méthodes de classes d'objets travaillant sur l'état BD : pour la classe Objet Rangeable, il faut se référer aux primitives BD suivies de O-Rgble,...

Remarquons aussi que les spécifications ne sont pas "fermées". Ceci signifie que les méthodes utilisées pour spécifier un objet, par exemple pour établir sa précondition, peuvent être des méthodes d'une autre classe d'objet. Un exemple de ce type est la méthode Objet-Chargé(o) qui apparaît dans la plupart des préconditions des méthodes du niveau modules d'accès mais qui est une méthode provenant des objets du contrôleur d'accès.

Une dernière remarque importante concerne les spécifications proprement dites : lorsqu'une méthode possède un même objet en entrée et en sortie par exemple, la méthode Modifier-Att(o, att) = o', la postcondition ne reprend que les modifications apportées à l'objet en entrée. Pour le reste, nous supposerons que l'objet en sortie possède la même partie privée que l'objet en entrée. Cette convention est introduite dans le but de ne pas trop alourdir les spécifications.

Les classes d'objets seront spécifiées dans leur ordre d'apparition dans la hiérarchie d'objets. De plus, une spécifiation complète n'a été réalisée que pour les classes d'objets

appartenant aux clusters implémentés. La raison principale conduisant à cette décision est que la méthode de prototypage influence les spécifications. Les classes d'objets apparaîtront dans l'ordre suivant :

- la classe OB, racine de la hiérarchie ,
- ses descendants directs, dans l'ordre : O-Rgt, O-Rgble, O-Info, O-Non-Rgble,
- les classes obtenues par héritage multiple : O-Struct, Info, Collection, O-Term et les classe terminales héritières de O-Term à savoir les classes Armoire, Poubelle et Table.

La fin de l'annexe est consacrée à la spéciifcation de l'état BD.

OBJET : OB

Structure

OB = CP[nom : STRING, att-OB : attributs-OB]

Cette définition est basée sur les structures de données intermédiaires suivantes :

attributs-OB = [att-OB-création, dates]
att-OB-création = CP[description : STRING, créateur : STRING, propriétaire : STRING]
dates = CP[date-création : DATE, date-der-cons : DATE, date-der-maj : DATE]
att-OB-lecture = attributs-OB

Invariants

- $\forall$ o,o' : o $\neq$ o' $\Leftrightarrow$ Lire-Nom(o) $\neq$ Lire-Nom(o') o,o' : OB
- $\forall$ o : date-création(o) $\leq$ date-der-maj(o) $\leq$ date-der-cons(o)
- $\forall$ o : objet-chargé(o) $\Rightarrow$ Existe-OB-BD(Lire-Nom(o))

Ce dernier invariant signifie qu'il n'y a en mémoire centrale que des objets qui existent dans la base de données.

Interface

Créer : STRING x att-OB-creation $\rightarrow$ OB x **BD**
Accès-Direct : **BD** x STRING $\rightarrow$ OB
Gestion-Date-Cons : OB $\rightarrow$ OB
Gestion-Date-Maj : OB $\rightarrow$ OB
Modifier-Att : OB x att-OB-création $\rightarrow$ OB
Recopier-BD : OB $\rightarrow$ **BD**
Supprimer : OB $\rightarrow$ **BD**
Lire-Att : OB $\rightarrow$ att-OB
Lire-Nom : OB $\rightarrow$ STRING
Existe-Objet : **BD** x STRING $\rightarrow$ BOOL
Objet-Chargé : OB $\rightarrow$ BOOL

Remarque : L'état BD apparaissant dans la signature des méthodes délimitant le cycle de vie des objets est un paramètre particulier qui n'est pas renvoyé à l'utilisateur de la méthode. Sa présence est requise pour décrire de manière complète et non ambiguë l'action de ces méthodes. Il signifie que l'appel de ces méthodes modifie le contenu de la base de données.

Etat

L'état utilisé par les méthodes de cette classe d'objets est l'état BD décrit à la fin de ces spécifications.

Spécifications des méthodes

Constructeurs

• Créer(nom, att) = o

o : OB
nom : STRING
att : att-OB-création
message : STRING

Précondition : /

Postcondition : ¬ Existe-OB(nom)
⇒ Enregistrer-OB(nom, liste-att) = o[1]

Excepté : Existe-OB(nom) ⇒ message = 'objet existant'

• Accès-Direct(nom) = o

o : OB
nom : STRING
message : STRING

Précondition : /

Postcondition : Existe-OB(nom) ⇒ Charger-OB(nom) = o

Excepté : ¬ Existe-OB(nom) ⇒ message = 'objet inexistant'

Modificateurs

• Gestion-Date-Cons(o) = o'

o, o' : OB

Précondition : Objet-Chargé(o)

Postcondition : date-der-cons(o') = date-jour [2]

• Gestion-Date-Maj(o) = o'

o, o' : OB

Précondition : Objet-Chargé(o)

Postcondition : Gestion-Date-Cons(o) = o' ∧ date-der-maj(o') = date-jour

---

[1] Enregistrer-OB est une primitive BD. Pour connaître l'effet de la méthode Créer sur la BD, il faut se référer à l'état BD en fin d'annexe.

[2] La convention introduite plus haut implique donc que toutes les autres propriétés de o' sont identiques à celles de o.

• Modifier-Att(o, att) = o' — o, o' : OB; att : att-OB-création

Précondition : OB-Chargé(o)

Postcondition : Gestion-Date-Maj(o) = o' ∧ att-OB-création(o') = att

• Recopier-BD(o) — o : OB

Précondition : Objet-Chargé(o)

Postcondition : Recopier-OB(o)

• Supprimer(o) — o : OB

Précondition : Objet-Chargé(o)

Postcondition : Détruire-OB(o)

Observateurs

• Lire-Att(o) = att — o : OB; att : att-OB-lecture

Précondition : Objet-Chargé(o)

Postcondition : att = att-OB(o)

• Lire-Nom(o) = nom — o : OB; nom : STRING

Précondition : Objet-Chargé(o)

Postcondition : nom = nom(o)

• Existe-Objet(nom) — nom : STRING

Précondition : /

Postcondition : Existe-OB-BD(nom)

• Objet-Chargé(o) — o : OB; objet : Objet-Bureau

Précondition : /

Postcondition : ∃ objet : Existe-Num-Place(t, entrée-table(objet)) ∧ o ≠ OBvide ∧ Objet-Bureau-Chargé(objet) ∧ Lire-Place(t, entrée-table(objet)) = o

OBJET : O-Rgt

Structure

O-Rgt is a OB with CP[contient-origine : Liste-Objets, contient-réelle : Liste-Objets]

Les structures de données intermédiaires sont identiques à celles définies pour OB.

Invariants[3]

∀ o : o : O-Rgt,
o',o'' : O-Rgble

- ∀ o' : Existe-Objet(contient-origine(o), o')
  - ⇔ o = Loc-Origine(o')
  - ⇔ In(Localisation-Origine(O-Rgt(Lire-Nom(o)), O-Rgble(Lire-Nom(o'))))
  - ⇒ Donner-Type(Lire-Nom(o')) = 'objet rangeable'
- ∀ o'' : Existe-Objet(contient-réelle(o), o'')
  - ⇔ o = Loc-Réelle(o'')
  - ⇔ In(Localisation-Réelle(O-Rgt(Lire-Nom(o)), O-Rgble(Lire-Nom(o''))))
  - ⇒ Donner-Type(Lire-Nom(o'')) = 'objet rangeable'

Remarque : Donner-Type(Lire-Nom(o'')) = 'objet rangeable' signifie que le résultat renvoyé par cette fonction doit être un type objet définit comme étant un objet rangeable par exemple, 'objet-structurant' .

Interface

Tester-Localisation-Réelle : O-Rgt → BOOL
Premier-O-Rgble-Loc-Réelle: O-Rgt → O-Rgble
O-Rgble-Loc-Réelle-Suivant : O-Rgt x O-Rgble→ O-Rgble
Premier-O-Rgble-Loc-Origine : O-Rgt → O-Rgble
O-Rgble-Loc-Origine-Suivant : O-Rgt x O-Rgble→ O-Rgble
Tester-Localisation-origine : O-Rgt → BOOL

---

[3] Les invariants traduisent les relations existant entre les classes O-Rgt et O-Rgble. Ces relations sont liées à l'existence d'un type d'association entre les deux classes.

Spécifications des méthodes

Observateurs

• Tester-Localisation-Origine(o) = b

o : O-Rgt
b : BOOL

Précondition : Objet-Chargé(o)

Postcondition : b = Test-Liste-Vide(contient-origine(o))

• Premier-O-Rgble-Loc-Origine(o) = o1

o : O-Rgt
o1 : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : o1 = Lire-Premier-Objet(contient-origine(o))

• O-Rgble-Loc-Origine-Suivant(o, o1) = o2

o : O-Rgt
o1, o2 : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : o2 = Lire-Objet-Suivant(contient-origine(o), o1)

• Tester-Localisation-Réelle(o) = b

o : O-Rgt
b : BOOL

Précondition : Objet-Chargé(o)

Postcondition : b = Test-Liste-Vide(contient-réelle(o))

• Premier-O-Rgble-Loc-Réelle(o) = o1

o : O-Rgt
o1 : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : o1 = Lire-Premier-Objet(contient-réelle(o))

• O-Rgble-Loc-Réelle-Suivant(o, o1) = o2

o : O-Rgt
o1, o2 : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : o2 = Lire-Objet-Suivant(contient-réelle(o), o1)

OBJET : O-Rgble

Structure

O-Rgble is a OB with CP[est-contenu-origine : O-Rgt, est-contenu-réel : O-Rgt]

Les structures de données intermédiaires sont identiques à celles définies pour OB sauf celle relative à la création qui deviennent [4]:

att-O-Rgble-création = [description : STRING, créateur : STRING, propriétaire : STRING, O-Rgt-Origine : O-Rgt, O-Rgt-Réelle : O-Rgt]

Invariants

$\forall$ o : • $\exists!$ o' : o' = Loc-Origine(o) (o : O-Rgble, o', o'' : O-Rgt)

$\Leftrightarrow$ Existe-Objet(contient-origine(o'), o)

$\Leftrightarrow$ In(Localisation-Origine(O-Rgt(Lire-Nom(o'')), O-Rgble(Lire-Nom(o))))

$\Rightarrow$ Donner-Type(Lire-Nom(o')) = 'objet rangement'

• $\exists!$ o'' : o'' = Loc-Réelle(o)

$\Leftrightarrow$ Existe-Objet(contient-réelle(o''), o)

$\Leftrightarrow$ In(Localisation-Réelle(O-Rgt(Lire-Nom(o'')), O-Rgble(Lire-Nom(o))))

$\Rightarrow$ Donner-Type(Lire-Nom(o'')) = 'objet rangement'

Remarque : Donner-Type(Lire-Nom(o')) = 'objet rangement' signifie que le résultat renvoyé par cette fonction doit être un type objet définit comme étant un objet de rangement par exemple, 'armoire' .

Interface

Déplacer : O-Rgble x O-Rgt x O-Rgt → O-Rgble x O-Rgt x O-Rgt

Changer-Origine : O-Rgble x O-Rgt x O-Rgt → O-Rgble x O-Rgt x O-Rgt

Replacer : O-Rgble x O-Rgt x O-Rgt → O-Rgble x O-Rgt x O-Rgt

Loc-Origine : O-Rgble → O-Rgt

Loc-Réelle : O-Rgble → O-Rgt

---

[4] L'ajout de O-Rgt-Origine et O-Rgt-Réelle dans les attributs de création est lié aux contraintes de connectivité minimale des rôles auxquels O-Rgble participe.

