Université de Nice - Sophia Antipolis
UFR Sciences - Département d'Informatique
Centre National de la Recherche Scientifique
Laboratoire I3S

# Thèse de doctorat

**Spécialité: Informatique**

## Conception et réalisation d'un modèle de persistance pour le langage Eiffel

Philippe LAHIRE

Thèse présentée le Lundi 11 Mai 1992

**Composition du jury:**

| | |
|---|---|
| J-C. Boussard | Président |
| J. Bezivin | Rapporteur |
| N. Le Thanh | Examinateur |
| B. Meyer | Examinateur |
| S. Miranda | Directeur |
| P. Richard | Rapporteur |
| R. Rousseau | Examinateur |

Je remercie Jean-Claude Boussard pour ses conseils amicaux et pour avoir accepté d'être président de mon jury.

Je suis très reconnaissant à Jean Bezivin et P. Richard d'avoir accepté d'être mes rapporteurs et de m'avoir fourni leurs commentaires précis et constructifs.

S. Miranda, m'a fait confiance en acceptant d'être mon directeur de thèse; il a su durant ces trois années passées dans son équipe, m'offrir un certain nombre d'aventures passionnantes: qu'il trouve ici ma profonde gratitude.

En acceptant que je travaille à ses côtés en Californie, Bertrand Meyer m'a permis de vivre des expériences extrêmement enrichissantes; c'est sans nul doute lui qui m'a donné envie de faire de la recherche. Aujourd'hui, il me fait l'honneur de participer à mon jury: c'est une chose que je n'oublierai pas.

Un très grand merci à Roger Rousseau, qui m'a aidé par ses conseils, ses relectures attentives et son amitié à structurer mes idées et à administrer le projet *Business Class*, et à Nhanh Le Thanh pour son aide amicale dans les moments difficiles et ses conseils toujours éclairés dans des domaines très variés.

Je dois aussi beaucoup à toute l'équipe du projet *Business Class*; le détail de leur contribution est développé à la fin de ce mémoire. Je remercie en particulier Jacques Farré, pour sa gentillesse, son esprit d'équipe, sa générosité et ses nuits de travail à l'I3S, et Robert Chignoli pour son active participation et les relectures des différentes versions de ma thèse. Je tiens aussi à remercier Philippe Brissi, Jean-Marc Jugant et Violaine Séré de Rivières pour leur collaboration; je leur souhaite mes meilleurs voeux de réussite dans leur vie professionnelle.

Je voudrais aussi remercier tout le personnel de la Société des Outils du Logiciel pour leur aide, et en particulier son directeur Jean-Marc Nerson, Jean-Pierre Sarkis pour sa collaboration amicale, Philippe Hucklesby pour son amitié, ses commentaires judicieux sur différents articles, et Delphine Boullé pour son amitié.

Je remercie aussi les membres de l'équipe Base de données et du laboratoire I3S pour leur accueil et leur sympathie; je remercie en particulier Evelyne Vittori, Carine Fédèle, Annie Cavarero et Fatene Labene pour leur amitié sincère, Jean-Paul Partouche pour ses conseils techniques et son éternelle bonne humeur, Olivier Lecarme et Michel Rueher pour leurs conseils précieux, Danièle Gérin, Micheline Hagnéré, Cathy Foucard, Lionel Ballergeau et Guy Tessier pour leur soutien logistique efficace et sympathique.

Un merci particulier à Martine Collard, ma compagne de bureau, pour sa gentillesse et sa patience; je lui souhaite bonne chance pour sa thèse.

Je voudrais enfin remercier mes parents pour leur confiance, leur amour et leur soutien en toutes circonstances, ainsi que Monsieur Alain Ramaioli pour son amitié et tout ce qu'il m'a apporté depuis des années.

# 1

# Introduction

## 1.1. Domaine et sujet de la thèse

La notion de donnée a beaucoup évolué depuis l'origine de l'informatique. Simple valeur passive, le plus souvent volatile[1] et sans structure au début des années 60, les données se sont progressivment enrichies de propriétés et de possibilités de manipulation.

La complexification des structures a nécessité des langages pour les décrire; c'est donc principalement dans le domaine des langages informatiques, du génie logiciel, puis plus tard de la C.A.O, que l'on s'est intéresé aux "données-composants", riches en syntaxe et en sémantique, manipulables individuellement et objet d'étude en soi.

Parallèlement à cette évolution, le domaine des bases de données s'est d'abord intéressé à rendre les données persistantes[2]. L'allongement des références temporelles a rendu plus problématique l'évolution et la mise à jour de ces données (notamment lorsque celles-ci sont liées par des relations), les contrôles d'accès, les reprises en cas de panne, l'optimisation des échanges avec les disques. C'est aussi dans ce domaine que l'on manipule de grandes quantités de données, mais très semblables les unes aux autres et de structure simple: on agit davantage sur des collections de données obtenues par sélection, que de manière individuelle.

A partir des années 80, l'incroyable progression de la puissance des ordinateurs a rendu réaliste les techniques de l'intelligence artificielle, notamment la programmation par règle logique et le pradigme "programme = donnée", amplifié par l'approche à objets.

---

1 terme opposé à "persistant" (synonymes: éphémère, "en mémoire vive", temporaire)

2 le terme "persistant" est celui consacré dans la communauté de l'approche à objets, pour qualifier des instances dont la durée de vie dépasse la durée d'activation du programme qui les a créés (synonymes: rémanent, "sur disque")

Les données sont devenues plus actives, capables de réagir à des messages, de vérifier des règles de cohérence et de multiples contraintes, de se déplacer sur un réseau, de factoriser leurs propriétés communes. Cette évolution a eu une double conséquences sur les domaines du génie logiciel et des bases de données.

Le **génie logiciel** s'intéresse maintenant à des composants réutilisables et évolutifs, sur de longues périodes de temps; il a besoin d'un modèle de persistance qui lui permette de stocker ces composants, tout en autorisant leur diffusion sur un réseau, de maîtriser leur évolution, d'établir des liens sémantiques variés (héritage, ...), et de pouvoir les sélectionner sur différents critères: **l'approche à objets persistants est le cadre qui paraît tout indiqué.**

Quant au domaine des **bases de données**, il s'intéresse à présent à des données plus riches en structure et sémantique, déductives, multi-média, pour des applications beaucoup plus diversifiées que les seules données de gestion d'autrefois. Il s'ensuit des besoins nouveaux en matière de langage informatique pour décrire les structures, des traitements plus complexes, des accès navigationnels et interactifs. **Là encore, l'approche par objets paraît la plus prometteuse**.

Le rapprochement des problématiques des domaines des langages de programmation, du génie logiciel et des bases données s'accélère donc. L'industrie du logiciel a un besoin crucial de ce type de rapprochement pour développer des applications qui intègrent de nombreux aspects et techniques: interactivité, persistance, logique, programmation par composants, C.A.O, etc. L'urgence de ce besoin est attestée par des axes de recherche qui sont jugés prioritaires par le milieu industriel, notamment les projets européens *ESPRIT* ou *EUREKA*.

**Cette thèse se situe à l'intersection de ces trois domaines, en concevant et réalisant un modèle de persistance pour le langage Eiffel**, dans le cadre du projet ESPRIT II *"Business Class"* dont l'objectif est de développer un environnement de production de composants logiciels pour les applications de gestion. Eiffel [Meyer 91] est un langage de programmation à objets qui est largement utilisé dans le monde industriel et est actuellement l'un des plus avancés pour le génie logiciel. Le modèle de persistance qui est proposé dans ce travail intègre le savoir faire des bases de données et son implémentation repose sur l'un des principaux SGBD à objets d'aujourd'hui, le système $O_2$.

## 1.2. Cadre de ce travail

Pour mener à bien ce travail, il m'a d'abord fallu acquérir des compétences dans les domaines pré-cités et m'informer de l'état de l'art par une recherche bibliographique approfondie. Mes compétences en génie logiciel, après mes études en MST et DESS ISI, ont été renforcées par un stage d'un an en Californie, dans la société *Interactive Software Engineering* où j'ai développé un prototype graphique d'aide à la conception orientée objets, en langage Eiffel.

J'ai ensuite participé au démarrage du projet ESPRIT II n°2109, "*TOOTSI*", dans l'équipe base de données du laboratoire I3S. Ce projet de 3 ans a été lancé en févier en 1989, avec la participation des partenaires internationaux suivants: Télésystèmes, Ifatec, I3S, Saritel, Olivetti, Infotap. *TOOTSI* réalise un ensemble d'outils logiciels qui permet le développement d'applications qui utilisent des informations obtenus par accès direct à des serveurs distants. Le rôle de notre équipe était de concevoir et de réaliser un modèle pour un environnement de programmation à objets persistants, développé en langage C++. *TOOTSI* m'a permis d'apprendre à gérer un projet d'équipe dans un cadre européen et de développer mes connaissances en base de données.

Le laboratoire I3S fut ensuite contacté, en novembre 1989, pour une autre participation à un projet ESPRIT II n°5311 "*Business Class*", cité précédemment. Ce projet, d'une durée également de 3 ans, a démarré en février 1991. Les sociétés et laboratoires suivant y participent: Télésytèmes, SOL, Etnoteam, Eria, ALR, Datamont, CRISS et I3S. Les thèmes confiés à notre laboratoire portent sur l'intégration de la persistance dans le langage Eiffel et sur la gestion de versions. Mes activités antérieures m'ont naturellement prédisposé à participer au premier de ces thèmes, dont cette thèse constitue la partie principale et pour lequel j'ai exercé les responsabilités de chef de projet.

Dans TOOTSI, du fait de l'absence de relations privilégiées avec les auteurs et les implémenteurs de C++, l'intégration de la persistance dans le langage (NICE-C++) [Mopolo-Moke 91], a été réalisé avec les principaux traits suivants:

- le développement d'un serveur d'objets spécifique, qui réalise une partie du modèle du langage,
- l'utilisation d'ajouts syntaxiques dans un surlangage,

• un accès aux objets explicitement demandé par l'application,
• des restrictions sur les types de C++ qui ont accès aux services de la persistance,
• un accès associatif aux objets par sélection sur des collections, pour des critères simples.

Dans le cahier des charges de *Business Class*, les contraintes et les principes généraux établis avec nos partenaires sont les suivants:

• **1) variété des cas d'utilisation de la persistance**:

- composants logiciels à structure et sémantique forte, mais en nombre relativement restreint: composants Eiffel, éléments de base bibliographique, etc
- bases de données volumineuses en nombre et/ou taille d'objets, mais avec cette fois une complexité limitée,
- données très individualisées, comme dans les applications de bureautique et de CAO, ou au contraire avec donc de nombreuses propriétés factorisables dans des classes.

Il faut permetttre aussi bien les accès navigationnels qu'associatifs;

• **2) réutilisabilité des composants Eiffel** qui utilisent la persistance, en permettant à n'importe quel type d'être rendu persistant et en laissant le moins possible de traces écrites dans les classes qui manipulent des instances persistantes;

• **3) fiabilité** au sens correction, robustesse et sécurité, par:

- le respect de la philosophie du langage, notamment de son système de type, de ses assertions et exceptions,
- la propagation automatique du caractère persistant aux objets atteints transitivement par un objet persistant,
- la satisfaction des contraintes usuelles d'intégrité et de cohérence,
- la sérialisation des accès, rendant impossible les accès simultanés incompatibles,
- le contrôle des droits des personnes à réaliser certains types de manipulation;

• **4) efficacité**:

- minimiser les échanges entre les mémoires volatile et persistante,
- accélérer les accès aux caractéristiques des objets les plus fréquemment manipulées,
- affecter le moins possible le traitement des objets volatiles,
- éviter les effets secondaire sur le temps d'exécution des programmes par l'emploi de structures supplémentaire trop volumineuses;

• **5) Contraintes de réalisation** par le respect des délais imposés dans le calendrier du projet *Business Class*, et l'effort en ressources humaines limité à une vingtaine d'homme mois; ces deux contraintes nous impose à nous limiter à la conception et la réalisation d'un prototype exploratoire;

• **6) Exploiter** nos relations privilégiés avec l'auteur et les implémenteur d'Eiffel pour modifier l'exécutif d'Eiffel, voire de proposer des modifications au langage.

## 1.3. Principaux problèmes et plan de la thèse

Ce mémoire de thèse pouvait être développé suivant deux organisations possibles:

• problème par problème, et pour chacun d'eux explorer la bibliographie sur le sujet, les diverses conceptions possibles, choisir une ou plusieurs solutions et décrire leur implémentation;

• ou bien traiter, pour l'ensemble des problèmes, différents niveaux d'élaboration: pré-étude, modélisation, conception générale, spécifications et implémentation.

Nous avons choisi cette deuxième solution pour traiter globalement les interactions qu'il y a entre les différents problèmes, mais avec le risque de nous répéter lorsque nous abordons chaque problème.

Pour répondre au premier point du cahier des charges sur la variété des applications, nous avons procédé en plusieurs étapes:

• un rappel des points fort et faibles des SGBD et des langages actuels (§ 2.1),

• une étude de l'existant, afin d'évaluer les facilités offertes par la version actuelle d'Eiffel en matière de persistance (§ 2.3),

• la proposition de plusieurs niveaux d'intégration [Cardelli 85], pour répondre à la diversité des applications (§ 2.2)

• une modélisation de la persistance (§ 3.3), avec l'étude des solutions existantes (§ 4.1 & 4.2), et la conception en Eiffel (§ 6.1-6.5) des différentes formes d'accès aux objets persistants (accès navigationnel et associatif).

Pour répondre au second point sur la réutilisabilité, nous nous sommes appuyé sur les règles d'Atkinson qui permettent de garantir la transparence et l'orthogonalité de la persistance par rapport au système des types. La modélisation a été traitée dans les paragraphes 3.1 et 3.2, la conception et l'implémentation dans le chapitre 10.

Pour répondre au troisième point sur la fiabilité, nous avons introduit:

• le concept de Pcollection (modélisation: § 3.4, conception: § 6.6 - 6.7),

• le concept de transaction (§ 7.1), qui permet dans un univers parallèle, non seulement de réagir en cas de panne, mais aussi de sérialiser l'accès aux données persistantes,

• un mécanisme de protection (§ 7.2), qui débouche sur un besoin de partitionnement du monde persistant (§ 3.6, 6.8), lui-même introduit pour des raisons d'unification dans le modèle.

L'efficacité demandée au quatrième point est dans ce travail l'un de nos principaux soucis. Nous montrons pour cela la nécessité d'avoir des traitements en mémoire persistante, c'est à dire gérés par le gestionnaire d'objets persistants.

Ces traitements en mémoire persistante impliquent la propagation des types dans le

gestionnaire; nous avons étudié une propagation dynamique, puis statique (§ 9.1, 9.2). Les solutions d'implémentations sont développées dans le chapitre 12.

Par ailleurs nous avons cherché à améliorer les temps accès pour les deux types d'utilisation:

- lors de l'accès associatif, nous proposons un mécanisme (chapitre 11), pour l'approche dynamique;
- lors de l'accès navigationnel, nous proposons des optimisations dans la gestion des échanges incrémentaux entre les mémoires persistante et volatile (étude: § 8.3, implémentation: 10.2, 10.5, 10.9, 10.10).

Compte tenu des contraintes du projet mentionnées au cinquième point, nous avons réalisé un prototype basé sur le SGBD à objets O2 (chapitres 12 et 13), bien que nous préconisons l'utilisation d'un serveur d'objets spécifique à Eiffel (§ 8.1).

Pour clore ce travail, nous ferons d'abord un bilan critique, avec l'état d'avancement actuel et les contributions, puis nous donnerons des perspectives de recherche en matière de persistance (chapitre 14).

# Partie I

# Analyse du Problème

Dans cette partie, nous faisons le point sur les problèmes soulevés par l'existence de deux communautés d'outils pour modéliser, par une approche à objets,, les problèmes du monde réel: ceux des bases de données et ceux des langages de programmation.

Ensuite nous mettons en évidence l'existence de deux axes de recherches différents pour unifier les langages de programmation et les bases de données, en nous intéressant plus particulièrement aux langages persistants et en expliquant les règles de base d'une bonne intégration.

Enfin, nous présentons l'implémentation actuelle de la persistance en Eiffel 3.1, puis nous l'évaluerons; c'est à partir de cette étude que nous développerons notre approche dans le quatrième chapitre où nous proposons un modèle pour l'intégration de la persistance en Eiffel.

# 2

# Pré-études

## 2.1. Un fossé entre bases de données et langages

Le besoin de persistance d'informations est, avec le besoin de calcul numérique automatisé, à la base de l'émergence et du développement continue de l'informatique: les applications requièrent la mémorisation (la persistance) de données qu'elles traitent pendant leurs phases d'activation. De plus, nombreuses sont les situations qui nécessitent que ces données soient partagées et soient disponibles pour plusieurs applications qui se déroulent simultanément.

Pendant longtemps (jusqu'à la fin des années 1970), le développeur, pour modéliser et résoudre les problèmes du monde réel, a dû utiliser à la fois un langage de programmation et un système de gestion de base de données; le premier permettait la modélisation des structures et des traitements, tandis que le second offrait des fonctionnalités pour la manipulation des données persistantes. La faiblesse des outils de communication entre eux (interfaces permettant à des langages de programmation d'accéder aux données manipulées par les SGBD), pénalisaient le développeur:

- besoin de connaître deux modèles,
- problème de compatibilité entre les modèles (langage et SGBD),
- conversion des données d'un modèle à l'autre mélangée au code du programme,
- problème de conception: différentes représentations pour les mêmes données.

Avec l'apparition de l'approche orientée objet, et l'accroissement des besoins des applications, on constate une évolution. L'approche orientée objet permet de regrouper les structures et les méthodes qui les manipulent, d'augmenter la réutilisation et les capacités d'évolutions des composants; elle laisse ainsi entrevoir des possibilités nouvelles pour la gestion et la manipulation des objets persistants.

L'idée de créer des systèmes regroupant à la fois les fonctionnalités des langages et celles des systèmes de gestion de base de données, a fait son chemin, en suivant deux directions:

• introduire la possibilité de décrire des traitements dans les systèmes de gestion de base de données: SGBD orienté objets,

• introduire une gestion efficace des objets persistants dans les langages de programmation: langage de programmation intégrant la persistance

Nous allons, avant de développer ces deux axes de recherche revenir un peu sur les motivations qui ont poussé les langages de programmation et de base de données, classiques à évoluer.

### 2.1.1. Persistance des données pour les bases de données classiques

L'implémentation du besoin de persistance mentionné ci-dessus, a suivi l'évolution de la technologie et des concepts de l'informatique: fichiers et systèmes de fichiers arborescents, systèmes de bases de données hiérarchiques, systèmes de bases de données relationnelles, et aujourd'hui les SGBD à objets. Notre propos dans ce paragraphe est de montrer les apports et les lacunes des SGBD relationnels, et de présenter ainsi les motivations des développeurs de SGBD à se tourner vers les technologies à objets.

Le modèle relationnel [Codd 70] est très bien adapté aux applications classiques de traitement de l'information tel que la comptabilité, le contrôle d'inventaire, etc. Il a l'avantage d'être à la fois simple, et d'offrir des concepts puissants comme la manipulation ensembliste des données.

Cependant, le fait qu'il ne permette que des structures plates (pas de lien pour exprimer une hiérarchie), qu'il n'offre pas la possibilité de gérer des propriétés multivaluées (ex: type *collection*), qu'il ne fournisse pas d'outils pour modéliser des données riches en sémantique et ayant une structure complexe [Valduriez 87], le rendent inadapté pour le traitement d'applications avancées touchant à des domaines comme l'intelligence artificielle, la bureautique, la CAO, ou le génie logiciel.

Si l'on considère la construction d'environnements de programmation [Berstein 87, Godart 91] offrant des outils CASE, qui correspondent au domaine choisi pour le langage Eiffel, les carences du modèle relationnel peuvent se résumer par la difficulté qu'il a, à

répondre aux besoins suivants:

- stockage de versions multiples d'objets,
- stockage de gros objets pouvant avoir une taille variable,
- stockage d'un grand nombre de type,
- manipulation décentralisée du graphe des types dans une base,
- gestion de multiples représentations des données,
- intégration d'opérateurs flexibles et puissants pour la manipulation de graphes d'objets (navigation à travers les structures),
- support de types de données flexibles construits par l'utilisateur avec leurs méthodes associées,
- gestion de transactions longues.

Pour prendre en compte ces aspects, deux directions ont été suivies:

- l'extension des SGBD relationnels,
- la construction de SGBD orientés objets.

Nous rappellerons dans le paragraphe 2.2.1 les principales contributions dans ces deux domaines.

### 2.1.2. Description des traitements pour les langages de programmation

Les langages de programmation privilégient les aspects suivants:

- représentation des objets,
- description des traitements.

Cependant, ils prennent très peu en compte les aspects concernant la persistance comme:

- le stockage, la récupération et la modification des objets,
- le partage des objets par plusieurs applications,
- l'utilisation massive d'objets persistants,
- l'intégrité et la protection des objets.

En fait, la persistance est plutôt traitée sous l'angle de la sauvegarde [Atkinson 83], plutôt que sous celui d'une véritable gestion de la persistance apportant des solutions

pour traiter les points mentionnés ci-dessus. Typiquement, les langages offrent trois solutions pour la sauvegarde et la récupération d'objets:

• l'utilisation directe du **système de gestion de fichiers** (ex: Fichiers Unix à partir de C++ ou d'Eiffel), qui est peu adaptée pour stocker et mettre à jour des objets de taille différente (c'est typiquement un besoin des langages à objets); il faut alors fournir une couche supplémentaire pour gérer les conversions de représentation;

• l'utilisation d'**interfaces** vers des systèmes de gestion de base de données relationnelles (Oracle, Ingres, etc), ou objets (Ontos, $O_2$, etc). Ces interfaces permettent plus ou moins d'adresser les problèmes liés à la gestion de la persistance (partage des objets, maintien de l'intégrité et de la cohérence des objets), mais engendre des problèmes d'efficacité et ne sont pas adaptées pour résoudre les problèmes d'évolution et de réutilisation de composants:

- nécessité de gérer l'incompatibilité entre les deux modèles (langage et SGBD); elle sera d'autant plus difficile à gérer que la distance entre les deux modèles sera grande; en général, sauf au prix d'une solution très compliquée, certains types d'objets ne pourront devenir persistants,

- Manque d'efficacité, dû en particulier aux problèmes de conversion de modèle à modèle,

- Perte de lisibilité à cause notamment de la présence de plusieurs modèles, des mécanismes de conversion et de l'appel explicite des opérations d'entrée-sortie;

• la réalisation d'une couche plus ou moins évoluée permettant le stockage d'objets ayant des types différents (graphe d'objets). Cette couche existe notamment en Eiffel [Meyer 90], et en Smalltalk [Goldberg 89].

Les langages de programation prennent donc souvent mal en compte l'aspect persistant des objets manipulés; une étude réalisée dans [Cockshott 87] montre en particulier que la gestion explicite des entrée-sorties (environ 30% du code), entraîne une diminution de la lisibilité, de la robustesse, et de la fiabilité des applications, et provoque donc une perte importante en matière de productivité pour les entreprises.

## 2.2. Vers la construction de systèmes persistants

Deux types de langage sont en train de s'opposer, il s'agit des langages de

programmation de base de données, construits pour les SGBD à objets, et les langages persistants. Il est difficile de mettre une frontière nette entre ces deux types de langage, mais bien qu'il n'y ait pas de consensus à ce sujet, deux différences semblent se dégager:

• la première, qui semble la plus fondamentale, concerne leur philosophie [Maier 87]: les langages de base de données accèdent plutôt aux objets de manière associative (requête sur un ensemble d'objets), tandis que les langages de programmation persistants, privilégient l'accès navigationnel (navigation à travers un graphe d'objets), et n'encapsulent pas les itérations [Atkinson 83, Kaehler 90].

• la seconde concerne l'exécution des routines; souvent les gestionnaires d'objets qui gérent les entités persistantes manipulés par les applications implémentées par des langages de programmation, n'offrent pas de fonctionnalités permettant l'exécution des routines en mémoire persistante (c'est à dire sous contrôle du gestionnaire). Cela a pour effet de provoquer un important va et vient d'objets entre le gestionnaire et les applications clientes, entre les mémoires persistantes et volatiles.

### 2.2.1. Langage de programmation de Base de données à objets

Citons quelques articles présentant les principaux concepts que l'on doit retrouver dans des SGBD orientés objets. Dans [Maier 90] on trouve les fondements des SGBD-OO, alors que dans [Atkinson 89 c] il y a une synthèse sur les principaux points à considérer lors de la construction d'un SGBD-OO. L'évolution nécessaire des SGBD est évoqué à travers l'apparition des nouvelles applications et leurs besoins dans [Silberschatz 91].

#### Extensions de SGBD relationnels

Notre propos n'est pas d'étudier les extensions du modèle relationnel; on citera cependant quelques références concernant les principales contributions: Postgres [Rowe 87, Stonebraker 91], Starbust [Lohman 91], Exodus [Carey 90], Genesis [Batory 88].

## SGBD orientés objets

Plusieurs comparaisons récentes sur les SGBD orientés objets existants, donnent une vue synthétique des différentes contributions [Gardarin 88, Kim 91, Horowitz 91, Godart 91, Bancilhon 91 a&b], nous conseillons au lecteur de les consulter et nous présentons seulement les références et les objectifs principaux des principaux SGBD; ceux-ci se classent en plusieurs catégories:

- produits commercialisés: Gemstone, $O_2$, Vbase, Statice,

- prototypes de recherche en milieu industriel: Orion, Iris,

- prototypes de recherche en milieu universitaire: Encore, Postgres, Avance, OZ+, Extra, etc.

*Opal*, le langage de définition et de manipulation de *Gemstone* est une extension de *Smalltalk* aux bases de données. Il ne supporte que l'héritage simple et l'effacement automatique des objets est géré par un ramasse-miettes. il fournit un certains nombre d'interfaces vers les langages Smalltak, C et C++. Du point de vue des services qu'il offre pour la persistance, on trouve:

- la gestion des accès simultanés par plusieurs utilisateurs, à un même objet [Penney 87a],
- la présence d'opérations associatives acceptant en paramètre un sous-ensemble des expressions du langage [Horowitz 89, Goldberg 89],
- un gestionnaire d'objets basé sur une architecture client/serveur [Maier 86a, bretl 89],
- le traitement de l'évolution dynamique du schéma (graphe de classe) [Penney 87 b],
- un mécanisme transactionnel basé sur une gestion optimiste et pessimiste de l'accès concurrent [Servio 89, Bretl 89],
- un mécanisme à base d'index, pour l'optimisation des requêtes [Maier 86b].

Le système *$O_2$* [Lecluse 88, Bancilhon 88, Deux 90 & 91] commence seulement à être commercialisé par $O_2$-Technology; il intègre un modèle de données permettant la manipulation d'objets, de valeurs (objets sans identificateur), et de collection d'objets ou de valeurs (liste, ensemble).

Il offre comme Gemstone plusieurs interfaces (C, C++), mais fournit de plus un vrai

langage de requête: *O2-Query*. Les objets sont décrits par des classes qui comprennent la définition des structures et des méthodes permettant de les manipuler (décrites avec le langage O2-C qui est une extension de C).

*O2* est un système qui a privilégié certains aspects scientifiques comme l'intégration de l'approche objet au niveau du modèle et des langages de définition et de manipulation des données [Lecluse 88] ou comme les regroupements d'objets [Benzaken 89]; par contre, les mécanismes de transaction, et de protection sont classiques; la gestion sur le disque des objets persistants est assuré par le gestionnaire de disque WiSS (Wisconsin Storage System) [Chou 86]. Enfin, *O2* offre un mécanisme d'indexation prenant en compte l'utilisation d'expressions pointées.

*Orion* [Kim 89 b & 91] est plutôt orienté vers des domaines comme l'intelligence artificielle, ou la conception assistée par ordinateur; son langage d'implémentation (Common Lisp), n'est pas étranger à cette orientation. Les besoins en matière d'architecture et certains détails sur celle d'Orion lui-même, sont traités dans [Kim 90 a&b]. Ce prototype a permis entre autre la validation d'un certain nombre d'approches; elles concernent:

- la notion d'objet composite représente la relation *is-a-part-of*, permettant d'enrichir la sémantique d'une référence vers un objet [Kim 87 & 89 b],

- la gestion des versions (modélisation du cycle de vie) [Kim 88],

- la notification et la propagation des modifications dans un schéma (hiérarchie de classe) [Banerjee 87, Chou 88]

- un langage de requêtes [Banerjee 88, Kim 91]

- un mécanisme transactionnel, fournissant un modèle pour l'imbrication des transactions, et les transactions longues [Garza 88, Kim 91].

*Iris* [Fishman 87 & 89] est basé sur le modèle fonctionnel de données de DAPLEX [Shipman 81]; il supporte une forme limitée de règles, les liens inverses, des attributs multivalués (ensembles d'objets) la définition d'opérations par l'utilisateur; il offre aussi un langage de requêtes nommé *OSQL*. *Iris* est construit sur un système de stockage basé sur le modèle relationnel et inclut un nombre notable de fonctionnalités (gestion limitée des versions, optimisation des requêtes, accès conccurent, mécanisme transactionnel,

etc).

*Vbase* [Andrew 87] est très influencé par CLU [Liskov 77], il suit la même approche que *O2* dans le sens où il y a un langage pour spécifier les types (Type Definition Language), et un autre (C Object Processor), représentant un sur-ensemble de C, pour décrire les méthodes qui sont associées à ces types [Andrews 91a, Damon 91]. Dans [Harris 91] l'auteur décrit le langage de requêtes de Vbase, qui dérive du standard SQL. Vbase ne fournit qu'un ensemble relativement restreint de fonctionnalités:

- groupement des objets contrôlés par l'application,
- mécanisme de reprise sur panne,
- support rudimentaire pour l'accès simultané par plusieurs utilisateurs, à un même objet,
- mécanisme d'exception permettant de gérer des contraintes.

*Ontos* [Andrews 91 b] est le successeur de Vbase, et on citera seulement leur principale différence; elle concerne la philosophie de la conception du système: Ontos est par rapport à *Vbase* un outil de plus bas niveau (il suit en cela la philosophie de C++), permettant au développeur d'application de gérer le compromis entre les performances, la sécurité et le formalisme.

D'autres systèmes comme *Avance* [Bjornerstedt 88] et *OZ+* [Weiser 89] ont pour objectif l'aide au développement d'applications liées au domaine de la bureautique. En particulier *Avance*, qui a les mêmes bases de départ que *Opal* (Gemstone), aborde les problèmes de gestion de version [Bjornerstedt 89]. Il en va de même pour Encore [Zdonik 86], qui a été l'un des premiers à aborder le problème des versions [Skarra 86 & 87]; il introduit aussi d'autres fonctionnalités intéressantes [Skarra 89]:

- la gestion de transactions longues,
- un contrôle des accès en parallèle basé sur un mécanisme de verrouillage,
- un mécanisme de reprise sur panne basé sur la mémorisation d'historiques.

### 2.2.2. Langages de programmation intégrant la persistance

Quand on parle de langages persistants, il faut bien distinguer les différents niveaux de persistance [Cardelli 85]:

• le premier niveau est représenté par le modèle **Récupération/Stockage** (Fetch-Store model), qui ne gère aucune dépendance entre l'objet persistant stocké dans un fichier et sa copie en mémoire volatile. Ce niveau est par exemple caractérisé par la classe *STORABLE* ou *ENVIRONMENT* d'Eiffel [Meyer 90]; ce modèle de persistance est en général le modèle minimum offert par les langages à objets comme Loops, Flavors [Masini 89] ou C++;

• Le deuxième niveau correspond à un modèle de **Chargement/Mise à jour** (Load-Dump Model); il gère la dépendance entre un objet persistant et son unique copie dans une application (environnement mono-utilisateur); la mémoire volatile agit alors comme un cache par rapport à la mémoire persistante;

• Le troisième niveau est en fait une extension du précédent dans un environnement où plusieurs applications peuvent accéder aux mêmes données (Lock-Commit model); ce modèle représente en fait les langages persistants au sens d'Atkinson [Atkinson 89 a&b].

Dans la suite on privilégiera cette dernière catégorie, mais nous mettrons aussi en évidence cette gradation de la persistance.

#### La persistance dans les langages compilés

Certaines extensions à la persistance ont été réalisées à partir de langages compilés comme C/C++. Il s'agit essentiellement du *langage E* [Richardson 89 a&b] utilisé dans le projet Exodus [Carey 86 & 88], et de *O++* visant à interfacer le système Ode [Agrawal 89 a&b]. On peut aussi citer une contribution plus modeste comme *Presto* qui permet de supporter le système d'information *Adage* [Giavitto 88]

D'autres comme *PS-Algol* introduisent la persistance dans *S-Algol*, lui-même dérivant d'Algol68. *PS-Algol* est certainement le langage persistant le plus souvent évoqué dans la littérature; le lecteur pourra en trouver un historique dans [Atkinson 89]. Il est à noter qu'un autre langage, *Napier 88*, basé sur l'expérience de *PS-Algol* mais

intégrant notamment un type *relation*, est actuellement en cours de réalisation [Dearle 89].

Un autre langage persistant, *Galileo* [Albano 86 & 87, Atkinson 87], est essentiellement dédié à l'écriture d'applications de bases de données; il dérive d'une des premières versions du *langage ML* [Gordon 79], il intègre les concepts de classe, d'héritage et le polymorphisme d'inclusion.

*Amber* [Cardelli 84, Atkinson 87] est dérivée de Galileo et supporte l'héritage multiple, les concepts de type statique et dynamique autorisant une forme de polymorphisme. On remarque en particulier le type *fonction* et une gestion uniforme de la persistance à travers l'utilisation du type pointeur *Pntr*.

Enfin, parmi les autres langages intégrant des modèles de données avancés, il y a *Taxis* [Mylopoulos 80, Atkinson 87] plutôt orienté vers la gestion des données que vers l'aspect programmation. Il intègre les concepts de classe et méta-classe, d'héritage, de variable de classe. Enfin on pourra aussi citer *Adaplex* [Smith 83, Atkinson 87], qui est une extension d'*Ada* [Ichbiah 79] et *Daplex* [Shipman 1981]. Le but étant d'introduire, tout en gardant la philosophie du langage *Ada* de nouveaux types et de nouvelles structures de contrôle permettant la gestion de données persistantes issues de programmes écrits en Daplex.

Le langage *Pgraphite* [Wileden 88, Tarr 89] est lui aussi une extension du langage *Ad*a. Il met en oeuvre une couche objet auquel est associée une gestion d'objets persistants. Sa principale contribution consiste à fournir un mécanisme de définition de type (extension syntaxique de *Ada*), pour une catégorie d'objets, à savoir les graphes orientés, et à générer l'implémentation correspondante en *Ada*

On pourra citer encore *Trellis/Owl* [O'brien 86 & 88] qui est particulièrement adapté pour gérer des objets complexes de grande taille dans un environnement partagé et dans la perspective de transactions longues [Schaffert 86].

OOPS+ [Laen 88] est un langage orienté objets conçu dans le cadre du projet esprit KIWI. Il est dédié aux applications gérant la connaissance et inclut donc des possibilités de programmation logique.

**La persistance dans les langages interprétés**

Dans l'univers interprété, on trouve essentiellement des langages basés sur *Lisp* et sur *Smalltalk*.

*Pclos* [Paepcke 88 & 89] introduit la persistance dans le langage *Clos* (Common Lisp Object System) [Bobrow 88]. *Zeitgeist* [Ford 88], est implanté à l'heure actuelle sur le langage *Flavors* et bientôt sur *Clos*, et *LispO2* [Barbedette 89].

Le traitement de la persistance dans *Smalltalk* est seulement de niveau 1, sauf peut être pour la gestion des classes qui sont traitées comme des objets persistants. Par contre l'adjonction d'une bibliothèque de composants comme *BOSS* [pps 90], permet de passer au niveau 2. D'autres améliorations, touchant en particulier l'intégrité des objets comme l'introduction d'un mécanisme transactionnel sont traitées dans des extensions de Smalltalk [Low 89, Straw 89].

### 2.2.3. Pré-requis d'une bonne intégration

Quand on cherche à mesurer le niveau d'intégration de la persistance dans un langage de programmation, on doit considérer cette intégration à la fois du point de vue de la modélisation et de l'intégration dans le langage (vision des applications) et de l'implémentation de la gestion de la persistance (efficacité, services offerts), dont la qualité dépend **essentiellement de la compatibilité entre le modèle d'exécution du gestionnaire d'objets et celui du langage**.

Nous montrerons que pour réaliser une intégration qui prend en considération ces trois aspects: modélisation, intégration et implémentation, il faut être maître à la fois de l'exécutif (et éventuellement du compilateur), associé au langage, et du modèle de données servant de support au gestionnaire d'objets.

Si l'on n'a pas le pouvoir de modifier les mécanismes internes du langage, la transparence, la souplesse d'utilisation, l'orthogonalité et les fonctionnalités offertes s'en ressentiront. Par contre, plus le gestionnaire d'objets aura un modèle éloigné de celui du langage de programmation, plus apparaîtrons des problèmes de compatibilité et donc d'efficacité.

La situation idéale serait celle où, le modèle du gestionnaire et du langage coincideraient, et où l'exécutif et le code généré du langage pourraient facilement être adaptés pour gérer l'accès aux objets persistants.

**Notre contribution se situe essentiellement au niveau de la modélisation et de l'intégration de la persistance dans le langage Eiffel**. Notre choix du système *O2* pour implémenter la persistance (cf. § 8.1) nous permet une étude approfondie des problèmes

et la réalisation d'un prototype; mais celui-ci ne remplit pas complètement les conditions d'une intégration parfaite, qui autoriserait une utilisation totalement opérationnelle.

En effet, comme nous ne maîtrisons pas le modèle du gestionnaire d'objets utilisé (en l'occurrence $O_2$ [Bancilhon 91]), nous nous trouvons confronté à des problèmes d'efficacité dû à la distance entre les modèles *Eiffel* et $O_2$; ces inconvénients disparaitraient dans le cas où l'implémentation serait basée sur une gestion d'objets de plus bas niveau, comme celle des serveurs d'objets (Wiss [Chou 86], Kala [Simmel 91] ou NICEMP [Salgado 88]).

Comme cela a déjà été évoqué dans l'introduction, nous avons dans notre laboratoire, expérimenté les deux axes d'intégration: dans [Mopolo-Moke 91], C++ est étendu à la persistance en privilégiant l'efficacité (le serveur NICEMP a été adapté pour C++), alors qu'ici, c'est l'aspect modélisation qui a été privilégié (adaptation du compilateur et de l'exécutif d'Eiffel).

# 2.3. Etude de la persistance en Eiffel 3.1

## 2.3.1. Modèle

Dans la bibliothèque actuelle d'Eiffel, le modèle de la persistance repose sur deux concepts, ceux d'**objet persistant** et celui d'**environnement**. Tout objet inséré dans un *environnemen*t est implicitement rendu persistant. L'action d'insertion peut être explicite ou implicite.

**Objet persistant**

*Un objet est **persistant** si et seulement si il a été inséré dans un **environnement**; éventuellement, il peut, lors de son insertion, être associé à un nom par le programme: on dit alors qu'il est nommé.*

**Environnement**

*Un **environnement** est un ensemble d'objets nommés ou non de type ANY, où seuls les objets nommés sont directement accessibles par les opérateurs d'un environnement. Les autres ne sont accessibles qu'à travers des entités (au sens d'Eiffel).*

**Conséquence:**

Toute classe Eiffel étant implicitement héritière de la classe *ANY*, toute instance peut être rendue persistante, quelque soit son type. On notera donc en particulier que cette notion est parfaitement orthogonale vis à vis de la notion d'instance: un

environnement peut-être lui même inséré dans un environnement.

Les opérateurs d'un environnement permettent seulement des manipulations simples portant sur un objet; on trouve ainsi:

• *put*, qui permet d'insérer un objet racine de persistance en lui associant un nom symbolique. Cette insertion n'est possible que si le nom n'est pas déjà associé à un autre objet de l'environnement; elle provoque l'insertion implicite des objets référencés par la racine,

• *item*: extraction d'un objet racine à partir du nom qui lui a été associé,

• *has*: interrogation sur la présence d'un objet racine,

• *count*: nombre d'objets persistants dans l'environnement, éventuellement en fonction du type,

• *store*: mise à jour, dans la mémoire persistante des objets contenus dans l'environnement,

• *retrieve*: chargement dans la mémoire volatile des objets contenus dans la mémoire persistante associée à l'environnement afin de les rendre accessibles à travers les primitives de l'environnement.

## 2.3.2. Implémentation

**Structures d'intégration du modèle**

La mise en oeuvre repose sur l'implantation d'une classe spécifique; la classe *ENVIRONMENT* qui joue le rôle d'interface avec l'outil persistant. L'étude de cette interface montre en particulier que le choix de l'outil persistant est le système de fichier Unix, que les objets sont considérés dans la mémoire persistante comme étant de type *ANY*, et que les échanges entre la mémoire gérée par le système de fichiers et celle du programme se font sur la totalité des objets contenus dans l'environnement; il n'y a pas d'échange incrémental.

| **class** *ENVIRONMENT* |
|---|

```
creation   create
feature
  create is
        -- Create a new environment.
    ensure    suffix.equal (".env")

  is_open: BOOLEAN;
        -- Is current environment open?

  open is
        -- Close previous environment if any, and make current environment the active
        -- one: All objects created from now on, until next call to open or close, will
        -- belong with their dependents to this environment.
    ensure    open: is_open

  close is
        -- Make current environment be no longer active; if it was.
    ensure    closed: not is_open

  put (obj: ANY; key: STRING) is
        -- Record obj under key in this environment: Make obj and its dependents persistent.
        -- if key is already in use, set error to Conflict.
    require   key_not_void: not key.Void
    ensure    ok or (error = Conflict)

  force (obj: ANY; key: STRING) is
        -- Record obj under key in this environment: Make obj and its dependents persistent.
        -- if key is already in use, lose the previous association.
    require   key_not_void: not key.Void
    ensure    error = No_error

  change_key (old_key, new_key: STRING) is
        -- Record under new_key the object previously recorded under old_key;
        -- if no such object, set error to Not_found.
    require   keys_not_void: not (old_key.Void or new_key.Void)
    ensure    ok or (error = Not_found) or (error = Conflict)

  remove (key: STRING) is
        -- Dissociate key from object recorded under; if no such object, set error to Not_found.
    require   key_not_void: not key.Void
    ensure    ok or (error = Not_found)

  has (key: STRING): BOOLEAN is
        -- Is key in use?
    require   key_not_void: not key.Void

  item (key: STRING): ANY is
        -- Object recorded under key; Void value if no such key, and error set to Not_found
    require   key_not_void: not key.Void
    ensure    Result.Void implies not has (key)

  count (t: STRING): INTEGER is
        -- Number of objects of type t in environment (using "any" for type will yield the
        -- total count, "special" that of special objects)
    require   type_not_void: not t.Void
```

```
  persistent_count (t: STRING): INTEGER is
        -- Number of persistent objects of type t in environment (using "any" for type will
        -- yield the total count, "special" that of special objects)

  set_file (f: FILE) is
        -- Make f the file where environment is stored if requested explicitly or implicitly.
     require   once_set: persist_file = void;  file_created: not f = void;
               good_suffix:  is_env_file (f)
     ensure    not persist_file.Void

  store is
        -- Store the environment's persistent objects to the associated external file.
     require   good_file: not persist_file.Void
     ensure    ok or (error = Not_stored)

  retrieve  is
        -- Load the environment's persistent objects from the associated external file.
     require   good_file: not persist_file.Void and then persist_file.exists
     ensure    ok or (error = Not_retrieved)

  store_on_end is
        -- Enable automatic storage on normal termination. This is not the default.

  store_on_failure is
        -- Enable automatic storage on abnormal termination. This is not the default.

  no_store_on_end is
        -- Disable automatic storage on normal termination. This is the default.

  no_store_on_failure is
        -- Disable automatic storage on abnormal termination. This is the default.

  ok: BOOLEAN is
        -- Did last retrieve, store, put or remove operation succeed?
     ensure    Result = (error = No_error)

  error: INTEGER;
        -- Code of last error produced by a routine of the class (Possible codes are given
next)

  No_error: INTEGER is unique;
  Not_found: INTEGER is unique;
  Conflict: INTEGER is unique;
  Not_stored: INTEGER is unique;
  Not_retrieved: INTEGER is unique;

  current_keys: ARRAY [STRING] is
        -- Array of keys in use, starting from 1

  share (other: like Current) is
        -- Make Current and `other' share the same persistent objects.

  set_suffix (new_suffix: STRING) is
        -- Change file suffix.
```

```
    require   not new_suffix.Void; new_suffix @ 1 = '.'
    ensure    suffix.equal (new_suffix)

  suffix: STRING;
        -- Suffix used for environment files

end -- class ENVIRONMENT
```

.Figure 2.1: Gestion de la persistance en Eiffel 3.1

**Principes d'utilisation**

Ce mécanisme de la persistance s'utilise selon l'un des deux types de lien existant en Eiffel: une classe pourra soit hériter de la classe *ENVIRONMENT*, soit établir un lien de clientèle et disposer d'un attribut de type *ENVIRONMENT*. Le plus souvent c'est cette dernière solution qui sera utilisée.

Exemple: La création et le remplissage d'un environnement (par un lien de clientèle)

```
class CREATE_ENVIRONMENT          -- allow the input of persons in an environment
feature
  base: ENVIRONMENT;
  env_file: FILE;

  create_env is
    do
       !!base.create;                  -- base reference a new occurence
  !!env_file.create("env_file");       -- env_file  reference a new occurence of FILE
       base.set_file(env_file);        -- set the file used for base storage
       base.store_on_end;              -- all objects of base are saved at session termination
       input (10);                     -- input 10 objects and their dependents in base
       base.close;                     -- store all objects in env_file, and close base
    end; -- create_env

  input (nb: INTEGER) is
          -- Input nb persons and their dependents in base generating a unique key
          -- when inserting a new  named object in the environment
    do ... end; -- input

end -- class CREATE_ENVIRONMENT
```

Figure 2.2: Création d'un réservoir d'objets persistants

Un objet persistant est associé au type le plus général du langage (type *ANY*). Il faudra donc mettre en oeuvre un moyen de vérifier la compatibilité des types, chaque fois qu'un objet persistant est associé à un attribut.

Exemple: Utilisation d'un environnement existant

```
class USE_ENVIRONMENT                    -- allow the use of an environment of persons
feature
  base: ENVIRONMENT;
  env_file: FILE;

  use_env is
    local
      p: PERSON
    do
      !!base.create;                  -- base reference a new occurence of ENVIRONMENT
      !!env_file.create("env_file");  -- env_file reference an occurence of FILE
      base.set_file(env_file);        -- set the file used for the storage of base
      base.retrieve;                  -- retrieve all persistent objects from env_file
      p ?= base.item ("Laurie");      -- p should reference a person of key = "Laurie"
      if not p = void then                -- check object exist and is of type "person"
        -- .... use of person referenced by p ....
       end;
      base.close;                     -- store all objects in env_file, and close base
    end; -- use_env
end -- Class USE_ENVIRONMENT
```

Figure 2.3: Utilisation d'un réservoir d'objets persistants

### 2.3.3. Connexion à l'environnement

Comme le montre l'exemple précédent, le programme doit associer un fichier à l'environnement (primitive *set_file*); c'est par l'intermédiaire de ce fichier que l'on réalise la connexion entre l'outil persistant (système de fichier Unix) et l'environnement. Ici, aucune surcouche ne gère la persistance; la classe *ENVIRONMENT* la délègue entièrement au système de fichier.

Seule la conversion entre le format persistant et le format Eiffel est réalisé par les opérateurs de l'environnement gérant l'échange entre les mémoires persistantes et volatiles (*store* et *retrieve*).

Tous les objets d'un environnement sont stockés dans le même fichier, la gestion des protections, le partage des objets persistants, et les échanges entre la mémoire persistante et la mémoire volatile sont gérés par le système de fichiers.

### 2.3.4. Critique de l'existant

#### Critique de la modélisation

La notion d'environnement montre que la persistance est introduite **au niveau des**

**objets et non pas au niveau des types**[1] ; par ailleurs, une fois qu'il est rendu persistant, un objet n'est pas manipulé d'une façon différente de celle utilisée pour les objets volatiles. On notera aussi que la synchronisation entre l'objet persistant et son image en mémoire est réalisée explicitement par le programme Eiffel, ou automatiquement en fin de session.

Un environnement accepte tout type descendant de la classe *ANY* qui est implicitement un parent de toute classe Eiffel, y compris de la classe *ENVIRONMENT*.

Ce modèle de persistance garantit que tout objet, référencé par un objet explicitement inséré dans un environnement E, est lui aussi, **automatiquement inséré** dans E (et donc rendu persistant). D'autre part, tout objet non persistant qui devient référencé par un objet déjà persistant, devient automatiquement persistant lorsque l'environnement est mis à jour. Par ailleurs, l'**intégrité référentielle** des objets est assurée par les primitives *store* et *retrieve* associées au langage.

L'association explicite d'une clé à un objet persistant permet de rendre l'identification des objets explicitement insérés dans un environnement indépendant de leur contenu et donc des propriétés de la classe.

Le fait d'associer une clé seulement aux objets insérés explicitement, **rend impossible l'accès direct aux objets insérés implicitement**; au contraire cela nécessite pour eux une navigation à travers les liens de clientèle, à partir d'un objet rendu explicitement persistant. Ceci est normal, dans le cadre de la sauvegarde du graphe d'objets correspondant à un programme Eiffel, mais **n'est pas du tout adapté à des applications de base de données manipulant un grand nombre d'objets persistants**.

Ce manque d'homogénéité entre le traitement des objets objets rendus explicitement ou implicitement persistants est très pénalisant dans ce dernier cas, **surtout du fait de l'absence de toute facilité pour réaliser des sélections sur les objets d'un environnement**.

La **principale faiblesse du modèle actuel** provient du fait qu'il n'y a aucun traitement en mémoire persistante, c'est à dire aucun traitement réalisé par l'outil persistant. Ainsi toute action (accès à un objet, modification par appel de procédure ou de fonction) prenant en considération le type de l'objet, est réalisée en mémoire volatile. En mémoire

---

[1] Il existe dans l'environnement Eiffel une classe *STORABLE,* qui permet de gérer la persistance à un niveau encore plus élementaire: 1 fichier par objet racine. On pourrait toutefois, en redéfinissant l'opération de création, rendre automatiquement persistant tout objet créé.

persistante, il n'y a pas de notion de type, sauf *ANY* qui est très général et donc inutilisable pour des traitements en mémoire persistante propres à une catégorie d'objet. La synchronisation avec la mémoire persistante est réalisée par les primitives *store* et *retrieve* de l'environnement.

Dans le même sens, le rangement et le chargement des méthodes n'est géré que par reliure statique, avec l'éditeur de liens du système d'exploitation. Il s'ensuit qu'il faut lors du chargement d'un objet, s'être assuré au préalable que les méthodes de sa classe sont bien reliées au programme (en déclarant par exemple un attribut de son type), pour pouvoir les utiliser. Pour les applications qui ignorent le type exact des objets à manipuler (une base de descriptifs d'appareils par exemple), cela oblige à placer dans la reliure statique l'union de toutes les classes pouvant apparaître. Ceci conduit à un code de taille excessive et peu pratique à produire.

**Conséquences:**

Le modèle ne prend pas automatiquement en compte des problèmes comme le partage des objets (sérialisation des accès, maximalisation du parallélisme), la garantie de l'intégrité (accès simultanés, panne logicielle ou matérielle), la minimisation de l'influence du caractère persistant des objets sur le traitement des objets volatiles (espace mémoire, temps de calcul), l'optimisation des performances pour l'accès aux objets qui sont persistants.

**Critique de l'implémentation**

La mise en oeuvre de la persistance ne gérant pas le partage des objets dans le cadre d'une approche incrémentale, tous les objets d'un environnement doivent être chargés avant tout action sur l'un de ses dépendants.

Par ailleurs, la mise en oeuvre est très dépendante de l'outil persistant. Par exemple, au niveau des protections le grain est celui du système de fichier Unix (la consultation et la modification des objets persistants dépendent des droits d'accès du fichier associé à l'environnement); la classe *ENVIRONMENT* n'offre aucun service de protection supplémentaire.

Par ailleurs, il n'est possible de consulter qu'un environnement à la fois, ce qui diminue la flexibilité du système: il est difficile dans ces conditions de réaliser un partitionnement des données indépendamment des programmes Eiffel, ce qui est

notamment utile du fait de la gestion des protections qui est basée sur le système de fichier Unix. Par ailleurs, le chargement en une fois de tous les objets suppose une limitation de fait, sur le nombre d'objets persistants qu'il est possible de manipuler simultanément.

## 2.4. Approche suivie dans ce travail

### Objectifs

Cette étude a pour but l'intégration de la persistance dans le langage et l'environnement *Eiffel*, en se conformant à un triple objectif:

• Intégrer des moyens de manipulation sélective d'une collection d'objets, représentant les instances d'une classe Eiffel,

• Etendre le concept de persistance dans le langage Eiffel par la prise en compte des critères d'Atkinson (Cf. § 3.1) ainsi que de la possibilité de décrire des traitements en mémoire persistante,

• Concevoir un monde *Eiffel* uniformisant la gestion des entités persistantes (classes et objets), qui soit à la fois homogène et indépendant des choix d'implémentation (fichiers Unix, Serveur d'objets ou SGBD-OO).

## Notre contribution

Par rapport au langage actuel, notre approche apporte plusieurs améliorations, notamment sur:

- **La modélisation de la persistance:**

  - homogénéisation de l'identification des objets, afin de permettre les mêmes possibilités d'accès pour tous les objets persistants (accès navigationnel **et** associatif);

  - introduction du concept de *Collection* qui sert de support aux traitements sélectifs (algébriques), et aux traitements en mémoire persistante, et du concept de *Pcollection* qui permet l'extension du langage des assertions (précondition, postcondition, invariant);

  - transparence des échanges entre les mémoires persistante et volatile;

  - uniformisation de la gestion des composants et des objets persistants du système;

- **les fonctionnalités de la gestion de la persistance:**

  - propagation du type d'un objet dans la mémoire persistante (y compris la propagation des méthodes), afin de rendre possible les traitements en mémoire persistante et la gestion de l'évolution des composants;

  - gestion de la sécurité et de l'intégrité des objets persistants,

- **les techniques d'implantation du *Monde persistant* d'Eiffel:**

  - adaptation de l'exécutif du langage afin de rendre implicite les échanges (chargement, libération, mise à jour, etc), avec la mémoire persistante,

  - utilisation d'une architecture Client/Serveur pour préserver l'indépendance entre les programmes Eiffel et les outils d'implantation de la persistance,

  - utilisation de l'apport des bases de données pour gérer l'aspect persistant (optimisation des échanges, mécanisme de transaction, etc).

# 3

# Modélisation de la persistance

Ce chapitre développe les principaux aspects d'un modèle de persistance pour un langage de programmation orienté objets. Ce modèle a pour objectif de traiter les deux grands types de manipulation d'objets:

• soit par unaccès navigationnel dans un graphe d'objets, ce qui correspond à l'utilisation des langages de programmation classiques;

• soit au niveau collectif, par une sélection d'objets, ce qui est nécessaire pour la gestion de grands volumes de données, et représente donc le mode d'utilisation privilégié des langages de base de données.

Dans ce qui suit, le paragraphe 3.1 décrit les principes essentiels que doit suivre un modèle de persistance: le mot-clé est **transparence** pour l'application.

Le paragraphe 3.2 présente le concept d'**objet persistant,** en accord avec les principes énoncés dans le paragraphe précédentes.

Le paragraphe 3.3 présente un point essentiel du modèle de persistance: l'accès par les valeurs, par opposition à l'accès par un nom symbolique; il amènera à la définition des concepts de **collection** et de **requête sélective**.

Dans le paragraphe 3.4 nous étendons le concept de collection à celui de **Pcollection** afin de prendre en compte la gestion de l'ensemble des instances persistantes d'un même type; la mise en oeuvre du concept de Pcollection permettra une meilleure gestion des contraintes. Dans un langage de programmation comme Eiffel cela permettra d'étendre le pouvoir expressif des assertions.

Enfin, le paragraphe 3.5 propose le concept d'**objet obsolète** pour gérer la récupération des objets persistants qui ne sont plus utiles.

# 3.1. Principes de modélisation de la persistance

Notre approche est basée sur un certain nombre de règles de modélisation. Elles s'appuient sur les trois règles d'or définies par M. Atkinson [Atkinson 1983 1989a, 1989b], auxquelles nous ajoutons deux autres règles qui permettent de préserver les principales qualités du langage Eiffel et promouvoir les solutions efficaces.

**Règle 1: l'indépendance de la persistance**

*Le statut de persistance pour un objet doit être indépendant des programmes qui les manipulent: un objet persistant ne l'est pas pour des programmes particulier. Réciproquement, le code des programmes manipulant des objets persistants doit pouvoir être indépendant de leur statut persistant ou volatile.*

L'intérêt de cette règle se situe essentiellement dans le domaine de la réutilisation des composants. Il est important de ne pas être obligé de considérer, au moment de l'écriture d'un composant, le statut des objets manipulés (persistant ou volatile).

exemple: Le code d'une routine doit pouvoir, dans la plupart des situations, ne pas tenir compte du statut volatile ou persistant des objets qui lui sont passés en paramètre ou référencés par des attributs.

On permet ainsi d'augmenter la lisibilité des routines et surtout l'indépendance des classes vis-à-vis d'une application Eiffel; la plupart des classes ont en effet vocation à être réutilisée dans des applications ayant des objectifs différents.

De même il est aussi important pour la flexibilité des applications d'avoir la possibilité de retarder le moment de décider qu'un objet doit devenir persistant.

**Règle 2: l'orthogonalité des types de données persistants**

*En accord avec le principe général de conception des langages de programmation sur la complétude des types de données, tous les objets, quelque soit leur type, doivent pouvoir être rendus persistants et manipulés comme tels sans limitations*

Quelque soit son type, un objet a le droit de devenir persistant; on ne veut pas, à cause de la non possibilité de rendre certaines structures persistantes, influencer les choix de conception d'un programme.

**Règle 3: l'orthogonalité de la gestion de la persistance**

*La manière d'introduire et de reconnaitre le statut persistant d'un objet est orthogonale au système de type, au modèle d'exécution et aux structures de contrôle du langage.*

Cette règle garantit l'intégrité des objets quelque soient leur statut: tous les objets référencés par un objet persistant restent accessibles après la terminaison du processus, c'est-à-dire sont eux-mêmes persistants. Réciproquement, un objet ne peut être éffacé que lorsqu'il n'y a plus aucun objet qui le référence.

Conséquences:

Il ne doit pas y avoir de mots-clés permettant de spécifier qu'un attribut ou une entité locale adresse un objet persistant; cette approche est cependant utilisée dans certains langages, comme dans l'exemple suivant.

soient
- *a* un attribut,
- *PERSON*, une classe existante,
- **persistent**, le mot clé indiquant que l'attribut référence un objet persistant,

une écriture possible pour exprimer la possibilité que la propriété *a* référence un objet persistant serait:

```
.....
a: persistent PERSON
.....
```

Figure 3.1: Une approche pour dèsigner des propriétés persistante

De même, on ne doit pas définir un objet comme étant instance d'un certain type (ex: *STORABLE_TYPE*), pour que celui-ci puisse devenir persistant (solution qui était choisie dans la version 2.1 d'Eiffel).

soient
- *PERSON*, une classe existante,
- la classe ***STORABLE_TYPE***, qui permet de spécifier que les instances peuvent être persistantes,

une écriture possible pour indiquer que la classe *PERSON* a accès aux services de la persistance serait:

```
class PERSON
inherit   STORABLE_TYPE
    .....
end -- PERSON
```

Figure 3.2: Une approche pour dèsigner des types persistants

Les règles n°1 et n°3 permettent de diminuer le code à écrire et d'augmenter la robustesse des programmes en garantissant que toutes les informations accessibles à partir d'un objet persistant lui resteront accessibles tant qu'il ne décidera pas le contraire (notamment, pas de références pointant dans le vide); un programme n'a pas à contrôler explicitement le va-et-vient des données entre la mémoire volatile et la mémoire persistante

En plus de ces règles énoncées par Atkinson, il nous a semblé utile d'introduire deux nouvelles règles de modélisation. La première se situe comme les trois précédentes au niveau de l'application, alors que la deuxième se place plus au niveau de la mise en oeuvre et donc du serveur d'objet (apport des bases de données).

**Règle 4: L'accès aux objets persistants**

*Tout objet persistant doit être accessible à la fois par son identificateur (accès navigationel) et par son contenu (accès sélectif ou associatif).*

Cette règle est nécessaire, car les accès navigationnels sont insuffisants dans les cas suivants:

- le partage des objets par divers utilisateurs,
- le nombre éventuellement d'objets,
- la diversité des applications,
- le fait qu'un objet puisse être rendu persistant explicitement ou implicitement,

rend parfois insuffisant l'accès à un objet uniquement par son identificateur persistant, et qu'il faut lui ajouter des accès par critères de sélection.

**Règle 5: La sous-traitance des traitements**

*Les traitements réalisés sur un objet persistant doivent être sous-traités de manière transparente à l'outil gérant la persistance; ils sont appelés traitements en mémoire persistante. Les mêmes traitements réalisés sur un objet volatile doivent être à la charge de l'exécutif Eiffel.*

Cette dernière règle, est importante pour exploiter les capacités de traitement en mémoire persistante; l'objectif étant de lui laisser toutes les tâches mettant en jeu la consultation, la sélection ou la modification d'objets persistant. Cela aura pour

conséquence d'augmenter la performance (optimisation de la sélection et de l'accès), la sécurité (sérialisation des accès), l'intégrité (contrôle de type, liaison dynamique, garantie du maintien de la cohérence des objets en cas de panne).

**Autre point important**

*L'introduction de la persistance ne doit pas perturber les autres aspects du langage.*

## 3.2. Concept d'objet persistant

Tout d'abord donnons une définition des notions de programme et de processus Eiffel, telles qu'elles seront intégrées dans le modèle.

**Définition 3.1: programme et processus (Eiffel)**

*On appelle **programme** Eiffel tout système de classes ayant une classe choisie comme racine. Un **processus** Eiffel est une exécution possible, d'un programme Eiffel, pour un utilisateur donné.*

**Définition 3.2: utilisateur (Eiffel)**

*On appelle **utilisateur** (Eiffel), toute personne ayant un nom et un identifiant secret, qui est responsable de la création d'un processus Eiffel.*

La notion de persistance est une propriété intrinsèque qui peut être associée à tout objet afin de permettre d'exprimer son indépendance vis-à-vis des différents programmes et processus qui l'utilisent. Cette propriété implique la possibilité de réutiliser les objets persistants dans des programmes différents.

**Définition 3.3: Objet persistant / Objet volatile**

*Un objet Eiffel est dit **persistant** si, et seulement si sa durée de vie est indépendante de celle du processus Eiffel qui l'a créé. Au contraire, tout objet dont la durée de vie est limitée à celle du processus Eiffel qui l'a engendré, est appelé objet **volatile***

### Contraintes de la modélisation

En accord avec les principes mis en avant dans le paragraphe 3.1, nous donnons une série de contraintes permettant de mieux exprimer la sémantique du concept d'objet persistant.

**C3.1**: la qualité de persistance d'un objet est indépendante de son type (règle n°1);

**C3.2**: tout objet volatile Eiffel peut devenir persistant et inversement tout objet persistant peut devenir volatile (règle n°2);

**C3.3**: tout objet a un et un seul identificateur, indépendant des processus Eiffel qui le manipulent (règle n°3). Cette identificateur est appelé Identificateur d'Objet Persistant (POI), et est créé lorsque l'objet devient persistant et celui-ci le garde tout au long de son existence;

**C3.4**: quand un objet volatile Eiffel devient persistant, tous les objets qu'il référence (s'il en existe) deviennent aussi persistants (règle n°3). En cas d'attachement d'un objet volatile à un attribut d'objet persistant, l'objet devient lui-même persistant;

**C3.5**: Un objet local à une routine (entité locale, résultat de fonction ou paramètre) n'est pas un attribut et est donc à priori volatile; il ne peut devenir persistant que par une action spécifique ou par attachement à un attribut d'objet persistant.

Remarque:

Un objet persistant peut être partagé par plusieurs processus Eiffel différents (de C3.4 et de la définition 3.3).

# 3.3. Modélisation des mécanismes de sélection

Nous avons montré l'importance de pouvoir atteindre un objet persistant, en fonction de son contenu. On présente donc à travers le concept de *requête sélective*, une modélisation des traitements sélectifs pour le langage Eiffel.

Le paragraphe 3.3.1, souligne l'importance d'une définition formelle d'un langage de requête, qu'il soit intégré dans un langage de programmation ou utilisé interactivement.

Dans la modélisation d'un mécanisme de sélection, il faut considérer la description des critères de sélection ou des opérations mais aussi la cible, la forme du résultat d'une requête, et les différents opérateurs associés au mécanisme de sélection; ces aspects seront abordès dans les paragraphes 3.3.2, 3.3.3, et 3.3.4.

Dans le paragraphe 3.3.5 enfin, nous discutons l'adéquation de notre modèle aux critères de qualité mentionnés dans le paragraphe 3.3.1.

## 3.3.1. Qualités d'un mécanisme de sélection

Dans ce paragraphe, nous étudions les langages de requêtes des SGBD; pour les langages de programmation les critères de qualité important seront les mêmes, sauf l'aspect interactif.

### Descriptivité du mécanisme

Un des aspects principaux d'un langage de requêtes est la descriptivité du langage [Bancilhon 89, Heuer 91, Loizou 91], c'est à dire sa faculté à permettre l'accès associatif, sa simplicité d'utilisation et sa concision. On ne doit pas avoir à expliquer le comment de la réalisation d'une requête (programmation procédurale), mais seulement le résultat que l'on veut obtenir (programmation déclarative). On notera donc que l'utilisation d'itérateurs ne peut être à elle seule, une réponse pour l'obtention du label de «*mécanisme de sélection descriptif*».

Le langage de requêtes de référence est sans nul doute SQL, pourtant conçu pour le modèle relationnel, mais qui sert de base à la plupart des langages de requêtes des systèmes de gestion de base de données orientés objets; c'est le cas de *O2-Quer*y [Deux 91], Object SQL de Ontos [Andrews 91b], Vbase [Andrews 91a], iris [Horowitz 91], ou Orion [Kim 91]).

### Techniques pour l'optimisation

C'est le langage de requêtes qui gère lui-même le parcours des objets, c'est donc à lui de réaliser les optimisations nécessaires à ce parcours [Bancilhon 89]. On remarquera que ces optimisations sont différentes selon l'opérateur et le constructeur utilisé, c'est pourquoi il existe souvent un mot-clé pour caractériser une structure parcourable, ou un opérateur. Ainsi, même si les différences ne concernent que la flexibilité de la taille des structures (tableau, liste), ou l'unicité des éléments (liste et ensemble) on aura souvent des mots-clés différents pour les caractériser; c'est le cas dans *O2* pour utiliser une liste ou un ensemble [O2-Technology 91a]. Le même type de remarque s'applique aux opérateurs de sélection, ou aux itérateurs; on constate souvent la présence de structures de contrôle différentes pour parcourir une collection d'objets persistants ou pour une autre structure contenant des objets volatiles; c'est le cas dans l'interface C++ de *O2* [O2-Technology 91a], ou dans *Napier 88* [Morrison 89].

Ce problème d'optimisation de l'accès aux objets nous amènera en particulier à nous pencher sur les limites de la généralité des mécanismes permettant de mettre en oeuvre le processus de sélection.

### Accès aux objets persistants

On ne doit pas demander à un langage de requêtes d'être suffisant pour les traitements, sa principale raison d'être est la récupération de données [Bancilhon 89, Heuer 91]; leur description est l'affaire des langages de programmation, éventuellement de base de données lorsque le mécanisme de sélection s'intégre dans un SGBD. Lorsqu'un langage de requêtes est complet du point de vue des traitements (et donc permet les traitements itératifs), le système ne peut garantir qu'une requête va se terminer dans un temps fini; il est au contraire préférable qu'un langage de requêtes soit **sûr** [heuer 91, Loizou 91].

Par contre, le mécanisme de sélection doit être parfaitement adapté au modèle de données utilisé par le langage de programmation, afin d'éviter les problèmes d'incompatibilité entre mode d'exécution (*impedance mismatch*), et d'assurer l'orthogonalité de la sélection par rapport aux structures sur lesquelles on peut effectuer des sélection. C'est en particulier pour ces problèmes d'incompatibilité que l'intégration de la persistance dans un langage à objets, par l'utilisation d'une base de données relationnelle, ne semble pas raisonnable.

### Formalisme des langages de requêtes

D'autres qualités sont nécessaires pour les langages de requêtes [Loizou 91, Heuer 91] comme la fermeture, l'adéquation, l'orthogonalité, l'extensibilité, le formalisme de la sémantique et la présence d'opérateurs génériques.

Dire qu'un langage de requêtes est **fermé** signifie que tout résultat délivré par une requête est peut-être utilisé pour en initialiser une autre. Par exemple, le modèle du langage de requêtes de *Ontos* (Object-SQL) [Andrews 91 b], n'est pas fermé puisque le résultat d'une sélection est une valeur, alors que le modèle d'*Ontos* ne repésente que des objets avec un identificateur.

Le fait qu'il y ait adéquation entre le modèle de données et le langage de requêtes correspond au fait que le résultat s'adapte à tout le modèle et non pas seulement à une partie. Par exemple si les sélections ne peuvent se faire que sur un type particulier du modèle (ex: le type **Relation**), et que le résultat est encore une **Relation**, alors il y aura **fermeture** du langage de requêtes mais pas **adéquation**.

Un langage de requêtes est **orthogonal** et **extensible**, respectivement s'il permet une combinaison orthogonale de tous les opérateurs en fonction de la structure imbriquée des constructeurs de type utilisés dans la requête, et s'il rend possible l'ajout de nouveaux constructeurs de type (tableaux, listes).

Le langage de requêtes de *O2* est orthogonal; il permet entre autre de décrire la clause *From* d'une requête par une autre requête, par contre ce n'est pas le cas pour celui de Ontos ou pour les SQL relationnels.

Lorsqu'il est possible de décrire les opérateurs d'un langage de requêtes par une sémantique formelle, exprimable avec la logique du premier ordre, cela permet de mettre en oeuvre des stratégies d'optimisation assurant l'équivalence avec la forme initiale. C'est le cas en particulier du langage de requêtes *RELOOP* conçu à l'origine pour *O2* [Cluet 89].

Il est aussi nécessaire de trouver dans un mécanisme de sélection des opérateurs génériques sur les ensembles, les tuples et tous les autres constructeurs du modèle; parmi ces opérateurs on trouvera en particulier ceux permettant d'exprimer l'égalité: égalité de valeur, égalité d'identificateur, égalité profonde ou non).

### 3.3.2. Concept de collection

Tous les mécanismes de sélection s'applique sur des collections d'objets qui expriment soit l'ensemble des instances propres à un **type**, soit une partie seulement de ces instances. Nous introduisons donc dans le langage Eiffel un concept de collection destiné à rassembler les données à sélectionner.

Dans le langage Eiffel, l'aspect intensionel d'un type est assez bien pris en compte par le concept de *classe*, mais par contre peu d'outils existent dans le langage pour gérer les instances d'une classe. Il faut donc ajouter dans le modèle une structure que nous appelons *Collection*, qui permette de regrouper les instances d'une classe, avec des opérateurs pour réaliser des sélections, ou des opérations de mises à jour.

Cependant, en Eiffel, "classe" et ""type" ne coïncident pas à cause de la généricité. La

sémantique apportée par le type effectif d'un paramètre de généricité peut être plus riche que celle de la classe générique: le type *ARRAY [WINDOW]* évoque davantage un traitement de multifenêtrage qu'un traitement de tableau. Ainsi il est logique d'associer à chaque type une collection et non pas à chaque classe.

**Définition 3.4: Concept de collection**

*Soit:*

- *T l'ensemble des types Eiffel,*
- *O(i) l'ensemble des objets Eiffel* ***conformes*** *à i, avec* $i \in T$

*alors:*

*une collection C d'objets du type t notée C(t), est une paire (t, {oi}) où* $t \in T$, $oi \in O(t)$, *et* $\{oi\} \subset O(t)$; *le type t est appelé* ***base*** *de la collection.*

### Conséquences

- Une *collection* est indépendante du statut des objets (volatile, persistant),

- une *collection* ne peut contenir d'objets appartenant à deux classes différentes, sauf s'il existe un lien d'héritage entre ces deux classes. La *base de la collection* est le type le plus haut dans la hiérarchie,

- une collection peut contenir toutes les instances d'un type,

- un objet peut se trouver dans plusieurs collections.

## 3.3.3. Concept de requête sélective

Nous définissons dans ce paragraphe les mécanismes de sélection qui doivent agir sur les collections.

Pour modéliser une *requête sélective*, on introduit dabord un concept de *formule bien formé*, où les quantificateurs existentiel et universel sont associés à une collection. L'association des quantificateurs au concept de collection plutôt qu'à celui de type (comme dans le modèle relationnel), permet une plus grande souplesse d'utilisation: on peut non seulement manipuler tous les objets d'un même type (extension de ce type), mais aussi tous les objets conformes à ce type (à un sous-type).

Introduire des quantificateurs existentiel et universel nous amène ensuite à définir les concepts de variables libres et liées débouchant sur celui d'expression sélective.

## Variable de collection

On veut appliquer des critères de sélection sur tous les objets d'une collection; il faut donc introduire des "curseurs" permettant de modéliser le parcours de la collection, ces curseurs sont appelés des **variables de collection**.

**Définition 3.5: Variable de collection**

*Soit V une entité locale*[1] *du langage Eiffel, V est appelé variable de collection si et seulement si il existe une collection C telle que les valeurs possibles de V appartiennent à C.*

## Construction d'un critère de sélection

On définit les entités et combinaisons d'entités (attribut, fonction, constante), faisant partie de la définition d'une expression Eiffel, qui doivent pouvoir participer à la construction d'un critère de sélection.

**Définition 3.6: Concept de terme**

*soient pour le langage Eiffel:*

- *T, l'ensemble des noms de type*[2]
- *A, l'ensemble des attributs,*
- *F, l'ensemble des fonctions et de constantes,*

*alors:*

- $a \in A, f \in F, v \in T$, *sont des termes,*
- *si t1 et t2 sont des termes, alors t1.t2 est un terme*

---

[1] On rappelle que les entités locales sont soit des entités déclarées dans la clause *local* d'une routine Eiffel, soit des paramètres de cette routine.

[2] un nom de type est soit un nom-de-classe, soit nom-de-classe [nom-de-classe1, ..., nom-de-classeN], cf. Eiffel [Meyer 91].

