Modèle et langage à objets pour la programmation d'applications réparties

Marie Laurence Meysembourg-Männlein

THESE

présentée par

Marie Laurence MEYSEMBOURG - MÄNNLEIN

pour obtenir le titre de

DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE
de GRENOBLE
(arrêté ministériel du 5 juillet 1984)

Spécialité : **INFORMATIQUE**

MODELE et LANGAGE à OBJETS pour la PROGRAMMATION d'APPLICATIONS REPARTIES

Soutenue le 5 juillet 1989
devant la Commission d'Examen composée de

Sacha Krakowiak	Président
Françoise André	Rapporteurs
Jean Bézivin	
Jacques Mossière	Directeur de thèse
Roland Balter	

Thèse préparée au sein du Laboratoire de Génie Informatique

TITRE DE LA THESE EN FRANÇAIS

Modèle et Langage à Objets pour la Programmation d'Applications Réparties

RÉSUMÉ DE LA THESE EN FRANÇAIS

Cette thèse a été effectuée dans le cadre du projet Guide mené conjointement par le Laboratoire de Génie Informatique et le Centre de Recherche Bull de Grenoble depuis mi 86. Guide est le support d'un ensemble de recherches sur la programmation des applications réparties. Ces recherches sont entreprises sur la base du développement d'un système expérimental : le système Guide. Ce dernier est un système d'exploitation réparti à objets qui fournit un haut niveau d'intégration (invisibilité de la répartition notamment). Un premier prototype du système fonctionne depuis fin 88 et permet la programmation et la mise en œuvre d'applications par l'intermédiaire d'un langage de programmation spécifique. La thèse contient une présentation et une évaluation du modèle de programmation par objets mis en œuvre par ce langage : la présentation met en évidence les principaux choix de conception et les justifie ; l'évaluation est basée sur une étude comparative des modèles d'objets de Guide, de Trellis/Owl, d'Emerald et d'Eiffel, et sur des expériences de programmation réalisées sur le prototype. Elle met en évidence les aspects caractéristiques du modèle d'objets de Guide, ses apports et ses limites.

TITRE DE LA THESE EN ANGLAIS

Object Model and Language for Distributed Programming

RÉSUMÉ DE LA THESE EN ANGLAIS

This thesis was prepared within the Guide project - a collaboration underway since summer 1986 between le Laboratoire de Génie Informatique and le Centre de Recherche Bull de Grenoble. The Guide project is the framework for research in the area of distributed applications and is based on the development of an experimental operating system. The system is distributed and object oriented; its provides a high degree of integration, and especially location-independant remote execution. An initial prototype, which became operational at the end of 1988, allows applications to be written and executed in the Guide programming language. This thesis contains a presentation and evaluation of the object programming model embodied in this language. The presentation describes and explains the design decisions. The evaluation is based on a comparative study of the object models proposed in the Guide, Trellis/Owl, Emerald and Eiffel languages and on the experience gained in programming the Guide prototype. The characteristics of the object model of Guide are considered, together with its advantages and constraints.

MOTS CLES : objet, classe, héritage, conformité, répartition, persistance

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S

Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G
CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

E.F.P.
CHARUEL Robert

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Je tiens à remercier

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourrier de Grenoble et responsable du projet Guide, pour la grande confiance qu'il m'a faite tout au long des trois années pendant lesquelles j'ai travaillé à ce projet : il m'a vraiment donné les moyens de faire un travail très intéressant. Je le remercie également pour l'honneur qu'il m'a fait en acceptant la présidence du jury de cette thèse.

Madame Françoise André, Professeur à l'Université de Rennes I et Monsieur Jean Bézivin, Professeur à l'Université de Nantes, qui ont accepté d'être rapporteurs de mon travail. Je les remercie particulièrement pour l'intérêt qu'ils lui ont porté et pour l'évaluation qu'ils en ont fait.

Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, pour l'aide précieuse qu'il m'a apporté lors de la conception et de la rédaction de ce document.

Monsieur Roland Balter, Directeur du centre de Recherches Bull de Grenoble, qui a accepté de faire partie du jury de cette thèse. Je le remercie également pour les discussions très intéressantes que nous avons eues.

Le Ministère de la Recherche et de la Technologie et la Communauté Economique Européenne qui ont financé mon travail.

Je tiens à remercier également l'ensemble des membres du projet Guide : le travail que je présente ici est en effet le fruit d'une recherche collective.

Merci également à tous pour la bonne ambiance qui règnait dans l'équipe, et merci plus particulièrement à Xavier pour sa fougue, Dominique pour son impertubabilité, Michel pour son détachement, Hiep pour sa discrétion, Roland pour sa constante bonne humeur, Eric pour sa fièvre, André pour son calme, Hervé pour son sourire ...

MODELE ET LANGAGE À OBJETS POUR LA PROGRAMMATION D'APPLICATIONS RÉPARTIES

Plan de la thèse

CHAPITRE I

INTRODUCTION

1. PRÉSENTATION DES OBJECTIFS DE LA THESE

Le travail de recherche que j'ai effectué au Laboratoire de Génie Informatique s'est déroulé dans le cadre de la conception et de la mise en œuvre du premier prototype du système Guide (Environnement Distribué et Intégré des Universités de Grenoble).

Le projet Guide est un projet de grande ampleur dont les objectifs principaux sont la conception et la réalisation d'un système d'exploitation expérimental pour un ensemble de postes de travail individuels et de serveurs interconnectés par un réseau local. Il est le support d'un ensemble de recherches sur la conception et la réalisation des systèmes informatiques répartis. Les thèmes scientifiques abordés couvrent les domaines suivants :

- les principes de conception et les techniques de réalisation des systèmes d'exploitation répartis : conservation et désignation de l'information, allocation de ressources, synchronisation des activités, résistance aux défaillances ;
- les méthodes de structuration et de programmation des applications réparties ; les modèles à objets pour l'expression de ces applications ;
- la conception d'interfaces homme-machine et son application aux systèmes répartis ; les méthodes et les outils pour la gestion d'objets structurés, notamment les documents et les images.

Les recherches faites dans ces différents thèmes sont menées avec le souci de mettre au point une réalisation en vraie grandeur : un premier prototype du système Guide fonctionne depuis octobre 1988 sur un réseau hétérogène constitué de machines MS-3 Matra-DataSystème (Sun-3) et de SPS-7 Bull reliées par un cable Ethernet à 10 Mbit/s.

Deux années ont été nécessaires à la conception et à la réalisation de ce premier prototype du système Guide. Durant cette première phase de travail, je me suis surtout intéressée aux aspects suivants :

- la définition du modèle de programmation,
- la définition du langage associé,
- la définition du schéma de traduction de ce langage en code exécutable,
- l'utilisation du modèle et du langage pour la programmation d'applications.

Depuis la mise en œuvre du premier prototype a commencé un travail d'expérimentation et d'évaluation du modèle de programmation et du langage. Cette thèse s'inscrit dans ce processus d'expérimentation et d'évaluation. L'objectif que j'ai poursuivi en la rédigeant est double :

- Présenter et motiover les choix de conception et de réalisation du modèle d'objets de Guide.
- Evaluer la première version du modèle et tirer de cette évaluation un ensemble de propositions pour la réalisation d'une deuxième version.

Les différents chapitres qui constituent la thèse sont organisés conformément à ces objectifs. Avant d'en détailler le contenu dans la section 3, je donne dans la section suivante une rapide description des principaux choix de conception du système Guide.

2. Principaux choix de conception du système Guide

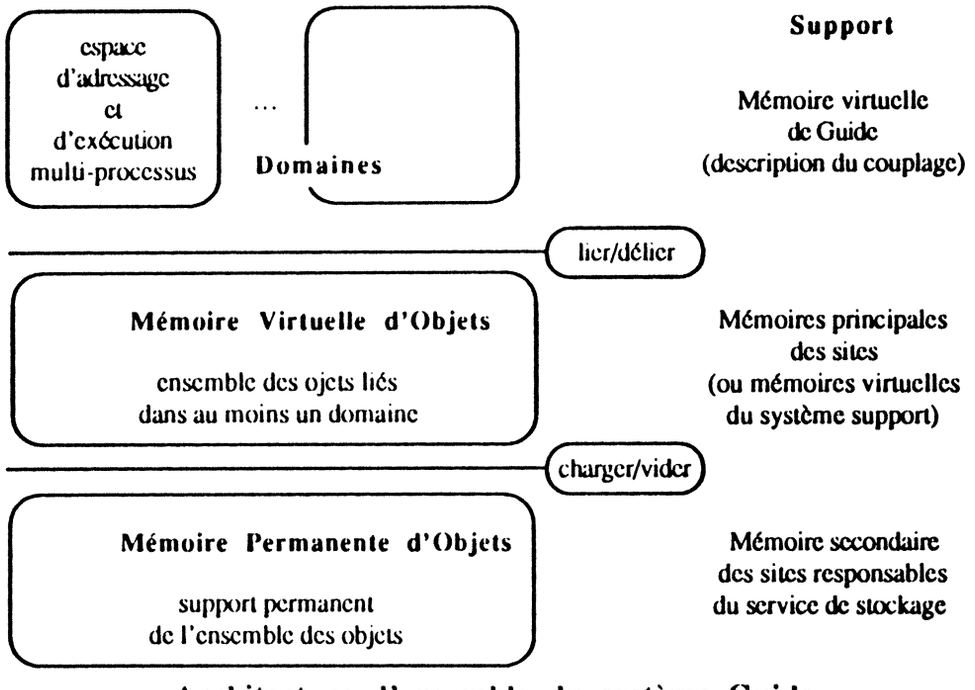
Les principaux choix de conception du système Guide ont été arrêtés en 1987. Ils sont présentés en détail dans le rapport [Guide R1] et seul un rappel des éléments nécessaires à la suite de notre propos est fait ci-dessous.

Guide a pour vocation d'être un système à haut degré d'intégration, dans lequel chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti. L'utilisateur doit y être déchargé de la nécessité de connaître la localisation des ressources qu'il demande ou le détail des protocoles de communication permettant d'y accéder.

Ce souci d'intégration a conduit à la définition d'un modèle d'exécution et de conservation aussi indépendant que possible de la réalisation physique et de la répartition ; le choix d'un modèle à objets comme structure de base du système a notamment facilité la définition abstraite de ses composants.

Le système Guide est présenté comme une machine virtuelle multi-processus ; la visibilité du parallélisme qui permet la gestion d'activités multiples simultanées offre un outil d'exploitation de la répartition, mais aussi de la puissance des postes de travail modernes, éventuellement multi-processeurs.

Objets et processus sont dissociés dans le souci de garder une structure légère tant pour les uns que pour les autres ; une unité d'exécution, c'est à dire une application, est constituée d'un ensemble d'objets passifs et d'un ensemble de processus opérant sur ces objets, la composition de ces deux ensembles évoluant dynamiquement. Une unité d'exécution est appelée **domaine** et le nom d'**activités** est donné aux processus [Guide R3].



Architecture d'ensemble du système Guide

La figure précédente décrit l'architecture générale du système Guide. Dans ce dernier, les objets sont gérés et conservés dans un espace universel auquel les domaines accèdent par couplage ; cet espace est conçu en deux niveaux : le niveau supérieur, ou Mémoire Virtuelle d'Objets (MVO) et le niveau inférieur, ou Mémoire Permanente d'Objets (MPO). La **Mémoire Virtuelle d'Objets** est le support des objets liés dans au moins un domaine ; la **Mémoire Permanente d'Objets** est le support des objets dont la durée de vie n'est pas liée à celle d'un domaine ou d'une activité. Le choix d'un espace d'objets à deux niveaux permet d'isoler la partie du système chargée de la conservation

permanente des objets, qui joue ainsi le rôle d'un service réparti réalisé par l'ensemble des sites pourvus d'une mémoire secondaire réservée au stockage [Guide R4].

Un des choix de conception les plus importants a été celui de l'adoption d'un modèle à objets comme structure de base du système et notamment pour la description des applications destinées à s'exécuter dans l'environnement Guide.

La programmation par objets se caractérise par une organisation de l'information en entités appelées objets. Chaque objet regroupe un ensemble de données, ou variables d'état, et un ensemble de procédures d'accès à ces données, appelées opérations ou méthodes. Tout objet non élémentaire est construit par composition d'autres objets ; on peut ainsi définir des compositions d'objets de différents niveaux de complexité. L'état d'un objet ne peut être consulté ou modifié que par appel de ses opérations. La notion de classe permet de spécifier des opérations et une organisation de données communes à un ensemble d'objets, les exemplaires de la classe, ainsi qu'un mécanisme de génération de ces exemplaires.

Les motivations propres au choix d'un modèle à objets pour la programmation des applications dans Guide sont les suivantes :

- Un modèle à objets fournit un outil de structuration et de construction modulaire du logiciel.
- Les principes de séparation entre interface et réalisation sont directement applicables et le passage obligatoire par une interface pour l'accès à un objet facilite la mise en œuvre de mécanismes de protection.
- La possibilité de disposer de réalisations multiples d'une même interface facilite la construction de systèmes hétérogènes.
- La possibilité de substituer dynamiquement une réalisation à une autre facilite la conception de mécanismes de reconfiguration et de reprise après erreur ou incident, ainsi que la mise au point de systèmes expérimentaux.
- Un système de répartition dynamique de charge peut être réalisé par la migration d'objets, considérés pour cela comme les unités de distribution.

Un choix important dans la conception du modèle a été celui de la définition d'un langage spécifique pour sa mise en œuvre. Cette décision a été prise dans le souci de rendre plus rigoureux le processus de conception et de développement des applications Guide et de faciliter leur intégration au système d'exploitation. En effet, l'exploitation d'un langage existant comme base du langage de programmation Guide paraîssait moins judicieuse compte tenu des spécificités du modèle d'objets (abstraction de données,

contrôle statique) et du modèle d'exécution (chargement statique du code en Mémoire Permanente d'Objets, liaison dynamique du code en mémoire d'exécution).

Une première version du compilateur Guide permet actuellement la génération d'applications pour le premier prototype du système ; compte tenu du peu de moyens humains disponibles et dans un souci de mise en œuvre rapide de ce prototype, le travail de génération du code exécutable a été limité à la production d'un code de haut niveau ; notre choix s'est porté sur le langage C pour deux raisons : la puissance et la souplesse de ce langage ainsi que la disponibilité d'un compilateur sur les machines destinées au développement du système Guide.

Participation au projet ESPRIT Comandos

Les travaux menés dans Guide l'ont été en partie dans le cadre du projet ESPRIT n°834, Comandos (Construction and Managment of Distributed Office Systems). Le travail de conception du modèle d'objets de Guide a notamment beaucoup bénéficié de cette interaction. Ce modèle tel qu'il est défini aujourd'hui est un sous-ensemble de celui défini dans Comandos : le souci de l'équipe grenobloise a été de restreindre l'ensemble des concepts dans l'objectif d'obtenir un modèle minimal et cohérent.

La différence essentielle entre les modèles d'objets de Guide et de Comandos est la prise en compte dans le second de concepts issus des bases de données. Dans sa première phase de conception, le problème de l'intégration à Guide de ce type de fonctions a uniquement fait l'objet d'une réflexion préliminaire. Cette réflexion est poursuivie actuellement de façon approfondie. L'objectif visé est de fournir sur le deuxième prototype du système une version du langage de programmation Guide intégrant notamment les notions de collection d'objets et de recherche sélective d'objets par leur contenu.

3. PRÉSENTATION DU PLAN DE LA THESE

Chapitre II. Je m'intéresse dans ce chapitre aux fondements des techniques de programmation par objets. Une étude des principes qu'elles préconisent m'a semblée intéressante car leur application est encore récente et relativement peu d'expériences en sont issues. Plus précisément, l'étude que je propose est constituée d'une présentation générale des concepts de la programmation par objets et d'une analyse des avantages que leur application représente.

Chapitre III. Ce chapitre constitue la partie essentielle de cette thèse. Il contient une présentation détaillée du modèle d'objets de Guide et décrit les motivations des principaux choix de conception. Il contient également une première évaluation du modèle faite sur la base d'une comparaison entre le modèle Guide et trois autres modèles d'objets : Emerald [Black 86a, 86b], Trellis/Owl [O'Brien 85a, 85b] et Eiffel [Meyer 88]. Plusieurs propositions sont dégagées de cette étude comparative.

Chapitre IV. Ce chapitre complète et approfondit la présentation du modèle faite en III : il décrit deux applications réalisées sur le premier prototype du système. L'expérience de programmation qu'il relate est instructive car elle a permis d'évaluer les apports et les limitations du modèle ainsi que sa réalisation. L'intérêt de ce chapitre réside également dans l'illustration de l'exploitation dans un programme d'application des différents éléments définis dans le modèle.

Chapitre V. Ce chapitre décrit le processus de mise en œuvre d'une application : compilation et exécution. Son intérêt n'est pas essentiel au regard de l'objectif que je me suis fixé, mais il donne une idée précise de la chaîne de production d'une application et met en évidence les particularités du système d'exécution telle que la répartition. Cette mise en évidence me paraît importante car la répartition a dû être prise en compte dans la définition du modèle, bien qu'elle ne soit pas visible au niveau des applications.

Chapitre VI. Ce sixième et dernier chapitre de la thèse en constitue la conclusion. Il rappelle les résultats obtenus, synthétise mon évaluation du modèle d'objets de Guide et reprend les propositions les plus importantes que je fais dans les chapitres précédents.

Annexes. Trois annexes complètent le manuscrit. La première décrit les éléments de la bibliothèque Guide nécessaires à la compréhension des exemples de programmation donnés au chapitre IV. La seconde décrit la mise en œuvre des mécanismes de contrôle d'accès aux objets partagés ; elle peut être considérée comme un complément au chapitre V. Enfin, la troisième annexe décrit les possibilités d'intégration de logiciels existants dans les applications Guide. Elle met en évidence la spécificité du modèle d'exécution et les contraintes qu'elle impose dans les relations entre Guide et les autres mondes.

CHAPITRE II

LA PROGRAMMATION PAR OBJETS
Présentation des concepts

1. DES PROCÉDURES AUX OBJETS

Depuis la mise en service des premiers ordinateurs, les programmeurs conçoivent et expérimentent des méthodes d'analyse et de production du logiciel dans le but de faciliter leur travail et d'améliorer la qualité de leurs produits. La définition des langages de programmation dits de haut niveau s'inscrit dans cette recherche constante de confort et de sécurité et les différentes méthodes de programmation proposées par ces langages illustrent l'évolution des techniques de programmation vers une production de code d'une qualité toujours meilleure. La méthode de programmation par objets apparaît comme une étape dans cette évolution et l'objectif de ce chapitre est de montrer en quoi cette méthode apporte une contribution significative à la recherche de qualité en production de logiciel.

L'utilisation d'un outil, quel qu'il soit, conduit généralement à une évaluation de ses possibilités et de de ses limites relativement à l'usage qu'on veut en faire. Il n'en est pas autrement des outils de programmation que sont les langages, et l'évolution des techniques de programmation est ainsi née de l'expérience acquise par les programmeurs.

Cette expérience et le constat qu'elle entraîne conduisent généralement à l'élaboration de conventions de programmation qui peu à peu amènent le programmeur à la définition d'un nouveau langage dans lequel elles sont intégrées. C'est de cette manière qu'ont été définies puis intégrées dans des langages les techniques de programmation procédurale, modulaire et par objets.

On peut remarquer que le souci constant des programmeurs est d'utiliser des techniques leur offrant toujours un degré supérieur de décomposition et d'abstraction. La

décomposition facilite l'analyse d'un problème et en clarifie les solutions, alors que l'**abstraction** conduit à une plus grande réutilisabilité du code.

Le premier outil de décomposition fournit par les langages a été la **procédure**. Elle offre un premier niveau de structuration et de réutilisation. Elle a permis la mise en œuvre de bibliothèques destinées par exemple au traitement des tables, des fichiers, des fonctions mathématiques.

Le **module**, qui constitue un second outil de décomposition, permet d'associer une structure de données à l'ensemble des procédures destinées à son traitement. La représentation de la structure peut être encapsulée à l'intérieur du module et rendue invisible à ses utilisateurs qui ne peuvent l'utiliser que par l'intermédiaire des procédures qui lui sont applicables. Le module permet ainsi la représentation de données sous une forme asbtraite et une plus grande indépendance entre les différents composants d'un programme.

Utilisés en tant qu'unités de structuration, les modules offrent la possibilité d'isoler et de cacher les détails. Cette propriété est utilisée sous diverses formes dont les deux suivantes sont les plus courantes [Wirth 84] :

- le module contient un ensemble de données dont il cache la représentation en ne permettant leur accès que par l'intermédiaire de procédures qu'il exporte,
- le module exporte un type de données et les procédures de traitement qui lui sont associées.

La seconde forme d'utilisation est une généralisation de la première qui permet par exemple de réaliser un module représentant une pile. Lorsque plusieurs piles similaires sont nécessaires dans une même application, la seconde forme permet de concevoir le module comme un gestionnaire de piles, et de définir ainsi autant de piles que nécessaire par l'intermédiaire du type exporté.

Le concept d'**objet** est issu de cette deuxième forme d'utilisation des modules : un objet n'est pas autre chose qu'un ensemble de données dont l'utilisation n'est possible que par l'intermédiaire d'un ensemble de procédures spécifiques. Ces procédures constituent la partie visible ou **interface** de l'objet. Programmer en termes d'objets avec un langage modulaire est donc possible, mais le choix de l'objet comme unique unité de structuration permet de fournir aux programmeurs des possibilités nouvelles, l'uniformité du modèle permettant une meilleure exploitation du degré d'abstraction mis en œuvre.

D'une manière très générale, un programme peut être vu comme une suite d'actions destinées à effectuer un certain traitement sur un certain ensemble de données, le traitement étant décrit par une suite d'instructions et les données par un ensemble de variables. Les variables permettent de désigner les données dans les instructions qui les traitent. Soit le **domaine de définition d'une variable** l'ensemble des données que cette variable peut désigner dans un programme. L'application du principe d'abstraction qui consiste à séparer la spécification du comportement d'un objet des détails de sa réalisation, permet la définition d'un modèle de variables plus souple et donc plus intéressant que celui des modèles de programmation classiques[1].

Les domaines de définition des variables dans les modèles à objets sont en effet définis comme un ensemble d'objets qui partagent un même comportement plutôt qu'une même représentation physique. De cette façon, deux objets piles qui se comportent de la même manière mais qui sont réalisés différemment (l'un par un tableau et l'autre par une liste par exemple) peuvent être désignés par une même variable. La surcharge de noms (déjà possible dans Ada, cf 3.4) apparaît maintenant comme intrinsèque au modèle mais doit être résolue dynamiquement : le code d'une procédure appelée sur un objet dépend en effet de la réalisation de ce dernier, et ne peut être déterminé qu'à l'exécution puisque l'objet est désigné par une variable définie indépendamment de sa réalisation. Cette remarque met en évidence la contrepartie de la souplesse qu'apporte la mise en œuvre du principe d'abstraction : la perte d'efficacité due à la **liaison dynamique**[2] qu'elle impose.

Cette dualité entre abstraction et efficacité constitue un problème réel notamment lorsqu'un modèle à objets doit être mis en œuvre sur un système d'exécution réparti. C'est le cas du modèle Guide, et nous verrons dans le chapitre suivant les solutions proposées pour réduire le coût d'exécution des applications.

Dans les modèles de programmation par objets, les concepts de type (entité qui définit la représentation d'une structure de données) et de module (entité qui spécifie les procédures applicables à la structure de données) sont unifiés en un seul et même concept : la classe. Une **classe** spécifie un modèle d'objets par la définition d'un modèle de données et de l'ensemble des procédures destinées au traitement de ces données. Elle

---

[1] Les modèles de programmation par objets ne sont cependant pas les premiers à intégrer le principe d'abstraction de données qui constitue la base des techniques de programmation par types abstraits [Liskov 74], [Bert 83], [Cardelli 85].

[2] Dans la terminologie des objets, 'liaison dynamique' est le terme utilisé pour désigner la recherche dynamique de code nécessaire à l'exécution des appels d'opération sur les objets.

fonctionne également comme un générateur d'objets. Ces derniers sont créés dynamiquement et parfois même détruits automatiquement par des mécanismes de ramasses-miettes quand ils deviennent inutiles.

L'unification des concepts de type et de module permet d'intégrer au modèle un mécanisme de construction de classes par enrichissement de classes existantes : l'**héritage**. Ce mécanisme constitue un aspect clé des techniques de programmation par objets. Il permet notamment d'exploiter le partage de comportement entre objets comme par exemple entre un objet document et un objet rapport, en définissant la classe du second comme une spécialisation de la classe du premier. Outre l'économie d'écriture qu'il permet de réaliser lors de la définition de la classe des objets rapports, l'héritage augmente aussi le degré de polymorphisme des variables de désignation d'objets : rapports et documents partageant un même comportement, une variable qui peut désigner un document peut également désigner un rapport.

Le degré de correction du code, obtenu dans les langages classiques par le typage des variables et un contrôle statique de types, peut être obtenu dans les langages de programmation par objets par la mise en œuvre d'un contrôle statique du domaine de définition des variables, défini sur la base des relations de partage de comportement entre objets. Ce **contrôle statique** permet d'allier sécurité et souplesse en permettant l'exploitation du polymorphisme inhérent à l'application du principe d'abstraction.

On peut remarquer que la mise en œuvre d'un contrôle statique conduit à la définition de constructeurs pour la réalisation des classes paramétrées. Un **constructeur** définit un modèle de classe paramétrée (*Table of [T,U]* par exemple) et permet la création de classes à partir de ce modèle par l'attribution d'une valeur effective à chaque paramètre de généricité (*Table of [String,Integer]* et *Table of [Real,Real]* par exemple). Nous verrons dans la section suivante que les possibilités des mécanismes de généricité fournis dans les modèles à objets contraints par un contrôle statique sont comparables à celle du mécanisme de généricité contrainte fourni dans le langage Ada.

La section suivante contient une présentation détaillée des concepts de base de la programmation par objets introduits ci-dessus : la notion de classe, le contrôle statique et les outils de construction de classes que sont l'héritage et les constructeurs.

2. LES CONCEPTS DE BASE DES MODELES À OBJETS

Dans les modèles de programmation par objets, une application consiste en un ensemble d'objets qui interagissent par appel de leurs procédures d'accès. Programmer une application consiste à décrire l'ensemble des objets qui la constituent par l'intermédiaire de leurs classes.

L'objet de la présente section est de fixer les concepts qui constituent la base de ces modèles et d'introduire le vocabulaire généralement employé pour les décrire. Ces concepts sont au nombre de quatre :

- la classe, modèle et créateur d'objets,
- le contrôle statique, qui fixe les règles de programmation,
- l'héritage, qui permet une programmation incrémentale,
- les constructeurs, qui permettent la description des classes paramétrées sous une forme générique.

Il est important de noter que certains modèles ne donnent pas lieu à un contrôle statique. L'exemple le plus connu de ce type de modèles est celui mis en œuvre par le langage et l'interpréteur Smalltalk [Goldberg 83], [Goldberg 85]. En réalité, seuls les concepts de classe et d'héritage sont les pierres de base de la programmation par objets. Je m'intéresse cependant dans cette thèse aux environnement de développement dédiés à la production de logiciel plus qu'au prototypage. C'est la raison pour laquelle je présente dans cette partie introductive le principe du contrôle statique et le concept de constructeur qui en est directement issu. Ce contrôle fournit en effet une aide importante à la production de logiciels corrects. On le retrouve dans la majorité des langages de programmation par objets développés à l'heure actuelle[1] et l'apparition de versions typées de Smalltalk 80 illustrent également son intérêt [Johnson 86].

Les exemples qui servent de support aux sections suivantes sont donnés dans un langage simple dont la syntaxe reprend les formes syntaxiques les plus couramment employées dans les langages de programmation par objets. Les instructions d'appel d'une opération sur un objet respectent notamment le format suivant, avec un point comme séparateur :

```
<objet_cible> <séparateur> <nom_opération> <liste_paramètres>
```

[1] C++ [Stroustrup 86], Eiffel [Meyer 88], Trellis-Owl [O'Brien 85a], O2 [Lécluse 88], etc.

Exemple

```
aStack.push (x) ;
aDocument.edit ;
```

2.1 Les classes

Les classes sont les entités modèles et créatrices d'objets. La définition d'une classe est constituée de la description des éléments qui constituent un objet : données et procédures. L'ensemble des données est appelé **état** : ce terme fait référence à l'évolution de la valeur de l'objet au cours de la vie. Les procédures sont souvent désignées par les termes de **méthode** ou **opération**.

Une classe permet de créer dynamiquement des objets conformes au modèle qu'elle spécifie. Un objet créé à partir d'une classe donnée est appelé **exemplaire** de cette classe ; on dit aussi que cette dernière est la classe de l'objet.

L'exemple ci-dessous décrit une réalisation possible des objets piles d'entiers.

```
class IntStack is
   (définition de l'état d'un exemplaire de la classe)
   contents  : Array [10] of Integer ;
   top       : Integer = 0 ;
   (définition des procédures d'accès)
   (définies sur les exemplaires de la classe)
   procédure isEmpty : Boolean is
     begin
       return (top = 0)
     end ;
   procédure isFull : Boolean is
     begin
       return (top = 10)
     end ;
   procédure push  (newItem : Integer) is
     begin
       top := top + 1 ;
       contents[top] := newItem ;
     end ;
   procédure pop : Integer  is
     begin
       top := top - 1 ;
       return contents[top+1]
     end ;
end IntStack.
```

La classe est aussi créateur d'objets. A cet effet, elle est souvent définie comme un objet particulier possédant une procédure, généralement appelée **new** ou **create**, qui

permet justement la création d'un nouvel exemplaire. Dans certains modèles, la spécification des données et des procédures qui constituent un objet classe sont entièrement à la charge du programmeur. Dans d'autres, les objets classes sont des exemplaires d'une classe prédéfinie. Ces différentes solutions sont comparée dans le chapitre suivant.

2.2 Le contrôle statique

L'objectif du contrôle statique est de vérifier avant exécution qu'un objet est toujours utilisé conformément à la spécification de son comportement.

Le comportement d'un objet est défini par la spécification de l'ensemble des procédures définies sur lui, et qui constituent sa partie visible ou **interface**. Chaque procédure est spécifiée par sa **signature** qui définit son nom et la liste de ses paramètres, et par la description de son effet. En général, cette dernière est tout à fait informelle, et c'est la raison pour laquelle le contrôle statique est limité à une vérification syntaxique du respect de l'interface des objets, constituée de l'ensemble des signatures des procédures qu'ils possèdent. L'implication de cette limitation est discutée au chapitre suivant.

C'est donc la propriété de partage d'interface qui permet de définir formellement le contenu du domaine de définition d'une variable de façon à ce que le respect de ce domaine garantisse une utilisation correcte des objets qu'il contient. Le contrôle statique mis en œuvre doit garantir deux choses :

- le respect du mode d'accès aux objets conformément à leur interface par le contrôle de la validité des instructions d'appel de procédures sur les objets,
- le respect du domaine de définition des variables par le contrôle de la validité des instructions d'affectation.

Conditions de conformité

Le terme **conformité** est celui généralement utilisé pour qualifier la règle qui permet de spécifier les domaines de définition des variables : c'est la raison pour laquelle il est introduit ici avec cette signification.

Les conditions de conformité énoncées ci-dessous permettent de définir statiquement les domaines de définition des variables de façon à garantir que si ces domaines sont respectés, l'accès aux objets qu'ils contiennent est correct.