Spécifications des méthodes

Modificateurs

• Déplacer(o, o-orig, o-dest) = <o', o-orig', o-dest'>

o, o' : O-Rgble
o-orig, o-orig', o-dest
o-dest' : O-Rgt
message : STRING

Précondition : Objet-Chargé(o) ∧ Objet-Chargé(o-orig) ∧ Objet-Chargé(o-dest)

Postcondition : Loc-Réelle(o) = o-orig
⇒ Loc-Réelle(o') = o-dest ∧
¬ Existe-Objet(contient-réelle(o-orig'), o) ∧
Existe-Objet(contient-réelle(o-dest'), o)

Excepté : Loc-Réelle(o) ≠ o-orig ⇒ message = 'objet origine incorrect'

• Changer-Origine(o, o-orig, o-dest) = <o', o-orig', o-dest'>

o, o' : O-Rgble
o-orig, o-orig', o-dest,
o-dest' : O-Rgt
message : STRING

Précondition : Objet-Chargé(o) ∧ Objet-Chargé(o-orig) ∧ Objet-Chargé(o-dest)

Postcondition : Loc-Origine(o) = o-orig
⇒ Loc-Origine(o') = o-dest ∧
¬ Existe-Objet(contient-origine(o-orig'), o) ∧
Existe-Objet(contient-origine(o-dest'), o)

Excepté : Loc-Origine(o) ≠ o-orig ⇒ message = 'objet origine incorrect'

• Replacer(o, o-orig, o-dest) = <o', o-orig', o-dest'>

o, o' : O-Rgble
o-orig, o-orig', o-dest,
o-dest' : O-Rgt
message : STRING

Précondition : Objet-Chargé(o) ∧ Objet-Chargé(o-orig) ∧ Objet-Chargé(o-dest)

Postcondition : Loc-Réelle(o) = o-orig ∧ Loc-Origine(o) = o-dest
⇒ Loc-Réelle(o') = o-orig ∧
¬ Existe-Objet(contient-réelle(o-orig'), o) ∧
Existe-Objet(contient-origine(o-dest), o)

Excepté : Δ Loc-Origine(o) ≠ o-dest ⇒ message = 'objet destination incorrect'
Δ Loc-Réelle(o) ≠ o-orig ⇒ message = 'objet origine incorrect'

Observateurs

• Loc-Origine(o) = o'

o : O-Rgble
o' : O-Rgt

Précondition : Objet-Chargé(o)

Postcondition : o' = est-contenu-origine(o)

• Loc-Réelle(o) = o' Précondition : Objet-Chargé(o) Postcondition : o' = est-contenu-réel(o)	o : O-Rgble o' : O-Rgt

OBJET : O-Info

Structure

O-Info is a OB with CP[mots-clés : SEQ[STRING], est-organisé-en : Liste-Objets]

Les structures de données intermédiaires sont identiques à celles définies pour OB.

Invariants

∀ oi : oi : O-Info
c,c' : Collection

- ∀ c : Existe-Objet(est-organisé-en(oi), c)
  - ⇔ In(organisation(O-Info(Lire-Nom(oi)), Collection(Lire-Nom(c))))
  - ⇔ Existe-Objet(organise(c), oi)
  - ⇒ Donner-Type(Lire-Nom(o')) = 'collection'
- ∄ c' : Existe-Objet(est-organisé-en(oi), c') ∧ Lire-Nom(oi) = Lire-Nom(c')

Interface

Ajouter-Mot-Clé : O-Info x STRING → O-Info
Supprimer-Mot-Clé : O-Info x STRING → O-Info
Lister-Mots-Clé : O-Info → SEQ[STRING]

Spécifications des méthodes

Modificateurs

• Ajouter-Mot-Clé(oi, mc) = oi' — oi, oi' : O-Info

Précondition : Objet-Chargé(oi)

Postcondition : mots-clés(oi') = Append(mots-clés(oi), mc)

• Supprimer-Mot-Clé(oi, mc) = oi' — oi, oi' : O-Info, i : INTEGER

Précondition : Objet-Chargé(oi)

Postcondition : if ∃ i : 1 ≤ i ≤ length(mots-clés(oi)) : mots-clés$_i$(oi) = mc
then mots-clés(oi') = delete-ith(mots-clés(oi), i)
else mots-clés(oi) = mots-clés(oi')

Observateurs

• Lister-Mots-Clé(oi) = lmc — oi : O-Info, lmc : SEQ[STRING]

Précondition : Objet-Chargé(oi)

Postcondition : lmc = mots-clés(oi)

OBJET : O-Non-Rgble

Structure

O-Non-Rgble is a OB

Les structures de données intermédiaires sont identiques à celles définies pour OB.

Invariants

Cette classe d'objets ne doit respecter aucun invariant particulier si ce n'est ceux dont elle hérite.

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

OBJET : O-Struct

Structure

O-Struct is a O-Rgt, is a O-Rgble with att-O-Struct : atttributs-O-Struct

attributs-O-Struct = CP[critère-rgt: STRING, ordre-regroup: type-regroup, étiquette: STRING]
type-regroup = {alphabétique, numérique, chronologique, idéologique, manuel, aucun}
att-O-Struct-création = CP[att-OB-création, attributs-O-Struct]
att-O-Struct-lecture = CP[att-OB-lecture, attributs-O-Struct]

Invariants

$\nexists$ os' : Détecter-cycle(os,os') $\wedge$ os' = Loc-Origine(os) — os, os' : O-Struct

ou : Détecter-cycle(os,os') =
    if Loc-Origine(os) = os'
    then TRUE
    else Donner-Type(Lire-Nom(os')) = 'objet structurant' $\Rightarrow$ os' = Loc-Origine(os')

$\nexists$ os' : Détecter-cycle(os,os') $\wedge$ os' = Loc-Réelle(os)

ou : Détecter-cycle(os,os') =
    if Loc-Réelle(os) = os'
    then TRUE
    else Donner-Type(Lire-Nom(os')) = 'objet structurant' $\Rightarrow$ os' = Loc-Réelle(os')

Ces invariants traduisent la contrainte d'intégrité de non-existence de circuits dans les relations d'inclusion définies sur les objets de rangement rangeables.

Interface

Etiqueter : O-Struct x STRING → O-Struct
Changer-Regroupement : O-Struct x STRING x type-regroup → O-Struct
Lire-Etiquette : O-Struct → STRING

Spécifications des méthodes

Modificateurs

• Etiqueter(os, e) = os' — os, os' : O-Struct; e : STRING

Précondition : Objet-Chargé(os)

Postcondition : étiquette(os') = e

• Changer-Regroupement(os, cr, or) = os' Précondition : Objet-Chargé(os) Postcondition : critère-rgt:(os') = cr ∧ ordre-regroup(os') = or	os, os' : O-Struct cr : STRING or : type-regroup
Observateurs • Lire-Etiquette(os) = e Précondition : Objet-Chargé(os) Postcondition : e = étiquette(os)	os : O-Struct, e : STRING

OBJET : Info

Structure

Info is a O-Info, is a O-Rgble
with CP[att-O-Info : attributs-O-Info, copies : Liste-Objets, a-pour-original : Info]

Les structures de données intermédiaires sont un peu plus compliquées que pour les autres objets car selon que la méthode utilisée (création ou lecture), les attributs à fournir varient :

attributs-O-Info = CP[présence : BOOL, confidentialité : BOOL, mot-de-passe : STRING, support : STRING]
attributs-O-Info' = CP[confidentialité : BOOL, mot-de-passe : STRING, support : STRING]
attributs-O-Info'' = CP[présence : BOOL, confidentialité : BOOL, support : STRING]
att-O-Info-création = CP[att-O-Rgble-création, attributs-O-Info']
att-O-Info-lecture = CP[att-OB-lecture, attributs-O-Info'']

Invariants[5]

$\forall$ i : i, i' : Info

- $\forall$ i' : Existe-Objet(copies(i), i')
  $\Leftrightarrow$ i = a-pour-original(i') $\wedge$ i $\neq$ i'
  $\Leftrightarrow$ In(Copie(Info(Lire-Nom(i)), Info(Lire-Nom(i'))))
  $\Rightarrow$ Donner-Type(Lire-Nom(i)) = Donner-Type(Lire-Nom(i'))
- $\nexists$ i' : Existe-Objet(copies(i), i') $\wedge$ Lire-Nom(i) = Lire-Nom(i')
- $\exists$ i'' : i'' = a-pour-original(i) $\wedge$ i $\neq$ i'
  $\Leftrightarrow$ In(Copie(Info(Lire-Nom(i)), Info(Lire-Nom(i''))))
  $\Leftrightarrow$ Existe-Objet(copies(i''), i)

Interface

Ajouter-Copie : Info x Info $\rightarrow$ Info x Info
Emprunter : Info x STRING $\rightarrow$ Info
Restituer : Info $\rightarrow$ Info
Changer-Conf : Info x STRING $\rightarrow$ Info
Changer-Mot-De-Passe : Info x STRING x STRING $\rightarrow$ Info
A-Pour-Original : Info $\rightarrow$ Info

---

[5] Ces invariants traduisent au niveau des objets, les contraintes d'intégrité définies sur le type d'association récursive copie.

Lire-Première-Copie : Info → Info
Lire-Copie-Suivante : Info x Info → Info
Mot-Passe-Valide : Info x STRING → BOOL
Présence : Info → BOOL

Spécifications des méthodes

Modificateurs

• Ajouter-Copie(i-orig, i-copie) = <i-orig', i-copie'>

i-orig, i-copie, i-orig', i-copie' : Info

Précondition : Objet-Chargé(i-orig) ∧ Objet-Chargé(i-copie)

Postcondition : Copies(i-orig') = Ajouter-Objet(Copies(i-orig), i-copie) ∧ a-pour-original(i-copie') = i-orig'

• Emprunter(i, mot-passe) = i'

i, i' : Info
mot-passe : STRING
message : STRING

Précondition : Objet-Chargé(i)

Postcondition : Mot-Passe-Valide(i, mot-passe) ⇒ présence(i') = FALSE

Excepté : ¬ Mot-Passe-Valide(i, mot-passe) ⇒ message = 'mot de passe incorrect'

• Restituer(i) = i'

i, i' : Info

Précondition : Objet-Chargé(i)

Postcondition : présence(i') = TRUE

• Changer-Conf(i, mot-passe) = i'

i, i' : Info
mot-passe : STRING
message : STRING

Précondition : Objet-Chargé(i)

Postcondition : Mot-Passe-Valide(i, mot-passe) ⇒ confidentialité(i') = NOT(confidentialité(i))

Excepté : ¬ Mot-Passe-Valide(i, mot-passe) ⇒ message = 'mot de passe incorrect'

• Changer-Mot-De-Passe(i, anc-mp, nouv-mp) = i'

i, i' : Info
anc-mp, nouv-mp : STRING
message : STRING

Précondition : Objet-Chargé(i)

Postcondition : Mot-Passe-Valide(i, anc-mp) ⇒ mot-de-passe(i') = nouv-mp

Excepté : ¬ Mot-Passe-Valide(i, anc-mp) ⇒ message = 'mot de passe incorrect'

Observateurs

• A-Pour-Original(i) = i-orig — i, i-orig : Info

Précondition : Objet-Chargé(i)

Postcondition : i-orig = a-pour-original(i)

• Lire-Première-Copie(i) = c1 — i, c1 : Info

Précondition : Objet-Chargé(i)