Exemple:

soient

- des attributs ou fonction de la classe *STUDENT*
  . *address*, *name*, *degree* de type *STRING*,
  . *age*, *birth_date*, *grant*, de type *INTEGER*,
- *a_student*, *roomate*, deux attributs de type *STUDENT* d'une classe *C*

alors, par exemple: *address*, *age*, *a_student.age*, *roomate.address*, *a_student*, sont des termes. On notera aussi (si *equal* est une fonction de la classe *STRING*), que: *a_student.address.equal ("Nice")*, est un terme.

On utilise les opérateurs de comparaison Eiffel afin de combiner entre eux les termes et les littéraux pour former les plus petites expressions booléennes composant un critère de sélection.

**Défintion 3.7: Atomes booléens**

*Soient:*

- *l, m ∈ L, des littéraux (entier, réel, booléen, caractère, ...),*
- *t, u ∈E, des termes,*
- *δ ∈¥, un opérateur de comparaison : <, =, >, /=, ≤, ≥,*

*alors les formes suivantes sont des atomes: l δ m, t δ l, l δ t, t δ u*

Exemple:

Soient les entités de l'exemple précédent, alors:

- *a_student.age = 18*,
- *a_student.age > roomate.age*,
- *a_student.address = roomate.address*,

sont des atomes.

On montre comment combiner les termes et atomes définis ci-dessus pour construire des critères de sélection (formule bien formée) complexes représentant des conjonctions, disjonctions, des négations d'atomes, ou introduisant des quantificateurs.

Dans un souci de concision, nous n'avons pas intégré dans une formule bien formée tous les opérateurs logiques offerts par le langage Eiffel.

Les opérateurs *not*, *and*, *or*, de la logique du premier ordre ont leur équivalent en Eiffel et les opérateurs *xor* et *implies* peuvent s'écrirent comme des conjonctions et des disjonctions, et les opérateurs *or else* (respectivement *and then*) sont des *or*

(respectivement *and*) conditionnels et non commutatifs.

**Définition 3.8: Formule Bien Formée**

*Soient:*

- *A l'ensemble des atomes,*
- *L l'ensemble des symboles d'opérateurs logiques: and, or, not,*
- *S l'ensemble des symboles séparateurs: (, ),*
- *C l'ensemble des collections,*
- *V l'ensemble des variables de collection,*
- *T, l'ensemble des types,*

*On appelle* ***Formule Bien Formée*** *(FBF) à une seule variable libre $x \in V$, notée $[x \in V]$ FBF, toutes les formes suivantes:*

- *Tout atome défini à partir des termes d'un type $t \in T$ est une FBF,*
- *si $[x \in V]$ A, $[x \in V]$ A1 et $[x \in V]$ A2 sont des FBF*

  *alors:*

  - *($[x \in V]$ A) est une FBF,*
  - *$[x \in V]$ A1 and $[x \in V]$ A2 est une FBF,*
  - *$[x \in V]$ A1 or $[x \in V]$ A2 est une FBF,*
  - *not $[x \in V]$ A est une FBF,*
  - *$\forall y \in V_i$ / $[x \in V]$ A est une FBF,*
  - *$\exists y \in V_i$ / $[x \in V]$ A est une FBF.*

Exemple: Formules bien formées

Soient

*x* une variable de collection sur une collection C d'objets de type *STUDENT*,

*t* un terme retournant une entité de type *STUDENT*,

*y* une variable de collection sur une collection *d'objets de type FLAT*,

Alors, on pourra écrire:

- (x.address = “Nice”) or (x.address = “Paris”) and (x.name = t.name)
- x.name.equal ("DURANT"),
- x.name = "Laurie" and $\forall$ y $\in$ *C* / y.nb_rooms = 10.

Ce dernier exemple illustre une des possibilités du modèle pour l'écriture des requêtes emboîtées.

L'intégration des quantificateurs dans une formule bien formée nécessite la définition des concepts de variable liée et libre, pour une variable de collection.

**Définition 3.6: Variable libre / Variable liée**

*Soient*

- *$[x \in V_C] F$,*
- *$V(F)$ l'ensemble des variables de collection de F,*

*On appelle* ***ensemble des variables liées*** *de la formule F, noté Vliée(F), le sous-ensemble de $V_C(F)$ défini par récurrence par:*

- *si F est un atome: Vliée(F) = $\varnothing$,*
- *si F est de la forme (A and B): Vliée (F) = Vliée (A) $\cup$ Vliée (B),*
- *si F est de la forme (A or B): Vliée (F) = Vliée (A) $\cup$ Vliée (B),*
- *si F est de la forme not A: Vliée (F) = Vliée(A),*
- *si F est de la forme $\forall y \in V$ A: Vliée (F) = Vliée (A) $\cup \{y\}$,*
- *si F est de la forme $\exists y \in V$ A: Vliée (F) = Vliée (A) $\cup \{y\}$.*

*On appelle* ***ensemble des variables libres*** *de la formule F, noté Vlibre (F), le sous-ensemble de V (F) défini par récurrence par:*

- *si F est un atome: Vlibre (F) = V (F),*
- *si F est de la forme (A and B): Vlibre (F) = Vlibre (A) $\cup$ Vlibre (B),*
- *si F est de la forme (A or B): Vlibre (F) = Vlibre (A) $\cup$ Vlibre (B),*
- *si F est de la forme not A: Vlibre (F) = Vlibre (A),*
- *si F est de la forme $\forall y \in V$ A: Vlibre (F) = Vlibre (A) - $\{y\}$,*
- *si F est de la forme $\exists y \in V$ A: Vlibre (F) = Vlibre (A) - $\{y\}$.*

A partir des concepts de *formule bien formée* et de *variables libres et liées*, on déduit celui d'*expression sélective*, qui nous permet de définir les fonctions et les itérations sélectives; on généralisera enfin au concept de requête sélective.

**Définition 3.10: Expression sélective**

*Soit $x \in V$ une variable de collection, d'une collection C, on appelle* ***expression sélective sur C****, toute Formule Bien Formée (FBF) F à une seule variable libre x, notée* ***$[x \in C] F$***

**Définition 3.11: Fonction sélective**

*On appelle **fonction sélective** sur une collection C, toute fonction retournant une sous-collection de C, vérifiant une expression sélective sur C. Cette sous-collection est appelée une **sélection**. On peut noter la description d'une fonction sélective sur C comme suit:*

*requête-identifier [x ∈ C] F*

*où*

- *requête-identifier est le nom de la fonction,*
- *F est une expression sélective sur C*
- *x est la variable libre de F.*

Nous avons donc défini ici la forme du résultat d'une requête, c'est une collection, de même classe de base, représentant un sous-ensemble de la collection correspondant à la cible de l'opération de sélection. Le fait que l'on obtienne une nouvelle collection permet d'envisager la composition de fonctions sélectives.

Exemples:

Soit E une collection d'objets de la classe *STUDENT*,

1) RQ1: Quels sont les étudiants de E qui habitent à Nice et qui,
   - soit s'appellent DURANT,
   - soit sont agés de plus de 30 ans?

On définit ainsi la fonction sélective *RQ1 [x ∈ E] F* où F = (x.address = "Nice") and ((x.name = "DURANT") or (x.age > 30))

2) RQ2: Quels sont les étudiants de E habitant Nice qui ont le même nom qu'un autre étudiant de E habitant Paris?

On définit ainsi la fonction sélective *RQ2 [x ∈ E] F où* F = ∃*y ∈ E /* (x.address = "Nice") and (y.address = "Paris") and (x.name = y.name)

**Définition 3.12: Itération sélective**

*On appelle **itération sélective** sur une collection C, toute itération qui active une procédure R sur tous les objets de C, qui vérifient une expression sélective F sur C. On peut noter la description d'une telle procédure comme suit:*

*iteration-identifier R / [x ∈ C] F*

*où*

- *iteration-identifier est le nom de l'itération,*
- *R est la procédure (décrite dans la classe de base de C) à appliquer à chaque objet,*
- *F est une expression sélective sur C,*
- *x est la variable libre de F.*

Exemples:

Soit E une collection d'objets de la classe *STUDENT*,

1) Pour tout étudiant de S qui est né après 1970 et qui a le Bac C, on distribue une bourse de 700 F par mois.

On définit ainsi l'itération sélective *PROC1 P / [x ∈ E] F* où:
F = (x.birth_date > "1/1/1970") and (x.degree = "Bac C") et,
P = x.set_grant (700)

2) Pour tout étudiant de S, on distribue une bourse de 700 F par mois.

On définit ainsi l'itération sélective *PROC2 P / [x ∈ C] F* où:
F = *True* et,
P = x.set_grant (700)

On notera que l'avant dernier exemple présenté ici est une combinaison entre une sélection et une commande de mise à jour, alors que le dernier est seulement une mise à jour.

**Définition 3.13: Requête sélective**

*On appelle **requête sélective** une fonction ou une itération sélective.*

La description des concepts d'expression sélective et de requête sélective nous montrent que plusieurs collections peuvent être la cible d'une même requête sélective. Les cas où une requête sélective peut avoir plusieurs cibles sont:

• une ou plusieurs expressions pointées (navigation dans le graphe des références) participent à la définition de l'expression sélective,

• plusieurs collections de même type sont concernées par la requête, à cause de regroupement d'objets réalisées antérieurement (composition de requêtes sélectives),

• collection contenant d'autres collections, par la présence d'une hiérarchie entre les types (structure contenant des objets conformes à un type commun),

• la requête sélective contient plusieurs niveaux d'emboîte ment (est formée de plusieurs requêtes sélectives sur la même classe, ou des classes différentes.

### 3.3.4. Description des opérateurs de manipulation d'une collection

Une fois construite par une requête sélective, ou par une action explicite de création, une collection doit pouvoir être manipulée par des opérateurs ensemblistes et des opérateurs séquentiels

**opérations sur les ensembles**

On pose:

- $c \in C(t)$, *l'ensemble des collections du type t,*
- $c1 \in C(t1)$, *l'ensemble des collections du type t1,*
- $c2 \in C(t2)$, *l'ensemble des collections du type t2,*

**Définition 3.14: Opérateur union**

*l'opérateur Union noté* $\cup$ *est une fonction:* $E(t1) \times E(t2) \rightarrow E(t1)$, *avec t2 est conforme à t1, et* $c1 \cup c2 = \{x \mid x \in c1 \text{ or } x \in c2\}$

**Définition 3.15: Opérateur intersection**

*l'opérateur intersection noté* $\cap$ *est une fonction:* $E(t1) \times E(t2) \rightarrow E(t2)$, *avec t2 est conforme à t1, et* $c1 \cap c2 = \{x \mid x \in c1 \text{ and } x \in c2\}$

**Définition 3.16: Opérateur différence**

*l'opérateur différence noté $\Delta$ est une fonction: $E(t1) \times E(t2) \rightarrow E(t1)$, avec t2 est conforme à t1, et $c1 \Delta c2 = \{x \mid x \in c1 \text{ and } x \notin c2\}$*

**Définition 3.17 Opérateur d'inclusion**

*l'opérateur d'inclusion noté $\subset$ est une fonction: $E(t1) \times E(t2) \rightarrow \{True, False\}$, avec t2 est conforme à t1, et $c1 \subset c2 = \{x \mid x \in c1 \rightarrow x \in c2\}$*

### 3.3.5. Discussion sur le modèle

Le paragraphe 3.3.1 a présenté les qualités que devaient avoir le modèle d'un langage de requête; il s'agit de:

- la complétude et la sureté,
- la fermeture,
- le formalisme,
- l'adéquation au modèle du langage de programmation (Eiffel dans notre cas).

Le modèle décrit ci-dessus répond à un formalisme basé sur la logique des prédicats. Pour intégrer le concept de collection, il a fallu introduire les définitions concernant les formules bien formées, les quantificateurs existenciels ou universels, et les variables libres ou liées. Ces définitions permettent d'établir une correspondance avec la logique des prédicats, et d'obtenir une sémantique formelle.

Le modèle garantit la fermeture du langage de requête; en effet toute requête sélective se fait sur une collection et le résultat de ces requêtes est une autre collection contenant des éléments conformes au type de base de la collection sur laquelle est effectuée la requête sélective. Il est donc possible d'effectuer des compositions de requêtes sélectives.

L'adéquation aux autres structures du langage Eiffel est aussi assurée. En effet le langage Eiffel supporte un seul type de structure, les *classes* (expansées, génériques, ..) et un seul type d'entité dynamique, l'*objet*; les requêtes sélectives opèrent sur des collections d'objets d'une classe de base, et leur résultat est aussi une collection.

Le modèle introduit un minimum de concepts (essentiellement expression, fonction et itération sélective). Ils permettent de définir un large éventail de critères de sélection, qui peuvent s'appliquer non seulement aux ensembles mais à toute structure parcourable (graphe, liste, file, ....) (cf. § 6.6).

L'orthogonalité n'est pas non plus oubliée, puisqu'une expression sélective peut

contenir des fonctions sélectives, introduisant ainsi des possibilités d'emboîte ment. De plus, une requête sélective peut s'appliquer sur des collections de collection, puisque la classe collection hérite elle aussi du type le plus général.

Par ailleurs, le langage est complet puisque d'une part dans un critère de sélection on accepte des fonctions du langage Eiffel (voir définitions 3.11 et 3.12), et d'autre part, toute action d'une itération sélective est une procédure Eiffel. L'avantage de cette démarche est le niveau d'intégration du mécanisme de sélection dans le langage Eiffel: toute expression Eiffel retournant un résultat est exprimable dans une expression sélective. Par contre, l'inconvénient sous-jacent est que notre mécanisme de sélection n'est pas sur; on ne peut garantir qu'une requête sélective se terminera (boucle infinie, ...).

## 3.4. Modélisation de l'extension d'un type

L'extension d'un type représente l'ensemble des instances de ce type; si l'on considère les langages de programmation classique, il est rare de disposer implicitement de cette possiblité, à cause de son surcoût important, et du fait qu'elle corresponde à un besoin moins fondamental pour des objets volatiles.

Par contre, dans les langages de programmation de base de données, il n'est pas rare de disposer de ce service pour les instances persistantes. Cependant le fait de considérer l'extension d'un type soulève un certain nombre de contreverses. Ainsi [Bloom 87], pose trois questions fondamentales:

- Q1: est-ce que les avantages d'un tel mécanisme justifie l'augmentation de complexité? Pour des systèmes contenant un nombre important de types, pour lesquelles le besoin de gestion des instances se fait sentir, une gestion explicite alourdit le code; la philosophie des langages de programmation est donc de plutôt laisser cette tâche au programmeur. La gestion de l'extension des types permet un meilleur contrôle des contraintes d'intégrité et des modifications dans la hiérarchie des types (surtout dans un univers persistant); elle autorise l'introduction d'opérateurs existenciels et universels de sémantique réduite, utile cependant pour les contrôles d'assertions. Cependant, exécuter des sélections sur toutes les instances d'un type peut engendrer des problèmes d'efficacité, surtout quand une application n'aurait besoin de consulter qu'une collection d'objets avec très peu d'éléments.
- Q2: est-ce qu'une classe vue comme une collection interfère avec son utilisation et sa sémantique lorsqu'elle est vue comme un type? il faut considérer deux aspects:

- la qualité du schéma conceptuel: le regroupement automatique réalisé par le système (extension de la classe) peut induire une diminution des groupements explicites qui seraient nécessaires dans la phase de conception (groupement des objets moins pertinents),

- le respect de l'encapsulation: la vision abstraite des objets peut être affectée par la gestion de l'extension d'une classe qui, en fournissant un accès direct aux objets, n'oblige pas à un accès navigationnel à travers l'interface des classes clientes, qui révèleraient la violation de certaines contraintes associées à un objet. cet aspect sera développé dans le paragraphe 3.4.3;

• Q3: quelle est l'interaction avec les autres fonctionalités du langage, en particulier, les langages peuvent ils supporter la gestion automatique du stockage, avec utilisation de ramasse-miettes pour l'effacement, si la notion d'extension de type est supportée? En d'autres termes, si chaque classe a son extension, alors il restera toujours au moins une référence sur un objet, et le ramasse-miettes ne le supprimera jamais (les références de l'extension peuvent cependant être traitées de manière particulière).

[Bretl 89] en rajoute une supplémentaire (Q4): l'extension d'un type doit-elle contenir les instances des sous-types descendants (types conformes)? On remarquera que la gestion de l'extension des types conformes est fortement liée aux possibilités des outils qui implémentent la requête (cf. § 3.4.3 et 4.2.2).

Dans une première section (paragraphe 3.4.1), nous proposons le concept de **Pcollection** et nous montrons comment il pourrait être utilisé dans un langage à objets comme Eiffel pour étendre le pouvoir expressif des assertions.

Dans le paragraphe 3.4.2, nous décrivons les contraintes de modélisation nécessaires à la prise en compte des interrogations soulevées ci-dessus (préservation de la vision abstraite des objets, profondeur d'une Pcollection).

### 3.4.1. Concept de Pcollection

#### Définition des concepts

Une *collection* peut être volatile ou persistanteet es gérée explicitement par l'application. Nous introduisons ici des collections particulières appelées **Pcollections**, qui sont gérées par l'exécutif Eiffel, et permettent d'adresser toutes les instances persistantes d'un type et, accessibles à des processus logiquement reliés (voir la partie II de ce travail).

**Définition 3.18: Pcollection d'un typee**

*A tout type d'un ensemble de processus Eiffel logiquement reliés est associée une unique collection contenant tous ses objets persistants. Cette collection persistante est la* ***Pcollection*** *associée au type considéré.*

#### Quantifications, expressions logiques et assertions

L'intérêt des quantifications n'est évidemment pas limité aux seules manipulations de collections. L'une des grandes qualités du langage Eiffel est la présence des **assertions** qui sont une aide très efficace pour la mise au point et pour la documentation des programmes. Aussi, disposer d'une grande expressivité pour écrire des assertions est fondamental pour atteindre ces objectifs. L'introduction de quantificateurs dans les expressions logiques et les assertions d'Eiffel permettrait donc d'enrichir de manière utile ce langage, sans ajouter de nouvelles structures syntaxiques.

Toutefois, il n'est évidemment pas possible d'utiliser les quantifications avec le même sens qu'en mathématique ou en logique du premier ordre, c'est-à-dire en considérant tous les éléments d'un ensemble fini ou non, dénombrable ou pas. Il faut ici se restreindre à prendre comme référenciel une collection. Dans bien des cas, ce sera une collection particulière, explicitement construite. Mais on pourra aussi envisager la collection de toutes les instances persistantes d'une classe: la **Pcollection** associée à cette classe, à partir de laquelle, on peut générer n'importe quel sous-ensemble: par exemple l'ensemble des instances d'une classe, créée depuis une certaine date, par exemple, depuis le lancement d'une application.

Comme l'on peut craindre un coût important à l'exécution pour évaluer de telles quantifications, il faudra la plupart du temps n'évaluer les assertions avec quantificateurs qu'en phase de mise au point. Probablement, cela nécessitera d'affiner le mécanisme de sélection des assertions à évaluer, pour permettre de choisir clause par clause ce qui doit être évalué: les *labels* d'assertions pourraient aussi servir à cela.

```
deferred class GROUP
feature
        zero: like Current is deferred end;

        infix "+" (other: like Current): like Current is deferred end;

        prefix "-": like Current is deferred end;

        value: VALUE;      -- value of current object

        set (new:  VALUE) is
        ensure      value = new
        end; -- set

invariant           -- Pcollection invariant
        -- associativity:        x, y, z ∈ PCOLLECTION      (x + y)+z = x + (y + z)

        -- neutral_element     x ∈ PCOLLECTION            zero + x = x + zero = x

    -- symetric element:   x ∈ PCOLLECTION            ∃ -x, x + -x = -x + x = zero

end -- class GROUP
```

Figure 3.3: Spécification quasi-formelle d'un groupe avec quantificateurs.

L'utilisation de quantifications dans des invariants de classes donne des perspectives nouvelles au langage Eiffel, pour écrire des pseudo-axiomes qui ne portent que sur la *pcollection* d'une classe, mais offrent une bonne approximation des axiomes d'une spécification algébrique; cela rapprocherait Eiffel de la famille des langages formels de spécifications exécutables, (voir fig. 3.3).

Ces pseudo-axiomes devraient aussi être démarqués des autres propriétés de l'invariant de classe qui concernent les propriétés d'état des instances: il faut distinguer les **invariants de classe** des **invariants d'instance**.

## 3.4.2. Contraintes de modélisation

Pour répondre aux questions posées au début de le paragraphe 3.4, nous introduisons un certain nombre de contraintes.

**Préservation de l'abstraction et de l'intégrité des objets**

On rappelle qu'Eiffel offre des mécanismes pour contrôler la visibilité (exportation de primitives), et l'intégrité des objets par rapport aux spécifications (vérification d'assertion). Ces services permettent d'augmenter la réutilisabilité et l'extensibilité des composants logiciels, leur fiabilité et leur robustesse. Il est donc important de respecter l'abstraction ou l'encapsulation des données.

Le problème du respect de la vision abstraite des objets n'est pas spécifique à la gestion de l'extension des classes, mais il est seulement rendu moins acceptable du fait de la persistance des objets et de leur partage entre applications. Il est en effet possible d'accéder à un objet par tous les objets qui le référencent et donc de violer les assertions d'un ou plusieurs objets, sans en être averti (au moins immédiatement).

Le problème mis en évidence ici est un problème de **partage d'objets** (voir fig. 3.4); il existe déjà dans les différentes versions d'Eiffel, mais n'est pas vraiment grave s'il n'affecte que des objets volatiles.

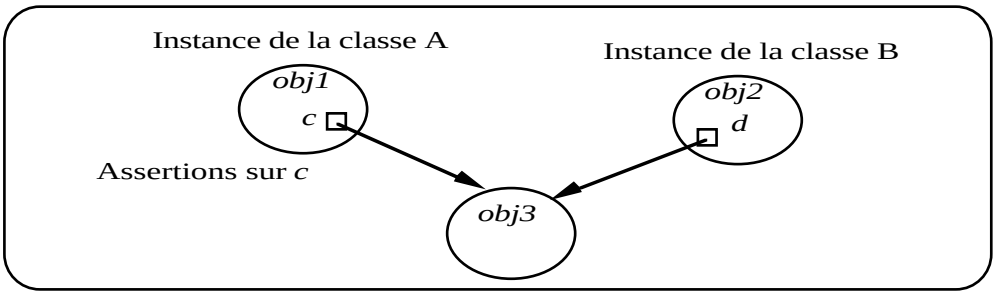


Figure 3.4: Le problème du partage d'objets

En modifiant *obj3* par *obj2* qui ne possède pas d'assertions (ou des assertions plus faibles), on peut, sans que l'application s'en rende compte, violer les assertions de la classe A. Ce cas peut se produire en particulier lorsque la classe B est une structure parcourable (tableau, liste, ensemble, ...). Cependant, cela correspondrait à une erreur de conception ou de programmation, on notera que ce type d'erreur serait plus grave si les instances concernées sont persistantes.

Ainsi, si l'objet *obj3* est persistant, l'erreur se prolonge dans le temps et peut n'être découverte qu'après avoir été propagée à un grand nombre d'objets, affectant ainsi, dangereusement la cohérence du monde persistant.

Le fait même de réaliser des requêtes de sélection ou de modification sur un ensemble d'instances, laisse supposer que l'utilisateur va avoir besoin, entre autre, de faire

référence aux primitives de la classe dont ces instances sont issues (attributs, fonctions et procédure). Or, la vision abstraite d'un objet est basée sur deux principes [Bancilhon 91]:

- un objet regroupe à la fois des données et des opérations,
- la seule façon d'accéder au contenu d'un objet est d'utiliser une des opérations de l'objet, on ne peut accéder au contenu directement

Dans [Bancilhon 91] l'auteur argumente dans le sens que différentes positions peuvent être prises vis à vis de l'encapsulation selon que l'on accède aux objets à travers un langage de requêtes spécifique (interaction avec l'utilisateur), ou à travers un langage de programmation, et selon le mode d'utilisation (consultation ou modification). Ainsi un langage de requêtes spécifique doit pouvoir violer l'encapsulation, mais pas un langage de programmation. En fait c'est un problème de destination entre erreur d'utilisation (pouvant être exprimée par le langage de requêtes) et erreur de programmation pouvant l'être par le langage de programmation.

En Eiffel l'abstraction des données est respectée pour les accès navigationnel. Dans ce but, le langage met en oeuvre plusieurs mécanismes:

- un mécanisme d'exportation des données (données publiques éventuellement de manière limitée, ou privées),

- les attributs ne sont pas modifiables par d'autres objets, mais seulement consultables, à moins que l'interface publique offre des opérations pour le faire,

- un mécanisme d'assertion (invariant, précondition, postcondition), permettant de contrôler la mise à jour des objets et donc d'en préserver la cohérence en phase de mise au point.

Dans [Loizou 91], on encourage cette approche puisque l'idée avancée est que l'encapsulation est un problème qui doit être traité par le modèle (par exemple les clauses publiques ou privés dans Eiffel ou C++), et non pas au niveau du langage de requêtes dans le cas d'un langage de base de données.

L'abstraction n'est pas violée; ce serait le cas si obj3 (Fig. 3.4) pouvait être modifié par B à son insu. C'est un problème de contraintes d'intégrité globale qui sont mal exprimées par des assertions d'instances, mais devraient être exprimées à un niveau plus global

(Base, Monde, ...).

Il est possible à tout processus Eiffel d'accéder à une instance persistante en contournant les assertions mises en place dans une classe; il suffit pour cela de faire une requête sur la *Pcollection* correspondant à la classe de l'objet. Cela a pour effet de rendre possible la violation des assertions (invariant, précondition de routines), qui peuvent être incluses dans une classe cliente de la classe correspondant à la *Pcollection*, sans que le programme ne soit averti de ces violations.

On a vu que ce type de violation peut exister dans la version 3 d'Eiffel, mais le partage des objets est alors toujours réalisé explicitement; ce n'est pas le cas pour les **Pcollections** qui sont gérées automatiquement par le système, et cela augmente donc les risques d'erreurs.

En conséquence, nous proposons de limiter les risques de violation d'assertions en restreignant les opérateurs de collection accessibles à travers une Pcollection.

**Définition 3.19: limitation des opérateurs d'une Pcollection**

*Les seules requêtes sélectives autorisées à partir des Pcollections sont les fonctions sélectives.*

Il y a plusieurs raisons à ce choix:

• on ne veut pas agraver le problème de partage d'objets, mais le mécanisme ne doit pas être lourd à mettre en oeuvre,

• le principal objectif d'une *Pcollection* est de compléter le mécanisme d'assertion,

• Une fonction, selon [Meyer 88], ne devrait pas inclure d'instructions modifiant l'état abstrait d'un objet: elle représente seulement un attribut calculé.

On notera qu'il est toujours possible de réaliser des mises à jour (itération sélective), d'une *Pcollection* à travers la vue de la classe *COLLECTION*,maist dans ce cas, l'action sera explicite.

Pour une solution plus complète du problème de partage d'instances, il faudrait mettre en oeuvre un mécanisme similaire à celui des composants (clause d'exportation, éventuellement limitée), mais cela, risque d'être excessivement lourd...

### Profondeur d'une *Pcollection*

Il nous semble essentiel, contrairement à ce que dit [Bretl 89], qu'une *Pcollection* contienne non seulement les instances de son type de base, mais aussi toutes celles de tous les types conformes; ceci, essentiellement parce que notre objectif est le renforcement des contrôles d'intégrité.

Cependant, quand on gère l'extension d'un type, et que l'on effectue des requêtes sur ses instances, il faut considérer la hiérarchie à laquelle il appartient. Il se pose alors le problème de comment exprimer l'ensemble des éléments sur lequel porte la requête; on doit pouvoir considérer seulement certains des sous typess, et donc définir la profondeur et la largeur d'une requête.

Prenons un exemple: soit une classe *PERSONNE* ayant parmi ses descendants les classes *EMPLOYE* et *RETRAITE*. Comment exprimer qu'une requête devra porter uniquement sur l'ensemble des instances de la classe *EMPLOYE*, ou bien, à la fois sur celles des classes *EMPLOYE* et *RETRAITE*, ou plus généralement sur toutes les instances de la classe *PERSONNE* (incluant entre autres, les instances de *EMPLOYE* et *RETRAITE*).

Dans un langage de programmation comme Eiffel, il n'y a pas vraiment de problèmes pour exprimer la profondeur; deux solutions s'offrent à nous:

- réaliser une composition de fonctions sélectives sur la ou les *Pcollection(s)* concernée(s) en délivrant l'union des résultats,

- prendre la *Pcollection* parente de toutes les Pcollections concernées et restreindre l'ensemble des objets concernés en enrichissant le critère de sélection avec des règles de conformances sur le type des objets. Cela suppose des opérateurs de manipulation de types plus puissant qu'actuellement.

## 3.5. Modélisation du concept d'objet obsolète

Le problème de l'effacement des objets n'est pas seulement un problème d'implantation lié à la gestion de l'extension des classes (cf. § 3.4), mais aussi un problème méthodologique: doit-on forcément **supprimer** une information qui n'est plus utilisée?

Pour des objets volatiles, si pendant l'exécution d'un programme un objet n'est plus référencé, il peut être supprimé par le ramasse-miettes (technique d'ailleurs utilisée par l'exécutif Eiffel). En effet, la plupart du temps ces objets correspondent à des variables locales ayant servi à des calculs intermédiaires, et qui ne sont plus utilisés.

Pour des entités persistantes (classes, objets persistants) il faut tenir compte d'autres aspects:

• le partage des objets se fait dans un univers décentralisé, où les applications peuvent tantôt être génératrices d'informations, tantôt simplement utilisatrices. Une fois que l'entité est créée par une application et rendue persistante, elle est partagée par les autres applications, et échappe à son seul contrôle.

• l'application ayant créé l'entité persistante peut très bien ne plus exister et aucune autre application être capable de la générer à nouveau, si elle est supprimée à tord,

• l'application utilisant une entité persistante n'a pas les connaissances suffisantes pour savoir si, d'une part elle est la dernière à référencer l'entité et si d'autre part l'information correspondant à cette entité est devenue obsolète ou seulement plus temporairement utilisée. Une application ne peut donc pas mesurer les effets de la suppression d'une référence vers cette entité,

• une application utilisatrice doit pouvoir être en mesure d'avertir les autres qu'une information est devenue obsolète et interdire son utilisation.

Par rapport à ces remarques on peut adopter l'une des deux alternatives suivantes:

• supprimer l'effacement automatique et demander une validation préalable pour laisser agir le ramasse-miettes, en mémoire persistante;

• laisser supporter la responsabilité à celui qui a créé l'instance afin que l'objet ne puisse être effacé par erreur, en fournissant des informations supplémentaires lors de la création (statut ineffaçable, personne autorisée, clé), soit au niveau de l'instance, soit à celui de la Pcollection (regroupement).

Nous avons d'abord été tenté par la seconde solution, qui semblait la plus homogène par rapport à l'approche de langages à objets comme Eiffel. Mais cette solution présente les inconvénients suivants:

• alourdir inutilement le schéma de conception en introduisant des regroupements supplémentaires,

• une erreur de conception aurait des conséquences graves à long terme du fait qu'elle toucherait des entités persistantes,

• interférences avec la gestion des Pcollection (cf. § 3.4).

Pour ces raisons nous avons retenu la 1ère solution.

### Remarque

On notera qu'il existe plusieurs formes d'obsolescence:
- obsolescence du contenant (structuration de l'information),
- obsolescence du contenu (validité de l'information).

La première se rapporte à des problèmes d'évolution de composants et de gestion de versions; c'est la cas pour les routines Eiffel déclarées obsolètes [Meyer 88 & 91]. La seconde est typiquement un problème d'effacement tel qu'il a été évoqué ci-dessus.

### Définition des concepts

Dans ce paragraphe nous définissons le concept d'**objet obsolète**; l'idée directrice qui lui correspond est qu'une entité persistante qui n'est plus référencée ne doit pas forcément être supprimée.

Le concept d'objet obsolète devrait permettre de résoudre à la fois le problème d'implantation souligné dans le paragraphe 3.4, et le problème de méthodologie mis en évidence au début de ce paragraphe.

**Définition 3.20: Objet obsolète**

*Un objet obsolète est un objet qui n'appartient plus à aucune Pcollection*

On remarquera que si un objet est retiré de la Pcollection associée à sa classe, alors il n'appartient plus à aucune autre; ce n'est pas le cas s'il est enlevé d'une Pcollection associée à une classe descendante.

## Contraintes de modélisation

**C. 3.6**: un objet persistant ne peut être effacé que s'il est obsolète

Comme cela vient d'être discuté, on ne peut pas toujours considérer que tout objet doit être supprimé automatiquement lorsqu'il n'est plus référencé; la suppression peut être automatique pour des objets volatiles [Meyer 88], mais doit être soumis à un contrôle si l'entité est persistante. Une approche orthogonale traitant identiquement une classe, un objet persistant, et un objet volatile, conduirait à l'effacement automatique des classes qui n'appartiennent à aucune bibliothèque ou application.

L'introduction du concept d'objet obsolète permet d'avoir une démarche plus souple. Un objet qui devient obsolète, sera supprimé automatiquement; dans le cas contraire, le ramasse-miettes ne pourra faire son office. Tant qu'un objet n'est pas obsolète, cela signifie que l'information est toujours jugée d'actualité par son créateur.

**C.3.7**: Un objet partagé ne peut devenir obsolète qu'explicitement

On a vu qu'un objet devient obsolète dès qu'il est retiré de la *Pcollection* correspondant à son type. Il semble raisonnable de dire que lorsqu'un objet persistant n'existe que parce qu'il est référencé par d'autres (cas d'objets référencés seulement par un objet), alors il devient lui aussi obsolète. Par contre l'obsolescence ne peut être propagée aux objets communs à plusieurs objets. On notera que les objets expansés et donc privés à d'autres objets sont naturellement effacés avec l'objet qui les contient (un objet expansé n'appartient à aucune Pcollection).

**C.3.8**: Il doit exister un mécanisme pour alerter un processus actif de l'obsolescence d'un objet persistant qu'il manipule.

Quand un objet obsolète est consulté par un processus actif l'exécutif du langage Eiffel doit avertir l'application que l'objet n'est plus accessible.

## 3.6. Concept de Monde Persistant

L'introduction de la persistance dans un langage, amène la question suivante: doit-on traiter différemment une classe d'un objet persistant alors que ce sont tous deux des entités persistantes?

On étudiera les problèmes et les solutions qui concernent sa propagation et son implémentation dans les chapitres 9 et 12. Mais d'un point de vue conceptuel l'idée où une classe serait vue des applications comme un objet persistant est séduisante; elle permet d'envisager:

- un accès banalisé aux informations (attributs, routines, ...) correspondant à une classe;

- la réalisation rapide d'outils pour inspecter et sélectionner les classes ou les routines que l'on dèsire réutiliser (voir chapitre 15),

- un support pour la gestion de versions.

Pour unifier le traitement et la gestion de toutes les entités persistantes d'une installation Eiffel sur une machine donnée, c'est à dire les **classes avec leur structure et leurs routines**, et les objets persistants, nous introduisons le concept de **monde persistant.**

**Définition 17: Le monde persistant**

*On appelle Monde persistant l'ensemble de toutes les entités persistentes Eiffel sur une machine donnée. On appelle entité persistante Eiffel, les classes et les instances persistantes.*

D'un point de vue opérationnel, ce concept sera affiné par la suite par l'introduction, entre autre, d'un mécanisme de partitionnement (cf. § 6.8.3). Mais à partir du concept de *Pcollection* (paragraphe 3.4), on peut déjà poser un certain nombre de contraintes.

**C3.9**: Un processus est associé à un moment donné à un seul monde persistant,

**C3.10**: Le monde persistant est partagé par tous les processus de la machine sur lequel il a

été créé.

**C3.11**: Le monde persistant est vu comme une **Pcollection** d'objets du type le plus général, et donc il offre des opérateurs de sélections sur des critères communs à tous les objets

**C3.12**: le monde persistant est unique et c'est la première entité persistante créée; cette création est faite lors de l'installation de l'environnement Eiffel,

**C3.13**: tous les objets persistants sont potentiellement accessibles à partir du monde persistant.

# Partie II

# Conception de la persistance en Eiffel

Dans cette seconde partie, l'objectif est d'appliquer les concepts important du modèle précédent, au langage Eiffel.

Dans les chapitres 5 et 6, nous proposons une intégration des concepts du modèle:

- entité persistante: objet persistant, objet obsolète et classe (chapitre 5),
- collection, requête sélective, pcollection (extension de type) et monde persistant (chapitre 6).

Cette intégration devra vérifier les principes de modélisation présentés dans le paragraphe 3.1, et considérer les différentes solutions utilisées par les langages persistants que l'on retrouve le plus souvent dans la littérature. A ce sujet, une étude approfondie des points les plus importants relatifs aux concepts ci-dessus, est proposée à titre de préambule dans le chapitre 4; cette étude différencie les aspects liés au parcours navigationnel des objets (approche des langages de programmation), de ceux considérant la sélection d'objets qui correspond plus à une approche base de données.

Dans le chapitre 7, nous présentons un certain nombre de solutions pour le contrôle par les applications Eiffel, de certains aspects concernant la gestion des objets persistants. Ces aspects concernent essentiellement:

- le partitionnement des objets persistants (Pview)

- l'intégrité des objets (transaction Eiffel),

- la sécurité (gestion des droits d'accès pour les classes et les objets persistants).

# 4

# Persistance et langages de programmation

Dans ce chapitre, nous nous attardons sur un certain nombre de points, relatifs aux critères de modélisation (cf. § 3.1), et nous présentons leur intégration dans les langages que l'on retrouve le plus souvent dans la littérature.

Nous distinguons les problèmes généraux d'intégration pouvant s'appliquer à l'accès navigationnel (comment rendre persistant un objet ou y accéder), des solutions liées à l'accès associatif (sélection et mise à jour globale d'une collection d'objets). L'accès navigationnel est le moyen naturel d'atteindre un objet dans un langage de programmation, mais l'introduction de la persistance et donc de la possibilité de gérer une grande quantité d'objets persistants, crée de nouveaux besoins comme celui de pouvoir disposer de mécanismes de sélections.

## 4.1. Philosophie de l'intégration

Les deux premiers points importants des principes de modélisation (règles N° 1 et 2, § 3.1), sont l'orthogonalité de la persistance par rapport au système des types, et la transparence de la gestion des objets persistants par rapport à celle des objets volatiles. Cela revient essentiellement à considérer les contraintes suivantes:

• le code d'une manipulation d'objets doit être le même, qu'il s'agisse d'instances persistantes ou volatiles;

• tout objet doit pouvoir devenir persistant, de la même manière et les mêmes opérateurs doivent lui être accessibles;

• la déclaration d'une propriété doit être indépendante du fait qu'elle peut ou non référencer des objets persistants;

• la liaison avec l'espace des objets persistants et les échanges entre les mémoires persistante et volatile doit être transparent à une application.

Dans les paragraphes 4.1.1 à 4.1.3, nous montrons comment ces points sont traités dans les principaux langages persistants. En particulier, dans le paragraphe 4.1.1 nous étudions l'orthogonalité de la persistance par rapport au système de types. L'établissement de la liaison entre une application et les espaces d'objets persistants est abordé dans le paragraphe 4.1.2. Dans une troisième partie (paragraphe 4.1.3), nous évoquons l'orthogonalité de la gestion des objets persistants (accès, mise à jour), par rapport au traitement des objets volatiles.

Dans le paragraphe 4.1.4 nous évoquerons l'uniformisation des traitements des objets persistants et des classes.

### 4.1.1. Orthogonalité par rapport au système de type

Parmi les langages étudiés certains d'entre eux respectent les contraintes exprimées ci-dessus; cependant on retrouve souvent un ou plusieurs choix incompatibles avec ces contraintes:

• la persistance est déclarée au niveau des classes par ajouts syntaxiques. Le traducteur peut ainsi détecter les portions de programme manipulant des objets persistants, et donc modifier le code généré dans le but d'introduire d'autres opérateurs et de préparer la structure interne des instances de classes "persistantes". Ces mots-clés peuvent en particulier être rajoutés au niveau de la classe, ou au niveau des propriétés, afin de préciser qu'elles manipulent des objets persistants; ceci permet essentiellement de mémoriser l'identificateur persistant ou de modifier les opérateurs d'accès; cette dernière technique est souvent employé pour optimiser l'accès aux objets lorsqu'il ne peuvent être que volatiles;

• la persistance est intégrée par héritage d'une classe spéciale. Il est alors possible de déclarer dans cette classe, tout ce qui est nécessaire pour gérer les objets persistants: un identificateur d'objet persistant, des nouvelles primitives d'accès, *etc*. Dans un langage typé cette approche pose le problème de la compatibilité de type. Ainsi, dans le cas ou la persistance d'une classe *PERSONNE* est obtenue par héritage (par exemple, la classe *PERSONNE_PERSISTANTE* hérite de *PERSONNE* et d'une classe spéciale), il n'est pas possible d'affecter un attribut de type *PERSONNE* à un attribut de type *PERSONNE_PERSISTANTE*, ce qui oblige le concepteur à des choix de

programmation pas toujours naturels.

• la persistance des objets est décidée lors de la création,

• l'application doit gérer explicitement l'accès aux objets persistants (voire leur mise à jour), à travers des primitives spéciales; cette solution permet d'éviter la modification de la gestion de l'accès aux objets au niveau de l'exécutif ou de la syntaxe. Dans certains langages, l'utilisation de ces opérateurs se fait implicitement par un mécanisme de surcharge des opérateurs;

• seul un sous-ensemble des types du langage peuvent devenir persistants; souvent les restrictions concernent les types générant des objets de longueur variable ou des types génériques (tableau, liste, ..).

Dans *Presto* [Giavitto 88], toute instance d'une classe *C* déclarée "persistante" est automatiquement rendue persistant dès sa création; une classe est "persistante" si elle hérite d'une autre classe (nommée *RESSOURCE*). Il est à noter que pour ne pas rendre fastidieuse la description d'une classe persistante, des facilités syntaxiques sont introduites et un préprocesseur génère la classe C++ correspondante. Dans Pclos [Paepcke 89], la persistance est intégré aussi par héritage.

Dans *Pgraphite* [Wileden 88, Tar 89], la technique utilisée est un peu similaire mais la notion de classe persistante est implanté par un paquetage Ada qui est automatiquement généré par le pré-processeur à partir de spécifications faites dans une syntaxe comportant la notion de classe. D'autre part, la sémantique est légèrement différente, puisque l'objet instance de ce type n'est pas persistant dès la création: c'est l'association d'un identificateur persistant qui le rend effectivement persistant.

Dans le langage E il faut faire précéder la déclaration d'une classe dont certaines instances peuvent être persistante, par les deux lettres **db**. Soit une classe de nom *PERSONNE* on pourra avoir l'une des deux écritures suivantes, selon que ses instances ont ou non, la possibilité d'être persistante:

```
class PERSONNE                    dbclass PERSONNE {
    ....                              ....
    }   /* class PERSONNE            }   /* class PERSONNE
```

Un objet est persistant dès sa création; un objet est persistant s'il est référencé par un attribut "persistant", c'est à dire précédé du mot clé **persistent**; le programmeur doit donc

se servir d'un attribut différent en fonction du fait qu'il veut adresser un objet persistant ou non. Certaines raisons à ces choix sont:

• l'orthogonalité par rapport à C++: on peut ne pas toujours accéder à un objet par son début (utilisation de ++ pour l'accès aux éléments suivants d'un vecteur);

• Simplification de la mise en oeuvre de la gestion des objets persistants (voir chapitres 8 et 10).

Dans *Zeitgeist,* pour rendre un objet persistant, il suffit de l'introduire dans une base de données par des primitives permettant l'insertion de l'objet, mais ces primitives ont un paramètre supplémentaire pour identifier l'objet en mémoire persistante. La présence d'un identificateur externe, et son calcul à la charge du développeur pose des problèmes en terme de réutilisabilité: l'accès à un objet dépend fortement de l'application qui l'a rendu persistant.

Dans *PS-Algol* [Atkinson 88], la persistance d'une donnée est obtenue lorsque celle-ci est introduite dans la ou les bases de données associées au programme; on notera que chaque base de données est une table particulière où chaque entrée est une racine de persistance (objet associé à un nom symbolique). La persistance d'un objet peut être explicite (insertion dans la table d'une nouvelle racine de persistance), ou implicite: attachement à un objet déjà persistant.

Dans *O++* on peut rendre persistant un objet à tout moment pendant l'exécution; il devient persistant lorsqu'on lui applique la primitive *pnew* qui est l'équivalent de *new* (langage C), mais retourne une adresse en mémoire persistante au lieu d'une adresse en mémoire volatile.

Dans *Smalltalk,* un objet devient persistant par sauvegarde d'une image d'exécution, la récupération se fait elle aussi en une seule fois, c'est typiquement une persistance de niveau 1 (voir cf. § 2.2.2). Cependant il existe des bibliothèques de composant comme BOSS [pps 90], qui permettent de rendre persistants des objets et des classes par insertion dans un réservoir d'objets (Binary object storage).

### 4.1.2. Liaison avec le monde persistant

Pour accéder aux objets persistants, une application doit donner d'une part une

représentation du monde persistant, et d'autre part établir un lien avec le monde des objets persistants afin de pouvoir atteindre un objet.

Le besoin de partage des données persistantes demande à ce que l'on n'établisse pas de lien définitif entre un espace d'objets persistants et une application, il faut donc que ce lien s'établisse à l'exécution pour une durée limitée et que l'application ait accès à des points d'entrée ou racine de persistance. D'une manière générale, la liaison entre l'application et un espace d'objets persistants peut être commandée explicitement par l'application ou gérée implicitement par l'exécutif. Nous allons étudier comment ce lien est réalisé par les principaux systèmes persistants.

Pour les langages de programmation classiques (persistance de niveau 1, paragraphe 2.2), les espaces de données persistantes sont caractérisés par des fichiers, le point d'entrée est une chaîne de caractères correspondant au nom du fichier et le lien est réalisé par des routines systèmes.

Dans le *langage E* [Richardson 89], un réservoir d'objets est associé implicitement à un module de programme, de même, on accède aux objets par des racines de persistances qui sont des variables déclarées comme étant persistantes (**persistent** *nom-type nom-variable*) . Les racines de persistance et le lien entre l'application et l'espace des objets persistants sont élaborés statiquement (phase de compilation).

Dans *PS-Algol* [Atkinson 83 & 85], un réservoir de données (matérialisé par un fichier) est nommé par une chaîne de caractère; le lien est établi à l'exécution par une routine retournant la racine de persistance à partir de laquelle il est possible (comme dans le langage E) de naviguer à travers les références. On notera que ce mécanisme se généralise pour le compilateur qui établit un lien avec le réservoir contenant les procédures. Dans *Napier 88* [Atkinson 87 a], (successeur de PS-Algol), le lien n'est plus établi à travers une chaîne de caractères mais par un nom traité comme une variable.

Dans *Alltalk* [Straw 89], et dans NICE-C++ [Mopolo-Moke 91], un espace persistant est représenté par une base de données identifiée par un nom. Le lien avec l'application est fait à l'exécution (ouverture de la base de données). Les racines de persistance sont repésentées par des clés associées à des objets. On notera que dans Nice-C++, cette clé fait partie de la description (mot-clé) de la classe auquel appartient l'objet.

Dans *OPAL*, le langage de programmation associé à Gemstone, le lien est réalisé par

des noms d'espace d'objets persistants, passés en paramètre au compilateur du langage. Les racines de persistance sont représentés par des couples (nom, objet) formant un dictionnaire de symboles. Dans *$O_2$-C* (système *$O_2$*) et COP (système Vbase), l'approche est similaire.

Remarque:

On notera qu'un autre moyen de généraliser implicitement le concept de racine de persistance à tous les objets, est d'utiliser un langage de requêtes qui permet l'accès à un objet par ses valeurs et non par un symbole.

### 4.1.3. Accès aux objets persistant

Dans cette section nous ne traitons pas les opérateurs d'accès aux racines persistantes (souvent associées à des noms symboliques), mais plutôt les opérateurs d'accès aux objets pour naviguer à partir de ces racines. Le respect des règles de transparence nécessiterait l'absence d'appel explicite à ces opérateurs mais cela n'est pas toujours le cas.

Dans *Pgraphite* [Wileden 88], l'accès aux objets est explicite (routine GetPID), qu'ils soient persistants ou volatiles; les primitives d'accès se trouvent dans un package *Ada* utilisé pour intégrer la persistance dans le langage. On notera que l'accès est presque transparent puisque les mêmes primitives peuvent (éventuellement) être appliquées que l'objet soit persistant ou non.

Dans *Pclos*, [Paepcke 89] l'accès et la mise à jour des objets persistants sont réalisés automatiquement par l'exécutif du langage. Cependant pour certaines opérations de mise à jour ne faisant pas intervenir d'affectation (changement d'un élément dans un tableau), l'application doit gérer la demande de mise à jour.

Dans *PS-Algol* [Atkinson 88], l'accès aux objets se fait incrémentalement et de manière transparente; on notera que les routines (*procédure de 1ere classe*), sont aussi stockées et chargées comme les autres données par le compilateur. C'est un bel exemple d'orthogonalité comparable à Smalltalk; cependant *PS-Algol* n'étant pas un langage à objets, il n'y a pas de notion de méta-classe.

Dans le *langage E* [Richardson 89], la déclaration d'une variable comme persistante implique que les objets qu'elle référence soient persistants; cela permet l'implémentation d'un accès par défaut d'objets, transparent à l'application.

Dans *Presto* [Giavitto 88], la classe *RESSOURCE* (cf. § 4.1.1) implante la notion de

pseudo-pointeur qui remplace les pointeurs C usuels en utilisant le mécanisme de surcharge d'opérateurs de C++. Ce mécanisme permet de cacher la gestion explicite de l'accès aux objets persistants.

En Eiffel 3.1 comme en *Smalltalk*, l'accès aux objets persistants est réalisé automatiquement, mais en fait tous les objets persistants sont chargés et mis à jour globalement. Des dérivées de *Smalltalk* [Low 89, Straw 89] introduisent des possibilités d'échanges incrémentaux, mais si dans *Alltalk* [Straw 89] l'accès aux objets est implicite, cela n'est pas le cas dans [Low 89] où l'accès et la mise à jour des objets est explicite (GetPSObject, PutPSObject, ....).

### 4.1.4. Persistance et liaison dynamique

La cohabitation entre le partage des objets par plusieurs applications (liaison au même espace d'objet persistant) et le polymorphisme pose un problème pour les systèmes de classes compilés: un objet persistant peut référencer un objet dont la classe (et donc les méthodes), ne se trouvent pas dans le code exécutable de l' application dèsirant accéder aux propriétés de l'objet.

De même un problème existe lorsque le langage gère l'extension des classes: quel ensemble d'objets considérer ? Celui correspondant seulement aux instances des classes héritières présentes dans le processus Eiffel ? Ou bien celui correspondant aux instances de toutes les classes héritières ?

Nous étudions ici comment ce problème est résolu par différents langages persistants.

Dans PS-Algol, le problème ne se pose pas vraiment en ces termes puisque ce langage n'est pas vraiment orienté objets, même si la notion de *procédure de 1ère classe* [Atkinson 85] et la notion très générale de pointeur (type *PNTR*) permet d'envisager l'implantation des techniques orientés objets [Philbrow 89]. Cette implantation de l'approche orientée objets au dessus de PS-Algol pouvant être faite différemment pour chaque application, il est difficile de donner les choix faits au sujet du polymorphisme et de l'héritage dans PS-ALGOL, d'autant plus que seul l'héritage simple est discuté.

Par contre, la liaison dynamique avec une racine de persistance nécessite un traitement polymorphique sur le nom symbolique de l'objet, c'est-à-dire faire correspondre les bonnes routines, en fonction du type de l'objet associé au nom symbolique; la liaison étant faite statiquement, aucun traitement particulier n'est à faire au cours de l'exécution.

Par ailleurs, les possiblités de sélection étant limitées à la recherche dans une table de

noms symboliques, le choix de l'ensemble sur lequel porte la sélection ne pose pas de problème.

Dans *Napier 88*, la notion d'héritage n'existe pas encore [Dearle 89], et la notion de *procédure de 1ère classe* développée dans *PS-Algol* est présente. Le polymorphisme est comme dans *PS-Algol* plutôt vu comme une combinaison entre la surcharge d'opérateurs et les types génériques.

Dans le *langage E*, l'implantation de la persistance devrait entraîner la non-existence du problème, l'objet persistant est associé à un module selon les règles de visibilité de C++; cependant, la possibilité de déclaration d'une variable persistante comme externe fait resurgir le problème. Pour le résoudre, la table des fonctions virtuelles de C++ (seule les fonctions virtuelles permettent le polymorphisme), est gérée de manière spéciale et initialisée en conséquence au début de chaque exécution [Richardson 89].

*Smalltalk* est un langage interprété, et le problème n'existe pas vraiment puisque l'interprète a accès à toutes les classes existantes et donc à leurs méthodes et leurs attributs. Il semble que ce soit pareil dans *Lispo2*.

Dans *OOPS+* [Laen 88], on peut faire la même remarque que dans *Smalltalk* sauf que le mécanisme est plus complet (mélange interprété-compilé); si la méthode n'est pas compilée, elle est interprétée à l'exécution.

# 4.2. Accès associatif dans les langages de programmation

## 4.2.1. Autres expériences d'intégration de mécanisme de sélection

Dans tout langage de requêtes ou mécanisme de sélection intégré dans un langage de programmation, il faut définir:

- le critère de sélection,
- la cible de cette sélection (un ensemble, une liste, l'extension d'une classe, ...),
- le format et la nature du résultat.

### Langage de requêtes versus mécanisme de sélection

On a vu (section 3.3.1), qu'une des grandes différences entre un langage de requêtes et un mécanisme de sélection intégré dans un langage de programmation classique est le caractère interactif du langage de requête.

Dans un langage de requêtes, c'est au moment de la définition de la sélection à effectuer qu'il faut donner les renseignements sur la cible, le critère et le résultat.

C'est ainsi, qu'un certain nombre de travaux, dans ce domaine de recherche, comme ceux de A. Heuer et M. Scholl [Heuer 91], développent l'intérêt de posséder un opérateur *project* et *extent* (ce dernier étant proche de l'opérateur *join* du modèle relationnel), et citent certains projets comme Encore [Zdonik 86], COCOON [Heuer 90] et OSCAR [Scholl 89]. De même, d'autres travaux réalisés par [Loizou 91] montrent une approche pour intégrer la projection et la jointure dans une hiérarchie de classes.

Comme chacun sait, ces opérateurs permettent de construire à la fois la cible et le résultat de la requête; cependant le fait qu'il existe une hiérarchie entre les classes (lien héritage), pose un problème pour situer le résultat par rapport à cette hiérarchie, ainsi que, parfois la profondeur de la requête.

Dans un langage de programmation, qu'il soit interprété ou compilé, la collection sur laquelle porte la requête est dèsignée ou construite avant la sélection et le format du résultat, c'est à dire le choix des attributs utiles pour l'application exécutant la requête, est défini par l'application après la sélection; on notera en particulier que le résultat d'une requête peut servir à divers moments de l'exécution, en particulier pour construire la

cible d'une requête exécutée postérieurement; cela n'est pas le cas pour un langage de requêtes puisque le résultat est éventuellement exploité par des requêtes englobantes, mais plus souvent affiché à l'écran.

### Itérateur simple versus accès associatif

L'intégration dans un langage de programmation peut se faire soit par un itérateur simple (*one-at-a-time operator*), qui en général laisse la totalité du contrôle de la sélection au programme, soit par un mécanisme de sélection qui permet de spécifier une condition (*set-at-a-time operator*). Dans ce cas, la responsabilité de la sélection est de déléguer la responsabilité de la sélection au gestionnaire d'objets, ce qui assure une plus grande efficacité. Parfois, la séparation est moins nette, et il peut y avoir une combinaison des deux: l'itérateur simple étant plus souvent réservé pour l'exécution d'une opération de modification.

L'itérateur ou le mécanisme de sélection peut être représenté par une structure syntaxique dans le langage de programmation, ou par une routine admettant comme critère de sélection, une routine ou une chaîne de caractères. Dans les deux premier cas, le critère de sélection est fixé avant la traduction du programme (critère compilé), alors que dans le second, c'est lors de l'appel que ce critère est connu du système (critère interprété).

Selon que les modèles du gestionnaire d'objet et du langage sont identiques ou non, et que le traducteur du langage intégre ou pas un mécanisme de traduction vers le langage de requêtes du gestionnaire, il sera plus ou moins facile de rendre transparent le passage de la requête entre l'application et le gestionnaire, et d'optimiser son exécution.

Pour une bonne intégration, il faut autant que possible, mettre en oeuvre un mécanisme qui permette de rendre transparent la différence entre le modèle du langage et celui du gestionnaire d'objets. Ceci met en évidence plusieurs problèmes:

- le choix d'une traduction statique ou dynamique du critère de sélection vers le modèle du gestionnaire (voir chapitre 11),

- la mise en oeuvre des contrôles, syntaxiques et sémantiques, qui doit elle aussi pouvoir s'envisager de manière statique ou dynamique, incrémentale ou non.

*Remarque:*

Pour interfacer un système de gestion de base de données relationnelle la distance conceptuelle entre les modèles fait que la plupart du temps le couplage est faible. On a ainsi souvent recours à l'utilisation d'une routine avec une chaîne de caractères en paramètre (selon la syntaxe du SGBD).

### Communication entre l'exécutif du langage et le gestionnaire d'objets

Le mode de communication du critère de sélection dépend de l'indépendance du gestionnaire d'objets par rapport au langage de programmation.

Si le gestionnaire est un SGBD existant (présence de deux modèles) dont la réalisation ne s'est pas déroulée conjointement avec le langage, la transmission du critère se fera le plus souvent par l'intermédiaire d'une chaîne de caractères, qui sera interprétée par le langage de requête; il y a alors opposition entre la compilation du langage de programmation et l'interprétation du critère de sélection par le SGBD.

Par contre, si le gestionnaire est un serveur d'objets construit pour le langage de programmation, et donc que les requêtes peuvent être analysées en même temps que les autres structures du programme, alors une transmission plus efficace n'utilisant pas des chaînes de caractères pourra être mise en oeuvre (la requête est alors compilée).

### Le système $O_2$

Pour le système $O_2$ [$O_2$-Technology 91a], le choix s'est plutôt porté vers une intégration syntaxique plutôt que par composant. En effet, si on constate l'existence implicite de la hiérarchie montrée par la figure 4.1, pour modéliser les structures parcourables, celle-ci n'est pas implantée sous forme de composant, mais par des mot-clés, et leur parcours se fait pas des structures de contrôle ayant une syntaxe particulière:

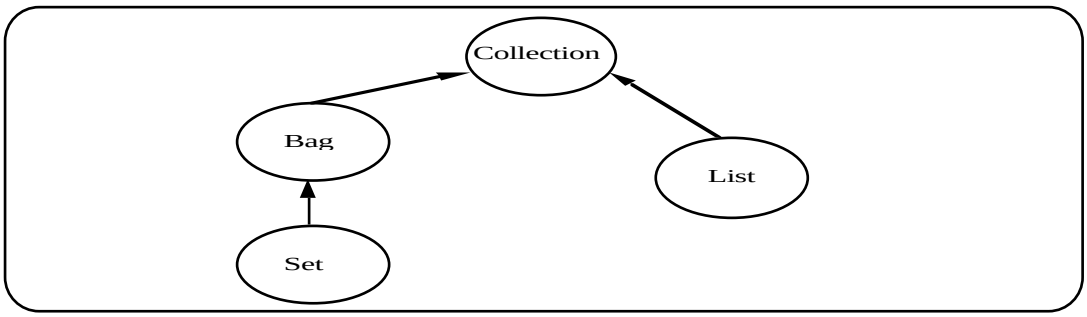


Figure 4.1: hiérarchie des collections d'objets en $O_2$

D'autre part, on utilise ici la transmission des données par des chaînes lorsque le

langage de programmation est indépendant de *O2* comme c'est le cas pour *C++*. On dispose alors de modèles de classe *C++* représentant des classes génériques correspondant à des ensembles, des listes ou des itérateurs, et permettant d'effectuer des requêtes sur des collections nommées (contenant que des objets persistants), par l'intermédiaire de plusieurs primitives. On trouve en particulier l'opération *apply* (fig. 4.2), qui permet d'appliquer une méthode à une collection, en procédant par filtrage avec un critère de sélection.

Soit une méthode *afficher_info* admettant pour paramètre une instance de la classe *PERSONNE*, soit *this*, le mot clé permettant de pointer sur l'instance courante, on peut écrire:

```
O2_SET_PERSONNE personnes                          -- ensemble de personnes
personnes.apply (afficher_info, "this.age > 21")
```

Figure 4.2: mécanisme d'itération à partir de C++

De même, on dispose de classes itérateurs dont le nom est: *nom-classe*_ITERATOR. Le constructeur de cette classe pouvant accepter en paramètre, un prédicat (expression de *O2-Query*), afin de filtrer les objets. Dans ces classes (simulation des classes génériques), on dispose de plusieurs opérations:

- une fonction *next* qui retourne le prochain élément de la collection à lire,
- une fonction *end* qui teste si tous les éléments de la collection ont été lus,
- une procédure *close* qui indique la fin de l'itération

Soit la classe *PERSONNE*_iterator (fig. 4.3), un itérateur sur des personnes, on appliquera une suite d'instruction I1..In sur l'ensemble des personnes agées de plus de 21 ans en écrivant:

```
PERSONNE p;
j = PERSONNE_iterator ("this.age > 21");
while (!j.end) {
    p = j.next;
    ..... Suite d'instructions I1...In .....
}
```

Figure 4.3: Itérateur pour le parcours des collections d'objets *O2*, à partir de *C++*

Les mêmes possibilités sont offertes à partir de C, sauf que les primitives décrites ci-dessus, ne sont pas incluses dans des classes et admettent en paramètre la collection sur laquelle faire la sélection.

Dans le langage de programmation *O2-C*, de *O2*, on peut utiliser le langage de requêtes

*O2-Query* (interactif), par l'intermédiaire de la méthode *o2query*, accessible à tout module écrit en *O2-C* (fig. 4.4). Ainsi, toute requête exprimable par *O2-Query*, comme: "*rechercher dans la liste l contenant des personnes, celles qui satisfont un critère q"* faisant intervenir les variables q1, .., qn, déjà déclarées et initialisées:

```
o2 list (PERSONNE) l;
o2query (l, " select x
             from x in PERSONNE
             where q (q1, ... qn)",
        q1, ...,qn)
```

Figure 4.4: Accès à *O2-Query* (langage de requêtes de *O2*), à partir de *O2-C*

De même, à partir de *O2-C,* il est possible d'effectuer des sélections sur une collection à travers une structure de contrôle itérative (voir fig. 4.5).

```
o2 list (PERSONNE) l;        /* pourrait être de type o2set (PERSON)
o2 PERSONNE q
for (q in l where q (q1, ... qn))  { instructions}
```

Figure 4.5: Itérateur sur des collections d'objets dans *O2-C*

Ici, le critère *q* est toujours fonction des variables *q1, .., qn*, déjà déclarées et initialisées, mais l'expression sélective est une instruction comme les autres et n'est donc pas placée dans une chaîne de caractères.

On notera que dans *O2-Query* il est aussi possible d'appliquer la notion de quantificateur (universel et existentiel) sur une collection. Soit *personnes*, un ensemble *O2*. La figure 4.6 montre un exemple d'utilisation des quantificateurs.

```
forall x in personnes: x.age < 21      -- retourne True si toutes les personnes
                                       -- ont moins de 21 ans

exists x in personnes: x.age < 21      -- retourne True si au moins une
                                       -- personne a moins de 21 ans
```

Figure 4.6: Les quantificateurs dans *O2-Query*

Ces facilités ne sont pas prédéfinies dans *O2-C* ou les interfaces C ou C++, puisque l'on peut toujours les recréer. Seule la méthode *o2-query* permet de les exprimer puisque la chaîne de caractères définissant le critère de sélection est une requête de *O2-Query*.

**Statice**

Dans STATICE [Weinreb 91], l'intégration de la notion de requête se fait par l'intermédiaire d'une fonction *for-each*, dont la syntaxe est celle de common-lisp. avec cette fonction, il est possible de faire des sélections combinées éventuellement avec des mises à jour. Comme le montre l'exemple ci-dessous, le niveau d'intégration est élevé, et ceci pour deux raisons, conformes à ce que l'on a dit plus haut:

- le critère de sélection est défini par des instructions du langage, sans usage de chaînes de caractères,
- le gestionnaire d'objet est implémenté en Common-lisp et donc adapté à lui.

Cependant, il n'est pas possible, en utilisant cette primitive de prendre en compte l'aspect hiérarchique des types, puisque la primitive for-each n'est intégrée dans aucune structure (voir fig. 4.7). Pour que la requête s'effectue sur un sous-ensemble des instances de la classe *PERSONNE*, par exemple celui donné par la propriété *enfants* de la classe *PERSONNE*, *(p PERSONNE)*, serait remplacé par *(p PERSONNE-enfants)*;

```
Sélection sur les instances de la classe PERSONNE:

   (   for-each (   (p PERSONNE)   (a ADRESSE)
                        .... séquence de fonctions lisp .....
                    :where   (and  (eql (PERSONNE-adresse p) a)
                                   (eql (PERSONNE-nom p) laurie)
                              )
               )
   )
```

Figure 4.7: itérateur sur des structures Lisp dans Statice

Note: Dans la description de la requête, les '-' remplace le '.' plus communément utilisé dans SQL, ou en Eiffel, pour identifier la provenance des propriétés.

### Vbase et Ontos

Comme dans **O2**, *Vbase* permet de réaliser des requêtes suivant deux approches: interface utilisateur (de type SQL) [Harris 91], et application [Andrews 91 a]. Nous étudierons seulement l'aspect intéressant les applications. Les structures parcourables (ensemble, ou autre agrégation de type), sont implantées par des composants pouvant être étendus par héritage; ces composants sont reconnus par le système et traités spécialement.

Il est possible de demander l'exécution d'une requête de type SQL (voir fig. 4.8), à travers une classe *QueryIterator*. La solution utilisée est un mélange de celles de l'interface C++ et de la routine *o2_query* de *O2*:

```
QueryIterator aQuery = ( "select PERSONNE.nom from PERSONNE");
while (aQuery.moreData ())              -- moreData est équivalent à end dans O2
   {
      aQuery.yieldRow (nom);            -- yieldRow est presque équivalent à next de
O2
      .... suite d'instructions....
   }
```

Figure 4.8: Encapsulation de O-SQL dans Ontos et Vbase

Mais plus intéressante est la notion d'itérateur (voir fig. 4.9). Un itérateur est une méthode particulière qui se différencie des autres essentiellement par un mot clé *iterator*. Cette notion s'inspire des itérateurs d'alphard [Shaw 81] et CLU [Liskov 77 & 81] et combine deux approches:

- syntaxique: adjonction d'une structure itérative *(for-in-where* de $O_2$),
- par composant: méthode modélisant le concept d'itération (*apply* de $O_2$).

Un itérateur de Vbase est donc une méthode presque comme les autres (sauf qu'elle ne peut retourner aucun résultat), pouvant faire partie de la définition de toute classe. A ce titre, un itérateur sera à la fois décrit en TDL (Type définition language) pour les spécifications et en COP (C object processor) pour l'implémentation des méthodes.

```
Spécification:
        iterator apply_afficher_info (les_personnes: set [PERSONNE] )
              yields (PERSONNE)
              method (apply_afficher_info_method);

Implémentation:
        iterator apply_afficher_info_method (les_personnes)
            obj set [obj PERSONNE] les_personnes
        {
            PERSONNE$display (les_personnes)
        }
```

Figure 4.9: Les itérateurs de Vbase: Une intégration syntaxique

La méthode *apply_afficher_info* ne retourne pas de valeur, mais exécute une séquence d'instructions sur les objets sélectionnés. Elle pourrait être décrite dans toute structure itérable; typiquement, ces structures sont des tableaux, des listes ou des ensembles, ou bien elles en dérivent.

Ontos [Andrews 91 b], reprend un certain nombre de facilités offertes par Vbase, mais correspond à une extension de C++ à la persistance, au lieu de C.

## Objectstore

Pour le système Objectstore [Lamb 91], l'intégration des mécanismes de sélection dans objectstore-C++ (son langage de programmation associé), se fait par deux approches syntaxiques:

- une structure itérative externe aux classes de type COLLECTION (voir fig. 4.10), et
- une approche basée sur l'opérateur *[::]* intégrées dans ces classes (voir fig. 4.11).

Parmi les collections possibles on trouve essentiellement, les tableaux, tables relationnelles, listes, ensembles et sacs.

Ainsi, dans Objectstore-C++, pour réaliser des mises à jour des éléments d'une collection, on peut utiliser, la construction *foreach*:

Pour augmenter de 10% le montant de la tirelire des enfants de la personne représentée par *p*, (*enfants* est une propriété de PERSONNE de type *oS_set (PERSONNE *)*: on écrira:

```
PERSONNE *p;
foreach (PERSONNE * x, p->enfants)
        x->tirelire *= 1.1;          -- x est la variable libre de la
requête
```

Figure 4.10: l'itérateur dans Objectstore-C++

Cet itérateur qui est une structure du langage de programmation de Objectstore permet une intégration forte et donc permet d'éviter l'utilisation des chaînes de caractères pour transmettre un critère de sélection; celui-ci pourra faire partie du corps de la boucle, et donc, de nouvelles collections pourront être créées.

Une autre méthode plus souple pour réaliser une sélection sur une collection (mode déclaratif), est d'utiliser l'opérateur *[::]*.

La figure 4.11 montre comment sélectionner tous les enfants ayant plus de 1 an d'une personne identifiée par *p* (*enfants* est une propriété de PERSONNE de type *os_Set (PERSONNE*)*.

```
PERSONNE *p;
......
os_Set (PERSONNE *) Resultat = p->enfants [: age > 1:]
```

Figure 4.11: Utilisation d'un itérateur dans Objectstore-C++

Il est aussi possible de faire des requêtes à partir des langages C et C++, en utilisant l'opérateur *[::]* mais placé dans une chaîne de caractères. La figure 4.12 décrit cette approche pour l'exemple de la figure 4.11.

```
PERSONNE *p;
oS_set (PERSONNE *) res;
res = p->enfants->query ('PERSONNE *', "[:age > 1 an]")
```

Figure 4.12: Encapsulation de l'itérateur de Objectstore-C++ dans C++

On notera que le premier argument de la routine Query représente le type des éléments de la collection, le deuxième est une chaîne de caractères exprimant la requête en Objectstore-C++.

### Iris et Pclos

Le système IRIS, a une approche plutôt fonctionnelle, dans la mise en oeuvre des expressions pointées; ainsi, pour l'exemple traîtant des personnes, avec *p* une propriété de type PERSONNE, on écrira: *ville (adresse (p))* au lieu de *p.adresse.ville*.

Mise à part un SQL objet (OSQL) [Fishman 89, Fishman 90], Iris propose des facilités pour exécuter des requêtes à partir de LISP et C, à travers des interfaces implémentée par une bibliothèque de fonctions qui permettent la modélisation d'itérateurs à travers des routines (*open-scan*, *get-next*, *close-scan*), comme pour $O_2$ ou *Vbase*. Cependant une limitation importante pour un SGBD se voulant orienté objets, bien que basé sur une base de données relationnelle, est l'impossibilité de définir des prédicats contenant des fonctions. Tous les prédicats doivent être de la forme:

```
(column1 eq "laurie") and ( (column4 gt 3) or (column4 neq 2) )
```

On notera aussi l'existence d'un langage persistant, *PCLOS* [Paepcke 88, Paepcke 89], correspondant à une extension du langage *CLOS* à la persistance, basée sur Iris.

La syntaxe utilisée pour décrire une requête est très influencée par le langage d'implémentation (Common lisp), ce qui a pour conséquence d'avoir des itérateurs implémentés par des routines mais ne nécessitant pas l'emploi de chaînes de caractères. Une requête sera définie par:

- le type de protocole utilisé (Pclos est destiné à être utilisé indifféremment sur plusieurs systèmes différents, et il faut donc spécifier le protocole de conversion),
- le type des objets concernés,
- un critère de sélection.

La figure 4.13 montre comment rechercher l'ensemble des personnes s'appelant *Laurie* et agées de plus de *21 ans.*

```
(find-all  iris-protector      -- protocole de conversion
           'PERSONNE           -- cible de la requête
           '(and
                 (= nom "laurie")
                 (> age 21)   )   )
```

Figure 4.13: La notion d'itérateur dans Pclos

## Encore

Le système ENCORE [Zdonik 88], est influencé par Smalltalk [Goldberg 89] et Simula; le concept de classe qu'il supporte correspond à une spécialisation du type *collection* et permet de traiter l'extension d'une classe. Il supporte aussi le concept de classe générique (notamment la classe *SET*). Les collections sont donc implantées par des composants et contiennent:

- des opérateurs ensemblistes: la différence, l'union, l'intersection;
- des opérateurs dérivant de l'algèbre relationelle: selection, projection, jointure.

Les opérateurs qui dérivent de l'algèbre relationnelle et implémentent le mécanisme de sélection, sont définis par les règles [Shaw 89] présentées dans la figure 4.14.

Soit:
S et R des collections d'objets,
p une forme de la logique du premier ordre,
$a_i$ les noms d'attribut de type $f_i$,
$f_i$, les noms de fonction retournant des objets de type $f_i$,

alors:
Select (S, p) = {s | (s in S) et p (s)}
Image (S, f) = {f (s) | s in S}
Project (S, < ($a_1$, $f_1$), ....., ($a_n$, $f_n$)>) = {< $a_1$:$f_1$(s), ....,$a_n$:$f_n$(s) > | s in S}
Ojoin (S, R, A, B, p) = {< A: s, B: r > | s in S et r in R et p (s, r)}

Figure 4.14: Modélisation du mécanisme de sélection dans Encore

La notion de prédicat (classe *PREDICATE*) est celle communément utilisée dans les langages de requêtes.

## Orion

Dans la littérature qui nous a été accessible, nous n'avons pas trouvé d'informations

sur la présence d'un langage de programmation propre à ORION qui serait non interactif, et intégrerait des facilités pour exécuter des requêtes.

### Exodus et le langage E

E est le langage de programmation [Richardson 89], qui est intégré dans le système EXODUS [Richardson 87, Carey 90]; le langage E fournit le concept de collection, décrit par une classe générique, l'instanciation se fait comme suit (cf. § 2.2.2):

```
dbclass nom-classe-COLLECTION: COLLECTION [nom-classe];
```

On notera que la classe *nom-classe*_COLLECTION correspondant à une collection de *nom-classe* (par exemple *PERSONNE*) peut éventuellement contenir des instances d'une classe descendante de *nom-classe.*

Comme pour Vbase, le mécanisme de sélection du langage E s'inspire du concept d'itérateur de CLU [Liskov 77]. Cette notion d'itérateur, s'intègre au niveau des méthodes d'une classe, par ajout de mot-clés; donc toute classe peut décrire des méthodes intégrant un accès associatif vers les éléments d'une structure parcourable (liste, pile, ensemble, collection). On notera qu'un itérateur peut être associé à toute classe, mais que son utilisation (description de la méthode initialisant l'itération), est propre à la classe où il est décrit.