Conditions de conformité

Une interface I' est conforme à une interface I si et seulement si

• pour chaque procédure op définie dans I il existe une procédure de même nom dans I',

• chaque procédure op de I a le même nombre d'arguments et de résultats que son homologue dans I',

• l'interface de chaque argument formel dans la signature de la procédure op de I est conforme à l'interface de l'argument formel correspondant dans la signature de la procédure op de I',

• l'interface de chaque résultat formel dans la signature de la procédure op de I' est conforme à l'interface du résultat formel correspondant dans la signature de la procédure op de I.

Toute interface I est conforme à elle-même.

Le domaine de définition d'une variable *v* définie par une interface *I* est l'ensemble des objets qui possèdent une interface conforme à *I*. Le respect de ce domaine est garanti par la vérification statique de la validité de toute affectation d'une nouvelle valeur à *v* :

• Soit l'instruction d'affectation <*v* := *e*> dans laquelle *e* désigne une expression. Cette instruction n'est acceptée par le compilateur que si l'expression *e* désigne un objet appartenant à *D*, c'est à dire dont l'interface est conforme à *I*.

Lors de l'appel d'une procédure *op* sur un objet désigné par une variable *v*, le respect de l'interface de l'objet est garanti par la vérification statique de l'existence dans son interface d'une procédure pouvant satisfaire la requête. Deux choses sont à vérifier :

• Il existe dans l'interface de *v* une procédure *op* dont le nombre de paramètres formels est égal au nombre de paramètres effectifs donnés dans l'instruction d'appel.

• Les paramètres effectifs donnés dans l'instruction d'appel sont tels que le domaine des arguments et le domaine des résultats de la procédure soient respectés.

Le passage de paramètres lors de l'appel d'une procédure est en effet réalisé par un certain nombre d'affectations : affectation de la valeur de chaque argument à la variable qui le désigne dans le code de l'opération, et affectation de la valeur de chaque résultat à la variable qui le désigne dans l'instruction d'appel. Le deuxième point cité dans le paragraphe précédent correspond donc à un double contrôle d'affectation.

La forme récursive des conditions de conformité peut être explicitée par une illustration imagée basée sur le concept d'entonnoir. Cette schématisation, donnée ci-dessous, est tout à fait intuitive et n'a d'autre objectif que de permettre à chacun d'associer

à la notion de conformité une image qui me paraît en faciliter la compréhension et la maîtrise.

Présentation intuitive des conditions de conformité

La présentation qui suit est basée sur l'analogie entre l'exécution d'une séquence d'instructions et un flot de données circulant dans une canalisation constituée d'un ensemble de boîtes mises bout à bout, chaque boîte schématisant l'exécution d'une opération sur un objet (cf Figure.1).

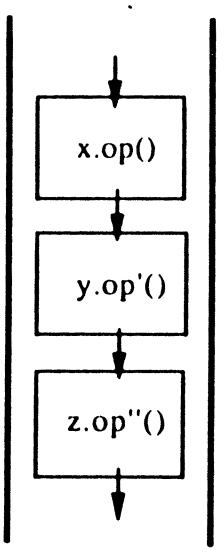


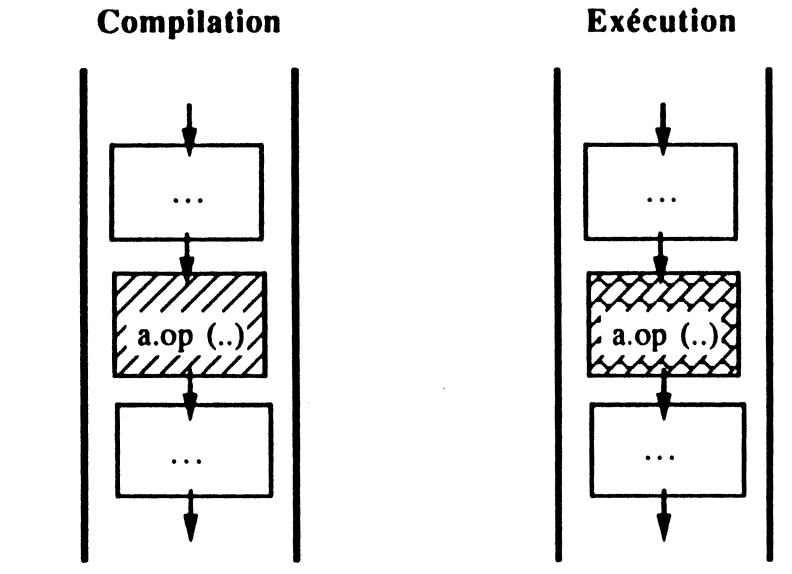
Figure 1. Analogie entre un programme et une canalisation

Soit *I* l'instruction *a.op (...)* dans laquelle *a* est une variable qui désigne un objet.

Soit *A* l'interface qui spécifie le domaine de définition de *a*.

Le contrôle de conformité effectué sur *I* à la compilation est fait sur la base de la définition de *op* telle qu'elle est donnée dans *A* ; le mécanisme de liaison dynamique mis en œuvre à l'exécution de *I*, provoque l'appel de la méthode *op* définie sur l'objet désigné par *a* à ce moment précis. Cet objet n'a pas forcement la même interface que *a* mais une interface *A'* qui lui est conforme, les conditions de conformité garantissant justement que le contrôle effectué statiquement reste valide pour toute opération *op* choisie lors de la liaison dynamique.

En termes de boîtes, on peut représenter par les deux schémas suivants l'interprétation statique de *I* par le compilateur et l'interprétation dynamique de *I* par le système d'exécution :

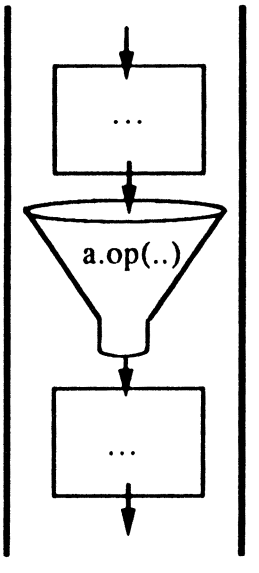


Interprétation statique de I par le compilateur

Interprétation dynamique de I par le système d'exécution

Dans le schéma de gauche, *op* est représentée conformément à sa définition dans l'interface *A*, et dans le schéma de droite, conformément à sa définition dans l'interface *A'*.

Le contrôle de conformité assure la validité du choix de **op** lors de l'exécution de *I*. Cela signifie entre autre qu'il n'y a aucune fuite (perte d'informations) dans la canalisation qui représente *I*, lorque le flot de données qui correspond à une exécution y circule. Cette 'garantie anti-fuite' peut être symbolisée par un **entonnoir** :



Cette nouvelle forme symbolise le fait que l'opération *op doit être capable de traiter le flot de données entrant tel qu'il a été filtré par le compilateur*, et qu'elle *doit produire un flot de données sortant conforme à celui escompté par le compilateur.*

La forme évasée du haut de l'entonnoir explicite le point 3 de la règle de conformité : tout argument *arg*, autorisé par le compilateur sur la base de la définition de *op* dans *A*, doit être accepté par l'opération *op* définie dans *A'* ; sa valeur doit donc pouvoir être affectée à l'argument *arg'* correspondant. L'instruction d'affectation <*arg'* := *arg*> n'est valide que si l'interface de *arg* est conforme à celle de *arg'*, et c'est bien ce que stipule le point 3 de la règle.

La forme rétrécie du bas de l'entonnoir explicite le point 4 de la règle de conformité : tout résultat *res'*, produit par l'opération *op* de *A'* doit pouvoir être interprété comme un résultat attendu par le compilateur sur la base de la définition de *op* dans *A* ; sa valeur doit donc pouvoir être affectée au résultat *res* de *op* dans *A* auquel il correspond. L'instruction d'affectation <*res* := *res'*> n'est valide que si l'interface de *res'* est conforme à celui de *res*, et c'est bien ce que stipule le point 4 de la règle.

Notion de vue

La mise en œuvre d'un contrôle statique permet de définir la notion de vue. Les vues sont couramment utilisées dans le domaine des bases de données : elles consistent à ne rendre visible à un utilisateur que l'ensemble des caractéristiques de la base qui lui sont utiles et constituent ainsi une certaine protection contre des manipulations qu'il n'est pas censé faire.

Définir une vue sur un objet consiste à fournir à certains utilisateurs une interface réduite pour cet objet. Une vue constitue ainsi un moyen de protection statique contre une utilisation non prévue de l'objet.

2.3 La règle d'héritage

L'héritage constitue un des aspects essentiels des méthodes de programmation par objets. Il permet la conception incrémentale de logiciel en autorisant la construction de classes par raffinements successifs. Il consiste en la prise en compte du partage de comportement entre objets lors de la définition de leurs classes.

Considérons par exemple les objets rapports et les objets documents. Tout rapport **est un**[1] document. Rapports et documents ne partagent donc pas seulement un même comportement (toute procédure applicable à un objet document l'est aussi à un objet rapport) mais également un ensemble de propriétés physiques. L'héritage permet l'exploitation de cette relation de partage dans la description de la classe des objets rapports que l'on peut considérer comme une spécialisation de la classe des objets documents.

Une classe *C'* construite par héritage à partir d'une classe *C* hérite pour ses exemplaires de tous les éléments, modèle d'état et procédures, définis dans la classe *C*. Il suffit au programmeur de spécifier dans *C'*, les variables d'état et les procédures d'accès qui sont propres à ses exemplaires. Il a en outre la possibilité de redéfinir un élément hérité, notamment le code d'une procédure : on parle alors de **surcharge**.

Héritage et surcharge

Dans les modèles contraints par un contrôle statique, les possibilités de surcharge sont restreintes de façon à ce que la validité du contrôle soit toujours garantie. Le partage de comportement exprimé par une relation d'héritage implique en effet un partage d'interface et contraint en conséquence les classes construites par héritage a définir pour leurs exemplaires une interface conforme à celle des exemplaires de chacune de leurs super-classes. Les implications du contrôle statique sur l'héritage sont reprises en détail au chapitre III.

Une classe construite par héritage est appelée **sous-classe** de la classe dont elle hérite, qui elle est appelée sa **super-classe**. La relation d'héritage est transitive. On désigne par le terme d'**ancêtre** d'une classe *C* toutes les classes à partir desquelles *C* a été construite. Reciproquement, les classes **descendantes** de *C* sont toutes celles qui sont construite à un niveau quelconque à partir de *C*.

Exemple

```
class Document is
   (définition de l'état d'un exemplaire de la classe)
   titre  : String ;
   auteur : String ;
   date   : Integer ;
   ...
   (définition des procédures d'accès)
   (définies sur les exemplaires de la classe)
```

[1] Je fais référence ici à la relation 'is-a' telle qu'elle est définie dans le domaine des bases de données.

```
    procédure éditer is ... end ;
    procédure imprimer is ... end ;
    ...
end Document.

class Rapport inherits Document is
    (complément de la définition de l'état)
    (d'un exemplaire de la classe)
    organisme   : String ;
    numéro      : String ;
    ...
    (définition des procédures d'accès)
    (définies sur les exemplaires de la classe)
    procédure éditer is
        ...
        end ;
    procédure imprimer is
        ...
        end ;
    ...
end Rapport.
```

Dans l'exemple ci-dessus, les procédures *éditer* et *imprimer* sont redéfinies car elles prennent en compte les variables d'état propres aux objets rapports : le nom de l'organisme qui publie le rapport, le numéro d'identification de ce dernier, etc.

Même si une procédure héritée est surchargée, il est souvent possible de faire référence à sa forme héritée dans le corps de la classe qui la surcharge. La procédure imprimer de la classe *Rapport* peut ainsi être définie de la façon suivante :

```
procédure imprimer is
  begin
    <impression de la valeur des variables d'état rajoutées>
    Document'imprimer ;
      (exécution de la procédure imprimer)
      (telle qu'elle est définie dans la super-classe)
  end ;
```

Classes différées

Certains modèles fournissent le concept de **classe différée** qui permet une meilleure exploitation de la factorisation de code possible avec l'héritage. Une classe différée est une classe dans laquelle sont définies des procédures dont la réalisation n'est pas précisée. L'intérêt de la définition d'une telle classe réside dans l'économie conceptuelle qu'elle permet d'obtenir en regroupant la description des propriétés

communes à un ensemble d'objets possédant chacun des caractéristiques propres, et dont les classes respectives peuvent être construites par héritage de la classe différée.

Les objets piles d'entiers peuvent par exemple être définis par l'intermédiaire d'une classe différée dans laquelle sont définis l'ensemble des éléments qui ne dépendent pas de leur organisation physique :

```
deferred class IntStack is
   (la définition de l'état des exemplaires de la classe)
   (est laissée à la charge des classes descendantes)

   (définition des procédures d'accès)
   (définies sur les exemplaires de la classe)

   procédure nbItems : Integer is
     deferred ;

   procédure isEmpty : Boolean is
     begin
       return (nbItems = 0)
     end ;

   procédure isFull : Boolean is
     deferred ;

   procédure push (newItem : Integer) is
     deferred ;

   procédure pop : Integer  is
     deferred ;

   procédure changeTop (newTop : Integer) is
     oldTop : Integer ;
     begin
       oldTop := pop ;
       push (newTop) ;
     end ;

end IntStack.
```

La classe des piles d'entiers réalisés par un tableau de 10 éléments est décrite ci-dessous comme une sous-classe de la classe *IntStack*.

```
class TableIntStack inherit IntStack is
   (définition de l'état des exemplaires de la classe)

   contents  : Array [10] of Integer ;
   top       : Integer = 0 ;

   (définition des procédures d'accès)
   (définies sur les exemplaires de la classe)
   (isEmpty et changeTop sont héritées sans surcharge)
   (la réalisation des autres procédures est spécifiée)

   procédure nbItems : Integer is
     begin
       return top
     end ;
```

```
    procédure isFull : Boolean is
      begin
        return (top = 10)
      end ;

    procédure push (newItem : Integer) is
      begin
        top := top + 1 ;
        contents[top] := newItem ;
      end ;

    procédure pop : Integer  is
      begin
        top := top - 1 ;
        return contents[top+1]
      end ;

  end TableIntStack.
```

On peut remarquer que seules les procédures dont la spécification était différée ainsi que le modèle de l'état des objets de classe *TableIntStack* sont définis dans cette dernière.

Déclaration par association

Une autre possibilité intéressante dans le contexte de l'héritage est la **déclaration par association**. Elle consiste à déclarer une variable par l'intermédiaire d'une autre de la façon suivante :

```
v' : like v ;
```

Ce type de déclaration est particulièrement intéressant lorsque l'association porte sur la variable qui désigne, dans le corps d'une procédure, l'objet sur lequel elle s'exécute : cet objet est généralement désigné par les mots clé **self** ou **current**. Dans une classe *C*, la déclaration <*v* : *like self*> est égale à la déclaration <*v* : *C*>, alors que dans une sous-classe *C'* de *C*, elle correspond à la déclaration <*v* : *C'*>. Considérons par exemple la classe *Number* décrite ci-dessous. Le mécanisme de déclaration par association permet de spécifier de façon claire l'ensemble des propriétés communes aux objets nombres.

```
deferred class Number is

  {la définition de l'état des exemplaires de la classe}
  {est laissée à la charge des classes descendantes}

  {définition des procédures d'accès}
  {définies sur les exemplaires de la classe}

  procédure add (Number) : like self is
    deferred ;

  procédure minus (Number) : like self is
    deferred ;

  procédure mult (Number) : like self is
    deferred ;
```

```
    procédure div (Number) : like self is
      deferred ;
end Number.
```

Les déclarations par association sont interprétées par le compilateur relativement à la classe dans laquelle elles apparaissent. La classe des nombres entiers étant par exemple construite par héritage de la classe *Number*, la signature de la procédure *add* définie dans cette classe spécifie que l'addition d'un nombre à un entier produit un entier.

2.4 Les constructeurs

Un constructeur est un modèle de classes paramétrées. La possibilité de paramétrer une classe par une ou plusieurs autres est importante car elle évite des définitions redondantes dans les modèles qui mettent en œuvre un contrôle statique. Les objets piles peuvent par exemple être définis par l'intermédiaire du constructeur *Stack of [T]* décrit ci-dessous :

```
constructor Stack of [T] is
  contents  : Array [10] of T ;
  top       : Integer = 0 ;
  procédure isEmpty : Boolean is
    begin
      return (top = 0)
    end ;
  procédure isFull : Boolean is
    begin
      return (top = 10)
    end ;
  procédure push  (newItem : T) is
    begin
      top := top + 1 ;
      contents[top] := newItem ;
    end ;
  procédure pop : T  is
    begin
      top := top - 1 ;
      return contents[top+1]
    end ;
end Stack.
```

Alors que dans le corps de la pile, aucune procédure n'est appelée sur les objets composants, il est souvent intéressant de pouvoir imposer une contrainte sur le paramètre de généricité. Dans l'exemple suivant, on spécifie par exemple que le paramètre de généricité est contraint par la classe *Number*. Cette contrainte permet d'utiliser les procédures définies dans la classe *Number* sur les variables définies par le paramètre

formel *Elt* : on peut supposer en effet que les objets désignés par ces variables ont tous un comportement conforme à celui des objets nombres.

```
constructor Matrix of [Elt : Number] is
   state : Array [10,10] of Elt ;
   procédure add (argMatrix : Matrix) : like self is
      i, j : Integer ;
      newMatrix : like current ;
      begin
         for i := 1 to 10 do
         for j := 1 to 10 do
(*)        newMatrix[i,j] := state[i,j].add(argMatrix[i,j]) ;
      end ;
         end ;
         return newMatrix ;
      end ;
   procédure minus (Matrix) : like self is
      ...
      end ;
   procédure mult (Matrix) : like self is
      ...
      end ;
   procédure div (Matrix) : like self is
      ...
      end ;
end Number.
```

Avec les paquetages génériques, le langage Ada fournit des possibilités tout à fait similaires à celles présentées ci-dessus. Une rapide comparaison peut cependant être intéressante. Considérons à cet effet la description du constructeur *Matrix* donnée ci-dessous dans ce langage :

```
generic
   type Number is limited private ;
   with function add (x,y : Number) return Number ;
   with function minus (x,y : Number) return Number ;
   with function mult (x,y : Number) return Number ;
   with function div (x,y : Number) return Number ;
package MatrixDef
   type Matrix is limited private ;
   function add (x,y : Matrix) return Matrix ;
   function minus (x,y : Matrix) return Matrix ;
   function mult (x,y : Matrix) return Matrix ;
   function div (x,y : Matrix) return Matrix ;
--
définition du corps du paquetage
--
   function add (x,y : Matrix) return Matrix is
      i, j : Integer ;
      newMatrix : Matrix ;
      begin
         for i := 1 to 10 do
         for j := 1 to 10 do
```

```
(*)       newMatrix[i,j] := add (x[i,j], y[i,j]) ;
        end ;
        end ;
        return newMatrix ;
      end ;
    function minus (x,y : Matrix) return Matrix is
      ...
    function mult (x,y : Matrix) return Matrix is
      ...
    function div (x,y : Matrix) return Matrix is
      ...
end MatrixDef
```

La contrainte de généricité est spécifiée par une liste de conditions alors que l'application du principe d'abstraction dans les langages de programmation par objets permet de la spécifier simplement par la mention d'une classe.

D'autre part, la surcharge de nom qui consiste à associer un même identificateur à des procédures différentes est résolue statiquement. Soit le paquetage *MatrixDef_T* construit par l'intermédiaire du constructeur *MatrixDef* :

```
package MatrixDef_T is
new MatrixDef (T, add_T, minus_T, mult_T, div_T)
```

MatrixDef_T est construit par duplication du code du paquetage générique *MatrixDef* dans lequel l'ensemble des éléments relatifs au paramètre formel *Number* sont remplacés par les paramètres effectifs *T*, *add_T*, *minus_T*, etc. Dans les modèles à objets par contre, la mise en œuvre d'un constructeur de classe peut être faite par partage du code du constructeur entre toutes les classes crées sur son modèle. L'instruction marquée (*) dans le corps du constructeur *Matrix* est donc bien résolue dynamiquement contrairement à l'instruction correspondante dans le corps du paquetage générique *MatrixDef*.

On peut remarquer qu'en fait la mise en œuvre du principe d'abstraction rend la surcharge de noms intrinsèque au modèle. C'est d'ailleurs la raison pour laquelle la notion de fonction générique telle qu'elle est introduite dans Ada n'existe pas dans les langages de programmation par objets : dans ces langages, tout identificateur de fonction est potentiellement surchargé.

3. CONCLUSION

Le nombre restreint de concepts : **objet, classe et héritage**, qui constituent la base des techniques de programmation par objets permet la définition de modèles minimaux et néanmoins puissants :

• Le niveau d'abstraction élevé mis en œuvre par le concept d'objet conduit à un niveau de décomposition fin et favorise une organisation du logiciel en composants réutilisables et extensibles.

• La définition de l'objet comme unité de structuration favorise un travail d'analyse et de conception basé sur l'organisation de l'information traitée par les programmes et non sur les traitements que ces derniers doivent mettre en œuvre. L'analyse descendante des applications qui consiste en une décomposition de la tâche à remplir en sous-tâches plus simples peut donc être remplacée par une analyse en va et vient basée sur les traitements et sur les données, et qui favorise le partage de composants logiciels entre différentes applications.

• La mise en œuvre d'un contrôle statique permet de fournir aux programmeurs une aide à la production de logiciels corrects tout en leur permettant de conserver la puissance d'expression induite par le niveau d'abstraction du modèle.

Le polymorphisme induit par le degré d'abstraction mis en œuvre pose cependant un réel problème d'efficacité pour la résolution duquel le support d'exécution choisi pour la réalisation du modèle est déterminant.

Les techniques de la programmation par objets sont utilisées aujourd'hui dans divers domaines d'applications tels que par exemple le génie logiciel, la gestion de bases de données et l'intelligence artificielle. Pour ma part, je m'intéresse dans cette thèse à l'application de ces techniques à la programmation d'applications générales. Je tente notamment de montrer dans la suite que des outils de programmation classiques tels que les exceptions ou le parallélisme peuvent être naturellement intégrés dans un modèle de programmation par objets. D'autre part, j'insiste sur le fait que la nature du support d'exécution choisi pour la mise en œuvre de ce type de modèle est importante au regard des problèmes d'efficacité cités précédemment, mais aussi parce qu'un modèle réalisé sur un support d'exécution classique ne peut offrir les mêmes possibilités qu'un modèle réalisé sur un système à objets. Le second permet en effet d'intégrer des outils tels que la gestion d'objets permanents ou la gestion de la répartition : ce sont notamment ces deux aspects qui font l'originalité du modèle Guide.

CHAPITRE III

LE MODELE D'OBJETS DE GUIDE
Présentation Comparative

1. INTRODUCTION

Les objectifs de ce chapitre sont la présentation détaillée et l'évaluation du modèle d'objets de Guide. Il explicite et justifie les choix faits, sur la base d'une étude comparative de Guide et de trois autres modèles d'objets. Les modèles retenus comme supports de comparaison sont Trellis/Owl, Emerald et Eiffel.

Ils ont été choisis car ils représentent la tendance qui après l'introduction des concepts de la programmation par objets dans Simula [Dahl 70], leur intégration dans Smalltalk [Goldberg 85] et la consécration qui a suivit, vise à en tirer de véritables outils de développement de logiciel notamment par l'introduction d'un mécanisme de contrôle statique de types.

Trellis/Owl, Emerald et Eiffel sont également des modèles qui comme Guide sont destinés à la programmation d'applications générales. Cette similitude d'objectifs à naturellement conduit à la définition, dans chacun d'eux, d'éléments représentatifs des mêmes concepts. Ils sont de plus tous les trois mis en œuvre par des langages de programmation propres et non comme l'extension d'un langage existant. Ce choix d'implémentation favorise à mon sens la conception d'un modèle cohérent puisqu'elle n'est génée par aucune contrainte inhérente à une mise en œuvre sur un langage support de trop haut niveau. J'ai notamment préféré Eiffel à C++[Stroustrup 86] pour cette raison.

On peut remarquer aussi qu'Emerald et Trellis/Owl font partie des modèles dont l'étude a constituée la base du travail de conception du modèle d'objets de Guide. Argus [Liskov 85] et SR [Andrews 82] ont fait l'objet d'un travail d'analyse similaire mais ils définissent tous deux des modèles de programmation assez différents de celui de Guide :

c'est la raison pour laquelle j'ai choisi de ne pas les présenter dans l'étude comparative que je propose ici.

A la différence d'Emerald, Trellis/Owl et Eiffel sont mis en œuvre sur des supports mono-utilisateur et centralisés et sont, en ce sens, très différents de Guide. Cette différence, qui pourrait a priori rendre plus délicate une étude comparative, est au contraire une source d'informations car elle permet une meilleure justification de certains choix faits dans Guide. On peut remarquer d'ailleurs à ce sujet que les aspects liés à la répartition ont été pris en compte dès le début des travaux de conception des modèles Emerald et Guide. De ce fait, les fonctions qui en dépendent (localisation d'un objet, appel d'un objet distant) sont assurées par le système d'une façon invisible aux applications, ce qui a permis la définition des éléments fondamentaux des deux modèles indépendamment des contraintes liées à la répartition.

La présentation comparative qui constitue le corps de ce chapitre n'est pas faite selon le schéma séquentiel classique : présentation de chacun des modèles et synthèse. Les différents éléments (objets, classes, héritage, conformité, généricité, etc) qui constituent la base des principes de programmation qui nous intéressent ici, sont présentés l'un après l'autre. Chacun est introduit par une rapide définition générale ; sa mise en œuvre dans les quatre modèles étudiés est alors détaillée puis une évaluation des solutions présentées est donnée sous forme de synthèse (les présentations générales données en début de section s'appuient pour certaines sur les éléments introduits au chapitre II).

Une présentation parallèle m'a semblée plus intéressante qu'une présentation séquentielle car elle facilite la mise en évidence des aspects importants des modèles que je présente. Elle me permet également d'éviter la redondance de propos inhérente à une présentation séquentielle de modèles voisins alors qu'il s'agit justement de mettre en valeur leurs similitudes et leurs différences. Un tel schéma facilite aussi la comparaison des éléments présentés et s'insère ainsi naturellement dans cette thèse dont un des objectifs est l'évaluation du modèle Guide.

Il m'a semblé utile de définir clairement les mécanismes fournis dans chacun des quatre modèles pour la définition des objets et des classes. En conséquence la section 3 qui présentent ces mécanismes peut paraître longue et redondante : la condenser ne me paraîssait pourtant pas souhaitable puisqu'elle fixe les bases des sections suivantes et en permet la compréhension. L présent chapitre ne peut cependant pas être considéré comme un manuel pour l'utilisation des langages qu'il présente. Seuls les éléments nécessaires et importants pour la cohérence et la compréhension de l'ensemble sont présentés. Les

références des documents descriptifs des modèles Trellis/Owl, Emerald et Eiffel sont données dans les sections 1.2, 1.3 et 1.4[1]. Elles contiennent une présentation générale de ces modèles et définissent les contextes dans lesquels les langages qui les mettent en œuvre ont été conçus et développés. La présentation générale du modèle Guide a été faite au chapitre I.

Précédemment à la présentation comparative qui constitue les sections 3 à 11 du chapitre, la section 2 contient un tableau récapitulatif dont le but est de donner un aperçu général des mécanismes fournis dans chacun des modèles et d'introduire l'ensemble des points abordés dans l'étude comparative.

La section 12 constitue la conclusion du chapitre. Elle reprend sous une forme synthétique les points les plus importants que le travail de comparaison présenté dans les sections précédentes a permis de mettre en évidence.

2.1 TRELLIS/OWL

Trellis/Owl est un langage de programmation développé par Digital Equipment. Il est mis en œuvre dans l'environnement de programmation par objets Trellis. Cet environnement est constitué d'un compilateur et d'un ensemble d'outils d'aide au développement ; il a été développé sous VMS sur une station VAX II. Dans la suite, je désigne le modèle et le langage Trellis/Owl par le nom plus simple de Owl.

Le support d'exécution du langage Owl est mono-site et mono-utilisateur. Les objets sont passifs. Le modèle d'objets combine l'héritage multiple à un contrôle statique de types. L'héritage permet la construction de types par spécialisation ; il consiste en l'exploitation d'une relation 'est-un'.

Une proposition d'extension du langage Owl pour la mise en œuvre d'une base de données à objets est présentée dans [O'Brien 86]. Cette extension doit permettre le partage d'information entre applications. Elle est encore à l'état de projet [O'Brien 88] et aucun document de présentation détaillée n'est disponible.

Le langage Owl est présenté dans les documents [O'Brien 85a] et [O'Brien 85b] et l'environnement de développement Trellis dans [O'Brien 87].

---

[1] Pour plus de clarté, ces documents ne sont, en général, pas référencés dans le texte.

2.2 EMERALD

Emerald est un langage de programmation par objets destiné à la programmation d'applications réparties et défini dans le cadre d'un projet de recherche de l'Université de Washington. Il présente un intérêt particulier du fait qu'il est issu d'un projet de deuxième génération ; il a en effet été développé sur la base de l'expérience acquise dans le projet Eden, projet mené dans la même université, par la même équipe ([Almes 85], [Black 85]).

Le projet Eden a débuté en septembre 1980 et avait pour but l'expérimentation d'un modèle à objets pour la programmation distribuée ; la priorité portait à l'époque sur la définition d'un système. Pour éviter le travail considérable inhérent à la conception d'un nouveau langage, l'équipe s'était limitée à la modification d'un langage existant : Concurrent Euclid. Dans une telle approche, on est forcement limité par les fonctions du langage support et le langage résultant n'est pas toujours conforme aux concepts initiaux. L'objectif du projet Emerald est précisément de s'affranchir de toutes ces contraintes par la définition d'un nouveau langage. L'expérience acquise dans Eden a été prépondérante, mais les auteurs d'Emerald se sont également fortement inspirés des systèmes Argus [Liskov 85] et Smalltalk [Goldberg 85].

Les concepts de bas niveau tels que la multiplicité des sites, la localisation des objets et la gestion de la concurrence sont intégrés dans le langage. Les objets encapsulent les notions de données et de processus et constituent les unités de distribution. Le langage ne fournit pas de possibilité d'héritage mais son compilateur met en œuvre un contrôle statique de type basé sur le partage d'interface entre objets.

Le modèle et le langage Emerald sont présentés dans les documents [Black 86a] et [Black 86b].

2.3 EIFFEL

Eiffel est un langage de programmation par objets mis en œuvre dans un environnement de développement propre. Il est disponible sous la forme d'une version commercialisée supportée par Unix ; son compilateur utilise C comme langage intermédiaire.

La conception d'Eiffel a été menée avec le souci de fournir un outil de développement puissant pour la production de logiciels réutilisables et extensibles. Les principes de la programmation par objets ont été retenus dans cette optique.

Le support d'exécution du langage est mono-site et mono-utilisateur. Les objets sont passifs. Le langage offre l'héritage multiple comme outil de réutilisation de code. Le modèle d'objets est typé ; il permet notamment la définition d'assertions et fournit ainsi aux programmeurs un outil d'aide à la production de logiciels corrects.

La langage Eiffel est présenté dans [Meyer 88]. La présentation que j'en fais ici est également basée sur la documentation livrée avec la version 2.1 du compilateur et sur quelques expériences de programmation.

2. TABLEAU RÉCAPITULATIF

Le tableau présenté sur les pages qui suivent a pour objectif de donner une vue synthétique de l'ensemble des points abordés dans les sections 3 à 11 du chapitre.

Il décrit les principaux mécanismes fournis dans les modèles présentés et introduit les éléments de syntaxe les plus importants de chacun des langages correspondants.