Postcondition : c1' = Lire-Premier-Objet(copies(i))

• Lire-Copie-Suivante(i, c1) = c2 — i, c1, c2 : Info

Précondition : Objet-Chargé(i)

Postcondition : c2 = Lire-Objet-Suivant(copies(i), c1)

• Mot-Passe-Valide(i, mot-passe) = b — i : Info, mot-passe : STRING, b : BOOL

Précondition : Objet-Chargé(i)

Postcondition : b = (mot-de-passe(i) = mot-passe)

• Présence(i) = b — i : Info, b : BOOL

Précondition : Objet-Chargé(i)

Postcondition : b = présence(i)

OBJET : Collection

Structure

Collection is a O-Info, is a O-Rgble

with CP[att-Collection : attributs-Collection, organise : Liste-Objets]

Les structures de données intermédiaires se définissent de la manière suivante :

attributs-Collection = CP[critère-classement : STRING, ordre-classement : type-regroup]
att-Collection-création = CP[att-O-Rgble-création, attributs-Collection]
att-Collection-lecture = CP[att-OB-lecture, attributs-Collection]

Invariants

$\forall$ c : c : Collection
oi, oi' : O-Info

- $\forall$ oi : Existe-Objet(organise(c), oi)
  $\Leftrightarrow$ Existe-Objet(est-organisé-en(oi), c)
  $\Leftrightarrow$ In(Organisation(O-Info(Lire-Nom(oi)), Collection(Lire-Nom(c))))
  $\Rightarrow$ Donner-Type(Lire-Nom(oi)) = 'objet informationnel'
- $\nexists$ oi' : Existe-Objet(organise(c), oi') $\wedge$ Lire-Nom(oi') = Lire-Nom(c)

Interface

Ajouter-Composant : Collection x O-Info $\rightarrow$ Collection x O-Info
Retirer-Composant : Collection x O-Info $\rightarrow$ Collection x O-Info
Changer-Classement : Collection x STRING x type-regroup $\rightarrow$ Collection
Premier-Composant : Collection $\rightarrow$ O-Info
Composant -Suivant : Collection x O-Info $\rightarrow$ O-Info
Vide : Collection $\rightarrow$ BOOL

Spécifications des méthodes

Modificateurs

• Ajouter-Composant(c, oi) = <c', oi'>

c, c' : Collection
oi, oi' : Info
message : STRING

Précondition : Objet-Chargé(c) ∧ Objet-Chargé(oi)

Postcondition : c ≠ oi ⇒
Ajouter-Objet(organise(c), oi) = organise(c') ∧
Ajouter-Objet(est-organisé-en(oi), c) = est-organisé-en(oi')

Excepté : c = oi ⇒ message = 'objet identique'

• Retirer-Composant(c, oi) = <c', oi'>

c, c' : Collection
oi, oi' : Info

Précondition : Objet-Chargé(c) ∧ Objet-Chargé(oi)

Postcondition : c ≠ oi ⇒
Supprimer-Objet(organise(c), oi) = organise(c') ∧
Supprimer-Objet(est-organisé-en(oi), c) = est-organisé-en(oi')

Excepté : c = oi ⇒ message = 'objet identique'

• Changer-Classement(c, ccl, ocl) = c'

c, c' : Collection
ccl : STRING
ocl : type-regroup

Précondition : Objet-Chargé(c)

Postcondition : critère-classement(c') = ccl ∧ ordre-classement(c') = ocl

Observateurs

• Premier-Composant(c) = oi1

c : Collection
oi1 : O-Info

Précondition : Objet-Chargé(c)

Postcondition : oi1' = Lire-Premier-Objet(organise(c))

• Composant -Suivant (c, oi1) = oi2

c : Collection
oi1, oi2 : O-Info

Précondition : Objet-Chargé(c)

Postcondition : oi2 = Lire-Objet-Suivant(organise(c), oi1)

• Vide(c) = b

c : Collection
b : BOOL

Précondition : Objet-Chargé(c)

Postcondition : b = Test-Liste-Vide(organise(c))

OBJET : O-Term

Structure

O-Rgt-Term is a O-Rgt, is a O-Non-ORgble

Invariants

Aucun invariant n'est nécessaire pour cette classe d'objets. Elle doit simplement respecter les invariants des classes dont elle hérite les propriétés.

Interface

Les méthodes appartenant à l'interface de cette classe sont les méthodes des interfaces des classes dont elle hérite.

OBJET : Armoire

Structure

Armoire is a O-Term

Invariants

Aucun invariant n'est nécessaire pour cette classe d'objets. Elle doit simplement respecter les invariants des classes dont elle hérite les propriétés.

Interface

Les méthodes appartenant à l'interface de cette classe sont les méthodes des interfaces des classes dont elle hérite.

OBJET : Table

Structure

Table is a O-Term

Invariants

Aucun invariant n'est nécessaire pour cette classe d'objets. Elle doit simplement respecter les invariants des classes dont elle hérite les propriétés.

Interface

Les méthodes appartenant à l'interface de cette classe sont les méthodes des interfaces des classes dont elle hérite.

OBJET : Poubelle

Structure

Poubelle is a O-Term

Invariants

Aucun invariant n'est nécessaire pour cette classe d'objets. Elle doit simplement respecter les invariants des classes dont elle hérite les propriétés.

Interface

Les méthodes appartenant à l'interface de cette classe sont les méthodes des interfaces des classes dont elle hérite.

ETAT : BD

Structure

La structure de l'état BD est celle décrite par le schéma conceptuel des données présenté à la section 3.3 "Schéma E/A de l'environnement de bureau étendu".

Invariants

Les invariants de cet état sont de deux types : les invariants exprimés dans la sémantique du schéma conceptuel (contraintes de connectivité, type des attributs) et les invariants définis dans la partie 3.3.8 "Contraintes d'intégrité" qui présente les contraintes d'intégrité non directement exprimables dans le formalisme graphique du modèle E/A.

Remarque : Afin de ne pas confondre, dans les déclarations, les objets de la BD et les objets chargés en mémoire centrale, les objets de la BD seront déclarés en gras.

Spécifications de l'état

• Donner-type(nom) = t — nom, t : STRING

Précondition : Existe-OB(nom)

Postcondition : (In(Info(nom)) ⇒ t = 'information') ∨
(In(Collection(nom)) ⇒ t = 'collection') ∨
(In(O-Struct(nom)) ⇒ t = 'objet structurant') ∨
(In(Table(nom)) ⇒ t = 'table') ∨
(In(Armoire(nom)) ⇒ t = 'armoire') ∨
(In(Poubelle(nom)) ⇒ t = 'poubelle')

• Existe-OB-BD(nom) — nom : STRING

Précondition : /

Postcondition : In(OB(nom))

• Enregistrer-OB(nom, att) = o — nom : STRING, att : att-OB-création, o : OB

Précondition : ¬ Existe-OB(nom)

Postcondition : Existe-OB(nom) ∧
att-OB-création(OB(nom)) = att ∧ dates(OB(nom)) = date-jour ∧
att-OB(o) = att-OB(OB(nom)) ∧ nom(o) = nom

• Charger-OB(nom) = o — nom : STRING, o : OB

Précondition : Existe-OB(nom)

Postcondition : att-OB(o) = att-OB(OB(nom)) ∧ nom(o) = nom ∧ date-der-cons(OB(nom)) = date-jour[6]

• Recopier-OB(o) — o : OB

Précondition : Objet-Chargé(o)

Postcondition : att-OB(OB(Lire-Nom(o))) = Lire-Att(o)

• Détruire-OB(o) — o : OB

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-OB(Lire-nom(o))

• Existe-O-Rgt-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-OB(nom)∧ In(O-Rgt(nom))∧ In(Is-a(O-Rgt(nom), OB(nom)))

• Enregistrer-O-Rgt(nom, att) = o — nom : STRING, att : att-OB-création, o : O-Rgt

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-OB(nom, att) = o ∧ Existe-O-Rgt(nom)

• Charger-O-Rgt(nom) = o — o : O-Rgt, o', o'' : **O-Rgble**[7]

Précondition : Existe-O-Rgt(nom(o))

Postcondition : Charger-OB(nom) = o ∧
∀ o' : nom(o') ∈ est-le-lieu-de-rgt-origine(O-Rgt(Lire-Nom(o)))
⇔ Existe-Objet(contient-origine, Charger-O-Rgble(nom(o'))) ∧
∀ o'' : nom(o'') ∈ est-le-lieu-de-rgt-réel(O-Rgt(Lire-Nom(o'')))
⇔ Existe-Objet(contient-réel, Charger-O-Rgble(nom(o''))

---

[6] Mise à jour de la date de dernière consultation dans la BD puisque l'objet vient d'être chargé.

[7] Afin de ne pas confondre les objets de la BD avec les objets chargés en mémoire centrale, les objets BD seront déclarés en gras.

• Recopier-O-Rgt(o) — o : O-Rgt; o', o'' : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : Recopier-OB(o) ∧
∀ o' : Existe-Objet(contient-origine, o')
⇔ Lire-Nom(o') ∈ est-le-lieu-de-rgt-origine(O-Rgt(Lire-Nom(o))) ∧
∀ o'' : Existe-Objet(contient-réel, o'')
⇔ Lire-Nom(o'') ∈ est-le-lieu-de-rgt-réel((O-Rgt(Lire-Nom(o)))

• Détruire-O-Rgt(o) — o : O-Rgt

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-O-Rgt(Lire-Nom(o))

• Existe-O-Rgble-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-OB(nom) ∧ In(O-Rgble(nom)) ∧ Is-a(O-Rgble(nom), OB(nom))

• Enregistrer-O-Rgble(nom, att) = o — nom : STRING; att: att-O-Rgble-création; o : O-Rgble

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-OB(nom, att) = o ∧ Existe-O-Rgble(nom) ∧
a-pour-loc-origine(O-Rgble(nom)) = Lire-Nom(O-Rgt-origine(att))) ∧
a-pour-loc-réelle(O-Rgble(nom)) = Lire-Nom(O-Rgt-réel(att))) ∧
est-contenu-origine(o) = O-Rgt-origine(att) ∧
est-contenu-réel(o) = O-Rgt-réel(att)

• Charger-O-Rgble(nom) = o — nom : STRING; o : O-Rgble; o',o'' : **O-Rgt**

Précondition : Existe-O-Rgble(nom)

Postcondition : Charger-OB(nom) = o ∧
∃ o' : nom(o') = a-pour-loc-origine(O-Rgble(Lire-Nom(o)))
⇔ est-contenu-origine = Charger-O-Rgt(nom(o')) ∧
∃ o'' : nom(o'') = a-pour-loc-réelle(O-Rgble(Lire-Nom(o'')))
⇔ est-contenu-réel = Charger-O-Rgt(nom(o''))

• Recopier-O-Rgble(o) — o : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : Recopier-OB(o) ∧
a-pour-loc-orig(O-Rgble(Lire-Nom(o))) = Lire-Nom(est-contenu-orig(o)) ∧
a-pour-loc-réelle(O-Rgble(Lire-Nom(o))) = Lire-Nom(est-contenu-réel(o))

• Détruire-O-Rgble(o) — o : O-Rgble

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-O-Rgble(Lire-Nom(o))

• Existe-O-Info-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-OB(nom) ∧ In(O-Info(nom)) ∧ In(Is-a(O-Info(nom), OB(nom)))

• Enregistrer-O-Info(nom, att) = o — nom : STRING; att : att-O-Info-création; o : O-Info

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-OB(nom, att) = o ∧ Existe-O-Info(nom)