Dans un programme écrit en E, on doit spécifier l'utilisation des itérateurs par des mot-clés, à la fois dans la méthode décrivant la boucle d'itération, et dans les méthodes qui l'utilise. La description de la boucle d'itération se fait dans une routine qualifiée par le mot clé *iterator*, et renvoyant un objet dont le type correspond à celui du paramètre générique de la collection (voir fig. 4.15).

Ainsi on cherche par exemple à écrire la routine *item* de la classe COLLECTION, permettant d'obtenir tour à tour tous les élements de la collection. On admet que le type générique de la collection est dèsigné par *elt_type* et que celle-ci contient les caractéristiques suivantes:

- *get_first*: le premier élément de la collection,
- *get_next*: le prochain élément de la collection,
- *get_item* l'élément courant.

on écrira:

```
iterator elt_type * COLLECTION: for_all ()  {
```

```
                    for (elt_type * p = get_first(); p != NULL; p = get_next())
                        yield & (get_item())
                }
```

Figure 4.15: Description d'un itérateur dans langage E

Tout composant du programme, y compris la classe contenant la méthode d'itération (ici *for_all*), peut utiliser cet itérateur (voir fig. 4.15).

Pour ce faire, on utilise une structure *iterate* permettant d'utiliser le moteur de l'itération (dans notre exemple: *for_all*), et de décrire l'action à effectuer sur tous les éléments de la collection. Supposons que dans la classe *PERSONNE* on trouve une méthode *inc_age* augmentant de 1 l'âge d'une personne.

```
dbclass PERSONNE_COLLECTION: COLLECTION [PERSONNE];
PERSONNE_COLLECTION * c;
global_inc ()    {
        iterate (PERSONNE * p = c->for_all ())
                p.inc_age;
    }
```

Figure 4.15: Utilisation des itérateurs du langage E

On notera enfin qu'un itérateur peut utiliser un ou plusieurs autres itérateurs et décrire des requêtes de sélection sur des structures parcourables, avec des progressions asynchrones. C'est donc une généralisation de la structure *for* pour la manipulation des itérateurs de CLU [Richardson 87] (voir fig. 4.17). En effet dans CLU on ne peut réaliser que des itérations imbriquées (a), alors que dans le langage E, les parcours peuvent se faire en parallèle (b). On suppose que *iter_i()* et *iter_j()* sont deux itérateurs déclarés suivant le modèle ci-dessus.

```
(a)
    for i in iter_i () do
        for j in iter_j () do
            < suite d'instructions >
        end;
        < suite d'instructions >
    end;
```

```
(b)
    iterate (i_type i = iter_i (); i_type j =
iter_j () {
            < suite d'instruction >
            if (<condition>) next i; else next j;
            }
```

Figure 4.17: comparaison entre les itérateurs des langages CLU et E

Dans (b), le nombre d'itérateurs possibles est à priori non limité et les structures conditionnelles, ainsi que l'ordre d'utilisation du mot-clé *next*, est laissé au choix de l'application.

### Trellis / Owl

Dans l'extension de Trellis / Owl à la persistance [O'Brien 86, O'Brien 88], il est possible de construire des itérateurs pour parcourir des structures (set, ..). Un itérateur (voir fig. 4.18), permet de parcourir les collections d'objets; ils seront utilisés pour construire des opérations de sélection comme *select*; l'approche est similaire à celle utilisée par le langage E [Richardson 89] (voir fig. 4.15 et 4.16).

Ainsi, si on veut implanter l'itérateur *elements* pour un ensemble, on pourra, par exemple, le décrire dans une classe ITERATION. Dans la figure 4.18, l'itération porte sur une collection de personnes.

```
operation elements (e: SET [PERSONNE]) -- e est la cible de l'itération
        yields (PERSONNE)
```

Figure 4.18: description d'un itérateur en Trellis/Owl

Pour utiliser cet itérateur il faut écrire dans une opération définie dans une classe utilisatrice, la boucle *for* de la figure 4.19.

```
var c: SET [PERSONNE];
for p: PERSONNE in elements (c) do
        <Suite d'instructions>
end for
```

Figure 4.19: utilisation d'un itérateur en Trellis/Owl

On trouve aussi le concept de collection (*db_collection*) qui est voisin de celui d'ensemble, mais on lui associe un certain nombre d'itérateurs prédéfinis comme *select*, qui pourra être utilisé de la même manière que l'itérateur *elements*, vu ci-dessus (voir fig. 4.20).

```
operation select (me, p: DB_PREDICATE)      -- ou me représente la collection courante
        yields (elt_type)
```

Figure 4.20: un mécanisme de sélection à partir de prédicats dans Trellis/Owl

Le prédicat *p* ne permet pas de contenir des objets non persistants (c'est à dire des objets propres à une application); un contrôle au moment de la compilation décidera si un prédicat est possible ou pas. Un exemple de prédicat sur le type PERSONNE est décrit dans la figure 4.21.

```
var p: DB_PREDICATE:=
                                    and (equal (
                                            attribute ("nom"),
                                            constant ("laurie")    ),
                                        greater_than (
                                                attribute ("age"),
                                                constant (21)    )   )
```

Figure 4.21: les prédicats dans Trellis / Owl

## Smalltalk-80 et alltalk

Dans Smalltalk-80 [Goldberg 83, Masini 89], les structures de type collection sont implantées par des composants (voir fig. 4.22)

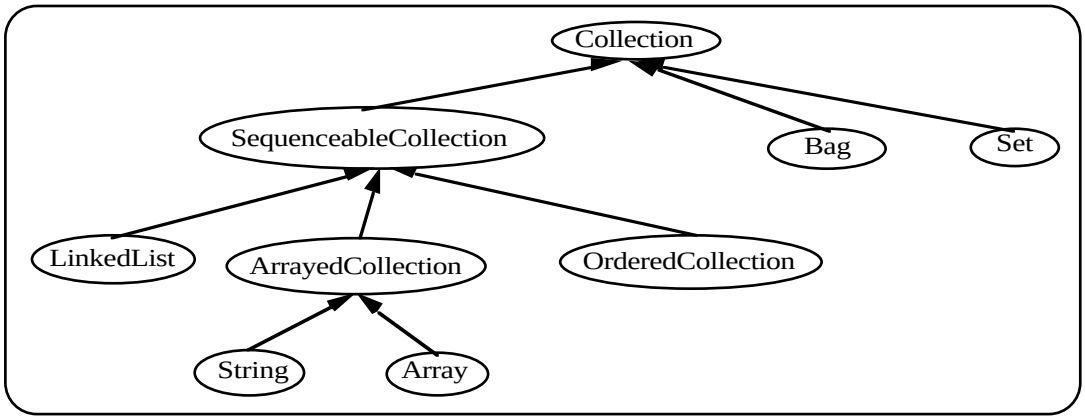


Figure 4.22: La hiérarchie des collections dans Smalltalk

Cette classification permet de différencier les structures à taille fixe (*array*, ...), des structures à taille variable (*Set, Bag*, ...), les structure ordonnées (*LinkedList*), des structures non ordonnées (*Set, Bag*, ...), les structures admettant plusieurs exemplaires d'un même élément (*LinkedList*, *Bag*, ...), d'autres, exigeant l'unicité (*Set*, ...).

Les mécanismes de sélections sont introduits par des routines dans les composants héritant de la classe *COLLECTION*. Ainsi si toutes les collections partagent les mêmes spécifications, chaque structure a son propre mécanisme de sélection.

Les opérations qui sont définies sur les collections permettent à la fois de manipuler globalement la collection, d'accéder individuellement aux objets et de réaliser des conversions (ex: un *bag* converti en *set*). Les principales opérations invoquant l'ensemble des objets d'une collection sont décrites ci-dessous:

• *do: [:e | ... suite d'instructions ... ]* permet d'appliquer la suite d'instructions à chaque élément *e* de la collection recevant le message, et ne retourne pas de résultat,

• *select: [:e | expression booléenne ]* sélectionne les objets de la collection courante vérifiant l'expression booléenne et les retourne dans une collection de même type,

• *reject: [:e | expression booléenne ]* sélectionne les objets de la collection courante ne vérifiant pas l'expression booléenne et les retourne dans une collection de même type,

• *collect: [e: | propriétés de l'élément]* retourne une collection d'éléments contenant comme caractéristique les seules propriétés mentionnées; cet opérateur représente un peu le concept de projection du modèle relationnel,

• *detect: [:e | expression booléenne ]* retourne le premier élément de la collection courante vérifiant l'expression booléenne, et s'il n'y en a pas rapporte une erreur,

• *inject: une_valeur into[:$e_1$ ...etc...:$e_n$ | suite d'instructions]* met à jour e1 à partir de combinaison des $e_i$ exprimée par la suite d'instructions.

Une expression booléenne (utile pour définir le critère de sélection), ne peut contenir qu'une seule variable libre, par contre elle accepte les principaux opérateurs de comparaison (égalité, inégalité) pouvant porter sur l'identité des objets, et les opérateurs booléeens (et, ou, not). Les opérandes d'une expression booléenne peuvent être des expressions pointées, des littéraux ou des attributs, mais pas des messages, ce qui interdit l'emboîte ment des requêtes.

Alltalk [Straw 89, Mellender 89] est une extension de Smalltalk pour intégrer la persistance. Les facilités pour réaliser des requêtes sont basées sur le modèle de Orion.

## Gemstone et OPAL

Le langage OPAL est le langage de programmation du SGBD Gemstone [Maier 86, Servio 89]. Celui-ci est largement influencé par le langage Smalltalk [Goldberg 83] dont il est une extension, comme le montre la hiérarchie de classe décrite dans la figure 4.23.

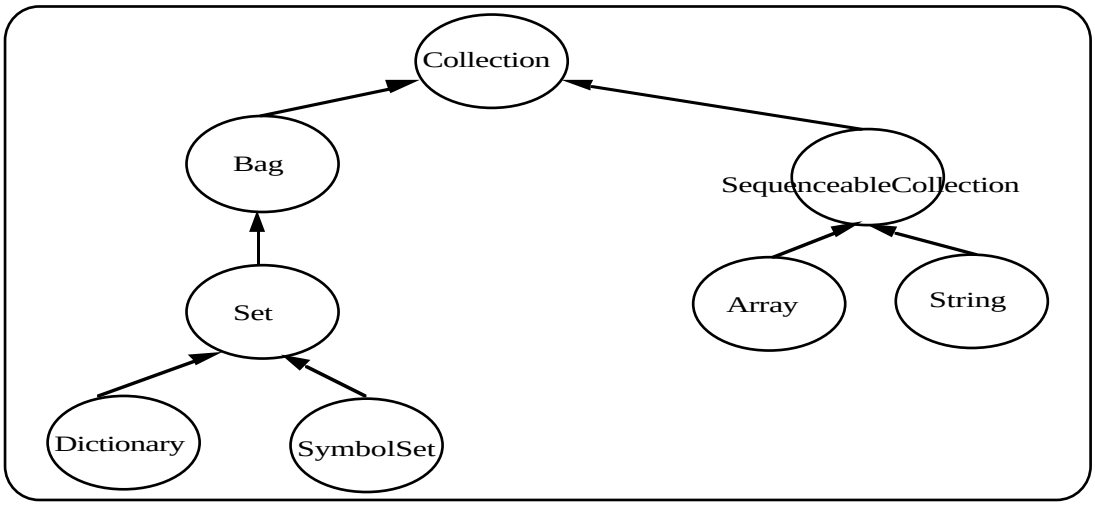


Figure 4.23: la hiérarchie des collections dans Gemstone

Comme dans *Smalltalk*, les primitives implantant le mécanisme de sélection sont intégrées dans les différents types de collection. On peut dailleurs retrouver les principales primitives implantées en *Smalltalk*: *detect, reject, select*.

On notera enfin que lorsqu'une application veut utiliser des structures d'index pour augmenter l'efficacité des accès, elle devra utiliser les symboles { } au lieu de [ ]. Ainsi pour l'opération *select* on écrira:

| *select: {e | expression booléenne }* versus *select: [:e | expression booléenne ]* |
|---|

## ODE et O++

Dans le langage O++ de ODE [Agrawal 89 a, Agrawal 89 b], qui est une extension de C++, on relève plusieurs structures permettant d'effectuer des requêtes.

L'itérateur *for* permet de faire des requêtes sur l'extension d'une classe (*cluster*) ou sur un sous-ensemble de ses instances (*derived cluster*).

Soit *x1, ...., xn* les variables libres applicables respectivement sur les éléments de $C_1$, ...., $C_n$, où $C_i$ est un *Cluster* ou un *derived Cluster*, on montre, dans la figure 3.24, un exemple d'itérateur.

| *for* x1 *in* C1, ...., xn *in* Cn |
|---|

```
suchthat    expression-booléene (x1, ..., xn)
by          expression (x1, ...., xn)
< suite d'instructions>
```

Figure 4.24: les itérateurs dans O++

La suite d'instructions sera exécutée pour tout objet vérifiant l'expression booléenne précédée par *suchthat*, dans l'ordre prescrit par l'expression suivant le mot-clé *by*. On notera que les clauses *suchthat* et *by* sont facultatives, et qu'il est possible d'écrire des itérateurs *for* imbriqués.

## PS-Algol et Napier 88

Un des pionniers des langages persistants fut PS-Algol [Atkinson 83, Atkinson 89], auquel succéda Napier 88. [Morrison 89, Dearle 89].

Dans Napier, on a introduit un itérateur pouvant s'appliquer aux structures de type collection (tableau, index, ensemble), et sur des environnements. Un environnement est un ensemble d'itentificateurs de variable, de constante ou de routine; chacun des identificateurs a un type et il est associé à une valeur.

PS-Algol, comme Napier 88 ne sont pas des langages à objets. Il n'y a pas de hiérarchie entre les types (pas de lien d'héritage); donc la collection sur lequel la routine doit s'exécuter doit être passé en paramètre des opérations.

Cependant, un type peut contenir des routines et la liaison entre le nom d'un identificateur et son contenu ou son corps est réalisé dynamiquement (polymorphisme au sens de surcharge). On notera qu'il est possible, grâce à la notion d'environnement et au polymorphisme, de gérer automatiquement, la notion d'extension de type.

Dans la figure 4.25 on décrit la syntaxe d'un itérateur en Napier 88; on montrera à l'aide d'un exemple comment l'utiliser (voir fig. 4.26)

```
for each [t:type] aName: name [t] -> aValue: t in <collection ou environnement> do
    begin  ! si plusieurs instructions
    .....
    end
```

Figure 4.25: Un itérateur dans Napier 88

La forme décrite ci-dessus n'est utile dans son ensemble que pour des cas complexes; par exemple supposons l'existence d'un environnement contenant des instances de plusieurs types.

Le type est représenté par la variable *t*, et *aName* (de type*name [t]*) représente l'identificateur de la valeur courante retournée par *aValue*. *aName* retourne l'identificateur à chaque itération et agit comme un index (l'ordre d'accès est lexicographique). Cela a l'avantage de permettre l'exécution polymorphique des routines se trouvant dans le corps *begin ... end*, et d'utiliser; selon les besoins, l'identificateur *aName*, le type *t* de la valeur, ou la valeur elle-même *aValue*.

L'exemple suivant permet de mettre en évidence une utilisation simplifiée. Supposons l'existence d'instances du type *PERSONNE*, regroupées dans un ensemble *personnes*, et une routine *afficher_info* de la classe *PERSONNE*:

```
for each -> x in personnes do
        afficher_info (x)
```

Figure 4.26: exemple simple d'itération avec Napier 88

L'itérateur ci-dessus permet d'appliquer la routine *afficher_info* sur toutes les éléments des personnes contenues dans *personnes*.

## OZ+

OZ+ [Weiser 89], fournit un certain nombre de types permettant la gestion de structures complexes; il s'agit des listes, des ensembles, des tableaux et des arbres, on peut aussi définir des dérivés de ces types. Les mécanismes de sélection sont basés sur une structure itérative *for* implanté par une opération (appelée *rule*); voici les spécifications de cette opération sur un exemple. Supposons que l'on dispose d'une opération *afficher-info* dans une classe *PERSONNE*, la figure 4.27 montre comment afficher les informations d'un ensemble de personnes:

```
.....
personnes: set (PERSONNE)
for (personnes; afficher-info)
.....
```

Figure 4.27: notion d'itérateur dans OZ+

## Galileo

Galileo [Albano 90, Albano 89, Atkinson 87], fournit des types permettant de gérer des collections d'objets de même type; il y a principalement les séquences (*seq*) et les ensembles d'objets (*setof*).

Les principales structures syntaxiques permettant de réaliser une sélection sont: *all* et *for* (fig. 4.28).

```
all x in <collection d'objets> with q (x)
for x in <collection d'objets> with q(x) do p(x)
```

Figure 4.28: itérateurs dans Galileo

Dans cette définition, *q(x)* représente une condition sur des caractéristiques de *x*, et *p(x)* représente une action sur des caractéristiques de *x*. *All* permet de retourner tous les éléments qui satisfont le critère *q*, tandis que *for* permet d'effectuer une action *p* sur tous les éléments de la collection d'objets qui satisfont *q*.

## 4.2.2. Solutions pour résoudre les principaux problèmes importants

### Extension de classe

Certains langages ne gèrent pas les extensions de classe: c'est le cas du langage Eiffel; mais il est cependant très facile quoique fastidieux, de rajouter cette possibilité explicitement dans chaque classe (voir fig. 4.29).

```
class PERSONNE
Creation
      create
feature
  personnes: SET [PERSONNE] is        -- Résultat partagé par toutes les instances
      once
          Result.create
      end; -- personnes

  create is
      do
          personnes.put (Current)     -- Current représente l'objet courant
      end -- create

end -- class PERSONNE
```

Figure 4.29: Gestion des instances volatiles d'une classe en Eiffel

Dans le système *O2* [$O_2$-Tecnology 91a] l'approche est identique à celle d'Eiffel (voir fig. 4.30). Il suffit de déclarer une constante nommée, de type *set* pour chaque classe, de l'initialiser à la création de la classe, et de l'alimenter à chaque création d'une nouvelle instance (l'objet devient alors automatiquement persistant).

```
Déclaration dans une méthode ou un programme:
        constant name: Personnes: set (PERSONNE)
Dans une méthode de création de la classe personne:
        personnes += set (self)          -- self représente l'objet courant
```

Figure 4.30: Gestion des instances d'une classe en $O_2$

De même, Gemstone [Bretl 89], Exodus [Carey 88, Carey 90], Vbase [Andrews 91 a], Ontos [Andrews 91 b], ObjectStore [Lamb 91] ou OZ+ [Weiser 89], préfèrent laisser cette gestion aux soins des applications, à travers des structures parcourables comme les listes ou les ensembles.

Dans Trellis/Owl [O'Brien 86], l'approche est nuancée. L'extension d'un type (appelée *collection*), est gérée mais, indépendamment de la hiérarchie de type. Une collection sera une classe générique contenant des primitives comme:

- le nom de la collection,
- le type des objets et ceux entrant dans sa composition,
- les collections associées aux classes parentes (super-collections).

C'est en fonction de ces paramètres que seront réalisées les requêtes. Les arguments avancés pour une telle approche sont essentiellement:

- ne pas utiliser l'héritage tantôt pour la description des types, tantôt pour l'ensemble des instances,

- l'appartenance d'un objet à une collection peut être dynamique: elle dépend de critères propres à une application, alors que l'appartenance d'un objet à un type est figée au moment de la création de l'objet.

D'autres ont choisi de gérer l'extension des classes mais aussi de permettre de réaliser des requêtes sur des collections plus restreintes, c'est le cas de Statice [Weinberg 91] et Orion [Kim 91]

Le langage Smalltalk [Goldberg 83 & 89], permet l'accès aux instances d'une classe à travers la primitive *allInstances* (l'ensemble des instances est calculé).

Napier 88 offre à travers les espaces de noms (*name*) et les environnements (*env*), une possibilité de gérer automatiquement l'extension d'une classe [Atkinson 87 a], mais il est aussi possible de réaliser des itérations sur des collections d'objets.

Dans O++ [Agrawal 89 a], l'approche est assez similaire; on a choisi de gérer l'extension de type, mais celle-ci peut être morcelée. A chaque type est associé un *Cluster*, et tout instance d'un type est insérée implicitement dans le *Cluster* correspondan,t lors de sa création. Cependant plusieurs Clusters peuvent être associés à la même classe (le choix du cluster est fait implicitement ou explicitement lors de la création de l'objet).

Selon l'auteur, cette approche permet de répondre aux problèmes d'efficacité lorsqu'une requête porte sur l'extension d'une classe (faiblesse sémantique du critère de groupement des objets, optimisation difficile lorsqu'on considère des sous-ensembles).

D'autres systèmes, encore, ont choisi de gérer l'extension d'un type, mais en laissant une certaine liberté à l'application comme dans ENCORE [Zdonik 88, Shaw 89], où la liaison implicite entre une collection (appelée *class*), et son type est optionnelle; les classes correspondant à des types liés par un lien d'héritage sont elles mêmes liées par le lien *est-une-sous-classe-de*.

Enfin certains systèmes comme Iris [Fishman 89, Horowitz 91], n'offrent qu'une gestion basée sur les extensions de classes; c'est surtout le cas de systèmes reposant sur le modèle relationnel pour stocker des objets.

Gemstone suit une approche différente de Smalltalk pour la gestion de l'extension d'une classe [Bretl 89]; une des raisons est la gestion de l'effacement (voir sections 3.4 et 3.5). Dans Gemstone, dès qu'un objet n'est plus atteignable à travers une racine de persistance, l'objet est supprimé (même approche qu'Eiffel).

La vision de Statice en matière d'effacement est différente: lorsqu'une entité est effacée, alors la valeur nulle est affectée à tous les attributs qui le référencent. Comme l'objet restera toujours référencé par l'extension de la classe.

### Profondeur de la requête

Dans des langages de requêtes interactifs il faut exprimer la profondeur au moment de la description de la requête. Pour cela, il faut ajouter une syntaxe au langage permettant de limiter le polymorphisme (en indiquant une contrainte supplémentaire). On remarquera que si au contraire on veut augmenter le polymorphisme, il faudra choisir une classe plus haute dans la hiérarchie des classes.

Ainsi, dans Orion, si l'on veut sélectionner seulement les employés d'un ensemble

de personne, on écrira:

```
select PERSONNE (:M is-a EMPLOYE)
```

Dans POSTQUEL [Rowe 87] et Orion [Kim 91], la profondeur de la requête peut être étendue à tous les héritiers en le spécifiant explicitement (le symbole '*' suit alors le nom de la structure concernée). Orion va plus loin en proposant un mécanisme permettant de choisir seulement certaines branches de la hiérarchie (voir fig. 4.31).

```
select PERSONNE * difference RETRAITE      (1)
select RETRAITE union EMPLOYE              (2)
```

Figure 4.31: Le choix de la cible d'une sélection

Dans le premier cas (1), on obtient l'ensemble des personnes n'incluant pas des retraités, alors que dans le second (2), on obtient l'ensemble des retraités et des employés, les autres personnes étant exclues.

Dans le langage O++ [Agrawal 89a], on distingue une requête faite sur toute une hiérarchie, d'une autre faîte sur une seule classe, par l'utilisation d'un itérateur différent (*forall* par opposition à *for*). On notera que dans le cas du forall, un seul cluster peut être considéré. La forme prise par l'itérateur *forall* est très similaire à celle de l'itérateur *for*. Par exemple on imprime à l'écran le nom de toutes les personnes de plus de 21 ans (voir fig. 4.32).

```
forall p in PERSONNE suchthat (p.age > 21) by (p.age)
printf ("nom: %s \n", p.nom);
```

Figure 4.32: Dèsignation de la cible d'une sélection en O++

On notera par ailleurs, qu'il est possible de restreindre la profondeur d'une requête en utilisant l'opérateur *is*. Dans la figure 4.33 on imprimera seulement le nom des personnes qui sont des employés.

```
forall x in PERSONNE suchthat x is EMPLOYE
printf ("nom: %s \n", p.nom);
```

Figure 4.33: Restriction de la cible d'une sélection en O++

Dans Treillis / Owl [O'Brien 86], ce sont les paramètres de la classe *COLLECTION* (cf. § 4.2.1), initialisés lors de la création d'une instance qui permettent de régler la profondeur de la requête, c'est donc une gestion manuelle. Une collection pourra donc en ajoutant une collection dans l'ensemble des collections parentes se déclarer sous-collection de celle-ci. Ainsi les instances qu'elle contient seront concernées par une requête sur la collection parente.

Le problème de la profondeur se pose dès que l'on a une hiérarchie de collections, même si celles-ci ne sont pas des extensions. Cependant comme c'est le cas dans la plupart des autres systèmes étudiés (cf. § 4.2.1) qui ne gèrent pas l'extension des classes, le problème de la profondeur de la requête se pose moins puisque c'est l'application qui décide des regroupements.

### Cible(s) d'une requête

Une collection est une cible de requête lorsque le système lui associe une variable libre d'éléments de collection pour la parcourir; le nombre de cibles dans une requête dépend essentiellement de:

- la possibilité de naviguer dans le graphe des objets persistants,
- la présence d'une hiérarchie dans les classes,
- des facilités offertes pour le regroupement,
- de l'acceptation des emboîtements,
- d'un besoin de compatibilité par rapport au modèle relationnel.

Vbase, Gemstone en intégrant les facilités de requête dans les collections [Kim 91, Horowitz 91], et en permettant des critères avec des expressions pointées, marquent leur préférence pour les requêtes ayant une seule collection cible.

Lorsqu'on écrit des requêtes emboitées, il faut pouvoir faire référence à plusieurs collections d'objets dans une requête; on notera que parfois une requête avec plusieurs niveaux d'emboîtement consulte plusieurs fois la même collection (certaines cibles de la requête concernent alors la même collection); c'est le cas d'Object-Store [Lamb 91] ou Vbase [Andrews 91 a]. Par contre, Gemstone ne permet pas la définition de plusieurs cibles et ne permet pas de requêtes emboîtées.

Lorsque l'on introduit une hiérarchie entre les classes on a vu qu'il fallait définir la

profondeur d'une requête; la requête peut alors concerner plusieurs cibles.

De même, on a vu que la plupart des systèmes permettaient de regrouper des objets arbitrairement dans des collections, ils se doivent alors de pouvoir réaliser des requêtes simultanément sur plusieurs d'entre-elles. C'est le cas de ODE [Agrawal 89 a] avec le concept de *cluster*, de *O2* [O2-Technology 91 a] avec les *filters* de *O2-Query*, et de Static [Weinreb 91, Kim 91].

### Résultat d'une requête

Dans ENCORE [Shaw 89], le résultat d'une sélection sur une classe (en fait une collection) est une autre classe associé au même type. La nouvelle classe est une sous-classe de la classe initiale; ainsi on peut faire des requêtes non seulement sur l'extension d'un type mais aussi sur des sous-ensembles d'instances. On notera que le type d'un objet est fixé statiquement en fonction de la collection de départ.

Dans *O2* [O2-Technology 91 a] et Object-Store [Lamb 91], l'approche est semblable en ce qui concerne les structures possibles d'un résultat. En effet, en *O2* une requête peut rendre deux types de résultat possible: des objets et des valeurs. Tous les objets retournés par une requête (et leur classe aussi), existent déjà dans le schéma de données, par contre les valeurs peuvent être construite à l'exécution [Bancilhon 89].

Dans Orion, le résultat d'une requête est toujours un sous-ensemble des instances d'une classe.

Dans IRIS, et dans EXODUS, largement influencés par les SGBD relationnels fournissent comme résultat un ensemble de tuples.

### Encapsulation des données

Dans *O2*, le langage de requêtes viole l'encapsulation dans son mode spécifique (à travers le langage de requête), mais il ne la viole pas lorsqu'il est incorporé dans un langage de programmation (C++, C), ou lorsqu'il est utilisé à partir du langage de programmation de base de données de *O2* (*O2*-C).

Dans Orion, l'encapsulation n'est jamais violée: seules les méthodes peuvent être utilisées dans les requêtes.

En ce qui concerne EXODUS, l'encapsulation est violée pour les valeurs (notion similaire à celle de *O2*), mais pas pour les objets (instances de classes), pour lesquels une requête ne peut faire référence qu'à des fonctions appropriées.

# 5

# Concepts de base

## 5.1. Concept d'objet persistant

### 5.1.1. Intégration dans le schéma de conception

En Eiffel, le type le plus général est représenté par la classe *ANY* qui est héritée par toutes les autres classes, de manière implicite. L'objectif de cette classe est de procurer à tous les objets un ensemble de primitives permettant d'augmenter la souplesse d'utilisation et la lisibilité des programmes. La persistance étant un service qui doit pouvoir être offert à tout objet d'un processus, les routines qui s'y rapportent doivent naturellement être intégrée dans la classe *ANY*.

Selon notre approche, le fait qu'une routine manipule des objets persistants ou non doit être complètement transparente à un programme Eiffel (cf. § 3.1); par contre, dans certains cas, il pourra être utile d'avoir des informations sur les objets référencés, notamment pour définir des critères de sélection. Il faudra donc ajouter des primitives supplémentaire à la classe *ANY* (voir fig. 5.1) pour:

• savoir si un objet est persistant ou non;

• accéder à l'identificateur d'un objet; on notera qu'il est déjà possible en Eiffel d'accéder, pour chaque objet, à son adresse, il suffira donc de généraliser et de retourner, lorsqu'un objet est persistant, non pas son adresse mais son identificateur;

• obtenir d'autres renseignements comme la date de création ou de dernière modification d'un objet qui peuvent être intéressants pour la maintenance des objets.

**Remarque**:

Il est très important de n'enrichir les caractéristiques de la classe ANY que par des fonctions et non pas par des attributs, afin de ne pas grossir la taille de tous les objets Eiffel. A ce sujet, on remarquera que l'identificateur d'objet persistant (POI) ne fait pas partie des attributs d'un objet, mais qu'il est cependant possible d'en obtenir la valeur sous forme de chaîne de caractères par une fonction: l'identificateur persistant d'objet n'a pas à être manipulé directement par une application (sauf peut-être dans des composants de bas niveau).

D'autre part, toujours dans un souci de ne pas pénaliser les objets volatiles, leurs dates de création ou de modification sont toujours égales à 0, cette information étant peu pertinante pour ces objets.

Enfin, le fait de mémoriser les dates de création et de dernière modification pour chaque objet persistant, est très couteux et donc difficilement envisageable, d'autant plus qu'une précision de l'ordre de la journée est le plus souvent suffisante. On pourrait ainsi mémoriser cette information globalement à un processus.

```
class ANY
feature
        .......
        is_persistent: BOOLEAN
                -- is current object persistent

        object_id: STRING
                -- object reference (address or persistent identifier) with a string format

        creation_time: INTEGER;
                -- creation time of persistent object

        last_modification_time: INTEGER;
                -- last modification time of persistent object
        .......
invariant
        -- is_persistent implies object_id is a persistent object identifier
        not is_persistent implies (creation_time = 0 and last_modification_time = 0)

end -- Class ANY
```

Figure 5.1: Informations concernant la persistance, placées dans la classe *ANY*

### 5.1.2. Influence de la persistance sur le langage

Les opérations de création, suppression, copie, duplication, sont communes à tous les objets. Dans la version 3, elles sont intégrées dans le langage, à travers des routines fortement dépendantes de la structure interne des objets, tandis que dans la version 2.3, elles sont implantées par des mots-clés: *Create, forget, clone*. Leur rôle doit être précisé lorsqu'elles s'appliquent à des objets persistants.

On étudiera aussi l'influence de la persistance sur deux catégories de routines: *once* qui a la particularité d'être exécutées une seule fois, et *external*, qui est décrite dans un autre langage.

#### opérateur de création

Avant de pouvoir utiliser un objet en Eiffel, il faut l'avoir créé au moyen d'une routine se trouvant dans la clause *creation* d'une classe; dans les versions 2.3 et antérieures, cette clause n'existait pas et l'initialisation d'un objet était effectuée par la routine *Create*. Dans notre modèle, par défaut au moment de sa création, tout objet Eiffel est temporaire (préservation de la philosophie du langage), et l'espace qu'il occuppe est identique à celui d'un objet Eiffel de la version 3.

Par contre, si l'objet est l'instance d'une classe pour laquelle on a redéfini la routine Create, en incluant l'ordre de rendre persistant tous les objets créés, alors toutes les instances de la classe seront persistantes dès la création (extension explicite de la classe).

#### opérateur de suppression

Dans le langage Eiffel, la suppression d'un objet ne peut être demandée explicitement: on peut seulement supprimer un lien entre deux objets par l'intermédaire d'une affectation de la valeur *Void* à la source du lien (*nom_attribut:= Void*); dans la version 2.3, ceci était à la charge de la primitive *Forget* (*nom_attribut.forget*). Dès qu'un objet n'est plus référencé, la suppression d'un objet peut être réalisée automatiquement par le ramasse-miettes de l'exécutif Eiffel.

Dans notre modèle la suppression effective d'un objet persistant est sous la responsabilité de l'outil qui gère la persistance. Cette suppression est soumise à la triple condition que l'objet ne soit plus référencé, ni par d'autres objets persistants, ni (pendant la durée du programme) par d'autres objets volatiles et que cet objet ait été déclaré obsolète (cf. § 5.1.3).

**Conséquences:**

• la sémantique de l'affectation de la valeur *void* est la même pour un objet persistant que pour un objet temporaire; la différence vient du fait que dans le premier cas la suppression est un traitement en mémoire persistante, alors que dans le deuxième elle est l'oeuvre du ramasse-miettes d'Eiffel.

• soit l'attribut *A* d'un objet persistant *obj1* référençant un objet *obj2*; faire *A:= Void*, supprime la référence ver *obj2* en mémoire persistante (si l'utilisateur a les droits suffisants, comme nous le verrons au chapitre 7),

• Si un objet *O* est référencé seulement par des objets candidats à être supprimés, il devient lui aussi candidat à être supprimé; cette règle est récursive sur tout le graphe d'objets, ayant pour racine l'objet *O* (pour plus de détail voir chapitre 8).

### Opérateur de duplication

On rappelle qu'en Eiffel, l'opération *Clone* ne permet la duplication d'un objet qu'au premier niveau (les objets référencés sont partagés). L'opération *deep_clone* permet une duplication profonde de tous les objets partagés.

L'application sur un objet persistant de la primitive *Clone*, suit la même règle et provoque le retour d'un objet volatile. Cela permet à cette opération de s'adapter aussi bien à une duplication temporaire que durable, en fonction du contexte du programme.

L'opération *deep_clone* implantée au niveau de la classe *ANY*, permettra d'obtenir une copie profonde volatile d'un graphe d'objets persistants, dans le même souci de rendre la programmation la plus souple possible.

Remarque: Si le résultat d'une copie est affecté à un attribut d'objet persistant, il deviendra automatiquement persistant (Règle n°4, paragraphe 3.1).

### Copie d'un objet

En cas de copie du contenu d'un objet *obj1* dans un autre objet *obj2*, ou de son attachement à un objet expansé *obj3*, (il devient dans ce cas privé à *obj3*) les identificateurs ne sont pas affectés. On n'aura donc pas à considérer le fait qu'un objet est persistant ou volatile (sauf peut-être noter la mise-à-jour à effectuer). Si l'objet *obj2* est persistant alors son nouveau contenu sera propagé en mémoire persistante.

### Opérateur d'égalité

L'opérateur d'égalité (noté =) dans le langage Eiffel, porte sur les références (sauf pour des objets expansés où il porte sur des valeurs). Ainsi, il faudra étendre l'opérateur d'égalité afin de permettre l'égalité d'identificateurs persistants (POI). Cependant, deux références ne pourront être égales que si elles sont de même type (égalités d'adresse ou de POI).

### La sémantique d'une routine Once

Le problème d'une fonction ou d'une procédure *once* est que dans la version 3 elle est adaptée à l'usage d'objets non persistants mais très peu à celle d'objets persistants.

Exemples: utilisations d'une primitive Once

• l'initialisation de l'écran en début de session doit être refait pour chaque processus Eiffel d'un même programme, (unicité de l'exécution d'une primitive *Once* sur la durée d'un processus)

• mémorisation d'un objet partagé. Il doit pouvoir être conservé pour la prochaîne exécution du programme (unicité de l'exécution de la primitive Once sur l'ensemble des processus représentant un même programme).

Cette différence d'utilisation reflète en fait la différence entre une action réalisée une seule fois, et une variable de classe [Goldberg 89]. Peut être faudrait-il différencier ces deux notions dans le langage.

On peut certes introduire introduire cette différenciation par l'adjonction de routines [Lahire & al. 91], mais une telle solution ne pourra être qu'une pièce rapportée. Il sera préférable de n'utiliser les primitives *once* que pour exprimer des fonctions ou des primitives ne devant être exécutées qu'une fois pendant la durée de vie du processus (sémantique actuelle), et fournir en supplément la notion de **variable de classe** pour exprimer le partage d'une propriété par les instances d'une classe. Ainsi, la propriété décrite par la variable de classe aurait la même persistance que sa classe. Le concept de monde persistant d'Eiffel (cf. § 3.6) devrait faciliter l'intégration du concept de variable de classe.

### Cas des objets privés à d'autres objets

Déclarer un attribut comme *Expanded*, entraîne le non partage de l'objet désigné par

cet attribut (récursivement) et exhibe le caractère privé de cet objet. Le caractère expansé d'un objet devra être pris en compte notamment par les opérateurs d'affectation, de copie et de duplication.

Un objet expansé n'a pas une existance par lui-même, il fait parti de l'objet qui le désigne, et donc à ce titre, dans Eiffel 2.3, il ne fera pas parti de l'extension de la classe correspondante; dans Eiffel 3, où une classe peut être expansée, celles-ci n'auront donc pas de **Pcollection** associée. Cette solution est parfaitement orthogonale à la gestion des types dits simples (*INTEGER, BOOLEAN, CHARACTER, REAL*, ..).

#### Cas des routines externes

Une routine externe (clause *external*), est par définition écrite dans un autre langage qu'Eiffel, elle n'est donc pas sous le contrôle de l'exécutif Eiffel. Une conséquence immédiate est "l'obligation morale" de ces routines, d'avertir l'exécutif Eiffel lorsqu'elle veulent utiliser, modifier ou mettre à jour un objet persistant. Tous les mécanismes de gestion automatique des objets persistants pris en compte habituellement par l'exécutif sont à la charge de la routine externe. L'accès aux objets Eiffel par des routines Eiffel appelées à partir de routines externes sera étudié dans le chapitre traitant de l'adaptation de l'exécutif (voir chapitre 10).

On notera que si la routine externe n'est pas dépendante de la structure des objets, alors si le gestionnaire d'objets a un moyen d'accès au langage externe, elle pourra être intégrée sans autre modification. Par contre, si la routine dépend de la structure des objets, il faudra une intervention du développeur.

### 5.1.3. Influence de la persistance sur l'application

Nous avons introduit le concept d'**objet obsolète** dans le paragraphe 3.5; un objet est obsolète lorsqu'il est retiré de la Pcollection associée à sa classe. Nous allons présenter une intégration possible du concept d'*objet obsolète*.

## Intégration du concept d'objet obsolète

Si un objet obsolète est consulté par un processus, alors une exception *obsolete-object*, est déclenchée (cf. § 7.1). Selon le niveau d'appel où l'erreur est récupérée (dans la procédure ou au niveau de la classe principale), un traitement plus ou moins spécifique sera possible, et devrait avoir besoin de l'identificateur persistant de l'objet où a eu lieu l'exception. Si l'exception n'est pas récupérée, alors le comportement de l'exécutif Eiffel sera identique a celui observé lorsqu'une erreur se produit sur un objet volatile (ex: affichage de la pile des appels depuis le déclenchement de l'exception).

En cas d'exception *obsolete-object*, la pile des appels pourra produire un certain nombre de renseignements sur l'objet obsolète, comme par exemple le nom de l'utilisateur ayant déclaré l'objet obsolète, ou l'identificateur d'un autre objet compatible destiné à le remplacer.

Bien que l'accès à un objet obsolète ne doit pas être une opération fréquente on pourra aussi ajouter dans la classe *ANY*, un moyen de savoir si un objet est obsolète (voir fig. 5.2); cela se révèlera utile surtout dans le cas où une application cherche à accèder à un objet qu'elle a rendu obsolète pendant son exécution.

```
class ANY
feature
        .......
   is_obsolete (obj: ANY): BOOLEAN is
            -- is obj obsolete ?
        .......
invariant
   not is_persistent implies not is_obsolete
end -- Class ANY
```

Figure 5.2: L'obsolescence d'un objet dans la classe *ANY*

De même il sera utile de pouvoir disposer dans la classe *EXCEPTION* de la bibliothèque Eiffel (fig. 5.3), si l'objet est persistant ou si l'exception est de type *obsolete-object*,

- de l'identificateur du dernier objet concerné par une exception,
- du nom de l'utilisateur sur la machine ayant déclaré l'objet obsolète,
- de l'identificateur de l'objet remplaçant (de type conforme).

On notera que l'information concernant le numéro de l'objet est déjà donné par la pile des appels d'Eiffel, mais que son intérêt est largement augmenté quand cela concerne un objet persistant.

```
class EXCEPTION
feature
        .......
        object_id_of_last_exception: STRING
                -- identifier of the object where the exception occured
                -- (when exception occured on a persistent object)

        new_object_id: STRING
                -- identifier of new object to be (eventually) considered when
                -- exception is obsolete-object

        user_name: STRING
                -- User name when exception is obsolete-object
        .......
end -- Class EXCEPTION
```

Figure 5.3: Enrichissement de la classe EXCEPTION

## Exécution d'un système Eiffel

A l'heure actuelle, les routines de toutes les classes se trouvent dans le binaire de l'application. Cela pose un problème dans le cas d'un traitement sélectif, ou plus généralement lors de la consultation ou de la modification d'un objet persistant: le mécanisme de liaison dynamique rend possible l'accès à une instance dont la classe n'est pas incluse dans le binaire de l'application.

Exemple: consultation et modification d'un objet persistant

Supposons que dans le monde persistant on trouve des instances de la classe *CIRCLE*, qui hérite de la classe *FIGURE*. Supposons aussi que dans un processus en train de s'exécuter et ne contenant pas la classe CIRCLE, on cherche à lire à partir d'un attribut de type *FIGURE*, un objet persistant de type *CIRCLE* (ce qui est parfaitement valide en Eiffel). On devra pouvoir appliquer à cet objet toutes les méthodes décrites dans la classe qui le modélise, et accéder à ses attributs.

Le problème est que, ni les routines de la classe *CIRCLE*, ni ses attributs ne sont connus du processus, qui est donc incapable de poursuivre l'exécution du programme. Dans la version actuelle d'Eiffel, ceci n'est pas possible (l'opérateur **?=** interdit l'affectation), et pourtant cela semble être une situation courante: plusieurs applications peuvent ajouter des instances et ne contiennent pas forcément les mêmes classes.

Les deux solutions suivantes sont envisageable pour permettre l'exécution des routines d'un objet persistant entrant dans le cas de figure décrit ci-dessus:

• laisser à l'exécutif Eiffel le contrôle de l'exécution des routines et reprendre les techniques utilisés dans l'interprète pour faire la liaison entre les mondes compilés et interprétés, afin d'inclure dynamiquement dans un processus les classes, représentant des instances consultées ou modifiées, n'appartenant pas au système Eiffel;

• rendre les routines persistantes et réaliser l'exécution de la routine en mémoire persistante, sous le contrôle du gestionnaire d'objets (l'objet et la routine sont gérés par lui).

Nous préférons la première solution. Elle nous semble plus orthogonale, toutes les instances persistantes sont traités de la même manière. Ce choix sera argumenté dans les chapitre 9 et 12.

**Conséquence**:

Dans le premier cas, si une instance persistante d'une classe C n'appartenant pas au processus est consultée ou modifiée par ce dernier, la classe C est rajoutée au processus. Dans le second cas, le nombre de classe gérées par l'exécutif ne varie pas tout au long de la durée de vie d'un processus.

# 5.2. Généralisation de la notion d'objet persistant

## 5.2.1. Statut d'une classe

Dans le paragraphe 3.1, en particulier la règle N°5 sur les traitements en mémoire persistante, on met en évidence le besoin de retrouver la classe d'un objet en mémoire persistante; dans le paragraphe 3.6, on souligne l'intérêt d'une uniformisation de la gestion des entités persistantes (classes et objets persistants). Il faut donc disposer d'une structure qui permette la modélisation d'une classe (liens d'héritage, primitives, ....).

Dans la version 3 d'Eiffel, on peut actuellement obtenir certaines de ces informations, à travers un composant *E_CLASS*; cependant cette approche n'est plus acceptable dans la

perspective d'une gestion efficace et puissante des objets persistants:

• pas de gestion des accès multiples à une classe (utile notamment dans la phase de conception), et limitations aux classes mentionnées dans le fichier décrivant le système Eiffel (fichier *.eiffel*),

• accès lent aux informations stockées dans des fichiers, ou informations limités si on ne dispose que de celles disponibles dans l'exécutif

• manque de possibilités pour la réalisation de passerelles entre les mondes compilé et interprété,

• difficultés pour mettre en oeuvre des traitements en mémoire persistante; cela pénalise notamment la mise en oeuvre de langages de requêtes, dans le cas où l'évaluation se fait en partie en mémoire persistante (voir chapitre 11).

Si on examine la notion de méta-classe [Goldberg 89], celle-ci permet de considérer une classe comme un objet, ce qui est à la fois idéal pour le prototypage et esthétique.

En Eiffel, cette notion de méta-classe n'existe pas et Bertrand Meyer dans [Meyer 88 & 90], argumente dans ce sens. Ainsi, il y a très peu de primitives de classes (*Create*, .... ), et seuls les environnements interprétés utilisés pour le prototypage ou la mise au point ont des raisons d'offrir le paradigme *tout est objet*. Ainsi l'interprète Eiffel décrit dans [Brissi 90], utilise de manière interne la notion de Méta-classe. Ce besoin correspond au fait que dans un environnement interprété il est nécessaire de connaître des informations sur les classes de façon dynamique pour pouvoir les interpréter.

Même si d'un point de vue conceptuel nous préférons l'approche «*tout est objet*» nous dèsirons respecter la philosophie d'Eiffel, et dans notre approche nous découpons la représentation d'une classe en deux parties:

• La partie visible par les applications Eiffel représentée par les **instances persistantes** de la classe *EIFFEL_CLASS;* cette classe est une généralisation de la classe *E_CLASS* de la version 3, en tenant compte des informations propagées dans le monde persistant;

• La partie visible seulement par l'outil persistant[1] et l'exécutif Eiffel, qui permet de gérer l'accès navigationnel aux objets et l'exécution de leurs routines, mais n'intégre pas ces fonctionnalités sous forme d'objets.

Remarque:

Les contraintes et le modèle de l'exécutif Eiffel et de l'outil persistant n'étant pas forcément les mêmes, la partie immergée pourra avoir deux représentations différentes:

• celle correspondant à l'exécutif Eiffel, plus quelques adaptations (voir chapitre 10),

• celle utilisée par le gestionnaire d'objets, pour stocker, et réaliser des traitements en mémoire persistante (ex: les requêtes sélectives), (voir chapitres 11 et 12).

On étudiera les aspects concernant la redondance des représentations et la propagation des instances de *EIFFEL_CLASS* en mémoire persistante dans les chapitres 9 et 12.

### 5.2.2. Intégration dans le schéma de conception

Compte tenu des remarques qui ont été faites dans le paragraphe précédente, il faut pouvoir disposer pour chaque classe d'un objet persistant centralisant toute l'information concernant une classe (texte, code objet, sémantique du composant, ..), et permettant un accès efficace (informations non calculées). La modélisation de ces informations correspond à la classe *EIFFEL_CLASS* (voir fig. 5.4).

```
class EIFFEL_CLASS
feature

  create (class_name: STRING) is
        -- Create an object representing class class_name (chemin absolu)
    ensure     not is_error implies (is_persistent and name /= void and path /= void)

  name: STRING
        -- Eiffel class name, in upper case

  path: STRING
        -- Absolute file name, in lower case, of name

  set_access_rights (rights: STRING)
```

[1] Dorénavant nous parlerons de gestionnaire d'objets persistants (POM).

```
        -- set and propagate new access rights in persistent memory

    access_rights: STRING
        -- access rights of current class
      require   is_propagated

    source: TEXTE
        -- Source of class name

    binary: TEXTE
        -- Object code of class name

    compiled: BOOLEAN
        -- Has the class been compiled according to current version of source?

    parents: ARRAY [EIFFEL_CLASS]
        -- Parents of current class.

    suppliers: ARRAY  [EIFFEL_CLASS]
        -- Suppliers of current class

    ancestors: ARRAY  [EIFFEL_CLASS]
        -- Ancestors of current class

    clients: ARRAY  [EIFFEL_CLASS]
        -- Clients of current class

    descendants: ARRAY  [EIFFEL_CLASS]
        -- Descendants of current class

    compile (flag: STRING) is
        -- Compile class with  `flag' options (modification propagated by the translator)

    features: ARRAY [FEATURE]
        -- Features of class.

    is_deferred: BOOLEAN
        -- Is current class deferred ?

    compilation_status: BOOLEAN
        --  Flags.

    -- + Tous les les routines pour mettre à jour ces informations

end -- class EIFFEL_CLASS
```

Figure 5.4: Description d'une classe Eiffel en mémoire persistante

D'abord, au sujet du code objet, il faut savoir que ce code est différent pour le gestionnaire d'objets et l'exécutif Eiffel (ceci sera pris en compte au niveau de la mise en oeuvre, chapitre 12), mais une application Eiffel n'aura jamais besoin de voir que le code objet qui lui correspond.

Par ailleurs, on notera que comme pour tout objet persistant, on doit disposer (classe *ANY*), de la date de création et de dernière modification. Du point de vue de l'application une classe Eiffel est vraiment vue comme un objet persistant.

Chaque primitive d'une classe Eiffel a un nom, elle peut-être exportée éventuellement de manière limitée, être retardée. En fonction de sa nature, elle pourra être de type *once*, être obsolète, retourner une valeur éventuellement calculée, avoir des paramètres et un corps.

Toutes ces informations seront mémorisées par l'intermédiaire de deux classes (fig. 5 & 6) qui permettent de différencier les primitives déclarées dans une classe (*FEATURE*), de celles héritées et éventuellement renommées ou redéfinies (*INHERITED_FEATURE*).

```
class FEATURE
creation    create
feature

  create (feature_name: STRING, object: EIFFEL_CLASS) is
        -- creation of feature_name de from class object
    ensure     not is_error implies
                        (is_persistent and not object_class.void and name = feature_name)

  name: STRING
        -- name of the feature

  object_class: EIFFEL_CLASS
        -- class which contains current feature

  kind_of: INTEGER is
        -- kind of feature
    ensure     Result = is_a_function or is_a_procedure or
               is_a_constant or is_an_attribute or is_undefined

  exported_list: LINKED_LIST  [E_CLASS]
        -- list of classes which may use the feature thru client / supplier relationships

  is_secret: BOOLEAN
        -- is the feature secret?

  is_full_exported: BOOLEAN
        -- is the feature fully exported?

  is_deferred: BOOLEAN
        -- is a deferred feature?

  is_obsolete: BOOLEAN
        -- is an obsolete feature ?

  is_once: BOOLEAN
```

```
        -- is a once feature?

   type: STRING
        -- Type of the feature result if any

   parameters: LINKED_LIST  [ATTRIBUTE]
        -- parameters if any

   body: LINKED_LIST[STRING]
        -- body of the feature if any

is_a_function: INTEGER is unique
is_a_procedure: INTEGER is unique
is_a_constant: INTEGER is unique
is_an_attribute: INTEGER is unique
is_undefined: INTEGER is unique

invariant
   is_once implies not is_deferred;
   is_deferred implies (not is_once and body.void and kind_of = is_undifined);
   kind_of = is_a_procedure implies f_type.void;
   kind_of = is_an_attribute implies (body.void and parameters.empty);
   kind_of = is_a_constant implies (body.void and parameters.empty);
   is_secret implies exported_list.void;
   is_full_exported implies exported_list.empty

end -- Class FEATURE
```

Figure. 5.5: Description d'une primitive Eiffel

Lorsqu'on hérite d'une classe, on peut donc redéfinir ou renommer une ou plusieurs primitives. La classe *INHERITED_FEATURE* permet de prendre en compte ces modifications en modifiant l'implantation des caractéristiques pouvant être affectées par les opérations de redéfinition ou de renommage (*body*, *f_type*, *f_name*, *parameters*). On pourra ajouter certaines nouvelles possibilité de la version 3 d'Eiffel (clauses *export*, *define*, *select*, etc).

```
class INHERITED_FEATURE
inherit
   FEATURE
feature
   is_redefined: BOOLEAN is
        -- is The feature redefined?

   is_renamed: BOOLEAN is
        -- is the feature renamed?

end -- class INHERITED_FEATURE
```

Figure. 5.6: Description d'une primitive Eiffel héritée par une autre classe

# 6

# Mécanismes de sélection

Nous avons vu (section 3.3) que les opérations fondamentales du concept de *collection*, que nous appelerons **requêtes** ("query" ) ou **requêtes sélectives,** sont:

• **la fonction sélective,** qui porte sur une collection et rend le sous-ensemble de celle-ci **(une sélection)** dont les objets satisfont un critère,

• **l'itération sélective** qui active une primitive ou une procédure, pour chaque objet qui satisfait un critère.

• **les quantificateurs** existentiel ∃ et universel ∀.

Ce chapitre étudie **comment exprimer ces requêtes** dans le langage Eiffel, sans attacher trop d'importance aux problèmes de performance et de liaison avec le **gestionnaire d'objets persistants** *(POM)* qui seront abordès dans les chapitres suivant (voir chapitres 9 à 12). Ainsi, la persistance des collections ou des objets n'a aucune importance sur le plan de l'expressivité: lorsque nous aurons à détailler des algorithmes, nous nous placerons dans le cas plus simple d'objets volatiles**,** et sans nous préoccuper, ni des réécritures que pourront éventuellement effectuer les phases d'optimisation, ni des requêtes au *POM* dans le cas d'objets persistants. Ces algorithmes ne seront donnés ici qu'à titre indicatif, pour exhiber les informations qui leur sont nécessaires.

Pour exprimer des requêtes en Eiffel il nous faut:

• recenser tous les éléments textuels qui interviennent pour les exprimer;

• étudier comment les combiner et à quel moment les fournir, c'est-à-dire choisir une technique de paramétrage;

• s'assurer de la qualité de l'intégration de la solution retenue au langage Eiffel et à son environnement actuel, en s'efforçant de réutiliser le maximum de choses (classes existantes, constructions du langage, principes. . . ) et en préservant les qualités initiales du langage, notamment son orthogonalité (ceci peut nous amener à envisager des généralisations à d'autres cadres que celui de la persistance);

• étudier la faisabilité des controles statiques ou dynamiques;

• s'assurer des possibilités d'optimisations.

Comme c'est souvent le cas en conception, il n'est pas possible de traiter tous les problèmes de manière strictement séquentielle et indépendante: il faut une approche globale. Il n'est pas possible, non plus, de trouver du premier coup une bonne solution: il faut souvent tâtonner, procéder de manière incrémentale et heuristique. Aussi, pour argumenter les choix effectués, nous donnerons un résumé de notre démarche en présentant quelques solutions abandonnées, auxquelles le lecteur pourrait songer.

Le paragraphe 6.1 identifie les aspects à considérer pour exprimer des requêtes sélectives. Les deux paragraphes suivantes présenteront deux approches basées sur des adjonctions syntaxiques.

Dans le premier le moteur du mécanisme de sélection est propre à chaque structure parcourable (liste, ensemble, collection, ...); ce paragraphe fournit la construction progressive de la syntaxe d'un langage de requêtes que nous appelerons EQL ("Eiffel Query Language"), en intégrant différents aspects et en testant son expressivité sur des exemples. Dans le second, le moteur de la sélection est partagé par toutes les structures parcourables, comme c'est le cas avec la classe ITERATOR [Meyer 90]; ce parageraphe introduit une nouvelle structure itérative, pour l'intégration des requêtes.

Précisons que comme il ne nous appartient pas de modifier le langage Eiffel pour y inclure les possibilités d'EQL ou une nouvelle structure itérative, ce mini-langage et cette structure de contrôle ne serviront que de référence pour comparer les solutions développées en Eiffel dans les paragraphes suivants.

Le paragraphe 6.4 présente trois intégrations possibles par l'adjonction de composants (Eiffel 3). Deux sont basés sur l'intégration du moteur dans les structures: encapsulation des routines dans des classes, et utilisation de routines paramétriques, tandis qu'une troisème à base d'héritage répété, illustre une approche par partage du moteur du

langage de requête.

Le paragraphe 6.5 présente une variante de la solution à base de routines paramétriques mais avec les possibilités de la version 3 d'Eiffel.

Le paragraphe 6.6 fait le bilan de ces deux solutions, discute d'autres perspectives possibles (notamment l'interprétation du langage EQL) et donne nos recommandations sur les choix à opérer.

Enfin dans le paragraphe 6.7, nous généralisons la notion de requête à l'ensemble des structures traversables, et nous étudions une structure pour la gestion implicite des *Pcollections*.

Une dernière section (section 6.8), permettra de faire le bilan de l'intégration du langage de requêtes vis-à-vis de certains problèmes évoqués dans le paragraphe 6.1.

# 6.1. Problèmes à résoudre pour l'intégration

## 6.1.1. Eléments de base: type, variable libre et critère

Rappelons tout d'abord la syntaxe proposée dans le paragraphe 3.3 pour ces requêtes:

• *function_ident [x ∈ C] / criteria* pour une fonction sélective, où C est le nom d'une collection et *x* représente la variable libre de la collection qui permet de réaliser la sélection avec le critère *criteria* donné sous la forme d'une expression logique par une "formule bien formée".

• *iteration_ident [x ∈ C] x.routine-call / criteria* pour une itération sélective, où *C, x* et *criteria* ont la même signification que précédemment et où *routine-call* est l'appel à la méthode à appliquer à chaque objet satisfaisant le critère.

• *∃ x criteria* ou *∀ x criteria* pour des quantifications.

Cette syntaxe, dont le seul but est la modélisation, ne prend pas en compte le confort de programmation. Elle ne cite pas de contexte explicite comme nous le verrons plus loin, et paraît assez inadaptée aux habitudes d'écriture d'une approche à objets où l'on préfère citer explicitement l'objet qui active la méthode, soit comme destinataire d'une transmission de message, soit comme tête d'une expression pointée. En remarquant que

la **collection sur laquelle porte la sélection et son résultat sont des objets** et en introduisant explicitement que les quantificateurs sont des opérateurs de collections, on peut améliorer la syntaxe précédente selon la grammaire[1] donnée sur la figure 6.1.

```
selective_query      = selection | selective_iteration | quantification
collection             = collection_ident | selection
selection              = collection.select(x, criteria)
selective_iteration    = collection.do_if (x, criteria,routine-call)
quantification         = exists | forall
exists                 = collection.exists (x, criteria)
forall                 = collection.forall (x, criteria)
```

Figure 6.1: Deuxième notation pour les requêtes sélectives

Les identificateurs *do_if, exists* et *for_all* rappellent les primitives de même nom proposée dans la classe *ITERATION* de la bibliothèque Eiffel 3 qui ont le même sens, mais appliquées à des structures de données traversables au lieu de collections.

## 6.1.2. Composition de requêtes

La grammaire de la figure 6.1 met aussi en évidence des éléments communs à toutes les requêtes: *collection, x* et *criteria***.** Ceci suggère une factorisation évidente: pour effectuer une *"itération sélective"* (resp. une*"quantification"),* il suffit de construire une sélection et d'appliquer ensuite l'itération sur tous ses objets (resp. la quantification en considérant le cardinal "*count*" de la collection résultat).

```
selective_iteration    ⇔ collection.select (x, criteria).do_all(routine-call)
        exists         ⇔ collection.select (x, criteria).count > O
        forall         ⇔ collection.select (x, criteria).count = collection.count
```

Figure 6.2: Composition de requêtes

Cette perspective est d'autant plus intéressante, que le langage Eiffel donne la possibilité syntaxique à un appel de fonction de rendre un objet sur lequel on peut activer une primitive dans la même formule que l'appel, et ceci sans limitation du nombre d'appels: de ce point de vue, Eiffel est aussi un langage fonctionnel!

---

[1] Toutes les grammaires sont données dans la métalangue dérivée de BNF qui est utilisée pour définir le langage Eiffel [Meyer 91].

Ainsi, on pourrait écrire, sans critère, des itérations sélectives ou des quantifications en les appliquant à une sélection explicitement construite (figure 6.2). On peut même envisager de construire la sélection en plusieurs étapes, par composition de plusieurs sélections:

collection.*select (...).select ( ...) .select (...).do_all* (routine-call)

Cependant, cette technique est peu efficace puisqu'elle construit explicitement une collection pour la sélection, ce qui prend de la place en mémoire et consomme du temps de calcul. Ce coût pourra être jugé trop élevé, même dans le cas de collections d'objets volatiles. Ces possibilités de composition de requêtes ne sont donc intéressantes que dans les cas où les collections intermédiaires ont une autre utilité:

- soit qu'on veuille de toute manière les rendre persistantes,

- soit qu'elles puissent servir à d'autres requêtes sélectives.

Dans les autres cas, on intégrera tous les éléments du critère de sélection dans la même requête:

- une itération se fera alors directement sur la collection de départ,

- une composition de sélections $S_i$ de critères $C_i$ se fera en prenant $C = \bigwedge_i C_i$ comme critère de la requête sélective,

- on évitera d'utiliser l'attribut *count* pour les quantifications qui seront intégrées dans un unique critère, rendant ainsi possible d'éventuelles optimisations par réécriture.

Toutefois, on peut aussi implémenter les quantifications par comptage du nombre d'objets d'une collection qui satisfait un critère donné, sans construire explicitement la collection image du filtrage. Dans ce cas, le coût de calcul sera nettement diminué, pratiquement le même que dans le cas d'une quantification universelle, mais plus élevé que pour une quantification existentielle où il suffit d'arrêter la recherche au premier

objet adéquate. Ce quantificateur *"cardinal"* permettra aussi d'aborder des cas plus généraux que ceux qu'on peut écrire avec les quantificateurs classiques: "il y *a-t-il au moins 5 objets qui...", "il y a-t-il au plus 3 objets qui... ".*

Aussi, nous retiendrons l'intérêt du quantificateur cardinal, selon la grammaire proposée sur la figure 6.3.

```
quantification      = exists | forall | count
      count       = collection.count (x,criteria)
      exists        = collection.count (x,criteria) > O
      forall        = collection.count (x,criteria) = collection.count
```

Figure 6.3: Quantifications cardinales

### 6.1.3. Contexte d'une requête

L'écriture de critères ou d'arguments effectifs aux routines itérées nécessite souvent de citer des valeurs qui ne sont pas des caractéristiques *("features")* des objets à sélectionner. Il faut donc logiquement considérer les définitions de ces valeurs comme faisant partie des arguments - implicites ou explicites - des requêtes, puisque leur connaissance est nécessaire à l'exécution de celles-ci.

**définition 6.1: Notion de contexte**

*Nous appelerons* ***contexte*** *d'une requête, la portion de texte qui contient la définition de tous les éléments nécessaires à son écriture, et qui ne fait pas partie des primitives exportées par les objets à sélectionner.*

Une telle portion de texte peut contenir des déclarations de constantes manifestes, d'attributs (expansés ou comme références à des objets à créer), des fonctions locales participant à l'expression des critères et des procédures, notamment de création (clause *Creation*), qui permettent les initialisations de toutes ces valeurs.

On notera aussi que, dans le but de respecter la transparence de la gestion des objets persistants (cf. §3.1), les objets participant à la description du contexte pourront être persistants ou au contraire volatiles; cet aspect devra être pris en compte lors de la communication avec le gestionnaire d'objets (voir chapitres 11 et 12).

Par ailleurs, puisque nous sommes dans un cadre de langages à objets il peut être intéressant et orthogonal de permettre aussi des héritages pour accéder directement à des valeurs, sans nécessiter ni création explicite, ni notation pointée.

Ainsi, le contexte d'une requête correspond au texte d'une classe non générique, n'exportant pas forcément de primitives, mais offrant un cadre syntaxique convenable pour faire tous les héritages utiles, avec leurs renommages et re(définitions) éventuelles. Comme l'évaluation du critère et des arguments effectifs d'une requête se fait pendant l'éxécution du programme, il peut être utile de placer des clauses de rescousses (*rescue*)**.** Par contre, l'usage d'un invariant de classe nous semble peu utile ici. En conclusion, il faut à peu près **les possibilités expressives d'une classe pour décrire un contexte de requête.**

On peut donc envisager trois solutions:

• définir une classe Eiffel pour chaque contexte nécessaire et fournir le nom de cette classe ou de l'une de ses instances en argument de chaque requête: plusieurs requêtes pourront donc se partager le même contexte;

• définir une syntaxe particulière pour l'expression des requêtes qui mentionne un contexte local et spécifique de la requête;

• utiliser comme contexte la classe et la routine où est décrite la requête .

La première solution **(**figure 6.4**)** revient à étendre la syntaxe des requêtes élaborées jusqu'à présent, en leur ajoutant un argument pour le contexte.

La deuxième solution suppose de définir un langage de requêtes qui pourra ou non être intégré au langage Eiffel.

La troisième solution est la plus simple pour l'écriture, mais se prête moins bien à l'interprétation: il faut analyser une vraie classe Eiffel pour calculer le contexte.

```
            selection        =  collection.select (x, criteria, context)
selective_iteration          =  collection.do_if (x, criteria, routine-call, context)
                exists       =  collection.exists (x, criteria, context)
                forall       =  collection.forall (x, criteria, context)
                count        =  collection.count (x, criteria, context)
                context  =  eiffel-expression
```

Figure 6.4: Contexte d'une requete transmis en argument

Toutes ces solutions sont combinables. On peut par exemple admettre qu'une requête écrite dans un langage spécifique avec un contexte local, dispose aussi du contexte englobant de la routine et de la classe où elle est écrite: c'est une combinaison des

solutions 2 et 3. On peut aussi transmettre *Current* à un interprète de requêtes pour fournir comme contexte la classe englobant leur description, puisque la bibliothèque Eiffel permet de retrouver la classe de tout objet: c'est une combinaison des solutions 1 et 3.

### 6.1.4. Itérations vs requêtes sélectives

La bibliothèque Eiffel V.3 fournit un ensemble de classes pour décrire des **itérations.** Ces classes permettent de traverser (ou parcourir) différentes structures de données, notamment séquentielles. Les primitives de telles structures *traversables,* permettent:

- d'examiner si tous leurs items satisfont une condition: *all (i.e. forall),*
- d'examiner s'il existe un item qui satisfait une condition: *exists,*
- d'exécuter une routine pour tous leurs items: *do_all,*
- d'exécuter une routine pour tous leurs items qui satisfont un critère donné: *do_if,*
- en traversant leur structure dans un certain ordre, d'exécuter une routine pour tous leurs premiers items, jusqu'à ce qu'une condition soit remplie: *do_until,*
- en traversant la structure dans un certain ordre, d'exécuter une routine pour tous leurs premiers items, tant qu'une condition est remplie: *do_while.*

Il y a donc une similitude naturelle entre les structures traversables et les collections du point de vue des primitives sélectives. On pourrait aussi rapprocher le cas d'ensembles potentiellement infinis dont les éléments sont construits à la demande par des générateurs à la *Icon* [Griswold 90], ce qui permettrait de fournir tout un arsenal d'opérateurs itératifs utiles aux mathématiciens $\prod$, $\cap$, $\cup$, $\Sigma$, $\wedge$, $\vee$, $\otimes$, $\oplus$. Toutes ces possibilités ont vocation à être rapprochées dans la bibliothèque Eiffel par des héritages appropriés, en distinguant les structures ordonnées ou non.

Pour le moment, contentons nous de considérer les structures de données traversables et les collections: dans les deux cas, on dispose en mémoire persistante ou volatile d'un ensemble d'items que l'on veut examiner soit pour les consulter, soit pour les modifier. Comme les collections ont une nature ensembliste marquée, la notion

d'ordre entre les items n'est pas toujours nécessaire. Nous écartons donc les primitives *do_until* et *do_while* de la classe *ITERATOR*. Quant aux autres primitives, elles ont toutes un équivalent avec les requêtes envisagées pour les collections: *do_if, exists, forall*.

**Discussion**

Faut-il, d'un point de vue conceptuel, fusionner les notions de requêtes sélectives et d'itérations? Nous ne nous intéressons pas ici aux problèmes d'implantation évoqués au début de le paragraphe 4.2.1, mais seulement à l'intégration par rapport au langage Eiffel.

D'un point de vue mathématique, la notion d'ensemble est plus générale que celle de structure de données telles que les listes, les arbres, ou les tableaux, qui en sont des spécialisations: il y a des opérations algébriques de plus. Mais d'un point de vue informatique, un ensemble est une structure de donnée particulière qui ne tolère pas d'occurences multiples du même élément et qui est traversable comme les autres.

La notion de *traversée* est donc une notion située au dessus de celles d'ensemble, de collection et d'autres structures de données. Si l'on fusionne les notions de requêtes sélectives et d'itérations (ce qui paraît conceptuellement souhaitable), il faudra le faire à un niveau très élevé dans la hiérarchie d'héritage, avec toutes sortes de spécialisations pour tenir compte de contraintes particulières, notamment de performance et de persistance. Sans doute même, ces concepts d'itérations et de requêtes sélectives méritent de rejoindre les autres structures de contrôle du langage.

Dans l'étude menée dans les paragraphes 6.2 à 6.6, nous garderons la notion spécifique de *collection* comme champ d'étude; les éventuelles fusions des mécanismes présentés avec d'autres types de structures Eiffel sera évoqué dans le paragraphe 6.7.

### 6.1.5. Emboîtement de requêtes par quantifications

Lors d'une sélection en cours, il arrive souvent, d'entamer une autre sélection interne, et ceci à un nombre quelconque de niveaux, comme cela se fait pour des itérations: nous appellerons ce phénomène *"emboîtement"* de requêtes. Ceci ne doit pas être confondu avec la *"composition"*, qui combine certes plusieurs requêtes, mais de manière séquentielle.

L'emboîtement de requêtes apparaît donc dans l'écriture du critère de sélection, ce qui ne peut se produire qu'avec l'emploi de quantificateurs, ou de l'opérateur cardinal.

En langage mathématique, l'emboîtement n'apparait pas de manière explicite, mais s'observe dès qu'on utilise plusieurs variables libres quantifiées dans la même formule:

1. $\forall x, y \in \varepsilon: \Re(x, y) \Leftrightarrow \forall x \in \varepsilon, \forall y \in \varepsilon: \Re(x, y) \Leftrightarrow \forall x \in \varepsilon, y \in \varepsilon\ \Re(x, y)$
2. $\forall x \in \varepsilon, \exists y \in \xi: \Re(x, y) \Leftrightarrow \forall x \in \varepsilon, \exists y \in \xi\ \Re(x, y)$
3. $\exists x \in \varepsilon, \forall y \in \xi: \Re(x, y) \Leftrightarrow \exists x \in \varepsilon, \forall y \in \xi\ \Re(x, y)$
4. $\exists x \in \varepsilon, \exists y \in \xi: \Re(x, y) \Leftrightarrow \exists x \in \varepsilon, \exists y \in \xi\ \Re(x, y)$

On observe donc que les paramètres de diversité des quantifications sont:

- le nombre d'emboîtements et
- le nombre de référenciels distincts.

Comme l'expression des formules à plusieurs variables libres apparaît *a priori* utile, il faudra le permettre dans l'écriture des requêtes sélectives.

### 6.1.6. Paramétrage et factorisation des requêtes

Un autre problème est de choisir une technique de paramétrage et de factorisation pour communiquer à une requête ses arguments: *la collection, le contexte, le critère et sa variable libre.*

De manière idéale, il faudrait pouvoir opter entre:

- un paramétrage statique avec contrôles de types réalisables dès la compilation,

- un paramétrage dynamique avec contrôles de types effectués pendant l'exécution,

- une factorisation de tous les éléments textuels communs à plusieurs requêtes: partage de contextes ou de collections, voire de critères.

La factorisation des collections et la composition des requêtes suggèrent, comme nous l'avons vu à le paragraphe 6.1.2, de considérer les requêtes comme des primitives de collections. Il ne resterait alors à leur communiquer que leur contexte et leur critère, mais en remarquant que les cas de factorisation du contexte seront beaucoup plus fréquents que celui du critère. Cependant, dans le but d'avoir une étude la plus complète possible, nous nous pencherons aussi sur le partage du moteur du langage de requête.

Comme on peut envisager des adjonctions syntaxiques (définition d'un langage spécifique: *EQL,* ou de nouvelles structures de contrôle), et la description de composants en Eiffel, on dispose des solutions suivantes:

- Avec un langage *EQL* ou une structure de contrôle itérative ad hoc, on pourrait décrire pour chaque requête les éléments spécifiques du critère et du contexte local. Pour factoriser le contexte, il suffirait d'utiliser celui fourni par la classe et la routine englobante. Quant à la factorisation du critère, il suffirait de le placer dans une fonction booléenne de la classe englobante et d'appeler cette fonction dans la clause de critère (solution de le paragraphe 6.2.1).

- En Eiffel pur**,** on ne peut décrire le contexte que dans une classe et le critère que dans une fonction booléenne de cette même classe ou de l'une de ses descendantes. Deux possibilités s'offrent alors pour transmettre ces arguments:

  - *transmettre globalement le critère et son contexte*, soit en transmettant une instance de la classe du contexte en argument de la requête - mais alors, avec un seul critère possible -, soit en effectuant un héritage simple ou répété, avec des renommages et des redéfinitions pour distinguer et décrire les différents critères (solution de le paragraphe 6.3.2).
  - *transmettre séparément le critère et son contexte*. Eiffel ne permet actuellement

que d'encapsuler le critère dans une classe et de transmettre l'une de ses instances à la requête (solution de le paragraphe 6.3.1). Mais on pourrait songer à modifier Eiffel, pour permettre la transmission de routines paramétriques, comme cela existe en C ou en Pascal (solution de le paragraphe 6.3.3).

**Toutes ces possibilités de paramétrage et de factorisation des arguments des requêtes sont donc au cœur des choix à effectuer pour les exprimer**.

### 6.1.7. Approches possibles

En définitive, nous disposons des moyens linguistiques suivants:

• Définir un langage spécifique de requêtes *EQL*, avec plusieurs variantes possibles:

-*intégrer EQL au langage Eiffel*, avec plusieurs niveaux possibles: quantifications intégrées aux expressions logiques et aux assertions, syntaxe particulière pour seulement les critères ou pour toutes sortes de sélections et d'itérations.