	TRELLIS-OWL	EMERALD	EIFFEL	GUIDE
Objet état + procédures **actifs / passifs**	données + opérations publiques ou privées objets passifs	données + opérations objets actifs ou passifs	données + opérations exportées ou cachées objets passifs	données + opérations objets passifs

Classe
=
modèle
d'objets
qui définit
interface+realisation
	Type
	=
	interface
	(opérations publiques)
	et
	réalisation
	
	`type-module aType`
	`  component me.x : …`
	`  …`
	`  operation … is`
	`  public/private`
	`  begin … end ;`
	`  …`
	`end type-module ;`
		Type
		=
		interface
		(ou **type abstrait**)
		et
		réalisation
		(ou **type concret**)
		
		`aType`
		`export …`
		`  var x : …feature`
		`  …`
		`  operation …`
		`  … end ;`
		`  …`
		`end aType ;`
			Classe
			=
			interface
			(opérations exportées)
			et
			réalisation
			
			chaque classe
			définit un **Type** d'objets
			(interface + réalisation)
			
			`class aClass`
			`export …`
			`  class aClass`
			`  x : …`
			`  …`
			`  anOperation … is`
			`  do … end ;`
			`  …`
			`end aClass`
				Classe
				=
				interface + réalisation
				une classe réalise un type
				
				Type = interface
				
				`type aType is`
				`  method … ;`
				`  …`
				`end aType.`
				`---`
				
				`implements aType is`
				`  x : … ;`
				`  …`
				`  method … begin … end ;`
				`  …`
				`end aClass ;`

Classe = **créateur** **d'objets**	un Type est un objet pour lequel le programmeur doit spécifier explicitement données et opérations ---	la création d'objets se fait par l'intermédiaire de Constructeurs ---	une Classe possède une opération prédéfinie de nom **Create** qui permet	une Classe est un objet dont le type et la classe sont prédéfinis --- une des opérations définies

	une opération définie
	sur un objet type
	peut créer un exemplaire
	de ce type
	et dont l'évaluation
	provoque la création
	d'un objet du type
	
	`type-module …`
	`operation create (MyType,…`
	`returns (MyType)`
	`is allocate`
	`begin … end ;`
		un **constructeur**
		est une expression
		qui contient
		la déclaration d'un type
		
		`anNewObject <- object T`
		`<déclaration du type T>`
		`end T ;`
			la création
			de ses exemplaires
			--- 	
			une classe
			n'est pas
			un objet
			
			`anObject : aClass ;`
			`… …`
			`anObject.Create ;`
				sur les objets classes
				s'appelle **new**
				et
				permet la création
				d'exemplaires
				de la classe
				
				`anObject : ref aType ;`
				
				`anObject := aClass.new ;`

Héritage	**héritage multiple**	—	**héritage multiple**	**sous-typage et heritage simples**
Classes différées	un **Type abstrait** est un type dont les réalisations de toutes les opérations ne sont pas spécifiées	—	une **Classe différée** est une classe dont les réalisations de toutes les opérations ne sont pas spécifiées	—
Déclaration par association	—	—	une déclaration de variable peut être dépendante du contexte `v : like v' ;`	—
Surcharge	**Surcharge conforme** un type construit par héritage d'un ou plusieurs autres		**Surchage non conforme** possible (à cause de la possibilité de	**Surcharge conforme** un type construit par sous-typage d'un autre

	doit leur être conforme	—	déclaration de variables par association et de l'orthogonalité entre héritage et exportation)	doit lui être conforme --- une classe construite par héritage d'une autre doit réaliser un sous-type du type réalisé par sa super classe

Constructeurs	un **Constructeur** est un modèle de types paramétrés par un ou plusieurs types
	`type-module Set [Elt:Type]`
	…
	une contrainte peut être imposée sur un type paramètre (Elt conforme à aType)
	`type-module Table [Elt:subtype(aType)]`
	…
		les **types paramétrés** sont réalisés par le passage d'objets types en paramètres des opérations	un **Constructeur** est un modèle de classes paramétrées par des types
			`class Set [Elt]`
			…
			une contrainte peut être imposée sur un type paramètre (Elt conforme à aType)
			`class Table [Elt -> aType]`
			…
				un **Constructeur de types** est un modèle de types paramétrés
				`type constructor Table [Elt : aType] is`
				…
				un **Constructeur de classes** est un modèle de classes paramétrées
				`class constructor Table [Elt : aType] is`
				…
				une contrainte est imposée sur chaque paramètre

			Possibilité de mémorisation et de récupération explicites de l'état d'un objet	tout objet Guide est

Objets permanents	—	—	dans un fichier Unix
			
			`anObject.Store (aFile) ;`
			`…`
			`anObject.Retrieve (aFile)`
				potentiellement permanent
				
				il doit être explicitement détruit
				par une instruction
				
				`anObject.Destroy`
				
				ou par le ramasse-miettes

Exceptions	une **Exception** est un résultat particulier d'une opération
	
	`operation push (…)`
	`signals FullStack`
	`is …`
	
	une opération dont l'exécution se termine mal signale une exception à l'objet appelant
	
	`operation push (…) …`
	`is`
	`…`
	`signal FullStack ;`
	
	le bloc de traitement est une alternative à l'opération qui a échoué
		—	une **Exception** signale une condition anormale pendant l'exécution d'une opération
			
			(violation d'une assertion, appel d'une opération sur une variable non initialisée, erreur détectée par la machine ou le système)
			
			une exception qui survient pendant l'exécution d'une opération sur un objet peut être prise en compte par ce dernier
			
			`anOperation is`
			`…`
			`rescue`
			`<bloc de secours>`
				une **Exception** est un résultat particulier d'une opération
				
				`method push (…)`
				`signal FullStack`
				`begin …`
				
				une opération dont l'exécution se termine mal signale une exception à l'objet appelant
				
				`method push (…) …`
				`begin`
				`…`
				`raise FullStack ;`
				
				le bloc de traitement est une alternative à l'opération qui a échoué

	`push (aStack, …) ;`
	`except`
	`  on Fullstack do : …`
	
	les exceptions constituent un **outil de programmation**
			`end`
			
			le bloc de secours permet de remettre l'objet dans un état cohérent
			
			les exceptions constituent un **outil d'aide à la production de code correct**
				`push (aStack, …) ;`
				`except`
				`Fullstack : …`
				
				les exceptions constituent un **outil de programmation**

Parallélisme et Objets partagés	—	le parallélisme est implicite
		--- 	
		il est dû à l'existence simultanée de plusieurs objets actifs (objets qui possèdent un processus)
		
		le contrôle d'accès aux objets partagés est réalisé par encapsulation des données et opérations à protéger dans un **moniteur**
		
		`aType`
		`export …`
		`…`
		`monitor`
		`  …`
		`end monitor`
		`end aType`
			—	le parallélisme résulte de l'exécution d'un bloc parallèle
				
				`co-begin`
				`  anObject.anMethod (…) ;`
				`  …`
				`co-end` …
				
				le contrôle d'accès aux objets partagés est réalisé par l'intermédiaire de **conditions d'exécution** attachées à leurs opérations
				
				`class aClass … is`
				`  …`
				`  method op (…)`
				`  control <condition>`
				`  begin … end op ;`
				`  …`
				`end aClass.`

Répartition	—	la localisation des objets du système support est invisible aux programmeurs elle peut être prise en compte explicitement (localisation d'un objet, fixation, déplacement)	—	la localisation des objets du système support est invisible aux programmeurs elle peut être prise en compte explicitement (localisation d'un objet, fixation, déplacement)

3. DÉFINITION, CRÉATION ET INITIALISATION DES OBJETS

Un objet est constitué d'un ensemble de **données**, son **état**, associé à un ensemble de **procédures** ; ces dernières sont parfois appelées **méthodes** ou **opérations**. Il nous faut encore compléter la liste de ces constituants par un éventuel ensemble de **processus**, car bien que certains modèles dissocient complètement les composantes statique et dynamique d'une application, d'autres ont fait le choix d'intégrer ces deux composantes à la notion d'objet.

Une application conçue selon un modèle de programmation par objets est constituée d'un ensemble d'objets qui interagissent par le biais d'appels à leurs opérations. L'appel d'une opération sur un objet a en général la forme suivante :

```
<cible> <séparateur> <nom_opération> <liste_paramètres>
```

Les langages Emerald, Eiffel et Guide utilisent notamment ce format avec un point comme séparateur :

```
aStack.push (x)
aDocument.edit
```

La forme classique d'un appel de procédure est cependant parfois conservée ; le premier paramètre de l'appel désigne alors l'objet cible. C'est la forme utilisée dans le langage Owl :

```
push (aStack, x)
edit (aDocument)
```

Définition et création des objets

La plupart des modèles à objets font une distinction entre la description des objets et les objets eux-mêmes ; ils définissent une entité particulière, la **classe**, qui permet de définir un modèle et de créer à partir de ce modèle autant d'objets que nécessaire.

La définition d'une classe est constituée de la description des éléments qui constituent un objet : données, opérations, processus, etc. La classe est également constructeur d'objet et permet de créer des objets conformes au modèle qu'elle spécifie ; pour cela, le programmeur exécute une opération particulière appelée en général **new** ou **create**. Un objet créé à partir d'une classe donnée est appelé **exemplaire** de cette classe ; on dit aussi que cette dernière est la classe de l'objet.

Utilisation des objets

L'utilisation d'un objet n'est possible que par appel des opérations définies dans son **interface**. Cette dernière constitue ce qu'on appelle la partie visible ou publique de l'objet, par opposition à sa partie cachée ou privée. L'interface d'un objet est spécifiée lors de sa définition.

Initialisation des objets

Après l'allocation de la zone de mémoire nécessaire à un nouvel objet, il est important de l'initialiser. La création d'un objet et son initialisation correspondent à deux événements distincts mis en œuvre par deux opérations différentes, l'une qui alloue la zone mémoire nécessaire et intègre le nouvel objet à l'espace d'objets et l'autre qui effectue l'initialisation de l'état de l'objet. L'opération d'initialisation peut être considérée comme une opération de l'objet contrairement à l'opération de création. Cette dernière est souvent définie comme une opération propre à la classe de l'objet, notamment dans les modèles qui définissent les classes comme des objets particuliers.

Cette section présente les mécanismes définis dans les quatre modèles étudiés pour la définition des parties visible et cachée des objets, pour leur création et leur initialisation.

Le même exemple de programme sert de support aux sections qui suivent ; il présente la définition et la création d'un catalogue simple permettant d'associer un nom à un objet quelconque et de retrouver ce dernier à partir du nom donné. Le nom de la classe introduit dans l'exemple est *oneEntryDirectory*.

3.1 Owl

Owl est un modèle à objets passifs et ne définit aucune entité active correspondante à la notion de processus. Un objet est donc constitué d'un ensemble de données associé à un ensemble de procédures.

Owl n'utilise pas le terme de classe mais celui de **type** pour désigner l'entité modèle et créateur d'objets. L'**interface** d'un objet est constitué de l'ensemble des éléments déclarés publiques dans le corps de son type.

Les types de Owl

Une des particularités de Owl est la possiblilité offerte au programmeur de définir non seulement les opérations propres à ses objets mais également celles qui sont propres à

leurs types. Ces derniers sont en effet considérés comme des objets particuliers dont la définition est entièrement à la charge du programmeur.

Le type *oneEntryDirectory*correspondant à l'exemple introduit plus haut s'écrit de la façon suivante[1] :

```
type_module oneEntryDirectory

  component me.name : String
  component me.anObject : Anything

  operation Store (me, n : String, o : Anything)
    is
    begin
      me.name := n ;
      me.anObject := o ;
    end ;

  operation Lookup (me, n : String)
    returns (Anything)
    is
    begin
      if n = me.name
        then return me.anObject
        else return nil
      end if
  end ;

  operation create (Mytype)
    returns (Mytype)
    is allocate
    begin
      me.name := nil
      me.anObject := nil
    end ;

end type-module
```

La définition des propriétés d'un type et la définition des propriétés des exemplaires du type sont faites dans un même bloc comparable à celui donné ci-dessus. Dans ce bloc, les exemplaires du type et le type lui-même sont traités de manière uniforme ce qui implique la nécessité de spécifier si une propriété est destinée à l'un ou à l'autre. Le premier argument de toute opération, appelé argument de contrôle, donne cette information par le biais des mots clé *me* et *myType*. Les noms des composants sont paramétrés par les mêmes mots clé.

[1] Dans la version Owl de l'exemple (ainsi que dans celles données dans les trois autres langages) *Anything* désigne un type quelconque.

Les variables d'état d'un objet type ne sont pas visibles dans le corps des opérations définies sur ses exemplaires. Réciproquement, les variables d'état propres à un exemplaire du type ne sont pas visibles dans le corps des opérations définies sur l'objet type.

La création d'un objet de type *oneEntryDirectory*se fait par appel de l'opération create défini sur ce type :

```
aVariable := create (oneEntryDirectory) ;
```

L'exécution de l'instruction ci-dessus provoque la création d'un nouvel objet parce que le programmeur l'a explicitement spécifié en utilisant le mot clé *allocate* dans la déclaration de l'opération *create* ; l'utilisation de ce mot clé est restreinte aux opérations définies sur un type et qui produisent en résultat un objet de ce type (la clause *returns* doit être spécifiée et le type du résultat doit être le même que celui de l'argument de contrôle).

Si l'état initial de l'objet est dépendant du contexte d'exécution, l'opération de création peut être, par exemple, paramétrée de la façon suivante :

```
operation create (Mytype, n : String, o : Anything)
  returns (Mytype)
  is allocate
  begin
    me.name := n ;
    me.anObject := o ;
  end ;
```

Une opération qui provoque la création d'un objet est mise en œuvre par deux opérations distinctes, l'une définie sur l'objet type et l'autre définie sur l'objet exemplaire. On retrouve dans ce shéma la séparation naturelle entre la création de l'objet et son initialisation. La compilation de l'exemple ci-dessus provoquerait la génération automatique des deux opérations suivantes :

```
operation create (Mytype, n : String, o : Anything)
  returns (Mytype)
  is allocate
  begin
    var obj : Mytype := .. a new unintialized object ..
    create (obj, n, o) ;
    return obj ;
  end ;

operation create (me, n : String, o : Anything)
  is
  begin
    me.name := n ;
    me.anObject := o ;
  end ;
```

Création et initialisation d'un objet sont donc conceptuellement intégrées en une seule opération définie sur le type de l'objet. Cette intégration n'introduit aucune confusion dans le modèle puisque le type lui-même est défini comme un objet et visible comme tel.

Définition de l'interface d'un objet Owl

Les mots clé *public* et *private* permettent de distinguer dans la déclaration d'un type ce qui appartient à l'interface de l'objet de ce qui n'y appartient pas.

Dans la définition du type *oneEntryDirectory*donnée précédemment, la déclaration de chaque élément composant et opération doit donc être complétée par une indication spécifiant si cet élément appartient à l'interface de l'objet ou non :

```
type_module oneEntryDirectory

  component me.name : String
    is field private ;
  component me.anObject : Anything
    is field private ;

  operation Store (me, n : String, o : Anything)
    is public
    ...

  operation Lookup (me, n : String)
    returns (Anything)
    is public
    ...

end type-module
```

Une opération peut être déclarée publique ou privée ; dans le deuxième cas, elle n'appartient pas à l'interface de l'objet. Une opération privée ne peut être appelée que dans le code des opérations publiques et privées définies sur le même objet ; elle permet une factorisation de code entre ces opérations. Une telle opération est comparable à une procédure au sens classique du terme.

Un composant *c* peut être défini de différentes manières :

• Il peut correspondre à une variable d'état de l'objet (*is field*). S'il est de plus déclaré public, deux opérations, *get-c* et *put-c*, sont implicitement définies dans l'interface de l'objet pour permettre la consultation et la mise à jour de sa valeur. Je qualifierai dans la suite les composants déclarés *is field public* d'**attributs publics**.

• Il peut être défini indépendamment des variables d'état de l'objet mais correspondre conceptuellement à un composant public associé à une telle variable mais

pour lequel la définition des opérations *get* et *put* est à charge du programmeur. Je qualifierai dans la suite les composants de ce type de **composants calculés**.

La forme des appels aux fonctions *get* et *put* définies sur un composant public (calculé ou non) est similaire à celle utilisée dans les langages classiques pour la sélection des champs d'un n-uplet :

```
obj.<nom_composant> := ...   !i.e. put-nom_composant(obj, ...)
... := obj.<nom_composant>   !i.e. ... := get-nom_composant(obj)
```

Considérons par exemple l'extrait suivant d'une application de gestion de fenêtres :

```
type_module Window

component me.real_position : Point   ! position d'affichage
  is field private                   ! de la fenêtre
                                     ! sur l'écran

component me.position : Point
  get is (me.real_position)
  put is  ( delete (me) ;
            me.real_position := value ;
            display (me)) ;
...
end type_module ;
```

Le composant *position* ne correspond pas à une variable d'état d'un objet fenêtre, c'est un composant calculé. Le code des opérations *get* et *put* qui lui sont associées doit en conséquence être spécifié par le programmeur : l'opération *get* donne les coordonnées de la fenêtre sur l'écran et l'opération *put* mémorise une nouvelle position et réaffiche la fenêtre.

Un composant peut être déclaré sous une forme indexée ce qui permet au programmeur de déclarer des tableaux à une dimension :

```
component me.Table [3] : Integer ;
```

L'utilisation des composants pour la définition d'opérations telles que les fonctions sans argument et les opérations à un seul argument, simplifie la manipulation des objets. La possibilité de définir des composants calculés est intéressante car elle permet d'associer statiquement un certain nombre d'actions à la consultation ou à la mise à jour d'une variable d'état (dans l'exemple ci-dessus, la modification de la position d'une fenêtre provoque systématiquement son réaffichage).

3.2 EMERALD

Emerald est un modèle à objets actifs. Un objet peut contenir facultativement un processus : ce dernier, actif pendant toute la durée de vie de l'objet, est totalement

indépendant des exécutions des opérations de l'objet et effectue des tâches de fond. Un objet qui ne contient pas de processus a un rôle passif ; son réveil momentané est conditionné par l'arrivée de requêtes d'exécution de ses opérations.

La notion de classe n'existe pas dans Emerald ; elle est remplacée par celles de **type** et de **constructeur d'objet**. Un type définit un modèle d'objets ; les objets construits selon ce modèle sont dits de ce type. La fonction de création d'un nouvel objet est fournie par les constructeurs.

L'interface d'un objet est spécifiée par son **type abstrait**. Un type désigne en fait l'association entre un type abstrait et une réalisation possible de ce type abstrait, appelée aussi **type concret**.

Les types et les constructeurs d'Emerald

Un constructeur d'objets est une expression dont l'exécution a pour effet la création d'un seul objet ; la représentation, les opérations et le processus de l'objet sont définis dans le corps du constructeur qui contient de cette façon la description du type de l'objet. Lorsque plusieurs objets du même modèle sont nécessaires, le constructeur peut être encapsulé dans une instruction de boucle et plus généralement dans une opération définie sur un autre objet ; ce dernier remplace alors la 'classe créateur d'objets' telle qu'elle est définie dans les autres modèles.

Reprenons l'exemple du type *oneEntryDirectory*. Le constructeur d'un objet de ce type est décrit ci-dessous :

```
object  oneEntryDirectory
  var name : String ;
  var anObject : Anything ;

  operation Store [n : String, o : Anything]
    name <- n
    anObject <- o
  end Store

  function Lookup [n : String] -> [o : Anything]
    if n = name
      then o <- anObject
      else o <- nil
    end if
  end Lookup

  initially
    name <- nil
    anObject <- nil
  end initially
end oneEntryDirectory
```

Un constructeur est une expression particulière dont l'exécution produit un nouvel objet et doit donc être utilisé dans une affectation comme cela est illustré ci-dessous :

```
aVariable   <-   object   oneEntryDirectory
                 ...
                 end oneEntryDirectory
```

Dans le corps d'un constructeur, le programmeur peut décrire la séquence d'instructions qui correspond à l'initialisation de l'état de l'objet créé par le constructeur. Cette séquence introduite par le mot clé *initially* est exécutée automatiquement lors de la création de l'objet.

Les types abstraits d'Emerald

Dans la définition d'un type, le type abstrait réalisé est spécifié à l'aide de la clause export. Le constructeur décrit précédemment doit donc être complété par la clause suivante :

```
export Store, Lookup
```

Dans la déclaration d'un type, une opération qui n'apparaît pas dans la clause export est privée ; elle n'appartient donc pas à l'interface des objets du type.

Un type abstrait peut aussi être défini indépendamment des types qui le réalisent par le biais de la déclaration d'une constante de type *type*. Le type abstrait implémenté par le type concret *oneEntryDirectory*peut ainsi être déclaré de la façon suivante :

```
const OED = type OED
  operation Store [String, Anything]
  function Lookup [String] -> [Anything]
end OED
```

La raison d'être de la possibilité de définir un type abstrait indépendamment des types qui le réalisent est liée à la nature du contrôle statique d'accès aux objets mis en œuvre dans Emerald ; elle est explicitée en section 5.

3.3 EIFFEL

Eiffel est un modèle à objets passifs dans lequel les processus sont invisibles aux programmeurs. Les objets d'Eiffel sont définis par les **classes** qui constituent les entités modèles et créatrices d'objets de ce langage. L'**interface** d'un objet est définie explicitement dans le corps de sa classe par une liste d'exportation.

Les classes d'Eiffel

Une classe décrit la représentation de ses exemplaires ainsi que l'ensemble des méthodes qui leur sont applicables. Toute classe possède implicitement une opération appelée *create* qui permet la création de ses exemplaires.

La classe *oneEntryDirectory*s'écrit en langage Eiffel de la façon suivante :

```
class oneEntryDirectory
feature

   name : String ;
   anObject : Anything ;

   Store (n : String, o : Anything) is
   do
     name := n ;
     anObject := o ;
   end ;

   Lookup (n : String) : Anything is
   do
     if n = name
        then result := anObject
        else result := nil
     end
   end ;

end oneEntryDirectory
```

La création d'un objet se fait par l'appel de l'opération *create* implicitement définie sur sa classe :

```
anObject : oneEntryDirctory ;
...
anObject.create ;
```

Lors de la création d'un objet, les données composant son état sont systématiquement initialisées à des valeurs prédéfinies ; par exemple, '0' pour un entier et 'faux' pour un booléen. Le programmeur, qui souhaite qu'à leur création l'état des objets d'une classe donnée soit fonction du contexte ou constitué de valeurs différentes des valeurs prédéfinies, peut spécifier la séquence d'initialisation appropriée dans le corps de cette classe. Cette spécification se fait par le biais d'une définition explicite de la méthode *create* comme par exemple dans la classe *oneEntryDirectory*telle qu'elle est définie ci-dessous :

```
class oneEntryDirectory
...
   create (aName : String) is
     name := aName ;
   end ;
end oneEntryDirectory
```

La classe de l'objet créé par une instruction *<entité>.<create>* est toujours celle spécifiée dans la déclaration de *<entité>*. Si l'opération *create* est explicitement définie dans cette classe, l'instruction de création doit respecter sa spécification. Par exemple, la création d'un objet de la classe *oneEntryDirectory*décrite dans l'exemple précédent doit respecter le format suivant :

```
anObject.create ("une chaîne de caractères")
```

Dans le code d'une opération déclarée dans une classe *C*, le mot clé ***current*** permet de faire référence à l'objet courant, c'est-à-dire celui sur lequel s'exécute l'opération. Ce mot clé désigne une variable prédéfinie dont la classe est *C*. Il permet par exemple à l'objet courant de se transmettre lui même en paramètre d'un appel d'opération sur un autre objet :

```
anObject.op (..., current, ...) ;
```

Lors de l'appel d'une opération par l'objet courant sur lui-même, le mot clé *current* est implicite :

```
op (...) ;   -- instruction équivalente à <current.op (...)>
```

Déclaration par association

Déclarer une variable par association consiste à déclarer que son type est le même que celui d'une autre variable précédemment déclarée (cf chapitre II) :

```
r : oneEntryDirectory ;
...
v : like r ;
```

Lors de la définition d'une classe, une variable peut ainsi être déclarée du même type que l'objet courant : *<v : **like current** >*.

Les assertions

Dans Eiffel, la définition d'une classe peut être enrichie par un ensemble d'assertions : pré-conditions, post-conditions et invariants de classe. Une assertion est une expression booléenne dont la valeur permet de contrôler la validité des requêtes effectuées sur les exemplaires de la classe.

Exemple

```
class aClass ...
feature
  ...
  anOperation (...) is
    require
```

```
        <pré-conditions>
      do
        <code de l'opération>
      ensure
        <post-conditions>
      end ;
    ...
  invariant
    <invariants de classe>
  end ; -- aClass
```

Pré-conditions et post-conditions sont des assertions attachées à une opération. Une pré-condition spécifie les propriétés que doit satisfaire un exemplaire de la classe avant toute exécution sur lui de cette opération. Parallèlement, les propriétés que doit satisfaire un exemplaire de la classe après toute exécution d'une opération sont spécifiées par une post-condition. Un invariant est une assertion attachée à un objet. Elle spécifie une propriété que doit satisfaire l'objet après sa création et après toute exécution d'une opération appartenant à son interface.

Les assertions définissent un contrat qui lie une classe à ses clientes (on désigne sous le terme de cliente d'une classe, toutes les classes qui utilisent un ou plusieurs de ses exemplaires).

Définition de l'interface des objets Eiffel

L'interface d'un objet Eiffel doit être définie dans sa classe par une clause *export*. Cette clause donne la liste des noms de toutes les opérations publiques de l'objet.

La classe *oneEntryDirectory*décrite précédemment doit donc être complétée par la clause suivante :

```
export Store, Lookup
```

L'interface des objets de classe *oneEntryDirectory*est donc composée des opérations *Store* et *Lookup* dont les signatures sont spécifiées dans le corps de la classe.

Les variables *name* et *anObject* sont des variables d'état de l'objet ; dans Eiffel, elles sont aussi appelées **attributs**. Une fonction sans argument qui retourne la valeur d'un attribut peut être définie comme telle dans le corps d'une classe. Par exemple, on peut donner aux utilisateurs des objets de classe *oneEntryDirectory*l'accès à la valeur de l'attribut *name* de ces objets :

```
class oneEntryDirectory
export name, Store, Lookup
feature
  name : String ;
```

```
    anObject : Anything ;

    Store (n : String, o : Anything) is
      ...
    Lookup (n : String) : Anything is
      ...
end oneEntryDirectory
```

Les exemplaires de la classe *oneEntryDirectory*définie ci-dessus, possèdent donc deux attributs (*name* et *anObject*) et trois opérations (*name*, *Store* et *Lookup*).

3.4 GUIDE

Guide est un modèle à objets passifs, comparable en ce sens aux modèles Owl et Eiffel. Il a par contre été conçu pour être mis en œuvre sur un système multi-sites et multi-utilisateurs et offre des possibilités de gestion explicite du parallélisme. Les processus sont représentés dans le modèle par des objets particuliers appelés **activités**. Les objets activités sont les seuls objets actifs de Guide ; ces objets ne sont pas accessibles directement aux utilisateurs qui les manipulent implicitement par le biais d'instructions telles que celle qui permet de lancer une application (cf chapitre V) ou celle qui permet de décrire un bloc de traitement parallèle (cf section 11).

D'une manière similaire à Eiffel, Guide définit les classes comme les entités modèles et créatrices d'objets. Une **classe** est cependant considérée comme un objet particulier. L'interface d'un objet Guide est spécifiée par son **type** (ce terme est employé dans Guide au sens où celui de type abstrait est employé dans Emerald).

Les classes de Guide

Une classe définit un modèle d'objets. Chaque classe est considérée comme un objet particulier dont l'état contient par exemple le code des opérations définies sur ses exemplaires. Une des opérations, ou méthodes, définies sur les objets classes est l'opération prédéfinie *new* qui permet de créer de nouveaux objets. Aucune possibilité n'est offerte au programmeur pour l'ajout de nouvelles opérations sur les objets classes.

En langage Guide, le code de la classe *oneEntryDirectory*est le suivant :

```
class oneEntryDirectory is

  name : String ;
  anObject : Anything ;

  method Store (in n : String, o : Anything) ;
  begin
    name := n ;
    anObject := o ;
```

```
    end Store ;

    method Lookup (in n : String) : Anything ;
    begin
      if n = name
        then o := anObject
        else o := nil
      end
    end Lookup ;

end oneEntryDirectory.
```

La création d'un objet se fait par appel de la méthode *new* définie sur sa classe. Cette méthode est définie sous la forme d'une fonction dont la valeur désigne le nouvel objet :

```
anObject := oneEntryDirectory.new ;
```

L'initialisation des objets est laissée à la charge des programmeurs qui disposent à cet effet de deux possibilités. Si tous les exemplaires d'une même classe doivent avoir une même valeur initiale, ils peuvent spécifier cette valeur dans la déclaration des variables d'état :

```
name : String = "\0" ;
anObject : Anything = nil ;
```

L'initialisation des exemplaires d'une classe peut également être paramétrée mais devra alors être mise en œuvre par une méthode définie par le programmeur à cet effet :

```
class oneEntryDirectory is
  ...
  method Init (IN stg : String, obj : Anything) ;
  begin
    name := stg ;
    anObject := obj ;
  end Init ;
  ...
end oneEntryDirectory.
```

Une telle méthode n'a aucun statut particulier et son exécution sur un objet nouvellement créé est à charge des utilisateurs qui devront écrire la séquence suivante :

```
anObject := oneEntryDirectory.new ;
anObject.Init (...) ;
```

Dans le code d'une méthode, le mot clé *self* permet de désigner l'objet sur lequel elle s'exécute. Un objet peut donc appeler une de ses propres opérations, ou se transmettre lui-même en paramètre d'un appel d'opération à un autre objet :

```
self.op (...) ;
...
anObject.op (...,self,...) ;
```

La notion d'opération privée n'existe pas dans Guide. Toutes les opérations, ou méthodes, définies sur un objet sont publiques. Le programmeur a cependant la possibilité de définir des procédures :

```
class aClass
implements aType is

  procedure p (...) ;
  begin
    ...
  end p ;

  method op
  begin
    ...
    p (...) ;
    ...
  end op ;

end aClass.
```

A la différence d'une instruction d'appel de méthode sur un objet, une instruction d'appel de procédure est traduite statiquement en une instruction de branchement alors que l'appel d'une méthode sur un objet met en œuvre des mécanismes de liaison propres au système de gestion d'objets.

Les types de Guide

Un type spécifie l'interface de ses exemplaires. Les types sont déclarés indépendamment des classes qui les réalisent et la définition d'une classe doit mentionner le nom du type qu'elle réalise.

Le type *OED* des objets de la classe *oneEntryDirectory*décrite précédemment est le suivant :

```
type OED is
  method Store (in n : String, o : Anything) ;
  method Lookup (in n : String) : Anything ;
end OED.
```

Si l'initialisation des objets de classe *oneEntryDirectory* est réalisée par une méthode de nom *Init* par exemple, celle-ci doit être mentionnée dans le type *OED* car elle appartient à l'interface des objets de ce type.