• Charger-O-Info(nom) = o — nom, m : STRING; o : O-Info; **c : Collection**

Précondition : Existe-O-Info(nom)

Postcondition : Charger-OB(nom) = o ∧
∀ m : m ∈ mots-clés(O-Info(nom)) ⇔ m ∈ mots-clés(o) ∧
∀ c : nom(c) ∈ est-organisé-en(O-Info(nom))
⇔ Existe-Objet(est-organisé-en(o),c)

• Recopier-O-Info(o) — o : O-Info; m : STRING; c : Collection

Précondition : Objet-Chargé(o)

Postcondition : Recopier-OB(o) ∧
∀ m : m ∈ mots-clés(o) ⇔ m ∈ mots-clés(O-Info(Lire-Nom(o))) ∧
∀ c : Existe-Objet(est-organisé-en(o),c)
⇔ Lire-Nom(c) ∈ est-organisé-en(O-Info(Lire-Nom(o)))

• Détruire-O-Info(o) — o : O-Info

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-O-Info(Lire-Nom(o))

• Existe-O-Non-Rgble-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-OB(nom) ∧ In(O-Non-Rgble(nom)) ∧
In(Is-a(O-Non-Rgble(nom), OB(nom)))

• Enregistrer-O-Non-Rgble(nom, att) = o — nom : STRING; att : att-O-Non-Rgble-création; o : O-Non-Rgble

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-OB(nom, att) = o ∧ Existe-O-Non-Rgble(nom)

• Charger-O-Non-Rgble(nom) = o — nom, m : STRING; o : O-Non-Rgble

Précondition : Existe-O-Non-Rgble(nom)

Postcondition : Charger-OB(nom) = o

• Recopier-O-Non-Rgble(o) — o : O-Non-Rgble

Précondition : Objet-Chargé(o)

Postcondition : Recopier-OB(o)

• Détruire-O-Non-Rgble(o) — o : O-Non-Rgble

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-O-Non-Rgble(Lire-Nom(o))

• Enregistrer-O-Struct(nom, att) = o — nom : STRING; att : att-O-Struct-création; o : O-Struct

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Rgble(nom, att-O-Rgble-création(att)) = o ∧
Enregistrer-O-Rgt(nom, att-OB-création(att)) = o ∧
Existe-O-Struct(nom) ∧ att-O-Struct(O-Struct(nom)) = att-O-Struct(att) ∧
att-O-Struct(o) = att-O-Struct(att)

• Charger-O-Struct(nom) = o — nom : STRING; o : O-Struct

Précondition : Existe-O-Struct(nom)

Postcondition : Charger-O-Rgt(nom) = o ∧ Charger-O-Rgble(nom) = o ∧
att-O-Struct(o) = att-O-Struct(O-Struct(nom))

• Recopier-O-Struct(o) — o : O-Struct

Précondition : Objet-Chargé(o)

Postcondition : Recopier-O-Rgt(o) ∧ Recopier-O-Rgble(o) ∧
att-O-Struct(O-Struct(Lire-Nom(o))) = att-O-Struct(o)

• Détruire-O-Struct(o) — o : O-Struct

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-O-Struct(Lire-Nom(o))

• Existe-Info-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-O-Rgble(nom) ∧ Existe-O-Info(nom) ∧ In(Info(nom)) ∧
In(Is-a(Info(nom), O-Rgble(nom))) ∧ In(Is-a(Info(nom), O-Info(nom)))

• Enregistrer-Info(nom, att) = i

nom : STRING
att : att-Info-création
i : Info

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Rgble(nom, att-O-Rgble-création(att)) = i ∧
Enregistrer-O-Info(nom, att-O-Info-création(att)) = i ∧
Existe-Info(nom) ∧
att-Info(Info(nom)) = att-Info(att) ∧ att-Info(i) = att-Info(att)

• Charger-Info(nom) = i

nom : STRING
i : Info
i' : **Info**

Précondition : Existe-Info(nom)

Postcondition : Charger-O-Rgble(nom) = i ∧ Charger-O-Info(nom) = i ∧
att-Info(i) = att-Info(Info(nom)) ∧
a-pour-original(i) = Charger-Info(est-une-copie-de(Info(nom))) ∧
∀ i' : Lire-Nom(i) = a-pour-copie(Info(nom(i')))
⇔ Existe-Objet(copies(i), Charger-Info(nom(i')))

• Recopier-Info(i)

i, i' : Info

Précondition : Objet-Chargé(o)

Postcondition : Recopier-O-Rgble(i) ∧ Recopier-Info(i) ∧
att-Info(Info(nom)) = att-Info(i) ∧
est-une-copie-de(Info(Lire-Nom(i))) = Lire-Nom(a-pour-original(i)) ∧
∀ i' : Existe-Objet(copies(i), i')
⇔ a-pour-copie(Info(Lire-Nom(i'))) = Lire-Nom(i)

• Détruire-Info(i)

i : Info

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-Info(Lire-Nom(i))

• Existe-Collection-BD(nom)

nom : STRING

Précondition : /

Postcondition : Existe-O-Rgble(nom) ∧ Existe-O-Info(nom) ∧ In(Collection(nom)) ∧
In(Is-a(Info(nom), O-Rgble(nom))) ∧ In(Is-a(Info(nom), O-Ìnfo(nom)))

• Enregistrer-Collection(nom, att) = c

nom : STRING
att :
att-Collection-création
c : Collection

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Rgble(nom, att-O-Rgble-création(att)) ∧
Enregistrer-O-Info(nom, att-O-Info-création(att)) ∧
Existe-Collection(nom) ∧ att-Collection(c) = att-Collection(att) ∧
att-Collection(Collection(nom)) = att-Collectiont(att)

• Charger-Collection(nom) = c — nom : STRING, c : Collection, oi : **O-Info**

Précondition : Existe-Collection(nom)

Postcondition : Charger-O-Rgble(nom) = c ∧ Charger-O-Info(nom) = c ∧
att-Collection(Collection(nom)) = att-Collectiont(c) ∧
∀ oi : nom(oi) ∈ organise(Collection(nom))
⇔ Existe-Objet(organise(c), Charger-O-Info(nom(oi)))

• Recopier-Collection(c) — c : Collection, oi : O-Info

Précondition : Objet-Chargé(o)

Postcondition : Recopier-O-Rgble(c) ∧ Recopier-Info(c) ∧
att-Collectiont(c) = ∧ att-Collection(Collection(nom))
∀ oi : Existe-Objet(organise(c), oi)
⇔ Lire-Nom(oi) ∈ organise(Collection(Lire-Nom(c)))

• Détruire-Collection(c) — c : Collection

Précondition : Objet-Chargé(o)

Postcondition : ¬ Existe-Collection(Lire-Nom(c))

• Existe-O-Term-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-O-Non-Rgble(nom) ∧ Existe-O-Rgt(nom) ∧ In(O-Term(nom)) ∧
In(Is-a(O-Term(nom), O-Non-Rgble(nom))) ∧
In(Is-a(O-Term(nom), O-Rgt(nom)))

• Enregistrer-O-Term(nom, att) = ot — nom : STRING, att : att-O-Term-création, ot : O-Term

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Non-Rgble(nom, att) = ot ∧
Enregistrer-O-Rgt(nom, att) = ot ∧ Existe-O-Non-Rgble(nom)

• Charger-O-Term(nom) = ot — nom : STRING, ot : O-Term

Précondition : Existe-O-Term(nom)

Postcondition : Charger-O-Non-Rgble(nom) = ot ∧ Charger-O-Rgt(nom) = ot

• Recopier-O-Term(ot) — ot : O-Term

Précondition : Objet-Chargé(ot)

Postcondition : Recopier-O-Non-Rgble(nom) = ot ∧ Recopier-O-Rgt(nom) = ot

• Détruire-O-Term(ot) — ot : O-Term

Précondition : Objet-Chargé(ot)

Postcondition : ¬ Existe-O-Term(Lire-Nom(ot))

• Existe-Armoire-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-O-Term(nom) ∧ In(Armoire(nom)) ∧
In(Is-a(Armoire(nom), O-Term(nom)))

• Enregistrer-Armoire(nom, att) = a — nom : STRING, att : att-Armoire-création, a : Armoire

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Term(nom, att) = a ∧ Existe-Armoire(nom)

• Charger-Armoire(nom) = a — nom : STRING, a : Armoire

Précondition : Existe-O-Term(nom)

Postcondition : Charger-O-Term(nom) = a

• Recopier-Armoire(a) — a : Armoire

Précondition : Objet-Chargé(a)

Postcondition : Recopier-O-Term(a)

• Détruire-Armoire(a) — a : Armoire

Précondition : Objet-Chargé(a)

Postcondition : ¬ Existe-Armoire(Lire-Nom(a))

• Existe-Table-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-O-Term(nom) ∧ In(Table(nom)) ∧
In(Is-a(Table(nom), O-Term(nom)))

• Enregistrer-Table(nom, att) = t — nom : STRING, att : att-Table-création, t : Table

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Term(nom, att) = t ∧ Existe-Table(nom)

• Charger-Table(nom) = t — nom : STRING, t : Table

Précondition : Existe-O-Term(nom)

Postcondition : Charger-O-Term(nom) = t

• Recopier-Table(t) — t : Table

Précondition : Objet-Chargé(t)

Postcondition : Recopier-O-Term(t)

• Détruire-Table(t) — t : Table

Précondition : Objet-Chargé(t)

Postcondition : ¬ Existe-O-Term(Lire-Nom(t))

• Existe-Poubelle-BD(nom) — nom : STRING

Précondition : /

Postcondition : Existe-O-Term(nom) ∧ In(Poubelle(nom)) ∧ In(Is-a(Poubelle(nom), O-Term(nom)))

• Enregistrer-Poubelle(nom, att) = p — nom : STRING; att : att-Poubelle-création; p : Poubelle

Précondition : ¬ Existe-OB(nom)

Postcondition : Enregistrer-O-Term(nom, att) = t ∧ Existe-Poubelle(nom)

• Charger-Poubelle(nom) = p — nom : STRING; p : Poubelle

Précondition : Existe-O-Term(nom)

Postcondition : Charger-O-Term(nom) = p

• Recopier-Poubelle(p) — p : Poubelle

Précondition : Objet-Chargé(p)

Postcondition : Recopier-O-Term(p)

• Détruire-Poubelle(p) — p : Poubelle

Précondition : Objet-Chargé(p)

Postcondition : ¬ Existe-O-Term(Lire-Nom(p))

Annexe 3 :
Spécification du contrôleur d'accès

Introduction

Cette annexe reprend les spécifications des classes d'objets constituant le contrôleur d'accès. Rapellons que le contrôleur est né suite à deux décisions de conception :

- l'introduction de la notion de cycle de vie des objets,
- le chargement des objets consultés en mémoire centrale dans la table résidente.

Le rôle du contrôleur d'accès est de gérer l'accès aux objets. Il remplira donc les tâches suivantes :

- la gestion des entrées de la table résidente, notamment l'allocation et la désallocation des espaces réservés aux variables objets du programme,
- le contrôle d'accès lors de toute demande de chargement d'un objet, c'est-à-dire la vérification de l'absence d'un cycle de vie déjà entamé pour la variable objet demandant l'accès,
- les contrôles d'exécution lors de l'appel des méthodes de classes d'objets du niveau BD. En effet, les méthodes appelées par l'utilisateur doivent être filtrées par le contrôleur afin qu'il vérifie si l'objet concerné par l'appel n'a pas été détruit et correspond à un cycle de vie entamé. Ce filtre se traduit dans les postconditions par la présence de la méthode Objet-Chargé(o) qui gère les exceptions (c'est-à-dire les conditions de fin anormales lors de l'appel des méthodes).