-*ne pas intégrer EQL à Eiffel*, mais évaluer les requêtes par un interprète spécialisé, plus simple qu'un interprète complet du langage.

• Ajouter un itérateur. Deux types de solutions sont possibles compte tenu du caractère spécifique de la structure de contrôle.

- la structure pourra être utilisée telle quelle pour construire des composants spécifiques (même approche que la boucle *from-until-loop*); cette approche est générale et orthogonale, mais fastidieuse s'il faut écrire un grand nombre de requêtes.

- adopter une approche similaire à la classe ITERATION, mais en utilisant le nouvel itérateur, à la place de la boucle *from-until-loop*.

• Utiliser Eiffel tel qu'il est (V.3. . . ) en utilisant les mécanismes existants, notamment les liens d'héritage et de clientèle. Plusieurs variantes sont également possibles, pour produire ou utiliser les classes:

- *les classes sont écrites à la main et compilées:* cette solution est nécessaire pour les interfaçages les plus variés avec le monde persistant, mais s'avère peu pratique

pour exécuter les requêtes qui émanent d'utilisateurs interactifs;

- *les classes sont construites automatiquement par un outil:* à partir d'un langage graphique ou textuel comme *EQL*, mais utilisé dans un environnement interactif. Les classes produites peuvent alors être soit compilées, soit interprétées par un interprète du langage Eiffel complet. Une expérience menée à l'I3S [Brissi 90] montre qu'il est possible d'interpréter des classes Eiffel dans un contexte mixte de classes compilées ou interprétées. On pourrait donc se contenter de n'interpréter que les classes contruites pour exprimer les requêtes. On notera que cette approche pourra être aussi utilisée pour la 2ème solution de l'approche *EQL*.

• Etendre Eiffel et son environnement pour accroître ses possibilités de paramétrage dynamique, soit en étendant les possibilités génériques, soit en permettant plus de souplesse dans les listes d'arguments de routines: transmission directe de routines (*routines paramétriques ou routines-arguments).* On utilisera les améliorations apportées, dans ce domaine, par la version 3 du langage.

Ces quatre grands types de solutions sont développés dans les trois sections suivantes. Comme la conception du langage EQL est libre de toute contrainte d'Eiffel, on devrait obtenir dans ce langage l'expressivité maximum, ce qui servira de référence pour le comparer avec les autres approches.

On notera qu'une approche basée sur le passage d'une chaîne de caractères [Lahire 91], exprimant le critère de sélection ou l'action à exécuter a été abandonnée car elle s'intègre fort peu au langage: par exemple, les vérifications de types sont faites dynamiquement et une erreur entraîne la recompilation de l'application.

# 6.2. Intégration des requêtes dans la syntaxe

## 6.2.1. Construction d'un langage de requêtes: EQL

Nous définirons successivement les sélections, les itérations sélectives et les quantifications, en donnant à chaque fois quelques exemples d'illustration.

**Sélections**

Nous gardons le caractère fonctionnel de la syntaxe vue précédemment pour

permettre des compositions de sélections ou d'itérations successives, puisque c'est une possibilité intéressante du langage Eiffel. Pour le contexte global, nous prenons celui de la routine et de la classe qui englobent chaque requête. Le contexte local est placé dans la requête elle-même, en reprenant toutes les clauses de la syntaxe d'une classe Eiffel a *priori* utiles: *Indexing, Class_header, Formal_generics, Exports*. A ces différences près, la grammaire proposée (figure 6.5) ressemble à celle de SQL [Miranda 89] ou de 02-Query [Bancilhon 89, $O_2$-Technology 91 a].

```
selective_query  =  selection | selective_iteration | quantification
selection        =  collection.select  freevar_ident
                                          criteria
                                        local-context
                                    end
collection       =  collection_ident | selection
criteria         =  [ suchas criteria ]
local-context    =  [ inherit Parent_list ]
                      [ external {Externallist ";".... } ]
                      [ feature {Feature_declaration ";". . } ]
                      [ rescue Compound ]
```

Figure 6.5: Définition des sélections en EQL.

L'identificateur de variable libre *freevarident* permet de choisir un nom plus approprié que x. Le critère est donné sous la forme d'une *formule bien formée* vue à le paragraphe 3.3.3 mais en autorisant l'usage du ";" à la place d'un "*and,*" ou d'un "*and then*", comme dans le langage d'assertions d'Eiffel.

Avant de voir le premier des exemples, rappelons que leur intérêt est purement illustratif et qu'ils ne prétendent en aucune façon être réalistes: on supposera toujours que toutes les informations nécessaires à leur exécution sont disponibles dans le monde persistant.

Exemple de sélection simple:

**C**onsidérons les sélections suivantes:

- "Sélectionner les 4 pièces à vendre, d'un prix compris entre 800MF et 1.2MF, situés dans le quartier Cimiez de la ville de Nice".
- "Sélectionner les 3 pièces à louer de la ville d'Antibes".

Le programme 6.6 en donne une première traduction en *EQL*, avec trois versions de définition de contexte: *local explicite, local hérité ou global.* Ce contexte définit des valeurs comme *to_sell* et *Nice* qui sont supposées être ici des codes symboliques entiers.

Dans le cas d'un contexte explicite placé dans une clause *feature* globale ou locale, il faut créer les instances des classes qui exportent les valeurs nécessaires. Nous avons choisi de placer ces créations dans des fonctions *once,* mais on aurait pu tout aussi bien utiliser des variables initialisées par une unique procédure de création. Le cas d'un contexte global fournit évidemment les notations les plus simples; l'emploi de l'héritage accentue encore la simplification de l'écriture.

```
class FLAT_SELECTIONS                -- version 1
inherit    FLAT; CITIES              -- global context: provides Nice, Antibes, tolet. . .
creation   create
feature

  fc, fsl, fs2: COLLECTION [ FLAT ];

  create is
     do    !!fc.create;                                  -- . . . filling of fc
           version 1 _1;    versionl_2;    version 1 _3;
     end;  -- create

  version1_1 is
        -- version with explicit local context
     do fsl:= fc.select f suchas
                        f.rooms_nb = 4; f.status = to_sell; 800_000 <= f.price <= 1_200_000;
                        f.district = Cimiez; f.city = Nice
                feature
                        to_sell: INTEGER is
                            local x: FLAT;
                            once x.create; Result:= x.to_sell end;
                        Nice: INTEGER is
                            local x: CITIES
                            once x.create; Result:= x.Nice end;
                end;
     end;  -- version1_1

  version1_2 is
        -- version with inherited local context
     do fsl:= fc.select f suchas
                        f.rooms_nb = 4; f.status = to_sell; 800_000 <= f.price <= 1_200_000;
                        f.district = Cimiez; f.city = Nice
                inherit FLAT; CITIES
                end;
     end;  -- version1_2

     version1_3 is
        -- version with inherited global context
     do fsl:= fc.select f suchas
                        f.rooms_nb = 4; f.status = to_sell; 800_000 <= f.price <= 1_200_000;
                        f.district = Cimiez; f.city = Nice
                end;
        fs2:= fc.select f suchas
```

```
                        f.rooms_nb = 3; f.status = to_let; f.city = Antibes
                end;
    end;  -- version1_3

end-- class FLAT_SELECTION
```

Figure 6.6: Sélections avec différents contextes (EQL).

## Variable libre: implicite ou explicite?

La nature même du critère de sélection oblige à citer souvent les caractéristiques de l'objet à sélectionner, et donc le nom de la variable libre dans les notations pointées. Ceci amènera sûrement les programmeurs à choisir des noms de variables libres courts, voire réduits à une lettre, ce qui affaiblit l'intérêt du choix des variables libres pour la lisibilité.

On peut donc songer à s'en débarasser, en supposant par exemple que toute caractéristique non pointée est implicitement préfixée par le nom de l'objet candidat à une sélection. Dans le cas où une routine nécessiterait en argument de citer ce candidat, on pourrait utiliser un mot réservé comme *Self* (puisque *Current* dèsigne déja la collection en cours de sélection) .

Cependant, cela conduirait à des risques de confusions avec les caractéristiques locales, qui n'ont pas davantage de préfixe explicite, comme le montre le programme 6.7 . Nous écartons donc cette simplification.

```
.......
  version1_4 is
        -- Version avec variable libre implicite
    do fsl:= fc.select ?? suchas
                        rooms_nb = 4; status = to_sell; 800_000 <= price <=
1_200_000;
                        district = Cimiez; city = Nice
                end;
    end;  -- version1_4
.......
```

Figure 6.7: Sélection avec variable libre implicite (non EQL).

## Itérations sélectives

En plus du contexte et du critère de sélection, il faut écrire la primitive à itérer qui peut être:

• une action modifiant l'état de chaque objet sélectionné, décrite par une procédure,

• ou une consultation de chaque objet, représentée par une fonction, sans effet de bord.

Pour définir la routine à activer, on doit considérer les cas suivants:

• l'activation d'une routine exportée par les objets à sélectionner: dans ce cas, la notation "*freevarident.routine-call*" explicite à la fois la variable libre et la syntaxe d appel;

• l'activation de plusieurs routines, éventuellement conditionnées par des critères complémentaires, pour bénéficier de la factorisation d'une traversée. Pour décrire ces actions on peut envisager:

- soit d'intégrer la syntaxe d'un corps de routine dans celle de la requête,

- soit de permettre à celle-ci d'activer une routine du contexte, local ou global, ayant au moins la variable libre comme argument.

Cette dernière solution nous paraît préférable car elle permet de factoriser le même traitement sur diverses sélections et simplifie la syntaxe du langage EQL: **nous la retiendrons** (figure 6.8). Toutefois, pour homogénéiser les notations, nous utiliserons la clause *with* qui explicite la variable libre dans tous les cas, même pour le premier où elle n'est pas nécessaire.

```
selective_iteration     = collection.do
                                iterative-routine
                                criteria
                                local-context
                              end

iterative-routine       = [ freevarident.routine-call  with freevar_ident ]
```

Figure 6.8: Définition des itérations sélectives en EQL.

Exemple d'itération sélective simple

Considérons l'exemple suivant: *augmenter le salaire de 3.5% de tous les salariés d'une entreprise qui ont plus de vingt ans d'ancienneté.*

En supposant:

• que la classe *WAGE_EARNER* exporte l'attribut *admission_date: DATE* et les caractéristiques *set_salary* et *salary*,

• que la classe *DATE* fournisse des opérations de comparaisons entre dates et la fonction *to_year* de conversion en année,

• et que la classe *SYSTEM* donne la date du jour par *current_date*,

le programme (fig. 6.9) donne deux versions pour l'itération sélective souhaitée avec un contexte global.

```
class ACTIONS
inherit    DATE; SYSTEM        -- context
creation   create
feature

  company: COLLECTION [WAGE_EARNER];

  create is
     do
        !!company.create; - - filling of company
                  -- direct version:
        company.do
                     senior.set_salary(senior.salary * 1.035) with senior
                        suchas senior.to_year(current_date -senior.admission_date) >= 20
                     end;
                  -- version with local function:
        company.do
                     senior set_salary(inc(senior.salary, 3.5)) with senior
                        suchas senior.to_year(current_date - senior.admission_date) >= 20
                     feature
                        inc(value, percent: REAL): REAL is
                           do Result:= value * (1. + percent / 100.) end
                  end;
     end;-- create

end -- class ACTIONS
```

Figure 6.9: Action itérative simple (EQL).

Exemple de composition d'itérations sélectives

Nous avons déja vu à le paragraphe 6.1.2 l'intérêt (et aussi les risques éventuels sur l'efficacité) des compositions de sélections, ou des compositions de sélections et d'itérations sélectives.

Reprenons l'exemple des anciens salariés et supposons maintenant que l'on veuille en plus *donner une prime annuelle de 5 OOOF aux vétérans de plus de 30 ans de maison*. Le programme (fig. 6.10) donne deux solutions en EQL version 2. Il montre aussi la factorisation de la fonction *seniority* qui est déclarée au niveau du contexte global et qui est utilisée trois fois dans la première version, avec trois requêtes successives.

```
class NESTED_QUERIES
inherit     DATE; SYSTEM        -- context
creation    create
feature
      company: COLLECTION [ WAGE_EARNER ];
      seniors: COLLECTION [ WAGE_EARNER ];
      seniority (s: WAGE_EARNER): REAL is
            do Result:= to_year (current_date - s.admission_date) end; -- seniority

      create is
      do
         !!company.create; - - filling of company
-- three step version:
         seniors:= company.select x suchas seniority(x) >= 20 end;
         seniors.do s.set_salary (senior.salary * 1.035) with s end;
         seniors.do veterant.set_bonus (5000) with veterant suchas seniority (veterant) >=
30 end;

-- compact version: the most performant:
         company.do reward(x) with x suchas seniority (x) >= 2O
                     feature
                           reward(s: WAGE_EARNER) is
                              do    s.set_salary (v.salary * 1.035);
                                    if seniority (s) >= 30 then s.set_bonus (5OOO) end;
                               end -- reward
                     end;
      end;-- create

end-- class NESTED_QUERIES
```

Figure 6.10: Composition de requêtes (EQL).

## Expression des quantifications

L'expression des quantificateurs en EQL ne pose pas de problèmes majeurs comme le montre la syntaxe proposée sur la figure 6.11 .

```
exists   = collection.exists freevar_ident
                             criteria
                             local-context
                         end

forall   = collection.forall freevar_ident
                             criteria
                             local-context
                         end

count    = collection.count freevar_ident
                             criteria
                             local-context
                         end
```

Figure 6.11: Définition des quantificateurs en EQL.

```
class EXISTS
inherit
        CITIES        -- context
creation
        create
feature
  students: COLLECTION [ ETUDIANT ];

  create is
     do
        !!students.create; -- filling of students
        io.putbool (students.exists s suchas s.age > 60 end);                -- case 1
        students.do io.putstring (s.name) with s suchas s.age = 20 end       -- case 2
        students.select x suchas                                             -- case 3
                            x.adress.city = Nice;
                            exists y suchas      -- students.exists y suchas .....
                                  y.adress.city = Paris; y.name = x.name
                            end;
                    end;
     end;  -- create

end-- class EXISTS
```

Figure 6.12: Quantifications existentielles (EQL).

Exemples de quantifications existentielles

Donnons quelques exemples de quantifications, d'abord simples, puis avec emboîtements:

1. "dire s'il existe un étudiant de plus de 60 ans",

2. "imprimer le nom de tous les étudiants de 20 ans",

3. "quels sont les étudiants niçois qui ont le même nom qu'un étudiant parisien",

Le programme (fig. 6.12) en donne une traduction en EQL. On remarque que dans le troisième exemple avec emboîtement, la notation "*exists - end*" sous-entend "*Current.exists ... end*", c'est-à-dire la collection en cours de sélection.

Exemples de quantifications universelles et cardinales

Considérons la requête: *"afficher tous les appartements à louer à Antibes, tels que leurs voisins n'aient pas de piano et aient au plus deux enfants de moins de 8 ans"*. Une écriture possible de cet exemple fantaisiste est donnée par le programme (fig. 6.13). Ce programme donne aussi un exemple d'emboîte ment à trois niveaux, avec une hétérogénéité des types des objets intervenant dans les sélections et montre l'emploi de *count* dont l'usage a été déconseillé à le paragraphe 6.1.2 pour des raisons d'efficacité. Ici, la cardinalité qui nous intéresse n'est ni celle d'une collection vide (ce qui serait fourni par *exists*), ni celle d'une collection complète (ce qui s'obtient avec un *forall*): son utilisation est donc nécessaire.

```
class NESTED_SELECTIONS          -- version 1
inherit    FLAT; CITIES          -- context
creation   create
feature
  fc: COLLECTION [ FLAT ];

  create is
     do !!fc.create;    -- filling of fc
        fc.do x.display with x suchas
                              x.status = to_let; x.city = Antibes;
                              forall y suchas
                                    x.adress = y.adress and x /= y implies
                                    not y.has_piano;
                                    not y.children.count z suchas z.age < 8 end <= 2:
                              end;
```

```
            end;
      end;  -- create
end-- class NESTED_SELECTIONS
```

Figure 6.13: Quantifications universelles et cardinales (EQL).

Quant au programme (fig. 6.14), il traduit en EQL la spécification de la structure algébrique de groupe déja vue sur le programme 3.1. Il est clair que l'évaluation de l'invariant de cette classe n'a d'intérêt que pour:

• tester l'exactitude de sa formulation, c'est-à-dire s'assurer que la documentation est conforme à la réalité,

• tester l'intégrité d'une collection d'objets en vérifiant à des instants adhoc q les instances d'une classe vérifient une certaine condition.

```
deferred class GROUP
feature
   zero: like Current is deferred end;
   infix "+" (other: like Current): like Current is deferred end;
   prefix "-": like Current is deferred end;
   value: VALUE; -- value of the current object
   set (new: VALUE) is do value:= new end;

invariant -- class invariant
   -- associativity:
      pcollection.forall x
                  suchas forall y suchas forall z suchas (x + y) + z = x + (y + z) end end end;

   -- neutral_element:
      pcollection.forall x suchas zero + x = x and x + zero = x end

   -- symetric element:
      pcollection.forall x suchas forall y suchas x + -x = zero end end;
end - - class GROUP
```

Figure 6.14: Spécification quasi-formelle d'un groupe (EQL).

## Syntaxe complète du langage *EQL*

Pour terminer cette définition d'*EQL*, la figure 6.15 reprend toutes les définitions des requêtes sélectives en *EQL*. Les exemples donnés précédemment montrent qu'*EQL* a une bonne expressivité, sur un éventail assez large de situations et avec emboîtements de quantifications. On remarque aussi la souplesse apportée par le fait que l'on peut combiner les emboîtements et les compositions de requêtes et choisir librement les factorisations les plus adéquates pour les contextes et les critères.

```
selective_query      =  selection | selective_iteration | quantification
selection            =  collection.select freevar_ident
                                criteria
                                local-context
                            end
selective_iteration  =  collection.do
                                iterative-routine
                                criteria
                                local-context
                            end
iterative-routine    =  [ freevarident.routine-call with
                        routine-call with freevar_ident ]
quantification       =  exists | forall | count
exists               =  collection.exists freevar_ident
                                criteria
                                local-context
                            end
forall               =  collection.forall freevar_ident
                                criteria
                                local-context
                            end
count                =   collection.count freevar_ident
                                criteria
                                local-context
                            end
criteria             =  [ sucha s criteria ]
local-context        =  [ inherit Parent_list ]
                        [ external {Externallist ";". . . } ]
                        [ feature {Feature_declaration ";". . . } ]
                        [ rescue Compound ]
```

Figure 6.15: Définition complète du langage EQL

## 6.2.2. Adjonction d'un itérateur externe

Nous proposons une autre approche syntaxique permettant de prendre en compte le cas où les itérations sélectives ne sont pas intégrées dans le type collection (fig. 6.16).

```
selective_iteration  =  from [routine-body]
                        loop freevarident. on varident
                        [when criteria do routine-body ]   -- inversement possible
                        end

routine-body = -- contenu de la clause loop d'un schema from-until-loop-end
```

Figure 6.16: Définition des itérations sélectives à travers des itérateurs simples

Dans cette description, il faut ajouter que *varident* doit être une variable dont le type est conforme à une collection d'objet d'un type T, et que *freevarident* doit avoir un type conforme à T.

```
.......
nom_routine: COLLECTION [FLAT] is
  local
     -- contexte local a la routine peut inclure une clause inherit conformément à EQL
  do
     ..... Contenu habituel d'une routine
     from  c.start;
     loop i                              -- i attribut de type FLAT
     on c                                -- c attribut de type COLLECTION [FLAT]
     when i.age > 10
     do
          from d.start;
          loop j                         -- j est un attribut de type PERSON
          on d                           -- d est un attribut de type COLLECTION [PERSON]
          when i.address = j.address and d.age < 40
          do
              if not Result.has (i) then Result.put (i) end
          end
     end

  ..... Contenu habituel d'une routine

  end -- nom_routine
```

Figure 6.17: Exemple de requête emboitée par itérateur simple

Les principaux avantages de cette approche sont:

• limiter le nombre de mots-clés à rajouter,

• garder la philosophie du langage,

• offrir une structure supplémentaire permettant de modéliser n'importe qu'elle combinaison d'itérations sélectives, incluant les itérations emboîtées,

• offrir un support pour une gestion optimisée des itérations sélectives; il est en effet difficile de reconnaître à la traduction les cas de sélection dans un schéma **from-until-loop-end.**

Les inconvénient concernent:

• l'impossibilité de réaliser directement des compositions de sélection (il faut alors déclarer des fonctions contenant des itérateurs),

• fournir seulement des itérateurs (*one-at-a-time operator*) et non pas un vrai mécanisme de sélection (*set-at-a-time operator*) avec une approche déclarative,

• et surtout introduire une redondance entre les boucles **from-until-loop-end,** et **from-loop-on-when-end**; conceptuellement en effet, tout schema **from-loop-on-when-end,** peut être décrit par un schéma **from-until-loop-end.**

Dans la figure suivante (fig. 6.17) on donne un exemple de sélection itérative emboîtée permettant de retourner *tous les appartements de moins de 10 ans dont au moins un de ses occupanta moins de 40 ans.*

## 6.3. Intégration des requêtes par composants Eiffel V. 3

Il n'y a pas de problème pour décrire tous les arguments d'une requête en Eiffel, on a tout ce qu'il faut:

• la *collection* se décrit naturellement par une instance,

• la *variable libre* par un argument formel du type des éléments de la collection,

• le *critère* par une fonction booléenne ayant la variable libre en argument,

• la *routine* à activer par une routine ayant également la variable libre en argument,

• et le *contexte* par une classe accessible au critère et à la routine à activer.

Le problème principal est celui du paramétrage signalé à le paragraphe 6.1.6: comment atteindre commodément tous ces arguments, tout en offrant toutes les possibilités de variations et de factorisations souhaitables? Dans une solution en Eiffel pur compilé, il faut que la définition du critère ou de la routine à itérer soit dans la classe qui définit le

contexte, localement ou par héritage. Si l'on veut garder des possibilités de compositions de requêtes, celles-ci doivent être des primitives de la classe *COLLECTION*. Ce problèmeest donc général:

*"Comment communiquer une routine définie dans une classe X, à une primitive d'une autre classe?"*

Eiffel n'offre actuellement que deux moyens pour cela:

• **l'encapsulation** des routines à transmettre dans une classe qui définit le contexte, et dont l'une de ses instances est transmise à la requête. Celle-ci atteint la routine attendue par une relation de *"clientèle"*.

• **l'héritage** avec redéfinitions et renommages.

Comme nous le verrons ensuite, aucune de ces deux solutions n'est vraiment satisfaisante, ce qui nous amènera à envisager une *troisième solution* à base de routines paramétriques, ce qui suppose de légères modifications avec une compatibilité ascendante avec l'environnement actuel.

### 6.3.1. Solutions à base d'encapsulation de routines dans des classes

Toutes requêtes sélectives reçoivent leur critère en argument, par l'intermédiaire d'un objet, instance d'une classe construite spécialement pour définir le critère. On profite de cette occasion pour définir dans cette classe le contexte, qui peut être factorisé par héritage.

Pour permettre les contrôles de types, on utilise une classe *QUERY_ARG*, qui fournit le modèle commun à toutes les classes qui définissent un critère *criteria* et/ou une routine à itérer *action*. Nous avons choisi une classe non retardée pour ne pas nécessiter de définitions inutiles: les classes descendantes ne redéfiniront que le strict nécessaire. La généricité n'est utile ici que pour contrôler la conformité des types des objets sélectionnés, mais ce n'est pas nécessaire pour le fonctionnement de la technique. S'il le faut, on pourra développer des solutions à généricité plus dynamique, voire sans contrôle, par polymorphisme.

Le programme (fig. 6.18) donne un aperçu des classes *COLLECTION* et *QUERY_ARG*.

La classe *COLLECTION* fournit des algorithmes simplifiés pour les collections en mémoire volatile. Dans le cas de collections persistantes, le code fera appel aux primitives du POM.

Les routines avec un nombre variable d'arguments (Eiffel 3.1), sont utilisées pour traiter les cas de requêtes emboîtées.

```
class COLLECTION [ ITEM -> ANY ]
feature

   select(q: Q UERY_ARG[ ITEM ]; args: ARRAY [ANY]): like Current is
         -- return all item i in Current satisfying q.criteria(i, args)

   do_if (q: QUERY_ARG [ITEM ]; select_args, action_args: ARRAY [ANY]) is
         -- apply q.action(i, action_args) to every item i of Current
         -- satisfying q.criteria(i, select_args)

   do_all (q: QUERY_ARG [ ITEM ]; args: ARRAY [ANY]) is
         -- apply q.action(i, args) to every item i of Current

   exists (q: QUERY_ARG [ITEM]; args: ARRAY [ANY]): BOOLEAN is
         -- is q.criteria(i, args) for at least one item i of Current?

   all (q: QUERY_ARG [ITEM]; args: ARRAY [ANY]): BOOlEAN is
      -- is q.criteria(i, args) for all item i of Current?

   count (q: QUERY_ARG [ ITEM ]; args: ARRAY [ANY]): INTEGER is
      -- number of item i of Current satisfying q.criteria(i, args)

   forth is          -- move cursor to next position

   start is          -- move cursor to first position

   item: ITEM is     -- return item under cursor

end -- class COLLECTION
_____________________________________________________________________
class QUERY_ARG [ ITEM -> ANY ]
feature

   criteria (i: lTEM; args: ARRAY [ANY]): BOOLEAN is
      -- criteria of a selective query

   action (i ITEM; args: ARRAY [ANY]) is
      -- action on a selective query

end -- class QUERY_ARG
```

Figure 6.18: Requêtes selectives clientes de leurs routines paramétriques (Eiffel).

Exemple de sélections simples

Le programme (fig. 6.19) donne l'équivalent en Eiffel V.3 du programme (fig. 6.6) en EQL. Si l'on veut utiliser différents critères pour des sélections ou d'autres primitives sélectives, il faudra construire autant de classes que de critères...

```
class FLAT_SELECTIONS                                   -- version 2.1
creation    create
feature
  fc, fsl, fs2: COLLECTION [ FLAT ];
  ql: QUERYARGl; q2: QUERYARG2;
  . . .
  create is
      do
         !!ql.create; !!q2.create;
         ....                              -- remplissage de fc
         fsl:= fc.select(ql, << >>);
         fs2:= fc.select(q2, << >>);
      end; -- create

end-- class FLAT_SELECTIONS
_____________________________________________________________________
class QUERY_ARGl
inherit
         FLAT;     CITIES;                      -- provides context: to_sell, Nice, Cimiez
         QUERY_ARG [FLAT]
            redefine criteria
feature
  criteria(f: FLAT, args: ARRAY [ANY]): BOOLEAN is
      do
         Result:= f.rooms_nb = 4 and f.status = to_sell and 800000 <= f.price <= 1200000 and
                  f.district = Cimiez and f.city = Nice
      end -- criteria

end - - class QUERY_ARG 1
_____________________________________________________________________
class QUERY_ARG2
inherit
         FLAT;     CITIES;                          -- provides context: to_let, Antibes
         QUERY_ARG[FLAT]
            redefine criteria
feature
  criteria(f: FLAT, arg: ARRAY [ANY]): BOOLEAN is
      do
         Result:= f.rooms_nb = 3 and f.status = to_let and f.city = Antibes
      end    -- criteria

end -- class QUERY_ARG2
```

Figure 6.19. Sélections simples par encapsulation de routines (Eiffel).

On peut toutefois légèrement diminuer le nombre de classes à écrire, si l'on n'a besoin que d'un critère et d'une routine itérative au plus: il suffit de fusionner la classe

*QUERY_ARGl* dans la classe *FLAT_SELECTIONS* par un héritage et de transmettre "*Current*" comme argument effectif de "*select*" (programme, fig. 6.20).

```
class FLAT_SELECTIONS                          -- version 2.2
inherit
    FLAT;    CITIES;                           -- provides context: to_sell, Nice, Cimiez
    Q UERY_ARG[FLAT]
        redefine criteria
creation
    create
feature
  fc, fsl, fs2: COLLECTION [ FLAT];
  q2: QUERY_ARG2;
  .....
  criteria(f: FLAT, args: ARRAY [ANY]): BOOLEAN is
    do
      Result:= f.rooms_nb = 4 and f.status = to_sell and 800000 <= f.price <= 1200000 and
               f.district = Cimiez and f city = Nice
    end; -- criteria

  create is
    do
      !!ql.create; !!q2.create;
      .....                                    -- remplissage de fc
      fsl:= fc.select(Current, << >>);
      fs2:= fc.select(q2, << >>);
    end   -- Create

end-- class FLAT_SELECTIONS
_____________________________________________________________________
class QUERY_ARG2
inherit
    FLAT;    CITIES;                              -- provides context: to_let, Antibes
    QUERY_ARG[FLAT]
        redefine criteria
feature
  criteria(f: FLAT; args: ARRAY [ANY]): BOOLEAN is
    do
        Result:= f.rooms_nb = 3 and f.status = to_let and f.city = Antibes
    end   -- criteria

end -- class QUERY_ARG2
```

Figure 6.20: Sélections simples par encapsulation et héritage des routines (Eiffel).

Exemple d'emboîtement de requêtes

L'emboîtement de requêtes nécessite par contre de transmettre de l'information supplémentaire aux itérations internes, comme la référence à la collection en cours de sélection, ou à l'objet candidat à une sélection, ou à d'autres objets de types différents dans les cas complexes. Le programme (fig. 6.21) est l'équivalent Eiffel du programme 6.13 donné en EQL.

```
class NESTED_SELECTIONS                          -- version 2
creation
        create
feature
  fc: COLLECTION[FLAT];
  q3: QUERY_ARG3;

  create is
     do
        !!q3.create;
        .....        -- remplissage de fc
        fc.do_if (q3, <<fc>>);          -- apply q3.action (i,<< fc >>) to every item i of fc
                                        -- satisfying q3.criterial(i, << fc >>)
     end   -- create

end -- class NESTED_SELECTIONS
_____________________________________________________________________
class QUERY_ARG3
inherit
        FLAT;    CITIES;                        -- context
        QUERY_ARG[FLAT]
           redefine criteria, action
creation
        create
feature
     q4: QUERY_ARG4;

     action (f: FLAT, args: ARRAY [ANY]) is
        do
           f.display                            -- args n'est pas utilisé
        end;   -- action

     criteria (x: FLAT, args: ARRAY [COLLECTION [FLAT]]): BOOLEAN is
              -- is q4.criterial(i,x) for every item i of args @ 1?
        do
           Result:= x.status = to_let and x.city = Antibes and
                    args.item (1).all (q4, << x >>);
        end;   -- criteria

     create is do !!q4.create end   -- Create

end - - class QUERY_ARG3
```

```
class QUERY_ARG4
inherit
        FLAT; CITIES;
        QUERY_ARG[FLAT]
            redefine criterial
creation    create
feature
    q5: QUERY_ARG5;

    criterial(y: FLAT, args: ARRAY [ANY]): BOOLEAN is
        do
            Result:= x.adress = args.item (1).adress and args.item (1) /= y implies
                     not y.has_piano  and y.rooms.count(q5, << >>) <= 2
                          -- number of item i of y.rooms satisfying q5.criteria(i, << >>)
        end;  -- criteria

    create is do !!q5.create end -- Create

end -- class QUERY_ARG4
_____________________________________________________________________
class QUERY_ARG5
inherit
        FLAT; CITIES;
        QUERYARG [PERSON]
            redefine criteria
feature

    criteria(x: PERSON, args: ARRAY [ANY]): BOOLEAN is
        do
            Result:= x.age < 8
        end    -- criteria

end -- class QUERY_ARG5
```

Figure 6.21: Emboîtement de requêtes par relation de clientèle (Eiffel).

La complexité de l'écriture (et donc de la lecture) de cette solution en Eiffel se passe de commentaires, si on la compare à celle en EQL. Plus grave, les itérations emboîtées sont dissociées, comme dans le cas de composition de requêtes. Dans ces conditions, il doit être difficile de réaliser des optimisations, même en comptant sur l'habileté du POM.

On notera dans la figure 6.21, qu'en réalité, les primitives communes à la classe ITERATION et à la classe COLLECTION sont décrites en utilisant un héritage (héritage d'implémentation) de la classe LINEAR_ITER de la bibliothèque Eiffel, mais que pour des raisons de clarté (indépendance spécification / implémentation), il a semblé préférable de décrire le code explicitement.

### 6.3.2. Solutions à base d'héritages répétés

Ce type de solution, qui est utilisé dans les classes d'itérations de la bibliothèque Eiffel V.3, définit dans la même classe les requêtes sélectives et les routines paramétriques: critère ou action à itérer. Comme chaque requête sélective a un algorithme de traversée des collections, commun à tous les cas d'emploi, chacune est programmée une bonne fois pour toutes, mais en faisant directement appel dans son code à sa routine-argument, sans transmission dynamique. Pour adapter ces routines appelées à chaque cas, on utilise ici les possibilités de renommage et de redéfinition.

Cependant, si une requête $\Re$ appelle directement dans son code un argument C susceptible d'être renommé et redéfini en plusieurs routines *Cl, C2 ... Cn*, il faut bien distinguer les variantes $\Re 1, \Re 2 \ldots \Re n$ de $\Re$ qui déclenchent l'appel voulu à l'une des routines Ci. Ceci peut être obtenu en renommant - et donc en dupliquant - les routines appelantes, lors d'héritages répétés n fois de la classe qui les définit, pour chacune des *n* occurences des routines appelées.

D'autre part, il n'est plus possible de garder la définition des requêtes sélectives dans la classe COLLECTION: il faut ici transmettre à chaque requête la collection à traverser, car son code est déja lié au critère ou à l'action à itérer. On laisse cependant les primitives de traversées (*item, forth, off*...) dans les collections.

Les requêtes sont donc placées dans une nouvelle classe *SELECTION* (programme fig. 6.22) qui ajoute à l'implémentation de ces requêtes une définition neutre du critère *criteria* et de la routine à itérer *action*, lesquelles seront redéfinies pour chaque besoin.

Comme dans la version précédente, les routines à nombre variable d'arguments sont utilisées pour les requêtes emboîtées.

```
class SELECTION [ ITEM -> ANY ]
feature
  collection: COLLECTION [ ITEM ];    - - anchor

  select (c: like collection, args: ARRAY [ANY]): like collection is
          -- return all item i of c satisfying criteria(i, args)

  do_if(c: like collection, select_args, action_args: ARRAY [ANY]) is
          -- apply action(i, action_args) to every item i of c
          -- satisfying criteria(i, select_args)
```

```
  do_all(c: like collection; args: ARRAY [ANY]) is
            -- apply action(i, args) to every item i of c

  exists(c: like collection; args: ARRAY [ANY]): BOOLEAN is
            -- is criteria(i, args) for at least one item i of c?

  all(c: like collection; args: ARRAY [ANY]): BOOLEAN is
            -- is criteria(i, args) for all item i of c?

  count (c: like collection, args: ARRAY [ANY]): INTEGER is
          -- number of item i of c satisfying criteria(i, args)

  criteria(i: ITEM): BOOLEAN is
            -- Criteria to apply on item i

  action (i: ITEM) is
            -- action to apply on item i

  start (c: like collection) is
            -- move cursor to first position

  forth (c: like collection) is
            -- move cursor to next position

  off (c: like collection): BOOLEAN is
            -- Is cursor off?

end -- class SELECTION
_____________________________________________________________________
class COLLECTION [ITEM -> ANY]
feature
  item: ITEM
          -- Current item of collection

  off: BOOLEAN
          -- Is there not a current item

end  -- class COLLECTION [ITEM]
```

Figure 6.22: Requêtes sélectives dans la même classe que leurs arguments (Eiffel).

Exemple de sélections simples

Le programme (fig. 6.23) donne l'équivalent avec cette technique par héritage du programme (fig. 6.19) qui utilisait l'encapsulation. Si l'on veut utiliser différents critères pour des sélections ou d'autres primitives sélectives, il ne faut plus construire autant de classes que de critères, mais seulement faire des héritages répétés.

Cependant, on a perdu la possibilité de composer les requêtes sélectives. puisqu'elles ne sont plus des primitives de collection. On peut aussi s'interroger sur la duplication

possible du code des requêtes renommées: sur l'exemple, select qui appelle criteria a un code identique à sel2 qui appelle crit2, à la seule différence de l'appel au critère.

```
class FLAT_SELECTIONS                    -- version 3
inherit
        FLAT; CITIES;                    -- provides context: to_sell, tolet, Nice, Antibes, ...
        SELECTION [ FLAT ]               -- corresponding to QUERY_ARGl,
           redefine criteria;
        SELECTION [ FLAT ]               -- corresponding to QUERY_ARG2,
           rename
              select as q2_select,
              criteria as q2_criteria
           redefine q2_criteria
creation   create
feature
  fc, fsl, fs2: COLLECTION[FLAT];

  create is
     do
        .....                                -- filling fc
        fsl:= select(fc);                    -- Apply criteria to fc
        fs2 = q2_select(fc);                 -- Apply q2_criteria
     end; -- create

  criteria (f: FLAT, args: ARRAY [ANY]): BOOLEAN is
        -- Crtieria of query_arg1 query (args is not used)
     do
        Result:= f.rooms_nb = 4 and f.status = to_sell and 800000 <= f.price <= 1200000 and
                 f.district = Cimiez and f.city = Nice
     end; -- criteria

  q2_criteria(f: FLAT; args: ARRAY [ANY]): BOOLEAN is
        -- Crtieria of query_arg2 query (args is not used)
     do
        Result:= f.rooms_nb = 3 and f.status = to_let and f.city = Antibes
     end; -- criteria

end -- class FLAT_SELECTIONS
```

Figure 6.:23: Sélections simples et leurs citères dans la même classe (Eiffel).

Exemple d'emboîtement de requêtes

En examinant le programme (fig. 6.24) qui traîte le cas des requêtes emboîtées vu précédemment (programme 6.21 ), on constate une nette amélioration sur le plan de la concision d'écriture: il suffit d'écrire une seule classe.

Mais la lisibilité laisse beaucoup à dèsirer, ce qui entraine des risques d'erreurs sont nombreux. Ceci est dû aux liaisons implicites des requêtes aux critères et aux actions appelées, et est encore obscurci par les renommages nécessaires. Il suffit au lecteur de

compléter les commentaires signalés par le signe "@@", pour s'en convaincre.

```
class NESTED_SELECTIONS                    -- version 3
inherit FLAT; CITIES;                      -- context of query
        SELECTION [ FLAT ]                 -- corresponding to QUERY_ARG3
           redefine criteria, action       --      -> use: do_if, all, criteria, action

        SELECTION [ FLAT ]                 -- corresponding to QUERYARG4
           rename                          --      -> use: q4_criteria, q4_action
              do_if as q4_do_if,
              all as q4_all,
              criteria as q4_criteria,
              action as q4_action
           redefine q4_criteria
        SELECTION [ ROOM]                  -- corresponding to QUERY_ARG5
           rename                          --      ->use: q5_count, q5_criteria, q5_action
              do_if as q5_do_if,
              all as q5_all,
              count as q5_count,
              criteria as q5_criteria,
              action as q5_action,
              ....
           redefine q5_criteria
creation   create
feature
  fc: COLLECTION [ FLAT ];

  create is
  do ....                                  -- filling fc
        do_if (fc, << fc >>, << >>);       -- @@
     end; -- create

  action (f: FLAT, args: ARRAY [ANY]) is
        do    f.display
        end; -- action

  criteria (x: FLAT, args: ARRAY [COLLECTION [FLAT]]): BOOLEAN is
     do                                    - - @@
          Result:= x.status = to_let and x.city = Antibes and q4_all (args.item (1), x);
        end; -- criteria

  q4_criteria (x: FLAT, args: ARRAY [FLAT]): BOOLEAN is
     do                                    -- @@
          Result:= x.adress = args.item (1).adress and x /= args.item (1) implies
                      not x.has_piano and q5_count (x.rooms) <= 2
        end; -- q4_criteria

  q5_criteria (x: ROOM, args: ARRAY [ANY]): BOOLEAN is
     do
        Result:= x.length < 8 and x.hight > 2
     end; -- q5_criteria
end-- class NESTED_SELECTIONS
```

Figure 6.24: Emboîtement de requêtes par relation de clientèle (Eiffel).

Enfin, nous trouvons que la transmission des collections en argument des requêtes selectives est d'un style trop procédural, incohérent dans une approche à objets.

### 6.3.3. Solution à base de routines paramétriques

Avant de voir comment implémenter en Eiffel une transmission dynamique de routine paramétrique, voyons ce que cela apporterait sur le plan expressif.

#### Apport des routines paramétriques

Notre solution est conceptuellement proche de celle par encapsulation. Les requêtes sélectives restent des primitives de la classe COLLECTION, ce qui permet de les composer. Mais contrairement à cette première solution, on ne cherche plus à écrire autant de classes que de routines paramétriques, en les transmettant directement aux requêtes qui en ont besoin. Pour l'exprimer, on utilise la notation de la figure 6.25.

```
Formal_arguments              =   "(" Argument_declarationlist ")"
Argument_declarat ion_list    =   { Argument_dleclaration_group ";" . . . }
Argument_declaration_group    =   Entity_declaration_group I
                                  Formal_parametric_routine
Formal_parametric_routine     =   Formal_object_name!Formal_type_name!
                                  Formal_routine_name
                                  [Formal_arguments, Type_mark]
Formal_obj ect_name           =   Identifier
Formal_type_name              =   Identifier
Formal_routine_name           =   Identifier
Address                       =   "$" [Qualifier] Feature identifier
```

Figure 6.25: Une syntaxe possible pour les routines paramétriques en Eiffel

Au niveau de la déclaration formelle, cette syntaxe précise à la fois l'identité de l'objet pour les manipulations dans le corps de la routine, son type et la routine qui doit être l'une de ses primitives. Ceci permet de vérifier statiquement l'existence et l'exportation de la routine. Les règles du polymorphisme s'appliquent pour l'objet représentant le paramètre effectif.

Au niveau de la transmission effective, la seule chose à donner est l'adresse de la routine et de son objet. On ne peut pas se contenter de donner le nom de celle-ci, car cela pourrait être confondu avec un appel de fonction sans argument; pour s'intégrer le mieux possible avec les notations d'Eiffel 3, nous utilisons l'opérateur "$.

Le programme (fig. 6.26) donne un exemple d'utilisation de cette syntaxe et le programme (fig. 6.27) son utilisation dans la nouvelle version de la classe *COLLECTION*.

```
class INTEGRALS
creation    create
feature
  integral(a,b: REAL; q!REAL_FUNC!f(x: REAL): REAL): REAL is
      do
         . . .     x:= q.f(y);   . . .
      end;  -- integral
end -- class INTEGRALS
_____________________________________________________________________
class INTEGRATIONS
inherit     INTEGRALS
  create is
      local x, y: REAL; t: TRIGO
      do
         !!t;
         io.putreal(integral(-2 * pi, +2 *pi, $sin));
         io.putreal(integral(-2 * pi, +2 * pi, $t.sin));     -- idem
      end; -- create
end -- class INTEGRATIONS
```

Figure 6.26: Exemple d'utilisation de routines paramétriques

```
class COLLECTION [ ITEM -> ANY ]   -- Implementation of collection in volatile memory
inherit
 LINKED_SET [ ITEM ]          -- provide: start, forth, count, off, item, intersect, merge, .....
      rename count as nb_elements
      export {NONE} count; {ANY} nb_elements, start, forth

feature
  select (criteria: like query; args: ARRAY [ANY]): like anchor is
            -- return all item i in Current satisfying criteria(i, args)
      require
         is_not_a_boolean_function:              is_a_boolean_function_of(criteria,args.item (1));
         is_first_parameter_conforms_to_items: is_conforms_to_items (criteria, args.item (1))
         -- forall i in args from 2 to n:          args.item (i).conforms_to(criteria, args.item (1), i)

  do_all (action: like query; action_args: ARRAY [ANY]) is
            -- apply action(i, action_args) to every item i of Current
      require
         is_first_parameter_conforms_to_items:  is_conforms_to_items (action, args.item (1))
         -- forall i in args from 2 to n:          args.item (i).conforms_to (action, args.item (1), i)

  do_if (criteria, action: like query; args1, args2: ARRAY [ANY]) is
            -- apply action(i, args1) to every item i of Current satisfying criteria(i, args2)
      require
         is_not_a_boolean_function:              is_a_boolean_function_of(criteria,args1.item(1));
         is_first_parameter_conforms_to_items:  is_conforms_to_items (criteria, args1.item (1))
         -- forall i in args from 2 to n:          args.item (i).conforms_to(criteria, args1.item(1), i)
         is_first_parameter_conforms_to_items:   is_conforms_to_items (action, args2.item (1))
         -- forall i in args from 2 to n:          args.item (i).conforms_to (action, args2.item (1), i)
```

```
exists (criteria: like query; args: ARRAY [ANY]): BOOLEANis
        -- is criteria(i, args) for at least one item i of Current?
    require
        is_not_a_boolean_function:              is_a_boolean_function_of(criteria,args.item (1));
        is_first_parameter_conforms_to_items:   is_conforms_to_items (criteria, args.item (1))
        -- forall i in args from 2 to n:        args.item (i).conforms_to (criteria, args.item (1), i)

all (criteria: like query; args: ARRAY [ANY]): BOOlEAN is
        -- is criteria (i, args) for all item i of Current?
    require
        is_not_a_boolean_function:              is_a_boolean_function_of(criteria,args.item (1));
        is_first_parameter_conforms_to_items:   is_conforms_to_items (criteria, args.item (1))
        -- forall i in args from 2 to n:        args.item (i).conforms_to (criteria, args.item (1), i)

count (criteria: like query; args: ARRAY [ANY]): INTEGER is
        -- number of item i of Current satisfying criteria(i, args)
    require
        is_not_a_boolean_function:              is_a_boolean_function_of(criteria,args.item (1));
        is_first_parameter_conforms_to_items:   is_conforms_to_items (criteria, args.item (1))
        -- forall i in args from 2 to n:        args.item (i).conforms_to (criteria,args.item (1), i)

            -- features used for preconditions / postconditions

is_a_boolean_function_of (routine_adr: INTEGER, context: ANY): BOOLEAN is
        -- is the routine of address 'routine_adr' a boolean function of 'context'
    require   not routine_adr = Void

is_conforms_to_items (routine_adr: INTEGER): BOOLEAN is
        -- Does first actual parameter type of the routine of address routine_adr
        -- conforms to generic parameter
    require   not routine_adr = Void

            -- Secret Features
anchor: like Current;
query: INTEGER;
end -- class COLLECTION
```

Figure 6.27: Requêtes sélectives par routines paramétriques (Eiffel 3).

On notera que l'écriture ne correspond pas exactement à la syntaxe définie plus haut dans le texte, ceci à cause de l'absence de routines paramètriques en Eiffel, sauf quand elles participent à la définition de routines externes.

Rappel: Description d'une routine externe en Eiffel

```
nom-routine (nom-paramètre1: nom-type1; ....; nom-paramètren: nom-typen) is
    external "C"
    end    -- nom-routine
```

Exemple de sélections simples

Le programme (fig. 6.28) donne une quatrième version des exemples de sélections simples, en utilisant cette technique de transmission directe des routines paramétriques. Il réunit les avantages de toutes les versions précédentes en Eiffel:

• une seule classe est nécessaire pour l'écriture de toutes les requêtes et de leur contexte,

• la composition des requêtes reste possible puisque ce sont des primitives de la classe *COLLECTION*.

```
class FLAT_SELECTIONS                -- version 4
inherit   FLAT;   CITIES;            -- provides context: to_sell, to_let, Nice, Antibes, . ..
creation  create
feature
  fc, fsl, fs2: COLLECTION [ FLAT];

  create is
     do    ....          -- filling fc
        solution4;   solutionO;
     end;  -- create

  solution4 is
        -- Eiffel with parametric routines
     do
        fsl:= fc.select ($criteria1, << Current >>);      -- équivalent à "$Current.critl"
        fs2:= fc.select ($criteria2, << Current >>);      -- équivalent à "$Current.critl"
     end;  -- solution4

  criteria1 (f: FLAT; args: ARRAY [ANY]): BOOLEAN is
     do                                                   -- args is not used
        Result:= f.rooms_nb = 4 and f.status = to_sell and 800000 <= f.price <= 1200000 and
                 f.district = Cimiez and f.city = Nice
     end;  -- criteria1

  criteria2 (f: FLAT; args: ARRAY [ANY]): BOOLEAN is
     do                                                   -- args is not used
        Result:= f.rooms_nb = 3 and f.status = to_let and f.city = Antibes
     end;  -- criteria2
-------------------------------------------------------------------------
  solution0 is      - - EQL
     do
        fsl:= fc.select f suchas
                        f.rooms_nb = 4; f.status = to_sell; 800000 <= f.price <= 1200000;
                        f.district = Cimiez and f.city = Nice
                end;
        fs2:= fc.select f suchas    f.rooms_nb = 3; f.status = to_let; f.city = Antibes     end;
     end   -- solution0
end-- class FLAT_SELECTIONS
```

Figure 6.28: Sélections simples avec routines paramétriques (Eiffel 3).

La version équivalente en EQL, qui est rappelée en bas du programme, en est très proche: la seule différence est qu'on n'est pas obligé en EQL d'écrire une fonction pour chaque critère.

Exemple d'emboîtement de requêtes

Le programme (fig. 6.29) donne enfin la dernière version de l'exemple des quantifications emboîtées avec cette technique. Là encore, la comparaison avec la solution de référence en EQL, qui est donnée en bas du programme, est éloquente: cette version en quasi-Eiffel est à peine plus complexe que dans le langage spécifique et est nettement plus simple à lire et à écrire que les versions précédentes en Eiffel. Son seul inconvénient par rapport à EQL est encore d'obliger une décomposition du critère en différentes fonctions explicites, mais toutefois dans la même classe de contexte.

```
class NESTED_SELECTIONS                    -- version 4
inherit
    FLAT;   CITIES;                        -- context
creation   create
feature
  fc: COLLECTION [ FLAT ];

  create is
    do
      ....                                  -- filling of fc
      solution3;   solutionO;
    end;  -- create

  solution3 is
      -- Eiffel with parametric routines
      -- apply action1 (i) to every item i of fc satisfying criteria1 (i, fc)
    do
        fc.do_if ($criteria1, $action1, << fc >>, << >>);
    end;  -- solution3

  action1 (x: FLAT; args: ARRAY [ANY]) is
      -- action applied on each item of a collection
    do   x.display
    end;  -- action1

  criteria1 (x: FLAT; args: ARRAY [COLLECTION [ FLAT ]]): BOOLEAN is
      -- test applied on each item of a collection
    do
      Result:= x.status = to_let and x.city = Antibes and
               args.item (1).all($ criteria2, <<Current, x>>);
                          -- is criteria2(i, x) true for all item i of 1st item of args?
    end;  -- criteria1
```

```
criteria2 (x: FLAT, args: ARRAY [ANY]): BOOLEAN is
        -- test applied on each item of a collection
    do
        Result:= x.adress = args.item (1).adress and x /= args.item (1) implies
                    not x.has_piano and
                    x.rooms.count ($ criteria3, << Current >>) <= 2
                                        -- Number ot item i of x.children satisfying criteria3(i)
        end;  -- criteria2

criteria3 (z: ROOM, args: ARRAY [ANY]): BOOLEAN is
    do
        -- args is not used
        Result:= z.age < 8
    end;  -- criteria3
-------------------------------------------------------------------------
solution0 is      - - EQL
    do fc.do x.display with x suchas
                                x.status = to_let; x.City = x.antibes;
                                forall y suchas
                                        x.adress = y.adress; and x /= Y implies
                                        not y.has_piano; and y.children.count z suchas
                                                                        z.age < 8
                                                                end <= 2;
                                    end;
            end;
    end;  -- solution0
end -- class NESTED_SELECTIONS
```

Figure 6.29: Emboîtement de requêtes avec routines paramétriques (Eiffel 3).

## Discussion sur l'introduction de routines paramètriques

Leur introduction suppose une modification du langage Eiffel, mineure certes, mais une modification de langage n'est jamais anodine et doit obéir à des principes stricts:

• être utile à plus d'une situation particulière: un intérêt pour la seule transmission des arguments de requête ne serait pas suffisant. Cela suppose aussi que toute autre solution dans le langage soit mauvaise: c'est le cas ici;

• ne pas compromettre la fiabilité des programmes: c'est le cas ici où un contrôle statique ou dynamique est possible,

• être conceptuellement simple et non redondante avec d'autres mécanismes du langage: c'est aussi le cas comme nous l'expliquerons ci-après,

• être implémentable facilement et efficacement, notamment dans le langage cible C qui reste indispensable pour la portabilité: c'est possible puisque C sait transmettre des routines à une autre.

• ne pas compromettre l'orthogonalité du langage, ni sa simplicité: c'est justement le problème actuellement, puisqu'on peut transmetter une routine ou un attribut Eiffel à une routine externe, mais pas à une routine Eiffel...

Pour ce qui est de l'utilité conceptuelle, on pourrait penser que ce mécanisme fait double emploi avec la transmission d'objets, lesquels peuvent être équipés de routines: il n'en est rien. Si l'on a besoin de faire des hypothèses sur la classe qui englobe une routine dont on a besoin, alors il est clair que c'est une instance complète de la classe qu'il faut transmettre: on a besoin ici d'un type et pas d'une seule opération. C'est pourquoi nous proposons que la seule hypothèse sur la classe d'une routine transmise directement soit qu'elle soit héritière de ANY, ce qui est parfaitement cohérent avec la possibilité de transmettre une routine Eiffel à un programme externe.

Mais il arrive qu'on ait besoin d'une routine en tant que telle, et que sa classe d'origine n'ait aucune signification. En particulier, on peut souhaiter atteindre plusieurs routines de même signature, mais placées dans la même classe. Dans ce cas, il faut pouvoir choisir dynamiquement le lieu de résidence de la routine, ce que ne permet ni l'héritage, ni l'encapsulation.

De plus ce besoin de transmission directe nous paraît utile dans bien des cas, pas seulement pour la persistance et le calcul numérique, mais aussi pour pouvoir augmenter les moyens génériques du langage sans avoir à retravailler toute la hiérarchie des classes de la bibliothèque ou d'un projet.

En conclusion, nous aimerions que le lecteur se persuade que ce n'est pas parce qu'une routine fait partie d'une classe que cette notion n'a pas d'intérêt en soi: **une routine n'est pas toujours une primitive**.

## 6.4. Bilan et perspectives

Dans la partie précédente, deux grands types d'approches ont été étudiés: l'un prenant en compte les facilités de sélection comme des outils partagés par les composants qui en ont besoin (solutions basées sur l'héritage répété ou sur une nouvelle structure itérative), et l'autre intégrant la sélection dans un certain type de composant (solution par encapsulation, routine paramétrique, langage spécifique).

Etudions d'abord les qualités ou les défauts des différentes solutions étudiées dans ce chapitre:

• En ce qui concerne **l'expressivité,** ces solutions peuvent être classées dans l'ordre décroissant suivant: solution en *EQL*, puis les solutions en Eiffel dans l'ordre, avec routines paramétriques, avec héritage répété et avec encapsulation, puis enfin, la nouvelle structure itérative.

• Sur le plan de la **lisibilité,** on obtient le même ordre, mais en permutant les solutions avec héritage répété et encapsulation, car la solution avec héritage nous paraît moins lisible.

• Côté **contrôles statiques,** elles sont assez équivalentes si l'on envisage un traitement complet par le compilateur, ce qui suppose d'intégrer soit le langage *EQL*, soit la transmission directe des routines paramétriques.

• Enfin, pour ce qui est des **performances,** les solution basées sur un *EQL* intégré à Eiffel, ou sur une nouvelle structure itérative sont probablement celles qui donnent le plus de chances d'une traduction améliorée. Ensuite vient la solution avec transmission directe des routines paramétriques, puisque le compilateur peut calculer l'adresse effective des routines appelées, à chaque transmission en argument. La solution avec héritages répétés conduit probablement à une duplication intempestive de code; la solution avec encapsulation nécessite quant à elle une indirection supplémentaire par rapport à la transmission directe.

Considérons maintenant le choix entre **ajout syntaxique** et **approche par composant**. Faire des ajouts syntaxiques dans le langage par rapport à l'implantation de routines paramétriques n'a qu'un intérêt limité: celui de permettre une prise en compte des

problèmes d'optimisation en minimisant le surcoût de travail du compilateur; en effet une solution à base de composants nécessite une mise en oeuvre beaucoup plus complexe du point de vue des optimisations, surtout dans le cas de requêtes emboîtées.

Une autre approche serait d'utiliser un éditeur syntaxique associé à la grammaire d'un sur-langage d'Eiffel. On pourrait alors, soit décrire une grammaire décorée par des actions sémantiques assurant la traduction en Eiffel pur, soit utiliser une bibliothèque de grammaires existantes. Cette approche pourrait être intéressante pour construire des logiciels interactifs conviviaux

Plusieurs implémentations de toutes ces solutions sont possibles:

• Une intégration d'*EQL* (ou d'un langage analogue) à Eiffel ne nous parait pas absurde si l'on considère que la persistance et les quantifications sont des moyens d'expression aussi utiles que le paraléllisme par exemple. Toutefois, il ne faudrait pas imposer à tous ce surcroit de possibilités: un surlangage d'Eiffel ( "Eiffel+"?) pour la persistance et/ou pour le parallélisme nous paraitraît utile, avec évidemment une compatibilité ascendante d'Eiffel vers ce(s) surlangage(s). A défaut d'intégration à Eiffel, on peut envisager un interprète d'*EQL*, qui soutraiterait à un POM. Les requêtes seraient alors transmises par des chaînes.

• l'intégration d'un mécanisme de transmisson directe des routines paramétriques nous paraît non seulement souhaitable, mais utile à d'autres cas d'emploi que celui de la persistence. De plus il ne supposerait que des changements mineurs du langage et de son compilateur: **nous recommandons cette solution** A défaut, on peut chercher à implémenter quelque chose d'analogue mais moins efficace par les routines externes Eiffel 3 (voir fig. 6.27). En effet la transmission de routine en paramètre est introduite de manière partielle dans la version 3.1 d'Eiffel, à travers des routines externes [Meyer 91]. Cependant, on a vu dans les exemples ci-dessus (fig. 6.26-6.28), que la prise en compte, au moment de l'exécution (par des préconditions), des contrôles de type qui ne sont pas fait statiquement, induisait un surcoût et diminuait la lisibilité de la classe *COLLECTION;* par ailleurs, les opérateurs de cette classe devraient être développés en langage C...

• Quant aux solutions en Eiffel pur, nous préférons encore la solution par encapsulation, mais à la condition que les classes d'encapsulation soient produites automatiquement par un traducteur d'Eiffel+ (qui intégrerait EQL) par exemple. Leur

seul intérêt est de pouvoir être traduites par un compilateur Eiffel, car si l'on envisage leur interprétation, alors autant interpréter directement de l'EQL.

## 6.5. Adéquation au modèle

La solution que nous avons sélectionnée est donc celle basée sur le passage des critères en paramètres. Il faut maintenant s'assurer de l'adéquation de l'intégration avec le modèle présenté dans le paragraphe 3.3.

Les primitives de la classe *COLLECTION exists* et *all* implémentetn les quantifiacteurs universels et existentiels; elles se différencient des *fonctions sélectives* (select) par leur résultat (un booléen contre une collection).

Par ailleurs, une *expression sélective* est décrite par une fonction booléenne Eiffel; à ce titre, il est possible de faire référence dans son corps à d'autres fonctions, en particulier à des fonctions sélectives de la classe collection (*select*, *exists*, *all*, ...), qui est une classe comme une autre, et donc de rendre possible l'écriture de requêtes emboîtées.

De plus, le résultat de ces *fonctions sélectives* étant lui-même une collection de type conforme à la collection de départ, il est possible de réaliser des compositions de sélection.

Par contre *count*, qui retourne un résultat de type entier représentant le nombre d'objets vérifiant l'*expression sélective*, n'est pas une véritable fonction sélective; ce choix n'est pas orthogonal à la définition des *fonctions sélectives*, mais l'intérêt de sa présence a été développé dans le paragraphe 6.1.2.

Comme *procédure sélective*, on trouve *do_if* dont la description correspond exactement à la définition des procédures sélectives (cf. § 3.3.3), et *do_all*, qui est un cas particulier représentant la situation où le critère de sélection est toujours vrai.

## 6.6. Généralisation des structures traversables

Le besoin de réaliser des sélections se fait sentir pour n'importe quelle structure parcourable; nous proposons donc d'étendre les mécanismes de sélection à toutes ces structures.

La hiérarchie proposée ressembleà celle utilisée dans la bibliothèque Eiffel, à part quelques modifications:

• introduction de classes parentes à toutes les structures parcourables: *COLLECTION*, *BAG*,

• modification des liens d'héritages utilisés pour l'implantation,

• Ajoût de structures favorisant les objets persistants, et donc implémentées à travers le gestionnaire d'objets.

Ainsi pour les listes ou les ensembles, il y a des classes qui privéligient les performances quand on connaît le nombre approximatif d'élements (*FIXED_LIST*, *FIXED_SET*, d'autres qui privéligient la souplesse (*LINKED_LIST*, *LINKED_SET*), d'autres encore qui privilégient les performances pour des objets persistants, mais permettent quand même de manipuler des objets volatiles (*P_LIST*, *P_SET*).

On notera que toutes ces structures peuvent être, et contenir, des objets persistants ou volatiles. Seulement dans chacun des cas, une catégorie d'objet est privilégiée; par exemple, dans *LINKED_LIST* se sont les objets volatiles, et dans *O2_LIST* ce sont les objets persistants.

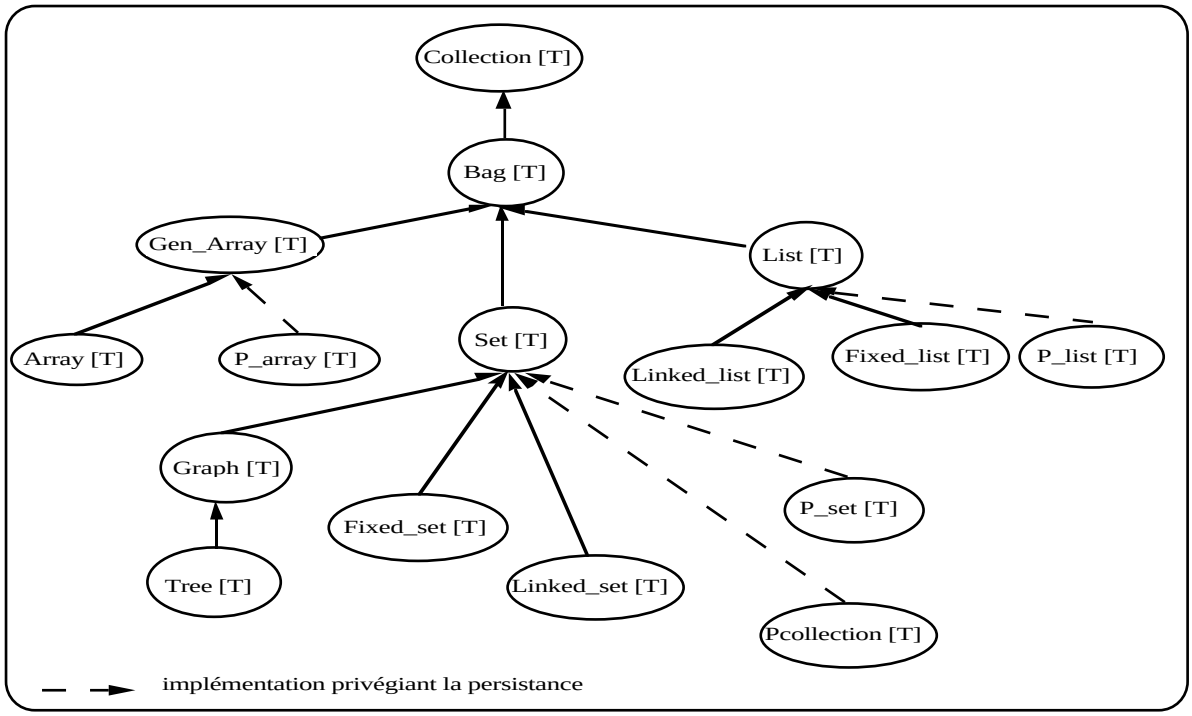


Figure 6.30: Hiérarchie des structures parcourables

En plus des structures traversables citées ci-dessus, nous maintenons une structure parcourable particulière, nommée *PCOLLECTION*, qui représente l'extension d'un type (cf. § 3.4.1).

**Remarque:**

Il pourrait être intéressant d'assimiler un graphe d'objets Eiffel (modélisé par les liens de clientèle entre classes), à une structure traversable descendant de la classe *COLLECTION* (*GRAPH*), dans le but de pouvoir définir des requêtes sélectives de manière navigationnelle. Ces requêtes permetront de décrire les post-conditions dans une classe Eiffel (le chapitre 10 utilise cette facilité pour décrire la classe *INTEGRATION_INTERFACE*).

L'accès à ce graphe d'objets pourrait se faire à travers le type le plus général (classe *ANY* en Eiffel) d'un système de classe (fig. 9.6). On introduit la propriété *object_traversal* permettant de réaliser la fermeture transitive à partir de l'objet courant; le type de cette propriété est *GRAPH* .

```
class ANY
      ......
feature
          ......
      object_traversal: GRAPH [ANY] is
            -- graphe correspondant à l'objet courant
end -- ANY
```

Figure 9.6: La traversée d'un graphe d'objets pour la ralisation de requêtes sélectives

# 6.7. Intégration du concept de *Pcollection*

On a vu dans le paragraphe 3.4 la définition du concept de *Pcollection.* On en propose maintenant l'intégration dans le langage. Le concept de Pcollection est accessible à une application par l'utilisation de la classe *PCOLLECTION*.

**Contraintes d'intégration**

Nous définissons essentiellement trois contraintes d'intégration:

**C6.1**: la *Pcollection* d'un type *T* est un sous-ensemble des instances de *T* inclus dans le monde persistant. Chaque instanciation d'une classe générique correspond à un type et possède donc sa propre *Pcollection.*

**C6.2**: la *Pcollection* du type *T* s'adapte automatiquement aux modifications du monde persistant. Cette adaptation est transparente au programme,

**C6.3**: une Pcollection est un objet directement accessible par son nom dans le monde persistant.

**Conséquences**

• Un processus peut ne pas avoir l'autorisation de consulter certains objets persistants (C6.1, C6.2). En effet le **monde persistant** contient tous les objets persistants, et donc surement des objets dont les protections en interdisent l'accès à un processus;

• les objets persistants d'une classe peuvent tous être manipulés sélectivement,

• les traitements d'une *Pcollection* sont effectués en mémoire persistants,

• un objet volatile ne peut appartenir à une *Pcollection*,

• la synchronisation entre le contenu de la *Pcollection* et celui du monde persistant est transparente pour un programme Eiffel,

• seuls les liens d'héritages peuvent rendre possible l'appartenance d'un objet persistant à plusieurs *Pcollection*. Soit *obj* une instance de la classe *STUDENT* qui hérite de la classe *PERSONNE*, *obj* appartiendra logiquement à la *Pcollection* associée à la classe *STUDENT*, et à celle associée à la classe *PERSONNE*.

## Intégration dans la hiérarchie des structures parcourables

Le schéma d'intégration du concept de *Pcollection* est représenté par la hiérarchie de classes décrite ci-dessus. Une *Pcollection* est une spécialisation de la notion d'ensemble.

Dans le cas où c'est *O2* qui assure l'implémentation du gestionnaire d'objets, une Pcollection est un cas particulier de la classe décrivant les ensembles *O2*(O2_SET).

Une Pcollection est un ensemble persistant dont l'identification est déduite par le système à partir du nom de la classe.

La *Pcollection* d'une classe peut-être manipulée par le programme comme une collection. Cependant, du fait de son caractère persistant, une *PCOLLECTION* pourra contenir certaines primitives supplémentaires.

## Intégration dans la classe *ANY*

Afin que chaque objet puisse réaliser des traitements sélectifs avec le plus de souplesse possible, en particulier dans le cas où on veut intégrer l'aspect logique pour améliorer le langage des assertions, il faut établir des liens entre la classe *ANY* et la classe *PCOLLECTION* dont le paramètre générique correspond au type de l'objet ou à la classe de son choix (voir fig. 6.31).

```
class ANY
feature
        ........
        pcollection: PCOLLECTION [Like Current] is
                -- returns the Pcollection corresponding to the class of the object

        class_pcollection (s: STRING): PCOLLECTION [ANY] is
                -- return the pcollection corresponding to class s
        ..........
invariant
        ..........
end -- Class ANY
```

Figure 6.31: Intégration du concept de Pcollection dans le type ANY

On a vu dans le paragraphe 3.4.2, le cas d'une structure de groupe; nous montrons dans la figure 6.32 un exemple de l'utilisation des *Pcollections*.

```
deferred class GROUP
feature
        zero: like Current is deferred end; -- zero
        infix "+" (other: like Current): like Current is deferred end; -- "+"
        prefix "-": like Current is deferred     end; -- "-"
        value: VALUE;     -- value of current object
        set (new: VALUE) is
            ensure     value = new
            end; -- set
invariant       -- class invariant
        -- associativity:           ∀ x, y, z ∈ pcollection (x+y)+Z=x+(y+z)
            p_collection.all($associativity)

        -- neutral_element:         ∀ x ∈ pcollection zero + x = x + zero = x
            p_collection.all($neutral_element)

        -- symetric element:        ∀ x ∈ PCOLLECTION x + -x = zero
        p_collection.all($symetric_element)

end - - class GROUP
```

Figure 6.32: Application des Pcollections sur les structures de groupe mathématique

Les routines *associativity*, *neutral_element*, *symetric_element*, sont déclarées dans la classe *GROUP*, ou éventuellement dans une classe parente. Les classes descendantes (par exemple *INTEGER*), pourront ainsi vérifier les propriétés d'un groupe d'entier à partir de la Pcollection associée à la classe *INTEGER*, en appliquant la primitive *all*. Dans le cas de l'associativité on constate la présence de plusieurs requêtes sélectives emboitées).

## Primitives spécifiques à une pcollection

On devra pouvoir, pour une *Pcollection:*

- accéder aux informations de la classe de base de la *collection* (*base_class*),
- accéder au type de la*Pcollection* ,
- retourner un objet persistant à partir de son identificateur persistant (*item_obj*),
- effacer un objet à partir de son identificateur d'objets (*remove_obj*),
- connaître l'identificateur persistant (POI), du dernier objet inséré explicitement (*last_poi*),
- dupliquer un objet persistant (*duplicate_obj*).

Ces fonctionnalités sont représentées dans la classe *PCOLLECTION*, décrite dans la figure 6.33.

```
class PCOLLECTION [P]
inherit    SET [P]    export {NONE} do_if,  do_all
Creation   create
feature
      create (s: STRING) is                -- Create a persistent collection
          ensure     e_name.equal (s)

      last_poi: POI;                       -- persistent object identifier of last inserted object

      item_obj (obj: POI): P is            -- return object corresponding to obj
          require    not obj.void
          ensure     not Result.void implies Result.is_persistent

      remove_obj (obj: POI) is             -- Make obj   obsolete
          require    not obj.void
          ensure     is_obsolete  (obj)

      duplicate_obj (obj: POI): P is       -- Duplicate object corresponding to obj
              ensure     not Result.void implies not Result.is_persistent
invariant
      is_persistent
end -- class PCOLLECTION
```

Figure 6.33: Intégration en Eiffel du concept de *Pcollection*

On notera que conformément à ce qu'il a été dit au sujet de l'encapsulation (cf. § 3.4.2), certaines routines de la classe COLLECTION (via la classe *SET*), ne sont pas exportées. On rappelle cependant, qu'une *Pcollection* peut être vue par l'utilisateur comme une *Collection* (polymorphisme).

# 6.8. Intégration du concept de monde persistant

Comme cela a été précédemment dit dans le paragraphe 3.6, l'intégration en Eiffel du concept de monde persistant a essentiellement pour objectif l'uniformisation du traitement des classes et des objets persistants. Dans une première section nous approfondissons cet aspect, et dans une deuxième nous présentons son intégration dans la hiérarchie des classes.

## 6.8.1. Unification de la gestion des entités persistantes

En Eiffel, il existe déjà la notion de monde persistant. Elle apparaît à travers un fichier (nommé *world*), qui est géré par l'exécutif, et a pour but de maintenir un inventaire des noms de classes Eiffel (sur chaque machine); par ailleurs, les différentes instances de la classe *ENVIRONMENT* contiennent les instances persistantes.

Nous unifions le traitement des classes et des objets persistants, en regroupant dans une instance de la classe *PERSISTENT_WORLD*, représentant le monde persistant dans notre approche (cf. § 3.6), tous les objets persistants et les classes.

En affinant la définition du concept de monde persistant, nous allons permettre la réalisation de traitement sélectifs sur les différentes *Pcollections*; cela sera en particulier intéressant pour les pcollections contenant les instances des classes *EIFFEL_CLASS*, *FEATURE* ou *ANY*:

- recherche par l'identificateur de l'objet,
- mise à jour globale en fonction de la date de création ou de modification,
- recherche de composants et de routines.

Ce dernier point sera en particulier détaillé dans les perspectives (chapitre 13).

Un objet devient persistant dès qu'il est inséré dans le monde persistant, il en va de même pour les classes; on étudiera dans le paragraphe 9.2 comment propager les classes dans le monde persistant, et comment mettre en oeuvre la classe *EIFFEL_CLASS* qui rassemble les informations correspondant aux classes. Naturellement cette classe devra permettre l'accès à tous les concepts contenus dans une classe Eiffel:

- attributs, (avec des informations sur leur type, l'expansion, leur nom, ...),
- routines (retardées, externes, obsolètes, types des paramètres et du résultat, leur nom,...),
- constantes (type, nom),
- liens d'héritages avec les clauses associées (renommage, redéfinition, ...),
- exportation éventuellement limitée des primitives,

....

### 6.8.2. Intégration dans le schéma de conception

La *Pcollection* correspondant au *monde persistant* est l'ensemble des objets accessibles à le gestionnaire d'objets, c'est à dire l'ensemble des objets persistants (*Pcollection* des objets de type *ANY*). Le fait qu'une instance de la classe *PCOLLECTION* puisse être le type de base d'une *Pcollection*, favorise l'orthogonalité par rapport au système de type d'Eiffel. L'intégration du monde persistant dans la hiérarchie des classes Eiffel est décrite dans la figure 6.34.