Dans le corps d'une classe, le nom du type réalisé est spécifié à l'aide du mot clé *implements*. La classe *oneEntryDirectory*doit donc être complétée par la clause requise :

```
class oneEntryDirectory is
implements OED ;
```

```
    name : String ;
    anObject : Anything ;

    method Store (in n : String, o : Anything) ;
    ...

    method Lookup (in n : String) : Anything ;
    ...
end oneEntryDirectory.
```

Les variables *name* et *anObject* constituent l'état d'un objet de classe *oneEntryDirectory*. Il est possible d'associer à une variable d'état de l'objet une opération de lecture et une opération d'écriture permettant aux utilisateurs de l'objet la consultation et la mise à jour de sa valeur. Une telle variable est appelée **variable d'état visible** et doit être déclarée dans le type de l'objet :

```
type aType is
  ...
  v : Anything ;
  ...
end aType.
```

Un objet du type *aType* décrit ci-dessus, possède donc une variable d'état visible de nom *v*. La déclaration de cette variable correspond à la déclaration implicite des deux méthodes suivantes :

```
method read-v : Anything ;
method write-v (in newValue : Anything) ;
```

dans le type *aType*, et à la déclaration implicite d'une variable d'état de nom *v* et du code des méthodes *read-v* et *write-v* dans toute classe qui réalise le type *aType* :

```
type aClass
implements aType is
  ...
  v : Anything ;
  ...
  method read-v : Anything ;
  begin
    return v ;
  end read-v ;

  method write-v (in newValue : Anything) ;
  begin
    v := newValue ;
  end write-v ;

end aType.
```

Une variable d'état visible dans un objet Guide est comparable à un attribut public dans un objet Owl. Mises à jour et consultations se font notamment selon la même syntaxe :

```
obj.v := ...        // i.e. obj.write-v (...)
... := obj.v        // i.e. ... := obj.read-v
```

3.5 SYNTHESE

Certains points parmi ceux présentés dans la présente section méritent une attention particulière. J'ai retenu les cinq suivants :

• Quels sont les intérêts d'un modèle à objets passifs (Guide, Owl, Eiffel) par rapport à un modèle à objets actifs (Emerald) ?

• Quels sont les avantages que peut présenter la solution qui consiste à définir les classes comme des objets (Owl, Guide) ?

• Comment réaliser l'initialisation des objets d'une manière pratique pour le programmeur et cohérente avec le modèle ?

• La déclaration d'attributs publics dans un objet Owl ou de variables d'état visibles dans un objet Guide peut paraître contraire au respect du principe d'abstraction, comment est-elle justifiée ?

• Associer des pré-conditions et des post-conditions aux opérations peut-il être intéressant (Eiffel) ?

Objets et Processus

Le choix d'un modèle à objets passifs est essentiellement pragmatique. La séparation entre la description des objets et celle des processus permet de définir des structures légères et donc plus faciles à gérer. La taille moyenne des objets est en général peu importante et de ce fait, l'association systématique d'un ou plusieurs processus à un objet est relativement coûteuse.

Le modèle d'exécution d'Emerald, défini sur la base des expériences acquises dans le projet Eden [Black 85], illustre d'ailleurs cette analyse. Dans Eden, tout objet constitue une machine virtuelle multi-processus et contient ainsi virtuellement une partie importante du système d'exploitation ce qui alourdit considérablement la gestion des objets et complique les mécanismes de sauvegarde. La gestion des mécanismes d'exécution a été grandement simplifiée dans Emerald où un objet peut être passif ou actif et dans ce cas, ne contient plus qu'un unique processus.

Dans Guide, la solution de séparer la gestion des objets de celles des mécanismes d'exécution a été retenue pour les raisons pratiques énoncées plus haut.

Quel intérêt à définir les classes comme des objets ?

La solution qui consiste à considérer les classes comme des objets particuliers présente les avantages suivants :

• elle permet la définition d'un modèle uniforme et cohérent,

• elle permet de fournir aux programmeurs les moyens d'adapter les caractéristiques des classes à leurs propres besoins,

• elle permet d'introduire naturellement les classes dans l'environnement d'exécution ce qui facilite le développement d'outils qui nécessitent un accès aux informations qu'elles contiennent pendant l'exécution des applications (interpréteur et debogueur symbolique par exemple).

Une classe peut être considérée comme un objet au niveau conceptuel sans forcément être réalisée comme tel :

• Sur le plan conceptuel une telle solution me semble intéressante car elle permet la définition d'un modèle uniforme et cohérent. L'opération de création d'un nouvel exemplaire d'une classe peut ainsi naturellement être définie comme une opération sur l'objet classe, et appelée comme telle selon le schéma standard d'appel d'une opération sur un objet comme cela est fait dans Owl et Guide. Owl laisse de plus entièrement à la charge des programmeurs la spécification des opérations définies sur les objets classes. Cette solution peut paraître lourde car elle les oblige à prévoir systématiquement une opération de création. Elle a cependant l'avantage d'offrir des possibilités intéressantes : définition de plusieurs opérations de création sur le même objet classe, mémorisation dans un objet classe d'informations relatives à ses exemplaires comme leur nombre par exemple.

• Sur le plan de la mise en œuvre, un problème d'efficacité se pose lorsque les classes sont gérées de la même façon que les objets. C'est une solution qui convient bien au prototypage et à l'expérimentation rapide de programmes, comme cela est largement illustré par l'environnement de développement du langage Smalltalk [Goldberg 85]. Par contre, dès que plus d'efficacité est nécessaire, une solution différente notamment plus statique doit être envisagée.

Dans Owl par exemple, les classes ne sont pas gérés de la même façon que les objets à l'exécution. Dans Guide par contre, les classes sont des entités qui comme les objets sont chargées en mémoire d'exécution à la demande, et sont partagées entre différentes applications. Seule une solution de ce type était envisageable compte-tenu du fait que le système est réparti et multi-utilisateur. L'efficacité reste donc de fait un souci permanent des réalisateurs du système.

Dans Eiffel les classes ne sont pas considérées comme des objets mais définies comme des entités statiques par opposition aux entités dynamiques que sont les objets. Cette solution, compréhensible d'un point de vue du schéma d'exécution, me paraît regrettable sur le plan de l'uniformité et de la cohérence du modèle : l'opération de création

d'un nouvel exemplaire d'une classe n'a pas un statut clair et nécessite dans certains cas un traitement particulier, comme par exemple dans la définition de la règle d'héritage (cf 4.2). Elle introduit notamment une ambiguïté dans l'interprétation des instructions qui s'écrivent <*entité*>.<*op*> : si *op* est différent de *create*, il s'agit d'une instruction d'appel de l'opération *op* sur l'objet désigné par *entité* ; dans le cas contraire, il s'agit de la création d'un nouvel objet dont la classe est celle de entité. Dans le premier cas, la valeur de entité n'est jamais modifiée contrairement au second cas dans lequel entité désigne le nouvel objet après l'exécution de l'opération *create*.

Initialisation des objets

Distinguer l'initialisation de l'état des exemplaires d'une classe de l'opération qui permet leur création (create, new) me semble être la solution la plus cohérente d'autant plus qu'en pratique création et initialisation correspondent bien à deux événements distincts. En conséquence, la séquence *initially* d'Emerald me semble être une solution intéressante car elle met de plus en évidence le fait qu'une séquence d'initialisation n'a pas à être visible dans l'interface des objets.

Attributs publics et Variables d'état visibles

Les attributs publics de Owl et les variables d'état visibles de Guide constituent des outils intéressants notamment en ce qui concerne la définition de n-uplets. On peut remarquer cependant qu'ils constituent une entorse au principe d'abstraction en donnant dans l'interface d'un objet, des informations relatives à sa structure interne. Cette entorse est justifiée par un des objectifs communs aux concepteurs de Owl et de Guide, celui d'intégrer au modèle et au langage des fonctions issues des systèmes de gestion des bases de données telles la gestion de collection d'objets, la sélection d'objets par leur contenu, etc. Considérons par exemple la définition de type suivante :

```
type-module Person
  component me.real_name : String
    is field public ;
  component me.real_age : Integer
    is field public ;
end  type_module
```

Pour l'application aux bases de données, la redondance de définitions inhérente à une encapsulation stricte aurait été difficilement acceptable pour la définition des n-uplets, structure de données couramment utilisée dans ce domaine d'application. Le type décrit précédemment aurait ainsi dû être déclaré :

```
type-module Person
  component me.name : String
    get is (me.real_name) ;
```

```
    put is (me.real_name := value) ;
  component me.real_name : String
    is field ;
  component me.age : Integer
    get is (me.real_age) ;
    put is (me.real_age := value) ;
  component me.real_age : Integer
    is field ;
end  type_module
```

La possibilité de définir des attributs publics dans Owl ou des variables d'état visibles dans Guide doit en fait être considérée comme un outil d'expression dont la justification est d'ordre syntaxique.

Intérêt de la possibilité de définir des assertions

Du mécanisme d'assertions fourni dans le langage Eiffel, la séparation entre la description du code d'une opération et la spécification des conditions de validité des requêtes d'exécution de cette opération (pré-conditions) me paraît être particulièrement intéressante. Elle met en évidence ces conditions de validité et permet d'y rendre attentifs les programmeurs. De nombreuses erreurs de programmation proviennent en effet de l'oubli de la prise en compte de ce type de conditions et les mettre en valeur constitue à mon sens une aide précieuse à la programmation.

Post-conditions et invariants de classe ne présentent pas le même intérêt dans le contexte de la programmation d'applications générales qui nous intéresse ici. Ce type d'assertions est surtout intéressant en phase de mise au point de programmes puisqu'elles permettent au programmeur de tester la validité du code qu'il a écrit. Leur contrôle est d'ailleurs proposé en option dans Eiffel où il n'est préconisé qu'en phase de mise au point. On peut remarquer aussi que l'expression des post-conditions est du ressort de la spécification formelle d'opérations et nécessite en conséquence des moyens d'expression que des langages tels que Eiffel et Guide ne sont pas destinés à fournir.

4. CONSTRUCTION DE CLASSES PAR HÉRITAGE

La construction de classes par héritage est l'élément essentiel des méthodes de programmation par objets. L'héritage permet la conception incrémentale du logiciel en autorisant la construction de classes par raffinements successifs : il a été introduit et défini au Chapitre II.

Les sections qui suivent décrivent les mécanismes d'héritage fournis dans les modèles Owl, Eiffel et Guide (Emerald ne fournit pas de possibilité de construction de

classes par héritage). Nous verrons dans la section 5 que la mise en œuvre du contrôle statique conduit à imposer un certain nombre de restrictions quant aux possibilités de surcharge.

4.1 L'HÉRITAGE MULTIPLE DANS OWL

Owl est un modèle qui offre des possibilités d'héritage multiple. Un type Owl peut être construit par héritage à partir d'un ou de plusieurs autres types et sera alors déclaré de la façon suivante :

```
type_module aType
  subtype_of anOtherType
  subtype_of ...
  ...
  inherit anOperation from aSuperType, ...
  exclude anOperation, ...
  ...
end type_module ;
```

Les conflits de noms prévisibles en cas d'héritage multiple sont résolus par le programmeur à l'aide des clauses d'héritage sélectif *inherit ... from* et de ré-définition : exclusion de l'opération héritée par *exclude* et nouvelle définition dans le corps du sous-type. Les composants sont implicitement hérités mais peuvent être exclus ou redéfinis. Une opération privée n'est héritée que si elle est qualifiée par le mot clé *subtype-visible*.

Une opération héritée, mais redéfinie ou exclue, peut être utilisée sous sa forme héritée ; son nom doit alors être précédé par le nom du super-type dans lequel est définie la forme héritée à exécuter :

```
type_module aType
  subtype_of superType exclude op
  ...
  operation op (...)
    ! surcharge
  is
  begin
    ...
  end ;
  ...
  operation example (...)
  is
  begin
    superType'op (...) ;     ! appel de op
  end ;                      ! définie dans le type superType
  ...
end type_module ;
```

Si un nom d'opération apparaît ainsi qualifié, l'opération correspondante est recopiée et renommée dans le type courant où elle est définie comme une opération privée.

Types abstraits

Owl fournit sous la forme des types abstraits la notion de classe différée telle qu'elle a été définie au chapitre II. Un **type abstrait** est un type pour lequel la spécification du code de certaines opérations est laissée à la charge des descendants.

4.2 L'HÉRITAGE MULTIPLE DANS EIFFEL

Eiffel est un modèle qui comme Owl permet l'héritage multiple ; une classe peut être construite par héritage d'une ou plusieurs autres classes. La ou les super-classes d'une classe Eiffel sont définies dans le corps de la classe par une clause *inherit*. Tous les arguments et toutes les opérations définis dans une classe pour ses exemplaires sont hérités par ses sous-classes. Renommage et redéfinition permettent de pallier aux conflits provoqués par l'héritage multiple :

```
class aClass export
  ...
inherit
  aSuperClass
    rename anOperation as ..., ...
    redefine anOperation, ...
  aSuperClass
    rename anOperation as ..., ...
    redefine anOperation, ...
  ...
feature
  ...
end -- aClass
```

L'opération *create* définie implicitement ou explicitement dans une classe n'est pas une opération qui s'applique à ses exemplaires mais à la classe elle-même (cf 3.3). Lors de la construction d'une classe *B* par héritage d'une classe *A*, l'opération *create* définie dans *A* n'est donc jamais héritée par *B* ; elle peut cependant être accessible dans *B* si elle est renommée (voir la classe *Array* décrite en Annexe III).

Le renommage d'une opération redéfinie permet l'accès à sa forme héritée ce qui évite l'introduction d'un outil supplémentaire tel que la qualification des noms dans Owl :

```
class C export
  ...
inherit
  SC
    redefine op ;
    rename op as op' ;
  ...
feature
  ...
  op (...) is   -- surcharge
  do ... end
  ...
```

```
    example (...) is
    do
      op' (...) ;    -- appel de op définie dans SC
    end
    ...
end -- C
```

Classes différées

Eiffel fournit la notion de classe différée telle qu'elle a été définie au chapitre II. Une **classe différée** d'Eiffel est comparable à un type abstrait de Owl.

Déclaration par association

Le mécanisme de déclaration par association prend toute sa puissance lors d'un héritage où il provoque des redéfinitions implicites. Une instruction de déclaration par association est en effet toujours évaluée relativement à la classe dans laquelle elle apparaît.

Ainsi dans le corps d'une classe C, la déclaration <*v : like current*> est équivalente à la déclaration <*v : C*>, et dans le corps d'une classe C' construite par héritage de C à la déclaration <*v : C'*>. De même, une variable déclarée <*v : like w*> dans le corps d'une classe C est implicitement redéfinie dans les sous-classes de C si la variable w est redéfinie.

Héritage et assertions

Les possibilités de redéfinition des assertions sont contraintes par une règle précise qui porte d'une part sur la redéfinition des assertions des opérations héritées, et d'autre part sur l'héritage des invariants de classes. Soit A une classe, et B une sous-classe de A :

- dans B , les pré-conditions des opérations héritées de A peuvent être moins fortes que celles définies dans A ,
- dans B , les post-conditions des opérations héritées de A peuvent être plus fortes que celles définies dans A ,
- tous les invariants de classe définis dans A s'appliquent à B .

Les contraintes imposées par cette règle sont justifiées par le fait que l'héritage consiste en l'exploitation d'une relation de partage de comportement entre les exemplaires d'une classe et ceux de toutes ces sous-classes. Leur respect garantit que le contrat qui lie une classe à ses clientes est rempli par toutes les sous-classes de cette dernière. Eiffel n'est cependant pas un langage formel et son environnement ne fournit aucun mécanisme de contrôle de la validité des assertions dans les classes construites par héritage.

Héritage et exportation

L'héritage ne porte pas sur la partie visible des objets : lors de la construction d'une classe par héritage, la clause d'exportation de la nouvelle classe doit donc être totalement spécifiée.

Soit une classe *C'* construite par héritage d'une classe *C*. Toute propriété (opération, fonction, attribut) visible dans *C* peut être cachée dans *C'* et réciproquement.

4.3 L'HÉRITAGE SIMPLE DANS GUIDE

Guide fournit dans sa version actuelle un mécanisme d'héritage simple : une classe peut avoir au plus une super-classe. Ce mécanisme doit être étendu pour permettre l'héritage multiple. Une classe construite par héritage d'une autre doit être déclarée de la façon suivante :

```
class aClass subclass of anOtherClass is
...
end aClass.
```

Les méthodes définies dans la super-classe sont implicitement héritées et peuvent être surchargées. Une procédure n'est jamais héritée (elle l'est cependant indirectement avec toute méthode qui l'utilise).

L'accès à la forme héritée d'une opération redéfinie est possible par le biais d'un mécanisme de qualification comparable à celui de Owl. L'héritage de Guide étant limité pour l'instant à un héritage simple, le qualificateur est unique, il s'agit du mot clé ***super*** :

```
class SC is
  ...
  method Op (...)
  begin
    ...
  end Op ;
  ...
end SC ;

class C is SC
  ...
  method Op (...)                    // surcharge
  begin
    ...
  end Op ;
  ...
  method example (...)
  begin
    super'Op (...) ;          // appel de Op définie dans SC
  end example ;
  ...
end aClass ;
```

La forme héritée d'une opération surchargée est considérée comme une procédure et non pas comme une méthode définie sur l'objet. Cette raison justifie la forme de l'instruction d'appel qui respecte celle d'un appel de procédure et ne fait pas mention d'un objet cible.

La classe Top

Les propriétés communes à tous les objets de Guide sont définies par le biais de la classe prédéfinie *Top*, super-classe implicite de toutes les autres. Cette classe définit par exemple des méthodes permettant de connaître dynamiquement la taille de l'objet et le nom de son type ou de sa classe (cf 5.5 et Annexe I).

4.4 SYNTHESE

Des mécanismes d'héritage présentés ci-dessus, on peut remarquer la puissance d'expression et le degré de réutilisation de code que fournissent les trois mécanismes suivants :

- le mécanisme de classes différées,
- le mécanisme de déclaration par association,
- l'orthogonalité entre l'héritage de réalisation et l'héritage d'interface.

Les classes différées

La notion de classe différée permet une factorisation maximum (interface, variables d'état et code d'opérations) des propriétés communes à un ensemble d'objets de classes différentes. Nous verrons dans la section suivante que la notion de type (interface) telle qu'elle est définie dans Guide permet uniquement l'expression du partage d'interface et le modèle Guide s'avère donc insuffisant au regard des outils de reutilisation qu'il est possible de fournir dans un modèle à objets.

Déclaration par association

La puissance du mécanisme de déclaration par association apporte un réel confort de programmation tel que l'illustre l'exemple de la classe *Number* donné au chapitre II. La possibilité de déclarer une variable de même classe que l'objet courant *<v : like current>* permet notamment d'exploiter des similitudes de comportement entre des objets de réalisations différentes comme le montre également les classes *Linkable[T]* et *BiLinkable[T]* décrites ci-dessous. Ces deux classes sont extraites de la bibliothèque Eiffel. Elles permettent de définir les listes chaînées à simple chaînage et double chaînage en décrivant respectivement les éléments qui les constituent. Les objets qui constituent les

listes à double chaînage (*BiLinkable*) sont définis à partir de ceux qui constituent les liste à simple chaînage (*Linkable*) :

```
class Linkable[T] export
  value, right, change_value, change_right
feature
  value : T ;
  right : like current ;

  Create (initial : T) is
    do value := initial end ;

  change_value (new : T) is
    do value := new end ;

  change_right (other : like current) is
    do right := other end ;
end ; -- class Linkable[T]

---

class BiLinkable[T] export
  value, right, left, change_value, change_right, change_left
inherit
  Linkable[T]
    redefine right, change_right
feature
  right, left : like current ;

  change_right (other : like current) is
    do
      right := other ;
      if not other.Void then  -- test si other est initialisée
        other.change_left (current)
      end
    end ;

  change_left (other : like current) is
    do
      left := other ;
      if (not other.Void) and (other.right /= current) then
        other.change_right (current)
      end
    end ;
end ; -- class BiLinkable[T]
```

Héritage caché

L'orthogonalité entre l'héritage de réalisation et l'héritage d'interface telle qu'elle existe dans Eiffel permet de réaliser ce qu'on pourrait appeler un **héritage caché**. Ce mode d'héritage permet par exemple de définir les objets listes réalisés sous forme de tableaux de taille fixe par une classe *FixedList* construite par héritage de la classe *List* et de la classe *Array*, *List* étant une classe différée qui laisse à la charge de ses descendantes la spécification du modèle de l'état de leurs exemplaires. Les opérations exportées par la classe *Array* n'ont pas de raison de l'être par la classe *FixedList* et ne sont donc pas

mentionnées dans la clause d'exportation de cette dernière. Cette façon de faire présente l'avantage suivant : elle permet d'exprimer le fait qu'une liste réalisée par un tableau est un[1] tableau (ce qui justifie l'héritage) mais ne doit pas être utilisée comme tel par les programmeurs (l'héritage est caché : les opérations autorisées sur les tableaux ne sont pas visibles sur la liste).

5. MISE EN ŒUVRE DU CONTROLE STATIQUE

La **règle de conformité** qui définit le contrôle statique mis en œuvre dans les quatre modèles présentés dans ce chapitre a été introduite et spécifiée au chapitre II. Elle permet de contrôler dès la compilation que les objets sont toujours utilisés conformément à leur spécification, c'est-à-dire par l'intermédiaire des opérations définies dans leur interface.

Les sections suivantes présentent l'exploitation de la règle de conformité telle qu'elle est faite dans Emerald, Guide, Owl, et Eiffel, et ses implications sur les mécanismes d'héritage fournis dans les trois derniers.

5.1 CONFORMITÉ DANS EMERALD ET DANS GUIDE

Dans Emerald et dans Guide, les domaines de définition des variables qui permettent de désigner les objets sont définis par des interfaces. Ce modèle est rendu possible par la séparation textuelle qui existe entre la déclaration de l'interface des objets et celle de leur réalisation (type abstrait/type concret - type/classe) et permet notamment de fournir la notion de vues (cf chapitre II).

La relation de conformité est définie sur l'espace des interfaces (espace des types abstraits dans Emerald et des types dans Guide ; j'utilise dans la suite la terminologie Guide). La déclaration d'une variable doit donc mentionner le nom d'un type. Une variable v déclarée de type T peut désigner l'ensemble des objets de type T ou d'un type conforme à T.

Héritage et conformité dans Guide

Le mécanisme d'héritage fournit dans Guide permet l'exploitation du partage de comportement entre objets. Héritage implique donc naturellement partage d'interface et en

[1] la relation 'est un' est définie au chapitre II.

conséquence une relation de conformité entre chaque type construit par héritage et son super-type.

Les types Top et Bottom de Guide

Les types de Guide sont organisés selon une hiérarchie dont le type *Top* est le sommet. Ce type réalise la classe de même nom définie comme la super-classe implicite de toutes les autres (cf 4.2). Le type *Top* décrit donc l'interface commune à tous les objets du système : tous les autres types lui sont conformes.

Parallèlement, le type *Bottom* est défini comme étant conforme à tous les autres types ; il est la base de la hiérarchie des types. Il n'existe qu'un seul objet de type *Bottom* ; il peut être désigné dans une application par la constante prédéfinie ***nil***. Comme le type de l'objet que cette constante désigne est conforme à tous les types, sa valeur peut être affectée à n'importe quelle variable. Une variable non initialisée est égale à *nil*. L'appel d'une opération sur une telle variable provoque une erreur à l'exécution.

L'introduction des types *Top* et *Bottom* permet de définir d'une façon cohérente les opérations communes à tous les objets du système et la notion de variable non initialisée.

L'instruction TypeCase et l'opération Affect de Guide

Le respect de la règle de conformité dans les affectations conduit à une généralisation progressive des variables et il est parfois nécessaire de 're-spécialiser une vue' sur un objet de façon à pouvoir accéder à l'ensemble des opérations définies dans son interface. L'intruction *TypeCase* et l'opération *Affect* sont fournies dans ce but et provoquent un contrôle dynamique de conformité.

L'instruction *TypeCase* permet la sélection et l'exécution d'une séquence d'instructions sur la base du type de l'objet désigné par la référence donnée en sélecteur :

```
typecase ref of
   T : ... end ;
   V : ... end ;
   ...
   other : ...
end ;
```

La branche choisie est celle désignée par le nom du type de l'objet désigné par *ref*, si elle existe ; dans le cas contraire et si elle est définie, la clause *other* est exécutée. Dans le bloc d'instructions associé à un type *T*, la référence *ref* est considérée de type *T*.

L'opération *Affect* permet d'affecter la valeur d'une référence à une autre sur la base du type effectif de l'objet qu'elle désigne :

```
ref := Affect (ref') ;
```

Si le type de l'objet désigné par *ref'* n'est pas conforme à celui de *ref*, l'exécution de l'instruction provoque une erreur à l'exécution.

Le contrôle dynamique de conformité est notamment intéressant lorsqu'une application récupère des références à des objets par appel à un service de cataloguage général comme le service de noms par exemple. L'objet serveur de noms permet en effet d'associer un nom symbolique à n'importe quel objet et de récupérer les références aux objets catalogués à partir de leur nom. De façon à fournir un service vraiment général, cet objet gère des références de type *Top* et après récupération de la valeur d'une telle référence, les applications doivent avoir recours au contrôle dynamique de conformité pour avoir accès aux opérations spécifiques de l'objet que cette référence désigne (cf 6.2).

5.2 Conformité dans Owl et Eiffel

Dans Owl et dans Eiffel, les domaines de définition des variables qui désignent les objets sont définis par des classes (ou des types dans Owl ; j'utilise la terminologie d'Eiffel dans la suite de cette section commune). Dans ces deux modèles comme dans Guide, l'héritage permet l'exploitation du partage de comportement entre objets et en conséquence doit respecter le partage d'interface : une classe construite par héritage doit donc être conforme à toutes ses super-classes.

La relation de conformité est définie sur l'espace des classes comme l'union entre la relation de partage d'interface et d'héritage. La déclaration d'une variable doit mentionner le nom d'une classe. Une variable v déclarée de classe C peut désigner l'ensemble des objets de classe C ou d'une classe descendante de C (ce qui est équivalent à dire 'ou d'une classe conforme à C').

Héritage et conformité dans Owl

Comme nous venons de le voir, la construction d'un type T par héritage d'un ou plusieurs autres types n'est valide que si T est conforme à chacun de ses super-types.

La résolution des conflits inhérents à l'héritage d'opérations de même nom lors d'un héritage multiple doivent être résolus par le programmeur. L'héritage de plusieurs

opérations de même nom n'est donc possible que s'il existe entre elles une relation de conformité.

Héritage et conformité dans Eiffel

Dans Eiffel comme dans Owl, une classe *C* construite par héritage d'une ou plusieurs autres classes doit être conforme à chacune de ses super-classes.

Les conflits provoqués lors d'un héritage multiple par un héritage simultané de plusieurs attributs ou opérations de même nom doivent être résolus par le programmeur à l'aide des possibilités de renommage et de redéfinition.

Commentaires sur le contrôle statique en Eiffel

Dans Eiffel, la conformité n'est pas réellement mise en œuvre telle que je l'ai présentée ci-dessus. Bien qu'elle soit définie de cette façon et que l'héritage soit présenté tel que je l'ai introduit dans les sections précédentes, le contrôle statique mis en œuvre dans le langage Eiffel est parfois insuffisant et conduit à l'acceptation de situations incorrectes. Ces situations sont provoquées par l'exploitation du mécanisme de déclaration par association et par l'orthogonalité entre héritage de réalisation et héritage d'interface :

• lorsqu'un argument *a* d'une opération *op* est déclaré *<a : like current,>* dans une classe *C*, il est implicitement redéclaré *<a : C'>* dans toute classe *C'* construite par héritage de *C* : cette redéfinition est contraire aux conditions de conformité et peut de ce fait conduire à des situations erronées,

• l'héritage caché est également contraire aux conditions de conformité qui imposent que toute opération appartenant à l'interface d'une classe appartiennent également à l'interface de chacune de ses sous-classes.

L'exemple donné ci-dessous illustre ces deux points :

```
class A export
  op1, op2, op3
feature

  obj' : A ;
  obj  : like current ; -- équivalent à <obj : A>

  op1 is
    do
      obj := obj' ;
    end -- op1 ;
  op2 (a : A) is
    do ... end ;
  op3 is
    do ... end ;
end -- class A
```

```
class B export
  op1, op2, op4                -- op3 n'est plus exportée
inherit
  A redefine op2 ;
feature

  -- attribut hérité non redéfini
  -- obj' : A ;
  -- attribut hérité implicitement redéfini <obj : B>
  -- obj  : like current ;

  -- op1 is                    -- opération héritée
  --   do                      -- non redéfinie
  --     obj := obj' ;
  --   end -- op1 ;

  op2 (b : B) is               -- opération héritée
    do                         -- argument et code redéfinis
      op4 ;
    end -- op2 ;

  -- op3 is                    -- opération héritée
  --   do ... end ;              -- non redéfinie

  op4 is                       -- nouvelle opération
    do ... end ;                 -- propre aux exemplaires de B

end -- class B
```

La classe *B* construite par héritage de la classe *A* est considérée dans Eiffel comme étant conforme à cette dernière et les séquences d'instructions suivantes sont donc acceptées par le compilateur :

```
a, a' : A ;
b : B ;

-- séquence 1
-- illustre un problème lié à l'orthogonalité entre
-- héritage et exportation

b.Create ;
a := b ;
a. op3 ;        -- violation de l'interface
                -- de l'objet de classe B désigné par a

-- séquence 2
-- illustre le problème lié à la possibilité de redéfinir
-- un argument d'une opération

a.Create ;
b.Create ;
a' := b ;
a'.op2 (a) ;  -- l'opération op4 appelée par op2 n'est pas
              -- définie sur l'objet de classe A désigné
              -- par l'argument a
```

La séquence 2 illustre aussi un des problèmes lié à la déclaration de variables par association (l'argument aurait pu être déclaré de type *<like current>*). Un autre problème lié à ce type de déclaration est illustré par l'instruction contenue dans l'opération *op1* telle qu'elle est définie dans la classe *B* : *<obj := obj'>* ; dans cette classe, le type de *obj'* n'est plus conforme au type de *obj* et l'affectation devrait être refusée par le compilateur, ce qui n'est pas le cas. Les séquences 1 et 2 sont acceptées par le compilateur. L'exécution de la première ne pose pas de problème puisque l'opération *op3*, bien que normalement non visible, est définie sur les objets de classe *B*. Lors de l'exécution de la séquence 2 par contre, la recherche de l'opération *op4* sur un objet de classe *A* provoque un comportement incohérent de l'application puisque l'inexistence de l'opération n'a pas été détectée[1].

5.3 SYNTHESE

Ces remarques sur Eiffel illustrent l'utilité du contrôle statique et des mécanismes de déclaration par association et d'héritage caché. Les concepteurs d'Eiffel n'ont en effet pas voulu privilégier l'un aux dépens de l'autre et de fait, le modèle obtenu présente des incohérences. Pour B. Meyer [News 88], ces incohérences ne portent pas à conséquences car d'après lui les cas qui les provoquent ne se rencontrent jamais dans les applications ! Il a cependant annoncé un document de mise au point sur le modèle de conformité et d'héritage dans Eiffel et il est probable qu'il y introduise la notion d'héritage non conforme telle que je la propose ci-dessous pour Guide[2].

Propositions pour le modèle de types et d'héritage dans Guide

La **séparation entre type et classe** doit être conservée. Elle présente les avantages suivants :

- possibilité de définir plusieurs réalisations différentes pour une même interface,
- possibilité de définir des vues.

Le premier avantage est notamment exploité pour l'association de différents schémas de synchronisation à des objets de même type (cf 11.2). Les classes différées présentent également ce premier avantage. Par contre, les vues ne sont pas réalisables dans les modèles Owl et Eiffel[3].

---

[1] Ces deux séquences ont été testées sur la version 2.1 du compilateur Eiffel.

[2] C'est d'ailleurs ce que proposent la plupart des utilisateurs d'Eiffel qui se sont exprimés dans [News 88].