La description détaillée du fonctionnement du contrôleur d'accès est réalisée à la partie "Duplication de la hiérarchie : contrôle d'accès aux objets".

Finalement, nous voudrions attirer l'attention sur les conventions et remarques abordées dans l'annexe 2 "Spécification des modules d'accès aux objets de la base de données". Elles sont aussi d'application ici.

OBJET : Objet-Bureau

Structure

Objet-Bureau = entrée-table : INTEGER

Invariants

$\forall$ o : Existe-Num-Place(t, entrée-table(o)) o : Objet-Bureau

Un invariant inexprimable à l'aide du langage de spécification choisi s'énonce comme suit : durant toute la durée de vie de la variable o la valeur prise par entrée-table est une constante (non modifiable).

Interface

A ce niveau de spécifications, les signatures des méthodes connues par l'utilisateur sont les signatures fournies au niveau des modules d'accès, il n'y a donc aucune raison de les rappeler ici.

Utilise

Ces objets utilisent les méthodes fournies par les classes d'objets définies au niveau des modules d'accès à la base de données. Les méthodes du contrôleur d'accès utilisent aussi le tableau t , instance de la classe Table-Res. Il constitue un état par les classes d'objets de ce niveau.

Spécifications des méthodes

Constructeurs

• Init() = obj

obj : Objet-Bureau
i : INTEGER

Précondition : /

Postcondition : Allouer-Place(t_{PRE}) = <t,i> $\wedge$ entrée-table(obj) = i

Modificateurs

• Créer(obj, nom, att) = obj'

obj, obj' : Objet-Bureau
o : OB, nom : STRING
att : att-OB-création
e : Elément-Table
message : STRING

Précondition : /

Postcondition : Libre-Place(t_{PRE}, entrée-table(objet))
⇒ Créer[1](nom, liste-att) = o ∧
nom(e) = nom ∧ occupation(e) = TRUE ∧
objet(e) = o ∧ Garnir-Place(t_{PRE}, e, entrée-table(obj)) = t

Excepté : ¬ Libre-Place(t_{PRE}, entrée-table(obj)) ⇒ message = 'cycle de vie entamé'

• Accès-Direct(obj, nom) = obj'

obj, obj' : Objet-Bureau
o : OB, nom : STRING
e : Elément-Table
message : STRING

Précondition : /

Postcondition : Libre-Place(t_{PRE}, entrée-table(obj))
⇒ Accès-Direct(nom) = o ∧
nom(e) = nom ∧ occupation(e) = TRUE ∧
objet(e) = o ∧ Garnir-Place(t_{PRE}, e, entrée-table(obj)) = t

Excepté : ¬ Libre-Place(t_{PRE}, entrée-table(obj)) ⇒ message = 'cycle de vie entamé'

• Modifier-Att(obj, att) = obj'

obj, obj' : Objet-Bureau
att : att-OB-création
o, o' : OB

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ Modifier-Att(o, liste-att) = o' ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

• Fin-Utilisation(obj) = obj'

obj, obj' : Objet-Bureau
o : OB

Précondition : /

Postcondition : ¬ Libre-Place(t_{PRE}, entrée-table(obj))
⇒ if (Lire-Place(t_{PRE}, entrée-table(obj)) ≠ OBvide
then [Recopier-BD(o) ∧
o = Lire-Place(t_{PRE}(obj), entrée-table(obj))] ∧
Libérer-Objet(t_{PRE}, entrée-table(obj)) = t)

Excepté : Libre-Place(t_{PRE}, entrée-table(obj))
⇒ message = 'cycle de vie non entamé'

---

[1] Il s'agit bien de la méthode Créer définie au niveau de la classe OB des modules d'accès à la BD. Le type de l'objet o renvoyé par l'appel de la méthode le confirme.

• Supprimer(obj) = obj'

obj, obj' : Objet-Bureau
o : OB

Précondition : /

Postcondition : ¬ Libre-Place(t_{PRE}, entrée-table(obj))
⇒ if (Lire-Place(t_{PRE}, entrée-table(obj)) ≠ OBvide
then [Supprimer(o) ∧ o = Lire-Place(t_{PRE}, entrée-table(obj))] ∧
Annuler-Place(t_{PRE}, entrée-table(obj)) = t

Excepté : Libre-Place(t_{PRE}, entrée-table(obj))
⇒ message = 'cycle de vie non entamé'

• Fini(obj)

obj : Objet-Bureau

Précondition : /

Postcondition : Libérer-Place(t_{PRE}, entrée-table(obj)) = t

Observateurs

• Lire-Att(obj) = liste-att

obj : Objet-Bureau
att : att-OB-lecture
o : OB

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ liste-att = Lire-Att(o) ∧ o = Lire-Place(t_{PRE}, entrée-table(obj))

• Lire-Nom(obj) = nom

obj : Objet-Bureau
nom : STRING
o : OB

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ nom = Lire-Nom(o) ∧ o = Lire-Place(t_{PRE}, entrée-table(obj))

• Existe-Objet(nom)

nom : STRING

Précondition : /

Postcondition : Existe-OB(nom)

• Objet-Chargé(obj) = b

obj : Objet-Bureau
b : BOOL
message : STRING

Précondition : /

Postcondition : if ¬ Libre-Place(t_{PRE}, entrée-table(obj)) ∧
Lire-Place(t_{PRE}, entrée-table(obj)) ≠ OBvide
then b = TRUE
else b = FALSE

Excepté : • Libre-Place(t_{PRE}, entrée-table(obj))
⇒ message = 'cycle de vie non entamé'
• ¬ Libre-Place(t_{PRE}, entrée-table(obj)) ∧
Lire-Place(t_{PRE}, entrée-table(obj)) = OBvide
⇒ message = 'objet détruit'

OBJET : Objet-Rgt

Structure

Objet-Rgt is a Objet-Bureau

Invariants

Les invariants définis pour cette classe sont identiques à ceux définis pour la classe Objet-Bureau. Il en est de même pour toutes les classes descendantes, nous omettrons donc la section invariant dans les spécifiations.

Spécifications des méthodes

Observateurs

• Tester-Localisation-Origine(obj) = b

b : BOOL
obj : Objet-Rgt
o : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ Tester-Localisation-Origine(o) = b ∧
o = Lire-Place(t_{PRE}, entrée-table(obj))

• Premier-O-Rgble-Loc-Origine(obj) = obj'

obj : Objet-Rgt
o : O-Rgt
obj' : Objet-Rgble
o' : O-Rgble

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ o' = Premier-O-Rgble-Loc-Origine(o) [2]∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

• O-Rgble-Loc-Origine-Suivant(obj, obj1') = obj2'

obj : Objet-Rgt
o : O-Rgt
o1', o2' : O-Rgble
obj1', obj2' : Objet-Rgble

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ o2' = O-Rgble-Loc-Origine-Suivant(o, o1') ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o1' = Lire-Place(t_{PRE}, entrée-table(obj1')) ∧
o2' = Lire-Place(t, entrée-table(obj2'))

---

[2] Donc l'objet lu est placé dans la table résidente, l'entrée correspondant est placée dans la variable objet déclarée par l'utilisateur.

• Tester-Localisation-Réelle(obj) = b

b : BOOL
obj : Objet-Rgt
o : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ Tester-Localisation-Réelle(o) = b ∧
o = Lire-Place(t_{PRE}, entrée-table(obj))

• Premier-O-Rgble-Loc-Réelle(obj) = obj'

obj : Objet-Rgt
o : O-Rgt
obj' : Objet-Rgble
o' : O-Rgble

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ o' = Premier-O-Rgble-Loc-Réelle(o) ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

• O-Rgble-Loc-Réelle-Suivant(obj, obj1) = obj2'

obj : Objet-Rgt
o : O-Rgt
o1, o2' : O-Rgble
obj1, obj2' : Objet-Rgble

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ o2' = O-Rgble-Loc-Réelle-Suivant(o, o1) ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o1 = Lire-Place(t_{PRE}, entrée-table(obj1)) ∧
o2' = Lire-Place(t, entrée-table(obj2'))

OBJET : Objet-Rgble

Structure

Objet-Rgble is a Objet-Bureau

Spécifications des méthodes

Modificateurs

• Déplacer(obj, obj-orig, obj-dest) = <obj', obj-orig', obj-dest'>

obj, obj' : Objet-Rgble
obj-orig, obj-orig',
obj-dest, obj-dest' :
Objet-Rgt
o, o' : O-Rgble
o-orig, o-orig', o-dest,
o-dest' : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj) ∧
Objet-Chargé(obj-orig) ∧
Objet-Chargé(obj-dest)[3]

⇒ Déplacer(o, o-orig, o-dest) = <o', o-orig, o-dest> ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj')) ∧
o-orig = Lire-Place(t_{PRE}, entrée-table(obj-orig)) ∧
o-orig' = Lire-Place(t, entrée-table(obj-orig')) ∧
o-dest = Lire-Place(t_{PRE}, entrée-table(obj-dest)) ∧
o-dest' = Lire-Place(t, entrée-table(obj-dest'))

• Changer-Origine(obj, obj-orig, obj-dest) =
<obj', obj-orig', obj-dest'>

obj, obj' : Objet-Rgble
obj-orig, obj-orig',
obj-dest, obj-dest' :
Objet-Rgt
o, o' : O-Rgble
o-orig, o-orig', o-dest,
o-dest' : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj) ∧
Objet-Chargé(obj-orig) ∧
Objet-Chargé(obj-dest)

⇒ Changer-Origine(o, o-orig, o-dest) = <o', o-orig, o-dest> ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj')) ∧
o-orig = Lire-Place(t_{PRE}, entrée-table(obj-orig)) ∧
o-orig' = Lire-Place(t, entrée-table(obj-orig')) ∧
o-dest = Lire-Place(t_{PRE}, entrée-table(obj-dest)) ∧
o-dest' = Lire-Place(t, entrée-table(obj-dest'))

---

[3] Il faut donc que les trois objets passés en paramètres à la primitive soient chargés en mémoire centrale et n'aient pas été détruit par l'utilisation de la méthode Supprimer.