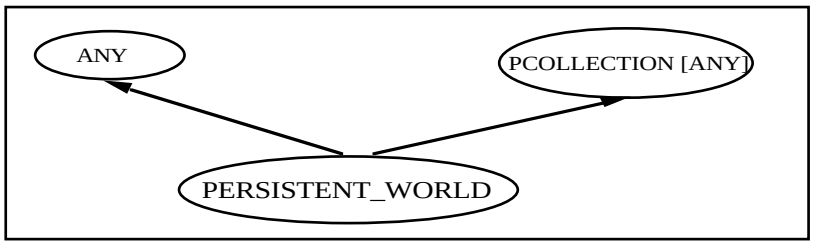


Figure 6.34: Positionnement du monde persistant

Les opérateurs du monde persistant (fig. 6.35) doivent permettre d'insérer, consulter ou supprimer un objet, soit à partir d'un symbole associé explicitement par les application (par l'intermédiaire de la classe *PERSISTENT_WORLD*), soit à partir d'un identificateur d'objet persistant (*POI*) choisi par le POM; ceci devrait permettre d'augmenter la souplesse et le confort de programmation.

```
Class PERSISTENT_WORLD
inherit
  PCOLLECTION [ANY]
feature
  put_by_key (obj: ANY; symbol: STRING) is
        -- Record obj under symbol in the persistent world; Make obj and ist dependents
        -- persistent. If symbol is already in use or if not enough privileges, set error
    require   not obj = void and not symbol = void
    ensure    (ok and obj.nb_ref = 1) or (error = conflict) or (error = unsufficient_privilege)

  remove_by_key (symbol: STRING) is
        -- Dissociate symbol from recorded object if any or else set error. Object
        -- attached to this symbol and its dependents are removed if they are not attached
        -- to other objects or symbols
    require   not symbol = void
    ensure    (ok and item (symbol).nb_ref = old item (symbol).nb_ref - 1) or
              (error = not_found)

  has_kzy (symbol: STRING): BOOLEAN is
        -- Is symbol in use?
    require   not symbol = void

  item_by_key (symbol: STRING): ANY is
        -- Object recorded under symbol; void value if no such symbol and
        -- set error to not_found
    require   not symbol = void
    ensure    Result = void implies not has(symbol)

  change_symbol (old_symbol, new_symbol: STRING) is
        -- Record under new_symbol the object previously recorded under old_symbol;
        -- if no such object set error to not_found
    require   not (old_symbol = void or new_symbol = void)
    ensure    ok or (error = conflict) or (error = not_found)

  ok: BOOLEAN
        -- Did last put, change_symbol or remove succeed?

  error: INTEGER
        -- code of last error produced by routines of the class (possible codes are given next)

  not_found: INTEGER is unique;
  conflict: INTEGER is unique;
  permission_denied: INTEGER is unique

invariant
    is_persistent
end -- class PERSISTENT_WORLD
```

Figure 6.35: Description des opérateurs du monde persistant

Il est important que l'accès au monde persistant soit accessible à chaque objet avec le plus de souplesse possible, que ce soit pour interroger le monde persistant ou bien pour rendre un objet persistant.

On ajoutera donc une primitive de type *once* à la classe *ANY* (fig. 6.36), représentant le monde persistant (l'instance est partagée par chaque objet persistant ou volatile du processus).

```
class ANY
feature
        .......
        pw: PERSISTENT_WORLD is
            -- returns the unique instance of the persistent world
            once
        .........
invariant
        .........
end -- Class ANY
```

Figure 6.36: Intégration du monde persistant dans la classe ANY

Ainsi si un objet veut se rendre persistant (1) ou rendre persistant (2) un des objets qu'il référence (ex: un objet référencé à travers un attribut *one_attribute*), il suffira d'écrire:

```
(1): pw.put (Current)
(2): pw.put (one_attribute)
```

## 6.8.3. Partitionnement du monde persistant

### Pourquoi un partitionnement du monde persistant

Le *monde persistant* est la structure unique contenant tous les objets persistants. Nous avons vu (section 3.4), l'intérêt d'instaurer le concept de *Pcollection*, et par la même, un regroupement en fonction des types. Cependant, ce mode de partitionnement pose un certain nombre de problèmes:

- Pas de séparation entre des objets de même type manipulés à des fins souvent différentes par des applications différentes,
- des partitions trop grandes, et donc un accès sélectif peu efficace,
- des critères de sélection difficiles à écrire: la quantité d'objets correspondant à une *Pcollection*, nécessite le rajout de sous-critères permettant de restreindre le nombre d'objets; il y a donc moins de souplesse lors des accès par les valeurs,
- une base peu propice pour la mise en oeuvre d'un mécanisme de protection efficace et adapté aux besoins des applications (cf. § 7.2).

Les problèmes cités ci-dessus montre qu'il faut trouver au dessus du partitionnement par les types, un autre plus fin basé sur les objectifs d'utilisation, qui devrait permettre une augmentation de l'efficacité et du confort de programmation. Naturellement ce nouveau grain de partitionnement ne gêne en rien les autres regroupements d'objets (création de collections persistantes), définis explicitement par les applications lors de leur conception.

### Mécanisme de partitionnement

Les problèmes mentionnés ci-dessus ne concernent que la gestion des objets persistants et non langage. Nous les résoudrons donc par des composants.

Il faut aussi respecter les principes suivants:

• la sélection d'un partitionnement ne doit nécessiter aucun changement dans le code d'une application;

• une application dont la conception n'est pas influencée par le partitionnement doit donner le même résultat qu'elle s'intéresse au monde persistant tout entier ou seulement à une partie de ce monde;

• le partitionnement doit induire aussi peu de modifications que possible dans la notion de *monde persistant* et de *Pcollection*;

• il doit être facilement implémenté par le POM choisi.

### Modélisation des concepts

L'idée conductrice de notre approche est simple: nous voulons permettre l'accès au monde persistant suivant plusieurs **points de vue**. Le *monde persistant* représente la vue la plus générale.

**Définition: Le concept de pview**

*On appelle* ***pview****, une collection d'objets représentant une partie du monde persistant. A chaque pview est associé un nom.*

Le concept de pview doit comme le monde persistant, favoriser la souplesse d'utilisation en fournissant à des applications des moyens:

• de manipuler navigationnellement un petit nombre d'objets dèsignés explicitement (objets associés à un symbole),

• d'effectuer des sélections sur des collections d'objets construites explicitement ou implicitement (Set, List, graph, Pcollection, ....),

• de vérifier des assertions (pré-conditions, post-condition et invariant), associés aux routines et aux propriétés des classes (Pcollection),

**Contraintes d'intégration**

Afin de ne perdre en rien l'intérêt du concept de monde persistant, ni celui de pcollection, on est amené à définir un certain nombre de contraintes:

• **C6.4:** le monde persistant est une *pview*;

• **C6.5:** la *pview* choisie est déterminée au moment de l'exécution (pas de modification du texte de l'application), mais des facilités pour changer de pview en cours d'exécution doivent être fournies;

• **C6.6:** une *pview* peut être composée de plusieurs autres *pview*s et donc partager leurs objets. Les ***pviews* composantes sont des sous-ensembles de la *pview* composée**; toutes les *pviews* sont des composantes du monde persistant;

• **C6.7**: une *pview* permet, comme le *monde persistant*, l'accès à la description des typess dont elle permet d'atteindre les instances;

• **C6.8**: la composante (directe ou indirecte) d'une *pview* ne peut être la *pview* composée elle-même;

• **C6.9**: à chaque type *T* ayant des instances accessibles par la *pview V* est associé une *pcollection* contenant toutes les instances de *T* dans *V* (même celles communes à d'autres vues),

• **C6.10**: une *pview* est un objet persistant (en particulier la *pview* correspondant au

*monde persistant*),

• **C6.11**: une *pview* peut contenir toute catégorie d'objets persistants (*pcollections*, objets associés à un symbole, pview, etc),

• **C6.12**: pendant son exécution, **à un instant donné, une application ne voit qu'une seule *pview*,** c'est celle sur laquelle l'utilisateur a décider de travailler, soit implicitement, soit explicitement.

## Conséquences

Le concept de *Pcollection* s'appliquera aux instances d'un même type, contenues dans une *pview*, et non plus à toutes les instances persistantes d'un type donné. Un objet accessible par plusieurs vues n'est pas dupliqué mais partagé.

On déduit que:

- soient:
  - une vue V, et V1, ..., Vn, des vues composantes,
  - $P_t$, $P_t1$, ...., $P_t$n, les Pcollections de V, V1, ..., Vn, associées à un type T

  alors: $P_t = P_t1 \cup ... \cup P_t$n

- il faut affiner la notion de Pcollection.

**Définition: Concept de Pcollection**

*A tout type d'un ensemble de processus logiquement reliés est associée une collection contenant tous les objets persistants de ce type dans la **pview**. Cette collection persistante est la **Pcollection** associée au type considéré .*

Le concept de *pview* ne permet pas de décrire des regroupements sur les types ou les classes; les regroupements de classes qui résultent de l'utilisation des pviews dépendent de ceux définis sur les objets, et on ne peut en déduire aucune signification particulière. Le concept de pview n'interfère donc pas avec celui de Cluster défini dans [Meyer 91].

## Exemple d'utilisation

Supposons que **le *monde persistant* d'une machin**e contienne les entités persistantes d'une compagnie aérienne. Ces entités persistantes concernent essentiellement des personnes (employés au sol, naviguants, constructeurs, et passagers), du matériel (avions, pièces de rechange), des services (vols, réservations, hôtels ...).

On considère trois départements de la compagnie (fig. 6.37), le premier s'occupe plus spécialement des passagers (informations permettant le paiement et un meilleur service), le deuxième, de la gestion du personnel (salaires, congés, primes, ...), le dernier, des informations sur la disponibilité des avions, sur les passagers et sur le personnel afin de gérer les vols de la compagnie.

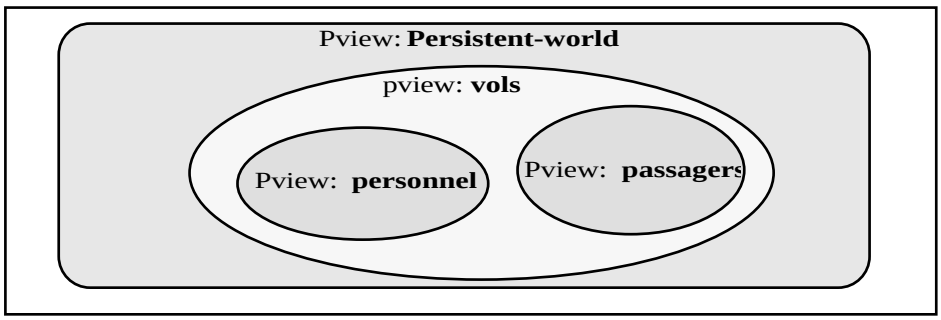


Figure 6.37: Exemple d'utilisation du concept de *Pview*

On a donc tout naturellement créé une vue pour la gestion des pilotes, une autre pour celle des passagers, et une enfin pour la gestion des vols qui partage les objets des deux premières vues. L'intérêt d'avoir plusieurs vues partageant des objets permet de:

- partitionner les entités persistantes de la compagnie, sans bouleverser les relations entre elles, quand c'est nécessaire,
- une meilleure efficacité pour les opérations de sélection,
- établir les bases d'un mécanisme de protection pour l'accès aux informations concernant les pilotes, les vols, les avions et les passagers.

### Intégration dans le schéma de conception

le choix de la pview est donc déterminé lors de la création du processus associé à l'application. Si l'utilisateur ne spécifie rien, alors par défaut le processus s'exécutera sur **le monde persistant tout entier**. Par contre, il pourra choisir une autre vue en procédant comme suit.

| $ nom_application **-pview** nom_pview .... autres paramètres de l'application |
|---|

Ainsi, le processus de l'application *nom_application* s'exécutera sur la vue *nom_pview* du monde persistant. Si on choisi cette solution, il faut garantir que l'introduction de l'option *-pview* ne modifie en rien le rang des paramètres dans la classe ARGUMENTS qui permet d'y accéder. On pourra donc préférer l'utilisation d'une variable d'environnement EIFFEL_PVIEW.

| export **EIFFEL_PVIEW** = *nom_pview* |
|---|

Le partitionnement du monde persistant impose d'ajuster la hiérarchie des classes de notre modèle afin d'intégrer la notion de *pview* dans la classe *PERSISTENT_WORLD* (fig. 6.38)

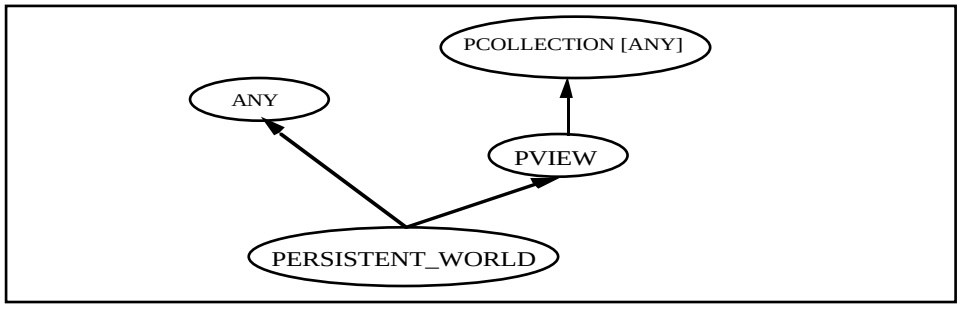


Figure 6.38: Nouveau positionnement du monde persistant

## Raffinement de la classe PERSISTENT_WORLD

Les caractéristiques de la classe *PERSISTENT_WORLD*, décrites au début de cette section, sont maintenant associées à une pview; la classe *PVIEW* contient les services suivants (fig. 6.39):

- gestion des objets associés à un symbole, accessibles par la pview,
- maintenance par requête sélective des objets accessibles par la pview,
- accès aux Pcollection associées à la pview.

La classe *PVIEW* introduit aussi un nouveau service, le partage des objets entre vues.

```
Class PVIEW
inherit    PCOLLECTION [ANY]
creation   create
feature
  name: STRING;    -- view name (is unique in persistent world)
  persistent_world: PERSISTENT_WORLD is   -- root view is the full persistent world

  create (n: STRING) is
     require   not persistent_world.exist_view (n)
     ensure    name = n; persistent_world.include (Current)

  include (view: PVIEW ) is                    -- include objects of view
     require   not view = void

  Does_include (view: PVIEW): BOOLEAN is       -- Does view is included in view
     require   not view = void

  sub_views: LINKED_LIST [PVIEW] is            -- pviews included in view

  put_by_key (obj: ANY; symbol: STRING) is
        -- Record obj under symbol in the persistent world; Make obj and ist dependents
        -- persistent. If symbol is already in use or if not enough privileges, set error
     require   not obj = void and not symbol = void
     ensure    (ok and obj.nb_ref = 1) or (error = conflict) or (error = unsufficient_privilege)

  remove_by_key (symbol: STRING) is
        -- Dissociate symbol from recorded object if any or else set error. Object attached
        -- to this symbol and its dependents are removed if they are not attached to
        -- other objects or symbols
     require   not symbol = void
     ensure    (ok and item (symbol).nb_ref = old item (symbol).nb_ref - 1) or
               (error = not_found)

  has_kzy (symbol: STRING): BOOLEAN is         -- Is symbol in use?
     require   not symbol = void

  item_by_key (symbol: STRING): ANY is
        -- Object recorded under symbol; void value if no such symbol and
        -- set error to not_found
     require   not symbol = void
     ensure    Result = void implies not has(symbol)

  change_key (old_symbol, new_symbol: STRING) is
        -- Record under new_symbol the object previously recorded under old_symbol;
        -- if no such object set error to not_found
     require   not (old_symbol = void or new_symbol = void)
     ensure    ok or (error = conflict) or (error = not_found)

        -- Error handling: error may take different code of error (not_found, .....)
  ok: BOOLEAN     -- Did last put, change_symbol or remove succeed?
  error: INTEGER  -- code of last error produced by routines of the class
invariant  is_persistent and persistent_world.does_include (Current)
end -- class PVIEW
```

Figure 6.39: Description des opérations d'une *pview*

Le monde persistant (classe *PERSISTENT_WORLD*) est une pview et donc hérite de toutes les caractéristiques de la classe *PVIEW*; cependant cette pview est la seule à ne pouvoir être incluse dans une autre (fig. 6.40).

On notera que si une *pview V2* est composante d'une autre *pview V1*, et que ni *V1*, ni *V2* ne sont déjà associées au monde persistant, alors si *V1* est attachée au monde persistant, *V2* le sera implicitement; attention, l'inverse n'est pas vrai.

```
class PERSISTENT_WORLD
inherit     PVIEW
feature
  delete_view (view: VIEW) is
          -- Delete view
      require    not view = void

  view_item (s: STRING): PVIEW is
          -- pview corresponding to name s
      ensure     not Result = void implies does_include (Result)
invariant
      is_persistent and name.equal  ("Persistent_world")
      -- forall view_item: not p.does_include (Current)
end -- PERSISTENT_WORLD
```

Figure 6.40: Nouvelle description du monde persistant

Il est possible de supprimer une vue *V* du monde persistant, mais la suppression d'une pview entraîne la perte de tout objet rendu persistant par un processus associé à cette vue; les objets appartenant aux vues incluses ne sont pas affectées. On notera que le concept d'*objet obsolète* s'applique aussi aux instances de la classe *PVIEW*.

L'introduction de ce partitionnement du monde persistant nécessite cependant quelques modifications au niveau de la classe *ANY* (fig. 6.41).

```
class ANY
feature
      ........
      pview: PVIEW is          -- returns the pview associated to the process
          once
      .........
invariant  .........
end -- Class ANY
```

Figure 6.41: Intégration dans la classe ANY

Dèsormais les processus n'accéderont plus au *monde persistant* mais à une vue de celui-ci (suppression de la caractéristique *persistent_world*, fig 6.35), qui est par défaut le *monde persistant* lui-même.

### Comment créer une *pview*

Reprenons l'exemple étudié ci-dessus et supposons qu'on en soit au stade où le développeur veuille construire une application de gestion des vols, et que les vues *personnel* et *passagers* (*personnel* et *passagers*) aient déjà été créées. On créé une vue conforme au partitionnement montré ci-dessus (fig. 6.37), par la routine *vol_creation* (fig. 6.42).

```
vol_creation is                    -- création d'une vue de nom "vol"
  local v: PVIEW                   -- pview "vol" n'existe pas déjà dans le monde persistant,
                                   -- mais que les vues "personnel" et "passagers" existent
     !!v.create ("vol");           -- la vue est créée dans le monde persistant qui donc inclut v
     v.include (v.persistent_world.item_view("personnel"));
     v.include (v.persistent_world.item_view("passagers"));
        .....                      -- éventuellement remplissage de v
  ensure
     v.sub_views.nb_elements ≥ 2;
  end  -- vol_creation
```

Figure 6.42: Exemple de création d'une pview

Dèsormais toute application peut atteindre les objets de la *pview* "vol", soit à travers la *pview* "vol" elle même, soit à travers le *monde persistant*, soit encore à partir de toute *pview* incluant "vol".

### Comment utiliser une *pview*

Quelque soit la vue utilisée, un processus peut toujours atteindre les objets regroupés dans cette vue, par l'intermédiaire de la propriété *pview* de la classe ANY. Tout objet de l'application peut comme avant l'introduction du partitionnement:

- se rendre persistant (1),
- atteindre un objet associé à un symbole (2),
- exécuter une routine sélective (3).

```
pview.put (Current)                (1)
pview.item_by_symbol ("objet 1")   (2)
pcollection.select ($assertion1)   (3)
```

# 7

# Fiabilité et sécurité

Deux points essentiels dans la gestion des objets persistants sont leur sécurité et leur intégrité. Les deux paragraphes du chapitre ont pour objectif l'intégration de ces services dans le langage Eiffel.

## 7.1. Intégrité des objets

L'exécutif Eiffel doit préserver le monde persistant et le processus lui-même contre les abus d'un programme (contrôle dynamique de type). Il doit aussi garantir pour un processus la préservation de la cohérence des données, quelque soient les interventions des autres processus, et les pannes qui peuvent se produire (mécanisme transactionnel).

### 7.1.1. Contre qui protéger les entités persistantes

Il faut d'abord protéger les entités persistantes contre les erreurs de programmation et interdire à ceux-ci de contourner les vérifications de type réalisées à la compilation: par exemple en affectant à un attribut de type *T* un objet persistant de type *Y*,si *Y* n'est pas conforme à *T*. En Eiffel cette vérification est faîte par l'opérateur d'affectation conditionnelle (*?*=). On notera cependant les limitations de cet opérateur qui détectera une incompatibilité lorsque Y n'est pas inclus dans le système de classes de départ (cf. § 5.1.3), même si *Y* est conforme à *T*.

Il faut aussi préserver les processus de toute action extérieure. Une fonction essentielle est de pouvoir garantir l'intégrité des objets persistants en cas de panne ou d'accès simultanés. Mais cette notion doit être aussi transparente que possible dans la phase de conception d'un programme. Pour parvenir à cet objectif, on va être amené à utiliser la notion de transaction implantée au niveau du *POM* afin de prendre en compte la sérialisation des accès aux objets persistants, et leur cohérence en cas de panne.

Dans un univers non parallèle le début, la validation et l'annulation d'une transaction est réalisé automatiquement, l'objectif étant uniquement de préserver la cohérence des objets en cas de panne.

Cependant, dès que plusieurs processus essaient de consulter ou modifier les mêmes objets simultanément, cette approche montre ses limites et une intervention de l'application devient nécessaire.

### 7.1.2. Notion de transaction pour un processus Eiffel

On rappelle la définition du concept de transaction que l'on peut trouver un peu partout dans la littérature.

**Définition 7.1: Transaction**

*Une transaction est une suite d'actions qui réussit ou échoue complètement, en garantissant la sérialisation des consultations et des modifications d'objets en cas d'accès concurrent sur ces objets par plusieurs processus.*

Dans la plupart des langages, la décision de rendre un objet persistant est décidé explicitement; le seul cas à notre connaissance de langage gérant implictement les transactions est Pclos [Paepcke 89]. Pclos accepte le déclenchement et la terminaison explicite des transactions, mais par défaut associe une transaction à chaque opération de chargement / mise à jour d'un objet persistant.

#### Définition d'une transaction Eiffel

La suite d'actions correspondant à une transaction ne peut pas être choisie au hasard par le système: elle doit s'appuyer sur la sémantique du programme, pour que la validation des modifications correspondent à la terminaison d'une tâche significative.

Dans un programme, une routine représente une suite d'actions correspondant à la réalisation d'un objectif (le rôle de la routine).

1ère tentative de définition d'une transaction:

*Une transaction correspond à l'exécution d'une routine Eiffel.*

Cette approche parait séduisante dans un premier temps mais elle ne permet pas de faire la différence entre une méthode créée dans un esprit de modularité, et une méthode représentant un service pour d'autres composants.

En Eiffel, la distinction entre ces deux types de routines est faite par la notion de primitive exportée ou secrète. Donc une approche plus fine serait de faire correspondre une transaction à chaque méthode exportée par une classe Eiffel.

2ème tentative de définition d'une transaction:

*Une transaction correspond à l'exécution d'une routine exportée Eiffel.*

Ceci n'est pas encore suffisant; il arrive souvent qu'une méthode exportée soit composée d'un ou de plusieurs appels à des primitives exportées ce qui provoquerait l'apparition de transactions emboîtées dont l'intérêt n'est pas prouvé dans le cas d'un langage de programmation. De plus cela serait pénalisant du point de vue de l'efficacité.

Par ailleurs, introduire la notion de transaction quand la méthode s'applique à un objet non persistant supposerait un mécanisme très lourd, gourmand en place, qui diminuerait les performances, et modifierait la philosophie du langage pour les objets volatiles, sans véritable intérêt pour l'utilisateur.

Il faut donc rendre le démarrage d'une transaction dépendante du contexte de l'appel de la méthode. On donnera donc la règle suivante.

**Définition 7.2: Transaction Eiffel**

*L'exécution d'une routine exportée Eiffel correspond à une transaction si elle est appelée par un objet non persistant et si elle s'applique à un objet persistant.*

**Conséquence:**

On a vu que tout objet référencé par un objet persistant ne peut référencer que des objets eux-mêmes persistants (règle n°4, § 3.1); donc le fait d'appliquer cette définition implique que toutes les routines appelées à travers des relations d'héritage ou de clientèle ne génèrent pas de nouvelles transactions même si la routine est exportée.

Ceci est à rapprocher du mécanisme d'évaluation des invariants d'instances: il n'est pas évalué tant qu'une primitive exportée est active; il l'est seulement à l'entrée et à la sortie de celle-ci.

## Intégration du mécanisme transactionnel dans Eiffel

L'intégration peut se faire par l'adjonction de morceaux de code (ex: macro) permettant de mettre en oeuvre une demande conditionnelle de début et de fin de transaction (cf. § 10.8). Ces adjonctions sont faites au moment de la compilation et donc sont tout à fait transparentes aux composants du programme. Elles doivent par ailleurs, respecter les contraintes suivantes:

**C 7.1**: si une transaction échoue, les objets persistants modifiés pendant son déroulement ne sont pas mis à jour; si elle réussit, alors ces objets sont effectivement mis à jour dans le monde persistant,

**C 7.2**: l'intégrité des valeurs contenues dans les objets volatiles est à la charge du programme qui pourra utiliser le mécanisme d'exception Eiffel,

**C 7.3**: le début et la validation d'une transaction, sont transparents au programme Eiffel dans le cas où les objets ne sont pas partagés par plusieurs processus,

**C 7.4**: si une transaction demande la modification d'objets qui sont utilisés par la transaction en cours d'un autre processus, alors la demande de modification échouera (voir § 7.1.4).

### 7.1.3. Extension d'une transaction Eiffel

Tout programme, qui veut utiliser le mécanisme transactionnel défini ci-dessus pour gérer l'intégrité, n'a donc rien à faire, à part peut-être de tenir compte des exceptions éventuelles déclenchées par le POM. Mais dans certains cas, sa structure nécessite l'extension d'une transaction Eiffel, et donc son déclenchement explicite.

**Définition 7.3: Extension d'une transaction**

*L'extension d'une transaction se fait en surchargeant les opérations de début, de validation ou d'annulation déclenchées par l'exécutif, par des appels explicites dans le programme.*

## Intégration dans le schéma de conception

La définition 7.3 nous amène à définir un composant *TRANSACTION* qui permet de démarrer, valider ou annuler une transaction (fig. 7.1).

```
class TRANSACTION
feature

  is_transaction: BOOLEAN is
        -- Is a transaction currently running

  start is
        -- Begin a new transaction if none has already started
        -- overload the policy defined by the run-time

  finish is
        -- finish and validate the current transaction
        -- have no effect, if there is no transaction undergoing execution

  abort is
        -- Cancel all modifications performed on persistent objects since
        -- the beginning of the transaction

end -- class TRANSACTION
```

Figure 7.1: Surcharge du mécansime de transaction par défaut

Selon son type, une application peut avoir besoin d'utiliser fréquemment le déclenchement explicite d'une transaction. Il sera donc intéressant de pouvoir disposer d'un accès aux services de la classe *TRANSACTION*, au niveau de la classe *ANY* (fig. 7.2).

```
class ANY
feature     ........
  transaction: TRANSACTION is
        -- Provide an access for handling the transaction mechanism
      once
      ......
end -- Class ANY
```

Figure 7.2: Intégration dans la classe ANY

## Exemple d'utilisation

Soit le composant suivant, qui effectue un traitement conditionnel sur un objet persistant, la figure 7.3 montre l'intérêt de surcharger le mécanisme de gestion implicite des transactions.

```
class CONCURRENT_USE
feature

  possible_concurrent_access is
        -- Overloading of transaction mechanism handled by the run-time
      local
        obj: PERSON;
        identifier: STRING
      do
        ......
        transaction.start;                          -- ⇐ Start an explicit transaction
        obj:= pview.item_by_symbol  (identifier);
        if obj.sex = "female" and not obj.husband_wife = void then
           obj.set_name  (obj.husband_wife.p_name)
        else
           -- other statements
        end; -- if
        transaction.finish                          -- ⇐ Ends an explicit transaction
      end; -- possible_concurrent_access
  .......
end -- class CONCURRENT_USE
```

Figure 7.3: Exemple d'utilisation de la gestion explicite des transactions

Dans cet exemple, si on ne surcharge pas les début et fin de transaction qui sont déclenchés par l'exécutif Eiffel, on pourra avoir plusieurs transactions différentes:

- la lecture du sexe de la personne (*obj.sex*),
- la lecture de la référence du conjoint (*obj.husband_wife)*,
- la lecture du nom du conjoint (*obj.husband_wife.p_name*)
- le changement de nom (*obj.set_name*)

Il n'y a pas de problème pour la conservation de l'intégrité des données en cas de panne, mais si d'autres processus interviennent sur le même objet, alors on ne peut plus garantir au moment de l'exécution de *obj.set_name* que les valeurs du sexe, ou du nom du conjoint n'ont pas été modifiées par ces processus. C'est pourquoi, on démarre et on termine explicitement la transaction, afin de garantir l'exclusivité de l'accès aux objets *obj*, *obj.husband_wife* et *obj.husband_wife.name* pendant toute la durée de la manipulation.

### 7.1.4. Echanges pendant une transaction

Pendant le déroulement d'une transaction, il y a un certain nombre d'échanges entre le POM et le processus Eiffel:

- des échanges d'objets (consultation, mise à jour),
- des notifications sur l'occupation des objets.
- des notifications d'erreur (du POM ou du processus),
- des notifications d'accès à un objet obsolète.

L'objectif est de faire en sorte que ces échanges soient le plus transparent possible pour un programme Eiffel. Ainsi, quand il n'y a pas d'erreur mais seulement des échanges d'objets ou des problèmes temporaires d'occupation d'objets, aucun traitement supplémentaire n'est demandé à une application: C'est le problème de l'exécutif, puisque c'est lui qui gère l'accès navigationnel. Par contre, si une erreur non récupérable se produit, alors l'exécutif Eiffel doit en avertir l'application en déclenchant une exception.

#### Notification d'occupation d'objet

Soit un processus Eiffel P1, avec une transaction T1 en cours, soit un processus Eiffel P2, avec une transaction T2, on peut se trouver face à un objet déjà occupé dans les cas suivants:

- T1 demande la lecture d'un objet *obj*, alors que T2 est en train de le modifier,
- T1 demande la modification d'un objet *obj*, alors que T2 est en train de le consulter ou de le modifier.

Conséquences:

Aucune erreur ne s'est encore produite, c'est à l'exécutif Eiffel de décider ce qu'il doit faire en fonction des consignes qu'il aura reçu par l'application; deux solutions sont possibles, il pourra:

- se mettre en attente pendant un certain temps et réessayer un certain nombre de fois, avant de déclencher une exception,
- annuler la transaction et déclencher une exception.

L'application pourra choisir (par exemple, lors de son initialisation), l'une ou l'autre

des solutions, par l'intermédiaire de primitives de la classe *TRANSACTION* (fig. 7.4).

```
class TRANSACTION
feature
  set_abort is
        -- Eiffel run-time will abort transaction as soon as a busy object is accessed

  set_wait is
        -- Eiffel run-time will wait until the busy object is released by other applications
end -- class TRANSACTION
```

Figure 7.4: Augmentation des primitives de gestion des transactions

### Notification d'accès à un objet obsolète

Au chargement, l'exécutif vérifie que l'objet n'est pas obsolète, si c'est le cas une exception nommé *obsolete_object* sera déclenché. On notera que cette vérification n'est plus faite lors des accès suivants à l'objet.

Si on rend un objet obsolète en cours d'exécution on peut tout de même y accèder pendant toute la durée de l'exécution. Pour vérifier qu'un tel objet est obsolète on utilisera la primitive *is_obsolete* de la classe ANY.

### Notifications d'erreurs

On trouve principalement trois types d'erreur:

- Les erreurs logicielles:
  - erreurs de programmation,
  - erreurs de l'utilisateur,
- les erreurs matérielles.

#### Erreur logicielle

Une erreur logicielle peut provenir:

- d'une erreur dans le programme (violation de préconditions, postconditions, invariants, signal Unix),
- d'une erreur d'utilisation qui est détectée par l'application,
- d'une erreur dans le POM.

Conséquences:

Dans les deux premiers cas, l'exception correspondante est directement déclenchée par l'exécutif Eiffel (V.3), alors que dans le deuxième cas, elle est seulement propagée par ce dernier.

Deux cas peuvent alors se présenter:

- l'exception n'est pas récupérée par le processus, la transaction est alors annulée et les objets persistants modifiés ne sont pas mis à jour,
- l'exception est récupérée et c'est à l'application de prendre en charge la gestion de l'erreur.

Dans ce dernier cas, il faut différencier le cas où l'erreur est récupérée au niveau de la méthode (clause *rescue* spécifique à une routine), de celui où l'exception est récupérée au niveau de la classe (clause *rescue* commune aux routines la classe). En effet, si dans le premier cas on peut espérer redresser la situation, cela devient beaucoup moins raisonnable dans le deuxième où il semble préférable d'envisager une terminaison prématurée mais organisée.

**Erreur matérielle**

L'erreur matérielle (panne de la machine, coupure d'alimentation), peut se produire à tout moment de l'exécution du processus (pendant les calculs ou pendant la mise à jour).

Deux cas différents peuvent se produire: Soit, le POM et le processus Eiffel sont sur la même machine, alors tous les deux s'arrêtent; soit le processus ou le gestionnaire d'objets, sont sur des machines différentes et deux situations sont possibles en fonction de la machine sur laquelle est localisée l'erreur:

- le processus s'arrête: le POM détecte qu'il n'y a plus de réponses, et annule la transaction,
- le POM s'arrête: le processus détecte qu'il n'y a plus de réponse et déclenche une exception.

Dans tous les cas, le POM garantit que les objets modifiés reprennent l'état qu'ils avaient avant le début de la transaction.

### 7.1.5. Conséquence des effets de bord sur la cohérence des données

Dans le cas ou une opération s'applique à un seul objet courant, comme dans celui où il s'applique à une collection d'objets, le traitement des objets persistants est identique à celui des objets temporaires. Ainsi une fonction peut contenir des effets de bord sur un objet persistant, mais la gestion de leurs conséquences est à la charge du programme.

On notera cependant qu'en ce qui concerne les traitements sélectifs, la présence de fonction avec effets de bord (notemment si elles mettent à jour le curseur de parcours), peut causer des problèmes plus importants. Ainsi, l'évaluation d'un critère de sélection entraînera l'exécution de ces fonctions sur les objets de la *collection*; et quelque soit le résultat (objet sélectionné ou non), la fonction peut avoir été exécutée (voir chapitre 11).

Dans le cas ou l'objet n'est pas sélectionné et qu'il y a des effets de bord, c'est extrêmement génant car le processus n'a pas connaissance des objets non sélectionnés et n'a aucun moyen d'annuler les modifications.

## 7.2. Protection des objets et des classes

Dans le monde Eiffel (V.3), tous les objets persistants créés sont en général utilisés par des processus de la même application, et sont regroupés dans les fichiers associés à des instances de la classe *ENVIRONMENT* (cf. § 2.3). Il est donc facile de gérer les droits d'accès à ces objets: ils sont définis par ceux du fichier associé à l'environnement.

Notre approche de la persistance, encourage le partage (consultation, modification), des entités persistantes (objets persistants ou classes), par différents processus, correspondant éventuellement à des programmes différents. Ces objets peuvent donc avoir des droits d'accès différents, en fonction de l'utilisateur et de l'utilisation, ce qui implique un grain de protection plus fin basé sur des regroupements d'objets.

Les objectifs sont les suivants:

• Contrôler les droits d'accès aux entités persistantes afin de permettre la confidentialité et le partage des objets et des classes,

• déterminer le contenu des *Pcollections* d'un processus, afin d'offrir des traitements sélectifs efficaces,

• offrir une plateforme pour la gestion de l'aide à la conception et de l'évolution des composants.

On ne peut donc pas se limiter à une gestion des protections basée sur le système de fichiers comme pour la classe *ENVIRONMENT* (Eiffel V.3).

### 7.2.1. Notions de ressource, d'intervenant et d'action

Trois notions interviennent dans la gestion des protections et du partage des objets:

- les ressources à protéger (classe, objet, programmes),
- les intervenants (utilisateurs, processus), contre qui protéger les ressources,
- les actions pouvant être faites sur une ressource par un intervenant.

### 7.2.2. La classification des intervenants

Il existe plusieurs catégories d'intervenants: les processus, et les utilisateurs (développeur d'application, chef de projet, utilisateur final de l'application).

On remarque qu'un processus est activé sous la responsabilité d'un utilisateur, et qu'un processus a un durée de vie limitée. On peut donc confondre les notions d'utilisateur et de processus.

Vis à vis d'une entité persistante, tout utilisateur dispose d'une qualité: il est, ou n'est pas administrateur de l'entité. Les administrateurs jouent un rôle particulier, car ils ont le droit de modifier les droits d'accès aux ressources; à la création de l'objet, seul le créateur est adminstrateur. Les autres peuvent être différenciés en fonction de leurs aptitudes à modifier les entités persistantes, et de leurs rapports avec les propriétaires des ressources.

Nous classons les utilisateurs en groupe comme sous UNIX où on distingue trois types d'utilisateurs: le propriétaire (administrateur initial), ceux appartenant au même groupe que lui (utilisateurs proches), et les autres. L'inconvénient de cette approche est la difficulté qu'il y a de gérer les groupes, car en général, on appartient à des groupes différents en fonction du projet sur lequel on travaille; dans le monde UNIX des travaux sont actuellement menés pour permettre d'individualiser les droits d'accès (chaque utilisateur a ses droits).

### 7.2.3. La classification des ressources

Il existe donc plusieurs sortes de ressources: le programme, la classe, et l'objet. Un programme représente l'unique moyen d'accéder au monde Eiffel, à partir du système hôte où sa représentation est un fichier exécutable. Etant donné son rôle de point d'entrée il semble préférable de laisser la gestion de ses protections au système hôte, même si ce choix affecte l'orthogonalité de la gestion des entités persistantes.

Cela permettra aussi de rendre possible l'activation d'un processus avec les privilèges du propriétaire du programme, quand ce mécanisme est pris en compte par le système d'exploitation.

Les protections d'une classe concernent surtout l'évolution des composants, c'est à dire le point de vue du développeur. L'introduction d'un mécanisme de protection au niveau des classes, devra permettre d'empêcher ou de réglementer l'utilisation et éventuellement la modification d'une classe.

On a vu (cf. § 5.2), que pour un programme Eiffel, une classe est représentée comme un objet persistant (instance de la classe *EIFFEL_CLASS*), au moins pour la consultation de ses caractéristiques, ce qui laisse apparaître la possibilité d'un traitement quasi-orthogonal des protections pour les objets et les classes. Cependant, comme on ne peut pas vraiment comparer le nombre d'objets persistants avec le nombre des classes. Il semble donc raisonnable de différencier le traitement des protections: il sera possible d'individualiser les protections au niveau de la classe, tandis qu'au niveau des objets on fera correspondre les protections à un groupe d'objets.

#### Remarque

Le fait d'associer un masque de protection à tout objet a été envisagé mais n'a pas été retenu pour des raisons de souplesse d'utilisation et surtout de coût (espace utilisé, efficacité).

### 7.2.4. Les actions possibles sur une ressource

Comme on vient de le voir, toutes les entités persistantes, sont des objets appartenant au *monde persistant;* leur accès doit donc être soumis à un contrôle qui tienne compte de la nature de l'action; on en recense actuellement quatre: la consultation, la modification, la suppression, et le changement des droits d'accès d'un objet persistant.

La définition suivante permet de formaliser les liens entre une ressource, un intervenant et les actions que celui-ci peut effectuer.

**Définition 24: Une ressource**

*Une ressource est une entité persistante Eiffel, qui est, vis à vis d'un intervenant:*

- *visible / invisible,*
- *modifiable / non modifiable,*
- *effaçable / non effaçable.*

### 7.2.5. Les contrôles: pour qui? quand?

Le contrôle de l'accès aux entités persistantes doit être réalisé à toutes les étapes du cycle de vie d'une application; son résultat dépend de l'utilisateur qui est responsable du processus Eiffel leur faisant référence.

Ainsi, dans les phases de conception et de modification d'une application, c'est à dire quand on utilise des outils de l'environnement de programmation associé au langage (traducteurs, outils de recherche de composants, ou d'aide à la conception), il faut contrôler l'accès aux classes et vérifier, par exemple, qu'un programmeur a effectivement le droit de réutiliser une classe particulière.

Dans la phase d'exploitation, il faut vérifier que l'utilisateur a le droit d'accéder aux objets persistants manipulés par l'intermédiaire d'une application figée. Naturellement, ces instances peuvent être éventuellement du type *EIFFEL_CLASS*. On n'a plus besoin dans cette étape, de vérifier les droits d'accès, pour l'utilisateur final, aux classes qui ont servi à la réalisation du programme; en effet, la vue que l'utilisateur de l'application a des classes (et non des instances), est celle que lui donne l'application (l'utilisation des classes a déjà été controlé lors de la traduction).

**Conséquences**

• le contrôle de l'accès aux classes d'un programme se fait seulement lors d'une consultation ou d'une modification de ces classes (le plus souvent dans des phases de traduction ou de conception); les performances des processus utilisant des objets persistants, n'en sont donc pas affectées,

• le seul type d'utilisateur à considérer pour le contrôle de l'accès aux classes, est le développeur d'application, sauf dans le cas d'application comme par exemple des explorateurs de classes.

### 7.2.6. Intégration du mécanisme en Eiffel

Pour permettre l'intégration de ce mécanisme en Eiffel, nous nous appuyons sur le concept de *pview* (cf. § 6.8.3). Les protections ne sont pas associées aux objets mais à des groupes d'objets. Ainsi **la protection d'un objet dépendra des droits d'accès définis au niveau d'une *pview***.

**Définition 7.4: Droits d'accès sur les objets**

*Un objet sera lisible et/ou modifiable et/ou effaçable si et seulement si, la pview associée au processus, permet de l'atteindre, et est elle-même lisible et/ou modifiable et/ou effaçable.*

La protection d'une classe est indépendante du concept de *pview*. La raison provient du fait que les instances d'une même classe peuvent être regroupées dans plusieurs vues pouvant être ou pas reliées par des liens d'inclusions (cf. § 6.8.3).

**Définition 7.5: Droits d'accès sur les classes**

*Une classe a les facultés d'être lisible et/ou modifiable et/ou effaçable individuellement, indépendemment de la pview par laquelle elle est atteinte.*

**Contraintes d'intégration**

• **C7.6:** Si pour un utilisateur U, la *pview* V n'est pas lisible, alors quelque que soient les droits d'accès de la classe, les informations de celle-ci ne seront pas visibles à partir de cette *pview*;
• **C7.7**: les droits d'accès d'une pview ne sont modifiables que par un des propriétaires

de la pview;

• **C7.8**: le *monde persistant* est toujours lisible et modifiable par tous les utilisateurs;

• **C7.9**: un processus ne peut inclure une *pview* V2 dans une autre *pview* V1, que si V1 est visible par lui, et si V2 est modifiable par lui;

• **C7.10**: en ce qui concerne l'inclusion des *pviews*, il se peut que certaines *pviews* incluses dans celle associée au processus à l'initialisation de l'exécution ne soient pas visibles / modifiables par le processus. Les objets qu'elles contiennent ne pourront donc pas être visibles / modifiables par le processus.

• **C7.11**: un utilisateur ne peut effacer une *pview* que s'il en est propriétaire.

**Cas d'erreur**

Si un objet n'est pas visible / modifiable par le processus, alors une exception sera déclenché; le nom de cette exception est *permission_denied* (voir chapitre 10).

## 7.2.7. Intégration dans le schéma de conception

```
Class PVIEW
      .....
feature
   is_readable: BOOLEAN is              -- Is view readable by the user of the Eiffel process

   is_writable: BOOLEAN is              -- Is view  writable by the user of the Eiffel process

   is_owner: BOOLEAN is                 -- Is view owned by the user of the Eiffel process

   set_access_right (rights: STRING) is -- set access rights if user is available to do it
      require   is_owner

   reset_default is
         -- reset default access rights (readable for all, writable by owner's group)
      require   is_owner
   .....
end -- class PVIEW
```

Figure 7.5: Protections associées à une pview du monde persistant

## La protection des objets

Puisque les droits d'accès d'un objet repose sur le concept de pview, il est naturel de permettre la modification des droits d'accès à partir de la classe PVIEW (fig. 7.5).

## La protection des classes

Les droits des classes sont gérées individuellement pour une classe, ces droits sont donc modifiables à partir de la classe *EIFFEL_CLASS* (fig. 7.6); une seule instance existe pour chaque classe, elle est partagée par toutes les instances de *PVIEW* qui y ont accès.

```
Class EIFFEL_CLASS
      .....
feature     ........
  is_readable: BOOLEAN is      -- Is view readable by the user of the Eiffel process
  is_writable: BOOLEAN is      -- Is view writable by the user of the Eiffel process
  is_owner: BOOLEAN is         -- Is view owned by the user of the Eiffel process

  set_access_right (rights:  STRING) is
        -- set access rights with string if user is available to do it
      require   is_owner

  reset_default is
        -- reset default access rights (readable for all, writable by owner's group)
      require   is_owner
end -- class EIFFEL_CLASS
```

Figure 7.6: Protections associées à une classe

Remarques:

Bien entendu, la bonne approche serait de créer un niveau d'abstraction supplémentaire et de regrouper les primitives de protection dans une classe retardée appelée *ENTITY_PROTECTION*. Par ailleurs il serait peut être intéressant, au niveau d'une *pview*, de permettre d'associer des protections à un objet associé à un symbole, ce qui permettrait d'individualiser plus encore le traitement des protections.

## Le cas du monde persistant

Les droits d'accès du *monde persistant* sont positionnés lors de l'installation de l'environnement Eiffel; par défaut, le monde persistant est toujours accessible en lecture à tous les utilisateurs Eiffel. On notera que comme son propriétaire est l'utilisateur ayant installé l'environnement Eiffel; lui seul est donc autorisé à le supprimer

# Partie III

# Mise en oeuvre de la persistance

Dans cette partie nous décrivons les choix techniques de la mise en oeuvre de la conception précédente. Les points importants concernent:

• la propagation dynamique ou statique des types en mémoire persistant, avec en particulier le cas des routines;

• l'adaptation de l'exécutif du langage Eiffel pour prendre en compte l'exécution automatique des échanges en les mémoires persistante et volatile (chargement, libération, mise à jour des objets);

• l'efficacité, avec les améliorations suivantes:
• gestion de l'image des objets persistants en mémoire volatile,
• chargement en mode navigationnel,
• sélection d'objets persistants;

• la conversion entre des représentation des classes et des instances, entre le gestionnaire d'objet (POM) et de langage Eiffel.

Après une étude sur les techniques utilisées par les langages existants, nous donnerons les architectures possibles, puis nous développerons individuellement ces différents points.

# 8

# Mise en oeuvre dans les langages

Dans la première partie de ce travail, nous avons montré qu'une bonne intégration passait par une gestion transparente des objets persistants; dans le paragraphe 2.2.3, nous avons développé l'idée que cette transparence nécessitait l'adaptation de l'exécutif du langage pour gérer automatiquement et de manière incrémentale, les échanges entre les mémoires volatile et persistantes. Ces adaptations supposent que l'on sait répondre aux questions suivantes:

- quelle technique utiliser pour implémenter un gestionnaire d'objets persistants (POM), et est-ce le POM ou l'exécutif du langage qui doit gérer l'exécution d'une routine sur un objet persistant?

- comment identifier un objet persistant, et comment mettre en oeuvre des échanges incrémentaux entre l'application et le gestionnaire d'objets?

- comment et quand propager le type d'un objet dans le monde persistant?

- comment optimiser les échanges entre l'application et le gestionnaire d'objets, en cas d'accès navigationnel ou associatif?

D'autres aspects, bien connus des personnes travaillant dans le domaine des bases de données, doivent être traités par le gestionnaire; il s'agit de:

- la préservation de l'intégrité des données (transaction),
- la protection des entités persistantes,
- le partitionnement du monde persistant (espace de noms),
- la gestion du type des objets,
- les mécanismes d'optimisation pour des accès associatifs.

Dans ce chapitre, nous présentons ces différentes techniques, telles qu'elles apparaissent dans la littérature.

# 8.1. Implémentation d'un gestionnaire d'objets persistants

Nous avons montré dans le paragraphe 2.2.2 que l'on pouvait considérer plusieurs niveaux de persistance; le niveau de persistance que l'on veut atteindre dépend essentiellement des fonctionnalités offertes par le gestionnaire d'objets utilisé pour l'implémentation.

Ainsi pour intégrer la persistance dans un langage de programmation à objets (stockage des objets, accès associatif, aspect transactionnel, gestion des types dans le monde persistant, ...), il nous semble raisonnable de ré-utiliser le savoir faire des bases de données. Ce savoir faire peut être introduit soit à travers le noyau d'un **serveur d'objets**, soit par un **système de gestion de base de données à objets complet** (*$O_2$*,, *Gemstone*, *Ontos*,, ....).

### Utilisation d'un serveur d'objets

Un serveur d'objets est un système qui gère le stockage et la récupération d'objets persistants sans faire aucune hyppothèse sur la sémantique des objets; il stocke seulement des suites d'octets. Une possibilité est donc que le gestionnaire d'objets soit construit à partir d'un serveur d'objets; le modèle de données qu'il supporte peut (doit) être celui du langage de programmation.

Un tel gestionnaire d'objets serait donc très bien adapté aux besoins du langage de programmation (un seul modèle de données), et permettrait une plus grande efficacité en évitant les conversions de modèle à modèle, en facilitant l'évolution du gestionnaire par rapport à celle du langage, et en évitant les problèmes de compatibilité entre modèles d'exécution. Le principal inconvénient est le coût de la réalisation d'un tel gestionnaire, puisque la couche à définir au dessus du serveur d'objets nécessite:

- l'adaptation des mécanismes d'optimisation (mécanismes d'indexation adaptés à l'utilisation d'expression pointées, et au polymorphisme),
- la modélisation des liens entre les structures (notamment lien d'héritage et de clientèle),
- la définition des mécanismes de sélection,
- la représentation et l'exécution des routines en mémoire persistante.

Parmi les serveurs d'objets existant dans la litérature, citons:

• *WISS* [Chou 86], le gestionnaire de disques utilisé par le système *O2*,

• *KALA* [Simmel 91], un gestionnaire de stockage transactionnel permettant de gérer des données non typées,

• *NICEMP* [Salgado 88], le gestionnaire d'objets de *NICE-C++* [Mopolo-Moke 91],

• *BOSS* (Binary Object Stream Service) [pps 90], un gestionnaire d'objets pour le langage Smalltalk. Il permet d'augmenter les possibilités du langage notamment en ce qui concerne les échanges incrémentaux.

**Utilisation d'un système de gestion de base de données**

Dans ce cas, l'intégration est plus faible, puisque le modèle du gestionnaire est celui d'un SGBD existant (par exemple celui de *O2, Oracle, Orion, Gemstone, ou Ontos*), qui est donc différent de celui du langage de programmation.

La couche à construire au dessus du SGBD ne demande pas de résoudre les problèmes de bas niveau, mais du fait de la présence de deux modèles (celui du SGBD et celui du langage), nécessite la mise en oeuvre d'un mécanisme de conversion élaboré pour passer d'un modèle à l'autre. Ceci est surtout vrai en ce qui concerne les classes.

Ce mécanisme de conversion, utilisé pour la propagation des types au POM, et pour les échanges d'objets persistants sera d'autant plus simple à réaliser que les modèles seront proches l'un de l'autre. Ainsi, un gestionnaire d'objets basé sur un SGBD relationnel nécessitera un mécanisme de conversion plus complexe qu'un gestionnaire utilisant les services d'un SGBD orienté objets et ne pourra prendre en compte qu'une partie du modèle (classes sans routines).

Une des conséquence sera naturellement une perte au niveau de l'efficacité et de la compatibilité des modèles. A priori, l'utilisation d'un gestionnaire implémenté à partir d'un SGBD ne pourra constituer qu'un prototype validant l'approche suivie pour l'intégration; nous essaierons cependant de montrer qu'il est possible, en prenant certaines précautions, d'avoir une efficacité raisonnable qui permet de rendre l'outil opérationnel pour des applications industrielles.

## 8.2. Identification des objets

Quand on évoque l'introduction d'objets persistants, il apparait tout de suite le problème de l'identification. On ne peut plus identifier un objet par son adresse puisque celle-ci devient obsolète dès que l'exécution de l'application se termine; on doit donc mettre en oeuvre un identificateur d'objet persistant noté *POI* dans notre approche.

Pour dèsigner cet identificateur les systèmes utilisent des noms différents: **Persistent identifier** (*PID*) pour PS-Algol, Pgraphite et Lispo2, **Object Identifier** (*OID*) pour le langage E, **long Object pointer** (*OOP*) pour Smalltalk associé à Loom[1] .

Les différences entre les identificateurs concernent surtout leur taille dont dépend essentiellement la grandeur de l'espace adressé.

Par exemple, dans *PS-Algol*, le *PID* est constitué d'un numéro de base de données (possibilité d'en ouvrir 256 simultanément), d'un numéro du bloc logique, de la position de l'objet dans ce bloc.

Dans *Smalltalk* (nous nous intéressons à la version implantée avec *Loom* [Kaelher 90]), le nombre d'objets maximum d'objets est $2^{31}$ car l'identificateur est décrit sur 4 octets; on a jamais besoin de se préoccuper dans quel réservoir de données (*repository*), est stocké un objet car à un moment donné, un seul est ouvert, et un objet ne peut référencer que des objets se trouvant dans ce réservoir.

## 8.3. Gestion incrémentale des objets persistants

Dès que l'on introduit des échanges incrémentaux entre la mémoire persistante (gérée par un gestionnaire d'objets) et une application, il se pose le problème du partage des objets; en effet, quand il est possible d'accéder à un objet suivant plusieurs chemins (voir fig. 8.1), il faut introduire un mécanisme qui permette à l'exécutif de déduire qu'une copie de l'objet se trouve déjà en mémoire volatile.

En effet, si après un premier accès à *d3* en passant par *d1,* on accède à nouveau à *d3,* mais cette fois ci par *d2*, alors on risque d'avoir deux copies de l'objet *d3*. nous montrerons comment cet aspect est traité dans les différents langages.

---

[1] Loom est un gestionnaire d'objets virtuels permettant de gérer $2^{31}$ objets smalltalk au lieu de $2^{16}$ dans la version basé sur Ooze (l'image d'exécution ne peut contenir que $2^{16}$objets).

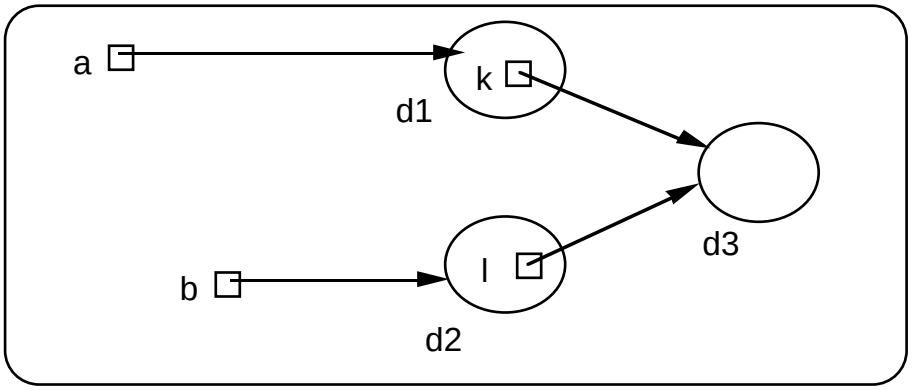


Figure 8.1: Partage des objets persistants dans un même processus

### 8.3.1. Correspondance entre les adresses en mémoire volatile et persistante

En général, l'exécution de primitives nécessite des manipulations en mémoire volatile (cf. § 8.4), et donc le chargement des objets concernés. Par ailleurs, nous avons montré dans le paragraphe 8.2 la nécessité d'associer un identificateur persistant (*POI*) à un objet persistant. Il faudra donc établir la correspondance entre l'adresse en mémoire persistante et l'adresse en mémoire volatile, afin de pouvoir déterminer si un objet est chargé et effectuer la "synchronisation" entre l'objet en mémoire persistante et sa copie en mémoire volatile.

Dans *PS-Algol*, la correspondance entre les adresses dynamiques et les adresses en mémoire secondaire est réalisée à partir d'une table appelée *PIDLAM* (Persistent Identifier to Local Address Map) qui est composée de cellules à deux éléments (l'une contenant l'identificateur persistant, et la seconde l'adresse en mémoire volatile). L'accès aux éléments de cette table se fait à l'aide de deux index permettant d'obtenir l'adresse locale à partir du PID et vice versa [Cockshot 90].

Dans le langage *Smalltalk* géré par *Loom* [Kaehler 90], un objet en mémoire volatile est référencé par un identificateur plus petit que l'identificateur en mémoire persistante (*Long OOP*), il est nommé *Short OOP*, un *Long OOP* a une taille de 32 bits, tandis qu'un *Short OOP* a une taille de 16 bits. Le but de cette conversion est de mettre en oeuvre un mécanisme de mémoire virtuelle permettant de gérer $2^{32}$ objets, contre $2^{15}$ objets dans une première implémentation, basée sur le système *Ooze* [Goldberg 89, Kaehler 81].

Le partage des objets est réalisé par la mise en oeuvre d'une table appelée *resident object table* (ROT). Cette table est indexée sur le *short OOP*, et chacune de ses entrées contient l'adresse du corps de l'objet, un compteur de référence et des bits d'information.

Dans le *langage E* et Exodus [Richardson 89 a & b], le problème de partage est résolu à la compilation, et sera abordé en même temps que la gestion des échanges avec la mémoire persistante, dans le paragraphe suivant.

### 8.3.2. Chargement, libération et mise à jour des objets

Dans la version de *Smalltalk* gérée par *Loom* [Kaehler 81 & 90], le chargement d'un objet se fait quand un défaut d'objet est constaté, c'est à dire lorsque l'objet atteint n'est pas déjà en mémoire. Lors du chargement les *Long OOP*, qui représente les références entre objets lorsqu'ils sont en mémoire persistante, sont convertis en *Short OOP*, et une entrée est créée dans la table *ROT*. les *Short OOP* et les *Long OOP* étant reliés par une formule de hâchage, il est facile, lors de la création des entrées dans la table, de vérifier que l'objet n'est pas déjà chargé et donc d'assurer l'unicité des copies.

La libération d'un objet ne peut intervenir que lorque celui-ci n'est plus référencé; la technique utilisée par le ramasse-miettes de Smalltalk pour s'apercevoir qu'un objet n'est plus référencé est le comptage des références. Par contre, pour éviter la saturation de la mémoire, un objet chargé en mémoire volatile pourra devenir une *"leave"* (objet de taille réduite, cf. § 8.3.3).

Dès qu'une affectation est effectuée sur un de ses champs, l'objet est considéré comme étant modifié, et cette information est gardée dans la table *ROT*; la mise à jour en mémoire persistante se fait automatiquement par le système, grâce à cette information.

Dans *PS-Algol* [Cockshot 90], les champs d'un objet contiennent, tantôt une adresse en mémoire volatile, tantôt un identificateur d'objet persistant (PID). Lors de l'accès à l'un des champs, l'exécutif teste si son contenu est un PID ou une adresse locale. Dans le cas où c'est un PID, la table PIDLAM est consultée pour vérifier si l'objet n'a pas été chargé par ailleurs: S'il ne l'est pas, il est chargé à partir de la mémoire persistante.

Lorsque la transaction en cours est validée, l'exécutif fait l'inventaire des objets modifiés et les met à jour sur le disque; le système garantit la cohérence des données en cas de panne (mécanisme de double copies). La libération des objets persistants se fait par l'intermédiaire d'un ramasse-miettes.

Dans le langage E, on ne charge pas des objets mais seulement la valeur de l'attribut utilisé dans l'expression. Les ordres de chargement et de libération avec mise à jour éventuelle sont prévus lors de la compilation après l'examen des expressions. Par exemple, on sait que l'expression retournant la valeur *a.b + a.d* ne provoquera pas de modification; mais si a est un objet persistant, alors un ordre de chargement devra être

prévu pour les valeurs b ou de d. Par contre, une expression *x = a.b + a.d*, provoquera une modification de x (s'il est persistant), et donc nécessitera la génération d'un ordre de mise à jour.

Le déclenchement de ces ordres est fait par dérepérage des opérateurs arrow (->), star (*) ou dot (.), à l'exécution. On rappelle que E dérive de C++ et donc supporte ses opérateurs et types de données.

La libération des valeurs, et éventuellement leur mise à jour, est faite dès la fin de l'expression ayant nécessité leur chargement.

Le langage E n'utilise donc pas de ramasse-miettes, ni n'effectue le chargement de l'objet complet; on comprend ainsi pourquoi l'exécutif de ce langage n'a pas besoin de gérer le partage des objets.

Les systèmes de gestion de base de données comme $O_2$ ou *Gesmtone*, et les serveurs d'objets *NICEMP* [Salgado 88], ou *WISS*, disposent d'une mémoire cache comme pour la gestion des mémoires virtuelles. Certains provoquent le chargement sur défaut d'objets, tandis que d'autres ont besoin d'un ordre explicite.

L'originalité de *Galileo* [Albano 87], repose sur une tentative d'optimisation des échanges entre l'application et le gestionnaire d'objets (cf. § 8.3.3). Le partage des objets dans une application se fait aussi à partir de tables d'adressages. La libération des objets est basée sur un ramasse-miettes et la mise à jour sur un mécanisme transactionnel dont la validation est déclenchée périodiquement.

### 8.3.3. Optimisation des échanges

Lorsqu'un objet est chargé dans *Smalltalk*, (associé à Loom), pour chaque champ référençant un autre objet, il faut éventuellement créer une entrée dans la table ROT. Naturellement, les cycles de référence sont détectés. Pour diminuer les échanges inutiles, deux possibilités sont offertes:

- faire correspondre à ces nouvelles entrées un objet (*leave*) de taille réduite (4 mots) qui contient principalement le *Long OOP* de l'objet. Cette solution favorise les performances, dans le cas où l'objet est effectivement utilisé par le programme, pour un coût en espace mémoire relativement raisonnable. S'il ne l'est pas, on a consommé inutilement de l'espace et du temps dans les échanges;
- associer aux champs un *Short OOP* spécial (*lambda*), indiquant que l'objet n'est pas chargé et qu'il faut aller récupérer son *Long OOP*, et sa valeur. Cette solution est

particulièrement adaptée lorsque l'objet n'est pas demandé par le programme: on économise sur le temps des échanges et sur l'espace mémoire; par contre le coût est plus important que dans le cas précédent si l'objet est effectivement utilisé.

Le choix du mode de chargement utilisé dépend de l'historique des échanges: si un champ était un *lambda* lors de la dernière mise à jour de l'objet en mémoire persistante, alors il le sera à nouveau lors d'un prochain chargement (présence d'un indicateur dans chaque champ).

Dans Galiléo [Albano 87], on fait l'hyppothèse qu'une application réalise la plupart de ses traitements sur les objets chargés les plus récemment. Ces derniers sont placés dans un tampon spécial où l'identificateur persistant coincide numériquement avec son adresse en mémoire volatile et ainsi permet un accès immédiat; on notera que ces objets ne sont pas libérés par le ramasse-miettes. Cette technique se révèle peu adaptée lorsque l'on considère les fonctions comme des objets.

## 8.4. Exécution des routines sur les objets

L'éxécution des routines qui manipulent des objets persistants peut être effectuée, soit sous le contrôle du gestionnaire d'objets, soit sous le contrôle de l'exécutif du langage, comme pour les objets volatiles.

Dans le premier cas il faut "rendre les routines persistantes", c'est à dire les traduire dans le langage associé au gestionnaire d'objets (ex: $O_2$), ce qui élimine toute solution à base de SGBD qui ne disposent pas de la définition de routines, notamment les SGBD relationnels.

Dans le second cas, on est confronté à des problèmes d'efficacité, surtout pour les accès associatifs. Ceci nécessite la mise en oeuvre de mécanismes ad hoc de conversions de représentation entre les mémoires volatiles et persistantes (voir chapitre 11). Pour les accès navigationnels (application d'une routine sur l'objet courant), la perte d'efficacité est beaucoup moins grâve, puisque le coût de la conversion touche un nombre limité d'objets. Il peut être intéressant de combiner les deux approches et d'utiliser les routines persistantes seulement pour des accès associatifs.

En théorie, les systèmes de gestion orientés objets qui permettent la construction de routines par les utilisateurs n'ont pas ce problème d'efficacité, puisque ce sont eux qui gèrent leur exécution des routines. Ils doivent alors réaliser la liaison dynamique entre

les routines et les objets par l'intermédiaire de leurs identificateurs persistants et optimiser l'accès et le chargement des objets. C'est le cas de *O2*, *Gemstone* et de *Vbase* [Andrews 91 b].

De même, des systèmes comme *PS-Algol* et *Napier 88*, sont conçus au départ comme des langages persistants. Leur exécution gère la liaison dynamique selon une étude du polymorphisme [Morrison 91].

Dans le *langage E*, l'exécution des routines se fait comme en C++; la liaison dynamique entre une routine et un objet est réalisée par des tables, initialisées à partir d'informations rassemblées et synthéthisées par le traducteur.

Quant à *Smalltalk*, le choix est laissé à l'utilisateur: soit l'application demande une exécution locale et la routine est incluse dans l'application qui pilote alors son exécution, soit l'application demande l'exécution de la routine à distance, et c'est alors le gestionnaire d'objets qui dirige l'exécution. Dans le premier cas, les objets concernés par l'exécution sont chargés dans la mémoire volatile associée à l'application, tandis que dans le deuxième, ces objets restent sous le contrôle du gestionnaire. Le choix de la routine se fait en accord avec le type dynamique de la propriété (liaison dynamique).

## 8.5. La propagation des types dans le monde persistant

Dans *Galileo* [Albano 87], la solution adoptée pour gérer la persistance des types est de ré-écrire le compilateur de Galileo en utilisant Galileo lui-même. Le langage intégrant la persistance gèrera ainsi les types comme des entités persistantes; cette approche est parfaitement uniforme et nous sommes en accord complet avec cette démarche, surtout qu'Eiffel V.3 est écrit en Eiffel (voir chapitre 9).

Dans les systèmes de gestion de base de données comme *Gemstone*, *Vbase/Ontos*, *O2*, ou *Orion*, les propriétés d'un type sont propagées dès leur définition (approche interprétative); les méthodes sont écrites dans un langage autre que celui utilisé pour la description des structures; ce n'est pas le cas pour *Gemstone* qui est une extension de *Smalltalk*. Ces langages dérivent de C pour *O2* et *Vbase*. Lors de leur traduction, les méthodes sont propagées dans le monde persistant.

## 8.6. Intégrité des objets

Dans Galileo [Albano 87], on considère trois types de pannes:

• les pannes prévisibles par l'application, qui sont provoquées par l'utilisateur de l'application, pendant son exécution.

• les pannes imprévisible provoquées par l'application elle-même,

• les pannes matérielles.

Les premières sont gérés par un journal associé à une commande *undo*; le *redo* a été abandonné à cause de son coût en espace et en temps, et l'interférence qu'il avait avec le ramasse-miettes et l'allocateur de mémoire. Les secondes sont gérées avec le mécanisme classique de double-copies [Salgado 88, Mopolo-Moke 91, Cockshott 90], activé périodiquement et combiné avec la commande *undo*. Pour prendre en compte les erreurs matérielles, un mécanisme de copie incrémentale en arrière plan permet de faire des sauvegardes complètes et périodiques du monde persistant; les principaux cas de reprise sur panne sont étudiés dans [Bouchet 79].

Dans *Ontos* [Andrews 91b], outre le mécanisme classique de transactions, on remarque:

• d'une part que les transactions peuvent être imbriquées, ce qui facilite l'implantation d'une opération *undo*,

• d'autre part que chacune de ces transactions peut être partagées par plusieurs processus.

Par ailleurs, comme nous l'avons vu (cf. § 7.1.4), il existe différents cas de blocages sur lecture ou écriture par différents processus. *Ontos* propose de décider au commencement de la transaction comment réagir en cas de tels conflits:

• attendre et re-essayer plus tard,
• annuler la transaction,
• continuer l'exécution sans avoir obtenu l'objet.

Dans *PS-Algol*, le mécanisme de transaction est classique et repose sur un mécanisme de double copies. Par contre Napier 88 ne supporte pas encore un tel mécanisme.

En Smalltalk, il n'y a pas de mécanisme de transaction, cependant celle-ci peut être

simulée par une sauvegarde du réservoir d'objet; en cas de panne il est possible de re-exécuter des modifications à partir d'un journal maintenu automatiquement. Par contre, Alltalk [Straw 89] et *O2-C++* [O2-Technology 91 a], proposent un vrai support (classique) de transactions.

## 8.7. Protection des objets

Dans *Napier 88*, les objets sont organisés sous la forme d'arbres de collection de noms. Pour accéder à l'un de ces environnements, il faut suivre les noeuds de l'arbre. Pour introduire un accès protégé vers un environnement il suffit de l'encapsuler dans une *procédure de 1ère classe* qui accepte en paramètre un mot de passe.

Dans les systèmes de base de données, les mécanismes de protections s'appliquent à différentes catégories d'entités (une base de données, un groupe d'objets, un groupe de classes, une classe, une méthode, ...), et permettent de donner certains privilèges aux utilisateurs.

Par exemple, dans *O2*, les privilèges concernent les consultations et les modifications de schémas (treillis de classes), de bases, de classes, d'objets nommés (racines de persistance), de méthodes, et de collections d'objets. Le titulaire de ces privilèges est soit le propriétaire de l'entité à protéger, soit un membre du groupe (au sens UNIX) du propriétaire, soit un utilisateur quelconque.

## 8.8. Partitionnement du monde persistant

Parce que toutes les données ne correspondent pas aux mêmes objectifs d'utilisation on a très tôt éprouvé le besoin de partitionner les données.

Le système de fichiers est un modèle rudimentaire de partitionnement des données persistantes; il est utilisé pour toutes les intégrations de la persistance de niveau 1 (cf. § 2.2.2).

Par contre, les langages gérant la persistance de manière plus évoluée (niveau 2 et 3), comme *Napier 88* [Atkinson 87 a, Morrison 89] introduisent souvent le concept d'espace de noms, où les noms permettent d'identifier des racines de persistance.

Dans les SGBD, l'unité de partitionnement est plutôt la **base de données**; c'est le cas dans *O2* qui permet non seulement le partitionnement mais aussi le partage des objets par plusieurs bases de données [O2-Technology 91]

# 8.9. Discussion sur le choix des solutions

### Serveur versus SGBD

Dans le paragraphe 8.1, nous avons mentionné les avantages et inconvénients des deux solutions (serveur ou SGBD). Dans le cadre du projet *Business Class*, notre objectif est plus de construire un prototype qu'un produit fini industriel; le gestionnaire d'objets sera donc basé sur le système de gestion de base de données *O2* [O2-Technology 91]. Notre choix s'est porté sur *O2* pour les raisons suivantes:

- son modèle de classes (héritage, lien client-fournisseur), est très proche de celui d'Eiffel. Ceci est important pour la propagation d'une classe Eiffel dans le monde persistant;

- il est écrit en C et C++, et génère du C, comme en Eiffel 3.1, ce qui facilite leur communication;

- il offre une série d'interfaces (*C*, *C++*, *O2-C*, *O2-Engine*, ...) qui permettent d'accéder au système O2 à plusieurs niveaux en fonction du problème considéré;

mais aussi pour d'autres raisons:

- *O2* et *Eiffel* sont des produits commerciaux reconnus dans le monde scientifique;
- la présence de relations entre Altair / O2-Technology et I3S;
- *O2* et *Eiffel* sont des produits européens, ce qui est nécessaire pour des projets ESPRIT comme *business class*.

### Identification des objets

Le coût de l'identification des objets dépend essentiellement de la taille de l'identificateur, et donc du nombre d'objets que l'on veut pouvoir gérer; cette taille dépend principalement du gestionnaire d'objets choisi.

Parfois le gestionnaire ne laisse pas forcément voir l'identificateur persistant, mais laisse manipuler simplement un descripteur d'objets en mémoire, dont la durée de vie peut être limité à une transaction, comme le cas *O2*.

Notre technique d'intégration est paramétrée pour s'adapter à ce problème et à toute

taille d'identificateurs; le prix à payer est une réactualisation des descripteurs d'objets après une transaction, et le passage par une indirection supplémentaire (descripteur d'objet vers identificateur persistant).

### Echanges incrémentaux

Nous avons étudié principalement trois approches: celles de Smalltalk, PS-Algol, et E. Dans la première, le chargement des objets se fait sur un défaut d'objets et chaque objet est chargé complètement, tandis que la libération se fait par l'utilisation d'un ramasse-miettes. L'avantage de cette solution est, qu'à part des modifications de l'exécutif, elle ne demande pas de modifications significatives du traducteur et permet de différer la mise à jour des objets. Par contre, cette solution altère les performances, dans le cas où un objet n'est consulté ou modifié qu'une seule fois.

Le langage E, ne charge jamais un objet mais un champ de l'objet et il le libère (en le mettant éventuellement à jour), tout de suite. Cette technique suppose une modification plus profonde du traducteur afin d'analyser les expressions du programme et de générer les appels (entrée/sortie) nécessaires. Mais, en contre partie, selon les optimisations mise en oeuvre lors de la phase de traduction, on peut s'attendre à des performances meilleures (pas besoin de ramasse-miettes, flot des échanges largement réduit).

La solution que nous avons adopté suit l'approche de Smalltalk ou PS-Algol; la raison essentielle est que nous voulons ajouter un service supplémentaire aux développeurs d'Eiffel, avec le moins possible de contraintes pour leurs implémentations. Par ailleurs, l'un de nos objectifs (cf. § 3.1), est aussi l'orthogonalité de la persistance par rapport au système de types d'Eiffel. Or la solution du langage E suppose, comme on l'a vu dans le paragraphe 4.1.1, des ajouts dans la syntaxe du langage.

### Propagation des types

Les langages que nous avons cités n'ont pas vraiment ce problème puisqu'ils ont été conçus pour intégrer la persistance. Cette propagation est donc toujours faîte au moment de la traduction. D'autres solutions, comme celle de différer cette propagation au moment de l'exécution peuvent être envisagées (chapitres 9 et 12); mais celle-ci risque de pénaliser fortement les performances de l'application, et se prête mal à la gestion de l'évolution des composants, surtout dans un univers où plusieurs développeurs travaillent en parallèle.

Nous avons, dans le cadre de ce travail, expérimenté deux solutions:

• la propagation pendant l'exécution,
• et la propagation avant l'exécution (pouvant être intégrée au moment de la compilation).

Cette dernière solution devrait nous permettre d'intégrer des optimisations qui autorisent des performances convenables.

**Exécution des routines**

Dans les solutions que nous avons développées (solutions statique et dynamique), en cas d'accès navigationnel qui est l'utilisation normale pour un langage de programmation, l'exécution des routines reste à la charge de l'exécutif et nous prenons donc une copie de l'objet persistant en mémoire volatile. On pourra étudier une solution pour remplir, au lancement de l'application, la table des routines gérées par l'exécutif à partir des informations sur les types qui se trouvent en mémoire persistante; cela permettrait de gérer la cohérence des routines en cas d'évolution des composants.