[3] B. Meyer prétend pourtant le contraire dans son livre mais l'exemple qu'il donne est inexploitable.

Guide doit fournir les mécanismes de **déclaration par association** et d'**héritage caché**. Ces deux mécanismes ne doivent cependant pas être introduits au détriment de la validité du contrôle statique effectué ce qui nécessite la définition de la notion d'**héritage non conforme**. L'héritage caché et l'héritage d'une classe dont un argument d'une opération est déclaré *<a : like current>* sont par exemple des cas d'héritage non conforme.

Enfin la notion de **classe différée** doit être définie dans Guide. Elle fournit un outil de factorisation d'interface, de variables d'état et d'opérations plus puissant et donc plus intéressant sur ce plan que l'outil de factorisation d'interface fournit par la séparation entre type et classe.

6. OBJETS PERMANENTS

On qualifie d'**objet permanent** un objet dont la durée de vie n'est pas liée à celles des applications qui l'utilisent. Outre les possibilités de stockage de l'information d'une exécution à l'autre d'une même application, un support de conservation d'objets permet aussi le partage d'information entre différentes applications.

Classiquement, les fonctions de conservation sont réalisées par le biais d'un système de gestion de fichiers ou d'un système de gestion de base de données. Eiffel utilise ainsi le système de gestion de fichiers d'Unix. Dans Guide par contre, les objets sont conservés par le biais du mécanisme de gestion d'objets intrinsèque au système.

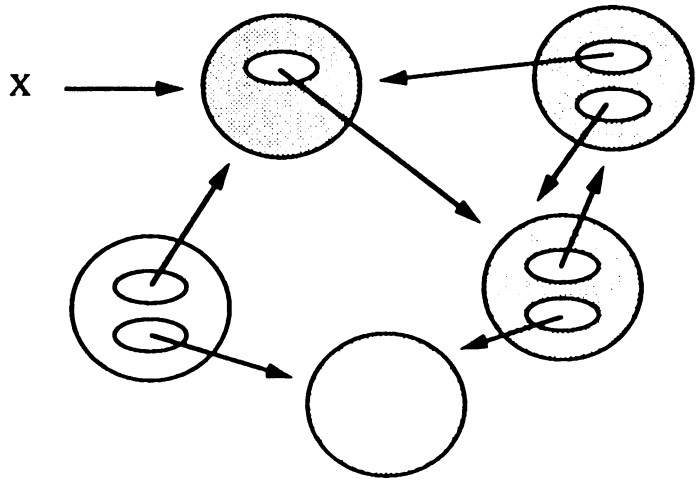
6.1 LA CLASSE *STORABLE* D'EIFFEL

Le support pour le stockage et la récupération de l'état d'un objet Eiffel est offert par le biais d'une classe de la bibliothèque : *Storable*. Les classes d'objets pour lesquelles les fonctions correspondantes sont nécessaires doivent en conséquence être construites par héritage de cette classe.

La classe *Storable* définit deux opérations, *store* et *retrieve*, qui permettent de recopier l'état d'un objet dans le contenu d'un fichier Unix, et réciproquement de récupérer l'état d'un objet à partir du contenu d'un fichier Unix :

```
anObject.store (<file_name>) ;
anObject.retrieve (<file_name>) ;
```

Dans le cas d'un objet composé (cf 7), l'état de chaque composant, sous-composant, etc est mémorisé. La structure issue d'un objet composé est illustrée sur la figure suivante :



Composants et sous-composants de l'objet désigné par X

Lors de la récupération de l'état d'un objet par l'opération *retrieve*, la classe de ce dernier doit être impérativement la même que celle de l'objet dont l'état a été précédemment mémorisé dans le fichier dont le nom est donné en argument de l'opération.

6.2 La Mémoire Permanente et le Service de Désignation de Guide

Le modèle Guide est mis en œuvre sur un système d'exploitation propre dans lequel les aspects relatifs à l'exécution et ceux touchant à la conservation permanente de l'information ont été intégrés.

Les objets permanents

Tout objet Guide est potentiellement permanent, et n'est susceptible d'être détruit que lorsqu'il devient inaccessible, c'est-à-dire lorsqu'il n'existe plus dans le système aucun autre objet qui le désigne. Un mécanisme de ramasse-miettes doit être mis en œuvre pour la destruction automatique des objets inaccessibles [Nguyen Van 89].

Bien qu'elle ne soit pas indispensable, une opération de destruction explicite est fournie dans le but de permettre au programmeur de détruire manuellement les objets qu'il sait inutiles. Cette opération de nom *destroy* est définie dans le type *Top* et de ce fait sur n'importe quel objet.

Le Service de Désignation

Le service de désignation symbolique de Guide permet d'associer un nom symbolique à un objet. Il est mis en œuvre par un objet prédéfini, permanent par définition. Associer un nom à un objet via ce service garantit donc de fait la pérennité de l'objet.

Une première étude sur l'organisation du service de désignation est présentée dans [Freyssinet 87]. Un premier prototype basé sur la gestion d'un espace de noms hiérarchique est en cours de réalisation [Vandôme 88].

Le service fourni par la version actuelle du système est simple et gère un espace de noms plat. La constante prédéfinie ***catal*** permet de désigner dans les applications l'objet prédéfini qui le met en œuvre. Sur cet objet sont notamment définies les opérations suivantes[1] :

```
Insert (in name : String [80] ; object : ref Top)
   // mémorise sous le nom donnée en premier argument
   // la référence de l'objet donnée en second argument

Search (in name : String [80]) : ref Top
   // rend une référence sur l'objet catalogué
   // sous le nom donné en premier argument
   // rend 'nil' si le nom donné n'existe pas

Delete (in name : String [80])
   // supprime du catalogue le nom donné en argument
   // et la référence associée
```

L'ensemble des classes connues du système à un instant donné sont des objets permanents catalogués dans l'objet désigné par la constante *catal*. Une instruction de création d'objet *<aClass.New>* provoque implicitement un appel de l'opération *Search* définie sur cet objet.

L'opération *Search* rend une référence de type *Top*. Elle ne peut donc être utilisée que par le biais de l'opération *Affect* qui force un contrôle dynamique de conformité :

```
r : ref T ;
...
r := Affect (catal.Search ("marieMailBox")) ;
```

[1] La notion de référence est définie en 7.4.

6.3 SYNTHESE

Destruction explicite d'un objet Guide

La définition de l'opération de destruction d'un objet Guide comme une opération définie sur l'objet lui-même, me paraît peu cohérente avec l'ensemble du modèle. L'exécution de cette opération sur un objet désigné par une référence *v* : *v.destroy* modifie en effet la valeur de *v*, ce qui est contraire à la sémantique des appels d'opération sur les objets.

La destruction d'un objet consiste non seulement en une modification de la référence qui le désigne mais aussi en une modification de l'objet. Définir l'opération correspondante comme une opération propre aux références ne serait donc pas non plus une solution cohérente. L'opération *destroy* est en fait comparable à l'opération *new* et devrait donc être définie comme une opération propre aux objets classes.

Objets permanents et cohérence

A partir du moment où la durée de vie d'un objet peut être supérieure à celle de l'application qui l'a créé, se pose le problème du maintien de la cohérence entre les objets dont l'état a été mémorisé et les classes de ces objets telles qu'elles sont définies dans le système. Ce problème de maintien de la cohérence est crucial dans le système Guide où tout objet est potentiellement permanent. La modification d'un type ou d'une classe intégré au système doit être systématiquement contrôlée et limitée aux modifications qui ne mettent pas en cause la validité des éventuels types et classes et exemplaires construits sur la base de la première version de l'élément modifié. Un mécanisme de gestion de versions doit être intégré [Scioville 89] ; il permettra notamment la réalisation du premier gestionnaire de types et de classes dont la définition est en cours [Jamrozik 89].

7. OBJETS ÉLÉMENTAIRES, OBJETS COMPOSÉS ET CONSTANTES

Un objet est défini par composition d'autres objets comme une structure de données complexe est définie par composition de structures de données plus simples. Les objets élémentaires comparables aux structures de données primitives des langages de programmation classiques (caractères, valeurs numériques, valeurs booléennes) sont définis par un ensemble de classes prédéfinies. Suivant le modèle à objets considéré, ces classes et les objets primitifs qu'elles permettent de créer sont appelés classes et objets **élémentaires**, classes et objets **de base**, ou encore classes et objets **simples**.

Tout objet non élémentaire est construit par composition d'autres objets, élémentaires ou non et peut être qualifié d'**objet composé**. Dans les sections précédentes, j'ai présenté le mode de désignation des objets : un objet ne peut être utilisé que par le biais d'une variable comparable à un pointeur, et la sémantique du passage d'objets en arguments ou résultats d'une opération est celle d'un passage par référence. Ce schéma de désignation impose la mise en œuvre systématique d'un mécanisme de liaison dynamique lors de l'accès à un objet. Le modèle d'objets Smalltalk est un exemple de modèle parfaitement uniforme, dans lequel un tel schéma de désignation est appliqué à tout objet, qu'il soit élémentaire ou non. Une telle uniformité est un avantage non négligeable pour le programmeur mais impose, et notamment lors de l'accès à des objets élémentaires, un sur-coût important en temps d'exécution. Cet inconvénient majeur explique pourquoi dans de nombreux modèles, différentes sémantiques sont proposées en fonction de la nature des objets, élémentaires ou non.

Les objets élémentaires peuvent être représentés directement par leur valeur, et tout appel à un tel objet remplacé statiquement, lors de la compilation, par l'instruction machine à laquelle il correspond. Les objets élémentaires sont en général des objets constants et une telle représentation ne modifie donc pas leur utilisation. Pour un objet constant, la sémantique d'un passage par valeur est en effet équivalente à celle d'un passage par référence. Par contre, le fait que les appels d'opérations sur les objets élémentaires soient compilés, impose une restriction du domaine de définition des variables qui les désignent. Le domaine de définition d'une telle variable est en effet forcément restreint à un ensemble d'objets de même classe.

La représentation directe d'un objet élémentaire par sa valeur est un schéma adopté dans de nombreux modèles. D'autres éléments répondant au même souci d'efficacité sont proposés mais sans toutefois être normalisés. Les quatre modèles étudiés ici proposent chacun une solution qui lui est propre ; cette diversité peut s'expliquer d'une certaine manière, par le fait que les objectifs de conception et les supports d'exécution de ces modèles ne sont pas identiques.

Les sections suivantes présentent les règles de représentation et de composition d'objets de Owl, Emeral, d'Eiffel et Guide, ainsi que les possibilités de déclaration de constantes fournies par ces modèles. Les possibilités de définition de constantes (objets et variables de désignation d'objets) sont en effet directement liées à la nature des objets et des variables mis en œuvre.

7.1 OWL

Le modèle d'objets de Owl est parfaitement uniforme et, en ce sens, comparable à Smalltalk. On y distingue cependant les objets non structurés comparables aux objets élémentaires définis précédemment et les objets structurés, comparables aux objets composés, mais cette distinction n'entraîne aucune différence dans leur définition et leur utilisation.

Déclaration de constantes Owl

Des objets constants de type *Integer*, *Real* et *String* peuvent être représentés par des valeurs littérales comme *5*, *7.09*, et *"Paul"*. Un objet constant est défini comme un objet qui ne possède aucune opération permettant de modifier son état ; c'est donc une propriété propre à son type.

La valeur d'une variable peut être fixée par le programmeur si ce dernier la déclare sous la forme d'un nom. Un **nom** est une référence constante à un objet :

```
define Pi : Real
         := 3.14 ;
define OfficePhone : PhoneNumber
         := Create (PhoneNumber, '76 51 45 26') ;
```

On peut remarquer qu'un nom ne désigne pas forcément un objet constant. Par exemple, l'objet désigné par le nom *OfficePhone* déclaré ci-dessus peut très bien changer de valeur s'il possède une opération de mise à jour.

7.2 EMERALD

Le modèle d'objets d'Emerald est parfaitement uniforme d'un point de vue de l'utilisateur. Tous les objets sont désignés et utilisés selon le même schéma. Le mode de représentation et de gestion de chaque objet est cependant choisi par le compilateur en fonction de caractéristiques propres à l'objet et du gain en efficacité qu'elles permettent d'obtenir. Emerald distingue trois types d'objets : les objets directs, les objets locaux et les objets globaux. Le choix du mode de représentation d'un objet est fait indépendamment du programmeur.

Les objets directs d'Emerald

Un objet direct est représenté par sa valeur et tout appel d'opération sur un tel objet est compilé. Les objets élémentaires (entiers, caractères, booléens) et les tableaux de taille fixe d'objets élémentaires sont des objets directs.

Les objets locaux et les objets globaux d'Emerald

Parmi les objets composés, Emerald distingue les objets globaux et les objets locaux. Il faut rappeler ici qu'Emerald est un modèle à objets destiné à la programmation d'applications distribuées. Les objets qui constituent une application sont donc répartis sur les différents sites du système support et un appel d'opération sur un objet peut nécessiter la mise en œuvre de mécanismes de communication inter-sites si la localisation de l'objet appelé n'est pas la même que celle de l'objet appelant.

Un objet est dit local à celui dans lequel il est défini, si ce dernier n'exporte aucune référence sur lui. Un objet local n'est donc pas partagé. Cette propriété garantit que si un objet local est toujours présent sur le même site que l'objet qui le définit, un appel d'opération sur lui ne nécessitera jamais la mise en œuvre du mécanisme de communication inter-sites : il ne peut en effet être appelé que par l'objet qui le définit, seul objet du système à le connaître. Dans le but de fournir un mécanisme d'appel local plus efficace que le mécanisme d'appel à distance, le système garantit à tout instant la localisation d'un objet et de tous ses objets locaux sur un même site.

Les objets globaux sont les objets potentiellement visibles par tous les autres et de ce fait, suceptibles d'être appelés par un objet distant. Le coût de l'appel à un objet distant est parfois réduit par le déplacement de l'objet appelé sur le site de l'objet appelant.

7.3 EIFFEL

Eiffel distingue les objets simples, qui sont des exemplaires des types simples (*Integer*, *Boolean*, *Real* et *Character*) des autres objets construits par composition.

Les objets simples d'Eiffel

Les objets simples sont des objets constants représentés directement par leur valeur. Les appels d'opérations sur un objet simple sont compilés.

Les objets composés d'Eiffel

Les objets composés sont désignés par des pointeurs typés conformément à la sémantique de la liaison dynamique et du passage par référence.

Déclaration de constantes en Eiffel

Les constantes de types simples peuvent être définies selon le modèle donné ci-dessous :

```
Zero : INTEGER is 0 ;
Percent : CHARACTER is "%" ;
Pi : REAL is 3.151492 ;
```

Les valeurs littérales représentant des constantes de types simples peuvent aussi être utilisées dans les instructions où la valeur d'un objet du même type est requise (comme argument d'une opération par exemple).

Les constantes littérales de type *String* telles que *"Hello"* ou *"Syntax Error"* sont permises également. Nous verrons que bien qu'elles soient traitées comme des expressions (efficacité oblige) elles sont conceptuellement conformes à l'outil proposé pour la définition de constantes de types non simples.

Les fonctions à évaluation unique

Une constante d'un type non simple peut être déclarée sous la forme d'une fonction dont la valeur n'est évaluée qu'une fois, lors du premier appel.

L'exemple suivant présente la déclaration d'une constante de type *Complex* qui désigne l'objet complexe de valeur *i* :

```
i : Complex is              -- objet complexe de partie réelle nulle
  once                    -- et de partie imaginaire égale à 1
    Result.Create (0,1) ;
  end ; -- i
```

Le mot clé *once* qui introduit le corps de la fonction indique qu'elle ne sera évaluée qu'une fois, lors du premier appel, et que la valeur calculée à ce moment là est mémorisée puis retournée à chaque nouvel appel. De même que l'affectation d'un nom à un objet Owl n'implique pas que cet objet soit constant, la seule constante dans la déclaration d'une fonction telle que la fonction *i* décrite ci-dessus, est la valeur de la fonction : si cette valeur est d'un type non simple, l'objet qu'elle désigne n'est pas forcément constant.

La forme littérale permise pour la représentation d'une chaîne de caractères, correspond conceptuellement à la déclaration et à l'appel d'une fonction à évaluation unique. La chaîne *"Hello"* peut ainsi être assimilée à une fonction de nom hello définie de la façon suivante :

```
hello : STRING is
     -- Chaîne de caractères de longueur 5
     -- contenant les caractères "H", "e", "l", "l", "o"
  once
    Result.Create (5) ;
    Result.enter (1, "H") ;
    Result.enter (2, "e") ;
```

```
    Result.enter (3, "l") ;
    Result.enter (4, "l") ;
    Result.enter (5, "o") ;
end ; -- hello
```

Les constantes et les fonctions à évaluation unique déclarées dans une classe peuvent être utilisées par les clients de cette classe si elles sont exportées ; les définitions des constantes propres à un type d'application peuvent ainsi être regroupées dans une même classe.

L'héritage multiple permet une utilisation facile des lots de constantes propres à un type d'application : plutôt que de déclarer les classes qui utilisent des paquets de constantes comme clientes de celle qui les déclare, elles peuvent être déclarées comme sous-classes et gagner ainsi en efficacité.

7.4 GUIDE

Guide distingue les objets composés et les objets élémentaires.

Objets élémentaires de Guide

Les objets élémentaires de Guide sont les exemplaires des classes prédéfinies *Integer*, *Real*, *Cardinal*, *Boolean* et *Char*. Ce sont des objets dont la valeur est constante.

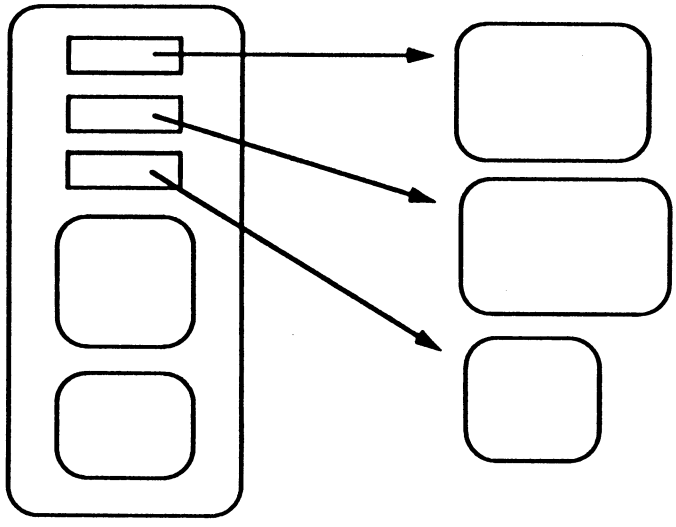
Objets composés de Guide, objets internes et références

Guide exploite les différents liens de composition qui peuvent exister entre un objet composé et ses composants. Deux niveaux de composition sont mis en évidence :

- la **composition de données** : l'état de l'objet composant fait partie intégrante de l'état de l'objet englobant,
- la **composition de références** : seule une variable qui désigne l'objet composant fait partie de l'état de l'objet englobant.

Dans le cas d'une composition de données, l'objet composant est dit interne à l'objet englobant. Un **objet interne** est intrinsèquement lié à l'objet qui le contient ; il est créé et disparait avec lui. Il n'est pas visible à l'extérieur de l'objet qui le contient et ne peut donc être partagé.

Dans le cas d'une composition de **références**, seule la référence est intrinsèquement liée à l'objet englobant ; l'objet composant qu'elle doit désigner, doit lui être explicitement associé. La destruction d'un objet composé de références ne provoque pas la destruction des objets que ces références désignent.



Composition de données et de références

Un objet interne est représenté dans l'état de l'objet qui le contient par les données qui constituent son état. De façon à différencier les variables qui désignent des objets internes de celles qui désignent des objets externes, les secondes sont explicitement déclarées sous la forme de références :

```
reference : ref aType ;
```

Puisqu'un objet interne est représenté directement par sa valeur, sa déclaration doit spécifier non seulement son type mais également sa classe, de façon à ce que le compilateur dispose des informations nécessaires à sa représentation :

```
internal-object : aClass ;
```

L'état d'un objet composé est donc constitué d'un ensemble d'objets internes et d'un ensemble de références, à des objets qui lui sont externes. La sémantique du passage d'un objet interne en paramètre d'une opération est celle du passage par valeur ; par définition l'objet correspondant n'est en effet pas partageable etc'est donc une copie de son état qui est transmise.

L'appel d'une opération sur un objet interne ne nécessite pas la mise en œuvre du mécanisme de liaison dynamique normallement requis lors de l'appel d'une opération sur un objet ; la connaissance de la classe d'un tel objet permet en effet une liaison statique. En pratique, le code des opérations définies sur un objet interne est lié statiquement au code des opérations définies sur l'objet englobant. Cette solution est efficace au regard du temps qu'elle permet de gagner à l'exécution mais le programmeur doit être conscient du

fait que l'utilisation d'objets internes annule le polymorphisme induit par la désignation d'objets via des références typées.

Dans la version actuelle du langage Guide, les objets élémentaires ainsi que les exemplaires des classes prédéfinies *Array* et *String* ne peuvent être utilisés que sous la forme d'objets internes. Les appels d'opérations sur les exemplaires de ces deux classes sont d'ailleurs compilés.

Dans une prochaine version du langage, il est envisagé d'intégrer à la bibliothèque Guide la description des types et des classes élémentaires, ainsi que des constructeurs *Array* et *String*. Ceci permettra aux programmeurs d'utiliser les objets correspondants aussi bien sous la forme d'objets internes que par le biais de références, et notamment d'en concevoir des spécialisations à l'aide des possibilités de sous-typage et d'héritage.

La distinction entre objets internes et références est également appliquée à la déclaration des variables locales d'une méthode ou d'une procédure. Pour distinguer les objets internes des objets déclarés par le biais de ces variables, on appelle ces derniers objets de travail. Un **objet de travail** n'est pas un composant de l'objet dans lequel il est défini, c'est une variable locale d'une procédure ou d'une méthode et sa durée de vie est limitée à la durée de l'exécution de la procédure ou de la méthode dans laquelle il est déclaré.

Déclaration de constantes en Guide

Les constantes Guide peuvent être de deux types : objets (internes ou de travail) ou références. Dans le premier cas, la valeur de l'objet peut être spécifiée sous la forme d'une expression à condition que cette dernière puisse être évaluée par le compilateur.

Dans la version courante du langage, les objets internes ou de travail constants sont soit des objets élémentaires, soit des chaînes de caractères.

Exemple

```
const size : Integer = 10*itemNbr;
const Pi : Real = 3.14;
const old_age : Cardinal = 99;
const bubble : Char = "°";
const ok : Boolean = true;

const system_name : String[5] = "Guide";
```

Les valeurs littérales représentant des objets élémentaires et des chaînes de caractères peuvent être utilisées dans toutes les instructions où la valeur d'un objet de même classe est requise.

Des identificateurs réservés permettent de désigner des objets constants prédéfinis. Par exemple, ***true*** et ***false*** désignent respectivement les deux exemplaires de la classe *Boolean*, ***nil*** désigne l'unique objet de type *Bottom* (cf 5.5), et ***catal*** désigne l'objet prédéfini qui fournit le service de désignation du système (cf 6.2).

Les constantes de type référence sont définies par le biais du service de désignation fournit par l'objet désigné par la constante prédéfinie *catal*. Ce service permet notamment d'associer un nom à un objet, ou de récupérer à partir d'un nom, une référence sur l'objet catalogué sous ce nom.

La référence déclarée ci-dessous a une valeur constante calculée à la compilation par un appel au service de désignation :

```
const myMailBox : ref MailBox = "marieMailBox";
```

La compilation d'une telle déclaration signale une erreur de conformité si le type de l'objet catalogué sous le nom *"marieMailBox"* n'est pas conforme au type *MailBox*.

7.5 SYNTHESE

La sémantique d'accès aux objets via des références conduit naturellement à un modèle dans lequel un objet dont une référence est passé en paramètre d'une opération est partagé entre l'objet appelant et l'objet appelé, et peut donc indifféremment être modifié par l'un ou l'autre. Cette possibilité de partage n'est cependant pas toujours souhaitable. Considérons par exemple un objet catalogue par le biais duquel on peut associer un nom symbolique à d'autres objets. La chaîne de caractères transmise au catalogue par un objet qui en catalogue un autre, est typiquement un objet qui ne doit pas être partagé entre l'appelant et le catalogue.

Pour répondre à cette nécessité, Eiffel donne la possibilité de dupliquer un objet. Charge reste dans ce cas au catalogue de dupliquer les objets chaînes de caractères qu'on lui transmet de façon à en garantir l'intégrité. Cette solution nécessite une vigilance constante de la part du programmeur et me paraît de ce fait, peu facile à exploiter.

Les objets internes de Guide sont non partageables par définition, mais constituent une solution qui peut paraître peu satisfaisante aux programmeurs. La définition du

concept d'objet interne répond en fait à deux nécessités : celle de pouvoir travailler sur des objets non partagés, et celle de gagner en efficacité par la réalisation des appels d'opération à ces objets sous la forme d'appels de procédure. Ce double objectif a conduit à la conception d'un modèle d'objets non uniforme dont la maîtrise n'est pas évidente pour un programmeur non averti. On s'aperçoit cependant que la notion d'objets internes est nécessaire dans Guide du fait de sa mise en œuvre sur un système réparti multi-utilisateurs. Alors que l'ensemble des classes qui constituent une application Owl ou Eiffel sont assemblées statiquement en un seul bloc de code exécutable, les classes nécessaires à une application Guide sont chargées dynamiquement en mémoire à la demande. La recherche dynamique du code des opérations appelées sur les objets (liaison dynamique - cf chapitre II) est donc forcément plus couteuse dans Guide qu'elle ne l'est dans Eiffel et Owl et justifie de ce fait l'importance de la notion d'objet interne[1].

8. CONSTRUCTEURS DE CLASSES

Un constructeur est un modèle pour la définition d'un ensemble de classes paramétrées. La possibilité de paramétrer des classes est importante car elle évite des définitions redondantes. Les tableaux et les n-uplets sont des exemples d'objets généralement construits à l'aide de classes paramétrées ; *Stack [T]* et *List [T]* sont des exemples de constructeurs.

Les constructeurs ont été introduits au chapitre II. Les sections suivantes présentent leur mise en œuvre dans les quatre modèles étudiés.

8.1 LES CONSTRUCTEURS DE OWL

Un type Owl peut être paramétré par un ou plusieurs autres types. Un exemple de constructeur est donné ci-dessous :

```
type_module Set [ElementType : Type]
  ...
  operation insert (me, new : ElementType) ;
    is
    begin
      var x : ElementType ;
      var y : Sequence [ElementType] ;
      ...
    end ;
  ...
end type_module ;
```

[1] Sur le prototype du système disponible début mars 1989, un appel d'opération sur un objet externe était 50 fois plus couteux qu'un appel d'opération sur un objet interne.

Dans le corps du constructeur, le nom donné à un type paramètre permet de le désigner. Ci-dessus, le nom du type paramètre (*ElementType*) est par exemple utilisé pour spécifier le type du second argument de l'opération *insert* et pour la déclaration de deux variables locales de cette opération.

Un constructeur peut être construit par héritage d'un autre constructeur ; le constructeur *Set [ElementType]* peut ainsi être construit par héritage du constructeur *Collection [ElementType]*.

La définition d'un constructeur est correcte si toute utilisation légale de ce constructeur produit une définition de type correcte. Un type paramètre peut être défini par le mot clé *Type* ou par une condition *subtype (P)*, où *P* désigne un type (par exemple *TableTriée [Elt : subtype (Comparable)]*. Si un type paramètre *T* est défini par le mot clé *Type*, il peut être remplacé par n'importe quel type lors d'une utilisation du constructeur. Dans le corps du constructeur, les possibilités d'utilisation des variables déclarées du type paramètre *T* sont donc limitées, puisqu'elles doivent être valides pour tous les types. Si un type paramètre *T* est défini par une condition *subtype (P)*, il peut être remplacé lors d'une utilisation du constructeur par n'importe quel type *P'*, à condition que *P'* soit sous-type de *P*. Cette restriction est intéressante car elle permet de supposer que les variables déclarées du type paramètre *T* dans le corps du constructeur possèdent au moins l'interface spécifiée par le type *P*.

Les possibilités d'utilisation dans le corps d'un constructeur des variables déclarées d'un type paramètre *T* sont donc dépendantes de la déclaration de *T*. Les conditions de validité d'une déclaration de constructeur sont énoncées par la règle donnée ci-dessous.

Conditions de validité d'un constructeur

La déclaration d'un constructeur Const[...,Elt : Type,...] n'est valide que si l'utilisation de chaque variable e déclarée de type Elt est restreinte aux cas mentionnés ci-dessous :

- e est utilisée en partie gauche d'une affectation et l'expression donnée en partie droite est de type Elt,
- e est utilisée en partie droite d'une affectation et l'entité donnée en partie gauche est de type Elt,
- e est utilisée comme paramètre d'une opération et le type du paramètre formel correspondant est Elt (ce qui implique que cette opération soit déclarée dans le constructeur),
- e est utilisée dans une expression booléenne de la forme 'e = t', 'e /= t', 't = e', ou 't /= e' et t est de type Elt.
- e est utilisée comme cible d'une instruction d'appel d'opération telle que <e.op (...)>, alors la déclaration du type paramètre formel Elt doit être contrainte par une condition Elt subtype(aType) et une opération op doit être définie dans le type

```
aType ; arguments et résultats effectifs doivent de plus respecter
les conditions de conformité.
```

Les constructeurs Owl sont mis en œuvre par macro-génération et substitution. Les noms définis dans un constructeur font cependant exception à cette règle (un nom est un désignateur d'objet dont la valeur est constante - cf 7.1) ; un nom déclaré dans le corps d'un constructeur est partagé par tous les types issus du constructeur. La déclaration d'un nom ne peut donc pas faire mention d'un type paramètre de généricité.

8.2 Les constructeurs d'Emerald

Types abstraits et types d'Emerald sont considérés comme des objets. Un constructeur de types est tout simplement un objet qui possède une opération capable de produire un type. La notion de constructeur de types n'est donc pas explicitement définie dans le modèle.

Un exemple de constructeur Emerald est l'objet prédéfini *Array*. Cet objet possède une opération *of* dont l'unique argument est un type abstrait et qui produit en résultat un objet possédant une opération *Create* sans argument. Le résultat de l'appel *<Array.of[T].Create>* est un objet tableau dont les éléments sont du type abstrait *T* ou d'un type conforme à *T*.

8.3 Les constructeurs d'Eiffel

Une classe Eiffel peut être paramétrée par une ou plusieurs classes : *Set [T]* et *TableTriée [Elt->Ccomparable]* sont des exemples de constructeurs de classes.

Eiffel fournit des possibilités de généricité universelle ou contrainte comparables à celles de Owl (cf 8.1).

Eiffel ne duplique pas le code des constructeurs qui, pour un constructeur donné est partagé par toutes les classes qui en sont issues. Cette mise en œuvre impose notamment le partage des fonctions à évaluation unique (cf 7.3) qui de ce fait ne peuvent produire un résultat dont le type est un paramètre de généricité.

8.4 Les constructeurs Guide

Un constructeur Guide peut être paramétré par plusieurs types, définis par le mot clé *of* dans l'en-tête de sa déclaration ; ce mot clé met en évidence le fait que les types paramètres permettent notamment de définir les composants, références et objets internes, des objets qui seront issus du constructeur. Les constructeurs de Guide sont comparables

à ceux de Owl et Eiffel à la différence cependant qu'ils tiennent compte de l'existence des objets internes.