• Replacer(obj, obj-orig, obj-dest) = <obj', obj-orig', obj-dest'>

obj, obj' : Objet-Rgble
obj-orig, obj-orig',
obj-dest, obj-dest' :
Objet-Rgt
o, o' : O-Rgble
o-orig, o-orig', o-dest,
o-dest' : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj) ∧
Objet-Chargé(obj-orig) ∧
Objet-Chargé(obj-dest)

⇒ Replacer(o, o-orig, o-dest) =<o', o-orig, o-dest> ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj')) ∧
o-orig = Lire-Place(t_{PRE}, entrée-table(obj-orig)) ∧
o-orig' = Lire-Place(t, entrée-table(obj-orig')) ∧
o-dest = Lire-Place(t_{PRE}, entrée-table(obj-dest)) ∧
o-dest' = Lire-Place(t, entrée-table(obj-dest'))

Observateurs

• Loc-Origine(obj) = obj-rgt

obj : Objet-Rgble
obj-rgt : Objet-Rgt
o : O-Rgble
o-rgt : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ o-rgt = Loc-Origine(o) ∧
o = Lire-Place(t, entrée-table(obj)) ∧
o-rgt = Lire-Place(t, entrée-table(obj-rgt))

• Loc-Réelle(obj) = obj-rgt

obj : Objet-Rgble
obj-rgt : Objet-Rgt
o : O-Rgble
o-rgt : O-Rgt

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ o-rgt = Loc-Réelle(o) ∧
o = Lire-Place(t, entrée-table(obj)) ∧
o-rgt = Lire-Place(t, entrée-table(obj-rgt))

OBJET : Objet-Info

Structure

Objet-Info is a Objet-Bureau

Spécifications des méthodes

Modificateurs

• Ajouter-Mot-Clé(obj, mc) = obj'

obj, obj' : Objet-Info
o, o' : O-Info

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ Ajouter-Mot-Clé(o, mc) = o'

o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

• Supprimer-Mot-Clé(obj, mc) = obj'

obj, obj' : Objet-Info,
mc : STRING
o, o' : O-Info

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ Supprimer-Mot-Clé(o, mc) = o' ∧

o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

Observateurs

• Lister-Mots-Clé(obj) = lmc

obj : Objet-Info,
lmc : SEQ[STRING]
o : O-Info

Précondition : /

Postcondition : Objet-Chargé(obj)

⇒ Lister-Mot-Clé(o, mc) = lmc ∧

o = Lire-Place(t_{PRE}, entrée-table(obj))

OBJET : Objet-Non-Rgble

Structure

O-Non-Rgble is a OB

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

OBJET : Objet-Struct

Structure

Objet-Struct is a Objet-Rgt, is a Objet-Rgble

Spécifications des méthodes

Modificateurs

• Etiqueter(obj, e) = obj'

obj, obj' : Objet-Struct
e : STRING
o, o' : O-Struct

Précondition : /

Postcondition : Objet-Chargé(obj) ⇒ Etiqueter(o, e) = o' ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

• Changer-Regroupement(obj, cr, or) = obj'

obj, obj' : Objet-Struct
cr : STRING
or : type-regroup
o, o' : O-Struct

Précondition : /

Postcondition : Objet-Chargé(obj)
⇒ Changer-Regroupement(o, cr, or) = o' ∧
o = Lire-Place(t_{PRE}, entrée-table(obj)) ∧
o' = Lire-Place(t, entrée-table(obj'))

Observateurs

• Lire-étiquette(obj) = e

obj : Objet-Struct
e : STRING
o : O-Struct

Précondition : /

Postcondition : Objet-Charge(obj)
⇒ Lire-Etiquette(o) = e ∧ o = Lire-Place(t_{PRE}, Lire-étiquette(obj) = e

OBJET : Information

Structure

Information is a Objet-Info, is a Objet-Rgble

Spécifications des méthodes

Modificateurs

• Copier(info-orig, info-copie) = <info-orig', info-copie'>

info-orig, info-copie, info-orig',info-copie' : Information
i-orig, i-copie, i-orig', i-copie' : Info
att : att-OB-lecture

Précondition : /

Postcondition : Objet-Chargé(info-orig) ∧ Libre-Place(info-copie)
⇒ Créer(nom, att-OB-création(att)) = i-copie ∧
att = Lire-Att(i-orig) ∧
Ajouter-Copie(i-orig, i-copie) = <i-orig', i-copie'> ∧
orig = Lire-Place(t_{PRE}, entrée-table(info-orig)) ∧
i-orig' = Lire-Place(t_{PRE}, entrée-table(info-orig')) ∧
i-copie' = Lire-Place(t, entrée-table(info-copie'))

Excepté : ¬ Libre-Place(t_{PRE}, entrée-table(info-copie))
⇒ message = 'cycle de vie entamé'

• Emprunter(info, mot-passe) = info'

info, info' : Information
i, i' : Info
mot-passe : STRING

Précondition : /

Postcondition : Objet-Chargé(info)
⇒ Emprunter(i, mot-passe) = i' ∧
i = Lire-Place(t_{PRE}, entrée-table(info)) ∧
i' = Lire-Place(t, entrée-table(info'))

• Restituer(info) = info'

info, info' : Information
i, i' : Info

Précondition : /

Postcondition : Objet-Chargé(info)
⇒ Restituer(i) = i' ∧
i = Lire-Place(t_{PRE}, entrée-table(info)) ∧
i' = Lire-Place(t, entrée-table(info'))

• Changer-Conf(info, mot-passe) = info'

info, info' : information
i, i' : Info
mot-passe : STRING

Précondition : /

Postcondition : Objet-Chargé(info)

⇒ Changer-Conf(i, mot-passe) = i' ∧

i = Lire-Place(t_{PRE}, entrée-table(info)) ∧

i' = Lire-Place(t, entrée-table(info'))

• Changer-Mot-De-Passe(info, anc-mp, nouv-mp) = info'

info, info' : Information
i, i' : Info
anc-mp, nouv-mp : STRING

Précondition : /

Postcondition : Objet-Chargé(info)

⇒ Changer-Mot-De-Passe(i, anc-mp, nouv-mp) = i'

i = Lire-Place(t_{PRE}, entrée-table(info)) ∧

i' = Lire-Place(t, entrée-table(info'))

Observateurs

• A-Pour-Original(info) = info-orig

info, info-orig : information
i, i-orig : Info

Précondition : /

Postcondition : Objet-Chargé(info)

⇒ A-Pour-Original(i) = i-orig ∧

i = Lire-Place(t_{PRE}, entrée-table(info)) ∧

i-orig = Lire-Place(t_{PRE}, entrée-table(info-orig'))

• Lire-Première-Copie(info) = copie1

info, copie1 : Information
i, c1 : Info

Précondition : /

Postcondition : Objet-Chargé(info)

⇒ Lire-Première-Copie(i) = c1 ∧

i = Lire-Place(t_{PRE}, entrée-table(info)) ∧

c1 = Lire-Place(t_{PRE}, entrée-table(copie1'))

• Lire-Copie-Suivante(info, copie1) = copie2

info, copie1, copie2 : information
i, c1, c2 : Info

Précondition : Objet-Chargé(i)

Postcondition : Objet-Chargé(info)

⇒ Lire-Copie-Suivante(i, c1) = c2 ∧

i = Lire-Place(t_{PRE}, entrée-table(info)) ∧

c1 = Lire-Place(t_{PRE}, entrée-table(copie1)) ∧

c2 = Lire-Place(t_{PRE}, entrée-table(copie2))

• Mot-Passe-Valide(info, mot-passe) = b

info : Information
i : Info
mot-passe : STRING
b : BOOL

Précondition : /

Postcondition : Objet-Chargé(info)
⇒ Mot-Passe-Valide(i, mot-passe) = b ∧
i = Lire-Place(t_{PRE}, entrée-table(info))

• Présence(info) = b

info : Information
i : Info
b : BOOL

Précondition : /

Postcondition : Objet-Chargé(info)
⇒ Présence(i) = b ∧ i = Lire-Place(t_{PRE}, entrée-table(info))

OBJET : Objet-Collection

Structure

Objet-Collection is a Objet-Info, is a Objet-Rgble

Spécifications des méthodes

Modificateurs

• Ajouter-Composant(col, o-info) = <col', o-info'>

Précondition : /

Postcondition : Objet-Chargé(col) ∧ Objet-Chargé(o-info)
⇒ Ajouter-Composant(c, oi) = <c', oi'> ∧
oi = Lire-Place(t_{PRE}, entrée-table(o-info)) ∧
oi' = Lire-Place(t, entrée-table(o-info')) ∧
c = Lire-Place(t_{PRE}, entrée-table(col)) ∧
c' = Lire-Place(t, entrée-table(col'))

o-info, o-info' : Information
col, col' : Objet-Collection
c, c' : Collection
oi, oi' : O-Info

• Retirer-Composant(c, oi) = <c', oi'>

Précondition : /

Postcondition : Objet-Chargé(col) ∧ Objet-Chargé(o-info)
⇒ Retirer-Composant(c, oi) = <c', oi'> ∧
oi = Lire-Place(t_{PRE}, entrée-table(o-info)) ∧
oi' = Lire-Place(t, entrée-table(o-info')) ∧
c = Lire-Place(t_{PRE}, entrée-table(col)) ∧
c' = Lire-Place(t, entrée-table(col'))

o-info, o-info' : Information
col, col' : Objet-Collection
c, c' : Collection
oi, oi' : O-Info

• Changer-Classement(col, ccl, ocl) = col'

Précondition : /

Postcondition : Objet-Chargé(col)
⇒ Changer-Classement(c, ccl, ocl) = c' ∧
c = Lire-Place(t_{PRE}, entrée-table(col)) ∧
c' = Lire-Place(t, entrée-table(col'))

col, col' : Objet-Collection
c, c' : Collection
ccl : STRING
ocl : type-regroup

Observateurs

• Premier-Composant(col) = oi1

col : Objet-Collection
c : Collection
o-info1 : Informaiton
oi1 : O-Info

Précondition : /

Postcondition : Objet-Chargé(col)
⇒ Premier-Composant(c) = oi1 ∧
c = Lire-Place(t_{PRE}, entrée-table(col)) ∧
oi1 = Lire-Place(t, entrée-table(o-info1'))

• Composant -Suivant (col, oi1) = oi2

col : Objet-Collection
c : Collection
o-info1, o-info2 : Information
oi1, oi2 : O-Info

Précondition : /

Postcondition : Objet-Chargé(col)
⇒ Composant-Suivant(c,oi1) = oi2 ∧
c = Lire-Place(t_{PRE}, entrée-table(col)) ∧
oi1 = Lire-Place(t_{PRE}, entrée-table(o-info1')) ∧
oi2 = Lire-Place(t, entrée-table(o-info2'))

• Vide(col) = b

col : Objet-Collection
c : Collection
b : BOOL

Précondition : /

Postcondition : Objet-Chargé(col)
⇒ b = Vide(c) ∧
c = Lire-Place(t_{PRE}, entrée-table(col))

OBJET : Objet-Terminal

Structure

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

OBJET : Objet-Armoire

Structure

Objet-Armoire is a Objet-Terminal

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

OBJET : Objet-Table

Structure

Objet-Table is a Objet-Terminal

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

OBJET : Objet-Poubelle

Structure

Objet-Poubelle is a Objet-Terminal

Interface

Aucune méthode spécifique à cette classe d'objets n'a été définie. Il ne faut pas perdre de vue qu'elle hérite de toutes les méthodes définies au niveau des objets de bureau.

Annexe 4 :
Spécification des objets complémentaires

Introduction

Cette annexe est consacrée à la spécification formelle des classes d'objets n'appartenant ni aux modules du contrôleur d'accès ni aux modules d'accès à la base de données.

Il s'agit de :

- la classe **Liste-Objets** qui décrit la structure de données choisie pour représenter en mémoire centrale, les types d'associations dans lesquelles sont impliquées les objets chargés. Les raisons qui ont conduit à l'introduction de cette classe sont expliquées à la section 4.2 "Spécification formelle des classes d'objets",
- la classe **Table-Res** destinée à contenir les objets chargés en mémoire centrale. C'est une instance de cette classe qui sera utilisée par le contrôleur d'accès pour qu'il puisse gérer l'accès aux objets.