Pour l'accès associatif (requêtes sélectives), nous proposons deux solutions:

• l'exécution des routines d'un critère de sélection en mémoire volatile sous le contrôle de l'exécutif, mais en réorganisant le critère afin de retarder l'évaluation des routines et donc de les appliquer sur un plus petit ensemble d'objets,

• la traduction des routines Eiffel en O2 afin que celles ci puissent être appliquées sur les objets persistants sous le contrôle du gestionnaire d'objets, lorsqu'elles se trouvent dans un critère de sélection.

Evidemment la deuxième solution permet d'envisager de meilleures performances, mais est beaucoup plus complexe à mettre en oeuvre; elle nécessite par ailleurs une compatibilité très forte entre les deux modèles d'exécution.

# 9

# Principes généraux de la mise en oeuvre

Nous avons évoqué dans le chapitre 8 les principaux aspects à prendre en compte pour la mise en oeuvre de l'intégration; celle-ci comprend plusieurs parties:

- l'implémentation des classes visibles par l'application (classes décrites dans la partie II),

- l'implémentation du POM suivant deux approches (statique et dynamique):
  - l'implémentation de la traduction des classes Eiffel vers $O_2$,
  - l'implémentation de la conversion des objets de Eiffel vers $O_2$ et vice-versa
  - le traitement des requêtes sélectives;

- l'adaptation de l'exécutif Eiffel pour gérer les échanges entre l'application et le POM.

L'objectif de ce chapitre est de présenter les principes généraux de la mise en oeuvre. En particulier nous avons étudié plusieurs architectures possibles pour le système Eiffel intégrant un POM, le paragraphe 9.1 a la charge de les présenter. D'autre part, la propagation des types, qui est un facteur essentiel de l'intégration, est discuté dans le paragraphe 9.2, Enfin les aspects de paramétrisation de la persistance sont abordès dans le paragraphe 9.3.

## 9.1. Architecture de l'intégration

Nous avons étudié trois types différents d'architecture:

- une solution basée sur une interface générale vers le SGBD, destinée à implémenter une approche dynamique où la propagation des classes et le parcours des objets sont effectués pendant l'exécution,

• une autre solution dynamique, n'utilisant pas une interface générale, mais une bibliothèque de primitives de bas-niveau écrites en C,

• une solution statique fondée sur une traduction (et propagation), avant exécution des classes Eiffel vers le SGBD, utilisant pour la connection au serveur une bibliothèque de primitives C.

### Approche dynamique orientée composant

Dans cette première solution, nous proposons d'intégrer la persistance au dessus d'un gestionnaire d'objets, en deux niveaux (fig. 9.1):

• une *partie avant* qui contient les fonctionnalités nécessaires à un langage de programmation qui intègre l'aspect persistant,

• une *partie arrière* qui décrit une interface vers les SGBD.

Un des objectifs de cette approche est de rendre les parties *avant* et *arrière* aussi générales que possible du langage et du SGBD concernés; elles dépendront cependant des fonctionnalités offertes par le modèle du langage, de l'exécutif, qui lui est associé, et de l'interface vers les gestionnaires d'objets.

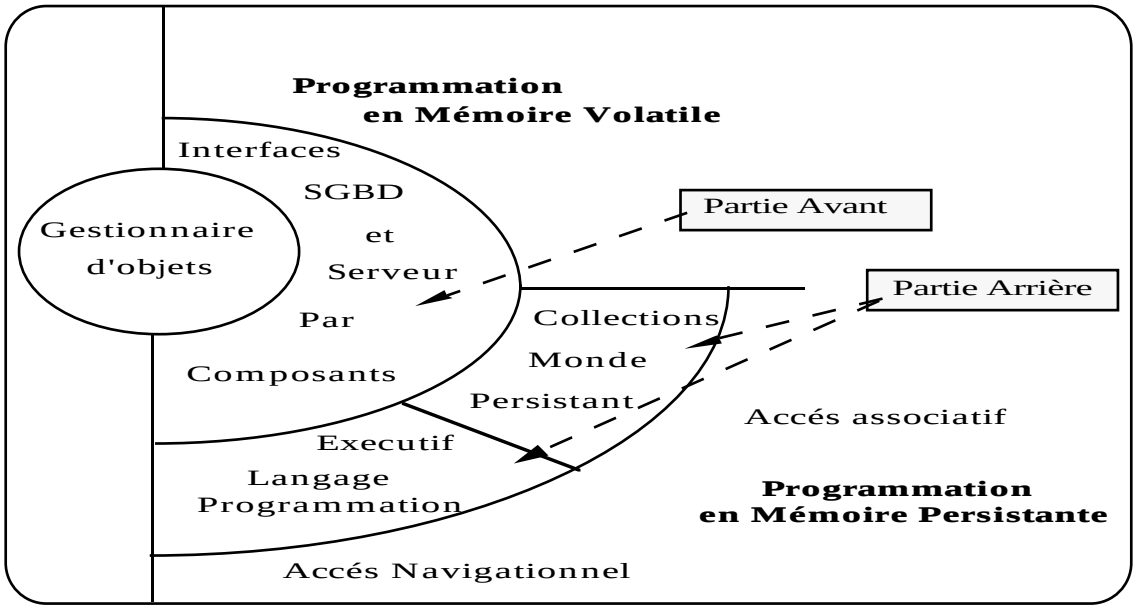


Figure 9.1: Architecture orienté composant du POM

La *partie arrière* peut-être, soit utilisée directement par l'application dans une optique d'interfaçage avec un système de base de données existant, soit servir à l'implémentation de la *partie avant*. Dans le cas d'un interfaçage, la manipulation des objets se fait en mémoire volatile (programmation volatile), et c'est à l'application d'établir la correspondance entre les entités gérées par l'exécutif et les entités gérées par le gestionnaire.

La *partie avant* intégre deux types de primitives:

• directement accessibles par des composants (*collection*, *monde persistant*, autres instances de classe), à partir d'appels explicites de l'application qui permettent principalement de rendre persistant des objets ou d'effectuer des sélections,

• gérées par l'exécutif du langage (et donc implicitement utilisées par les applications), permettant la gestion automatique de l'accès navigationnel aux objet persistants:

- gestion du partage des objets au sein d'une application,
- changement d'état d'un objet (volatile<->persistant),
- chargement, mise à jour et libération des objets,
- préservation de l'intégrité des données (panne ou accès concurrent),
- communication à l'exécutif du langage.

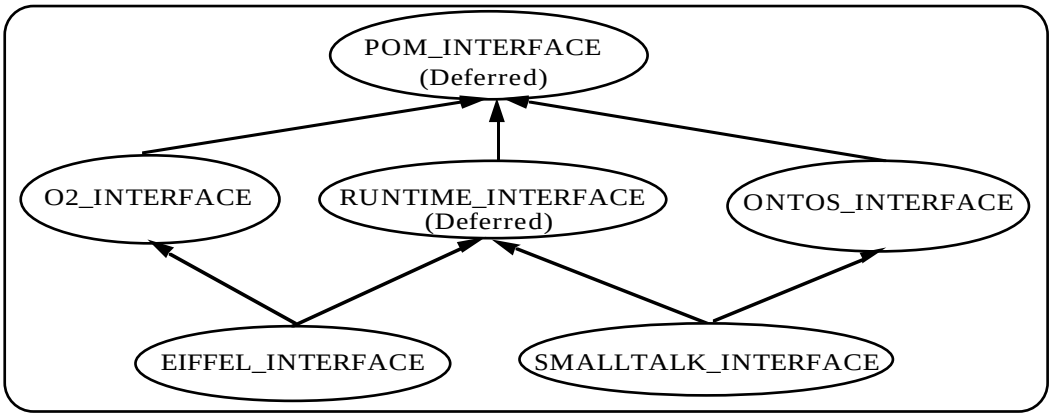


Figure 9.2: Généralisation de l'interface d'intégration de la persistance

Ces fonctionnalités pourront s'intégrer suivant la hiérarchie décrite dans la figure 9.2. Ainsi, l'interface minimale fournie par le POM correspondant à la classe retardée *POM_INTERFACE* (cf. § 10.1), est utilisée pour implémenter des routines décrivant les adaptations de l'exécutif (classe *RUNTIME_INTERFACE*); ces routines sont

décrites en détail dans le chapitre 10. La hiérarchie de la figure 9.2 représente deux exemples de spécialisation des classes *POM_INTERFACE* et *RUNTIME_INTERFACE*:

- l'exécutif Eiffel (*EIFFEL_INTERFACE)* et le SGBD $O_2$ (*O2_INTERFACE*) et
- celui de Smalltalk (*SMALLTALK_INTERFACE*) et Ontos (*ONTOS_INTERFACE*).

### Autre approche dynamique

Cette approche reprend les grandes lignes de l'approche dynamique orientée composants mais n'utilise pas une interface générale pour implémenter la partie avant du POM. L'avantage et de diminuer la taille des exécutables et le nombre de classes dans le système, en effet, une solution orientée composant conduit à des systèmes d'au minimum 130 classes.

La partie avant contiendrait en fait les primitives minimales pour implémenter les classes visibles de la partie arrière (voir 2ème partie), l'interface minimale nécessaire à l'exécutif Eiffel (cf. § 10.1), et la propagation des classes (voir chapitre 12).

### Approche statique

Cette approche fait aussi varier la taille d'une application Eiffel par rapport à la solution précédente. En effet les routines nécessaires à la mise en oeuvre de la propagation des classes n'a plus besoin de se trouver dans l'exécutable de l'application; par contre on associe à chaque classe du système, **deux routines supplémentaires permettant de convertir ses instances persistantes d'Eiffel vers le POM et vice-versa**.

## 9.2. Propagation des classes dans le monde persistant

Nous avons étudié deux approches possibles:

- une approche au plus tôt (approche statique),
- une approche au plus tard (approche dynamique).

## Approche au plus tôt

La persistance d'une classe (au sens de notre modèle), est détectée au plus tôt (dès que l'environnement Eiffel peut la connaître), c'est à dire au moment de sa première traduction. Le traducteur garantit que toutes les classes qui fait partie d'un système Eiffel ont une image dans le monde persistant. Dans cette solution, il semble raisonnable que les classes soient complètement propagées dans le monde persistant (attributs et routines). On étudiera dans le chapitre 12 les problèmes rencontrés pour faire correspondre une classe Eiffel à des classes $O_2$.

Cette approche introduit une forte redondance entre les informations gérées par l'exécutif et celles mémorisées par le gestionnaire d'objets, mais elle permet

- la réalisation de traitements en mémoire persistante, surtout pour les requêtes de sélection; cet aspect est développé dans le paragraphe 11.6,

- l'accès efficace pour toute application Eiffel aux informations associées à un composant (instance de la classe *EIFFEL_CLASS*),

- d'offrir une plateforme pour gérer le cycle de vie des composants et des objets,

- et surtout une **meilleure efficacité**, pour l'exécution d'un système de classe.

**Remarque**

La propagation peut se faire plus tard, mais de toute manière avant le début de l'exécution. Il faut que ce soit le compilateur qui effectue lui même ce travail, afin d'avoir la garantie que cette propagation est effectivement faite; on devrait ajouter pour ça une option dans le fichier de description du système (.eiffel).

## Approche au plus tard

Le type d'un objet est propagé dans le monde persistant au plus tard (seulement quand on en a besoin). Seuls les objets persistants voient leurs classes propagées.

Cette approche a l'avantage de rendre la propagation des types complètement transparente au compilateur. Il en va de même pour le développeur qui n'a pas à se préoccupper des classes qui doivent être propagées dans le monde persistant.

Les inconvénients sont cependant nombreux:

• la propagation du type d'un objet persistant au moment de l'exécution rend peu raisonnable l'idée de propager les routines dans le monde persistant et ne permet donc pas de respecter le critère n°5 (cf. § 3.1),

• le manque d'orthogonalité: une application n'a accès aux informations d'une classe que si celle-ci a des instances persistantes,

• le besoin de recourir à un mécanisme ad hoc et moins performant pour des requêtes sélectives qui ont des critères de sélection qui font référence à des routines (voir chapitre 11),

• l'absence d'une plateforme pour la gestion du cycle de vie.

**Recommandations**

Aucune de ces deux approches ne nous semble totalement convaincantes: intuitivement une classe devrait devenir persistante dès sa création.

Nous recommandons la conception d'un environnement intégré où éditeurs, traducteurs et applications partageraient une vue du monde persistant contenant toutes les instances de la classe *EIFFEL_CLASS*, lesquels pourraient, vu la diversité des utilisations, être fractionnée en plusieurs bibliothèques spécialisés. Comme nous l'avons déjà évoqué dans le paragraphe 8.5, étant donné que le compilateur Eiffel 3.1 est lui-même écrit en Eiffel, cela devrait faciliter telle implémentation. Un autre avantage serait de pouvoir intégrer des outils supplémentaires pour augmenter les possibilités de réutilisation (voir chapitre 13).

Le point délicat à résoudre serait le choix du gestionnaire d'objets; de notre point de vue, une solution à base de serveur d'objets (Kala, Wiss, ...) est celle qui conduirait à la meilleure intégration, avec le minimum de redondance et de limitations. Mais elle serait très couteuse à produire comme en témoigne les volumes d'homme/année qui ont été nécessaires pour réaliser des SGBD comme $O_2$ ou *Ontos*.

# 9.3. Paramétrisation de la persistance

La paramétrisation concerne les choix du niveau de persistance, du gestionnaire d'objets et de la mise en oeuvre de la gestion de l'extension d'une classe.

## 9.3.1. Choix du niveau de persistance et du gestionnaire

L'intégration de la persistance que nous proposons s'accorde avec le modèle *commit-lock* [Cardelli 85] (cf. § 2.2.2). On va montrer dans les chapitres 10, 11 et 12, les problèmes qu'il faut aborder dans la mise en oeuvre, et les différentes solutions, mais il va sans dire que certaines des solutions que nous proposons, si elles sont intéressantes du point de vue de l'environnement de programmation offert, sont néammoins coûteuses; il s'agît:

- du mécanisme de conversion des objets (représentation en mémoire volatile et en mémoire persistante),
- de la propagation du type dans le monde persistant (dans l'approche dynamique),
- et de la gestion de l'extension des classes.

Il faut donc proposer au développeur les trois niveaux de persistance [Cardelli 85]:

1) adaptation de la classe *ENVIRONMENT* actuelle de l'environnement Eiffel 3.1, pour des applications peu complexes qui manipulent un petit nombre d'objets persistants. Cette adaptation devrait réaliser:

- le chargement et la mise à jour des objets de manière implicite, au fur et à mesure de la navigation à travers un graphe d'objets persistants,
- la sélection des objets se fait manuellement par la clé ou par des sélections réalisées en mémoire volatile. Il n'y a pas de mécanisme de sélection en mémoire persistante (pas de propagation des types en mémoire persistante),

mais sans:

- la gestion de l'extension des types,
- le mécanisme de transaction,
- la gestion des objets obsolètes;

2) une implémentation de la persistance dans un environnement séquentiel, mais où le volume des données manipulées peut être important; les principaux services offerts sont:

- le chargement à la demande des objets gérés par l'exécutif,
- la gestion implicite des transactions,
- une propagation dynamique des types (pas de propagation des routines),
- un mécanisme de requête sélective, s'exécutant en partie en mémoire volatile, selon le contenu des critères de sélection (routines),
- la gestion de l'extension des classes (peut être dèsactivée),

3) une implémentation complète adaptée à la gestion d'un grand volume de données partagées par différentes applications; elle comprend en plus de la version précédente:

- une propagation statique des classes,
- la gestion implicite et explicite des transactions,
- le traitement en mémoire persistante des requêtes sélectives,
- la gestion de l'extension des classes (peut être dèsactivée).

## Choix du niveau de persistance

On doit donc permettre au développeur de choisir son niveau de traitement de la persistance lors de la compilation du système de classes; ceci pourra être mis en oeuvre dans le fichier d'information décrivant le système (*system description file*) (Eiffel V. 2.3). Une intégration dans le langage *LACE* (Eiffel V.3 ) pourra être aussi envisagée

```
.......
low-range persistance (Y / N)          -- this is the default
middle-range persistance (Y / N)
full persistence (Y / N):
.......
```

Cela permettra de diminuer la taille des exécutables et d'accélerer les performances de l'application quand c'est possible. En particulier, on sélectionnera la version de la classe *ANY* la plus adaptée au niveau voulu de persistance. Cette possibilité est plus importante encore, si le couplage avec le gestionnaire d'objets persistants (POM) est réalisé à partir de l'architecture décrite au paragraphe 9.1, lequel fait appel à une interface générale (augmentation prohibitive du nombre de classes de l'application).

Remarque: le degré de persistance peut aussi être détecté par le compilateur en fonction de l'utilisation, mais il vaut mieux que le programmeur le connaisse.

### Choix du gestionnaire d'objets

Il faut aussi permettre le choix du gestionnaire d'objet utilisé pour gérer la persistance, afin de rendre l'environnement aussi ouvert que possible au monde extérieur; ce choix devra, comme pour le niveau de persistance, être fait au moment de la traduction du système.

```
.......
Persistent Object Manager:        -- à choisir parmi:
                                  -- O2, Wiss, Kala, ...
.......
```

Remarque: Pour le choix du gestionnaire aussi, on pourra déterminer le choix du gestionnaire en fonction d'un lien d'héritage.

## 9.3.2. Gestion des *Pcollections*

La gestion des *Pcollections* représente un surcoût pour l'application, puisque chaque objet doit être inséré dans la *Pcollection* qui lui est associé, et qu'il faut gérer ces Pcollections par rapport à l'héritage des composants. Il faut donc envisager le cas où le développeur ne veut pas gérer l'extension des classes; cependant ce choix ne peut être propre à une application (surtout pour une extension de niveau 3), mais doit se faire au niveau d'une vue (pview), et sera partagé par toute application accèdant à cette vue (voir fig. 9.7).

```
class PVIEW
.....
feature     .......
  is_pcollection_handling: BOOLEAN is
        -- Is automatic pcollection handling, set?

  set_pcollection_handling is
        -- Set automatic handling of pcollection (should build pcollection
        -- from already persistent objects)

  unset_pcollection_handling is
        -- Unset automatic handling of pcollection
end -- class PVIEW
```

Figure 9.7: Activation / Dèsactivation de la gestion des Pcollections

Remarque:

Une gestion de l'activation au niveau d'une application aurait pour effet de fausser complètement la gestion des *Pcollections,* pour toutes les applications concernées par la même *pview*, et de faire perdre ainsi toute valeur à la vérification d'assertions adressant les *Pcollections*.

On notera que la possibilité d'activer et dèsactiver la gestion des pcollections doit s'accompagner de l'ajout d'assertion dans les propriétés de la classe ANY qui concernent les pcollections (fig. 9.8).

```
class ANY
feature
        ........
        pcollection: PCOLLECTION [Like Current] is
                -- returns the Pcollection corresponding to the class of the object
            require    is_pcollection_handling

        class_pcollection (s: STRING): PCOLLECTION [ANY] is
                -- return the pcollection corresponding to class 's'
            require    is_pcollection_handling
        ..........
invariant
        ..........
end -- Class ANY
```

Figure 9.8: Assertions et Activation / Dèsactivation de la gestion des Pcollections

# 10

# Propositions pour l'adaptation de l'exécutif

Nous avons expliqué (cf. § 2.2.3), qu'un des pré-requis pour une bonne intégration de la persistance est de laisser à l'exécutif gérer le va-et-vient des objets entre les mémoires persistante et volatile. Dans ce paragraphe nous étudions les différentes adaptations de l'exécutif pour lui faire remplir ces fonctionnalités. Nous rappellerons d'abord les principaux aspects d'une première approche que nous avons abandonné, et développerons ensuite celle qui est maintenant utilisée.

**Préambule**

Dans [Lahire & al. 91], nous avons proposé une gestion basée sur une mémoire tampon assurant la gestion du va-et-vient des objets persistants; cette technique crée un certain nombre de problèmes provenant de la possibilité qu'un objet persistant a d'être arbitrairement rejeté en mémoire persistante en cours d'exécution. En particulier, un objet peut être rejeté en mémoire persistante si le tampon est plein et que le chargement d'autres objets est demandé par l'application. Les problèmes soulevés par cette approche portent sur:

- la redondance avec la gestion virtuelle de la mémoire, effectuée par le système d'exploitation,

- la date de bloquage et débloquage d'un objet en mémoire (il faut détecter les états de bloquage),

- le choix de la taille du tampon (optimisation du rapport entre le nombre de va-et-vient et l'espace consommé),

- la manque d'intégration avec la philosophie choisie par l'exécutif Eiffel.

Par rapport à cette approche, la nouvelle est plus simple, plus efficace (pas d'indirection lorsque l'objet est en mémoire, pas de copie de l'objet lorsqu'il devient persistant), et mieux intégrée à l'exécutif: la libération est entièrement gérée par le ramasse-miettes.

## 10.1. Interface minimale avec le POM

Une description complète du POM est réalisée dans le chapitre 12, mais nous présentons ici l'interface minimale que doit fournir le POM à l'exécutif pour lui permettre de gérer les échanges entre les mémoires persistante et volatile. Les primitives de cette interface sont regroupées dans la classe *POM_INTERFACE* (fig. 10.1).

```
deferred class POM_INTERFACE
feature     ......
  exist (object: POI): BOOLEAN is
        -- Is object exist in persistent memory
    deferred

  create_object (type: STRING) is
        -- Create a persistent instance of class type
    deferred

  update_attribute (object: POI, attribute: STRING, value: POM_OBJECT) is
        -- Update attribute of object with value
    deferred

  get_attribute (object: POI, attribute: STRING): POM_OBJECT is
        -- Return the contents of attribute of object
    deferred

  update_object (object: POI, value: POM_OBJECT) is      -- Update object with value
    deferred

  get_object (object: POI): POM_OBJECT is                -- Return the contents of object
    deferred

  begin_transaction is          -- ask the POM to start a new transaction
    deferred

  commit_transaction is         -- ask the POM to validate and finish current transaction
    deferred

  abort_transaction is          -- ask the POM to cancel current transaction
    deferred
end -- class POM_INTERFACE
```

Figure 10.1: Interface minimale pour implanter les services gérés par l'exécutif

La classe *POI* représente le concept d'identificateur d'objet persistant, et la classe *POM_OBJECT*, le contenu d'un objet, dans le modèle du gestionnaire (donc déjà converti).

Par ailleurs, certaines primitives de la classe *POM_INTERFACE* comme par exemple, *begin_transaction*, *validate_transaction* ou *create_object*, peuvent être accessibles à travers d'autres classes, visibles par l'application.

## 10.2. Introduction d'une table des instances persistantes

On a vu dans le paragraphe 8.3 qu'en introduisant des échanges incrémentaux entre le monde persistant et une application, on pouvait susciter la présence en mémoire de plusieurs exemplaires du même objet.

```
deferred class RUNTIME_INTERFACE
inherit
      EXCEPTION;   -- raise, last_exception, etc.  language exception mechanism
      ERRORS;      -- is_error, last_error, etc.  error handling (exchanges with the POM)
      POM_INTERFACE   -- Access to the POM
creation   create
feature

  create is
     -- run-time initialization
     do
        !!p_instances.create;
        !!p_modified_instances.create
     end; -- create

  p_instances: P_TABLE is
        -- table handling persistent object descriptor in volatil memory
     deferred

  p_modified_instances: LINKED_LIST  [PODVM] is
        -- list of podvm corresponding to modified persistent-objects

  is_podvm (object: ANY): BOOLEAN is
        -- is object apersistent object descriptor in volatil memory?

end -- RUNTIME_INTERFACE
```

Figure 10.2: Outils pour la gestion du partage des objets persistants par l'exécutif

Pour éviter ce problème, nous proposons une solution proche de celle de *Loom*, en mettant en oeuvre un mécanisme basé sur une table qui mémorise les objets persistants présents en mémoire volatile, ce qui permettra d'assurer leur unicité.

Ce mécanisme pourra aussi être utilisé pour gérer la mise à jour des objets (fig. 10.2), ainsi que leur libération; il est intégré dans la classe *RUNTIME_INTERFACE*.

### Description des fonctionnalités de la *p_table*

Le type *P_TABLE* est une structure destinée à mémoriser les descripteurs d'objets persistants en mémoire volatile (notés *podvm*). Un *podvm* permet de mémoriser pour chaque objet persistant, son identificateur d'objet persistant (poi), un indicateur de modification, son adresse en mémoire volatile et un indicateur de libération.

La classe *P_TABLE* (fig. 10.3), doit permettre à la fois un accès par le *poi* et par l'adresse des objets (voir description des primitives de la classe *INTEGRATION_INTERFACE*).

```
class P_TABLE
feature
  put (e: PODVM) is
        -- add podvm e in table
     require   not has (e)

  remove (e: PODVM) is
        -- remove podvm e from table
     ensure    not has (e)

  has (e: PODVM): BOOLEAN is
        -- Is e in table?

  has_iop (e: IOP): BOOLEAN is
        -- Is e associated to a podvm ?

  has_object (e: ANY): BOOLEAN is
        -- Is e associated to a podvm ?
     require   e.is_persistent

  item_iop (e: IOP): PODVM is
        -- podvm Associated to e
     require   has_iop (e)

  item_object (e: ANY): PODVM is
        -- retourne le podvm correspondant à l'objet e
     require   has_object (e)
end  -- class P_TABLE
```

Figure 10.3: Table des objets persistants en mémoire volatile

Ainsi, on doit en particulier disposer des fonctionnalités suivantes:

- ajout, suppression d'un *podvm*,
- test de la présence d'un podvm particulier en fonction de l'identificateur d'objet

persistant, ou de l'adresse en mémoire volatile,

• accès à un *podvm* particulier en fonction de l'identificateur d'objet persistant, ou de l'adresse en mémoire volatile.

La vue abstraite d'un descripteur d'objet persistant (*podvm*) peut être décrit par la classe *PODVM* (fig. 10.4).

```
deferred class PODVM                    -- Persistent object descriptor in volatil mémory
feature
  poi: POI is
        -- Persistent object identifier
      deferred

  voi: ANY is
        -- Address of object in volatil memory if it is loaded
      deferred

  set_voi (object: ANY) is
        -- Make object, the new value of voi
      deferred
      ensure    voi = object and is_loaded

  set_poi (object: POI) is
        -- Make object, the new value of poi
      deferred
      ensure    poi = object

  is_loaded: BOOLEAN is
        -- is object loaded
      deferred
      ensure    Result implies voi /= void; not Result implies voi = void
invariant
  is_modified implies not voi = void
end -- class PODVM
```

Figure 10.5: Descripteur d'objet persistant en mémoire volatile (PODVM)

## Description de l'implémentation

Il est important de choisir la structure d'implémentation de la table et l'ordonnancement des descripteurs d'objets, avec l'efficacité, comme principal critère. C'est pourquoi nous avons attaché un soin particulier aux choix d'implémentation.

La *p_table* est en fait un ensemble de deux hash-tables permettant un accès rapide aux objets en fonction de l'identificateur d'objet persistant (*poi*) et de l'adresse en mémoire volatile (*voi*). Chacune des tables de hash-code sont implémenté par le mécansime montré dans la figure 10.6.

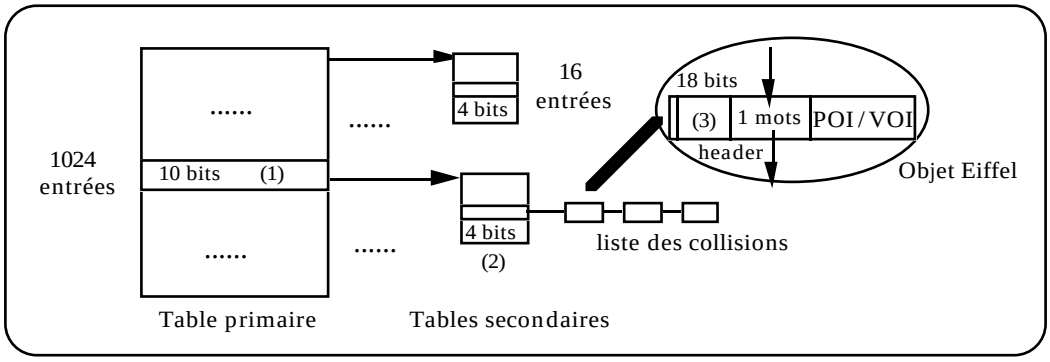


Figure 10.6: Description du mécanisme de la p_table

Le hash-code est construit à partir du voi ou du poi, selon la hash-table considérée; chacun d'eux a une longueur de 32 bits, découpée en trois morceaux ((1), (2), (3), sur la figure). Le premier (1) représente la clé d'accès dans la table primaire qui a une taille fixe et est allouée à l'initialisation; le second (2) est la clé d'accès dans la table secondaire, qui n'est allouée que si nécessaire; le troisième (3,) qui se trouve dans l'entête d'un objet Eiffel, représente la dernière partie de l'adresse.

L'objet Eiffel contient:

- dans l'entête:
  - des informations nécessaires à la gestion du mécanisme: marquage du chargement, de la modification de l'objet, indication que l'objet est volatile, persistant ou représente un podvm,

  - un chaînage dans la liste des collisions;

- comme contenu:
  - pour l'objet dans la table des *poi*: la valeur du *poi* ou du *voi* selon que l'objet est chargé ou pas en mémoire volatile,
  - pour l'objet dans la table des *voi*: la valeur du voi si l'objet est chargé (table des *poi*), sinon aucune entrée n'est présente dans la table

Le choix de mémoriser ces informations dans un objet Eiffel permet l'intégration du podvm dans le run-time: lors d'un accès à un objet le run-time rencontre, soit un podvm, soit un objet Eiffel (voir sections 10.3 et 10.4).

La formule de hashage est donc basée sur un découpage de l'adresse de l'objet; les choix de découpage ont été réalisés après une étude de la politique d'allocation de la mémoire, mais peuvent être aisément modifiés si d'autres statistiques sont obtenues (par exemple sur des machines différentes). Ce mécanisme de gestion a un certain nombre d'avantages, comme le montrent les chiffres du tableau suivant:

- la taille de la table de hash-code peut varier dynamiquement,

- le mode de répartition s'adapte à de gros volumes d'objets manipulés:
  - efficacité: un nombre d'indirections réduit pour accéder un objet,
  - coût en espace mémoire est raisonnable.

| **Nb Objets** | **Nb mots / Nb objets** | **Nb indirections** | **Nb recherches** | **Temps (sec)** |
|---|---|---|---|---|
| 10 | 518 | 1.10 | 30 | 0.00 |
| 100 | 52 | 1.51 | 249 | 0.01 |
| 1000 | 5.4 | 2.49 | 3 765 | 0.03 |
| 10 000 | 1.9 | 2.79 | 41 829 | 0.36 |
| 100 000 | 1.65 | 3.55 | 532 739 | 3.63 |
| 200 000 | 1.57 | 6.50 | 1 833 950 | 8.18 |
| 500 000 | 1.54 | 9.72 | 4 357 766 | 23.7 |

Les résultats présentés dans ce tableau montre que:

- le surcoût en matière d'espace considérable pour un faible nombre d'objets, se réduit au fur et à mesure que le nombre d'objets croît et devient raisonnable dès que l'on atteint 1000 objets manipulés,

- le nombre d'indirections pour atteindre un objet est insignifiant jusqu'à 100 000 objets, et reste raisonnable même pour un nombre important d'objets (moins de 10 indirections pour 500 000 objets),

- le temps consommé dans les échanges est réduit jusqu'à 200 000 objets et reste raisonnable au delà. On notera que les temps mesurés correspondent au nombre de recherches (aléatoires) effectuées mentionnés sur la même ligne du tableau (prés de 2 millions de recherches prennent un peu plus de 8 secondes). La machine utilisée pour réaliser les tests est un Sun sparc 2 de 20 Mips.

**Remarque**

Pour pouvoir considérer complètement un *podvm* comme un objet Eiffel, il faudrait rajouter un mot afin de permettre le chaînage du *podvm* dans la liste des objets Eiffel, et permettre le ramasse-miettes d'agir sur lui.

# 10.3. Passage d'un objet au statut de persistant

Un objet peut devenir persistant, soit par une action explicite de l'application (utilisation d'une primitive *put* des classes *PERSISTENT_WORLD* ou *PCOLLECTION*), soit automatiquement lorsqu'un objet temporaire est rattaché à un objet déjà persistant.

## 10.3.1. Mécanisme de base

Par l'activation de la primitive *make_persistent* sur un objet (cf. § 12.3), l'exécutif transmet au POM, l'objet et tous ceux qu'il référence. Il est important de rendre persistant les objets référencés pour garantir l'intégrité des références; à défaut, un objet persistant perdrait les informations, contenues dans le(s) objet(s) volatile(s) dès la fin de l'exécution de l'application.

```
deferred class RUNTIME_INTERFACE
inherit    CONVERSIONS      -- class implementing conversions POM <-> RUNTIME
feature    ......
  make_persistent (object: ANY) is
        -- Make persistent object with all its dependents
    require   not is_error and then not object.is_persistent
    deferred
    ensure    not is_error implies   object.object_traversal.all ($post_condition1)
    end   -- make_persistent

  post_condition1 (object: ANY): BOOLEAN is
        -- selection criteria for the post-condition of make_persistent
    do
        Result:=    (              -- Copy of referenced object
                        object.is_persistent  and p_instances.has_object  (object)  and
                        p_instances.item_object  (object).voi = object and
                        p_instances.item_object   (object).is_modified
                    ) or else   -- reference of the podvm
                    (   is_podvm  (object) and p_instances.has  (object) )
    end   -- post_condition1
end -- class RUNTIME_INTERFACE
```

Figure 10.7: Rendre un objet persistant

Le rôle de la routine *make_persistent* (fig. 10.7) est de mettre à jour la table des objets

persistants pour chaque objet rendu persistant, pour ces objets, le podvm contiendra le voi et le poi de l'objet et il indiquera que l'objet a été modifié. Ce *podvm* est inséré dans la table *p_instances* et dans la liste des objets modifiés (*p_modified_instances*). En outre cette routine garantit qu'un nouveau *podvm* n'est créé que si l'objet n'est pas déjà associé à un *podvm*.

La routine *make_persistent* utilise les primitives suivantes fournies par le POM (cf. § 10.1):

- *create_object* pour créer les instances persistantes et obtenir leur poi,

- *update_attribute* pour remplir les objets devenus persistants; si le modèle du gestionnaire et du langage ne sont pas identiques une opération de conversion devra être mise en oeuvre dans *make_persistent*.

On remarquera que dès que l'on rencontre un objet déjà persistant, il est inutile de chercher de nouveaux objets à rendre persistant; on rappelle que tout objet persistant ne référence que des objets persistants.

**Remarque:**

La classe CONVERSION permet d'implémenter les conversions de représentation entre le POM et l'exécutif du langage; La vue abstraite de cette classe est définie dans la figure 10.8. L'implémentation des routines de conversion dépendent à la fois du modèle supporté par le POM et de celui du langage, elles seront définies dans les classes EIFFEL_INTERFACE, etc.

```
deferred class CONVERSIONS
feature
  convert_to_pom (object: ANY): POM_OBJECT is
        -- convert object in a POM object
    require   not object = Void
    deferred

  convert_from_pom (object: POM_OBJECT): ANY is
        -- convert object into an object which may be handled by the programming language
    require   not object = Void
    deferred

end -- class CONVERSIONS
```

Figure 10.8: Conversion des objets (exécutif du langage <-> POM)

## 10.3.2. Passage d'un objet au statut de persistant par attachement

Dans les opérations de réattachement d'un objet à une entité, (opération d'affectation, passage de paramètres effectifs ou création d'objets), il faut considérer la possibilité qu'un objet a de devenir persistant et au cas échéant provoquer l'appel de la routine *make_persistent* (fig. 10.9).

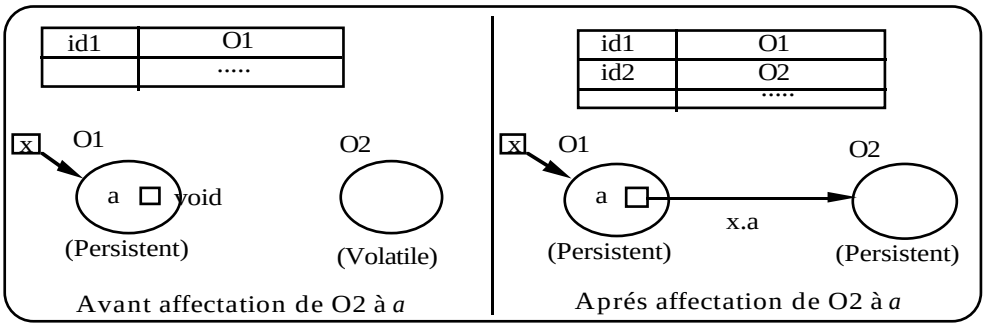


Figure 10.9: Propagation de la persistance lors d'une affectation

Soit un objet persistant *O1,* identifié par l'iop *id1*; on attache à un attribut *a* de *O1,* l'objet *O2* initialement volatile. L'exécutif assure que l'objet *O2* et les objets qu'il référence deviennent immédiatement persistants à leur tour; les mécanismes décrits précédemment sont alors combinés dans le schéma ci-dessus.

Dans le cas où le réattachement correspond à une affectation, on modifie l'opérateur *assign*; pour la création d'un objet (expansé ou non), référencé par un attribut de la classe, l'opérateur *creation_assign* sera aussi modifié (fig. 10.10). Le cas particulier de l'attachement par routines externes sera étudié dans le paragraphe 10.11.

Dans le cas d'un attachement à une entité locale (passage de paramètre, résultat de fonction, ou affectation à une variable locale), notre mécanisme ne prévoit pas de forcer les objets attachés à devenir persistants. Les raisons de ce choix tiennent essentiellement au fait qu'une entité locale est destinée à avoir une durée de vie limitée, et peut être utilisée fréquemment.

Compte tenu des choix et remarques faites ci-dessus, on donnera le principe d'intégrité référentielle:

**Principe d'intégrité référentielle**

*Un objet persistant ne référence par ses attributs que des objets persistants, mais les entités locales de ses routines peuvent référencer des objets temporaires.*

**Remarques**

Avec ce mécanisme, l'exécutif vérifie si à chaque affectation, l'objet est persistant ou non; il faudra donc que ce test soit très rapide (test sur un bit par exemple). On rappelle aussi que le coût de la conversion d'un objet, lorsqu'il est rendu persistant, peut être très important.

Si un objet volatile est rendu persistant par attachement à un objet et que cet attachement est supprimé avant la fin de la transaction, l'objet sera quand même propagé au POM; c'est son ramasse-miettes qui aura la charge de gérer la suppression de cet objet.

```
deferred class RUNTIME_INTERFACE
  .....
feature
  assign (object1, object2: ANY; attribute: ANY) is
            -- assign object2 to attribute of object1
      require   -- not is_error and then attribute is a field of object1
      do
            if object1.is_persistent and then is_volatil (object2) then
                make_persistent (object2);
                if not is_error then attribute:= object2
                else   raise (last_error)
                end; -- if
            else   attribute:= object2     -- volatil object
            end; -- if
      ensure     not is_error and object1.is_persistent implies
                           (   object1.object_traversal.all ($post_condition2) and
                               attribute.object_traversal.all ($post_condition1) )
      end; -- assign

  post_condition2 (object: ANY): BOOLEAN is
          -- selection of postcondition used in assign and creation_assign
      do  Result:= object.is_persistent or else is_podvm (object)  end      -- post_condition2

  creation_assign (object: ANY; attribute: ANY) is
            -- Create a new object and assign attribute to object
      require   not is_error and then      -- attribute is a field of object
      do     -- creation of of a new object, whose type conforms to  the one of attribute
            assign (object, new_object, attribute)
      ensure     not is_error and object1.is_persistent implies
                               ( object1.object_traversal.all ($post_condition2) and
                                 attribute.object_traversal.all ($post_condition1) )
      end; -- creation_assign

  is_volatil (object: ANY): BOOLEAN is
            -- Is object  volatil?
      ensure     Result = True implies not is_podvm (object) and not object.is_persistent
      end; -- is_volatil
end -- RUNTIME_INTERFACE
```

Figure 10.10: Description des primitives pour la propagation de la persistance

# 10.4. Chargement d'un objet persistant

Quand une application demande implicitement au gestionnaire d'objets le chargement d'un objet persistant, on peut envisager de:

- charger l'objet et tous ses dépendants,
- charger seulement l'objet et ses champs.

Le choix doit dépendre du contexte de l'application et du type des objets, et permettre l'optimisation de l'accès aux objets; on montrera dans le paragraphe 10.10 une réflexion à ce sujet.

Comme nous ne voulons pas revenir à la politique d'échange global de la classe *ENVIRONMENT*, de la version 3 d'Eiffel (*store* et *retrieve*), tous les objets accessibles ne sont pas obligatoirement présents en mémoire. Il faut donc introduire un mécanisme permettant à l'exécutif du langage de savoir si l'objet persistant est directement disponible ou non, afin qu'il puisse au besoin, demander son chargement en mémoire volatile.

Nous étudierons dans la prochaîne section les différentes situations pouvant provoquer le chargement d'un objet, mais voici les principes sur lesquels repose notre approche:

- la première fois que l'on atteint un objet persistant *id2*, par un attribut nommé *a*, c'est au *podvm* que l'on accède et non à l'objet lui-même (fig. 10.11),

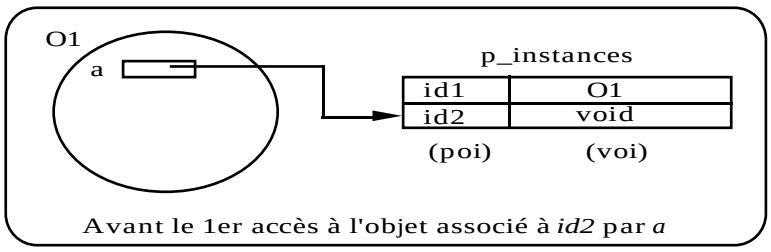


Figure 10.11: 1er accès à un objet persistant

- les fois suivantes, on accède à l'objet lui-même (à sa copie en mémoire volatile) (fig. 10.12),

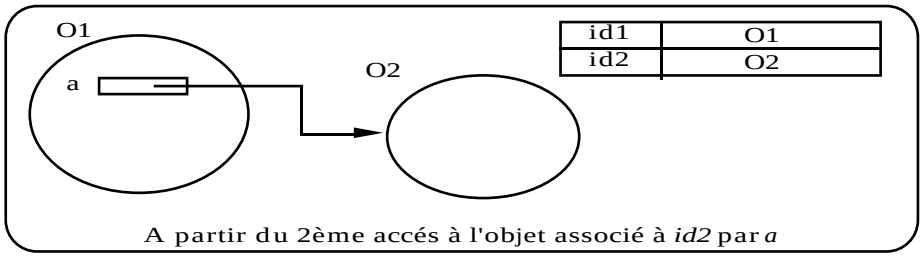


Figure 10.12: accès à un objet persistant sauf le 1er

• si l'attribut *a*, est modifié et référence maintenant un autre objet *id3*, alors lorsqu' on atteint *id3* par *a*, on accède soit au *podvm* de *id3*, soit à la copie de l'objet en mémoire volatile; cela dépend si la source de l'affectation référençait le *podvm* ou la copie volatile de *id3* (fig. 10.13).

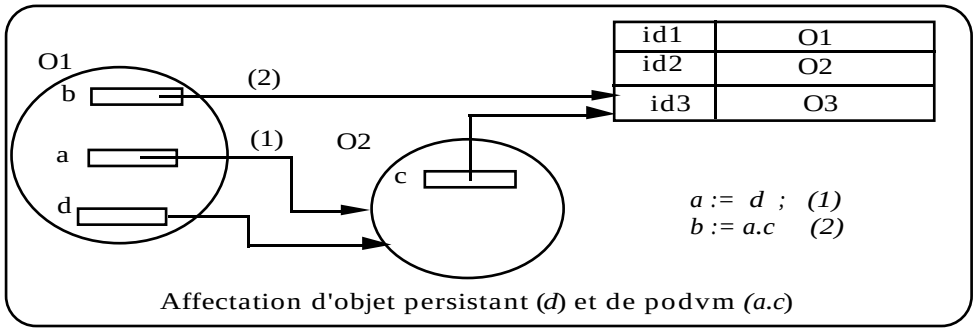


Figure 10.13: Accès à un objet persistant nouvellement attaché

A partir de ces 3 principes, on déduit les faits suivants:

• Ce mécanisme permet de mélanger les différentes politiques de chargement,

• les choix d'optimisation de l'opérateur d'affectation sont indépendants du mécanisme d'accès aux objets,

• le mécanisme d'accès aux objets prend en compte la fréquence d'utilisation d'un même chemin pour atteindre un objet (le passage par le *podvm* n'est fait que la première fois).

C'est donc à partir de la table *p_instances*, qui regroupe les différents descripteurs d'objets persistants (*podvm*), que nous allons gérer le chargement de ces objets.

Lorsqu'on charge un objet *O1*, alors pour chaque objet (non expansé) référencé directement par *O1*, un nouveau *podvm* est créé et inséré dans la table *p_instances* s'il

n'y figure pas déjà. On remarquera que l'existence du *podvm* ne veut pas forcément dire que l'objet est chargé en mémoire volatile: il ne sera effectivement chargé que si c'est nécessaire. Un podvm ne garantit que le moyen d'atteindre un objet persistant.

Le *podvm* contient l'identificateur persistant de l'objet (par exemple *id2*), qui permet d'établir directement la correspondance avec le *void* qui lors de la création du *podvm* est *void*. Un indicateur permet de déterminer si une copie de l'objet persistant existe en mémoire volatile; ceci est particulièrement utile dans l'hyppothèse où l'objet est partagé (différents chemins d'accès dans le graphe d'objets).

Quand l'objet *O2* (identificateur *id2*) est effectivement chargé, non seulement son adresse en mémoire volatile remplace la valeur *void* initialement présente dans le *podvm*, mais encore, l'attribut correspondant (ici *a*) ne référence plus le descripteur mais l'objet lui-même (fig. 10.14).

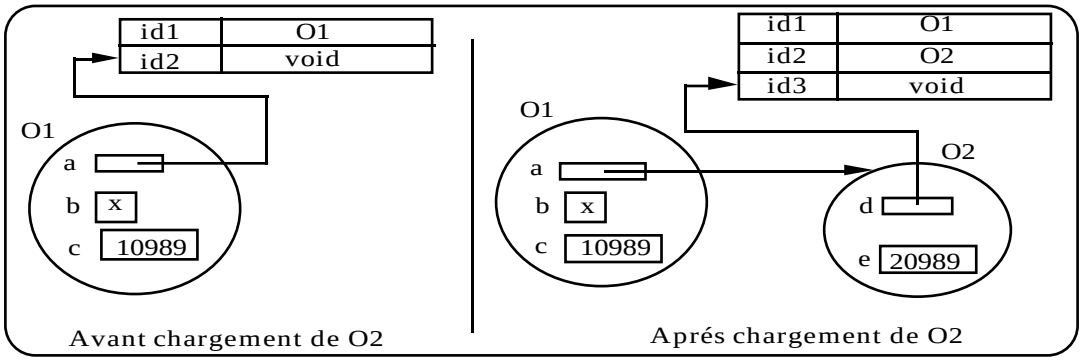


Figure 10.14: Mise à jour de la table des objets persistants

On en déduit le rôle de la routine *load_object* (fig. 10.15); elle permet le chargement de l'objet à partir de son poi, et garantit que chacun des objets référencés directement est associé à un *podvm*.

**Remarque**

On n'a pas choisi de mémoriser le podvm des objets non chargés dans les attributs qui le référencent, et ainsi de limiter la table *p_instances* à contenir les *podvm* des objets chargés. Pour tous les objets, il faudrait alors agrandir leur taille afin de pouvoir contenir tantôt un identificateur d'objet persistant (la taille varie selon le POM), tantôt une adresse en mémoire volatile, et fournir un mécanisme pour les différencier.

Dans notre solution, la perte de place est plus importante pour les objets persistants, mais elle n'affecte pas les objets volatiles, et rend la taille des objets indépendantes du POM. Une indirection supplémentaire est cependant le prix à payer pour récupérer l'adresse de l'objet en mémoire volatile.

```
deferred class RUNTIME_INTERFACE
  .....
feature
  load_object (object: POI): ANY is
            -- load object associated to object
          require   not is_error
          deferred
          ensure    not is_error implies
                                  p_instances.has_iop (object) and then
                                  p_instances.item_iop (object).voi = Result and then
                                  Result.object_traversal.all ($post_condition3)
          end; -- load_object

  post_condition3 (object: ANY): BOOLEAN is
          -- selection criteria of post-condition of load_object
      do Result:= is_podvm (object) and then p_instances.has (object) end -- post_condition3
end -- RUNTIME_INTERFACE
```

Figure 10.15: Primitives pour le chargement des objets persistants

## Cas d'erreurs

Lors du chargement d'un objet plusieurs erreurs peuvent se produire:

- l'objet n'est pas accessible en lecture,
- l'objet est en cours de modification par une autre application,
- aucun monde persistant n'est accessible,
- une erreur interne s'est produite au niveau du gestionnaire,
- pour un accès explicite de l'application: l'objet n'existe pas,
- l'objet est obsolète,

Il faut propager ces erreurs aux composants Eiffel ajoutés à un système Eiffel pour traiter la persistance, afin de lui permettre de réagir. On utilisera la classe *ERRORS*, et pour chacune des erreurs citées ci-dessus, on retournera les codes d'erreurs suivants qui seront traités dans les composants et provoqueront des exceptions:

- *read_permission_denied*,
- *write_conflict*,
- *pom_internal_error*,
- *object_does_not_exist*,
- *object_is_obsolete*.

# 10.5. Accès à un objet persistant

L'accès à un objet peut être explicite, et dans ce cas il se confond avec le chargement de l'objet (primitives des classes *PERSISTENT_WORLD* ou *COLLECTION*); dans la suite nous traiterons plus particulièrement le cas d'un accès réalisé implicitement par l'exécutif.

Pour automatiser le chargement des objets persistants, lors d'un accès navigationnel, il faut que l'exécutif constate un défaut d'objet, et au besoin, demande cet objet au POM.

En Eiffel, l'accès à un nouvel objet se fait par une notation pointée; ainsi pour accéder à une primitive *b* d'un objet *O2*, à partir de l'attribut *a* d'un objet *O1*, on écrira: *a.b*; (on suppose que l'objet courant est *O1)* (fig. 10.16).

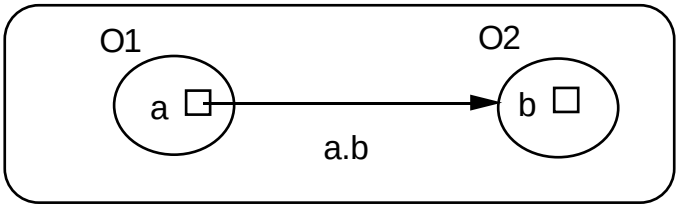


Figure 10.16: Accès à un objet par une expression pointée

Il faut donc enrichir la sémantique de l'opérateur '.', afin qu'elle prenne en compte les remarques ci-dessus. Lors de l'accès à *O2*, avec une notation pointée l'exécutif étudiera la nature de l'objet référencé par *a*, est-ce un:

- objet volatile ? (is_volatil (a)),
- objet persistant ? (a.is_persistent),
- podvm ? (is_podvm (a)).

note: *is_persistent* est une primitive du type ANY, car la persistance d'un objet fait partie de ses propriétés, alors que *is_podvm* et *is_volatil* font seulement partie de la mise en oeuvre, et à ce titre s'intègre dans la classe *RUNTIME_INTERFACE*.

Lorsqu'on cherche à accéder à *b*, à partir de *a*, l'exécutif se pose la question de savoir si l'objet *b* est un *podvm* ou pas (fig. 10.17).

Si ça en n'est pas un (ce n'est pas la première fois que *b* est atteint par *a*, ou *b* est un objet volatile), alors tout se passe comme si la persistance n'était pas intégrée. Dans le cas contraire, cela veut dire soit que l'objet n'est pas chargé (il faut alors demander son chargement au POM), soit qu'il l'est, mais sans n'avoir jamais été accèdé par *a* : il suffit

alors de le charger en utilisant la table des objets persistants (*p_instances*); dans les deux cas l'objet *b* sera ensuite attaché à l'attribut *a* (section10.2).

```
deferred class RUNTIME_INTERFACE
  .....
feature
  dot (object: ANY, attribute: ANY): ANY is
            -- Return object pointed by attribute of object
      require   not is_error
      do
            if is_podvm  (attribute) then
                if attribute.item = Void then
                      attribute:= load_object  (attribute.poi)
                else
                      attribute:= attribute.item
                end -- if
                if is_error then
                   raise  (last_error)
                else
                   Result:= access  (attribute)
                end -- if
            end; -- if
      ensure
            not is_error implies (
                      not is_podvm  (attribute)  and
                      (old attribute.is_persistent or else is_podvm  (old  attribute) ) implies (
                            old attribute.poi = p_instances.has_object (attribute).poi and
                            p_instances.item_object  (attribute).voi = Result and
                            attribute.object_traversal.all  ($post_condition3) ) and
                            object.object_traversal.all ($post_condition4)
      end; -- dot

  post_condition4 (object: ANY): BOOLEAN is
          -- selection criteria of dot postcondition
      do
            Result:=      (   is_podvm  (object)  and then p_instances.has  (object))
                          or
                          (   object.is_persistent and then p_instances.has_object  (object)
                              and then p_instances.item_object  (object).voi = object )
      end   -- post_condition4

  access (a: ANY): ANY is
            -- feature of Eiffel Run-time
      external "C"
      end -- access
end -- RUNTIME_INTERFACE
```

Figure 10.17: Primitive pour l'accès à un objet par une expression pointée

Rappellons que la structure d'un *podvm* est celle d'un objet Eiffel (entête, contenu), même si le podvm est traité de manière spéciale, par l'exécutif.

**Cas d'erreurs**

Lors d'un accès à un objet plusieurs erreurs peuvent se produire:

- l'objet n'est pas accessible en lecture,
- l'objet est en cours de modification par une autre application,
- aucun monde persistant n'est accessible,
- une erreur interne s'est produite au niveau du gestionnaire,
- pour un accès explicite de l'application, l'objet n'existe pas,
- l'objet est obsolète.

Il faut propager ces erreurs (produites en général par le chargement, cf. § 10.4), au niveau de l'application afin qu'elle puisse réagir. On utilisera le mécanisme d'exception intégré dans le langage, et pour chacune des erreurs citées ci-dessus, on déclenchera respectivement les exceptions suivantes:

- *read_permission_denied*,
- *write_conflict*,
- *pom_internal_error*,
- *object_does_not_exist*,
- *object_is_obsolete*.

## 10.6. Mise à jour des objets en mémoire persistante

Nous étudions successivement les principes généraux, puis la description des adaptations pour l'intégration du mécanisme de mise à jour.

**Principes généraux**

Dans un langage de programmation classique, on ne fait pas la distinction entre les modes *consultation-en-cours* et *modification-en-cours*, car un seul exemplaire de l'objet (volatile), existe et il ne peut donc être utilisé que par l'application qui l'a créée. Dans un cadre persistant par contre, du fait du partage des objets entre plusieurs applications, il faut assurer la synchronisation entre les copies en mémoires volatile et persistante. Le mécanisme mettant en oeuvre cette synchronisation doit permettre à la fois de:

- maximiser la disponibilité d'un objet,
- minimiser les opérations de mise à jour,

• maintenir l'intégrité des objets même en cas de panne ou d'accès simultanés par plusieurs applications.

Pour répondre à ces objectifs, les systèmes persistants introduisent la distinction entre les états *consultation-en-cours* et *modification-en-cours*, ce qui s'énonce par la règle suivante:

**Règle d'accès aux objets:**
*Plusieurs applications, peuvent consulter simultanément le même objet, mais pour le modifier il faut obtenir un accès exclusif, en consultation et modification.*

Cette règle doit être mise en oeuvre au niveau du POM et l'application n'a pas à s'en préoccupper. Cependant, en cas de modification, l'exécutif du langage à la charge:

• de signaler au POM le changement d'état (*consultation-en-cours* <->*modification-en-cours*),

• de transmettre au POM les changements intervenus dans la copie volatile de l'objet, afin que celui-ci puisse les répercuter en mémoire persistante.

Le choix de la fréquence de ces ordres au POM influent grandement sur la disponibilité des objets et sur l'efficacité du mécanisme.

Le choix du moment de signalement du changement d'état est trés dépendante des possibilités du POM. Idéalement, il faudrait que l'application puisse demander:

• le passage dans l'état *modification-en-cours* juste avant l'affectation, afin d'autoriser le plus longtemps possible les autres applications en consultation (cf. § 7.1). Le principal inconvénient est la perte de temps que cela entraîne (négociation et échange avec le POM), pour chaque affectation. Il faut aussi, ne pas trop se pénaliser en:

  - évitant de faire un traitement qui sera annulé par la suite, lorsque le passage dans l'état *modification-en-cours* n'est pas autorisé par le POM,

  - favorisant la récupération des exceptions *read_conflict* et *write_conflict*;

• le retour à l'état *consultation-en-cours* au plus tôt, juste après la synchronisation

avec la mémoire persistante et permettre ainsi aux autres applications d'agir ou de consulter les objets.

Mais il y a alors des risques de section critique, si le grain est trop fin, et donc tendance à annuler le bénéfice de ce petit grain par des transactions plus longues.

Pour limiter le nombre d'échanges avec la mémoire persistante, la transmission des modifications doit être réalisée le plus tardivement possible, c'est à dire, quand une séquence d'opérations qui sont sémantiquement liées, s'achève. Cela permet de réduire les échanges entre l'application et le gestionnaire, en assurant qu'un objet n'est mis à jour qu'une fois pendant la séquence d'instructions

Notre mécanisme met à jour les objets lors de la validation d'une transaction (cf. § 10.8) au moyen de la routine *update*. Le passage de l'état *consultation-en-cours* à l'état *modification-en-cours* est propagé au gestionnaire d'objets, au moment de la validation, ce qui n'est pas l'idéal pour l'application qui réalise la modification, mais favorise les applications en train de consulter l'objet.

On notera que si le POM autorise le vérouillage d'un objet, la routine *modify* pourra prendre en compte cette fonctionnalité et placer le mécanisme dans la situation optimale présentée ci-dessus.

### Description des adaptations

L'information de modification est mémorisée à deux niveaux:

- dans l'entête de l'objet par un indicateur,
- par l'insertion dans la liste des objets modifiés *p_modified_instances* (cf. § 10.2).

Nous avons fait ce choix pour des raisons d'efficacité; un objet peut être modifié plusieurs fois pendant une transaction, mais pour garantir la cohérence des données il ne doit apparaître qu'une fois dans la liste. Plutôt que de faire chaque fois une recherche dans la liste, il suffit de tester l'indicateur de modification associé à l'objet. Par ailleurs la présence d'une liste contenant seulement les objets modifiés permet de réduire au strict nécessaire le parcours à faire pour synchroniser les objets à la fin d'une transaction.

```
deferred class RUNTIME_INTERFACE
  .....
feature
```

```
assign (object1, object2: ANY; attribute: ANY) is
        -- assign object2 to attribute of object1
    require   -- not is_error and attribute is a field of object1
    do
        if object1.is_persistent and then is_volatil (object2) then
            make_persistent (object2)
            if not is_error then
                attribute:= object2;
                if not is_error then modify (object1) else raise (last_error) end -- if
            else
                raise (last_error)
            end; -- if
        else   -- volatil object
            attribute:= object2
        end; -- if
    ensure    not is_error and object1.is_persistent implies ( is_modified (object1) and
                                object1.object_traversal.all ($post_condition2) and
                                attribute.object_traversal.all ($post_condition1))
    end; -- assign

update (podvm: PODVM) is
        -- Update object associated to podvm and make it in a reading mode
    require   not is_error and p_modified_instances.has (podvm)
    ensure
        -- object is updated in persistent memory but not its dependents
        not is_modified (podvm) and not p_modified_instances.has (podvm)
    end; -- update

modify (object: ANY) is
        -- Mark modification of object in its header and insert
        -- object in p_modified_instances
    require   not is_error and object.is_persistent
    ensure    not is_error implies  is_modified (object)  and
                                p_modified_instances.has  (podvm)

is_modified (object: ANY): BOOLEAN
        -- is modification flag in the header of object set?

set_flag_modification (object: ANY) is
        -- Set modification flag in the header of object
    ensure    is_modified (object)

unset_flag_modification (object: ANY) is
        -- Unset modification flag in the header of object
    ensure    not is_modified (object)
end -- RUNTIME_INTERFACE
```

Figure 10.19: Mise à jour des objets persistants

On sait qu'un objet ne peut être modifié que par une affectation ou une opération de création sur l'un de ses attributs (fig. 10.19). C'est lors de l'appel à ces opérations que la l'indicateur de modification est positionné; pour cela on introduit un appel à la primitive *modify* dans les opérations *assign* et *creation_assign*.

Une fois que la mise à jour est effectuée, c'est à dire une fois que l'opération *update* est exécutée (cf. § 10.8), l'indicateur mémorisant la modification de l'objet est remis à zéro.

**Remarque:**

Si l'indicateur de modification des objets se trouvait dans le *podvm* on éviterait une indirection supplémentaire. Par contre, d'une part les changements dans le code généré sont plus importants, et d'autre part il est plus rapide pour l'exécutif de mémoriser la modification dans l'objet persistant lui-même, car la mise à jour du *podvm* nécessite une recherche dans la table. Pour cette raison, nous choisissons de mémoriser l'indicateur de modification dans l'objet lui-même, et non dans le *podvm* qui lui est associé.

**Cas d'erreurs**

Lors de la modification d'un objet, plusieurs erreurs peuvent se produire:

- l'objet est en cours de consultation par d'autres applications,
- l'objet est en cours de modification par une autre application,
- l'objet est protégé contre les modifications,
- une erreur interne s'est produite au niveau du gestionnaire,

Pour chacune des erreurs citées ci-dessus, on déclenchera respectivement les exceptions suivantes:

- *read_conflict*,
- *write_conflict*,
- *write_permission_denied*,
- *pom_internal_error*.

## 10.7. Libération d'objets en mémoire volatile

La libération de la copie d'un objet de la mémoire associée au processus est entièrement à la charge du ramasse-miette. Dès que l'objet n'est plus référencé par aucun autre objet, il peut être libéré. Dans le cas où l'objet a été modifié et n'a pas été mis à jour, alors avant d'être libéré, les changement devront être transmis au gestionnaire d'objets.

On modifie le ramasse-miettes afin de prendre en compte les objets persistants (fig. 10.20). Les principales modifications concernent:

• le marquage des objets persistants qui peuvent être libérés par le ramasse-miettes s'il ne sont plus référencés par aucun objet et la suppression de leur descripteur,

• le marquage des descripteurs qui ne sont plus référencés, et leur suppression ultérieure, si nécessaire,

• la combinaison de ces deux aspects pour éviter la libération d'un objet dont le *podvm* est encore référencé, et vice-versa;

• la mise à jour d'un objet modifié.

On rappelle que laprise en compte du *podvm* par le ramasse-miettes (intégration de la routine free_podvm), nécessite l'adjonction d'un mot dans chaque *podvm* (cf. § 10.2) et d'opérateurs de parcours dans la classe *P_TABLE* (start, off, forth, item).

```
deferred class RUNTIME_INTERFACE
  .....
feature     .....
  free (object: ANY) is
        -- free object (to be integrated in corresponding garbage collector routine)
    do
        if object.is_persistent then  -- object is not podvm
            if is_modified (object) then
                update (p_instances.item_object  (object))
            end; -- if
            p_instances.item_object (object).set_voi (void);
            free_object (object)
        end -- if
    end -- free

  free_object (object: ANY) is      -- Free object
    external "C"
    ensure    -- object is in the free list handle by the Run-time

  mark (object: ANY) is             -- mark object as a candidate to be freed
    external "C"
    ensure    is_marked  (e)

  unmark (object: ANY) is
        -- Make object not a candidate to be freed
        -- (routine of garbage collector )
    external "C"
    ensure    not is_marked (e)

  is_marked (object: ANY) is
        -- Is object a candidate to be freed
    external "C"
```

```
free_podvm is
        -- Free podvm which are not referenced anymore
    do
        from p_instances.start
        until p_instances.off
        loop
            if      not is_marked (p_instances.item) and
                (   p_instances.item.voi = void or else
                    not is_marked (p_instances.item.voi))
            then
                free (p_instances.item)
            else
                unmark (p_instances.item)
            end; -- if
            p_instances.forth
        end -- loop
    end; -- free_podvm


end -- class RUNTIME_INTERFACE
```

Figure 10.20: Libération d'un objet persistant de la mémoire volatile

Quand un objet sera retenu par le ramasse-miette pour être libéré, alors la routine *free* lui sera appliquée. Dans le cas où on choisit de traiter les *podvm* comme des objets Eiffel ordinaires, le ramasse-miettes va les considérer comme les autres et pouvoir les libérer. Cependant, cette libération ne peut suivre les règles utilisées pour les objets volatiles ou persistants, car au fur et à mesure qu'il est accédé par les objets, le *podvm* est de moins en moins référencé par les attributs, (après le 1er accès ils référencent directement la copie volatile de l'objet persistant: cf. § 10.5); un podvm qui n'est plus référencé peut donc continuer à être utile.

Il faudra donc, dans l'implémentation de la routine *free*, prendre garde à ne jamais libérer un *podvm*. Cette libération se fera par la routine free_podvm qui effectue le parcours de la table des objets persistants (*p_instances).*

Cependant cette libération ne se fera que si les deux conditions suivantes sont satisfaites:

- le podvm est un objet candidat à la libération,
- la copie de l'objet persistant associé au podvm a été libéré ou est candidat à l'être.

Une fois la libération effectuée, nous proposons de gérer les *podvm* libres comme les autres objets libres.

On notera que la libération des objets persistants suit exactement la même philosophie que celle des objets volatiles; en effet, comme pour ces derniers, une application trop gourmande en place pourra saturer la mémoire virtuelle (cette situation

est plus rare avec un ramasse-miettes).

La libération des objets de la mémoire persistante, sera géré par le gestionnaire d'objets, suivant le même principe; cependant on mettra en oeuvre l'approche décrite dans le paragraphe 3.5 (concept d'objet obsolète).

**Remarques**

La référence d'un *podvm* vers un objet persistant ne doit pas être prise en compte par le ramasse-miettes pour décider de la libération.

La méthode décrite ci-dessus a l'inconvénient suivant: si un objet persistant n'est plus référencé par d'autres objets, alors même si le descripteur est encore utilisé, l'objet sera libéré (mais pas le descripteur).

**Cas d'erreurs**

Lors de la libération d'un objet, plusieurs erreurs peuvent se produire:

- l'objet est protégé contre les modifications,
- une erreur interne s'est produite au niveau du gestionnaire,

Pour chacune des erreurs citées ci-dessus, on déclenchera respectivement les exceptions suivantes:

- *write_permission_denied*,
- *pom_internal_error*.

## 10.8. Mise en oeuvre des transactions

Afin de garantir l'intégrité des objets, nous avons montré dans le paragraphe 7.1 comment associer une transaction aux routines exportées d'une application Eiffel. Nous étudions maintenant comment implémenter ce mécanisme. En début et en fin de chaque primitive exportée nous rajouterons un appel aux routines *if_start_transaction* et *if_validate_transaction*; par ailleurs au cas ou une exception est propagée jusqu'à la racine de l'application, alors un ordre d'annulation de la transaction sera transmise au POM (*abort_transaction*);

Par ailleurs, il sera possible d'activer ces méthodes à partir d'un composant nommé *TRANSACTION*, appartenant à un système de classes Eiffel (cf. § 7.1).

Le mécanisme mis en oeuvre garantit la présence d'une seule et unique transaction à

tout moment de l'exécution d'un programme, que la transaction ait été déclenchée explicitement par l'application ou implicitement par l'exécutif (fig. 10.21).

```
deferred class RUNTIME_INTERFACE
  .....
feature     .......
  if_start_transaction is
        -- start a new transaction if it is required
     do
        if not is_transaction and Current.is_persistent and level = 0 then
           begin_transaction;                              -- feature of POM_INTERFACE
           level:= 1
        elseif is_transaction and level > 0
           level:= level + 1
        end -- if
     end -- if_start_transaction

  validate_transaction is
        -- validate current transaction if it is required
     do
        if is_transaction then level:= level - 1 end; -- if
        if is_transaction and level = 0 then
           from p_modified_instances.start
           until p_modified_instances.off
           loop
              update (p_modified_instances.item); p_modified_instances.forth
           end; -- loop
           commit_transaction;                             -- feature of POM_INTERFACE
           p_modified_instances.wipe_out
        end -- if
     ensure     all ($precondition5)
     end -- validate_transaction

  precondition5 (object: PODVM): BOOLEAN is
        -- precondition of routine validate_transaction
     do     Result:= not is_modified (object.voi)
                    -- and object.poi is updated in persistent memory
     end; -- precondition5

  cancel_transaction is
        -- cancel transaction if necessary
     do
        level:= 0
        abort_transaction;                                 -- feature of POM_INTERFACE
        p_modified_instances.wipe_out
     ensure     all ($precondition6)
     end -- cancel_transaction

  precondition6 (object: PODVM): BOOLEAN is
        -- precondition of routine abort_transaction
     do
           Result:= not is_modified (object.voi)
                    -- and object.poi is not updated in persistent memory
     end; -- precondition6
```

```
  is_transaction: BOOLEAN is
        -- Is there a current transaction
    do    Result:= level > 0
    end; -- is_transaction

  level: INTEGER is    -- For handling nested routines

end -- class RUNTIME_INTERFACE
```

Figure 10.21: Primitives pour la mise en oeuvre du mécanisme transactionnel

**Cas d'erreurs**

Si une exception est déclenchée, il ne faut pas immédiatement annuler la transaction, il faut laisser le temps à l'application de récupérer cette erreur, soit au niveau de la routine où l'erreur s'est produite, soit au niveau d'une routine appelante. Au cas où l'erreur n'est pas récupérée par l'application, il faut alors annuler la transaction en cours.

# 10.9. Gestion de l'accès associatif

Les techniques de gestion de l'espace des objets proposées par les langages de programmation (ramasse-miettes), sont très adaptées à un accès navigationnel des objets. Par contre elles ne peuvent prendre en compte des accès massifs comme les sélections. Ainsi, comment exprimer qu'un objet appartenant à une **sélection** (voir chapitre 6), doit être libéré?

Par ailleurs, on a vu la nécessité de pouvoir réaliser des compositions de sélections ou de ré-utiliser le résultat d'une sélection, plus loin dans le déroulement du programme.

Pour ces deux raisons, il semble donc intéressant de faire en sorte que le résultat d'une sélection reste sous le contrôle du gestionnaire d'objets. Ces objets ne sont donc pas présents dans la table *p_instances* des objets persistants atteints par navigation.

**Règle: Gestion de l'accès associatif**

*Le résultat global d'une sélection gérée par le gestionnaire d'objets est géré par lui et n'affecte pas le nombre d'objet persistants présents dans la mémoire volatile de l'application.*

Par contre, dès qu'ils sont effectivement utilisés par l'application, c'est à dire lorsque qu'ils sont extraits de la collection et attachés à d'autres objets, leur gestion sera réalisée par l'exécutif comme ci-dessus. Cette extraction se fera essentiellement à travers la routine item de la classe *COLLECTION*.

# 10.10. Optimisation des échanges

Afin d'augmenter les performances, on peut essayer d'optimiser le nombre d'objets à charger (et donc à libérer), dans la mémoire volatile. Cette optimisation peut être guidée par la sémantique des types et la nature des programmes.

## 10.10.1. Affinités sémantiques

Les *collection* s peuvent contenir un grand nombre d'objets persistants qui ne sont pas forcément utile au même moment. De plus, le traitement de ces objets se fait souvent par l'intermédiaire de sélections qui restreignent l'ensemble des objets à considérer (primitive *select*). L'accès à une *collection* ne doit donc pas entraîner le chargement de tous les objets qu'elle contient (cf. § 10.8).

Quand on considère un tableau, une liste, ou un ensemble, c'est en général pour accéder à tous leurs éléménts. Le chargement de telles instances entraînera donc le chargement de tous les objets qu'elles référencent.

Pour les autres instances, il est pratiquement impossible d'anticiper (à moins de mettre en oeuvre un mécanisme statistique coûteux), le besoin de chargement d'un objet, à partir du chargement d'un objet qui le référence. Ainsi, nous avons préféré dans ce cas un mécanisme de chargement à la demande (cf. § 10.5).

Ces politiques de chargement à la demande pourront être remplacés par d'autres politiques, sur demande explicite du programme.

## 10.10.2. Objets expansés

Pour lier un objet à un autre objet par un lien de clientèle privé (attribut de type *expanded*), cela suppose une liaison forte entre les objets. Cette liaison est à rapprocher de la notion de lien *composite* que l'on retrouve dans *Orion* [Kim 87 & 89]. Lorsque de tels liens unissent des objets, on constate souvent que la consultation de l'un suppose la consultation de ses attributs expansés. Ainsi, le chargement d'un objet doitt entraîner le chargement des objets liés à lui par un attribut expansé. C'est particulièrement adapté pour les types simples (*INTEGER, REAL, BOOLEAN, CHARACTER*).

# 10.11. Traitement des opérateurs

Pour l'opérateur de création, le seul point à prendre en compte, est le cas où l'objet créé devient persistant pour respecter le principe d'intégrité référentielle; il est réglé par la routine *creation_assign*.

## 10.11.1. opérateur d'égalité

L'opérateur d'égalité de référence (=) devra vérifier l'homogénéité des membres de l'égalité (fig. 10.22), afin de ne tester l'égalité de deux références que si elles adressent des objets de même type:

- podvm,
- objet persistant,
- objet temporaire.