Un exemple de constructeur de type est donné ci-dessous ; le constructeur de classe correspondant est décrit à la suite.

```
type constructor SimpleBox of [Element : T] is
  method put (in Element) ;
  method get : Element ;
end SimpleBox.

class constructor SimpleBox of [Element : T]
implements SimpleBox of [Element : T] is

  component : Element ;

  method put (in newcomponent: Element ) ;
    begin component := newcomponent end put ;

  method get : Element  ;
    begin return component end get ;

end SimpleBox.
```

Lors de l'utilisation d'un constructeur, un type *T* paramètre formel de généricité peut être remplacé par un type référence *<ref T'>* ou par une classe *<C>*. Conceptuellement lors de l'utilisation du constructeur, l'identificateur de chaque paramètre formel de généricité est textuellement remplacé par le paramètre effectif *<ref aType>* ou *<aClass>* qui lui correspond. Cette double possibilité est due à la distinction faite dans Guide entre les références et les objets internes qui constituent l'état d'un objet composé. Les constructeurs permettent de définir des structures de données paramétrées ; toutes les solutions possibles pour la réalisation de ces structures, composition de références et composition de données, doivent donc être fournies.

Le constructeur *SimpleBox of []* décrit précédemment permet ainsi la définition d'objets boîtes composés d'une référence ou d'un objet interne. Par exemple, les descriptions des types *SimpleBox [ref aType]* et *SimpleBox [aClass]* construits à l'aide du constructeur *SimpleBox* décrit précédemment sont les suivantes :

```
type SimpleBox of [ref aType]
is
  method put (in ref aType) ;
  method get : ref aType ;
end SimpleBox.

type SimpleBox of [aClass]
is
  method put (in aClass) ;
```

```
  method get : aClass ;
end DoubleBox.
```

L'utilisation d'un constructeur Guide n'est valide que si les types et classes choisis comme paramètres effectifs de généricité sont conformes au paramètre formel auquel ils correspondent. *SimpleBox of [ref aType]* n'est une utilisation correcte du constructeur *SimpleBox of [Element : T]* que si le type *aType* est conforme au type *T*. De même, *SimpleBox of [aClass]* n'est une utilisation correcte du même constructeur que si la classe *aClass* réalise un type conforme à *T*.

Cette restriction est intéressante car comme dans Owl et Eiffel elle permet d'élargir dans le corps d'un constructeur de classe, les possibilités d'utilisation d'une variable déclarée d'un type paramètre ; une telle variable possède en effet les propriétés du type spécifié comme paramètre formel : les conditions de validité des déclarations de constructeurs de classes Guide sont donc identiques à celles énoncées en section 8.1.

Il est intéressant de pouvoir paramétrer une structure de données par une valeur qui spécifie sa taille ; les tableaux et les chaînes de caractères par exemple, sont typiquement des structures de données dont la taille n'est pas fixe. Dans Guide, ces structures sont définies par le biais de constructeurs. Un constructeur de classes peut en effet posséder, en plus des paramètres types précédemment décrits, une ou plusieurs valeurs entières comme paramètres de généricité. Cette possibilité permet par exemple de définir facilement les boîtes de taille n :

```
type constructor Box of [Element : T] is
  method put (in Cardinal, Element) ;
  method get (in Cardinal) : Element ;
end Box.


class constructor Box [size : Cardinal] of [Element : T]
implements Box of [Element : T] is

  component : Array [size] of [Element] ;

  method put (in nbr : Integer, newcomponent: Element ) ;
    begin component[nbr] := newcomponent end put ;

  method get (in nbr : Integer) : Element  ;
    begin return component[nbr] end get ;

end Box.
```

Une valeur entière paramètre de généricité, telle que *size* dans l'exemple ci-dessus, est un paramètre propre à la structure interne des exemplaires de la classe paramétrée ; elle n'intervient pas dans la spécification de leur type. C'est la raison pour laquelle les

paramètres valeurs apparaissent uniquement dans la déclaration des constructeurs de classes. Les tableaux et les chaînes de caractères sont des exemples d'objets Guide définis par le biais de constructeurs.

Les constructeurs ne sont pas fournis par la version courante du compilateur Guide. On peut cependant remarquer qu'une duplication de code par macrogénération est nécessaire dans les cas où une classe issue d'un constructeur est utilisée pour la définition d'objets internes ou de travail.

8.5 SYNTHESE

Les constructeurs fournis par Owl, Eiffel et Guide offrent des possibilités comparables. On peut cependant remarquer que la définition du type *Top* dans Guide (sommet de la hiérarchie des types) permet une syntaxe uniforme dans la déclaration des paramètres formels de généricité : *[..., Elt : aType, ...]* alors que dans les modèles Owl et Eiffel, deux formes différentes sont requises selon que le paramètre effectif de généricité est contraint ou non (*'Elt : Type'* ou *'<Elt : subtype(T)'* et *'Elt '* ou *'Elt -> T'*).

On peut remarquer également que le traitement uniforme des objets internes et externes dans la déclaration des paramètres formels de généricité est restritive : l'affectation entre références n'est en effet astreinte qu'à la conformité de types alors que les conditions de validité des constructeurs Guide (cf 8.1) imposent l'égalité telle qu'elle est nécessaire dans les cas d'affectation entre deux objets internes. Ces conditions sont restrictives dans les cas où un paramètre formel de généricité est remplacé par un type référence. Les affectations suivantes ne sont par exemple pas permises dans le corps du constructeur *Box* :

```
component [i] := nil ;
...
anything := component [i] ; -- anything : ref Top ;
```

Si le paramètre effectif de généricité est un type référence : *Box of [ref T]*, de telles affectations sont pourtant valides et le rajout d'une contrainte de généricité stipulant cette condition sur le paramètre formel *Element* permettrait de les autoriser. Cette contrainte pourrait être exprimée par le mot clé *ref* dans l'en-tête du constructeur :

```
Box of [Element : ref Top] is
```

Elle permettrait notamment de faire appel au service de noms pour cataloguer des objets définis par l'intermédiaire d'un type paramètre de généricité.

9. GESTION DES EXCEPTIONS

Une exception désigne une condition anormale dans le déroulement d'un programme. Anormal prend ici le sens d'exceptionnel, c'est-à-dire qui se produit dans certaines situations particulières qui ne constituent pas le cas général. Une erreur dans un programme, une panne, l'abandon d'une transaction, un signal de fin de fichier, la violation d'une contrainte d'intégrité sont des exemples d'exceptions. Des mécanismes propres au traitement des exceptions sont définis dans de nombreux langages ; ils permettent une clarification du code par la séparation entre la description du déroulement normal d'un programme, et le traitement des cas exceptionnels qui peuvent se produire durant son exécution.

Parmi les quatre modèles étudiés dans cette thèse, seul Emerald ne fournit aucun outil spécifique pour le traitement des exceptions. Owl et Guide définissent des mécanismes similaires, comparables notamment à ceux des langages CLU [Liskov 79], et Modula-2+ [Rovner 86]. Une exception levée par l'exécution d'une opération *op* sur un objet Owl ou Guide provoque la terminaison de *op* et le signalement à l'appelant de la condition exceptionnelle détectée. Ce mécanisme est intéressant au regard de la simplification qu'il permet d'obtenir dans l'écriture du code (séparation du traitement des cas normaux de celui des cas d'erreurs) mais impose une séparation entre la détection et la gestion des cas exceptionnels. Les solutions proposées pour le traitement des exceptions dans Eiffel sont différentes car B. Meyer estime qu'une telle séparation est contraire aux principes de la programmation modulaire. Il juge incompatible à l'idée de correction et de robustesse, le fait qu'une procédure ayant détecté une erreur, puisse rendre le contrôle à la procédure qui l'a appelé sans mettre en œuvre elle-même aucune action appropriée.

9.1 OWL ET GUIDE

Comme ces deux modèles fournissent le même mécanisme d'exceptions, je me contente ici de décrire celui de Guide.

Une exception correspond à la terminaison anormale d'une opération. Un bloc particulier, appelé bloc de traitement d'exceptions, peut décrire le traitement prévu par le programmeur en cas de terminaison anormale d'une opération appelée dans un bloc d'instructions. Si un tel bloc de traitement existe, il est exécuté dès le signalement d'une exception par une instruction du bloc auquel il est associé, puis le contrôle est passé à la

première instruction qui suit ce bloc. La séquence de traitement d'une exception peut donc être considérée comme un substitut au bloc d'instructions inachevé.

Définition d'une exception

Une exception est définie par un identificateur et une éventuelle valeur ; cette valeur est de type *String* dans Guide et peut être de n'importe quel type dans Owl.

Les exceptions que peut signaler une opération (méthode ou procédure) doivent être déclarée dans sa signature ; elles sont introduites par le mot clé *signal* :

```
method push (in newElement : aType)
signal FullStack
```

Les exceptions prédéfinies correspondent aux conditions anormales détectées par le système ou le matériel sous-jacent. Elles sont implicitement déclarées dans la signature de toute opération. Quelques exemples d'exceptions prédéfinies sont :

- *Program_error (String)*, erreur détectée par le matériel : le paramètre précise la nature de l'erreur,
- *Failure (String)*, panne d'un site : le nom du site est donné en paramètre,
- *Activity_aborted*, abandon d'une activité,
- *Unexpected (String)*, exception non attendue : le nom de l'exception est passée en paramètre (la seule exception prédéfinie de Owl est comparable à celle-ci).

Signalement d'une exception

Les exceptions déclarées par le programmeur sont déclenchées explicitement par une instruction *raise* :

```
raise FullStack ;
...
raise OpenFileError ("f") ;
```

L'exécution d'une instruction *raise* provoque l'interruption de l'opération en cours et le signalement de l'exception à l'opération appelante. Si l'exception signalée doit retourner une chaîne de caractères, sa valeur doit être spécifiée entre parenthèses. Une instruction *raise* n'est valide que si l'exception signalée est déclarée dans la signature de l'opération qui la contient.

Dans un bloc de traitement, la propagation de l'exception en cours de traitement peut être spécifiée explicitement par une instruction *raise* sans paramètre.

Traitement d'une exception

Un bloc de traitement peut être attaché à une instruction d'appel d'une opération sur un objet ; il spécifie les exceptions attendues et les traitements associés :

```
x.op(...) ;
except
   Exception-1 (msg : String) : Bloc-a ;      // (1)
   Exception-2 : Bloc-b ;
   others :  raise Exception-k ("...") ;
end ;
...
x.op(...) ;
except
   Exception-1 : Bloc-c ;
   Exception-k : raise ;
end ;
```

Si une exception retourne une chaîne de caractères, celle-ci doit être déclarée dans le bloc de traitement pour pouvoir y être désignée (cf (1) dans l'exemple).

La clause *others* permet d'associer un traitement à toutes les exceptions dont le nom n'est pas mentionné explicitement dans le bloc.

Une exception non attendue provoque l'interruption de l'opération à laquelle elle a été signalée et le signalement d'une exception à l'opération appelante : si l'exception non attendue est déclarée dans la signature de l'opération à laquelle elle est signalée, elle est propagée à l'opération appelante ; dans le cas contraire c'est l'exception *Unexpected* qui est propagée. La même règle de propagation s'applique lorsque l'instruction *raise* est utilisée sans paramètre dans un bloc de traitement.

L'imbrication de blocs de traitement n'est pas autorisée. Si une exception se produit pendant l'exécution d'un tel bloc, l'opération en cours est interrompue et l'exception est signalée à l'opération appelante. Ce cas se produit notamment lors de l'exécution d'une instruction *raise* dans un bloc de traitement.

9.2 EIFFEL

Le mécanisme de signalement et de traitement des exceptions dans Eiffel a deux objectifs :

• il offre au programmeur la possibilité de prendre en compte des erreurs de programmation et d'inclure en conséquence dans le code des opérations définies sur ses classes des instructions constituant une sortie propre ou une tentative de correction,

• il permet la prise en compte d'incidents dûs à quelque condition anormale détectée par la machine ou le système.

Définition et signalement d'une exception

Une exception peut être déclenchée pendant l'exécution d'une opération *op* si une des situations suivantes se produit :

• la précondition de *op* ou un invariant de classe n'est pas vérifié à l'appel de *op* ,

• la postcondition de *op* ou un invariant de classe n'est pas vérifié lorsque *op* termine,

• une autre assertion est violée pendant l'exécution de *op*,

• un appel d'opération est tenté sur une variable non initialisée,

• une opération exécutée par *op* retourne une condition anormale détectée par la machine ou le système,

• une opération *op'* appelée par *op* se termine mal (la définition d'une mauvaise terminaison est donnée plus loin).

La violation d'une assertion attachée à un objet ne provoque une exception que si un contrôle dynamique a été requis par le programmeur lors de la compilation de la classe de l'objet (pour plus d'efficacité à l'exécution, le programmeur peut en effet spécifier que le contrôle des assertions ne doit pas être mis en œuvre ou ne doit être mis en œuvre que partiellement : test des pré-conditions, test des pré- et des post-conditions, etc).

Traitement d'une exception

Le traitement d'une exception déclenchée pendant l'exécution d'une opération *op*, doit être prévu dans l'opération *op* elle-même. Pour cela, le corps d'une opération peut contenir un bloc de secours contenant les instructions dédiées au traitement des exceptions que son exécution peut provoquer ; un bloc de secours est introduit par le mot clé *rescue* :

```
anOperation is
  require    -- précondition
    ...
  do         -- code de l'opération
    ...
  ensure       -- postcondition
    ...
  rescue       -- bloc de secours
    ...
end ;
```

Le bloc de secours est destiné à rendre le contrôle à l'opération appelante en lui signalant l'exception après nettoyage de l'environnement, ou à tenter un nouvel essai

d'exécution de l'opération interrompue après modification des conditions d'exécution. Une nouvelle exécution d'une opération interrompue est provoquée par une instruction *retry* :

```
one-editor-command is
        -- saisie d'une commande utilisateur,
        -- et, si possible,
        -- exécution de l'opération correspondante
  do
    ...
  rescue
    message ("Votre commande n'a pu être exécutée") ;
    message ("Veuiller en essayer une autre") ;
    ...    -- remise de l'objet éditeur
    ...    -- dans un état cohérent
    retry
end ;
```

La terminaison d'une opération par l'exécution de son bloc de secours est considéré comme une défaillance de l'opération, qualifiée en conséquence d'*opération mal terminée*. Une telle terminaison provoque le signalement d'une exception à l'opération appelante. Le choix d'une propagation systématique s'explique par le fait qu'un bloc de secours n'est pas une alternative au code décrit dans le corps de l'opération. Il est défini pour permettre une remise de l'objet concerné dans un état cohérent, c'est à dire qui satisfasse notamment les invariants de classe.

En pratique, tout contrôle d'exception est annulé pendant l'exécution d'un bloc de secours, il est donc crucial qu'un tel bloc soit parfaitement correct. Dans cette optique, il est important qu'il soit restreint au minimum d'instructions nécessaire à la remise dans en état correct de l'objet et éventuellement à une nouvelle exécution de l'opération.

9.3 SYNTHESE

La comparation des deux mécanismes présentés ci-dessus conduit à s'interroger sur l'exploitation qui peut en être faite : le mécanisme fourni par Owl et Guide constitue un véritable outil de programmation alors que celui mis en œuvre dans Eiffel est un outil d'aide à la production de code correct. Dans ce modèle, une exception représente réellement une condition anormale et provoque l'arrêt du programme, à moins qu'une tentative de ré-exécution de l'opération qui a échoué réussisse : il n'est pas possible pour le programmeur de considérer une exception comme un paramètre d'une opération comme cela est fait dans Owl et dans Guide.

Utiliser les exceptions comme outil de programmation est cependant intéressant et cet aspect du mécanisme de Guide doit être conservé. Il permet en effet de séparer dans la

description du code les situations courantes de celles qui le sont moins, et permet de mettre ainsi les cas particuliers en évidence. Deux points importants sont pourtant à revoir :

• Le premier concerne l'expression des blocs de traitement. Tel qu'il est défini actuellement, le langage est peu satisfaisant sur ce point. Il faudrait notamment prévoir des possibilités de factorisation des blocs car leur inévitable profusion dans un contexte où les exceptions sont utilisées comme outil de programmation est préjudiciable à la lisibilité du code.

• Le second point est relatif à la propriété de persistance qui caractérise les objets de Guide. Tel qu'il est défini actuellement, le modèle ne permet pas le traitement d'une exception par l'objet dans lequel elle a été déclenchée. Ce mécanisme peut conduire à des situations incohérentes, notamment dans les cas où une activité est interrompue brutalement sur un ordre de l'utilisateur (CTRL-C) ou sur un ordre du système (par exemple, lors de la terminaison d'un bloc parallèle). Dans ces cas là, l'objet dans lequel s'exécutait l'activité interrompue risque de se trouver dans un état incohérent, par exemple par rapport à ses conditions de synchronisation. Il est donc nécessaire de revoir la règle de recherche des blocs de traitement d'exceptions en permettant au programmeur d'associer au code d'une opération un bloc de secours du type du celui défini dans Eiffel (la règle de recherche du bloc de traitement serait alors comparable à celle mise en œuvre dans Ada [Ichbiah 83]).

10. RÉPARTITION

Les modèles Guide et Emerald sont mis en œuvre sur des systèmes distribués. Contrairement à ce qui se passe classiquement dans la conception d'un système réparti, les fonctions liées à la répartition ont été prise en compte dès l'origine dans la conception de ces deux systèmes. Ils ont pu ainsi être conçus comme des systèmes à haut degré d'intégration, dans lesquels chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti, plutôt que comme un système hôte connecté à d'autres systèmes hôtes par des voies de communication. L'intérêt majeur d'une telle intégration est la virtualisation possible des ressources qui consiste à décharger le programmeur de la nécessité de connaître la localisation de celles qu'il utilise ou le détail du protocole de communication permettant d'y accéder.

Des outils permettant la gestion explicite de la répartition sont cependant nécessaires, par exemple pour la mise en place de mécanismes de répartition de charges. Offerts naturellement aux programmeurs qui s'occupent de la gestion du système, ces

outils sont également indispensables aux programmeurs d'applications. Ils leur permettent en effet de prendre en compte la localisation des ressources propres à un site, telle une imprimante par exemple, et leur donnent ainsi une possibilité d'optimizer les traitements qui les utilisent.

10.1 EMERALD

Emerald fournit 4 primitives permettant la localisation explicite d'un objet sur un nœud du système :

- *Locate*, permet de déterminer sur quel nœud réside un objet,
- *Fix*, permet de fixer un objet sur un nœud,
- *Unfix*, permet de rendre un objet mobile après un Fix,
- *Move*, permet de déplacer un objet.

Un nœud est une entité logique de localisation. Il représente un espace d'adressage. Il est possible que plusieurs nœuds existent sur une même machine.

La mobilité des objets rend possible la diminution des inconvénients liès à un appel d'objet à distance, par le déplacement de l'objet appelé sur le nœud de l'objet appelant. Cependant, l'opportunité d'un tel déplacement dépend fortement de la taille de l'objet à déplacer, du nombre d'opérations en cours d'exécution sur cet objet et de la nature des appels, locaux ou distants, prévisibles sur lui après déplacement : elle est donc très difficile à évaluer. La connaissance d'une application suffit cependant à décider que dans certain cas, le déplacement d'un objet est opportun. Emerald fournit pour cela la notion de 'passage en paramètre par déplacement' ('call-by-move'). L'objet désigné par un paramètre ainsi défini est déplacé sur le nœud de l'objet appelant, permettant à ce dernier de l'utiliser de façon efficace. Le programmeur peut aussi spécifier qu'un objet déplacé doit regagner son nœud d'origine à la terminaison de l'opération qui a provoqué son déplacement.

10.2 GUIDE

Il est prévu de fournir dans Guide des primitives comparables aux primitives *Fix*, *Unfix*, *Locate* et *Move* définies dans Emerald. Le choix du déplacement ou non d'un objet lors d'un appel distant doit être fait par un mécanisme indépendant du programmeur.

Une première étude sur le sujet est présentée dans [Lunati 88]. Basée sur un important travail de simulation, elle donne les principes d'un mécanisme de partage de

charge conduisant à une bonne régulation (au moins théorique) des travaux sur l'ensemble du système Guide.

On peut noter que dans le système tel qu'il est réalisé actuellement, il est possible d'imposer le site de résidence d'un objet. Le site choisi est transmis en paramètre à l'opération new définie sur sa classe : *<newObject := aClass.new (aSite)>*. Cette possibilité permet notamment de forcer l'exécution des applications en environnement réparti.

10.3 SYNTHESE

La répartition du système d'exécution de Guide n'est pas étrangère au choix d'un modèle à objets pour le modèle de programmation. Un objet peut être considéré comme une ressource et constituer de ce fait une unité naturelle de distribution.

L'intégration des aspects liés à la répartition permet dans les programmes une désignation des ressources, donc des objets, indépendante de leur localisation. On peut remarquer que bien que la répartition n'intervienne pas au niveau du modèle de programmation du fait de son intégration dans le système d'exécution, elle impose un certain nombre de contraintes dans sa mise en œuvre. Elle justifie ainsi la mise en œuvre du mécanisme de chargement dynamique des classes à l'exécution (cf 7.5).

La transparence de la localisation des objets au niveau des applications présente également un avantage important lors du travail de mise au point. Une application peut en effet être testée en milieu centralisé avant de l'être en milieu réparti ce qui permet de s'affranchir dans un premier temps des contraintes inhérentes à la répartition : performances, problème de communication, etc.

11. PARALLÉLISME ET PARTAGE D'OBJETS

La possibilité de définir explicitement des traitements parallèles permet de mieux exploiter la grande puissance des postes de travail modernes, éventuellement multi-processeurs. Sur un système réparti, elle permet également l'exploitation du parallélisme inhérent à la répartition.

La définition d'un support au parallélisme nécessite la mise en œuvre de mécanismes de gestion de la concurrence permettant de garantir l'intégrité des données partagées par deux activités simultanées. Dans un modèle à objets, l'unité naturelle de

partage est l'objet ; le contrôle de la concurrence s'exprime donc en général, sous la forme de règles d'accès qui assurent la cohérence des accès simultanés à un objet partagé.

La suite de cette section présente les éléments relatifs à la gestion du parallélisme et de la concurrence dans Emerald et dans Guide. Une des prochaines versions d'Eiffel doit fournir une implémentation des processus et de ce fait un support explicite au parallélisme et à la programmation concurrente. Les outils correspondants font encore l'objet d'un travail de conception et ne sont pas présentés dans les documents actuellement disponibles.

11.1 EMERALD

Le modèle à objets actifs d'Emerald conduit naturellement à la conception d'applications parallèles. Objets actifs et contrôle explicite de la localisation permettent aux programmeurs de jouer sur le degré de parallélisme de leurs applications mais aucune structure n'est fournie pour l'expression explicite d'une exécution parallèle (comme par exemple la structure de contrôle *co_begin/co_end* de Guide, cf ci-dessous).

Le contrôle des accès concurrents à un objet partagé est réalisé par l'encapsulation des données et opérations à protéger dans un moniteur.

Objets partagés et moniteurs

Rappelons que chaque objet Emerald peut posséder un processus qui lui est propre ; ce dernier peut appeler et exécuter des opérations sur d'autres objets. La concurrence à l'intérieur d'un objet résulte de l'exécution parallèle du processus de l'objet (s'il existe) et des opérations exécutées sur lui par les processus d'autres objets.

La protection d'un objet partagé est définie dans le type de l'objet par le biais d'un moniteur dans lequel sont encapsulées les données et les opérations qui doivent faire l'objet d'un contrôle. Le type oneEntryDirectory décrit dans la section 3.2 pourrait ainsi être redéfini de la façon suivante :

```
oneEntryDirectory
  export Store, Lookup
  monitor

    var name : String ;
    var anObject : Anything ;

    operation Store [n : String, o : Anything]
      ...
    function Lookup [n : String] -> [o : Anything]
      ...
```

```
      initially
        ...
    end monitor
end oneEntryDirectory
```

Le processus d'un objet s'exécute à l'extérieur du moniteur ; il peut cependant appeler les opérations gérées par ce dernier pour accéder aux données partagées. La synchronisation de plusieurs processus à l'intérieur d'un moniteur est possible par le biais de variables conditions comparables à celle définies dans [Hoare 74].

11.2 GUIDE

Dans Guide, les traitements parallèles requis par une application sont explicitement définis comme tels, par le biais d'une structure de contrôle particulière appelée bloc parallèle. La protection des objets partagés est assurée par l'association d'une condition d'exécution à chaque opération définie sur un objet ; l'exécution d'une opération sur un objet n'est possible que si la condition correspondante est vérifiée.

Remarque. Guide est un système multi-utilisateurs dans lequel tout objet est potentiellement partageable. On peut noter que cette propriété implique naturellement la définition d'un mécanisme de gestion de la concurrence, indépendamment de la possibilité de définir des traitements explicitement parallèles.

Bloc parallèle

Un bloc parallèle décrit un ensemble d'instructions dont le programmeur demande explicitement l'exécution simultanée :

```
co_begin
   a1 : x.op (...) ;
   a2 : y.op (...) ;
   a3 : z.op (...) ;
co_end ;
```

Les structures de contrôle ne sont pas permises dans un bloc parallèle ; chaque instruction du bloc doit correspondre à l'appel d'une opération sur un objet.

Le schéma d'exécution d'un bloc parallèle est synchrone : l'activité[1] en cours est suspendue et pour l'exécution de chaque branche du bloc, une activité fille est créée. On notera que ce schéma ne permet pas l'exécution parallèle de l'activité mère et des activités

[1] Une activité Guide correspond à un processus Unix. Les activités ont été rapidement présentées au chapitre I ; voir le chapitre V pour une présentation plus détaillée.

filles. Cette restriction est imposée par le contrôle de la concurrence lorsque l'activité mère et les activités filles s'exécutent dans le contexte d'une transaction [Guide R3].

Une condition d'arrêt peut être spécifiée à la suite du mot clé *co_end*. Elle peut indiquer par exemple que l'exécution du bloc doit être interrompue dès que l'une des branches se termine : <*co_end first*> ou qu'elle ne doit pas être interrompue avant la fin de toutes les branches : <*co_end all*>.

```
Exemple

co_begin
   a1 : x.op (...) ;
   a2 : y.op (...) ;
   a3 : z.op (...) ;
co_end first ;
```

Une condition de terminaison peut être exprimée par une expression booléenne dans laquelle l'identificateur de chacune des branches du bloc peut être utilisé pour faire référence à la terminaison de la branche correspondante :

```
co_end (a1 or a2) ;
co_end (a1 and a3) ;
```

Les conditions prédéfinies *first* et *all* correspondent respectivement aux expressions booléennes <*a1 or ... or aN*> et <*a1 and ... and aN*>.

Objets partagés et conditions de synchronisation

Le contrôle de l'accès à un objet partagé est assuré par le biais d'une condition d'exécution associée à chacune de ses opérations. Une condition d'exécution est une expression booléenne dont la valeur doit être vraie pour qu'une activité qui requiert l'exécution de l'opération correspondante puisse en commencer l'exécution.

Conceptuellement, le programme d'une opération est équivalent à la séquence décrite ci-dessous :

```
while not (condition_d'exécution) do wait ;
<code de l'opération>
```

Les conditions d'exécution sont spécifiées dans les classes ; le même schéma de synchronisation est donc commun à tous les exemplaires d'une même classe. La condition d'exécution propre à une opération doit être spécifiée par une clause *control* dans sa déclaration :

```
method example (...)
control <condition d'exécution>
begin
```

```
   ...
end example ;
```

L'ensemble des conditions d'exécution spécifiées dans le corps d'une classe peuvent aussi être regroupées dans une clause *control* commune ; la condition propre à une opération est alors identifiée par le nom de cette dernière :

```
class C ... is
   ...
   ...
control
   op-1 : ... ;
   op-2 : ... ;
   ...
end C.
```

Une expression booléenne qui correspond, dans le corps d'une classe *C*, à la condition d'exécution associée à une opération *op* peut faire référence aux éléments suivants :

- les variables qui constituent l'état d'un objet de la classe *C*,
- les arguments de l'opération *op*,
- les compteurs de synchronisation.

Un compteur de synchronisation est un composant de l'état d'un objet déclaré implicitement par le compilateur et mis à jour par le système. Cinq compteurs sont associés à chaque opération *op* définie sur un objet *O* :

- ***invoked(op)*** qui comptabilise le nombre total d'activités ayant demandé l'exécution de la méthode *op* sur l'objet *O*,
- ***started(op)*** qui comptabilise le nombre total d'activités ayant été autorisées à exécuter l'opération *op* sur l'objet *O*,
- ***completed(op)*** qui comptabilise le nombre total d'activités ayant effectué une exécution complète de l'opération *op* sur l'objet *O*,
- ***current(op)*** qui comptabilise à un instant donné le nombre d'activités qui exécutent l'opération *op* sur l'objet *O*,
- ***pending(op)*** qui comptabilise à un instant donné le nombre d'activités qui attendent l'autorisation d'exécuter l'opération *op* sur l'objet *O*.

Les compteurs sont utilisés pour l'expression des conditions d'exécution mais leurs valeurs peuvent également être référencées dans le corps des opérations. Ce sont des variables prédéfinies accessibles au programmeur en lecture uniquement.

L'exemple suivant illustre l'utilisation des conditions d'exécution pour la définition des objets de classe *oneEntryDirectory*. Le schéma de synchronisation est le même que

celui spécifié à l'aide des moniteurs d'Emerald dans l'exemple de la section précédente : les opérations *Store* et *Lookup* sont exclusives avec elles-mêmes et mutuellement.

```
class oneEntryDirectory is

  name : String = "\0" ;
  anObject : Anything = nil ;

  method Store (in n : String, o : Anything) ;
  begin
    name := n ;
    anObject := o ;
  end Store ;

  method Lookup (in n : String) : Anything ;
  begin
    if n = name
      then o := anObject
      else o := nil
    end
  end Lookup ;

  control
    Store  : (current(Store)=0) and (current(Lookup)=0) ;
    Lookup : (current(Store)=0) and (current(Lookup)=0) ;

end oneEntryDirectory.
```

Ce schéma de synchronisation peut être assoupli sans risque d'incohérence en autorisant des exécutions simultanées de l'opération *Lookup*. Cette mise à jour est immédiate dans Guide, où il suffit de redéfinir la condition d'exécution de l'opération *Lookup* sous la forme *<Store : current(Store) = 0>*.

Si la condition d'exécution d'une opération donnée n'est pas spécifiée, elle a par défaut la valeur vraie. Le mot clé *exclusive* permet de spécifier qu'une opération donnée est exclusive. Ainsi, la clause de contrôle de l'exemple précédent peut être écrite plus simplement :

```
control
  Store  : exclusive ;
  Lookup : exclusive ;
```

Remarque. Les opérations *get* et *put* définies implicitement pour chaque variable d'état visible (cf 3.4) d'un objet sont exclusives par définition lorsque ce dernier est synchronisé.

Conceptuellement, le système maintient à jour pour chaque opération les cinq compteurs définis précédemment. En pratique, seuls trois compteurs sont nécessaires puisque les égalités suivantes sont vérifiées :

```
current (op)  = started (op) - completed (op)
```

```
pending (op)  = invoked (op) - started (op)
```

L'utilisation des conditions d'exécution pour la définition de différents schémas de synchronisation classiques est illustrée dans l'annexe II.

Conditions de synchronisation et héritage

Lors de la construction d'une classe par héritage, les conditions d'exécution associées à chaque opération sont implicitement héritées.

Une condition d'exécution peut cependant être redéfinie et dans ce cas sa nouvelle valeur devra être spécifiée dans le corps de la sous-classe. Une telle redéfinition peut être faite indépendamment de la redéfinition de l'opération elle-même.