OBJET : Liste-Objets

Structure

Liste-Objets = SEQ[objet]

objet: OB

Invariants

Init : Length(l) = 0

l : Liste-Objets
o,o' : OB

$\nexists$ o,o' $\in$ l : o $\neq$ o' $\wedge$ Lire-Nom(o) = Lire-Nom(o')

Interface

Créer-Liste-Objets : $\rightarrow$ Liste-Objets
Ajouter-Objet : Liste-Objets x OB $\rightarrow$ Liste-Objets
Supprimer-Objet : Liste-Objets x OB $\rightarrow$ Liste-Objets
Transférer-Objet : Liste-Objets x Liste-Objets x OB $\rightarrow$ Liste-Objets x Liste-Objets
Transférer-Liste : Liste-Objets x Liste-Objets $\rightarrow$ Liste-Objets x Liste-Objets
Taille-Liste-Objets : Liste-Objets $\rightarrow$ INTEGER
Lire-Premier-Objet : Liste-Objets $\rightarrow$ OB
Lire-Objet-Suivant : Liste-Objets x OB $\rightarrow$ OB
Existe-Objet : Liste-Objets x OB $\rightarrow$ BOOL
Test-Liste-Vide : Liste-Objets $\rightarrow$ BOOL

Spécifications des méthodes

Constructeurs

• Créer-Liste-Objets() = l

l : Liste-Objets

Précondition : /

Postcondition : l = []

• Ajouter-Objet(l, o) = l'

l, l' : Liste-Objets
o : OB

Précondition : /

Postcondition : l' = Append(l, o)

Modificateurs

• Supprimer-Objet(l, o) = l'

l, l' : Liste-Objets
o : OB
i : INTEGER

Précondition : Existe-Objet(l, o)

Postcondition : $\exists$ i : $1 \leq i \leq$ Length(l) : l_i = o $\wedge$ l' = Remove-ith(l,i)

• Transférer-Objet(l1, l1', o) = (l2, l2')

l1, l1', l2, l2' : Liste-Objets
o : OB
i : INTEGER

Précondition : Existe-Objet(l1, o) ∧ ¬ Existe-Objet(l1', o)

Postcondition : $\exists\, i : 1 \leq i \leq \text{Length}(l1) : l1_i = o \wedge l2 = \text{Remove-ith}(l1,i) \wedge l2' = \text{Append}(l1', o)$

• Transférer-Liste(l1, l1') = (l2,l2')

l1, l1', l2, l2' : Liste-Objets

Précondition : /

Postcondition : l2 = [] ∧ l2' = l1 + l1'

Observateurs

• Taille-Liste-Objets(l) = i

l : Liste-Objets
i : INTEGER

Précondition : /

Postcondition : i = Length(l)

• Lire-Premier-Objet(l) = o

l : Liste-Objets
o : OB

Précondition : /

Postcondition : if Taille-Liste-Objets(l) = 0
then o = OBvide
else o = First(l)

• Lire-Objet-Suivant(l, o) = o'

l : Liste-Objets
o, o' : OB

Précondition : Existe-Objet(l, o)

Postcondition : if o = Last(l)
then o' = OBvide
else $\exists\, i : 1 \leq i < \text{Length}(l) : l_i = o \wedge l_{i+1} = o'$

• Existe-Objet(l, o)

l : Liste-Objets
o : OB

Précondition : /

Postcondition : (o ∈ l)

• Test-Liste-Vide(l)

l : Liste-Objets

Précondition : /

Postcondition : (Length(l) = 0)

OBJET : Table-Res

Structure

Element-Table = CP[nom : STRING, occupation : BOOl, objet : OB]

Table-Res = [INTEGER → element]

element : Element-Table

Invariants

Init : Empty(t)

t : Table-Res
e,e': Element-Table

∀ e ∈ In-Codom(t) : nom(e) = Lire-Nom(objet(e))

∀ e, e' ∈ In-Codom(t) : objet(e) = objet(e') ⇔ nom(e) = nom(e')

Interface

Init : → Table-Res
Allouer-Place : Table-Res → Table-Res x INTEGER
Garnir-Place : Table-Res x Element-Table x INTEGER → Table-Res
Recopier-Place : Table-Res x INTEGER x INTEGER → Table-Res
Libérer-Place : Table-Res x INTEGER → Table-Res
Annuler-Place : Table-Res x STRING → Table-Res
Libérer-Objet : Table-Res x INTEGER → Table-Res
Existe-Num-Place : Table-Res x INTEGER → BOOL
Existe-Place : Table-Res x STRING → BOOL
Existe-Place-Multiple : Table-Res x STRING → BOOL
Libre-Place : Table-Res x INTEGER → BOOL
Lire-Place : Table-Res x INTEGER → OB
Lire-Num-Place : Table-Res x STRING → INTEGER
Lire-Nom-Place : Table-Res x INTEGER → STRING

Spécifications des méthodes

Constructeurs

• Init( ) = t

t : Table-Res

Précondition : /

Postcondition : t = { }

• Allouer-Place(t) = (t',i) — t, t' : Table-Res; i : INTEGER

Précondition : /

Postcondition : $i \notin$ In-Dom(t) $\wedge$ t' = Insert (t, i, <" ", FALSE, OBvide>)

Modificateurs

• Garnir-Place(t, e, i) = t' — t, t' : Table-Res; e : Element-Table; i : INTEGER

Précondition : Existe-Num-Place(i) $\wedge$ Libre-Place(i)

Postcondition : t' = Modify(t, i, e)

• Recopier-Place(t, i, j) = t' — t, t' : Table-Res; i, j : INTEGER

Précondition : Existe-Num-Place(t, i) $\wedge \neg$ Libre-Place(t, i) $\wedge$ Existe-Num-Place(t, j) $\wedge$ Libre-Place(t, j)

Postcondition : t' = Modify(t, j, t[i])

• Libérer-Place(t, i) = t' — t, t' : Table-Res; i : INTEGER

Précondition : Existe-Num-Place(t, i) $\wedge$ Libre-Place(t, i)

Postcondition : t' = Delete(t, i)

• Annuler-Place(t, s) = t' — t, t' : Table-Res; s : STRING; i : INTEGER

Précondition : Existe-Place(t, s)

Postcondition : $\forall$ i $\in$ In-Dom(t) : nom(t[i]) = s $\Leftrightarrow$ objet(t[i]) = OBvide

• Libérer-Objet(t, i) = t' — t, t' : Table-Res; i : INTEGER

Précondition : Existe-Num-Place(t, i)

Postcondition : t' = Modify(t, i, <" ", FALSE, OBvide>)

Observateurs — t : Table-Res; i : INTEGER

• Existe-Num-Place(t, i)

Précondition : /

Postcondition : (i $\in$ In-Dom(t))

• Existe-Place(t, s) — t : Table-Res; i : INTEGER; s : STRING

Précondition : /

Postcondition : ($\exists$ i $\in$ In-Dom(t) : nom(t[i]) = s $\wedge \neg$ Libre-Place(t, i))

• Existe-Place-Multiple(t, s)

t : Table-Res
i, i' : INTEGER
s : STRING

Précondition : /

Postcondition : (∃ i, i' ∈ In-Dom(t) : Existe-Place(t, Lire-Nom-Place(t, i)) ∧ Existe-Place(t, Lire-Nom-Place(t, i'))

• Libre-Place(t, i)

t : Table-Res
i : INTEGER

Précondition : Existe-Num-Place(t, i)

Postcondition : (occupation(t[i]) = FALSE)

• Lire-Place(t, i) = o

t : Table-Res
i : INTEGER
o : OB

Précondition : Existe-Num-Place(t, i) ∧ ¬ Libre-Place(t, i)

Postcondition : o = objet(t[i])

• Lire-Num-Place(t, s) = i

t : Table-Res
i : INTEGER
s : STRING

Précondition : Existe-Place(t, s)

Postcondition : ∃ i ∈ In-Dom(t) : nom(t[i]) = s

• Lire-Nom-Place(t, i) = s

t : Table-Res
i : INTEGER
s : STRING

Précondition : Existe-Num-Place(t, i)

Postcondition : ∃ s : Lire-Num-Place(t, s) = i

Spécifications d'implémentation

La structure logique de Table-Res est complétée au niveau implémentation par l'ajout de l'attribut **modification** qui permet de garder une trace de modifications réalisées par l'utilisateur sur les objets présents dans la Table-Res. Seuls les objets ayant été modifiés seront recopiés dans la BD, cette technique réduit le nombre de lectures/écritures sur le disque.

A ce stade, une longueur maximale, fixée à 100 est assignée à la Table-Res.

Structure

Element-Table = CP[nom : STRING, occupation : BOOl, objet : OB,
modification : BOOL]

Invariants

∀ t : length(t) ≤ 100

t : Table-Res

∀ i, j ∈ In-Dom(t) :

i,j : INTEGER

objet(t[i]) = objet(t[j]) ⇒ modification(t[i]) = modification(t[j])

Interface

Modification-Vrai : Table-Res x INTEGER → Table-Res
Nom-Modification-Vrai : Table-Res x INTEGER → Table-Res
Lire-Modification : Table-Res x INTEGER → BOOL
Test-Place-Vide : Table-Res → BOOL

Modificateurs

• Modification-Vrai(t,i) = t'

t,t' : Table-Res
i : INTEGER

Précondition : Existe-Num-Place(t, i)

Postcondition : modification((t'[i])) = TRUE

• Nom-Modification-Vrai(t,i) = t'

t,t' : Table-Res
i, j : INTEGER

Précondition : Existe-Num-Place(t, i)

Postcondition : $\forall j : 1 \leq j \leq Card(t) : nom(t[j]) = nom(t[i]) \Rightarrow$ Modification-Vrai(t,j) = t'

Observateurs

• Lire-Modification(t,i) = b

t : Table-Res
i : INTEGER
b : BOOL

Précondition : Existe-Num-Place(t, i)

Postcondition : modification(t[i]) = b

• Test-Place-Vide(t)

t : Table-Res

Précondition : /

Postcondition : (Card(t) < 100)

Annexe 5 :
Les techniques de transformation de schémas

Cette annexe est consacrée à une brève présentation des techniques de transformation de schéma E/A. Nous y aborderons uniquement les aspects nécessaires à la compréhension des transformations appliquées au schéma E/A d'un environnement de bureau étendu. Pour obtenir des détails supplémentaires sur les principes d'utilisation de ces techniques de transformation, nous renvoyons le lecteur aux deux ouvrages suivants [HAIN-89], [COJO-88] qui les abordent en détails.

1. Définitions

Les structures qui nous intéressent sont les structures de généralisation/spécialisation fournies par les relations is-a (voir figure A.5.1). Lorsqu'un type d'entité A est associé par une relation is-a à un type d'entité B, on dit que A joue le **rôle générique** et B le **rôle spécifique**. Par ce principe, le type d'entité B hérite de toutes les caractéristiques propres au type d'entité A. Le type d'entité B est alors décrit par l'ensemble des attributs et types d'associations auxquels les types d'entités A et B participent. Dans l'exemple présenté, le type d'entité B possède pour attributs id-A, att-A et att-B.