La routine testant l'égalité de contenu (*equal*) doit auparavant s'assurer que l'objet est chargé en mémoire.

```
deferred class RUNTIME_INTERFACE
  .....
feature
  .......
  object_kind (object: ANY): INTEGER is
        -- Return the kind of object
    require   not object = Void
    external "C"
    ensure    object.is_persistent  or is_volatil  (object) or is_podvm  (object)
    end;  -- object_kind

  old_equal (object1,  object2: ANY)is
        -- Return true if contents of object1 is equal to contents of object2
        -- (equal routine from a non persistent realease)
    require   not object1 = void and then not is_podvm (object1);
              not object2 = void and then not is_podvm (object2)
    external "C"
    ensure    object1 = old object1 and object2 = old object2;
    end;  -- old_equal

  infix "=" (object1, object2: ANY): BOOLEAN is
        -- Return True if references to object1 and object2 are equal
    do
          if object_kind  (object1) = object_kind  (object2) then
              Result:= object1 = object2
          elseif is_podvm (object1) and then not object1.voi = void then
              Result:= object2 = object1.voi
```

```
            elseif is_podvm (object2) and then not object2.voi = void then
                Result:= object1 = object2.voi
            else
                -- Result:= False
            end    -- if
        end; -- "="

    equal (object1, object2: ANY): BOOLEAN is
            -- Return true if contents of object1 is equal to contents of object2
        local
            temp1, temp2: ANY
        do
            if is_podvm (object1) then
                if not object1. is_loaded then   temp1:= load_object (object1.poi)
                else temp1:= object1.voi
                end -- if
            else   temp1:= object1
            end; -- if
            if is_podvm (object2) then
                if not object2. is_loaded then   temp2:= load_object (object2.poi)
                else temp2:= object2.voi
                end -- if
            else   temp2:= object2
            end; -- if
            Result:= old_equal (temp1, temp2);
        ensure    object1 = old object1 and object2 = old object2
        end; -- equal

end -- class RUNTIME_INTERFACE
```

Figure 10.22: Adaptation de l'opérateur d'égalié de référence

## 10.11.2. Opérateur de suppression

La valeur *void* doit avoir un équivalent dans sa représentation en mémoire persistante. Il faudra assurer sa traduction lors de la mise à jour de l'attribut et de l'objet concernés. Etant donné que dans la version 3.1 d'Eiffel, la valeur *void* est une primitive de type *NONE* (classe héritière de toutes les classes), il faudra s'assurer que tous les attributs ayant la valeur *void*, référencent un même objet de type *NONE*. La difficulté est de représenter la classe *NONE* en $O_2$, à cause du nombre d'ancêtres.

## 10.11.3. Opérateurs de duplication et de copie

Avant de faire la duplication ou la copie d'un objet, il faut s'assurer que cet objet n'est pas un descripteur d'objet persistant (podvm); si c'est le cas, la routine de duplication (*clone*) ou de copie (*copy*) devra faire en sorte de porter sur l'objet lui-même, en réalisant au besoin des chargements d'objets (fig. 10.23). Normalement ceci devrait être pris en compte par l'exécutif, mais les routines, qui impliquent la structure des objets sont écrites en C[1].

```
deferred class RUNTIME_INTERFACE
  .....
feature     .......
  old_clone (object: ANY): ANY is
        -- Return clone of object (clone routine from a non persistent realease)
    require    not object = void and then not is_podvm (object)
    external "C"
    ensure     not Result.is_persistent
    end;  -- old_clone

  old_copy (object1, object2: ANY)is
        -- copy object1 on object2 (copy routine from a non persistent realease)
    require    not object1 = void and then not is_podvm (object1)
               not object2 = void and then not is_podvm (object2)
    external "C"
    ensure     object1 = old object1 and object2 = old object2; object1.equal (object2)
    end;  -- old_copy

  copy (object1, object2: ANY) is
        -- copy object1 on object2
    local temp1,  temp2: ANY
    do
          if  is_podvm (object1) then
              if not object1. is_loaded then   temp1:= load_object  (object1.poi)
              else temp1:= object1.voi
              end -- if
          else   temp1:= object1
          end; -- if
          if  is_podvm (object2) then
              if not object2. is_loaded then   temp2:= load_object (object2.poi)
              else temp2:= object2.voi
              end -- if
          else   temp2:= object2
          end; -- if
          old_copy (temp1,  temp2);
    ensure     object1 = old object1 and object2 = old object2; object1.equal (object2)
    end; -- copy
```

[1] A l'heure actuelle nous ne disposons que des spécifications de la version 3.1 d'Eiffel. Pour les remarques concernant l'implémentation de l'exécutif Eiffel, nous nous basons donc sur la version 2.3 du langage.

```
clone (object: ANY): ANY is
        -- Return a clone of object
    local temp: ANY
    do
        if is_podvm (object) then
            if not object. is_loaded then    temp:= load_object (object.poi)
            else temp:= object.voi
            end -- if
        else   temp:= object
        end; -- if
        Result:= old_clone (temp);
    ensure    not Result.is_persistent
    end; -- clone


end -- class RUNTIME_INTERFACE
```

Figure 10.23: Adaptation de l'opérateur d'égalié de référence

# 10.12. Interface pour les routines externes

En ce qui concerne les routines externes (cf. § 5.1.2) il faut distinguer celles qui agissent sur la structure des objets, de celles qui font appel à des concepts extérieurs à Eiffel (fichier unix, aspect graphique, ....).

Celles qui agissent sur la structure des objets doivent être ré-écrites, afin de prendre en compte explicitement le chargement des objets, la propagation des modifications et la libération des objets (cf. § 10.10); il s'agît en particulier des classes *ANY*, *STRING* et *ARRAY*.

Pour les autres, aucune modification n'est nécessaire si aucun objet Eiffel persistant n'est chargé (classes *FILE*, *STD_FILES*, *WINDOW*, ...).

D'une manière plus générale, il faut offrir une interface externe permettant de charger, mettre à jour et libérer un objet manipulé à partir du langage C. Cette interface peut être un sous-ensemble des primitives de la classe *RUNTIME_INTERFACE* qui contiendrait:

- start_transaction, validate_transaction, cancel_transaction,

- load_object, make_persistent

# 11

# Optimisation des accès associatifs

Comme les traitements sélectifs doivent s'adapter au langage Eiffel, il est utile de minimiser les interactions entre l'exécutif Eiffel et le gestionnaire d'objets persistants (POM).

Dans le paragraphe 9.2, nous avons indiqué deux approches possibles pour propager les types dans le monde persistant:

- les attributs seuls,
- les routines et les attributs.

Dans le deuxième cas, les problèmes d'optimisation sont confiés au POM. Dans le premier, il est utile de transformer les sélections qui font intervenir des routines dans leur critère pour réduire les chargements d'objets.

Ce chapitre propose quelques améliorations possibles.

## 11.1. Motivations

Evaluer un critère de sélection en mémoire volatile signifie appliquer ce critère sur des objets qui se trouvent dans cette mémoire; cela nécessite donc le chargement préalable de ces objets, mais autorise la participation d'objets volatiles.

Les échanges d'objets entre les mémoires persistante et volatile sont coûteux, non seulement à cause des accès disques, mais aussi à cause du dialogue entre l'exécutif Eiffel et le POM et les nécessaires transformations de représentation entre les copies d'un objet en mémoire volatile (conformément à l'exécutif Eiffel) et en mémoire persistante (définie par le POM).

Une requête de sélection peut engendrer de nombreux échanges d'objets, en particulier lorsque le critère de sélection est complexe. Il est coûteux d'envisager le

chargement un à un des objets persistants d'une *collection*, pour leur appliquer ensuite le critère de sélection en mémoire volatile. Ceci, d'autant plus que le gestionnaire d'objets persistants est normalement capable d'effectuer des sélections en utilisant des méthodes éprouvées des bases de données, à base d'index notamment.

L'amélioration proposée ici vise:

• à minimiser les échanges en diminuant le nombre d'objets intervenant dans les critères,

• à maximiser le travail de sélection du gestionnaire d'objets pour profiter de ses propres optimisations.

Une expérience semblable que nous avons mené [Lahire 91, Lahire & al. 91], lors d'un interfaçage entre Eiffel et le SGBD relationnel Oracle montre un gain de vitesse dans un rapport de 5.

Le résultat rendu par une évaluation, que ce soit en mémoire persistante ou volatile, est un ensemble d'identificateurs d'objets persistants .

Les principes d'amélioration exposés ci-dessous peuvent être réalisés soit à la compilation (si le compilateur sait reconnaître un critère de sélection), soit lors de l'exécution de la requête de sélection. Dans le premier cas, le compilateur peut produire directement un code amélioré; dans le second cas, l'évaluation du critère est améliorée puis interprétée par l'exécutif, mais cela pose le problème de l'accès au texte source des classes.

## 11.2. Critères d’amélioration des requêtes

Il semble raisonnable de baser les choix d’amélioration à la fois sur la taille de l'ensemble d'objets sur lequel porte la requête et sur le nombre d'objets persistants déjà chargés en mémoire volatile.

### 11.2.1. Amélioration de la recherche sur les propriétés.

Dans le monde des bases de données, notamment relationnelles, l'accent est mis sur l'amélioration de l'accès par attribut. En effet, une relation ne peut contenir que des attributs et les langages de requêtes ne permettent que l'utilisation d'opérateurs ou de procédures prédéfinis. Ces opérateurs ou ces procédures ne portent que sur des types simples, ils sont sans effet de bord et ont une sémantique précise qui ne peut être modifiée par l'application, comme c'est le cas dans l'approche orientée objet.

L'amélioration de l'accès par les propriétés se fait par la création et la maintenance de structures d'index sur ces propriétés. Lors de l'exécution, le mécanisme de sélection n'accède pas effectivement aux objets qui ne sont donc pas chargés, mais utilise des index qui permettent d'atteindre directement l'ensemble des valeurs d'un attribut pour différents objets.

Dans notre approche, la gestion des index est déléguée au gestionnaire d'objets persistants. Le résultat d'une évaluation en mémoire persistante est un ensemble d'identificateurs d'objets persistants (POI). La stratégie permettant de décider à quels attributs sont associés des index reste à définir.

### 11.2.2. Amélioration de la recherche par les routines

Les études du modèle relationnel ont montré que son pouvoir de modélisation était insuffisant. L'orientation actuelle est soit de construire des SGBD étendant les possibilités du modèle relationnel (introduction des types abstraits, de la possibilité d'avoir des méthodes définies par l'utilisateur), soit la réalisation de SGBD orientés objets. Dans les deux cas, le problème de la prise en compte des méthodes dans un critère de sélection apparait; les mécanismes d'amélioration classiques ne sont plus applicables, puisqu'une fonction ne peut s'exécuter que sur des objets se trouvant en mémoire centrale.

Dans Postgres [Stonebraker 90], on considère trois catégories de fonctions; les possiblités d'amélioration sont différentes pour chacune d'elles:

- les fonctions qui permettent la description d'un type d'objet défini par l'utilisateur; dans ce cas l'accès n'est pas amélioré et les objets sont tous chargés en mémoire, afin d'évaluer le critère de sélection;
- les fonctions nommées *operator*; cette catégorie regroupe tous les opérateurs de

comparaison (<, ≤, >, ≥, =, ≠). L'évaluation des critères de sélection qui les utilisent se fait à travers des index associés à chacun des attributs (s'ils en ont). Cependant ces opérateurs ne s'appliquent qu'à des attributs de type de base. De nouveaux types de base, et leurs opérateurs peuvent être définis par l'utilisateur, mais cela suppose que ce dernier décrit un certain nombre de fonctions, en particulier pour la recherche ou le stockage de l'objet.

• les fonctions qui encapsulent un critère de sélection, ce qui permet d'écrire une fois pour toutes un traitement souvent utilisé.

La philosophie de notre approche est de servir avant tout la transparence du traitement de la persistance aux programmes: c'est au système de prendre en charge l'amélioration des opérations de stockage et de chargement des objets. Lorsque de nouveaux types sont construits par le développeur, celui-ci ne doit pas décrire de méthodes pour le système; par conséquent, l'usage de fonctions de type *operator* pour implanter des opérateurs de comparaison est écarté, sauf pour des types simples (integer, real, character ou éventuellement string).

On pourrait aussi citer le principe fondamental de génie logiciel selon lequel on ne doit pas placer dans des composants réutilisables d'informations spécifiques à une application.

Par ailleurs, l'emploi de fonctions est beaucoup plus fréquent dans le cadre de programmes que dans celui de bases de données. Au niveau de l'intégration dans Eiffel, deux approches sont possibles:

• laisser l'exécution en mémoire volatile, des routines à la charge de l'exécutif Eiffel, et implanter un mécanisme permettant de diminuer le nombre d'objets sur lesquels la routine sera appliqué,

• rendre les routines et leur exécution persistantes, et laisser l'amélioration à la charge du gestionnaire d'objets.

Avec la première solution, nous devons donc effectuer des améliorations sur les critères de sélections qui comportent soit des fonctions définies par l'utilisateur, soit des expressions pointées (une expression pointée peut être assimilée à une fonction retournant l'objet référencé par un attribut). L'amélioration sur les fonctions portera sur l'évaluation en mémoire volatile. Elle consiste essentiellement à retarder l'évaluation

des fonctions afin de minimiser le nombre d'objets sur lesquels la fonction sera appliquée: seuls les objets ayant passé la barrière de sélection des attributs seront considérés. Ce retardement sera obtenu au prix de l'analyse et de la réorganisation du critère de sélection.

La deuxième solution implique qu'il faille mettre en oeuvre un mécanisme général qui permette de traduire une classe Eiffel en une classe O2, afin que le gestionnaire puisse disposer de sa propre représentation des objets (structures et méthodes), afin de gérer complétement l'exécution des requêtes (cf. § 9.2 et chapitre 12).

### 11.2.3. Evaluation unique des fonctions.

Dans le cas où les fonctions utilisées dans le critère de sélection ne produisent pas d'effet de bord, les attributs et les fonctions du contexte de la requête sont constants pour une requête donnée. On peut les évaluer une seule fois par requête, indépendamment du nombre d'objets sur lequel porte la requête.

Si par contre, une fonction produit des effets de bord, le résultat de la requête est totalement imprévisible, puisqu'il est impossible de connaître l'ordre dans lequel les objets seront examinés.

Soit par exemple une classe *QUERY_WITH_SIDE_EFFECT* (fig. 11.1), permettant la consultation d'une *collection* d'objets de type *PERSON*. L'ordre dans lequel seront examiné les objets intervient dans la construction du résultat, ce qui est contraire à la sémantique d'une sélection sur une *collection*.

```
class QUERY_WITH_SIDE_EFFECT
      -- A class PERSON exists and contains an attribute age
   …
feature
   att: INTEGER;
   …
   fonc: INTEGER is
      do
            Result:= att;
            att:= att + 1
      end; -- fonc

   criteria (p: PERSON): BOOLEAN is
      do
            result:= p.age = fonc
      end; -- criteria
end -- Class QUERY_WITH_SIDE_EFECT
```

Figure 11.1: Critère de sélection avec effet de bord

Si l'ordre d'examen des personnes est $P_{10}$, $P_{11}$, $P_{12}$ (l'indice représente l'âge de la personne), et si l'attribut *att* vaut 10 au lancement de la requête, le résultat est l'ensemble $\{P_{10}, P_{11}, P_{12}\}$; par contre, si l'examen se fait dans l'ordre inverse, le résultat est l'ensemble $\{P_{11}\}$. Encore faut-il remarquer que nous nous limitons ici au cas où un contexte agit uniquement sur lui-même. Les conséquences d'utilisation de fonctions avec effets de bord sur des collections d'objets persistants seraient bien pire.

Dans la mesure où l'utilisation de fonctions avec effets de bords rend le résultat d'une requête imprévisible, la solution la plus simple est de considérer que les fonctions du contexte n'ayant comme paramètres que des primitives du contexte ne sont évaluées qu'une seule fois par requête, qu'elles aient ou non des effets de bord. Ceci a l'avantage de garantir dans tous les cas un résultat plus cohérent, tout en permettant un gain de temps: exécution unique et possibilité d'effectuer certaines sélections d'objets en mémoire persistante et non en mémoire volatile.

Notons que cette amélioration est exclue pour les autres appels de fonctions:

• si au moins un paramètre de l'appel est une primitive de la classe sur laquelle s'opère la sélection (ou une primitive d'une classe dont elle est cliente), la valeur retournée par la fonction est dépendante de la valeur de ce paramètre, qui elle même est propre à chaque objet de la *collection* (excepté les attributs constants et les routines *once*).

• si la fonction est une routine de la classe sur laquelle s'opère la sélection (ou une routine d'une classe dont elle est cliente), son résultat est certainement dépendant des valeurs des objets de la classe.

### 11.2.4. Evaluation en court-circuit.

En mémoire volatile, contrairement à l'évaluation en mémoire persistante où le gestionnaire d'objet est seul maitre des améliorations, il est possible d'arrêter une évaluation dès le moment où l'évaluation d'un critère ne peut plus que réussir ou qu'échouer (évaluation en court-circuit), selon les définitions des opérateurs booléens:

• A **or** B ⇔ A **or else** B ⇔ if A then true else B;
• A **and** B ⇔ A **and then** B ⇔ if A then B else false;
• A **implies** B ⇔ if A then B else true.

Remarque: Le court-circuit n'est pas possible dans le cas du *xor*:

A **xor** B ⇔ if A then ¬B else B

### 11.2.5. Exemple d'amélioration.

Ces principes étant établis, l'amélioration attendue pour évaluer le critère de la figure 11.2, est la suivante:

```
criteria (O: MYCLASS): BOOLEAN is
  do
        Result:= (O.ac1 < O.ac2.af) and O.ac2.ff (1) and (O.ac3 or O.ac4 or O.ac5 = fx (sx))
                  and then (O.fc (ax) + O.ac5 > 3)
  end -- criteria
```

Figure 11.2: Expression booléenne correspondant à un critère de sélection

• évaluer les termes de l'expression correspondant à des fonctions du contexte, ici fx (sx), mémoriser le résultat dans un pseudo-attribut local L; les valeurs des attributs du contexte sont directement accessibles et n'ont pas besoin d'être mémorisées;

• sur une *collection* d'instances de *MYCLASS*, évaluer en mémoire persistante (évaluation sous-traitée au POM) la *collection* répondant au critère:

ac1 < ac2.af **and** (ac3 **or** ac4 **or** ac5 = L)

soit C la *collection* résultat de cette évaluation;

• sur les objets de C, évaluer en mémoire volatile le critère (évaluation en *court-circuit*, qui peut économiser l'évaluation du second opérande):

ac2.ff (1) **and then** (fc (ax) + ac5 > 3)

qui donne la *collection* résultat.

Une partie de l'évaluation profite ainsi des mécanismes de sélection du gestionaire d'objets; le chargement d'objet est limité puisque la *collection* sur laquelle s'effectue l'évaluation en mémoire volatile est probablement moins importante que la *Pcollection* [Knut 73].

**Remarque:**

En faisant l'hypothèse que tout objet sélectionné n'est pas forcément consulté, on pourrait limiter encore plus le chargement des informations en mémoire (par exemple extraire seulement du gestionnaire d'objets un vecteur de bit permettant d'atteindre localement les objets sélectionnés). Ce vecteur de bits implique un ordre de rangement des objets dans la *Pcollection*. Cette approche peut avoir des inconvénients en cas de modification de la *collection* (ajout ou retrait d'objets) par d'autres processus. Il faudrait alors interdire ces modifications tant que le résultat de la sélection est utilisé par le programme, ce qui n'est pas réaliste.

## 11.3. Mécanisme d'amélioration des requêtes

Le critère d’une requête de sélection est analysé dans le but d'en différencier les parties qui peuvent être évaluées en mémoire persistante et celles qui nécessitent le chargement (et éventuellement la mise à jour) d'objets persistants. Cette analyse construit un arbre abstrait de l’expression formant le critère de sélection, calcule pour chaque noeud un poids, représentant son coût d'évaluation, puis réorganise cet arbre en fonction des coûts associés à chaque noeud, tout en tenant compte de la sémantique des opérateurs.

Cette réorganisation permet d’effectuer (quand c'est possible) en premier l'évaluation en mémoire persistante, puis celle en mémoire volatile, tout en respectant la sémantique d’Eiffel. Ensuite le système doit rassembler les ensembles d'objets sélectionnés afin de construire le résultat final de la requête.

### 11.3.1. Classes de primitives.

Nous serons amenés à distinguer (fig. 11.3):

• les primitives du contexte: attributs et routines de la classe où est définie la requête, paramètres et attributs locaux de la routine d'où est appelée la requête;

• les primitives de la *collection* courante: attributs et routines de la classe de base de la collection sur laquelle s'effectue la requête;

• les primitives appartenant à des collections de fournisseurs: attributs et routines d'une classe dont la classe de base de la collection, sur laquelle s'effectue la requête, est cliente.

```
criteria (O: MYCLASS): BOOLEAN is
  do
        result:=
              (O.ac1 < O.ac2.af) and O.ac2.ff (1) and (O.ac3 or O.ac4 or O.ac5 = fx (sx))
              and then (O.fc (ax) + O.ac5 > 3)
     end -- criteria

avec:
            • ac1, ac2, ac3, ac4, ac5, fc: attributs et fonction de la collection courante;
            • ax, sx et fx: attributs et fonction du contexte;
            • af, ff: attribut et fonction d'une collection de fournisseur;
```

Figure 11.3: Classes des primitives d'un critère de sélection

### 11.3.2. Arbre booléen.

L'arbre de syntaxe abstraite (arbre abstrait pour simplifier par la suite) d'une expression est l'arbre dans lequel chaque opérateur de l'expression est représenté par un noeud dont les fils sont les opérandes de cet opérateur. Evidemment, seuls les opérateurs des types de base sont considérés comme opérateurs dans l'arbre abstrait; les autres ne sont en fait que des fonctions déguisées et apparaissent comme appels de fonction dans l'arbre.

Soit A l'arbre abstrait d'une expression booléenne; on appellera *sous-arbre booléen* tout sous-arbre de A dont les noeuds sont des opérateurs booléens (not, or, and, or else, ... ) et dont les feuilles sont des *feuilles booléennes*. Une *feuille booléenne* est un sous-arbre qui représente une sous-expression booléenne (attributs de type booléen, comparaisons,

...), mais dont la racine n'est pas un opérateur booléen. Les autres sous-arbres (expressions arithmétiques par exemple) seront dits non booléens.

**Note**: Une fonction booléenne contenant un paramètre qui est un sous-arbre booléen est donc considérée comme une feuille booléenne.

Pour simplifier les opérations qui seront effectuées par la suite sur l'arbre, les opérateurs booléens associatifs sont considérés comme n-aires (fig. 11.4): par exemple, pour une expression de la forme A **or** B **and then** C **or** D, l'arbre est construit comme si les opérateurs **or** et **and then** étaient préfixés, et comme si l'expression était écrite: **or** (A, **and then** (B, C), D).

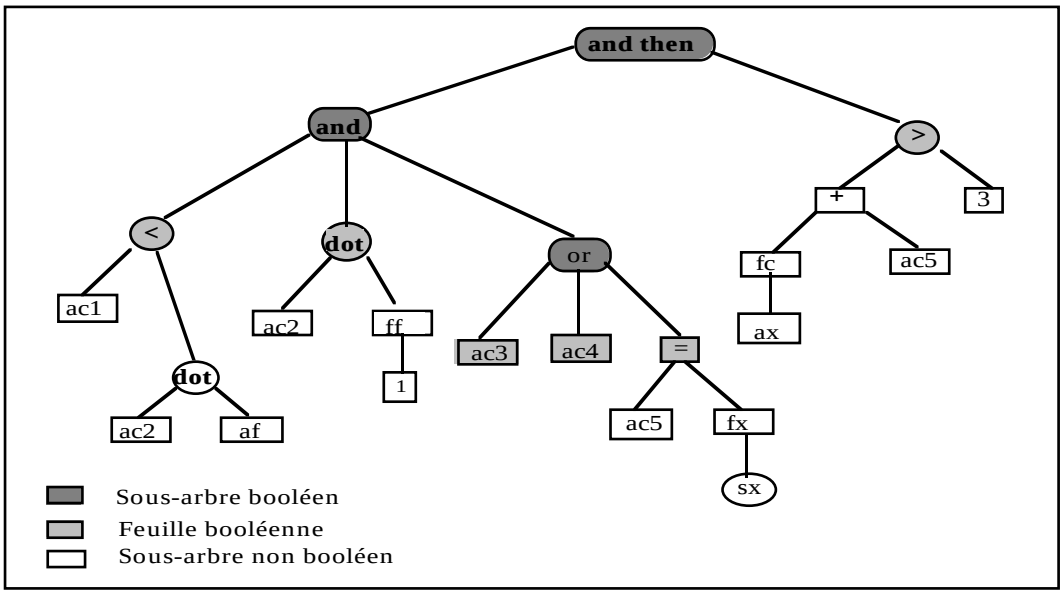


fig ure 11.4: Arbre abstrait de l'expression de la fig. 11.2

### 11.3.3. Poids d'un sous-arbre booléen.

Le poids d'un sous-arbre booléen synthétise le coût de l'évaluation de chacune des caractéristiques qui forment la sous-expression correspondante. Ce coût est fonction de la nature de ces caractéristiques (routine ou attribut) et de leur localisation (dans le contexte, dans la *collection* courante ou dans une *collection* de fournisseur). Le poids synthétisant ce coût est représenté par un quintuplet (Ff, Fc, Af, Ac, C), dont chaque élément représente le nombre d'occurences des caractéristiques, selon leur nature et leur localisation:

- Ff: fonctions appartenant à une collection de fournisseur,
- Fc: fonctions appartenant à la collection courante,
- Af: attributs appartenant à une collection de fournisseur,
- Ac: attributs appartenant à la collection courante,
- C: fonctions et attributs du contexte de la requête.

Un poids A de valeurs ($a_1$, $a_2$, $a_3$, $a_4$, $a_5$) est supérieur à un poids B de valeurs ($b_1$, $b_2$, $b_3$, $b_4$, $b_5$) s'il existe i ($1 \leq i \leq 5$) tels que $a_i > b_i$ et ($i = 1$ ou $a_j = b_j$ pour tout j tel que $1 \leq j < i$).

Le poids Pf d'une feuille est déterminé comme suit:

- Pf = (0, 0, 0, 0, 0) pour un littéral ou un attribut constant;
- Pf = (0, 0, 0, 0, 1) pour un attribut du contexte;
- Pf = (0, 0, 0, 1, 0) pour attribut de la collection courante;
- Pf = (0, 0, 1, 0, 0) pour un attribut d'une collection de fournisseur;
- Pf = (0, 1, 0, 0, 0) pour une fonction de la collection courante;
- Pf = (1, 0, 0, 0, 0) pour une fonction d'une collection de fournisseur;

Le poids Pn d'un noeud représentant un opérateur (booléen ou non) est la somme des poids de ses N sous-arbres:

$$Pn = \sum_{i=1}^{N} (Ff_i, Fc_i, Af_i, Ac_i, C_i) = (\sum_{i=1}^{N} Ff_i , \sum_{i=1}^{N} Fc_i , \sum_{i=1}^{N} Af_i , \sum_{i=1}^{N} Ac_i, \sum_{i=1}^{N} C_i )$$

L'opérateur lui-même n'a pas de poids.

Le poids d'un noeud correspondant à un appel de fonction est la somme des poids de ses paramètres et du poids propre de la fonction, sauf pour une fonction du contexte ayant au moins un paramètre qui n'est pas du contexte; Si Pf est le poids de la fonction, Pp la somme des poids de ses paramètres, le poids Pa de l'appel est calculé comme suit:

Pa = **si** Pf ≠ (0,0,0,0,1) **ou** Pf = (0,0,0,0,1) **et** Pp < (0,0,0,1,0) **alors** Pf + Pp
**sinon si** Pp ≤ (0,0,1,0,0) **alors** Pp + (0,1,0,0,0)
**sinon** Pp + (1,0,0,0,0)

Cette formule donne à l'appel d'une fonction du contexte le poids d'une fonction ayant la même localisation que le plus "lourd" des paramètres. Elle traduit le fait qu'une

fonction du contexte ne pourra faire l'objet d'une évaluation unique si au moins un de ses paramètres n'est pas du contexte.

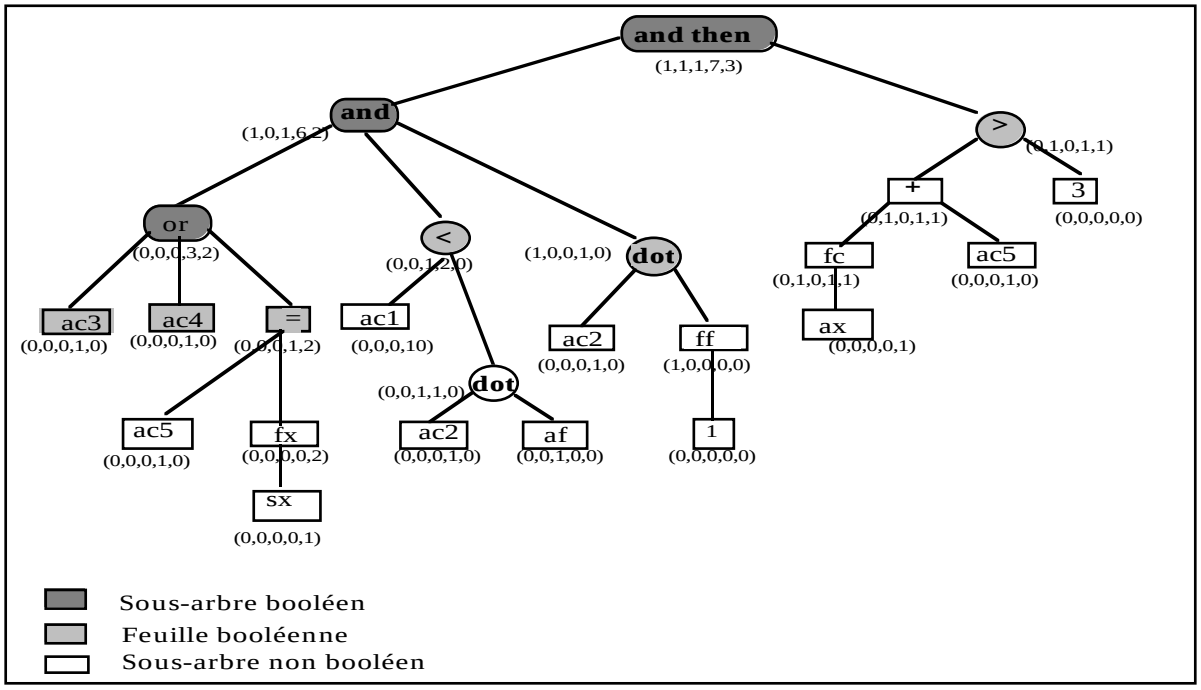


Figure 11.5: Arbre pondéré de l’expression de la fig. 11.2

Les valeurs données aux poids sont justifiées par les considérations suivantes sur les coûts d’une évaluation (fig. 11.5).

L’évaluation des constantes et des littéraux est immédiate. Un littéral ne demande l'accès à aucun objet. Une constante est une variable commune à tous les objets d'une *collection*, et dont la valeur est connue.

L'accès aux primitives du contexte ne nécessite aucune recherche. Les fonctions du contexte ne sont pas différenciées des attributs du contexte dans la définition des poids car, comme nous l'avons vu, leur évaluation est unique si tous les paramètres appartiennent au contexte. Mais si un des paramètres n'appartient pas au contexte, l'évaluation de la fonction doit se faire en mémoire volatile sur chaque objet sélectionné, ce que traduit le calcul de son poids.

Les fonctions et les attributs de la *collection* courante ou d'une *collection* de fournisseur sont distingués pour les raisons suivantes:

• l'appel d'une fonction laisse supposer que plus d'un objet peut être consulté et

même que plusieurs autres fonctions peuvent être appelées; au contraire, dans le cas d'un attribut, un seul objet est concerné;

• l'accès à une fonction nécessite le chargement de l'objet sur lequel elle s'exécute (l'objet courant), voire même d'autres objets. On a donc intérêt à retarder leur évaluation, alors que l'évaluation d'un sous-critère portant sur un attribut peut être réalisé en mémoire persistante par le POM.

• l'évaluation d'une primitive de la *collection* courante est supposée être moins coûteuse que l'évaluation d'une primitive d'une *collection* de fournisseur; en mémoire volatile (fonctions), il y a moins de liens à traverser; en mémoire persistante (attributs), il est supposé que même si le gestionnaire d'objets a recours aux techniques éprouvées des SGBD (par exemple du type opérateur relationnel *join*), l'appel à un opérateur de jointure nécessite l'accès à des objets de plusieurs collections (le nombre est proportionnel au nombre de relations de clientèle à traverser).

On voit que le poids permet de distinguer immédiatement les sous-arbres évaluables en mémoire persistante (Fc et Ff nuls) et ceux qui doivent être évalués en mémoire volatile (Fc ou Ff $\neq 0$).

### 11.3.4. Permutation de l'arbre.

Le poids des sous-arbres détermine l'ordre dans lequel seront évalués ces sous-arbres: si un sous-arbre A a un poids supérieur à un sous-arbre B, alors A sera évalué plus tard que B si la sémantique liée au noeud de leur père (opérateur commutatif ou non) le permet.

Un sous-arbre booléen sera dit permutable si sa racine représente un opérateur commutatif. La permutation de l'arbre consiste à ordonner les fils de tous les sous-arbres permutables selon leurs poids (fig. 11.6).

Un arbre permuté A répond à la définition:

pour tout sous-arbre booléen B de A,

• soit B n'est pas permutable;

• soit B est permutable: si $P_1$, ..., $P_n$ sont les poids de ses n fils, alors $P_{i-1} \leq P_i$ $(1 < i \leq n)$.

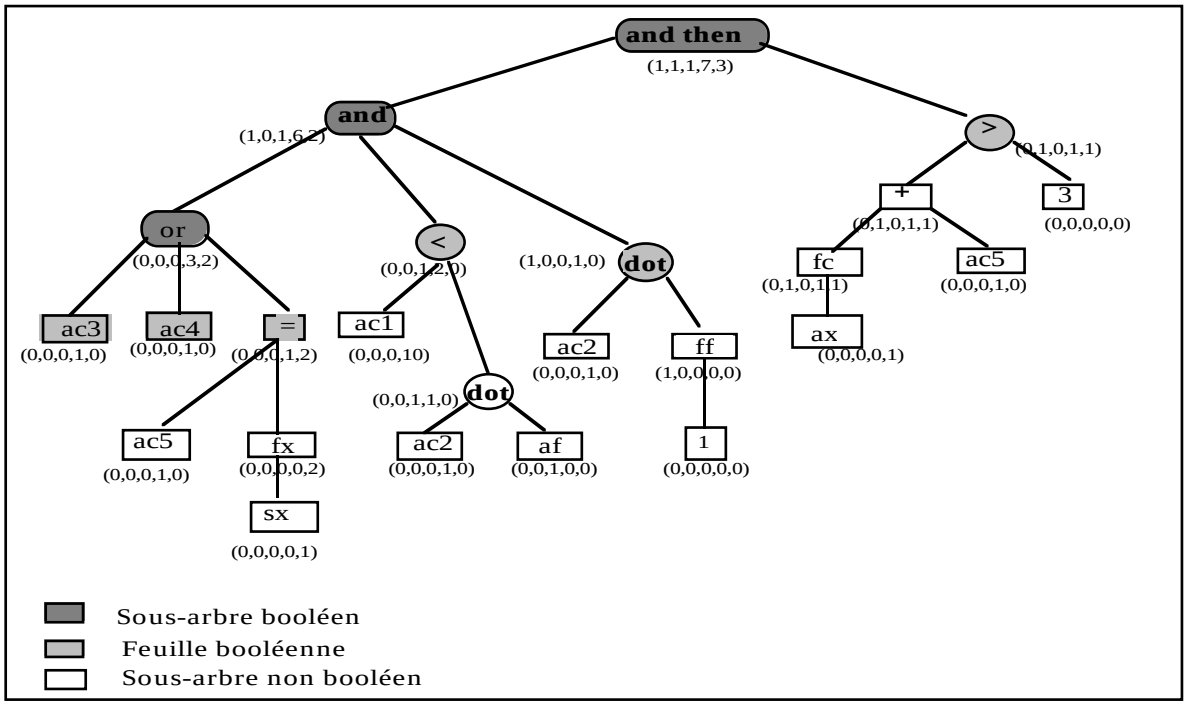


Figure 11.6: Arbre permuté de l'expression de la fig. 11.2

### 11.3.5. Réorganisation de l'arbre.

La réorganisation de l'arbre tend à regrouper les sous-arbres selon leur mode d'évaluation: en mémoire persistante ou en mémoire volatile.

La couleur d'un arbre (en fait de sa racine) sera définie comme suit:

• *verte* s'il peut être évalué complètement en mémoire persistante: tous ses fils sont donc de couleur verte;

• *rouge* s'il doit être évalué exclusivement en mémoire volatile: tous ses fils sont donc de couleur rouge;

• *orange* si une partie peut être évaluée en mémoire persistante et une autre en mémoire volatile (appelée évaluation mixte par la suite): ses fils sont donc multicolores.

La couleur d'une feuille booléenne est déterminée par son poids:

*vert* si Fc et Ff = 0, *rouge* si Fc ou Ff ≠ 0.

On appellera arbre bien partitionné un arbre booléen dont les fils $F_1$, ... $F_i$, ..., $F_n$ sont partitionnés en deux ensembles (l'un d'eux pouvant être vide):

- $F_1$, ..., $F_{i-1}$ verts (c'est à dire "à gauche");
- $F_i$..., $F_n$ rouges (c'est à dire "à droite").

Un arbre bien partitionné n'a donc pas de fils oranges, mais peut lui-même être vert, orange ou rouge.

Un arbre permuté dont aucun fils n'est orange est un arbre bien partitionné par construction.

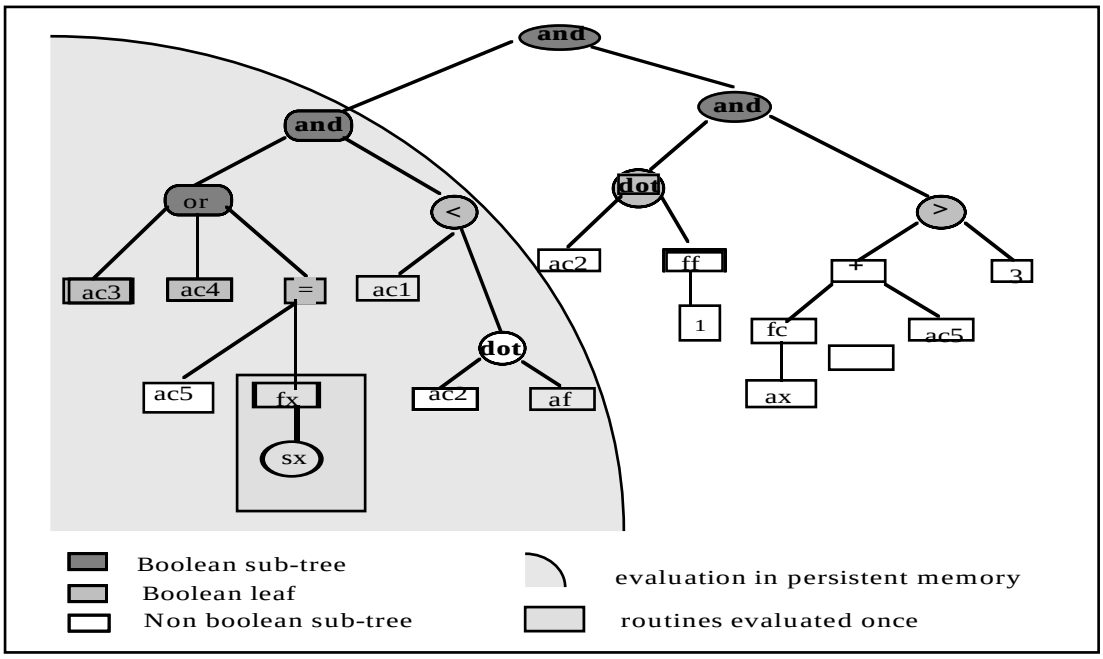


Figure 11. 7: Arbre bien partitionné de l'expression de la fig. 11.2

Un arbre booléen qui n'est pas bien partitionné peut être transformé en arbre bien partitionné sous certaines conditions:

- sa racine représente un opérateur associatif;

- ses fils F1, F2, ... Fn sont tels qu'il existe un fils Fi orange ($1 \leq i \leq n$), que les fils F1, ... Fi-1 sont verts et les fils Fi+1, ... Fn sont rouges (s'il n'y a pas de Fi orange et que les fils sont répartis en vert à gauche et rouge à droite, l'arbre est déja bien partitionné);

• le fils Fi orange est lui-même bien partitionné ou transformable en arbre bien partitionné, et représente le même opérateur que la racine de l'arbre (à la commutativité près: **and** = **and then**, **or** = **or else**; l'arbre ayant déja été permuté, l'ordre des fils est figé et rend cette distinction inutile);

Tout arbre bien partitionné dont la racine représente un opérateur associatif peut être transformé en arbre binaire bien partitionné: l'arbre op (V1, V2, ..., Vn, R1, R2, ..., Rm ), où les Vi sont verts et les Ri rouges, peut se réécrire op (V, R) avec V = op (V1, V2, ...Vn) et R = op (R1, R2, ..., Rm). C'est par exemple le cas de l'arbre de la figure 11.7.

# 11.4. Evaluation des requêtes améliorées

L'évaluation du critère (ou la production du code amélioré) se fait à partir des arbres bien partitionnés. Dans le cas d'un arbre vert ou d'un arbre rouge, il n'y a aucun probleme: soit le critère de sélection est intégralement fourni au POM (cas le plus favorable), soit la sélection est totalement évaluée en mémoire volatile (cas le plus défavorable) par une boucle sur chacun des objets de la *Pcollection*.

L'évaluation d'un arbre orange nécessite des choix plus délicats. Une évaluation mixte d'un arbre implique obligatoirement que sa racine soit évaluée en mémoire volatile. Par contre en fonction du grain de découpage et de la nature des sous-arbres, un plus ou moins grand nombre de ses sous-arbres peut être évalué en mémoire persistante. Dans le cas ou l'arbre n'est pas bien partitionné, il faut faire la connexion (union, intersection) entre les ensembles de POI résultant des évaluations de ses sous-arbres.

## 11.4.1. Evaluation d'un arbre binaire bien partitionné

Soit un arbre binaire bien partitionné op (V,R); soit Pc l'ensemble des objets de la *Pcollection* source de la requête; soit Ep l'ensemble d'objets délivré par le gestionnaire d'objets par l'évaluation de V; soit E'p le complémentaire de Ep (E'p = Pc - Ep = {o ∈ Pc / ¬V (o)). Alors le résultat E de l'évaluation de op (V,R) sera, selon op:

- **and**: E = {o ∈ Ep | R (o)};
- **and then**: comme **and**;
- **or:** E = Ep ∪ {o ∈ E'p | R (o)}
- **or else**: comme **or**;

• **xor**: $E = \{o \in Ep \mid \neg R(o)\} \cup \{o \in E'p \mid R(o)\}$
• **implies**: $E = \{o \in Ep \mid R(o)\} \cup E'p$

Rappelons que l'objectif de l'amélioration est tout particulièrement de limiter le nombre de chargements d'objets persistants en mémoire pour évaluer un critère, chargements supposés coûteux par rapport à l'évaluation proprement dite du critère. La réalisation effective de l'évaluation d'un arbre bien partitionné selon les formules précédentes dépend en grande partie des possibilités du gestionnaire d'objets:

• elles sont réalisables telles quelles pour le **and** et le **and then**, puisqu'il suffit d'appliquer R sur les objets de Ep;

• s'il est possible d'obtenir rapidement le complémentaire d'un ensemble par rapport à une *Pcollection*, ou si le gestionnaire d'objets est suffisamment efficace pour qu'il soit rentable d'effectuer une sélection avec le critère $\neg V$, il est intéressant d'évaluer le **or,** le **or else** et le **implies** selon ces formules;

• on voit que le **xor** nécessite d'appliquer une évaluation en mémoire volatile sur la totalité des objets de la *Pcollection* (une évaluation sur Ep, une autre sur E'p); autant donc évaluer le critère en totalité en mémoire volatile: $R = \{o \in Pc \mid V(o)$ **xor** $R(o)\}$.

Selon les possibilités du gestionnaire d'objet, les couleurs des arbres pourraient être amenées à changer par rapport à la situation actuelle. C'est déja le cas pour un arbre **xor**, même bien partitionné, dont la couleur sera toujours rouge suite à la remarque précédente. Ce peut être le cas pour d'autres arbres si le POM ne fournit pas l'opérateur correspondant. Il suffira pour cela de considérer, lors de la construction de l'arbre, que de tels opérateurs sont des fonctions du contexte pour que le calcul des poids entraîne immédiatement la modification dèsirée des couleurs.

### 11.4.2. Découpage en arbres bien partitionnés.

Un arbre orange qui n'est pas bien partionné peut cependant contenir des sous-arbres bien partitionnés. Le mécanisme d'amélioration tend à évaluer le plus possible de sous-arbres en mémoire persistante, mais cette politique peut aller à l'encontre de l'objectif principal, à savoir l'amélioration. En effet un découpage excessif va entraîner une multiplication des parcours des objets des ensembles. Pour un arbre orange, il y aura autant de boucles faites par le gestionnaire d'objet que de sous-arbres évaluables

complètement en mémoire persistante, et autant de boucles faites au niveau d'Eiffel que de sous-arbres évaluables exclusivement en mémoire volatile.

Prenons par exemple les critères:

a) P1 **and** V1 **or** P2 **and** V2
b) P1 **and** (V1 **or** P2 **and** V2)
c) P1 **and** V1 **and then** P2 **and** V2

où P1 et P2 sont évaluables en mémoire persistante et V1 et V2 en mémoire volatile; ils peuvent s'évaluer en 2 parcours en mémoire persistante et 2 parcours en mémoire volatile, accompagnés d'unions ou d'intersections des ensembles résultats. Si cela semble être une possibilité raisonnable pour a), il est probablement plus efficace pour b) de se contenter d'un parcours en mémoire persistante pour P1, et d'évaluer le reste du critère sur l'ensemble retourné par le gestionnaire d'objets. Dans le cas c), tout dépend des possibilités du gestionnaire d'objets: s'il offre la possiblité de soumettre une requête sur un ensemble d'objets fourni par l'évaluateur du langage de requêtes (ici le résultat de P1 **and** V1), il est intéressant de lui laisser évaluer P2 sur ce résultat; sinon, il est préférable d'adopter la même stratégie que pour b). Le respect de la semantique d'Eiffel interdit de procéder comme pour a), car l'opérateur **and then** impose que P2 **and** V2 soit évalué uniquement sur les objets vérifiant P1 **and** V1.

Le découpage de l'arbre en sous-arbres bien partitionnés va donc dépendre du gain de temps qu'il procure, c'est à dire si la grandeur de l'ensemble d'objets sur lequel vont s'effectuer les boucles est diminué, si le temps de chargement est grand par rapport à l'accès à un objet en mémoire volatile et si l'union ou l'intersection entre deux ensembles résultats d'évaluations n'est pas trop coûteuse.

Soit $T_{ch}$ le temps de chargement (incluant la conversion), d'un objet en mémoire centrale, soit $T_e$ le temps d'évaluation du critère sur un objet en mémoire persistante, soit $Np$ le nombre de boucles en mémoire persistante, soit $Nv$ le nombre de boucles en mémoire volatile, soit $S$ une statistique sur la proportion moyenne d'objets sélectionnés par une évaluation en mémoire persistante ($0 \leq S \leq 1$); si on peut écrire:

$$Tch * Nv * S + Te * Np < Tch$$

alors l'évaluation d'un sous-arbre orange pourra être effectuée en $Np$ évaluations en mémoire persistante et $Nv$ évaluations en mémoire volatile. Sinon il est préférable de

réaliser toute l'évaluation en mémoire volatile à partir du premier sous-arbre non évaluable en mémoire persistante.

### 11.4.3. Evaluation unique des fonctions du contexte.

Selon la nature des sous-arbres on peut constater des différences sur la date et la nécessité de cette évaluation. Dans un sous-arbre évalué en mémoire persistante, toutes les évaluations uniques doivent être effectuées avant que le critère correspondant sous-arbre soit soumis au gestionnaire d'objets. Par contre dans le cas ou l'évaluation du sous-arbre se fait en mémoire volatile, il faut tenir compte des possibles coupures de l'évaluation (*court-circuit*); la fonction ne doit être évaluée que lorsque le sous-arbre qui la contient doit être évalué; son comportement est alors celui d'une fonction *once*.

## 11.5. Etude des performances

### 11.5.1. Conditions de l'expérience:

**Caractéristiques matérielles et logicielles:**

- machine: DEC2100 sur Ultrix v2.1,
- logiciel, Base de données: Oracle v6.0,

**Langages et techniques utilisées:**

Nous avons utilisé le langage C, et les analyseurs Lex et Yacc pour réaliser l'interface avec Oracle, l'analyse des expressions et l'évaluation volatile des arbres sur des objets. Le langage Eiffel nous a offert des outils pour récupérer les informations nécessaires sur les classes associées aux *collections* manipulées par une requête, et pour le pilotage des tests.

On notera que notre effort n'a pas porté sur l'amélioration de l'implantation de la conversion entre un tuple et un objet Eiffel; notre objectif était plutôt de valider les idées.

**Structure et contenu de la base:**

La base de données contient cent objets de type *PERSON* faisant référence à des objets de type *ADDRESS* et par transitivité, *COUNTRY*, (fig. 11.8).

```
class PERSON                class ADDRESS              class COUNTRY
feature                     feature                    feature
   pname,first_name:STRING;     street, town: STRING;      pname, continent: STRING;
   age: INTEGER;             number, zip_code: INTEGER;    code: INTEGER;
   husband_wife: PERSON;        country: COUNTRY;       end -- class COUNTRY
   address: ADDRESS          end -- class ADDRESS
   fage (): INTEGER;
   fname (): STRING;
end -- class PERSON
```

Figure 11.8: Description de l'exemple utilisé pour l'étude des performances

Note: La classe *STRING* est une classe de la bibliothèque Eiffel contenant notamment les fonctions: *index_of, equal*

## 11.5.2. Description des exemples:

Dans les expressions Eiffel, "*c*" référence les instances de la *collection* associée à la classe *PERSON*, "*q*" référence une caractéristique de la requête, modélisée par une instance d'une classe descendant de QUERY.

Les différentes requêtes sont:

- R1: Requête totalement persistante:
  . sélectionner les personnes qui ont pour âge *b_age* - 5.
  . critère en Eiffel: *c.age = q.b_age - 5*

- R2: Requête totalement persistante:
  . sélectionner les personnes qui ont un âge supérieur à *b_age* et dont le pays d'habitation a pour code 9, ou bien celles dont le numéro de l'adresse est 5.
  . critère en Eiffel: *c.age >= q.b_age and c.address.country.code = 9 or c.address.number= 5*

- R3: Requête totalement volatile: uniquement des fonctions:
  . sélectionner les personnes dont le nom contient un 'b' ou dont la valeur de la fonction *fage* est *b_age*.

. critère Eiffel: *c.pname.index_of ('b', 1) > 0 or c.fage = q.b_age*

• R4: Requête "*and*" mi-persistante, mi-volatile; la partie persistante est très restrictive[1]

. sélectionner les personnes qui ont un âge égal à *b_age* et qui, soit se prénomment "gustave" soit habitent au numéro 10.

. critère Eiffel:*c.age=q.b_age and (c.first_name.equal("gustave") or c.addrese.number= 10)*

• R5: idem que précédemment, mais à partie persistante moins restrictive:

. sélectionner les personnes dont l'âge est supérieur à *b_age* et qui, soit se prénomment "gustave" soit habitent au numéro 10.

. critère Eiffel: *c.age > q.b_age and (c.first_name.equal ("gustave") or c.address.numéro = 10)*

• R6: Requête "*or*" mi-persistante, mi-volatile

. sélectionner les personnes qui ont un âge supérieur à *b_age* ou qui habitent en France.

. critère Eiffel: *c.age > q.b_age or c.address.country.pname.equal ("France")*

### 11.5.3. Les résultats:

R1, R2:

Les deux requêtes peuvent être traîtées intégralement par le POM. L'analyse étant faite, il suffit d'envoyer la requête et de récupérer les objets sélectionnés. Le gain provient du fait qu'il n'y a aucun chargement inutile d'objet, et qu'une fois chargés en mémoire, aucune évaluation volatile n'est nécessaire. Le temps total se décompose comme suit:

analyse et envoi de la requête: 14 sec. 382 msec.

chargement des objets sélectionnés: 13 sec. 786 msec.

Il va de soi que, plus le critère de sélection d'une requête totalement persistante provoquera la sélection d'un nombre important d'objets, moins l'amélioration apparaîtra.

---

[1] restrictive signifie que la requête provoque la sélection de plus ou moins de personnes.

R3:

Cette requête ne contient que des références à des fonctions et de ce fait doit être totalement évaluée en mémoire volatile. Un chargement de la totalité des objets est nécessaire afin d'évaluer sur chacun d'eux l'expression du critère.

Le seul gain (minime) pouvant apparaître, se trouve dans la réorganisation de l'expression et dans l'évaluation en court-circuit. Les mesures de temps effectuées montrent qu'aucun gain réel n'est mis en évidence: il y a équivalence entre les évaluations avec ou sans amélioration.

R4, R5:

Ces requêtes (s'articulant sur un connecteur logique *and*), sont évaluées en partie en mémoire persistante et en partie en mémoire volatile. Ces requêtes sont celles pour lesquelles l'amélioration peut paraître la plus intéressante; le gain de temps est d'autant plus important que la partie persistante est restrictive. L'évaluation se découpent en deux parties:

- la première consiste en la sélection par le POM d'un sous-ensemble de candidats susceptibles de satisfaire le critère de sélection,
- la seconde consiste en l'évaluation sur ces objets (alors chargés) de la partie volatile de la requête.

R6:

Ces requêtes (s'articulant sur un connecteur logique *or)*, sont évaluées en partie en mémoire persistante et en partie en mémoire volatile. Les requêtes de ce type nécessitent un chargement de la totalité des objets de la base, mais en deux partie:

- la première rendue par le gestionnaire d'objets dont on sait qu'ils font déjà partie de la sélection, et
- une seconde (le complémentaire de la précédente) sur laquelle la partie volatile de la requête va être évaluée.

Le fait que le gain soit peu important provient du fait que l'évaluation en court-circuit des fonctions de la requête, ne porte que sur une partie des objets de la base, celle où l'évaluation est de toute façon indispensable.

### 11.5.4. Synthèse:

Les temps sont en min:sec.ms. Le gain se calcule par rapport à la requête sans amélioration.

| requêtes | sans amélioration | avec amélioration | gain en % de temps |
|---|---|---|---|
| R1 | 02:23.419 | 00:31.371 | 78.13 |
| R2 | 02:23.010 | 00:16.411 | 88.52 |
| R3 | 02:23.164 | 02:23.220 | 0.00 |
| R4 | 02:22.610 | 00:16.016 | 88.77 |
| R5 | 02:22.089 | 01:10.040 | 50.71 |
| R6 | 02:25.347 | 02:20.773 | 3.15 |

### 11.5.5. Conclusion.

Le mécanisme d'amélioration des critères de sélection décrit ci-dessus peut s'effectuer soit à la compilation, soit à l'exécution d'une requête de sélection Il met en oeuvre des mécanismes classiques de compilation (arbres abstraits décorés) et ne pose pas de problèmes majeurs de réalisation. Dans la mesure où les requêtes de sélection sont des textes de petite taille (comparée par exemple à celle du texte d'une classe Eiffel), ce mécanisme n'est pas coûteux.

Les bénéfices qu'on peut en espérer sont bien sûr dépendants du critère de sélection. Toutefois, il est fort probable que la majeure partie de ces critères ne contiennent surtout que des comparaisons sur des attributs, et relativement peu de fonctions. Il est aussi assez probable que les opérateurs utilisés permettent de réorganiser ces critères afin d'obtenir des arbres binaires bien partitionnés, cas où le mécanisme s'avère efficace.

# 12

# Mise en oeuvre d'un POM à partir du SGBD O2

Pour valider notre approche nous construisons un prototype basé sur un gestionnaire d'objets construit à partir du SGBD $O_2$. La version d'Eiffel utilisée est la version 2.3 (la version 3 ne sera disponible que prochaînement). Notre objectif est d'une part de valider les concepts présentés dans le chapitre 3, et d'autre part, de parvenir à implémenter un gestionnaire d'objets persistants (POM), qui ait des performances raisonnables pour des application industrielles.

Les principaux aspects de l'implémentation du POM liés au langage Eiffel sont:

- la traduction des classes en $O_2$,
- la conversion des objets lors des échanges,
- les opérations pour la sélection d'objets,
- la gestion des protections et le mécanisme de transaction

Tous ces aspects vont être développé en précisant, quand c'est nécessaire, le détail des deux approches statique et dynamique.

## 12.1. Traduction des classes en $O_2$

La traduction des classes est nécessaire pour réaliser des traitements en mémoire persistante (en particulier des sélections); nous distinguonsw les fonctionnalités d'un compilateur qu'il n'est pas nécessaire d'intégrer dans le traducteur Eiffel-$O_2$, de celles qui restent indispensables.

Dans un premier temps nous avions envisagé de développer un traducteur qui permette d'exploiter les classes $O_2$, à la fois à partir d'Eiffel et d'$O_2$, en garantissant les règles de lisibilité (exportation), et la sémantique des classes.

Cette idée était séduisante puisqu'elle offrait des possibilités d'ouvertures supplémentaires vers l'extérieur; mais nous avons dû l'abandonner, essentiellement à cause des problèmes de compatibilité:

- absence de classes génériques en $O_2$,
- différences sensibles dans le traitement de l'héritage répété entre Eiffel et $O_2$,
- sémantique des données publiques / privées de $O_2$, différente de celle de routine exportée en Eiffel.

### 12.1.1. Principaux aspects

La traduction dynamique des classes est réalisée à partir des informations présentes dans l'exécutif Eiffel, et disponibles à partir des classes INTERNAL et INTERNAL2 fournies par Eiffel.

La traduction statique des classes est réalisée à partir d'une version modifiée de *es* (eiffel system) qui est l'outil gérant la compilation d'un système de classes; cette version réutilise les 3 premières passes du compilateur Eiffel, et remplace la passe 4 de génération du code C par une nouvelle qui produit du code *O2*.

#### Contrôles à la charge d'Eiffel

Lors de la traduction des classes Eiffel vers $O_2$, les mécanismes d'exportation d'Eiffel ne peuvent pas être totalement exprimés en $O_2$: exportation limitée à certaines classes, exportation dans les descendants par exemple. Cela a peu d'importance, car l'exportation n'a pour objet que de contrôler des règles de programmation, ce qui reste fait par le compilateur Eiffel. Vis-à-vis de $O_2$, considéré ici comme un langage de plus bas niveau, toutes les primitives seront exportées. Toutefois, si les classes $O_2$ produites par la traduction sont utilisées par d'autres applications sous le contrôle de $O_2$, les protections exercées par l'exportation ne s'exerceront plus, ce qui peut être gênant compte tenu de l'aspect interprété.

Dans l'approche dynamique, on propage tous les attributs, exportés ou non, mais pas les routines. La gestion de l'exportation, outre les difficultés de compatibilité signalées ci-dessus, aurait de plus des inconvénients sur les temps d'exécution. Dans l'approche statique, on propage toutes les caractéristiques, routines et attributs, exportés ou non.

Les contrôles de type ne sont pas réalisés dans l'approche dynamique, mais le sont par, les passes 2 et 3 du compilateur Eiffel dans l'approche statique.

#### Héritage multiple et répété

La traduction de l'héritage est certainement la partie la plus délicate de la traduction car si $O_2$et Eiffel offrent des possibilités de redéfinition et de renommage, les règles sont différentes; cette incompatibilité sera encore augmentée pour la version 3 avec l'apparition de la clause *select* [Meyer 91]; elle provient essentiellement des points suivants:

- redéfinition implicite des routines,
- renommage implicite des primitives en cas de conflit,
- renommage par rapport à la classe où est fait le renommage et non pas par rapport à la clause d'héritage.

| | renommage / redéfinition | | |
|---|---|---|---|
| Cas | pour l'héritage de la classe B dans C | old_name | new_name |
| 1 | Pas de renommage,<br>pas de redéfinition | Version de B | [invalide] |
| 2 | **rename** old_name as new_name | [invalide] | Version de B |
| 3 | **redefine** old_name | Version de C | [invalide] |
| 4 | **rename** old_name as new_name<br>**redefine** old_name | Version de C | Version de B |
| 5 | **rename** old_name as new_name<br>**redefine** new_name | [invalide] | Version de C |
| 6 | **rename** old_name as new_name<br>**redefine** old_name, new_name | Version de C | Version de C |

Figure 14.1: Les différents cas d'héritage en Eiffel

Le seul moyen de s'assurer que la sémantique de l'héritage d'Eiffel est bien traduite dans $O_2$ est de vérifier les six combinaisons autorisées par Eiffel dans une clause d'héritage (voir fig. 14.1). Pour des héritages multiples classiques, mais aussi répétés. En cas d'héritage répété, la traduction par simple transcription de certains cas pose des problèmes car elle produit des résultats non conformes à la sémantique d'Eiffel.

Pour résoudre ce problème de manière générale, il faudra introduire à la traduction, dans le cas de renommage, des classes intermédiaires dans $O_2$, avec de nouvelles routines permettant la prise en compte du polymorphisme.

Supposons que la classe B soit de la forme suivante:

```
class B  feature
        att1: T1; att2: T2; att3: T3; att4: T4; att5: T5; att6: T6;
        func1: T1; func2: T2; func3: T3; func4: T4; func5: T5; func6: T6
end -- class B
```

On peut décrire les 6 cas de **renommage / définition** pour une classe *C* qui hérite de *B*. Dans le cas 1, la traduction ne présente aucune difficulté (simple transcription en $O_2$);

dans le cas 2, la présence d'un renommage oblige à l'introduction d'une classe intermédiaire O2_TO_C, afin de contourner le renommage implicite en cas d'héritage répété.

| | | |
|---|---|---|
| *class* C -- Eiffel cas 2<br>*inherit* B<br>*rename* old_att2 as new_att2,<br>old_func2 as new_func2 | *class* O2_TO_C -- O2<br>*inherit* B<br>*rename* old_att2 as new_att2<br>old_func2 as new_func2 | *class* C -- O2<br>*inherit* O2_TO_C |

Dans le cas 3, il n'y a pas besoin de classe intermédiaire, et la redéfinition étant implicite en $O_2$, aucun traitement particulier n'est nécessaire, à part définir une nouvelle version de la routine:

| | | |
|---|---|---|
| *class* C -- Eiffel cas 3<br>*inherit* B<br>*redefine* att3, func3<br>feature<br>att3: FILS_T3;<br>func3: FILS_T3<br>end -- class C | Pas de classe intermédiaire | *class* C -- O2<br>*inherit* B<br>*type* tuple (att3: FILS_T3)<br>*method* func3: FILS_T3<br>*end* |

Le cas 4 est un cas complexe, mais souvent utilisé; il combine renommage et redéfinition, sa traduction reflète celle des cas 2 et 3: pas de traitement spécial pour la redéfinition et une classe intermédiaire pour éviter le renommage et le partage implicite des routines en cas d'héritage répété.

| | | |
|---|---|---|
| *class* C -- Eiffel cas 4<br>*inherit* B<br>*rename* old_att4 as new_att4,<br>old_func4 as new_func4<br>*redefine* old_att4, old_func4<br>*feature*<br>old_att4: FILS_T4;<br>old_func4: FILS_T4<br>*end* -- class C | *class* O2_TO_C -- O2<br>*inherit* B<br>*rename* old_att4 as new_att4,<br>old_func4 as new_func4<br>*type* tuple (new_att4: FILS_T4)<br>*method* new_function4: FILS_T4<br>*end*; | *class* C -- O2<br>*inherit* O2_TO_C<br>*rename* new_att4 as old_att4,<br>new_func4 as old_func4<br>*type* tuple (new_att4: T4)<br>*method* new_func4: T4<br>*end;* |

Les problèmes de traduction des cas 5 et 6 sont résolus avec la même technique.

| | | |
|---|---|---|
| *class* C -- Eiffel cas 5<br>*inherit* B<br>*rename* old_att5 as new_att5,<br>old_func5 as new_func5<br>*redefine* new_att5, new_func5<br>*feature*<br>new_att5: FILS_T5;<br>new_func5: FILS_T5<br>*end* -- class C | *class* O2_TO_C -- O2<br>*inherit* B<br>*rename* old_att5 as new_att5,<br>old_func5 as new_func5<br>*end*; | *class* C -- O2<br>*inherit* O2_TO_C<br>*type* tuple (new_att5: FILS_T5)<br>*method* new_func5: FILS_T5<br>*end;* |

| | | |
|---|---|---|
| *class* C -- Eiffel cas 6<br>*inherit* B<br>*rename* old_att6 as new_att6,<br>old_func6as new_func6<br>*redefine* old_att6, old_func6,<br>new_att6, new_func6<br>*feature*<br>old_att6: FILS_T6;<br>old_func6: FILS_T6;<br>new_att6: FILS_T6;<br>new_func6: FILS_T6;<br>*end* -- class C | *class* O2_TO_C -- O2<br>*inherit* B<br>*rename* old_att6 as new_att6,<br>old_func6 as new_func6<br>*type* tuple (new_att4: FILS_T6)<br>*method* new_function4: FILS_T6<br>*end*; | *class* C -- O2<br>*inherit* O2_TO_C<br>*rename* new_att6 as old_att6,<br>new_func6 as old_func6<br>*type* tuple (new_att6: T6)<br>*method* new_func6: T6<br>*end;* |

Remarque

Selon l'approche, statique ou dynamique, la traduction est plus ou moins complète. L'approche statique se doit de résoudre correctement ces problèmes, par contre comme l'objectif dans l'approche dynamique, n'est pas de réaliser des traitements en mémoire persistante, il semble tout à fait acceptable de traduire une hiérarchie applatie, afin de ne pas altérer exagérément les performances.

Par ailleurs de récents changements dans *O2* concernant le renommage (la clause de renommage fait référence à la classe d'où l'objet est originaire), ont permis d'envisager une traduction simplifiée.

## Classes génériques

La traduction des classes génériques est aussi un cas qui pose problème; dans *O2* il n'y a pas le concept de *classe générique*. La première solution qui a été envisagée est de laisser Eiffel garantir la cohérence des types, et se contenter de crééer une classe correspondant à la classe générique dont le type des paramètres serait contraint à être de type *OBJECT* (équivalent à la classe *ANY* en *O2*). Cette solution n'est cependant pas envisageable car les types simples (char, real, integer, boolean, string, bits), ne sont pas des classes en *O2* et donc n'héritent pas de *OBJECT*.

Une seconde approche est de crééer des classes (*CHARACTER, REAL, INTEGER, BOOLEAN, STRING, BITS*) munies d'un seul attribut du type correspondant (ex: la classe INTEGER aura un attribut *value* de type *integer*). Cette traduction ne serait utilisée que dans le cas des classes génériques. De toute manière la lourdeur de sa mise en oeuvre (elle nécessiterait la modification des instructions utilisant les classes *INTEGER*, etc), ne pourrait être envisageable que dans l'approche statique. Cependant, cette solution a deux inconvénients majeur:

- on ne pourrait pas respecter la définition des Pcollection relative au fait qu'une Pcollection est associé à un type et non à une classe (cf. § 3.4.1),

- et surtout la solution ne pourrait être générale et s'appliquer à tous les types expansés.

Pour ces deux raisons, nous rejetons la deuxième solution au profit de la solution suivante. Nous utilisons le processeur C et Lex pour générer à partir d'une classe générique Eiffel, autant de classe Eiffel (non générique), qu'il y a d'instanciations réalisées. La génération de ces classes est transparente à l'utilisateur et n'est utilisé que pour générer les classes $O_2$ correspondantes. Cette solution permet de prendre en compte les deux aspects cités ci-dessus.

Cette solution n'est pas adaptée à l'approche dynamique (comme la précédente dailleurs); on n'implémentera donc que certaines classes génériques indispensebles de la bibliothèque (*ARRAY,* LINKED_LIST,*COLLECTION*, etc), et cette traduction ne sera pas automatisée.

### Classes implémentées en C

Comme cela a déjà été abordé dans le paragraphe 10.11, les routines externes seront traitées différemment selon qu'elles nécessitent le chargement d'objets Eiffel, ou qu'elle font référence à la structure d'objets Eiffel.

Dans le cas où la routine ne fait référence qu'à des concepts extérieurs (fichiers unix, outils graphiques, etc), il n'y a aucune traduction à faire, il suffit d'incorporer les fichiers C dans $O_2$, comme c'est actuellement réalisé dans Eiffel.

Par contre dans les autres cas, la seule solution est une traduction manuelle de ces classes. Heureusement, celles-ci ne sont pas nombreuses dans la bibliothèque Eiffel; il s'agît essentiellement des classes *ARRAY, STRING, INTERNAL, ANY*.

Dans l'approche dynamique (routines non propagées), on pourrait aussi envisager

que, chaque fois qu'une classe contient un attribut de type *TO_SPECIAL*, celui-ci est remplacé par un de même nom mais de type *list*. Cependant, même si cela permet d'automatiser la traduction, cette solution est inefficace.

**Traduction des routines**

Une fois que les problèmes liés au polymorphisme sont résolus, la traduction ne pose plus vraiment de problèmes, les deux langages ($O_2$-C et Eiffel), génèrent du code C. Dans les cas complexes, on pourra reprendre les solutions Eiffel utilisées pour la génération de code (*implies*, *once*, etc).

**Cas des constantes**

Lorsqu'une constante est détecté, celle-ci sera traduite dans $O_2$ par un objet nommé constant.

# 12.2. Conversion des objets

On a vu plus haut, que lorsque les modèles du gestionnaire et du langage ne sont pas identiques il faut prévoir un mécanisme de conversion (cf. § 10.2).

## 12.2.1. Implémentation de l'approche dynamique

Dans l'approche dynamique, la conversion est faite à partir d'une version plus complète de la classe *INTERNAL*.

Avant de convertir un objet, le système se préoccupe de vérifier que la classe est déjà propagée dans la mémoire persistante; si ce n'est pas le cas, la classe Eiffel sera alors traduite, à partir des renseignements de l'exécutif et la classe $O_2$ sera générée. Cette traduction détecte et prend en compte le cas ou une classe est cliente d'elle même.

Dès que la classe est présente en $O_2$, la conversion de l'objet peut commencer, elle est réalisée à partir d'un parcours dynamique des attributs de la classe. Le cas des attributs de type *void* est aussi traité

### 12.2.2. Implémentation de l'approche statique

L'approche statique permet d'optimiser la conversion. A l'exécution, on garantit que la classe est présente.

Si le traducteur Eiffel-$O_2$ et indépendant du compilateur Eiffel, cette garantie est obtenue par une vérification au lancement de l'exécutif. Si le traducteur Eiffel-$O_2$ est intégré dans le compilateur Eiffel, cette garantie est fournie par la commande *es* (cf. § 9.3).

Lors de la traduction de Eiffel vers $O_2$, deux routines de conversions (Eiffel_to_O2, O2_to_Eiffel), sont générées; elles sont intégrées dans l'exécutif Eiffel au lancement du processus. Ces routines de conversions réalisent directement les affectations, sans effectuer le parcours des attributs de l'objet: nous attendons un gain appréciable.

## 12.3. Traitement des requêtes sélectives

### 12.3.1. Implémentation de l'approche dynamique

Dans l'approche dynamique, le traitement des requêtes sélectives est délicat, car il suppose la conversion d'une routine Eiffel en expression $O_2$, et donc l'accès au source de la classe Eiffel.

L'exécution des routines sélectives pourra utiliser le concept de prédicat dans *$O_2$-engine* [$O_2$-Technology 91b], cela risque d'être difficile compte tenu par exemple qu'on ne peut modéliser l'opérateur *or*.

Une autre implémentation est d'associer les requêtes sélectives à des routines de *$O_2$-C* insérées manuellement dans *$O_2$*. ces routines devront accepter en paramètre une chaîne de caractères repésentant le critère de sélection en *$O_2$*; elles utiliseront les services de la routine *o2query*.

Dans les deux cas, on notera qu'il est nécessaire d'analyser et de convertir l'expression Eiffel qui peut être complexe,, soit en appels aux primitives de *$O_2$-Engine* (1ère approche), soit en texte *$O_2$-C* (2ème approche). Cependant, ces deux approches ne fonctionnent pas actuellement dans *$O_2$* pour les raisons suivantes:

- le concept de prédicat n'est pas encore implémenté (seule l'interface existe),
- la liaison dynamique entre $O_2$ et C n'est pas encore implémenté.

Il nous est donc impossible d'implémenter une approche dynamique pour le traitement des requêtes sélectives.

### 12.3.2. Implémentation de l'approche statique

Comme les descendants de la classe *COLLECTION* seront implantés à partir d'un code en C, ils sont considérés au niveau de la traduction comme des classes particulières. Ils pourront être traduits soit par des séquences d'appels à des routines de *O2-Engine*, soit par des routines de *O2-C*.

Nous traitons ici la traduction dans le langage *O2-C*; on rappelle la structure *O2-C* qui permet de faire des itérations sur une collection d'objets:

```
for <variable-libre> in <collection> where <condition> {<instructions>}
```

Présentons la traduction des quantificateurs et des principales fonctions et itérations sélectives. D'une manière générale, chaque fois que l'on rencontrera un appel à une fonction ou itération sélective, on générera la structure *for-in-where* citée ci-dessus. Ainsi, l'expression:

```
c.fonction_selective ($criteria)    -- c est une collection
                                    -- fonction_selective = select, exists, all, do_if, ....
                                    -- signature de criteria: criteria (x: class_name)
```

sera traduite par:

```
for x in c where <partie gauche de Result de criteria> {<instructions>}
```

**Primitive** exists

L'expression suivante utilisant la primitive *exists*:

```
c: O2_SET [CLASS_NAME];          -- cas ou la collection est un ensemble O2
...   := c.exists ($criteria)
                                 -- signature de criteria: criteria (x: class_name)
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;                                  -- cas ou la collection est un ensemble
for (x in c where <partie gauche de Result de criteria> )
   {Result:= True}                                      -- Result ↔ nom correspondant au
code
                                                        -- généré par Eiffel
```

**Primitive** all

L'expression suivante utilisant la primitive *all*:

```
c: O2_SET [CLASS_NAME];                -- cas ou la collection est un ensemble O2
...   := c.all ($criteria)
                                       -- signature de criteria: criteria (x: class_name)
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;                 -- cas ou la collection est un ensemble
for (x in c where not (<partie gauche de Result de criteria>) )
   {Result:= False}                    -- Result ↔ nom correspondant au code
                                       -- généré par Eiffel
```

**Primitive** select

L'expression suivante utilisant la primitive *select*:

```
c: O2_SET [CLASS_NAME];                -- cas ou la collection est un ensemble O2
... := c.select ($criteria)            -- signature de criteria: criteria (x: class_name)
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;                 -- cas ou la collection est un ensemble
o2 set (class_name) res;               -- variable intermédiaire C

-- Creation d'une instance de Result
-- affectation de res au poi de Result
for (x in c where <partie gauche de Result de criteria> )
   { res += x;}                        -- Result ↔ nom correspondant au code
                                       -- généré par Eiffel
```

**Primitive** count

L'expression suivante utilisant la primitive *count:*

```
c: O2_SET [CLASS_NAME];                -- cas ou la collection est un ensemble O2
... := c.count ($criteria)             -- signature de criteria: criteria (x: class_name)
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;              -- cas ou la collection est un ensemble
for (x in c where <partie gauche de Result de criteria> )
   { Result = Result + 1;}          -- Result ↔ nom correspondant au code généré par Eiffel
```

**Primitive** do_if

L'expression suivante utilisant la primitive *do_if:*

```
c: O2_SET [CLASS_NAME];             -- cas ou la collection est un ensemble O2
... := c.do_if ($criteria, $action) -- signature de criteria: criteria (x: class_name)
                                    -- signature de action: action (x: class_name)
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;              -- cas ou la collection est un ensemble
for (x in c where <partie gauche de Result de criteria>)
   { o2_action (x)}                 -- o2_action est le code O2 de la routine action
```

Dans le cas ou *do_if* est remplacé par *do_all,* on obtiendra en *O2-C*:

```
o2 set (class_name) c;              -- cas ou la collection est un ensemble
for (x in c )   { o2_action (x)}    -- o2_action est le code O2 de la routine action
```

Remarque

Dans l'approche statique, on pourra implémenter les requêtes sélectives à partir d'objets de type *COLLECTION* qui sont volatiles. Il suffira, pour chacune des primitives décrites ci-dessus, d'écrire un schéma itératif en C qui appelle pour chaque élément de la collection le critère de sélection qui aura été traduit en *O2*.