Ce mécanisme permet d'associer facilement des schémas de synchronisation différents à des objets de même type comme l'illustre l'exemple suivant qui montre comment définir différents schémas de synchronisation sur des objets fichiers.

```
type File is
  method read ...
  method write ...
end File.

// classe File : pas de condition d'exécution
// sur les méthodes read et write

class File implements File is
  ...
end File.

// classe RW_File : schéma de synchronisation
// n lecteurs ou 1 rédacteur sans priorité

class RW_File subclass of File implements File is
  control
     read  : current (write) = 0 ;
     write : exclusive ;
end RW_File.

// classe RpW_File : schéma de synchronisation
// n lecteurs ou 1 rédacteur avec priorité aux lecteurs

class RpW_File subclass of File ... is
  control
     read  : current (write) = 0 ;
     write : exclusive and (pending (read) = 0) ;
end RpW_File.

// classe RWp_File : schéma de synchronisation
// n lecteurs ou 1 rédacteur avec priorité aux rédacteurs

class RWp_File subclass of File ... is
  control
     read  : (current (write) + (pending (write)) = 0 ;
```

```
        write : exclusive ;
    end RWp_File.
```

11.3 SYNTHESE

L'expression des conditions de synchronisation sous la forme de conditions d'exécution associées aux opérations est une solution séduisante. Elle permet d'extraire l'expression du contrôle, du code sur lequel il porte et conduit ainsi à la mise en évidence des contraintes et à une meilleure lisibilité du code. Elle s'intègre bien dans le modèle de programmation et évite l'introduction d'un concept supplémentaire comme le moniteur.

Un autre avantage intéressant de cette solution est le fait que la mise en œuvre des conditions d'exécution ne présente pas de difficulté particulière. Une condition d'exécution est compilée sous la forme d'une fonction booléenne associée à l'opération sur laquelle elle porte. Lors de l'appel d'une opération sur un objet, le système évalue la fonction booléenne et bloque l'activité appelante si besoin. Une liste d'activités en attente est associée à chaque objet synchronisé. Lorsqu'une activité termine l'exécution d'une opération sur l'objet elle réveille l'ensemble des activités désignées par cette liste. Pour chacune, le système réévalue la condition d'exécution (des exemples de classes synchronisées ainsi que la description du processus de mise en œuvre des conditions d'exécution sont donnés dans l'annexe II).

Un gros travail reste cependant à faire pour fournir un langage simple et clair pour l'expression des conditions de synchronisation. Les compteurs constituent certes un outil puissant, mais leur maîtrise est malheureusement peu évidente quand il s'agit de décrire des schémas de synchronisation un peu compliqués, comme le schéma des lecteurs/rédacteurs avec priorité par ordre d'arrivée, par exemple (cf classe *RWro_File* décrite en annexe).

La programmation des exemples donnés en annexe a mis en évidence la nécessité de fournir la notion de **méthode cachée**. En effet, pour la définition de certains schémas de synchronisation il est nécessaire de définir des méthodes dont le seul rôle est de permettre l'expression du contrôle (cf les méthodes *GetOrder*, *AskRead* et *AskWrite* de la classe *RWro_File*). De telles méthodes n'ont pas à apparaître dans l'interface des objets sur lesquels elles sont définies et en conséquence, il serait plus cohérent de permettre au programmeur de les déclarer ainsi. (Il est important de remarquer qu'il n'est pas envisageable de se passer de la notion de méthode cachée en permettant l'association de conditions d'exécution aux procédures. Dans Guide, une procédure a une sémantique tout à fait différente d'une méthode. Les procédures permettent une factorisation de code dans

l'écriture du corps des méthodes définies par une classe, elles ne sont jamais héritées ; l'appel à une procédure est réalisé par un branchement statique et lui associer une condition d'exécution remettrait en question ce mécanisme d'appel et le gain en temps d'exécution qu'il représente pour le programmeur).

12. CONCLUSION

La présentation du modèle d'objets de Guide faite dans les précédentes sections sur la base d'une comparaison avec les modèles d'objets d'Emerald, de Trellis/Owl et d'Eiffel a permis d'une part de mettre en valeur les aspects intéressants du modèle Guide, et d'autre part, de dégager un ensemble de points pour lesquels un travail d'approfondissement est encore nécessaire.

Parmi les aspects caractéristiques du modèle d'objets de Guide, on peut citer :

- La séparation entre type et classe qui permet d'associer différentes réalisations à une même interface, et de fournir la notion de vue.
- La définition et la réalisation des classes comme des objets ce qui permet de définir un modèle uniforme et d'introduire l'opération de création d'objets d'une manière cohérente.
- L'organisation hiérarchique de l'ensemble des types qui permet une définition propre des propriétés communes à tous les objets.
- L'expression des contraintes d'accès aux objets partagés sous la forme de conditions d'exécution qui s'intègrent parfaitement dans les schémas de définition et d'exécution des objets.

La mise en œuvre du modèle sur un système à objets réparti constitue cependant le point le plus important et sans doute le plus original de Guide. Le support implicite de conservation permanente des objets et l'invisibilité de leur répartition sur les différents sites permettent aux utilisateurs de concevoir et de programmer leurs applications sans avoir à tenir compte de la sauvegarde de l'état des objets qui les constituent ou de leur localisation dans le système. L'intégration de la répartition présente notamment l'avantage de permettre la programmation d'applications réparties de la même manière que s'il s'agissait d'applications centralisées, et leur mise au point en environnement mono-site.

Il faut remarquer aussi les contreparties des propriétés de persistance et de répartition du système à objets support :

• Le travail dans un environnement d'objets persistants nécessite la mise en œuvre d'un mécanisme de gestion de versions qui permette au programmeur de prendre en compte l'évolution de ses applications ; il rend nécessaire également la mise en œuvre d'un mécanisme de ramasse-miettes.

• L'exécution en environnement multi-site nécessite un chargement dynamique du code des classes et en conséquence une perte de performances qui justifie l'introduction de la notion d'objet interne et brise l'uniformité du modèle.

Parmi l'ensemble des points qui nécessitent un travail d'approfondissement, les suivants me paraîssent être les plus importants :

• L'opération *destroy* qui permet la destruction explicite d'un objet doit être définie comme une opération propre aux objets classes.

• Les possibilités d'intégration des notions de classe différée, de déclaration par association, d'héritage caché et enfin d'héritage non conforme doivent être étudiées lors de la définition du mécanisme d'héritage multiple.

• Le mécanisme de gestion d'exceptions doit être redéfini de façon à permettre le traitement d'une exception par l'objet qui l'a déclenchée.

• La notion de méthode cachée doit être introduite de manière à permettre une réalisation cohérente des schémas de synchronisation qui nécessitent la déclaration de méthodes ne devant pas apparaître dans l'interface des objets sur lesquels elles sont définies.

• Des outils plus faciles à maîtriser que les compteurs de synchronisation (*pending*, *started*, *current*, etc) doivent être fournis pour l'expression des conditions d'exécution.

L'évalution du modèle Guide présentée ici est reprise et approfondie au chapitre VI, sur la base notamment des exemples d'application décrits dans le chapitre suivant. Ces exemples, ainsi que ceux donnés dans l'annexe 2, illustrent les remarques faites ci-dessus. Le travail de programmation que leur réalisation a nécessité a permis une première évalution de l'utilisation du langage et du système Guide comme support de développement et vient compléter celle donnée ci-dessus (cf IV-4).

CHAPITRE IV

LE LANGAGE GUIDE
Exemples de programmation

1. PRÉSENTATION

Ce chapitre contient la description en langage Guide de deux applications simples : une application de messagerie, et une application de gestion de bibliothèque. Son objectif est double : il illustre d'une part les méthodes de décomposition d'applications en termes d'objets, et d'autre part la syntaxe ainsi que les possibilités du langage Guide : sous-typage et héritage, persistance, conformité et contrôle dynamique, exceptions, partage d'objets et utilisation des constructeurs. Comme nous l'avons vu en III, la répartition du système peut ne pas être visible au niveau des applications. Les applications décrites ici n'y font pas référence. Il est cependant important de remarquer que l'ensemble des objets décrits peuvent être indifféremment localisés sur un site quelconque du système.

Les deux applications présentées sont conçues selon un même schéma. Elles sont constituées d'un ensemble d'objets permanents dont les types et classes sont décrits dans la suite. Les services qu'elles fournissent aux utilisateurs leur sont accessibles par l'intermédiaire de deux objets temporaires : un **agent utilisateur** et un **agent de transfert**, dont le rôle est de faire le lien entre les utilisateurs et les objets permanents qui constituent les applications.

L'agent utilisateur est responsable de la présentation de l'application sur la machine hôte sur laquelle travaille l'utilisateur : c'est un objet d'interface. L'agent de transfert est chargé de transmettre les requêtes de l'utilisateur à l'application et les résultats dans l'autre sens : il réalise l'ensemble des services fournis par cette dernière. Comme le montre la figure suivante, un agent de chaque type est attribué à chaque utilisateur qui demande à utiliser les services d'une application.

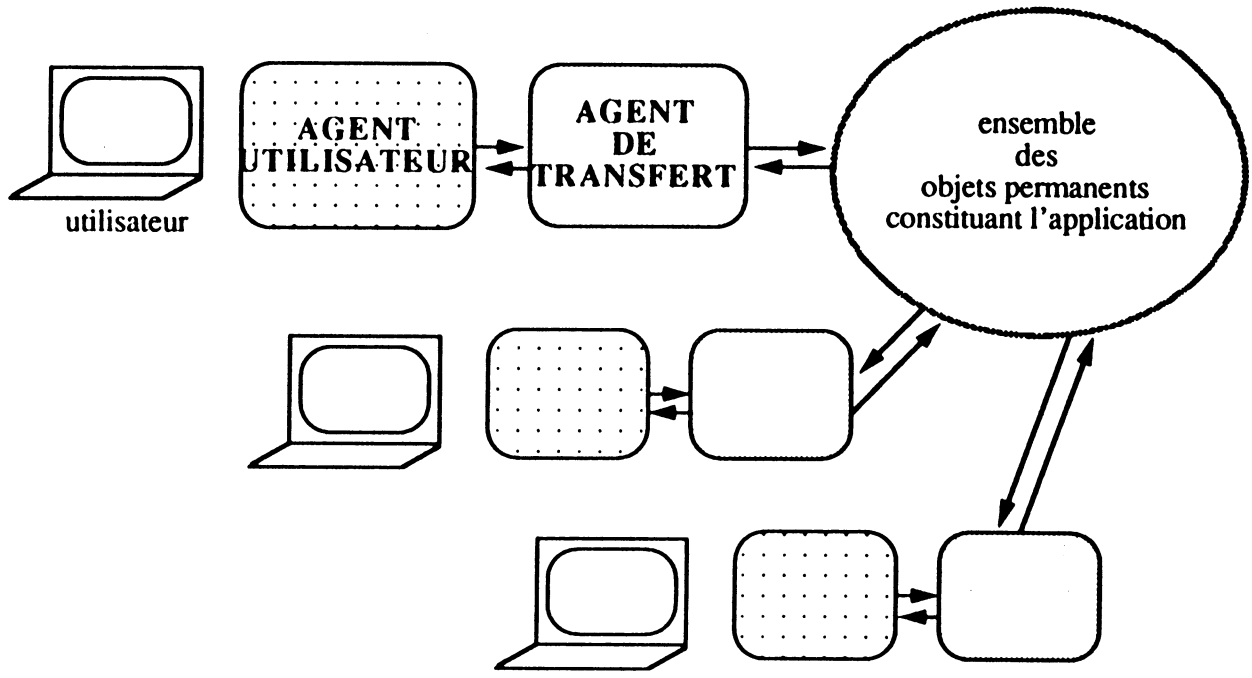


Figure 1. Agents utilisateur et Agents de transfert

L'agent de transfert transmet l'information relative à l'application à l'agent utilisateur par l'intermédiaire d'objets de communication (canaux, fichiers, etc ; le type *File* utilisé dans le deuxième exemple est décrit en annexe I). Sa réalisation est indépendante de la nature des machines hôtes depuis lesquelles les utilisateurs accèdent à Guide, et de laquelle dépend la réalisation des agents utilisateurs. Ces derniers ne sont pas décrits ici. On peut cependant remarquer que dans l'état actuel du prototype, ils peuvent fournir aussi bien une interface textuelle que graphique, la première étant réalisée par l'intermédiaire des canaux d'entrées-sorties prédéfinis *input* et *output* décrits en 3.1 au chapitre V, et la seconde par l'intermédiaire des fonctions du gestionnaire de fenêtres X-Window dont l'intégration dans Guide est décrite dans l'annexe III.

2. MESSAGERIE

La messagerie électronique est traditionnellement considérée comme l'application répartie typique. A ce titre, elle est souvent prise comme exemple d'illustration et de démonstration par les concepteurs de systèmes répartis [Liskov 83], [Almes 84]. Il était donc intéressant de la mettre en œuvre sur le système Guide.

Sous sa forme la plus simple, une messagerie électronique permet à un ensemble de clients, enregistrés au préalable par l'administrateur du service, d'effectuer les opérations suivantes :

• consulter la liste des clients enregistrés (opération *ListSubscriberName*),

• envoyer un message à un autre client (opération *Send*),

• consulter la liste des messages présents dans sa boîte aux lettres (opération *ListMsgHeaders*),

• lire un de ses messages (opération *GetMsg*),

• détruire un de ses messages (opération *DeleteMsg*).

Ces opérations sont fournies par l'objet 'agent de transfert' de l'application.

Les objets permanents qui constituent l'application sont représentés sur la figure 2. Leurs types sont les suivants :

• Messages : type *MailMsg*. Les objets de ce type sont les messages échangés entre les clients.

• Boîtes aux lettres : type *MailBox*. Les objets de ce type réalisent les boîtes aux lettres ; ce sont essentiellement des listes de références vers des messages.

• Répertoires : type *MailUserList*. Les objets de ce type réalisent l'association entre une liste de noms de clients et les boîtes aux lettres associées à chacun d'eux.

Le type et la classe de l'objet 'agent de transfert' sont décrits ci-dessous (je me suis limitée ici à la description des services fournis aux clients de la messagerie : les type et classe *MailTransferAgent* devraient donc être complétés par les opérations d'ajout et de suppression de clients dont l'accès est réservé à l'administrateur).

```
type MailTransferAgent is
  // envoi de messages
  method ListSubscriberName
         (out number : Integer ; text : String[80]) ;
  method Send (in receiver : String[10]; msg : ref MailMsg) ;
    signal UnkownReceiver ;
  // lecture et destruction des messages reçus
  method ListMsgHeaders (out number : Integer; text :
String[80]);
  method GetMsg (in number : Integer) : ref MailMsg ;
    signal BadNumber ;
  method DeleteMsg (in number : Integer) ;
    signal BadNumber ;
  // Initialisation de l'état de l'agent de transfert
  method Init
         (in name : String[10]; net : ref MailUserList) ;
```

```
    signal UserMustSubscribe ;
end MailTransferAgent.

class MailTransferAgent
implements MailTransferAgent is

usernet   : ref MailUserList ;
username  : String[80] ;
usermbox  : ref MailBox ;

method Send (in receiver : String[10]; msg : ref MailMsg)
  signal UnkownReceiver ;
  mbox : ref Mailbox;
  begin
  mbox := usernet.FetchSubscriber (receiver);
  except
    BadName :
      raise UnkownReceiver ;
  end ;
  mbox.Post (msg);
  end Send;

method GetMsg (in number : Integer) : ref MailMsg ;
  signal BadNumber ;
  begin
  return usermbox.Read (number) ;    // (*)
  end Read;

method ListMsgHeaders(out number : Integer ; text : String[80]);
  begin
  usermbox.ListHeaders (number,text);
  end ListMsgHeaders ;

method DeleteMsg (in number : Integer) ;
  signal BadNumber ;
  begin
  usermbox.Delete (number);    // (*)
  end DeleteMsg ;

method ListSubscriberName
    (out number : Integer ; text : String[80]) ;
  begin
  usernet.ListSubscriber (number,text) ;
  end ListSubscriberName ;

method Init
    (in name : String[10]; net : ref MailUserList)
  signal UserMustSubscribe ;
  begin
  usermbox  := net.FetchSubscriber (name) ;
  except
    BadName :
      raise UserMustSubscribe ;
  end ;
  username  := name ;
  usernet   := net ;
  end Init;

end MailTransferAgent.
```

On peut noter dans le code de la classe ci-dessus l'exploitation du mécanisme de propagation des exceptions non attendues dans les instructions marquées (*).

Lorsqu'un client appelle le service de messagerie, un objet exemplaire de la classe *MailTransferAgent* est dynamiquement créé pour répondre à ses requêtes jusqu'à la fin de sa session. Cet objet est créé sur le site d'émission de la requête d'accès au service mais les autres objets de l'application (messages, boîtes aux lettres, etc) peuvent être créés sur d'autres sites. Cette répartition est invisible au client. L'objet agent de transfert qui lui a été alloué devient suceptible d'être détruit dès la fin de la session. Cette destruction peut être provoquée explicitement par une commande *Quit* ou effectuée implicitement par le ramasse-miettes.

Un objet message contient l'identification de son expéditeur, un sujet et un texte. Le type et la classe *MailMsg* sont décrits ci-dessous :

```
type MailMsg is
   sender : String[10] ;
   subject   : String[10] ;
   text    : String[80] ;

   method Init (in String[10]; String[10]; String[80]);
end MailMsg.

--------

class MailMsg
implements MailMsg is

sender    : String[80] ;
subject   : String[80] ;
text      : String[80] ;

method Init (in sd, sb : String[10], tx :String[80]) ;
   begin
   sender := sd;
   subject := sb;
   text := tx;
   end Init ;

end MailMsg.
```

Le type et la classe *MailBox* décrits maintenant spécifient l'interface et la réalisation des objets boîtes aux lettres :

```
type Mailbox is

   method Post (in ref MailMsg);
      // ajout d'un nouveau message

   method Read (in Integer) : ref MailMsg ;
      signal BadNumber ;
      // la méthode retourne une référence sur le message désigné
      // s'il existe - sinon, elle retourne l'exception BadNumber
```

```
    method Delete (in Integer) ;
      signal BadNumber ;
      // la méthode extrait le message désigné de la boîte
      // s'il existe - sinon, elle retourne l'exception BadNumber
    method ListHeaders (out number : Integer; text : String[80]);
      // la méthode retourne le nombre de messages contenu dans
      // la boîte et un texte contenant leurs en-têtes
    method Init ;

end Mailbox.

--------

class Mailbox
implements Mailbox is

synonym MailMsgList = List of ref MailMsg ;

msglist : ref MailMsgList ;

method Read (in nbr : Integer) : ref MailMsg ;
  signal BadNumber ;
  begin
  if (0 < nbr) and (nbr <= msglist.NbItems) then
    return msglist.Go (nbr) ;
  else
    raise BadNumber ;
  end;
  end Read;

method Post (in new-msg : ref MailMsg);
  begin
  msglist.Last ;
  msglist.Append (new-msg) ;
  end Post;

method ListHeaders (out number : Integer; text : String[80]);
  msg : ref MailMsg ;
  begin
  number := msglist.NbItems ;
  text   := "" ;
  if (number # 0) then
    msg  := msglist.First ;
    text := msg.subject ;
    msg  := msglist.Next ;
    while (msg # nil) do
      text := text + "\001" + msg.subject ;
      msg := msglist.Next ;
    end ;
  end ;
  end ListHeaders ;

method Delete (in nbr : Integer) ;
  signal BadNumber ;
  begin
  if (0 < nbr) and (nbr <= msglist.NbItems) then
    msglist.Delete (nbr) ;
  else raise BadNumber ;
  end;
  end Delete;
```

```
method Init;
  begin
  msglist := MailMsgList.new;
  end Init;

end Mailbox.
```

Enfin, les type et classe des objets répertoires sont décrits ci-dessous :

```
type MailUserList is

  method CreateSubscriber(in name : String[80]) ;
    signal NameAlreadyUsed ;
  // création d'un nouvel abonné dont le nom est donné en argument
  // l'exception NameAlreadyUsed est déclenchée
  // si l'argument correspond au nom d'un abonné existant

  method FetchSubscriber (in name : String[80]) : ref Mailbox ;
    signal BadName ;
  // recherche d'un abonné
  // s'il existe, la méthode retourne une référence sur sa boîte
  // aux lettres - sinon, l'exception BadName est déclenchée

  method DeleteSubscriber (in name : String[80]) ;
    signal BadName ;
  // suppression d'un abonné
  // si le nom donné en argument ne correspond à aucun abonné
  // existant l'exception BadName est déclenchée

  method ListSubscriber (out number :Integer; text :String[80]);
  // cette méthode permet l'affichage de la liste des noms de
  // tous les abonnés par l'objet chargé de l'interface
  // elle produit le nombre d'abonnés et une chaîne de caractères
  // contenant la liste de leurs noms, chaque nom étant précédé
  // par le caractère spécial "\011"

  method Init ;

end MailUserList.

-------

class MailUserList
implements MailUserList is

synomym Entry = Record
                  name : String[1Ø] ;
                  mbox : ref MailBox ;
                end ;

synonym EntryList = List of Entry ;

userlist : ref EntryList ;

method CreateSubscriber(in name : String[80]) ;
  signal NameAlreadyUsed ;
  user : Entry;
  begin
  if (FetchUser (name, user.mbox) # 0) then
    raise NameAlreadyUsed
```

```
    else
      user.name := name ;
      user.mbox := Mailbox.new ;
      userlist.Append (user);
    end ;
    end CreateSubscriber ;

  procedure FetchUser
      (in name : String[80], out mbox : ref MailBox) : Integer ;
    // procédure de recherche d'un abonné
    // s'il existe, la procédure retourne son numéro d'ordre
    // la liste - sinon, 0
    // dans le premier cas, elle rend également une référence
    // sur sa boîte aux lettres
    user : Entry;
    nbr    : Integer = 1 ;
    begin
    userlist.First;
    user := userlist.Next;
    while (user # nil) do
      if (user.name = name) then
        mbox := user.mbox ;
        return nbr
      else
        user := userlist.Next ;
        nbr  := nbr +1 ;
      end ;
    end;
    return 0
    end FetchUser ;

  method FetchSubscriber (in name : String[80]) : ref Mailbox
    signal BadName ;
    mbox : ref MailBox ;
    begin
    if (FetchUser (name, mbox) # 0) then
      return mbox
    else
      raise BadName
    end;
    end FetchSubscriber ;

  method DeleteSubscriber (in name : String[80]) ;
    signal BadName ;
    nbr    : Integer;
    mbox : ref Mailbox ;
    begin
    nbr := FetchUser (name, mbox) ;
    if (nbr # 0) then
      userlist.Delete (nbr)
    else
      raise BadName
    end ;
    end DeleteSubscriber;

  method ListSubscriber (out number :Integer; text :String[80]) ;
    user : Entry;
    i : Integer ;
    begin
    text := "" ;
    number := userlist.NbItems ;
    if (number # 0) then
```

```
      userlist.First;
      for i := 1 to number do
        user := userlist.Next ;
        text := text + "\001" + user.name ;
      end ;
    end ;
    end ListSubscriber;

method Init ;
  begin
  userlist := EntryList.new ;
  end Init;

end MailUserList.
```

L'organisation de l'application messagerie telle qu'elle est présentée ici respecte les directives donnée dans la norme ISO-X400. On peut cependant remarquer que sa programmation dans le langage Guide a conduit à une structuration différente de celle traditionnellement rencontrée. Cette structure est caractérisée par le fait que la répartition n'apparaît jamais explicitement, ce qui est une conséquence du choix de transparence du système Guide. En fait, les divers objets (répertoires, boîtes aux lettres, messages) peuvent être répartis sur différents sites sans que cette répartition intervienne dans la programmation.

Différents aspects caractéristiques du modèle Guide sont mis en valeur et exploités dans l'application messagerie. Les plus importants sont rappelés ci-dessous :

• La mise en œuvre du principe d'abstraction qui consiste à cacher la réalisation physique des objets : les réalisations des messages, boîtes aux lettres et des autres objets utilisés par l'application peuvent être modifiées indépendamment les unes des autres.

• La propriété de conformité entre les interfaces : la diffusion de messages complexes tels que des documents par exemple est possible si le type des objets documents est conforme au type des objets messages. Lorsque Guide fournira l'héritage multiple, de tels objets pourront être construits facilement par héritage à partir des objets documents et messages.

• Le mécanisme de composition d'objets par références : il permet le partage d'objets et notamment ici de messages entre différentes boîtes aux lettres. Cette propriété permet de réaliser facilement la diffusion de messages.

• La transparence de la répartition : elle permet un découplage entre la structuration de l'application et la prise en compte des problèmes dûs à la localisation des objets. La répartition des objets sur les différents sites du système peut notamment être commandée indépendamment de l'application par des considérations de commodité, de sécurité, de disponibilité, de performances, etc.

2. GESTION D'UNE BIBLIOTHEQUE

L'application présentée dans cette section permet de gérer des informations relatives à un ensemble de documents stockés dans une bibliothèque et de les faire évoluer en fonction des emprunts effectués par les abonnés de la bibliothèque. Chaque document est représenté par un objet contenant sa référence bibliographique (titre, auteurs, etc) et un indicateur d'emprunt. Un emprunteur est représenté par un nom.

Cette application illustre notamment les possibilités du mécanisme d'héritage. Les différentes sortes de références bibliographiques (livre, thèse, rapport, etc) sont définies par des types dont chacun est caractérisé par un format particulier. L'héritage permet de mettre en commun facilement des parties de ce format entre plusieurs types.

Le service de gestion fourni par l'application est relativement simple. Il permet de maintenir à jour la liste des documents disponibles dans une bibliothèque par l'intermédiaire des opérations suivantes fournies par l'objet agent de transfert :

- affichage de la liste des bibliothèques existantes (opération *ListLibNames*),
- création d'une nouvelle bibliothèque (opération *CreateLib*),
- choix de la bibliothèque de travail (opération *SetWorkingLibName*),
- saisie de la référence d'un nouveau document (opération *AddDoc*),
- prise en compte de l'emprunt d'un document (opération *BorrowDoc*),
- prise en compte du retour d'un document emprunté (opération *GiveBackDoc*),
- affichage de l'état de la base de références (opération *ListBaseContents*).

Les quatre dernières opérations sont effectuées sur la bibliothèque choisie comme base de travail. L'application permet de gérer plusieurs bases de références dont les noms sont catalogués dans un objet du type prédéfini *Catal* (cf Annexe I). Un utilisateur travaille toujours sur une seule base à la fois : cette base est appelée base de travail. Au cours d'une même session, il peut cependant travailler successivement sur différentes bases. Les documents qui constituent une bibliothèque peuvent être de différents types. On considère ici les livres, les actes de congrès, les thèse et les rapports de recherche.

Les objets permanents qui constituent l'application sont décrits sur la figure 3. Leurs types sont les suivants :

• Documents : types *Document*, *Book*, *Proceeding*, *Thesis*, *Report*. Ces objets représentent les références des documents qui constituent les bibliothèques. Les types *Book*, *Proceeding*, *Thesis* et *Report* sont construits par sous-typage du type *Document* et leurs exemplaires peuvent donc être utilisés comme des exemplaires de ce type.

• Bases de documents : type *DocBase*. Les objets de ce type sont des listes de références à des objets de type *Document*.

• Bibliothèques : type *LibraryDataBase*. Les objets de ce type représentent les bibliothèques. Chaque bibliothèque est constituée de quatre objets de type DocBase qui contiennent réciproquement la liste des documents de type *Book*, de type *Proceedings*, de type *Thesis* et de type *Report* qui constituent la bibliothèque.

Le type *LibTransferAgent*, qui spécifie l'interface de l'objet 'agent de tranfert' de l'application est décrit ci-dessous ; la classe qui le réalise ne présente pas d'intérêt particulier et n'est donc pas décrite ici.

```
type LibTransferAgent is

  method ListLibNames : ref File ;
       // Retourne la liste
       // des noms des bibliothèques existantes

  method CreateLib (in newLibName : String[10]) ;
    signal NameAlreadyUsed ;
       // Crée une nouvelle bibliothèque

  method SetWorkingLibName (in libName : String[10]) ;
    signal BadName ;
       // Met à jour le nom de la bibliothèque
       // sur laquelle porteront les opérations
       // AddDoc, BorrowDoc, GiveBackDoc et ListBaseContents

  method AddDoc (in newDoc : ref Document) ;
    signal UnkownWorkingBase ;
       // Prise en compte d'un nouveau document

  method BorrowDoc
       (in docCopy : ref Document, borrower : String[10]) ;
    signal UnkownWorkingBase, BadDoc, NotAvailable ;
       // Emprunt d'un document

  method GiveBackDoc (in docCopy : ref Document) ;
    signal UnkownWorkingBase, BadDoc ;
       // Retour d'un document

  method ListBaseContents : ref File ;
    signal UnkownWorkingBase ;
       // Retourne un fichier contenent la liste des références
       // bibliographiques contenues dans la base de travail

end LibTransferAgent.
```

L'état d'une bibliothèque est mémorisé dans un objet de type *LibDataBase*. La figure 3 donne la spécification de ce type. La classe *LibDataBase* qui le réalise est décrite ci-dessous :

```
class LibDataBase
implements LibDataBase is

libName : String ;
bookBase,procBase,thesisBase,reportBase : ref DocBase ;

method AddDoc (in newDoc : ref Document) ;
  begin
  typecase newDoc of
    Book : bookBase.Append(newDoc) ;
    Proceedings : procBase.Append(newDoc) ;
    Thesis : thesisBase.Append(newDoc) ;
    Report : reportBase.Append(newDoc) ;
  end ;
  end AddDoc ;

method BorrowDoc
     (in docCopy : ref Document, borrower : String)
  signal BadDoc, NotAvailable ;
  doc : ref Document ;
  begin
  typecase docCopy of
    Book : doc := bookBase.Search(docCopy) ;
    Proceedings : doc := procBase.Search(docCopy) ;
    Thesis : doc := thesisBase.Search(docCopy) ;
    Report : doc := reportBase.Search(docCopy) ;
  end ;
  except NotPresent : raise BadDoc ; end ;
  doc.Borrow (borrower) ;      // (*)
  return doc ;
  end BorrowDoc ;

method GiveBackDoc (in docCopy : ref Document)
  signal BadDoc ;
  doc : ref Document ;
  begin
  typecase docCopy of
    Book : doc := bookBase.Search(docCopy) ;
    Proceedings : doc := procBase.Search(docCopy) ;
    Thesis : doc := thesisBase.Search(docCopy) ;
    Report : doc := reportBase.Search(docCopy) ;
  end ;
  except NotPresent : raise BadDoc ; end ;
  doc.GiveBack ;
  end GiveBackDoc ;

method ListBase : ref File ;
  begin
  return (bookBase.List
        & procBase.List & thesisBase.List & Report.List)
  end ListBase ;

method Init (in baseName : String) ;
  begin
  libName := baseName ;

  bookBase         := DocBase.new ;
  proceedingsBase  := DocBase.new ;
  thesisBase       := DocBase.new ;
  reportBase       := DocBase.new ;
  end Init ;

end LibDataBase.
```

Comme dans l'application précédente le mécanisme de propagation d'exception est exploité dans le code de la classe ci-dessus (cf instruction (*)).