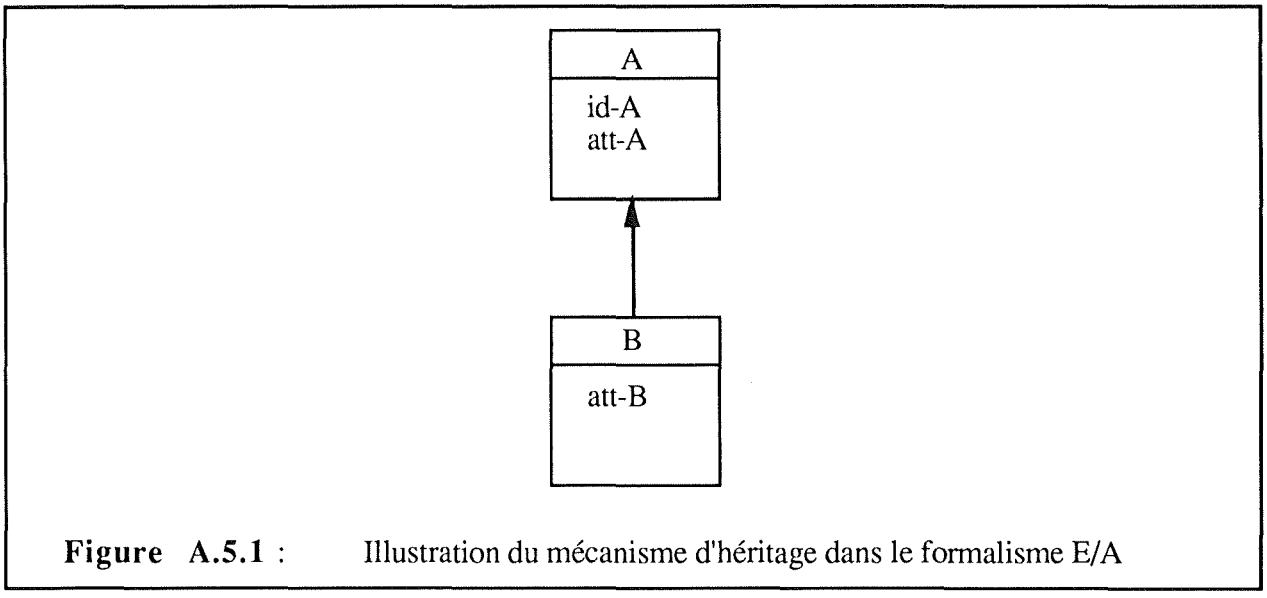

Figure A.5.1 : Illustration du mécanisme d'héritage dans le formalisme E/A

Supposons un type d'entité A qui possède pour types spécifiques les types d'entités B et C alors, s'il n'y a pas d'autres entités de type A que des entités qui appartiennent aussi aux types B ou C, le type d'entité A est dit **couvert** par les types d'entités B et C. De même, si une entité de type A est soit de type B, soit de type C mais jamais des deux types simultanément, il y a **disjonction** entre les types d'entité B et C. La couverture et la disjonction combinées donnent naissance à une structure de **partition** pour les types d'entités B et C. C'est le type de structure qui existe dans le schéma E/A d'un environnement de bureau étendu.

Un **schéma E/A de base** est un schéma E/A dont toutes les structures de généralisation/spécialisation ont été éliminées.

Les techniques de transformation proposées ici ont pour but de mettre un schéma sous cette forme. Ces techniques sont **réversibles,** c'est-à-dire qu'elles "transforment les données sans introduire ni perte ni bruit dans la base de données" [HAIN-89].

2. Présentation des techniques de transformation

2.1. La représentation des types d'entités génériques (R.T.E.G.)

Cette technique consiste à ne représenter que le type d'entité générique auquel on a ajouté les attributs de ses types d'entités spécifiques et les rôles des types d'associations dans lesquels sont impliqués ses types spécifiques (voir figure A.5.2). Cette transformation respecte le principe de représentation unique des objets du réel perçu. Les attributs en provenance des types d'entités B et C deviennent facultatifs. Pour gérer ces valeurs facultatives, un **méta-attribut** ayant pour domaine de valeurs {B, C} permet de déterminer le type spécifique de chaque entité.

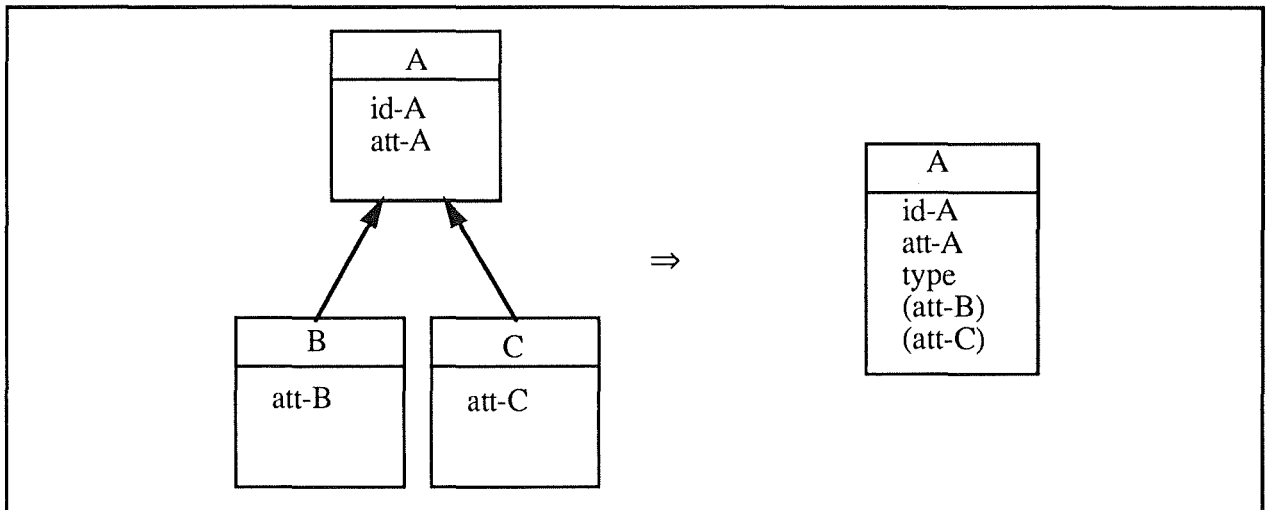

Figure A.5.2 : Effets de la technique de transformation de schéma par R.T.E.G

Dans le cas de structures de partition, à savoir celles qui nous intéressent, le méta-attribut est obligatoire et simple (monovalué). La valeur de type détermine l'existence possible de chaque

attribut et chaque rôle hérité. Pour ce faire, il faut imposer des **contraintes de structure** telles que :

> type = "A" implique que att-A existe et att-B n'existe pas et type = "B" implique que att-B existe et att-A n'existe pas.

De même si des types d'associations étaient initialement définis soit sur B, soit sur C, de nouvelles contraintes apparaissent. Nous n'aborderons pas ce sujet plus en détails car nous désirons nous limiter à la présentation des principes fondamentaux.

2.2. La représentation des types d'entités spécifiques (R.T.E.S.)

La technique de représentation des types d'entités spécifiques consiste à ne représenter que les types d'entités spécifiques après leur avoir attribué par héritage descendant les attributs propres au type d'entité générique et les rôles des types d'associations auxquels le type d'entité générique participe (voir figure A.5.3). Le résultat finalement obtenu respecte comme dans le cas précédent, le principe de représentation unique des objets du réel perçu. Dans ce cas, il n'y a pas de méta-attribut ni de contraintes de structure mais des contraintes sur les types d'associations initialement attribués au type d'entité A peuvent apparaître.

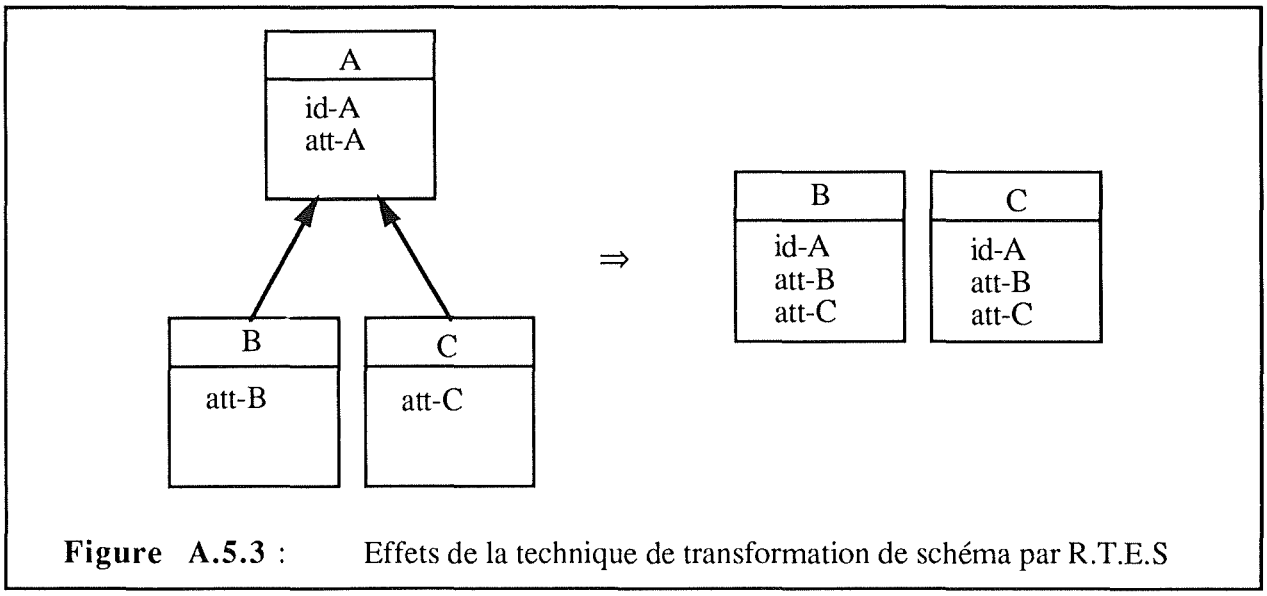

Figure A.5.3 : Effets de la technique de transformation de schéma par R.T.E.S

2.3. La matérialisation des relations is-a (M.R.is-a)

Cette troisième et dernière technique consiste à représenter explicitement les types d'entités génériques et les types d'entités spécifiques et à les mettre en relation par un type d'association signifiant qu'ils représentent un même concept du réel perçu (voir figure A.5.4). Ce type

d'association s'appelle is-a. Les rôles spécifiques du type d'association peuvent être représentés par un **rôle multidomaine** qui signifie que le rôle est soit joué par B soit joué par C.

Le principal défaut de cette technique de transformation est qu'un même concept est représenté dans la base de données par plusieurs types d'entités différents.

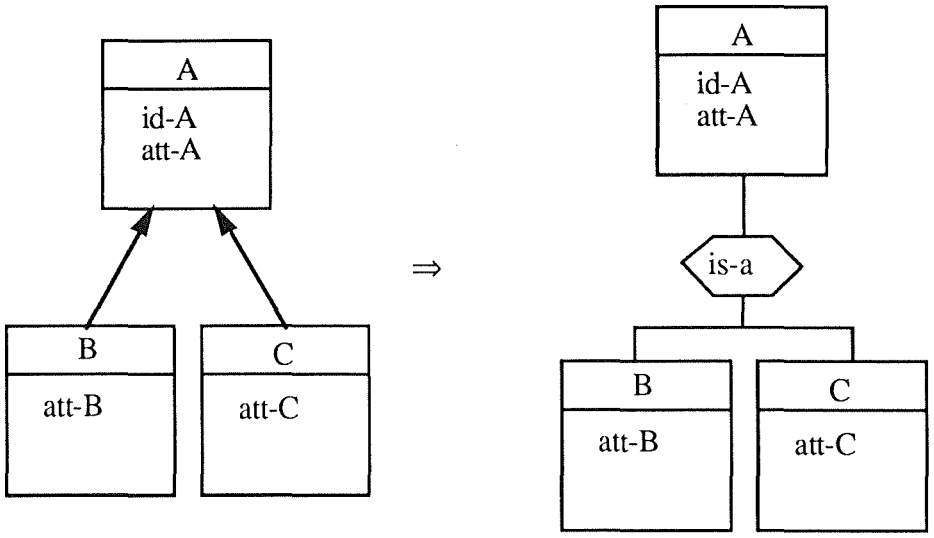

Figure A.5.4 : Effets de la technique de transformation de schéma par M.R.is-a