## Emboîtement de Requête

Dans le cas ou on a des requêtes sélectives emboitées, on emboîte ra de la même manière les structures for-in-where de *O2-C*. Supposons que l'on ait à traiter l'exemple Eiffel suivant:

```
c: O2_SET [CLASS_NAME];          -- cas ou la collection est un ensemble O2
... := c.select ($criteria1, c);

criteria1 (x: CLASS_NAME, c: O2_SET [CLASS_NAME]): BOOLEAN is
  do     -- signature de criteria2: criteria2 (x: CLASS_NAMR)
     Result:= x.attribut1 = x.attribut2 and c.count ($criteria2) = 1 and x.attribut3
  end -- criteria1
```

sera traduite en *O2-C* par:

```
o2 set (class_name) c;                          -- cas ou la collection est un ensemble
o2 set (class_name) res;                        -- variable intermédiaire C

-- Creation d'une instance de Result , et affectation de res au poi de Result
for (x in c where x.attribut1 = x.attribut2)
    {   (for (y in c where <partie gauche de Result de criteria2>)
            {Result1 = Result1 + 1};
        if (Result1 = 1 and x.attribut3) res += x;
    }                   -- Result et Result1↔ nom correspondant au code généré par Eiffel
```

**Requête sélective nécessitant un contexte**

Si le contexte de l'expression sélective est constitué uniquement de valeurs simples (INTEGER, BOOLEAN, CHARACTER, REAL), de chaînes de caractères (STRING) ou de références à des objets persistants, cela ne pose pas de problèmes de traduction.

Dans le cas où un objet volatile entre dans la description de l'expression, alors il faut convertir l'objet Eiffel en Objet *O2* avant de le transmettre au POM. Si la sélection modifie le contexte, se pose alors le problème de la mise à jour de sa copie en mémoire volatile.

# 12.4. Implantation de la classe*EIFFEL_CLASS*

Dans *O2*, une classe n'est pas un objet; mais un ensemble de routines offert par la couche basse de *O2* (*O2-engine*), permet de modifier les classes et de consulter les informations associées.

Pour limiter la redondance avec les informations gérées par *O2*, toute instance de la classe *EIFFEL_CLASS* ne contientquet les informations non stockées dans les structures gérées par O2 (concepts Eiffel n'ayant pas d'équivalent ou non traduits en O2). La classe *EIFFEL_CLASS* offrira simplement des primitives pour accéder aux autres informations.

L'information minimale à stocker dans *EIFFEL_CLASS* est le nom qui permet d'identifier la classe; c'est à partir de ce nom, que les primitives de *EIFFEL_CLASS* fourniront l'accès aux informations gérées par *O2*. Typiquement on laissera à *O2* la mémorisation des informations suivantes: attributs (nom, type, ....), routines (signature,

corps?, ...), droits d'accès, les liens d'héritage (avec l'expression du renommage et de la redéfinition)

Des primitives permettront leur modification et leur consultation. Par contre on pourra laisser à l'exécutif Eiffel la gestion d'autres informations de la classe *EIFFEL_CLASS*, comme par exemple:

- les critères qui se trouvent dans la clause **indexing** (base de la recherche de composants),

- les dates de modification et de création d'une classe.

La propagation d'une classe dans le monde persistant se fait dès la création d'une instance de la classe *EIFFEL_CLASS,* s'il n'y a aucune classe de même nom. La description de la classe *EIFFEL_CLASS* est donnée dans la partie II.

# 12.5. Autres problèmes d'intégration

## 12.5.1. Identification des objets

L'identification des objets est réalisée à travers le mécanisme des *poignées* (*handle*) de *O2*: on ne voit pas véritablement l'identificateur d'objet persistant, mais un descripteur en mémoire volatile qui permet de faire la correspondance entre l'adresse en mémoire volatile et l'identificateur persistant de l'objet *O2*.

Le fait que l'on ait deux mécanismes pour gérer la même chose, l'un au niveau de *O2* (*handle*) et l'autre au niveau d'Eiffel (*podvm*), implique une certaine redondance et une perte d'efficacité; on sent bien ici l'intérêt d'un POM ayant le même modèle que le langage.

## 12.5.2. Traitement des transactions

Le fait que l'on ne puisse manipuler à partir de *O2-Engine* que des *poignées*, pose un problème: à la fin de chaque transaction, la poignée n'est plus valide et il faut la régénérer en partant d'une racine de persistance, ce qui est inapplicable dans notre approche. Deux solutions s'offrent à nous:

- rendre la table *p_instances* persistante (mais cela pose des problèmes d'efficacité, cf. §

10.2), et l'associer à une racine de persistance,

• gérer à partir des pcollection un numéro unique qui puisse être utilisé à la place des poignées dans la table des *p_instances*;

c'est cette dernière solution qui a été choisie. Dans le cas où plusieurs accès simultanéss sur un objet sont permis, on devra rennoncer à l'amélioration du paragraphe 10.4 qui permet de supprimer une indirection des le 2ème accès. En effet après la fin d'une transaction, il faut redemander au POM, l'autorisation d'utiliser l'objet (et donc le recharger s'il a été modifié par une autre application). Dans un univers mono-utilisateur ce nouveau chargement est inutile, et donc l'amélioration du paragraphe 10.4 reste valide.

D'autre part, le fait qu'une nouvelle transaction débute implicitement à la connexion et à chaque terminaison d'une transaction, rend inutile les déclenchements de début de transaction, mais demande en même temps au POM de gérer une transaction même lorsqu'on ne manipule que des objets volatiles.

# Partie IV

# Bilan et perspectives de recherche

# 13

# Bilan de ce travail

Ce travail propose une approche qui permette d'intégrer la persistance dans les langages à objets selon un certain nombre de concepts dont ceux d'*objet persistant*, de *monde persistant*, de *pcollection*, de *pview* et de *requêtes sélectives*. Ces concepts autorisent un meilleur partage des objets entre applications, et une diminution du code à écrire pour les entrées-sorties d'environ 30%, sans pour autant et bien au contraire altérer la réutilisabilité des composants, et en préservant les capacités des composants à évoluer.

A l'issue de ce travail nous constatons cependant un certain nombre de problèmes liés à l'efficacité et au coût de la persistance. Le paragraphe 13.1 développe l'idée que toute la persistance n'est pas toujours utile et qu'il faut s'adapter aux besoins des applications; le paragraphe 13.2 dresse un inventaire des expériences que nous avons menées sur le coût de la persistance.

## 13.1. Intégration incrémentale de la persistance

Nous avons proposé une approche pour introduire la persistance dans les langages à objets comme Eiffel, à travers un mécanisme aussi transparent que possible aux applications. Nous garantissons aussi l'intégration de la plupart des services qui sont offerts par les systèmes de gestion de base de données, quand ceux-ci sont compatibles avec la philosophie des langages de programmation.

Ces services offerts sont:

• une gestion incrémentale des échanges (chargement, libération et mise à jour), entre les mémoires volatile et persistante (voir chapitre 10),

• des possibilités de sélection et de mise à jour globale d'ensembles d'objets (accès associatif), à travers le concept de *requête sélective* (cf. § 3.3 et chapitre 6),

• le renforcement du langage des assertions par l'introduction des pcollections qui permettent la gestion automatique de l'extension des types et l'utilisation de quantificateurs existenciel et universel (voir sections 3.4 et 6.7),

• un mécanisme transactionnel qui permet de garantir l'intégrité des données en cas de panne, les transactions sont soit implicitement associées à une routine exportée, soit déclenchée explicitement par l'application (cf. § 7.1),

• une gestion des protections des classes et des objets persistants, basé sur le concept de *pview*, et le partage des objets entre plusieurs vues (cf. § 7.2),

• une tentative d'unification du traitement des entités persistantes (classe et objet persistant), avec le concept de *monde persistant* (sections 3.6, 5.2 et 6.8),

• La généralisation des requêtes sélectives à toutes les structures parcourables.

La transparence est assurée grâce à:

• la gestion automatique du chargement, de la mise à jour et de la libération des objets persistants,

• la gestion implicite des transactions (dans un univers non conccurent),

• la propagation du type des objets en mémoire persistante (cf. § 9.2),

• l'orthogonalité de la gestion de la persistance par rapport au système de type: tous les objets persistants ont droit à tous les services de la persistance, (cf. § 3.2 et 5.1)

• l'indépendance de la persistance par rapport au code de l'application (voir sections 3.2 et 5.1).

Cependant, l'intégration d'une telle approche de la persistance est coûteuse (temps, espace mémoire), et n'est pas utile à tout type d'application; certaines peuvent se contenter d'une couche peu complexe au dessus du système de fichier comme le système BOSS (Binary object streaming service) associé à Smalltalk [pps 90].

Nous conseillons donc le développement d'une version améliorée de la classe *ENVIRONMENT* [Meyer 90] qui offrirait un sous-ensemble des services proposés dans notre approche, mais qui n'utiliserait pas un gestionnaire d'objets issu des bases de données.

## 13.2. Coût de la persistance

Le coût de l'introduction de la persistance est un point important quand on cherche à la réaliser dans un langage à objets comme Eiffel, qui est destiné à la construction d'applications opérationnelles dans le monde industriel. Nous avons étudié les problèmes de performances sous différents aspects (accès navigationnel et associatifs), et nous avons été confrontés aux choix suivants:

- choix du gestionnaire d'objets permettant la mise en oeuvre,
- choix des catégories de primitives à propager (attribut, routine),
- choix de la date de propagation des types en mémoire persistante.

Les études que nous avons menées montrent clairement, comme nous l'avons plus longuement développé dans le paragraphe 8.1, que la seule solution qui permette une utilisation industrielle est de construire un gestionnaire d'objets adapté au langage à objets choisi pour cible, et qui offre des fonctionnalités de bas niveau, mais bien adaptées à l'exécutif du langage.

Il existe un certain nombre de gestionnaires sur le marché qui pourraient être utilisés (Kala, Wiss, Nicemp, ...); mais pour pouvoir mettre en oeuvre des mécanismes de sélection avec des améliorations (index, ...) et l'application de méthodes sur des objets, le gestionnaire a besoin de connaître la sémantique des classes et des objets, et donc le modèle de données du langage de programmation. Cela nécessite la construction au dessus du gestionnaire sélectionné, d'une couche supplémentaire, qui prenne en compte le modèle du langage Eiffel.

L'implémentation d'un tel gestionnaire demande un effort de plusieurs hommes/année et il n'a donc pas été possible de l'envisager, ni dans le cadre du projet *Business class,* ni dans celui de cette thèse. Nous avons donc choisi d'expérimenter un SGBD ($O_2$), pour réaliser l'implémentation de la persistance et étudier les problèmes cités ci-dessus; les fonctionnalités offertes par ce SGBD nous permettront de valider les concepts proposés et les choix d'implémentation, sans nécessiter le développement des

couches basses d'un SGBD.

Le principal inconvénient de l'utilisation d'un SGBD comme $O_2$ (cet inconvénient serait le même avec n'importe quel autre SGBD), **provient de la redondance des modèles**. En effet, il faut d'une part traduire les classes dans un autre modèle avec les problèmes de compatibilité sous-jacents (voir chapitre 12), mais aussi, lors de chaque échange entre la mémoire volatile et la mémoire persistante, convertir les objets d'un modèle vers l'autre; le coût de cette conversion est non négligeable.

**Dans un premier temps nous avons** choisi de ne propager que les attributs (et pas les routines), dans le monde persistant (voir chapitre 11). Cette approche permet l'utilisation, à la fois de SGBD relationnels dont la maturité est démontrée (par opposition aux SGBD à objets), mais elle simplifie beaucoup la propagation des types (traitement incomplet de la généricité, aplatissement de la hiérarchie des classes, ..).

Le principal inconvénient de cette approche concerne l'orthogonalité des requêtes sélectives par rapport au langage Eiffel, et la transparence d'utilisation dans le cadre des sélections. Il n'est en effet pas possible, lorsque les accès associatifs sont sous le contrôle du SGBD, de faire référence à des routines dans un critère de sélection; cela nous oblige soit à supprimer la possibilité d'utiliser des routines dans un critère de sélection, soit à gérer l'exécution en mémoire volatile; nous avons choisi la deuxième solution.

Pour diminuer le coût d'une exécution en mémoire volatile, nous avons proposé un mécanisme pour réorganiser au mieux le contenu des critères de sélection afin de retarder l'évaluation des routines et de minimiser le nombre d'objets chargés (voir chapitre 11). Cependant cette approche montre ses limites dans un certain nombre de cas et ajoute toujours un coût non négligeable pour l'analyse des critères de sélection. On notera cependant des résultats intéressants par rapport à une sélection qui n'utilisent pas ce mécanisme.

**Dans un deuxième temps nous avons** choisi de **propager les routine**s (traduction complète des classes) dans le monde persistant, pour permettre leur exécutionsous le contrôle du gestionnaire, et donc d'éviter des chargements d'objets persistants, notamment pour les sélections.

La propagation des routines nécessite la mise en oeuvre d'un mécanisme complexe, surtout dans le cas d'un SGBD, puisqu'il s'agît alors d'une véritable traduction d'un langage vers un autre avec les problèmes de compatibilité (*impedance mismatch*) et de redondance que cela implique même quand les modèles sont relativement proche comme c'est le cas des langages Eiffel et $O_2$.

Parallèlement, nous avons étudié le choix de la date de propagation des types dans le monde persistant. Différer la propagation des types au moment de l'exécution (lors du stockage de la première instance) privilégie la transparence et n'implique pas l'ajout d'une passe de compilation; par contre il pénalise les temps d'exécution et augmente la taille du code (de l'ordre d'une centaine de classe Eiffel si on utilise une interface générale).

Le choix de propager des routines en mémoire persistante ne permet cependant plus de retenir une solution dynamique, car le temps de traduction serait trop important. Nous avons donc fait réaliser un traducteur statique qui génére des classes *O2* avec leurs routines à partir du source des classes Eiffel.

L'utilisation d'une approche statique et de la couche basse du SGBD (*O2-Engine*) pour établir la conversion vers les classes et les objets *O2*, laisse espérer des performances raisonnables en utilisation opérationnelle. Cependant nous pensons que l'utilisation d'un gestionnaire de plus bas niveau, enrichi d'une couche spécifique pour Eiffel, permettrait une utilisation plus simple, moins onéreuse et plus performante.

## 13.3. Etat d'avancement

L'implémentation de notre prototype est réalisée avec la version 2.3 d'Eiffel.

### 13.3.1. Composants Eiffel pour l'implémentation du modèle

Tous les composants nécessaires à l'implantation du modèle ont éré implémentés, à l'exception de la classe *EIFFEL_CLASS* et de quelques primitives de la classe *PVIEW* qui gèrent les protections et le partage des objets par plusieurs *pview*. En particulier, les composants réalisés sont:

- la hiérarchie correspondant aux collections (voir § 6.6),
- les classes *HERE, POI, PVIEW, TRANSACTION*,
- les composants permettant de gérer la connection avec *O2* et les erreurs.

Par ailleurs, certaines classes contenant des routines externes, dont la classe *ARRAY*, ont été modifiées pour intégrer la persistance (appels explicites pour rendre persistant et modifier des objets).

### 13.3.2. Composants pour l'nterfaçage avec O2

Il a été réalisé une interface complète avec *O2* (une centaine de classes): tous les

concepts $O_2$ sont modélisés par des composants Eiffel. Cette interface a servi de support pour l'implémentation de l'approche dynamique. Nous regrettons cependant de ne pas avoir pu l'adapter au schéma d'interfaçage type pour les SGBD, proposée par SOL[1] . Ceci est en partie dû au fait que les possibilités de requêtes de $O_2$, accessibles à ce jour à travers le langage C sont limitées.

### 13.3.3. Adaptation de l'exécutif Eiffel

A l'heure actuelle, une grande partie des modifications qu'il est nécessaire d'apporter à l'exécutif Eiffel sont réalisées. Cela inclus notamment:

- les appels nécessaire pour gérer les défauts d'objets
- le mécanisme de transaction,
- la table permettant de mémoriser les objets persistants chargés,
- la propagation de la persistance par attachement.

Par contre la modification du ramasse-miettes n'a pas été réalisé à cause des changements profond de celui-ci dans la version 3 d'Eiffel qui va sortir très prochainement.

Beaucoup d'attention a été porté à la réalisation de la table permettant de gérer les objets chargés; elle joue un rôle important pour l'aspect opérationnel de notre prototype.

les routines présentées au chapitre 10 ont toutes été implémentées et sont pratiquement identiques pour les versions dynamiques et statiques.

Le problème cité au chapitre 12 concernant la perte des poignées après la fin d'une transaction a été traitée en utilisant le numéro des objets dans leur **pcollection**; une routine est fournie à l'utilisateur pour compacter les *pcollection* qui ont subi un grand nombre de suppressions d'élements.

---

1 Société des Outils du Logiciel, distributeur d'Eiffel en France, président B. Meyer.

### 13.3.4. Implémentation de l'approche dynamique

Les composants permettant la traduction de toute classe Eiffel non générique, et de la classe *ARRAY*, à partir d'un graphe d'objets sont implémentés. La conversion des objets Eiffel en objet $O_2$ et vice-versa a aussi été réalisée. La traduction des classes et la conversion des objets tiennent compte des attributs à *void* et des cycles dans le graphe de clientèle.

Le mécanisme permettant l'amélioration de l'évaluation des requêtes sélectives a été réalisé pour $O_2$ et pour Oracle. Par contre il est limité, à l'heure actuelle à des requêtes non emboitées.

Les classes génériques n'ont pas pu être traduites correctement à cause de l'absence, dans l'exécutif, d'informations sur le type des paramètres génériques.

Les requêtes sélectives n'ont pu être implémentées à cause de l'absence provisoire, dans $O_2$ des outils nécessaires.

La hiérarchie des classes Eiffel et les routines ne sont pas propagées dans le monde $O_2$; ce choix a été fait essentiellement pour des raisons d'efficacité.

Dans cette approche nous avons, par manque de temps, aussi laissé de coté le traitement des attributs expansés.

### 13.3.5. Implémentation de l'approche statique

L'approche statique est celle qui est le plus complètement implémentée. Nous y avons consacré toute notre énergie, dés que nous avons estimé à leur juste proportion les limitations de l'approche dynamique.

la traduction de toutes les classes Eiffel, y compris les classes génériques, et celles contenant des attributs expansés a été implémentée. Deux routines de conversions (Eiffel vers $O_2$ et vice-versa) sont générées pour chaque classe traduite.

Il est possible de réaliser des requêtes sélectives, même emboitées, mais l'utilisation des index n'est pas possible à l'heure actuelle (implémentation incomplète des outils accessibles à travers *$O_2$-Engine*).

La traduction des routines a été implémentée mais n'est pas encore intégrée dans notre prototype. Le traitement de l'héritage répété souffre de limitations dans certains cas très complexes.

## 13.4. Contributions

Comme cela a été précisé dans l'introduction, ce travail s'est déroulé dans le cadre d'un projet ESPRIT II, nommé *Business Class*, avec les contraintes que cela entraîne au niveau de l'ampleur du travail demandé et du niveau de définition.

Naturellement, **l'implémentation des idées exposées dans cette thèse, n'est pas le fruit d'une seule personne, mais d'un travail d'équipe**.

Ma responsabilité personnelle se situe essentiellement au niveau de celle d'un chef de projet: administration, pré-études, conception du modèle de persistance, spécifications de l'implémentation, étude des problèmes et des solutions de mise en oeuvre. Pour ce qui est de l'implémentation, j'ai développé les composants Eiffel utilisés dans ce modèle de persistance et les modules de l'approche dynamique.

L'implémentation de l'approche statique est l'oeuvre de Jacques Farré qui a réalisé un travail de haute qualité et de grande ampleur pour le traducteur d'Eiffel-O2. C'est lui qui a réalisé le traducteur Eiffel-O2. C'est lui qui a conçu les schémas de traduction entre les deux mondes Eiffel et $O_2$, et a assuré l'essentiel de l'implémentation, avec la participation de Melle Violaine Séré de Rivières. En plus du traducteur, Jacques a trouvé une implémentation efficace de la table de hash-code qui gère les objets chargés, et une identification des objets persistants indépendante des poignées d'*O2*.

Robert Chignoli a implémenté en grande partie les requêtes sélectives en $O_2$, et a réalisé une petite application qui permet de tester la persistance en Eiffel.

Jean-Marc Jugant a réalisé une première version de l'interface d'Eiffel vers $O_2$. Il a fait un travail important, à un moment où les primitives de $O_2$ ($O_2$-Engine), n'étaient pas encore stabilisés.

Philippe Brissi a réalisé l'implémentation du modèle portant sur l'amélioration de l'évaluation des requêtes sélectives présenté au chapitre 11.

Les modifications de l'exécutif Eiffel ont été réalisées par Jean-Pierre Sarkis, selon nos spécifications spécifications du chapitre 10. Sans son efficace collaboration, nous n'aurions jamais pu réaliser une intégration transparente de la persistance.

Nhanh Le Thanh a contribué à la formalisation du modèle de la persistance, et a apporté son expérience en matière de base de données.

Roger Rousseau a apporté son expérience dans le domaine du génie logiciel et a contribué à un grand nombre de solutions par ses critiques et ses commentaires constructifs. En particulier il a contribué à la conception des requêtes sélectives et des Pcollections et il a influencé les choix d'implémentation pour la gestion des objets persistants.

# 14

# Perspectives

L'introduction de la persistance et l'arrivée prochaine de la version 3 d'Eiffel laisse entrevoir un certain nombre de perspectives pour enrichir l'environnement de programmation fourni aux utilisateurs. En effet, la persistance offre une plateforme idéale pour la modélisation du cycle de vie des composants et des objets. Deux axes nous semblent essentiels: l'uniformisation de la gestion des classes et des objets persistants, et un outil de gestion de version.

## 14.1. Enrichissement de l'environnement Eiffel

### 14.1.1. Rappel sur le compilateur Eiffel

Dans la version 2.3, le traducteur Eiffel génère un certains nombre d'informations qui sont mémorisées au fur et à mesure dans des fichiers (chaque fichier source d'une classe est associé à un répertoire contenant ces fichiers). Le choix de cette technique a un certain nombre de conséquences:

• non transparence de la gestion interne et donc manque de sécurité du fait des possiblité d'interférence (destruction des fichiers internes ou modification des dates),

• pas de gestion de la compilation parallèle de classe partagées (lié au système de fichier Unix),

• plusieurs passes sur le texte source et donc des erreurs lorsque le texte source est modifié pendant la compilation,

• une diminution des performances (lecture et écriture de fichiers texte plus ou moins complexe),

• difficulté et lenteur pour récupérer les informations concernant une classe pendant l'exécution,

• quasi-impossibilité de gérer proprement des versions de composant

Afin de résoudre ces problèmes il faudrait générer, des instances de la classe *EIFFEL_CLASS* , lors de la compilation, afin de mémoriser la représentation interne d'une classe, et de permettre sa maintenance dans une perspective de gestion des versions. Cet aspect a été amélioré avec la version 3 mais ne semble pas intégrer l'uniformisation (pour la maintenance), des objets persistants et des classes.

### 14.1.2. Besoins d'un interprète

De même, l'interprète Eiffel, développé au sein de l'équipe OCL (Objet et Composant Logiciel) du laboratoire I3S [Brissi 90], créé pendant la phase de traduction un certain nombre d'objets Eiffel qui gènèrent en plus des informations nécessaires à l'interprétation (représentation interne) des informations utiles et consultables sur les classes Eiffel. Pour ne pas recommencer à chaque interprétation toute la génération et permettre un minimum d'incrémentalité, il faut pouvoir sauvegarder ces objets en mémoire persistante.

On notera qu'un interprète est actuellement en cours de réalisation chez ISE[1] pour la version 3; nous ignorons s'il génère de vrais objets persistants ou seulement des fichiers qui contiennent les informations correspondantes.

### 14.1.3. Perspectives pour de nouveaux outils

Supposons que le traducteur Eiffel, pour poursuivre ce qui a été dit plus haut, gère des *collections* associées aux classes *EIFFEL_CLASS* et *FEATURE* (informations sur les classes) qui contiennent en particulier des critères de classification, comme ceux prévus dans les clauses *indexing* des classes Eiffel.

On dispose alors des requêtes sélectives qui permettent de réaliser à faible coût un outil de recherche de composants, où la gestion de l'accès aux objets est complètement prise en charge par le POM, ce qui garantit:

---

[1] Interactive Software Engineering, 1ère implémentation d'Eiffel, président B. Meyer.

• des performances intéressantes pour l'accès aux informations,

• une non-limitation sur la taille du graphe d'objets représentant les instances des classes *EIFFEL_CLASS* ou *FEATURE* manipulables,

• l'absence de pénalisation pour l'application, en ce qui concerne l'espace mémoire utilisé; dans la version actuelle, l'ensemble des instances de la classe E_CLASS qui contiennent des informations sur les classes est généré en une seule fois, et se trouve donc présent en mémoire.

## 14.2. Le cycle de vie des entités persistantes

Avec l'introduction de la persistance on élargit le facteur temps dans une application; il faut prendre en compte la gestion du cycle de vie des composants et des objets. Ce paragraphe a pour objectif de montrer quelques uns des points essentiels à prendre en compte pour introduire la gestion des versions dans un système persistant. Cet aspect ne fait qu'une timide apparition dans les langages de programmation, aussi les exemples trouvés dans la littérature sont en général issus du monde des bases de données (Orion, Gemstone, Encore, ..).

L'idée de gérer des versions n'est pas nouvelle; à l'origine, son but était de satisfaire plusieurs besoins [Autr 87]:

• le besoin, pendant le développement d'un logiciel, que plusieurs concepteurs puissent travailler sur des mêmes modules, sans être constamment affectés par les changements opérés par les uns ou par les autres,

• le besoin de pouvoir retracer l'historique d'un projet ou de pouvoir exprimer un savoir faire utile pour des conceptions ultérieures,

• faciliter la restauration en cas de repentir.

Depuis l'avènement de l'approche orientée objet, l'idée d'introduire la notion de version dans les modèles de données utilisés pour l'implantation de systèmes persistants a fait son chemin. Au cours de son cycle de vie, un composant, ou même un graphe de composants pourra passer par plusieurs états: ces états seront appelés des

versions. Dèsormais le concept de version doit permettre à la fois:

• l'amélioration de la gestion du développement d'un logiciel dans le cadre d'un projet pour lequel, un ou plusieurs concepteurs seraient concernés;

• la gestion de l'évolution d'un logiciel alors qu'il est déjà en service; il faut alors répercuter aux niveau des instances et des autres composants, les changements qui interviennent dans les composants modifiés.

Il faut aussi faire la distinction entre les versions d'instances et de classe. Nous nous attacherons plutôt ici aux versions de composants.

### 14.2.1. Versions pour l'évolution des composants

#### Modélisation de l'évolution d'un composant

Pour modéliser l'évolution du processus de conception, on trouve essentiellement deux types de versions: les versions *successives* et les versions *alternatives*, (ou parallèles) [Banerjee 87, Chou 88, Gardarin 88 b, Zdonik 86 b, Cohen 88].

Les versions *successives* permettent de modéliser la suite chronologique des modifications opérées au niveau du composant, afin qu'il puisse suivre l'évolution de l'application (introduction de la notion de temps). Le terme application peut aussi bien être appliqué au schéma d'une base de données qu'à un système de classes Eiffel.

Les versions *alternative*s permettent d'exprimer le fait qu'à un point donné du processus de conception, on a eu plusieurs solutions pour faire évoluer la définition du logiciel. On peut aussi trouver dans la littérature un autre type de version appelée version *parallèle*; elle permet de définir plusieurs représentations pour la même notion et se rapproche ainsi du concept d'héritage. Souvent ces deux types de version sont confondus, c'est le cas dans Orion [Kim 88 b]; cependant certains travaux comme ceux de E. Chang et R. Katz [Chang 89, Katz 87], introduisent le concept de lien d'équivalence qui permet de différencier les versions alternatives des versions parallèles.

Souvent la différenciation entre versions *successive* et *alternative* se fait implicitement, de part leur position dans un arbre de versions. Toutes les versions se trouvant à une même profondeur dans l'arbre décrivent des versions alternatives. C'est le passage à un niveau inférieur qui exprime le passage à la version suivante chronologiquement. Cette version est une version successive pour celle du niveau

précédent .

On notera qu'il est possible de faire la différence entre les versions successives créées à partir d'une seule version et celles crées à partir de plusieurs versions (arbre de versions) [Benzaken 88].

En plus des liens cités ci-dessus, on peut en trouver un autre permettant de relier toutes les versions d'un objet à un même objet logique appelé *objet générique* [Banerjee 87]. Dans Encore [Zdonik 86b], l'objet logique est plutôt vu comme une interface unique qui, représente les spécifications initiales et qui sera utilisée pour l'accès à toute version de l'objet (ce sont des gestionnaires d'erreurs qui gèrent les problèmes dûs à l'utilisation de versions différentes).

Certains systèmes qui fournissent pourtant une gestion de l'évolution de composants n'offrent pas de mécanisme supportant la notion de version d'instances. Il s'agit en particulier de *Gemstone, Gbase, Vbase, $O_2$* [Gardarin 88a]. Ceci a deux conséquences:

- la difficulté pour l'utilisateur de pouvoir revenir facilement sur une modification s'avérant peu pertinante

- la difficulté pour faire de la conception d'application en parallèle.

### Principales caractéristiques de la structure interne

Du point de vue de la structure interne du modèle de versions on constate que dans Orion [Banerjee 87] l'ensemble des versions est un arbre de descripteurs de versions. En effet pour chaque objet révisable, on a un objet générique qui gère les versions; cet objet contient principalement les propriétés suivantes:

- la version par défaut,

- un compteur de versions,

- une hiérarchie de descripteurs de versions qui contiennent, eux-mêmes, le numéro de la version de l'objet, un identificateur de version et la liste des descripteurs de version du niveau inférieur.

La gestion des versions de composants dans Geode [Gardarin 88b] utilise une technique similaire, basée sur la **notion d'identificateur logique d'objet**, (toutes les versions d'un objet sont accessibles à partir de cet identificateur). Pour implémenter la notion de versions de schéma (graphes de composants), le choix s'est porté sur la duplication de toutes les classes concernées par la modification ou l'ajout de certaines informations.

Dans Encore [Zdonik 86b & 87], on associe une interface pour l'ensemble des versions d'un type et un mécanisme d'exceptions qui permet de réagir aux problèmes de compatibilité; l'interface sera composée par toutes les primitives contenues dans l'ensemble des versions. Chaque version devra contenir des clauses pour réagir en cas d'erreur; ces erreurs peuvent être dues à une demande de consultation ou de mise à jour de propriétés qui n'existent pas ou qui ont changé de type. En conséquence, il faudra mettre à jour la version précédente avec les clauses d'erreur appropriées.

Les techniques employées dans Gypsy [Cohen 88] sont basées sur le système de fichiers d'Unix: les composants sont vus comme des fichiers qui se trouvent dans des répertoires. Pour Gypsy, un **répertoire** est un objet qui posséde un certain nombre de primitives de gestion et modélise la **notion d'ensemble de versions** pour un type donné; toutes les versions d'un composant sont référencées à partir d'un objet *répertoire*. et le système se sert de la structure arborescente du système de fichiers UNIX pour modéliser l'arbre des versions.

### L'accès et le choix de la version

L'accès aux versions se fait à l'aide d'identificateurs logiques ou physiques. Lorsque l'identificateur est physique, c'est à dire indique une adresse en mémoire secondaire, la création d'une version à un niveau donné de la hiérarchie demande une répercution des modifications à la fois vers les niveaux inférieurs et supérieurs, c'est le cas pour le système EXODUS [Carey 86]; il y a obligatoirement une liaison statique entre les versions.

Au contraire, quand les versions sont référencées par des identificateurs logiques, on peut mettre en oeuvre à la fois des liaisons statiques ou dynamiques entre les versions. La liaison entre deux versions est dynamique lorsqu'elle dépend de la date qui identifie la version, elle est statique lorsque la liaison identifie un contenu figé.

Dans la plupart des systèmes (Orion [Kim 88 a]), la liaison dynamique est matérialisée par la *version par défaut*, qui est souvent la plus récente. Parfois ce critère de *la version*

*la plus récente chronologiquement*, n'est pas applicable; c'est en particulier le cas pour des versions alternatives. De même, la liaison statique est, quand elle existe, souvent spécifiée par l'utilisateur.

Dans le modèle de version défini dans [Chang 89], on met en oeuvre le concept de noeud logique à l'aide d'un objet appelé *objet générique* qui permet l'accès aux différentes versions d'un objet. Le modèle de version permet soit de laisser le système décider de la version utilisée par une application, soit de relier une application à une version donnée de l'objet.

Dans les deux exemples précédents, les instances d'une classe ne sont pas maîtresse de la version auxquelles elles se réfèrent; le cas de Encore [Zdonik 86 b] est différent; la version utilisée n'est pas choisie au niveau de la classe mais de l'instance: cette liaison est statique. Pour faire face aux problèmes d'accès à des primitives inexistantes, le créateur d'une nouvelle version devra mettre à jour les versions précédentes en ajoutant des clauses d'erreurs traitant l'absence de la primitive. De même il décrira des clauses d'erreur dans la nouvelle version pour réagir à l'absence d'une primitive en cas de suppression. Par contre la notion d'objet logique existe sous la forme d'une interface, pour l'ensemble des versions d'un type.

### 14.2.2. Versions pour le développement des composants

On retrouve toujours à peu près trois niveaux d'élaboration de composants:

- un qui correspond au fait que le composant est instable,
- un autre qui exprime que le composant est au contraire assez stable pour être utilisé par les autres concepteurs,
- et un autre enfin, qui est réservé aux composants complètement opérationnels.

Dans Orion, [Chou 88], on considère les trois types de versions:

- une *version transitoire* modifiable à souhait par son propriétaire et réservée à son propre usage,

- une *version de travail*, obtenue à partir d'une *version transitoire*, qui ne peut être mise à jour, (elle peut simplement être supprimée par celui qui l'a conçu) et peut être utilisé par les autres concepteurs,

• une *version opérationnelle*, créée à partir d'une *version de travail*, qui réside dans la base de données publique, et qui ne peut ni être mise à jour, ni être effacée.

Dans le modèle défini par Chang, [Chang 89], ces trois types de version existent mais de manière moins explicite. En effet il introduit la notion d'espace de travail; ces espaces de travail sont *privés, partagé par un groupe* de personnes ou destinés à l'*archivage*; chacun d'eux correspond à un état du composant.

Le mécanisme général consiste:

• à charger une version dans son espace de travail privé, à partir d'un espace de type archive,
• d'y faire les modifications nécessaires qui ne seront visible aux autres que lorsque cette version sera placée dans l'espace de travail d'un groupe ou dans l'espace archive.

Le gestionnaire de versions de Gypsy a introduit aussi la notion d'espace de travail; cependant il n'y a qu'un type d'espace de travail. Les versions d'un objet sont regroupées dans un répertoire qui est accessible à un certain nombre d'utilisateurs, en fonction des droits d'accès; le mécanisme est similaire à celui mis en oeuvre par Chang, mais la possibilité de partager une version est obtenue par le fait qu'une version peut être attachée à plusieurs espaces de travail ou à plusieurs utilisateurs.

Dans Encore, aucun système ne contrôle le processus de conception: le développeur qui crée une nouvelle version doit mettre à jour les précédentes en leur ajoutant des clauses d'erreurs. Ceci permettra aux instances créées avec une ancienne version d'avoir un comportement cohérent au cas ou elle recevrait un message d'une instance créée sous une nouvelle version.

En fait, les modifications ne sont pas répercutées sur les classes utilisatrices. C'est seulement lors de l'accès à un objet, utilisant l'objet modifié, que l'on prend en compte les problèmes. On ne cherche pas à modifier les objets qui utilisent une version donnée, pour l'adapter à la dernière version; c'est la version elle-même qui est modifié; par l'adjonction de clauses d'erreurs.

### 14.2.3. La gestion des modifications

#### Modifications autorisées sur les objets

La diversité des modifications qui sont autorisées sur un graphe de composants est une indication de la puissance du gestionnaire de versions.

Les modifications que les principaux systèmes autorisent peuvent être groupés en trois grandes catégories:

- modifications du contenu d'une classe:
  - Ajout ou suppression d'une méthode ou d'une propriété,
  - modification d'une propriété, (changement de nom, du type, des contraintes),
- Ajout ou suppression d'une classe dans le graphe de classes
- modification des relations entre les classes

On notera cependant des différences entre les systèmes:

- dans Gemstone et Encore [Penney 87 b, Zdonik 87] on peut changer le type d'une propriété en un type plus général ou au contraire plus spécialisé, (sur-classe ou sous-classe), tandis que dans Orion [Kim 88 b], seul les changements pour un type plus général est possible;

- l'absence dans Gemstone, Encore ou Gbase [Gardarin 88 a, Nguyen 89] de la possibilité de renommer une classe et, dans le cas de Encore ou Gbase, de renommer une primitive. De ces trois systèmes, seul Encore offre la possibilité d'ajouter ou d'enlever un parent. Cette dernière restriction supprime un grand nombre de problèmes au niveau de la répercussion des modifications et elle influe sur les capacités à fournir une aide pour la conception.

- d'autres systèmes comme *Vbase, Iris*ou $O_2$ [Gardarin 88 a] ne possédent que peu de primitives qui permettent de modifier un schéma. Ainsi *Iris* autorise la suppression de méthodes et de propriétés, tandis que $O_2$ et *Vbase* ne fournissent que l'ajout et la suppression de classes.

### Répercussions des modifications sur le graphe d'objets

Certains systèmes offrent seulement un mécanisme de base pour structurer les différentes versions de composants, mais la répercussion des modifications est à la charge de l'application; c'est le cas de Geode [Gardarin 88 b] et Exodus [Carey 86]. Le

premier offre un mécanisme de versions qui permet de différencier les versions successives et alternatives, mais pas les références qui ne sont plus cohérentes après modifications. Dans le second, on peut seulement créer ou supprimer les versions d'un objet mais on a aucune aide pour l'adaptation des objets qui en dépendent.

D'autres systèmes fournissent en plus un mécanisme pour la répercussion des modifications. Du fait que ces systèmes mettent en oeuvre un accès logique vers les versions, la partie supérieure de la hiérarchie n'est pas dupliquée au moment de la modification d'un composant.

Il faut aussi faire la différence entre la répercussion des modifications au niveau des instances d'une classe et la répercussion au niveau des classes héritières ou clientes.

Au niveau de l'instance, cela dépend essentiellement du caractère statique ou dynamique de la liaison avec une version de la classe. Dans le cas où cette liaison est statique, comme dans Encore, l'instance n'est pas modifiée, ce sera à la version du composant à s'adapter; si une conversion explicite est demandée, alors l'instance ainsi modifiée n'est plus compatible avec l'ancien type (les deux versions sont des types à part entière qui n'ont aucun lien entre eux [Skarra 86].

Si la liaison est dynamique, alors deux attitudes pourront être adoptées. Soit l'instance est adaptée totalement à la nouvelle version, soit seule les propriétés en plus sont ajoutées. Cette dernière solution est difficile à gérer (problème de conflits en cas de liaison dynamique); cela explique pourquoi certains systèmes limitent les modifications possibles ou restreignent les cas d'utilisation [Banerjee 87, Penney 87 b]. Dans le cas d'Orion et de Gemstone, l'instance est adaptée à la nouvelle version; pour Orion, les objets composites [Nguyen 89, Kim 87] compliquent ce mécanisme.

Si on considère les répercussions au niveau de la classe, plusieurs attitudes sont possibles:

- la classe cliente s'adapte à la nouvelle version de manière automatique (propagation des modifications),

- la classe cliente est informée des modifications (notification des modifications) et se réserve la possibilité de s'y conformer ou non,

- les changements sont toujours fait **manuellement.**

Dans Orion et dans Gemstone, la propagation est automatique mais connaît certaines restrictions. Dans Encore, la direction qui a été choisie ne nécessite pas la propagation des changements au niveau de la classe (l'interface représente l'union des spécifications de l'ensemble des versions de la classe), sauf l'ajout éventuel de clauses d'erreur. Dans le modèle mis en oeuvre par E. Chang la propagation des modifications se fait manuellement.

Remarque Un autre problème se pose lorsqu'une méthode ou un attribut est supprimé: doit-on invalider ou modifier les méthodes qui les référencent?

**La répercussion des modifications sur les instances**

Un autre point important est le délai qui s'écoule entre le moment où un composant est modifié par l'utilisateur et le moment où ces modifications sont répercutées sur les instances.

Par ailleurs, si on considère les répercussions sur les classes qui l'utilisent, (héritiers ou clients), la répercussion est soit immédiate Gemstone [Penney 87 b], soit retardée jusqu'a l'accès de l'objet, comme dans Orion. On peut aussi faire varier ce délai de propagation en groupant ou non les modifications faites sur un objet ou un ensemble d'objets. Une autre solution est adoptée dans Encore: une instance n'est modifiée que sur demande explicite de l'utilisateur.

La façon d'informer les classes utilisatrices influence la date de répercussion: si le système informe une classe de la modification d'une autre par le positionnement d'indicateur consultés périodiquement, la répercussion sera obligatoirement différée, par contre si la notification de modifications se fait par envois de messages, alors la répercussion pourra, au choix, être immédiate ou différée.

### 14.2.4. Principaux problèmes

Parmi les problèmes techniques à résoudre, on trouve essentiellement les problèmes suivants:

• stockage des versions d'objets: l'évolution de l'état d'un objet correspond en fait un ensemble de modifications; se pose le problème de mémoriser physiquement ces modifications. le type d'organisation de la mémoire virtuelle [Gardarin 88 a] et le

choix de l'unité de transfert sont des facteurs prépondérants pour l'amélioration du couple performance/espace. Deux orientations sont possibles: soit on considère une entité dans sa globalité, (on tient alors compte de l'aspect fonctionnel de l'objet), soit on la considère comme un ensemble de parties de taille identique, (qui n'ont aucune signification sémantique);

• résolution des conflits dans les classes héritières: Dans Orion le problème des conflits de nom est contourné par l'utilisation de trois règles, basées comme c'est le cas dans [Ducournau 89], sur l'ordre des super-classes et sur le niveau de profondeur dans l'arbre. Dans Gemstone comme dans Encore il n'y a pas de résolveur de conflit; mais dans Gemstone, toute opération de propagation qui entraîne un conflit est annulée [Penney 87];

• l'accès concurrent: pour controler la cohérence des versions en cas d'accès multiples et simultanés, il faut mettre en oeuvre un mécanisme permettant de sérialiser les modifications;

• le coût de la gestion de versions: la gestion des versions entraîne un surcoût tant au niveau de l'espace utilisé pour stocker les objets, (il faut ajouter à chaque objet un certain nombre d'informations), qu'au niveau des performances; en effet, un certain nombre d'opérations comme la détermination de la version à utiliser, la gestion des valeurs manquantes, ralentissent le système.

# 15

# Conclusion

Nous avons expliqué dans l'introduction que nous croyions possible d'intégrer de manière transparente et complète, la persistance dans un langage de programmation de haut niveau, comme Eiffel [Meyer 91], et que cette intégration devait favoriser la réutilisation et l'évolution, à la fois des composants et des objets, tout au long du déroulement de leur cycle de vie.

Nous pensons avoir atteint notre objectif. Cependant, seule une utilisation en vraie grandeur, par exemple par nos partenaires du projet *Business Class*, validera réellement les concepts proposés dans cette approche.

Pour conclure, maintenant un certain nombre de remarques et de constatations amenées par notre expérience dans le domaine du génie logiciel et la réalisation de notre prototype.

Nous avons d'abord constaté que toute la puissance de la persistance n'est pas toujours utile à une application et qu'une solution dégradée sur le plan des performances peut souvent être utilisée, pourvu que la vision qu'ont les applications de la persistance soit identique. Le choix du niveau de persistance dépend essentiellement du nombre d'objets à manipuler et du contexte d'utilisation (accès mono-utilisateur ou multi-utilisateurs).

Nous avons ensuite montré que la solution la plus efficace est de disposer d'un gestionnaire d'objets spécifique à Eiffel. Cependant son développement serait d'un coût élevé, et ne serait peut être pas justifié pour seulement éviter la redondance des modèles. En effet, à partir d'une approche statique, c'est à dire par une traduction avant l'exécution, des classes Eiffel dans le modèle supporté par le gestionnaire d'objets, nous sommes persuadés que les performances restent très acceptables. De plus la redondance peut être cachée à l'utilisateur, car notre système assure la cohérence entre les différentes représentations de manière transparente.

Par contre, avec une approche dynamique, quelque soit le type de gestionnaire sur laquelle elle repose, cette solution n'est pas envisageable dans un contexte industriel, à cause de ses limitations (propagation incomplète d'une classe, faibles perspectives d'utilisation) et de son efficacité, même en incorporant les mécanismes d'amélioration, que nous proposons pour les requêtes sélectives.

L'implémentation d'une approche dynamique, basée sur $O_2$ représentait une première approche pour valider nos idées. A l'origine, nous pensions que cette solution était réaliste, mais au fur et à mesure que notre étude avançait, cette solution nous est apparue peu viable. Compte tenu, de la différence des modèles et donc des difficultés pour établir la correspondance entre les concepts qu'ils développent, notamment pour l'héritage et les classes génériques, les mécanismes à mettre en oeuvre sont apparus plus lourd que prévu et générateurs d'inefficacité. Ainsi, dans le cadre de *Business class*, les membres de l'équipe qui travaillent sur ce projet terminent d'écrire actuellement un traducteur d'Eiffel vers $O_2$ qui permet l'implémentation d'une approche statique plus efficace, et prenant en compte pratiquement tous les aspects du langage, rendant aussi transparente la redondance entre les modèles.

Par ailleurs, l'introduction de la persistance dans un langage à objets accroît les besoins au niveau de l'environnement de programmation. En particulier, la gestion du cycle de vie des composants devient un problème essentiel, et la sémantique traitant des aspects statiques et dynamiques d'un programme demande à être affinée; ces points ont été seulement abordés dans ce travail, et seront approfondis par la suite, par les membres du projet *Business Class.*

# Références bibliographiques


[Agrawal 89 a]: R. Agrawal, N.H. Gehani, *Rationale for the design of persistence and query processing facilities in the database programming language O++*, proc. 2nd international workshop on database programming language, Oregon, June 1989, pp.25-40

[Agrawal 89 b]: R. Agrawal, N.H. Gehani, *ODE (object database and environment): the language and the data model*, ACM, 1989, pp. 36-45

[Albano 86]: A. Albano, L. Cardelli and R. Orsini, *Galileo: a strongly-typed, interactive conceptual language*, proc. of the international workshop on object-oriented database systems, pacific-grove, California, 1986, also in [Zdonik 90], pp. 147-161

[Albano 87]: A. Albano, G. Ghelli and R. Orsini, *The implementation of galileo's persistent values*, in [Atkinson 87], pp. 253-263

[Albano 89]: A. Albano, G. Ghelli and R. Orsini, *Types for databases: the galileo experience, proc. of the 2nd workshop on database programming languages*, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 196-206

[Andrews 90]: T. Andrews and C. Harris, *Combining language and database advances in an object-oriented development environment*, in [Zdonik 90], pp. 186-196

[Andrews 91 a]: T. Andrews, *Programming with VBASE*, in [Gupta 91], pp. 130-177

[Andrews 91 b]: T. Andrews, *ONTOS: a persistent database for C++*, in [Gupta 91], pp. 387-407

[Atkinson 83]: M.P. Atkinson, P. Bailey, K.J. Chisholm, W.P. Cockshott, R. Morrison, *An approach to persistent programming*, the computer journal, Vol. 26 N°4, November 1983, pp. 360-365, also in [Zdonik 90] pp. 141-146

Atkinson 85]: M.P. Atkinson, R. Morrison, *Types bindings and parameters in a persistent environment*, proc. Appin Workshop on persistence and data types, Glasgow, Scotland, August 1985, Springer-Verlag pp. 3-20

[Atkinson 87a]: M.P. Atkinson, R. Morrison, *Polymorphic names and iterations*, proc. international workshop on database programming languages, France, eds. F. Bancilhon & P. Buneman, September 1987, pp. 206-223

[Atkinson 87b]: M.P. Atkinson, O.P. Buneman, *Types and persistence in database programming languages*, ACM surveys, Vol. 19, N° 2, June 1987, pp. 106-190

[Atkinson 88]: M.P. Atkinson, Peter Buneman, R. Morrison (Eds), *Data types and persistence*, pp. 292, 1988

[Atkinson 89 a]: M.P. Atkinson, *Persistent architectures*, proc. third international workshop on persistent object systems, eds. J. Rosenberg and D. Koch, Newcastle, Australia, January 1989

[Atkinson 89 b]: M. Atkinson, *Questioning persistent types*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 2-24

[Atkinson 89 c]: M. Atkinson, F. Bancilhon, D. DeWitt, K. dittrich, D. Maier and S. Zdonik, *The object-oriented database manifesto*, proc. international conference on deductive and object databases, Kyoto, Japan, December 1989

[Bancilhon 88]: F. Bancilhon, & al., *The design and implementation of $O_2$, an object-oriented database system*, lecture notes in computer science, N°334, ed. K.R. Dittrich, Advances in object-oriented database systems, 2nd workshop on object-oriented database systems, September 1988, Springer-Verlag, pp. 1-22

[Bancilhon 89 a]: F. Bancilhon, S. Cluet, C. Delobel, *A query langage for the $O_2$ object-oriented database system*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 122-138

[Bancilhon 89 b]: F. Bancilhon, *Query langages for object-oriented database system: analysis and a proposal*, proc. of the third GI-Fachtagung "Datenbanksysteme in Buro, Technik und Wissenschaft", Informatik-Fachberichte, springer-Verlag, Zurich, Mars 1989, pp. 1-18

[Bancilhon 91a]: F. Bancilhon, S. Gamerman, *Les systèmes de gestion de bases de données*

*orientés objets*, Communications of the ACM, Vol. 34, N° 10, October 1991, pp. 25-30

[Bancilhon 91b]: F. Bancilhon and C. Delobel, *Object-oriented database systems & Recent advances in OO-DBMS*, Tutorials of the TOOLS'91 international conference, Paris, France, March 1991 slides 106 & 156

[Banerjee 87]: J. Banerjee, W. Kim, *Semantics and implementation of schema evolution in object-oriented databases*, proc. ACM SIGMOD international conference on management of data, San Francisco, California, May 1987, pp. 311-322

[Banerjee 88]: J. Banerjee, W. Kim and K.C. Kim, *Queries in object-oriented databases*, proc. 4th international conference on data engineering, Los Angeles, California, February 1988

[Batory 88]: D.S. Batory & al., *Genesis: an extensible database management system*, IEEE transaction on software engineering, Vol. 14, N°11, November 1988, also in [Zdonik 90] pp. 500-518

[Benzaken 88]: V. Benzaken, C. Delobel, J.B. Ndala, *Gestionnaires de mémoire et d'objets*, 4ème journée bases de données avancées, BD3, Benodet, Mai 1988

[Benzaken 89]: V. Benzaken, C. Delobel, *Dynamic clustering strategies in the $O_2$ object-oriented database system*, Altair technical report 34-89, August 1989, pp. 1-20

[Berre 91]: A. J. Berre and T. L. Anderson, *The hypermodel benchmark for evaluating object-oriented databases*, in [Gupta 91], pp. 75-91

[Bjornerstedt 88]: A. Bjornerstedt and S. Britts, *Avance, an object management*, proc. OOPSLA'88, San Diego, California, 1988, pp. 206-221

[Bjornerstedt 89]: A. Bjornerstedt and C. Hulten, *Version control in an object-oriented architecture*, in [Kim 89] pp. 451-486

[Bloom 87]: T. Bloom, S. Zdonik, *Issues in the design of object-oriented database programming languages*, proc. OOPSLA international conference, October 1987, pp.441-451

[Bobrow 88]: G. Bobrow & al., *The Common lisp object system specification: chapter 1 and 2*, Technical report 88-002R, X3J13 standards committee document, 1988

[Pps 90]: Parc Place Systems, *Boss user's guide*, 1990

[Bouchet 79]: P. Bouchet & al., *Procédures de reprise sur panne*, Monographies d'informatique de l'AFCET, editions Hommes et Techniques, 1979

[Boussard 90]: J.C. Boussard, C. Michel, J.P. Partouche, R. Rousseau, M. Rueher, *Une évaluation du langage EIFFEL*, TSI, Vol. 9, N°4, 1990, p 331-373

[Bretl 89]: R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, M. Williams, *The Gemstone data management system*, in [Kim 89], pp. 283-308

[Brissi 90]: P. Brissi, *Etude et réalisation d'un interprète pour le langage Eiffel*, rapport technique, Ecole Supérieure des Sciences Informatiques, Juin 1990

[Cardelli 85]: L. Cardelli and D. MacQueen, *Persistence and type abstraction*, proc. workshop on persistence and data types, Appin, August 1985, in [Atkinson 88], pp. 31-42

[Carey 86]: M.J. Carey, D.J. Dewitt, *The architecture of the EXODUS extensible DBMS*, Proc. of the international workshop on object-oriented database systems, Pacific Grove, Sept. 1986

[Carey 88]: M.J. Carey, D.J. Dewitt, S.L. Vandenberg, *A data model and Query language for EXODUS*, proc. SIGMOD conference on management of data, Vol. 17, N°3, September 1988, pp. 413-423

[Carey 90]: M.J. Carey & al., *The EXODUS extensible DBMS project: An overview*, in [Zdonik 90], pp. 474-499

[Chang 89]: E. Chang, D. Gedye and R. Katz, *The design and implementation of a version server for computer-aided design data*, software-practice and experience, Vol. 19, N°3, March 1989, pp. 199-222

[Chou 86]: H.T. Chou & al., *Design and implementation of the Wisconsin Storage System*, software - Practice and Experience, vol. 15, N°10, October 1985

[Chou 1988]: H.T. Chou and W. Kim, *Versions and change notification in an object-oriented database system*, proc. 25th ACM/EEE design automation conference, June 1988, pp. 275-281

[Cluet 89]: S. Cluet, C. Delobel, C. Lecluse, P. Richard, *Reloop, an algebra based query language for an object-oriented database system*, proc. First international conference on deductive and object-oriented databases, December 1989, Kyoto, Japan

[Cockshott 87]: W.P. Cockshott, *Persistent programming and secure data storage*, Memex Inf. Engines Ltd., Edinburgh, Scotland Inf. & Software Technology (GB), Vol. 29, No. 5 June 1987

[Cockshott 88]: W.P. Cockshott, *Adressing mechanisms and persistent programming*, in [Atkinson 88], pp. 235-252

[Cockshott 90]: W.P. Cockshott, M.P. Atkinson and K.J. Chisholm, *Persistent object management system*, in [Zdonik 90], pp. 251-272

[Codd 70]: E.F. Codd, *A relational model of data for large shared data banks*, CACM Vol. 13, N°6, June 1970

[Cohen 88]: E. Cohen, D. Soni & al., *Version management in Gypsy*, ACM software engineering notes, SIGSOFT, Vol. 13, N°5, November 1988

[Copeland 84]: G. Copeland and D. Maier, *Making Smalltalk a database system*, SIGMOD'84, proceedings of annual meeting, SIGMOD record, vol. 14, N°2, pp. 316-325, 1984

[Damon 91]: C. Damon, G. Landis, *Abstract state and representation in VBASE*, in [Gupta 91], pp. 178-198

[Dearle 89]: A. Dearle, R. Connor, F. Brown and R. Morrison, *Napier 88 - A database programming language?*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 179-195

[Deux 90]: O. Deux & al., *The story of $O_2$*, IEEE Transaction on knowledge and data engineering, March 1990

[Deux 91]: O. Deux & al., *The $O_2$ system*, Communications of the ACM, Vol. 34, N° 10, October 1991, pp. 35-48

[Ducournau 89]: R. Ducournau et M. Habib, *La multiplicité de l'héritage dans les langages à objets*, TSI, AFCET, Vol. 8 N°1 - 1989

[Eliot 89]: J. Eliot, B. Moss, *Object orientation as catalyst for language-database integration*, in [Kim 89], pp. 583-592

[Fishman 87]: D.H. Fishman & al., *Iris: An object-oriented DBMS*, ACM transaction on office information systems, Vol. 5, N°1, January 1987, pp. 48-69, also in [Zdonik 90], pp. 216-226

[Fishman 89]: D.H. Fishman & al., *Overview of the Iris DBMS*, in [Kim 89], pp. 219-250

[Ford 88]: S. Ford & al., *ZEITGEIST: database support for object-oriented programming*, lecture notes in computer science, N°334, ed. K.R. Dittrich, Advances in object-oriented database systems, 2nd workshop on object-oriented database systems, September 1988, Springer-Verlag, pp. 23-42

[Gardarin 88 a]: G. Gardarin, *Les SGBD orientés objets*, journées d'étude "base de données déductives et bases de données orientées objets", Paris, Décembre 1988, pp. 59-89

[Gardarin 88 b]: G. Gardarin, P. Pucheral & al., *GEODE: un serveur d'objets extensible pour SGBDs en mémoire primaire*, journées d'étude "base de données déductives et bases de données orientées objets", Paris, Décembre 1988,

[Garza 88]: J.F. Garza and W. Kim, *Transaction in an object-oriented database system*, proc. ACM SIGMOD international conference on management of data, chicago, Illinois, June 1988, pp. 37-45

[Giavitto 88]: J. L. Giavitto, A. Devarenne, G. Rosuel, *PRESTO: des objets C++ persistants pour "Adage"*, journées d'étude "base de données déductives et bases de données orientées objets", Paris, Décembre 1988, pp. 41-56

[Godart 91]: C. Godart et F. Charoy, *Bases de données pour le génie logiciel*, Masson, 1991, pp. 166

[Goldberg 83]: A. Goldberg & D. Robson, *Smalltalk-80: the language and its implementation*, addyson-wesley, london, 1983

[Goldberg 89]: A. Goldberg & D. Robson, *Smalltalk-80: the language*, addyson-wesley, series in computer science, 1989

[Gordon 79]: M.J. Gordon, A.J. Milner, and C.P. Wadsworth, Edinburgh LCF, lecture notes in computer science, Vol. 78, Springer-verlag, New-york, 1979

[Griswold 90]: R.E. Griswold & M.T. Griswold, *The ICON programming language*, Prentice-hall Inc., Second edition, 1990

[Gupta 91]: R. Gupta and E. Horowitz, *Object-oriented databases with applications to case, networks, and VLSI CAD*, eds R. Gupta and E. Horowitz, Prentice Hall series in DKB, 1991

[Harris 91]: C. Harris, J. Duhl, *Object SQL*, in [Gupta 91], pp. 199-215

[Heuer 90]: A. Heuer, J. Fuchs, and U. Wiebking, *OSCAR: an object-oriented database system with a nested relational kernel*, proc. 9th international conference on entity-relationship approach, Lausanne, October 1990, pp. 95-110

[Heuer 91]: A. Heuer, M. Scholl, *Principles of object-oriented query language*, Technical

report, 1991, Inst. fûr Informatik, TU Clausthal, Erzstr. 1, W-3392 Clausthal-zellerfeld, germany

[Horowitz 91]: E. Horowitz and Q. Wan, *An overview of existing object-oriented database systems*, in [Gupta 91], pp. 101-116

[Ichbiah 79]: J.H. Ichbiah & al., *Rationale of the design of the programming language Ada*, Sigplan notices, ACM, Vol. 4, N°6, 1979

[Kaehler 81]: T. Kaehler, *LOOM - large object-oriented memory for smalltalk systems*, in [goldberg 83], pp. 251-269

[Kaehler 90]: T. Kaehler, G. Krasner, *LOOM - large object-oriented memory for Smalltalk-80 systems*, in [Zdonik 90], pp. 298-307

[Katz 87]: R.H. Katz and E. Chang, *Managing change in a computer-aided design database*, proc. 13th VLDB conference, Brighton 1987, pp. 455-462

[Kim 87]: W. Kim & al., *Composite object support in an object oriented database system*, proc. OOPSLA'87 international conference, Orlando, Florida, October 1987, pp. 118-125

[Kim 88 a]: W. Kim and H.T. Chou, *Versions of schema for object-oriented databases*, proc. 14th VLDB international conference, Los Angeles, California, August-September 1988, pp. 148-159

[Kim 88 b]: W. Kim, N. Ballou, H.T. Chou, J.F. Garza, *Integrating an object-oriented programming system with a database system*, proceedings of OOPSLA'88 conference, ACM, september 1988,

[Kim 89a]: W. Kim, F.H. Lochovsky, *Object-oriented concepts, databases, and applications*, eds W. Kim,, ACM press, Frontier series, September 1989,

[Kim 89b]: W. Kim, E. Bertino, J.F. Garza, *Composite objects revisted*, proc. ACM SIGMOD international conference, on management of data, Portland, Oregon, June 1989, pp. 337-347

[Kim 90a]: W. Kim, *Architecture of the Orion next-generation database system*, IEEE Transaction on knowledge and data engineering, March 1990

[Kim 90b]: W. Kim, *Architectural issues in object-oriented databases*, Journal of object-oriented programming, March / April 1990, pp. 29-38

[Kim 91]: W. Kim, *Introduction to Object-Oriented databases*, MIT press, Cambridge, Massachusets, London, England, 1991

[KimK 91]: K. C. Kim, *Performance of query optimization heuristics in object-oriented databases*, proc. international symposium on database systems for advanced applications, tokyo, Japan, April 1991, pp. 99-108

[Knut 73]: D.E. Knut, *searching and sorting*, Addison Wesley, or the art of computer programing Vol. 3, 1973

[Lahire 90]: P. Lahire, N. Le Than, S. Miranda, *Persistency for the EIFFEL language*, TOOLS'90 International conference, Paris, June 1990

[Lahire 91]: P. Lahire & P. Brissi, *An integrated query language for handling persistent objects in Eiffel*, TOOLS'91 international conference, Paris, France, March 1991

[Lahire & al. 91]: P. Lahire, R. Rousseau, N. Le Thanh, J. Farré, R. Chignoli, *Spécification and design of a model for an Eiffel object repository*, Report BC.D.LIS.T41.1, of the Business-class Esprit II project, August 1991, I3S Sophia Antipolis, pp 167

[Lamb 91]: C. Lamb, G. Landis, J. Orenstein, D. Weinreb, *The objectstore database system*, Communications of the ACM, Vol. 34, N° 10, October 1991, pp. 50-63

[Lecluse 88]: C. Lecluse, P. Richard, and F. Velez, *$O_2$, an object-oriented data model*, proc. ACM SIGMOD international conference on management of data, Chicago, illinois, June 1988, pp. 424-433, also in [Zdonik 90]

[Lecluse 88b]: C. Lecluse, P. Richard, F. Velez, $O_2$ *un modèle de données orienté objets*, 4ème journée base de données avancées, BD3, Benodet, Mai 1988

[Lecluse 89a]: C. Lecluse, P. Richard, *Langages orienté-objet et base de données: l'expérience* $O_2$, 5ème journée base de données avancées, BD3, Benodet, Mai 1989

[Lecluse 89b]: C. Lecluse, P. Richard, *Modeling complex structures in object-oriented*, proc. 8th ACM PODS 1989, pp. 360-368

[Liskov 77]: B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, *Abstraction mechanisms in CLU*, comm. ACM, vol. 20, N°8, August 1977, aussi dans [Zdonik 90]

[Liskov 81]: B. Liskov, R. Atkinson, T. Bloom & al., *CLU reference manual*, lecture notes in computer science, n° 114, springer-Verlag, pp. 190

[Loizou 91]: G. Loizou and P. Pouyioutas, *A query algebra for an extended object-oriented database model*, proc. international symposium on database systems for advanced applications, tokyo, Japan, April 1991, pp. 89-98

[Maier 86 a]: D. Maier, J. Stein, Development and implementation of an object-oriented DBMS, proc. OOPSLA'86 international conference, September 1986, pp. 472-482, also in [Zdonik 90], pp. 167-185

[Maier 86 b]: D. Maier, J. Stein, *Indexing in an object-oriented DBMS*, proc. international workshop on database systems, languages, Pacific Grove, California, september 1989

[Maier 87]: D. Maier, *Why database languages are a bad idea*, proc. international workshop on database programming languages, Roscoff, France, September 1987, pp. 277-287

[Maier 90]: D. Maier and S.B. Zdonik, *Fundamentals of object-oriented databases*, in [Zdonik 90], pp. 1-32

[Masini 89]: G. Masini & al., *Les langages à objets: langage de classes, langage de frames, langage d'acteurs*, Interéditions, 1989, pp 584

[Mellender 89]: F. Mellender, S. Riegel, A. Straw, *Optimizing smalltalk message performance*, in [Kim 89], pp. 423-450

[Meyer 88]: B. Meyer, *Object-oriented software construction*, prentice-hall, series in computer sciences, 1988

[Meyer 90]: B. Meyer, *The Eiffel language: Libraries*, Technical report, Interactive Software

Engineering, California, USA

[Meyer 91]: B. Meyer, *Eiffel: The language*, prentice-hall, series in computer sciences, 1991

[Miranda 90]: S. Miranda, *l'art des bases de données: comprendre et évaluer SQL*, Eyrolles, 1990

[Miranda 90 b]: S. Miranda, *DBMS future*, invited paper, INCOM conference, Daejong, Korea, November 1990

[Miranda 92 a]: S. Miranda, G. Gardarin, P. Valduriez, E. Penny, *Next generation DBMS*, cours Europace, April 92

[Miranda 92 b]: S. Miranda, *Semantically extended relational DBMS*: *a formal way to handle O2 - data bases*, 2nd international conference, Kioto, April 92

[Mopolo-Moke 91]: G. Mopolo-Moke, *NICE-C++: une extension C++ pour la programmation persistante à partir d'un serveur de base d'objets*, Thèse de doctorat, spécialité informatique, I3S, Université de Nice Sophia-Antipolis, 1991

[Morrison 89]: R. Morrison, F. Brown, R. Carrick, R. Connor, A. Dearle and M.P. Atkinson, *The Napier type system*, proc. of the third international workshop on persistent object systems, Newcastle, Australia, January 1989, eds. J. Rosenberg and D. Koch, Springler-Verlag, pp. 3-18

[Morrison 91]: R. Morrison, A. Dearle, R. Connor and A. Brown, *An ad hoc approach to the implementation of polymorphism*, Technical report series, Fide project, university of glasgow, September 1991 pp. 1-30

[Mylopoulos 80]: J. Mylopoulos, P.A. Berstein, and H.K.T Wong, *a language facility for designing database intensive applications*, ACM Transaction database systems, Vol. 5, N°2, June, 1980, pp. 185-207

[Nguyen 89]: G.T. Nguyen, D. Rieu, *Schema change propagation in object oriented databases*, information processing 89, North-Holland 1989, pp. 815-820

[O'Brien 86]: P. O'Brien, B. Bullis, and C. Schaffert, *Persistent and shared objects in Trellis/Owl*, IEEE, 1986, pp. 113-122

[O'Brien 88]: P. O'Brien, *Common object-oriented repository system*, Lecture notes in computer science 334, advances in object-oriented database systems, proc. 2nd international workshop on object-oriented database systems, September 1988, pp. 329-334

[O2-Technology 91 a]: O2-Technology, *The O2programmer's manual*, version 3.0, July 1991, Versailles, France

O2-Technology 91 b]: O2-Technology, *The O2-Engine programmer's manua*l, version3.1. July 1991, Versailles, France

[Paepcke 88]: A. Paepcke, *PCLOS: A flexible implementation of CLOS persistence*, proc. European conference on object oriented programming 1988, eds S. Gjessing and K. Nygaard, lecture notes in computer science, springer-verlag, berlin, pp. 374-389

[Paepcke 89]: A. Paepcke, *PCLOS: A critical review*, proc. OOPSLA international conference, 1989, pp. 221-237

[Penney 87 a]: J. Penney, J. Stein, and D. Maier, *Is the disk half full or half empty?: combining optimistic and pessimistic concurrency mechanisms in a shared, persistent object base,* proc. workshop on persistent object stores, Appin, Scotland, August 1987

[Penney 87 b]: J. Penney, J. Stein, *Class modification in the Gemstone object-oriented DBMS*, proc. OOPSLA'87 international conference, Orlando, Florida, October 1987, pp. 111-117

[Philbrow 89]: P. Philbrow, D. Harper and M. Atkinson, *Supporting an object-oriented programming methodology using PS-Algol*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 61-79

[Purdy 91]: A. Purdy, B. Schuchardt and D. Maier, *Integrating an object server with other worlds*, in [Gupta 91], pp. 101-116

[Richardson 87]: J.E. Richardson, M.J. Carey, *Programming constructs for database system implementation in EXODUS*, SIGMOD record, vol. 16, N°3, December 1987

[Richardson 89 a]: J.E. Richardson and M.J. Carey, *Implementing persistence in E*, proc. of the third international workshop on persistent object systems, Newcastle, Australia, January 1989, eds. J. Rosenberg and D. Koch, Springler-Verlag, pp. 175-199

[Richardson 89 b]: J.E. Richardson and M.J. Carey, *Persistence in the E language: Issues and implementation*, software - practice and experience, Vol. 19, N°12, December 1989, pp. 1115-1150

[Rowe 87]: L. Rowe and M. Stonebraker, *The Postgres data model*, proc. 13th VLDB conference, Brighton, England, sept. 1987, also in [Zdonik 88] pp. 461-473

[Salgado 88]: A.C. Salgado, *Contribution a un SGBD orienté objet (NICEBD): traitement des données et des interfaces multimédia*, Thèse de docteur en science mention informatique, I3S, Université de Nice - Sophia Antipolis, Octobre 1988

[Schlageter 88]: G. Schlageter, R. Unland & al., *OOPS - an object oriented programming system with integrated data management facility*, pp. 118-125

[Scholl 90]: M.H. Scholl and H-J. Scheck, *A relational object mode*l, proc. international conference on database theory, Paris, Springer, LNCS, December 1990

[Servio 89]: Servio-logic Dev. Corp, *programming in OPAL*, Beavarton, Oregon, April 1989

[Schaffert 86]: C. Shaffert & al., *An introduction to Trellis/Owl*, proc. OOPSLA'86 int. conference, Portland, Oregon, 1986

[Shaw 81]: M. Shaw, *Alphard form and content*, eds M. Schaw, Springer-Verlag, Berlin 1981

[Shaw 89]: G.M. Shaw, S.B. Zdonik, *An object-oriented query algebra*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 103-112

[Shipman 81]: D.W. Shipman, *the functional data model and the data language DAPLEX*, ACM Trans. database Syst. Vol. 6, N°1, March 1981, pp. 140-173

[Silberschatz 91]: A. Silberschatz, M. Stonebraker, J. Ullman, *Database systems: achievements and opportunities*, Communications of the ACM, Vol. 34, N° 10, October 1991, pp. 110-120

[Simmel 91]: S. Simmel, I. Godard, *The KALA basket - a semantic primitive unifying object transactions, acess control, versions and configurations*, proc. OOPSLA'91, September 91, pp. 230-246

[Skarra 86]: A. Skarra and S. Zdonik, *The management of changing types in an object-oriented database*, proc. OOPSLA'86 international conference, Portland, Oregon, October 1986

[Skarra 87]: A. Skarra and S. Zdonik, *Type evolution in an object-oriented database*, proc. 1987

[Skarra 89]: A. Skarra and S. Zdonik, *Concurrency control and object-oriented databases*, in [Kim 89], pp. 395-422

[Smith 83]: J.M. Smith, S. Fox, and T. Landers, *ADAPLEX: rationale and reference manual*, 2nd ed. Computer corporation of America, Cambridge, Massachussets, 1983

[Stonebraker 90]: M. Stonebraker, L.A. Rowe, and M. Hirohama, *The implementation of postgres*, IEEE transactions on knowledge and data engineering, Vol. 2 N°1, March 1990, pp. 125-141

[Stonebraker 91]: M. Stonebraker, G. Kemnitz, *The postgres next-generation database management system*, Communications of the ACM, Vol. 34, N° 10, October 1991, pp. 78-92

[Straw 89]: A. Straw, F. Mellender and S. Riegel, *Object management in a persistent Smalltalk system*, Software - practice and experience, vol. 19 N° 8, August 1989, pp. 719-737

[Tarr 89]: P.L. Tarr, J.C. Wileden, A.L. Wolf, *A different tack to providing persistence in a language*, proc. of the 2nd workshop on database programming languages, eds R. Hull, R. Morrison, D. Stemple, Morgan Kaufmann publishers Inc., 1989, pp. 41-60

[Vossen 91]: G. Vossen, *Bibliography on object-oriented database management*, Sigmod reccord, Vol. 20, N°1, March 1991, pp. 25-47

[Weinreb 91]: D. Weinreb, N. Feinberg, D. Gerson and C. Lamb, *An object-oriented database system to support an integrated programming environment*, in [Gupta 91], pp. 117-129

[Weiser 89]: S.P. Weiser, F.H. Lochovsky, *OZ+: an object-oriented database system* in [Kim 89], pp. 309-337

[Wileden 88]: J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr, *PGRAPHITE: an experiment in persistent typed object management*, ACM Press, SIGSOFT'88: third symposium on software development environments, Boston, Massachusetts, Nov. 1988, pp. 130-142

[Zdonik 86 a]: S.B. Zdonik, P. Wegner, *Language and methodology for object-oriented database environments*, proc. 19th Annual Hawaii international conference on system sciences, January 1986, pp. 155-171

[Zdonik 86 b]: S.B. Zdonik, *Maintening consistency in a database with changing types*, Object-oriented programming workshop, SIGPLAN Notice (USA), Vol. 21, N°10, June 1986, Yorkton Heights, NY

[Zdonik 90]: S.B. Zdonik, D. Maier, *readings in object-oriented database systems*, eds. S.B. Zdonik and D. Maier, 1990

# Table des figures

# Table des matières