Les différents documents stockés dans une bibliothèque peuvent être de différentes natures : livres, actes de congrès, thèses et rapports de recherche. Leurs types sont construits par sous-typage du type *Document* et leurs classes respectives par héritage de la classe qui le réalise. Le type et la classe *Document* sont décrits ci-dessous.

```
type Document is
   title  : String ;
   author : String ;
   date   : Date ;
   method Borrow (in borrower : String)
      signal NotAvailable ;
      // mise à jour de la référence lors d'un emprunt
      // la méthode retourne une exception si le document
      // n'est pas disponible
   method GiveBack ;
      // mise à jour de la référence lors du retour du document
   method List (in f_output : ref File) ;
      // complément du fichier avec le contenu
      // de la référence
end Document.

----------

class Document
implements Document is

borrowed    : Boolean = false ;
borrowerName  : String ;

method Borrow (in borrower : String)
   signal NotAvailable ;
   begin
   if borrowed then
      raise NotAvailable
   else
      borrowed    := true ;
      borrowerName  := Borrower ;
   end ;
   end Borrow ;

method GiveBack
   begin
   borrowed := false ;
   end GiveBack ;

method List (in f_output : ref File) ;
   begin
   f_output.WriteLnString ("%T" + title) ;
   f_output.WriteLnString ("%A" + author) ;
   f_output.WriteLnInteger ("%D" + date) ;
   end List ;

end Document.
```

La description du type *Book* et de la classe qui le réalise est donnée dans la suite. Les types et classes *Proceedings*, *Thesis* et *Report* sont construits de façon identique par sous-typage et héritage du type et de la classe *Document* (ils ne sont pas décrits explicitement ici).

```
type Book subtype of Document is
  editor : String ;
  editionNbr : Integer ;

  // Définitions héritées

  // title  : String ;
  // author   : String ;
  // date     : Date ;
  // method Borrow (in borrower : String)
  //   signal NotAvailable ;
  // method GiveBack ;
  // method List (in f_output : ref File) ;

end Book.

----------

class Book subclass of Document
implements Book is

// Méthodes héritées sans surcharge

// method Borrow (in borrower : String)
//   signal NotAvailable ;
// method GiveBack

// Méthode héritée et surchargée

method List (in f_output : ref File) ;
  begin
  super'List (f_output) ;
  f_output.WriteLnString ("%E" + editor) ;
  f_output.WriteLnInteger ("%N" + editionNumber) ;
  end List ;

end Book.
```

On peut remarquer dans le code de la méthode *List* ci-dessus l'emploi du mot clé *super* pour faire référence à la forme héritée de l'opération.

Chaque base de références bibliographiques (bibliothèque) est constituée d'un ensemble de sous-bases, contenant chacune une liste de références à des documents de même type. Chaque sous-base est réalisée par un objet du type *DocBase* décrit ci-dessous.

```
type DocBase subtype of List of [ref Document] is

  method Search (in docCopy : ref Document) : ref Document
    signal NotPresent ;
    // recherche d'une copie de la référence donnée en argument
    // dans la base de références
    // si elle est trouvée, la méthode retourne
```

```
    // une référence Guide sur la copie cataloguée,
    // sinon elle signale l'exception NotPresent

  method List : ref File ;
    // la méthode produit un fichier
    // décrivant l'état de la base de références

end DocBase.
```

La relation de conformité qui existe entre le type *Document* et ses sous-types *Book*, *Proceedings*, *Thesis* et *Report*, permet de concevoir les objets de type *DocBase* comme des objets génériques, au sens où ils correspondent chacun à une liste d'objets de type *Document*, ou d'un type qui lui est conforme. Le classement d'une référence à un document donné dans la bonne sous-base est laissée à la charge de l'objet de type *LibDataBase*. Ce dernier possède également une interface générique dans laquelle seul le type *Document* apparaît, mais prend en compte la nature effective de chaque référence par l'intermédiaire d'un contrôle dynamique de conformité (instruction *typecase*). Cette généricité permet de prendre en compte facilement un nouveau type de document (revues, articles, etc) : seuls la classe *LibraryDataBase* et l'objet 'agent d'interface' doivent pour cela être modifiés.

La classe qui réalise le type *DocBase* est décrite ci-dessous. Elle est construite par héritage de la classe *List of [ref Document]*. Le constructeur *List* est décrit en annexe II.

```
class DocBase subclass of List of [ref Document]
implements DocBase is

// variables d'état déclarées dans la classe List of [Document]
// et héritées par la classe DocBase
// contents : Array [100] of Document ;
// ptr, nbItems : Integer = 0,0 ;

method Search (in docCopy : ref Document) : ref Document
  signal NotPresent ;
  begin
  ptr := 0 ;
  while (ptr < NbItems) do
    if contents[ptr].Equal(docCopy) then
      return contents[ptr] end ;
    ptr := ptr + 1 ; end ;
  raise NotPresent ;
  end Search ;

method List : ref File ;
  result : ref File ;
  doc : ref Document ;
  begin
  result := File.new ; self.First ; doc := self.Next ;
  while doc # nil do
    doc.List (result); result.WriteLn; doc := self.Next; end;
  return result ;
  end List ;

end DocBase.
```

On peut remarquer dans le corps de la méthode *Search* l'appel de la méthode *Equal* pour la comparaison de l'état des objets désignés par les références *docCopy* et *contents[ptr]*. Cette méthode est réalisée par la classe *Top* et héritée par toutes les autres classes (cf III). Le type *Top* est décrit dans l'annexe I. Dans le code de la méthode *List*, on peut noter l'emploi du mot clé *self* pour l'appel d'opérations sur l'objet courant.

Les objets listes tels qu'ils sont réalisés par le constructeur *List* utilisé ici sont des objets non partageables. De façon à garantir l'intégrité des bases de références qui représentent les bibliothèques gérées par l'intermédiaire de l'application, il est donc nécessaire de compléter la classe *LibDataBase* décrite précédemment par la clause suivante :

```
control
  AddDoc      : exclusive ;
  BorrowDoc   : exclusive ;
  GiveBackDoc : exclusive ;
```

Cette clause garantit que les opérations *AddDoc*, *BorrowDoc* et *GiveBackDoc* définies sur les objets de classe *LibDataBase* sont toujours exécutées en exclusion mutuelle : plusieurs bibliothécaires peuvent ainsi travailler sans risque sur une même base de travail.

Comme dans l'application messagerie, certaines caractéristiques du modèle Guide sont exploitées dans cette application de gestion de bibliothèque. On peut citer notamment les suivantes :

• La persistance implicite des objets : la sauvegarde des bases de références n'a pas à être gérée par l'application.

• La mise en œuvre des conditions d'exécution : elles permettent le partage d'objets et notamment ici le travail simultané de différentes personnes sur la même base de références.

• Le mécanisme d'héritage : il permet l'exploitation du code existant pour la prise en compte de nouveaux types de documents tels que par exemples les journaux scientifiques ou les articles.

• La propriété de conformité : elle permet de conserver un niveau d'abstraction élevé dans la description des bases de références ce qui facilite l'évolution de l'application et notamment la prise en compte de nouveaux types de documents (journaux, articles).

• Le mécanisme de composition par références : il permet le partage d'objets entre différentes applications. Une application de gestion de bases de références bibliographiques comparable à l'application BIBTEX disponible sur Unix pourrait par exemple utiliser les mêmes objets que l'application de gestion de bibliothèque.

4. CONCLUSION

Les deux applications présentées dans ce chapitre ont été mises en œuvre sur le premier prototype réparti du système Guide. Les remarques suivantes, relatives à l'utilisation du modèle et du langage, sont tirées de cette première expérience.

Conception des applications

On a pu constater qu'une fois bien intégrées les constructions de la programmation par objets, ce qui nécessite une phase d'apprentissage, les applications peuvent être développées rapidement. Cette programmation amène une modification des méthodes de décomposition et de conception comme l'illustre l'exemple de la messagerie. Le principal outil conceptuel n'est plus la conception descendante habituellement pratiquée, mais une conception en va et vient organisée notamment sur la base des données manipulées par l'application. Elle consiste d'une part à regrouper dans une classe des données et des fonctions logiquement corrélées et indépendantes des autres, et d'autre part, à réutiliser les classes déjà existantes. Cette faculté de réutilisation incite d'ailleurs à concevoir d'emblée des classes réutilisables. Nous verrons cependant au chapitre VI que la phase d'apprentissage à la programmation par objets ne doit pas être négligée. Les concepteurs d'application influencés par leurs habitudes de programmation ont en effet naturellement tendance à n'exploiter qu'une petite partie des outils que leurs fournissent les modèles de programmation par objets.

Persistance des objets

La mise en œuvre du modèle Guide sur un système à objets persistants est perçue comme un progrès au niveau de la programmation : il n'est en effet plus nécessaire de se préoccuper de la sauvegarde explicite des objets utilisés par une application. Il faut néanmoins noter deux contreparties :

• La première est l'apprentissage nécessaire de la part du programmeur qui doit prendre conscience de la persistance explicite des objets qu'il crée. Cette propriété l'oblige par exemple à utiliser le service de noms avec précautions, notamment lors de la phase de mise au point d'une application. Des programmeurs pensant tester une nouvelle version de leur application se sont en effet rendu compte au bout d'un certain temps qu'ils continuaient à travailler sur les objets (permanents) créés lors de la première exécution de leur programme ! Ce problème met en évidence la nécessité d'un mécanisme de gestion de versions.

• La seconde contrepartie est l'encombrement rapide de la mémoire d'objets : un ramasse-miettes actuellement en cours de réalisation est en effet nécessaire dès que

plusieurs utilisateurs travaillent sur le système, et lorsque des applications plus importantes que celles décrites ici sont réalisées.

Invisibilité de la répartition

La dernière remarque concerne la localisation des objets et la transparence de la répartition du système support. Une conséquence du choix de la transparence est la disparition de la différence entre applications réparties et centralisées. Le système Guide permet la programmation d'applications réparties aussi simplement que la programmation d'applications destinées à s'exécuter sur des systèmes mono-sites. Le fait de renvoyer au système d'exécution le travail de localisation et de liaison simplifie le travail de conception en permettant notamment de mettre au point les applications en environnement centralisé avant d'étendre leur exécution à plusieurs sites. L'introduction explicite de la répartition est néanmoins possible dans les applications qui l'exigent.

Mise en œuvre et performances

Le choix de réaliser la première version du système Guide sur Unix était essentiellement motivé par le souhait de disposer rapidement d'une première version du système aux fins d'expérimentation. Cet objectif a été atteint mais on peut remarquer que ce choix de réalisation a comme contrepartie une dégradation des performances due à l'empilement de couches de logiciel et aux limitations de système Unix. Une analyse fine des temps d'exécution est en cours et doit permettre des améliorations. Compte tenu de ce problème de performances, le fait de pouvoir mettre au point les applications en milieu centralisé avant de les étendre sur l'ensemble du système présente un réel avantage.

L'expérimentation du langage et du système a permis également de mettre en évidence la nécessité de fournir un environnement de développement adapté aux besoins des utilisateurs. Cet environnement devrait notamment fournir les outils suivants :

- Un outil de mise au point : elle se fait actuellement sur le code C généré par le compilateur et il n'y a pas d'outil de mise au point répartie.
- Des outils d'aide au développement de programme tels qu'un éditeur syntaxique et un interpréteur de conformité par exemple.
- Un mécanisme de gestion de versions et notamment un gérant de types et de classes élaboré.

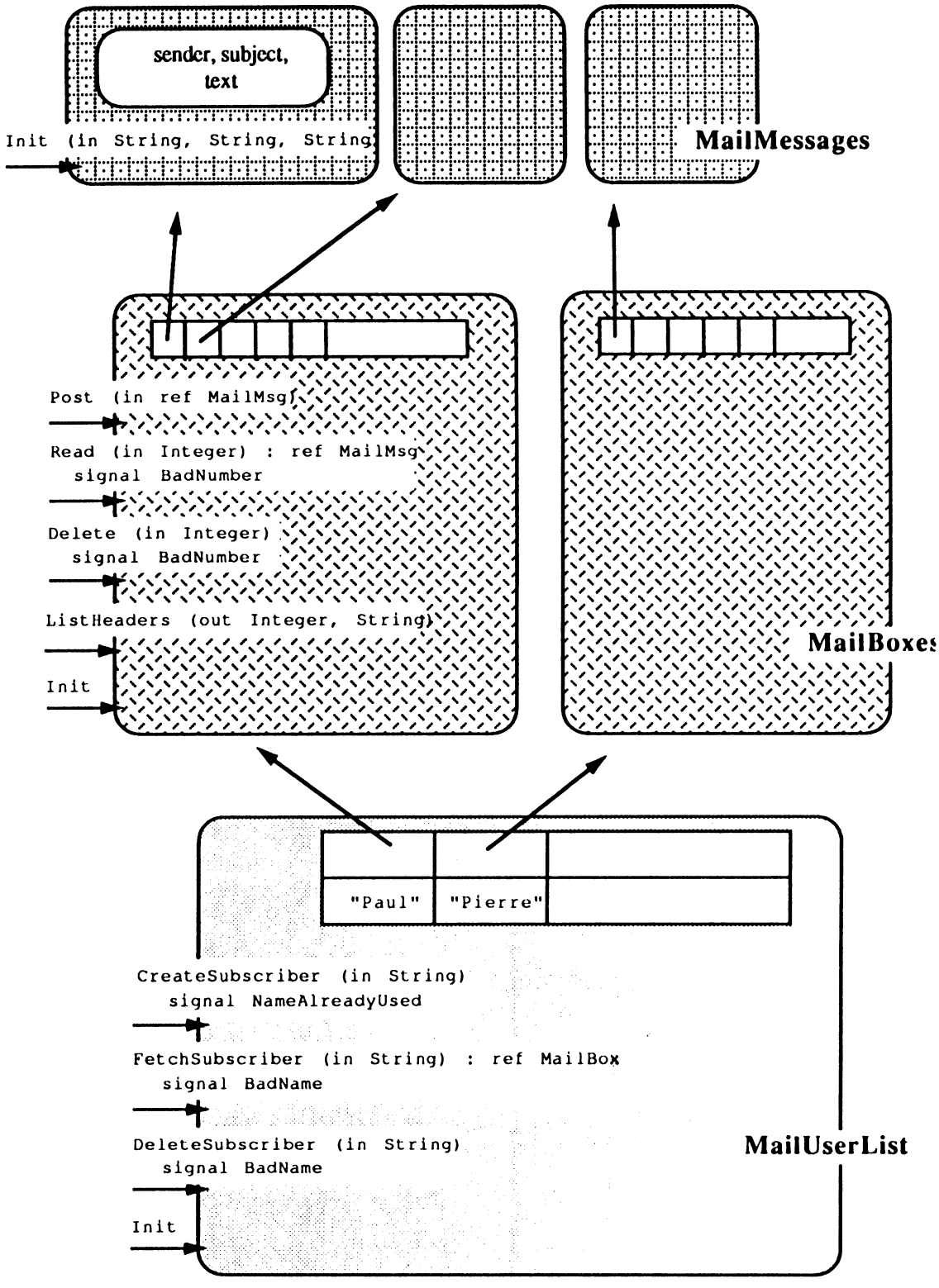


Figure 2. Objets permanents qui constituent l'application Messagerie

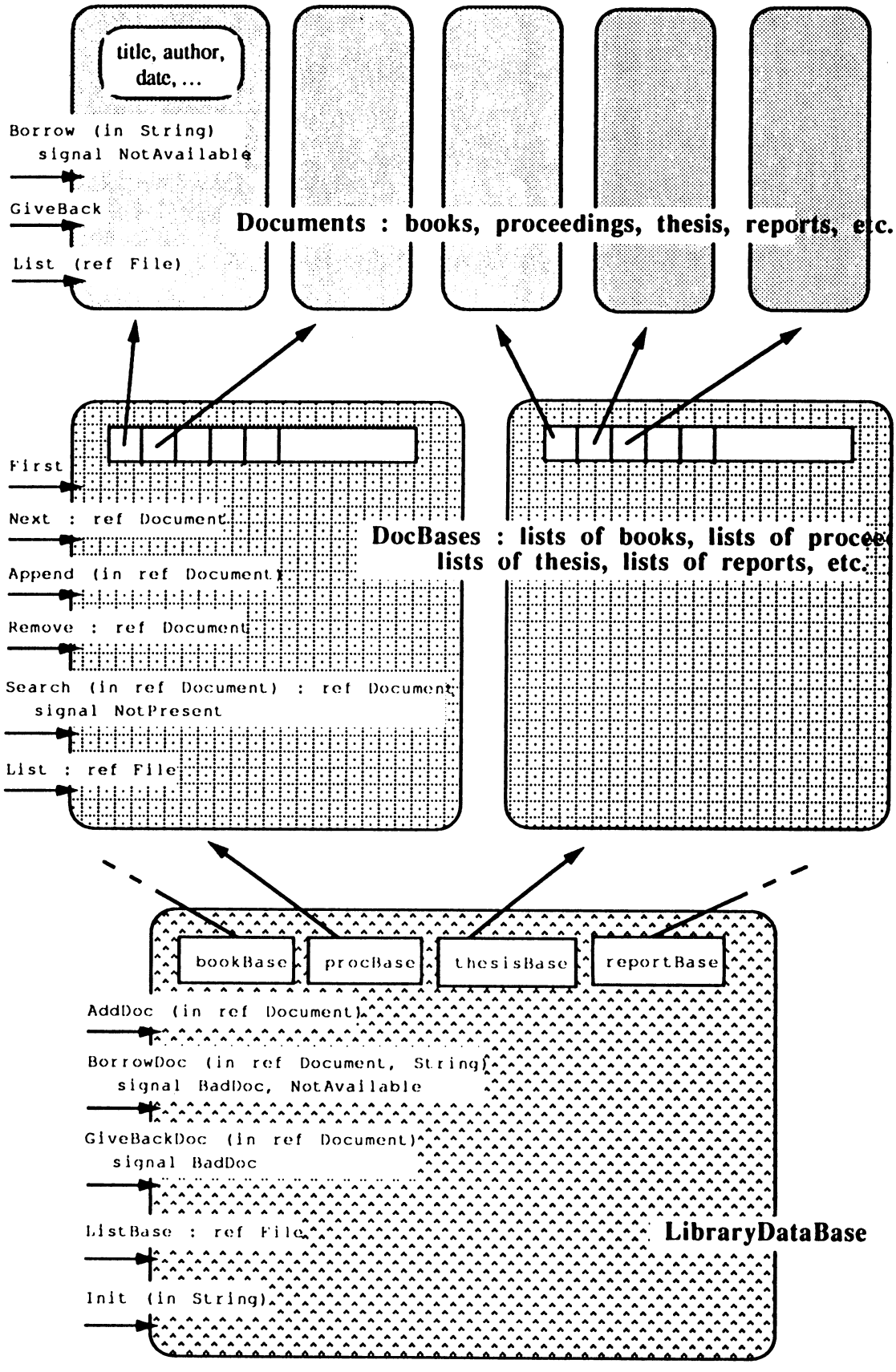


Figure 3. Objets permanents qui constituent l'application Bibliographie

CHAPITRE V

MISE EN ŒUVRE D'UNE APPLICATION GUIDE

1. INTRODUCTION

L'objet de ce chapitre est la description du processus de compilation et d'exécution d'une application dans le système Guide. Cette première section donne les bases nécessaires à la compréhension de la suite en décrivant les principes de la machine virtuelle Guide.

Je rappelle que l'objectif du modèle d'exécution de Guide est de présenter à ses utilisateurs une machine virtuelle multiprocesseurs dans laquelle le parallélisme est apparent et la distribution cachée (la machine virtuelle peut être multi-sites). Le premier prototype de cette machine a été développé dans un but d'expérimentation au dessus d'Unix System V [Decouchant 88]. C'est ce premier prototype qui est décrit ici.

L'architecture globale de la machine virtuelle, présentée sur la figure 1, fait apparaître deux niveaux d'abstraction. Le premier est construit directement sur les ressources physiques ; il est constitué d'un ensemble de processeurs virtuels qui partagent une mémoire d'objets commune contenant tous les objets du système : il constitue la machine à objets de Guide. Le second niveau est celui des applications Guide ; il met en œuvre les concepts de domaines et d'activités.

Un **domaine** est une machine multi-activités à mémoire commune, éventuellement multi-sites. A chaque **activité** d'un domaine est associé un processeur virtuel de la machine à objets. Ces processeurs virtuels partagent la même mémoire virtuelle. L'ensemble des objets qui composent le contexte d'un domaine est contenu dans la mémoire virtuelle de ce domaine.

Un processeur virtuel est une machine qui interprète des appels d'opération sur des objets. Un processeur virtuel est réalisé par plusieurs processeurs physiques dans les cas où il manipule des objets situés sur différents sites. La mémoire virtuelle d'un processeur virtuel contient l'ensemble des objets qu'il peut adresser.

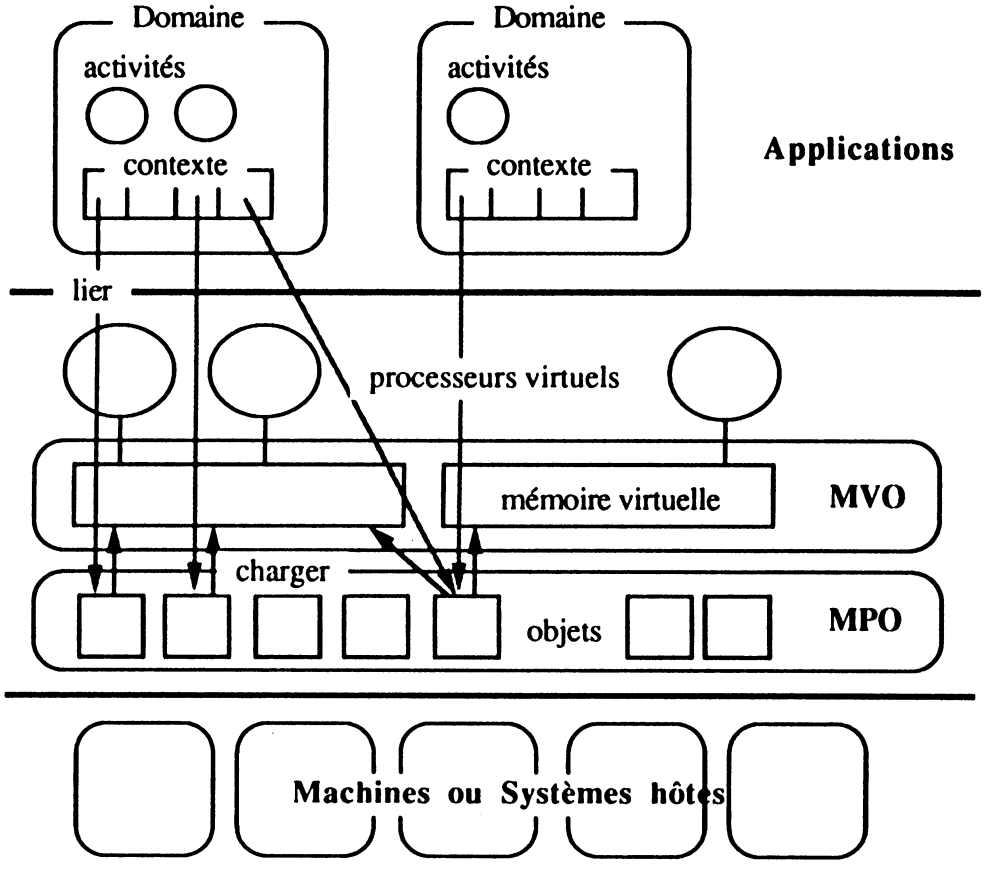


Figure 1. Architecture de la machine virtuelle Guide

La mémoire d'objets peut être vue comme une mémoire segmentée dans laquelle l'unité d'allocation et d'adressage est l'objet. Elle regroupe les concepts de Mémoire Virtuelle d'Objets et de Mémoire Permanente d'Objets. Elle contient tous les objets du système et masque la répartition. Un objet y est désigné par un identificateur unique.

Dans la suite de ce chapitre, je m'intéresse aux mécanismes mis en œuvre pour l'exécution d'une application sur la machine à objets de Guide telle qu'elle vient d'être définie, c'est à dire à un niveau auquel on fait abstraction de la répartition. Je ne donne que les informations nécessaires à la compréhension des principes généraux des mécanismes décrits. Une description détaillée des points abordés est donnée dans les différents rapports du projet dont la liste est donnée dans la bibliographie.

2. CONCEPTION D'UNE APPLICATION

Une application est constituée d'un ensemble d'objets interagissant par le biais d'appels à leurs opérations. Elle est définie par la désignation d'un objet et d'une opération de cet objet. Le code de cette opération peut d'une certaine manière être considéré comme le programme principal de l'application, et comparé dans ce sens au code de la fonction *main* d'une application écrite en langage C ou au code de la fonction *program* d'une application écrite en langage Pascal.

Dans l'état actuel du prototype, la programmation et la compilation d'une application sont faites sous Unix. Le compilateur gère un espace de types et de classes à deux niveaux constitué de deux répertoires :

- un répertoire commun à tous les utilisateurs contient les descriptions des types et des classes prédéfinis, ainsi que celles des types et classes fournis par la bibliothèque,
- un répertoire propre à chaque utilisateur contient les descriptions des types et des classes qui lui sont propres.

Programmer une application consiste à définir les types et les classes des objets qu'elle met en œuvre. Pour chaque élément (type ou classe) compilé, le compilateur génère un ensemble de fichiers qu'il stocke dans l'un ou l'autre des répertoires décrits ci-dessus. Chaque type ou classe peut être compilé séparément dans la mesure où les types ou classes qu'il ou elle utilise l'ont été précédemment.

2.1 COMPILATION

Le compilateur du langage Guide produit en cinq passes du code exécutable par le système Guide. Son architecture générale est décrite par la figure 2.

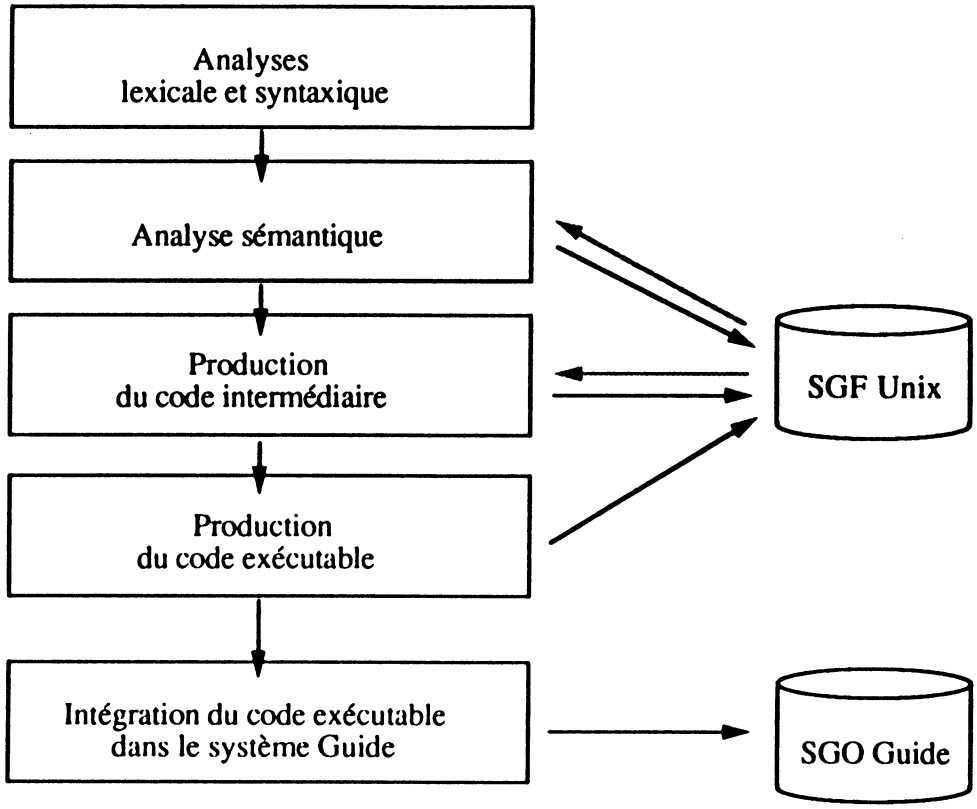


Figure 2. Architecture du compilateur Guide

Les passes 2, 3, et 4 produisent les fichiers qui constituent la base de données du compilateur. Cette base est constituée des deux répertoires décrits précédemment ; elle est utilisée lors des passes 2 et 3. Lors de la compilation d'une classe, la passe 5 produit l'objet Guide correspondant.

Seule la compilation d'une classe donne lieu à la production de code. Les deux dernières passes ne sont donc pas mises en œuvre lors de la compilation d'un type.

Analyses lexicale, syntaxique et sémantique - Passes 1 et 2

L'analyse lexicale consiste à transformer la chaîne de caractères correspondante au texte source en une chaîne de symboles. Elle détecte les caractères illégaux.

L'analyse syntaxique consiste à vérifier si la chaîne de symboles produite par l'analyseur lexical obéit aux règles grammaticales du langage Guide. Si aucune erreur n'est détectée, elle produit un arbre syntaxique qui représente l'objet compilé sous une forme interne au compilateur.

Les analyseurs lexical et syntaxique sont mis en œuvre à l'aide des outils LEX et YACC d'Unix.

L'analyse sémantique consiste à parcourir l'arbre construit par l'analyseur syntaxique et à le compléter par l'ensemble des informations nécessaires à la génération du code exécutable. Elle fait appel à la base de données du compilateur pour la recherche des descriptions des types et des classes utilisés[1] par l'objet (type ou classe) en cours de compilation et effectue le contrôle de conformité.

Le compilateur recherche la description d'un type ou d'une classe en parcourant d'abord le répertoire contenant l'ensemble des descriptions de types et de classes propres à l'utilisateur. Il parcours ensuite le répertoire contenant les descriptions des types et classes partagés.

Si aucune erreur n'est détectée pendant l'analyse sémantique (utilisation d'un type ou d'une classe non défini, conditions de conformité non respectées, etc), un fichier contenant la description de l'objet en cours de compilation est inséré la base de données du compilateur.

Production du code intermédiaire - Passe 3

La production du code intermédiaire consiste à définir les structures de données correspondantes aux objets et à traduire le code des méthodes et des procédures en code C. Cette passe s'applique donc essentiellement à la compilation d'une classe. Elle est cependant mise en œuvre à la compilation d'un type pour la production des séquences de code relatives à la construction des blocs de paramètres des opérations spécifiées par le type. Une opération définie sur un objet Guide est traduite en langage C en une procédure qui n'admet qu'un seul paramètre : l'adresse d'un bloc descriptif des paramètres effectifs de l'opération. Ce choix d'un modèle uniforme pour la réalisation des opérations est imposé par la nature dynamique de la liaison des objets et du code à l'exécution qui nécessite la mise en œuvre de primitives du système.

Les appels d'opération sur les objets sont traduits par le compilateur en deux parties distinctes : la constitution du bloc de paramètres et l'appel proprement dit. Ce dernier est réalisé de différentes façons selon que l'objet appelé est externe (désigné par une référence) ou interne (objet interne ou de travail) à l'objet appelant :

---

[1] L'ensemble des types et des classes dont la description est nécessaire sont ceux qui apparaissent dans les signatures de méthodes ou de procédures, dans les déclarations de variables et dans les clauses de sous-typage et d'héritage.

• l'appel d'une opération sur un objet désigné par une référence nécessite la mise en œuvre des mécanismes de liaison dynamique et fait en conséquence appel à une primitive du système Guide[1] (cette primitive appelée *GuideCall* est décrite dans la suite),

• l'appel d'une opération sur un objet interne ou de travail est réalisé par un branchement statique au code de l'opération, ce dernier étant lié statiquement au code de la classe en cours de compilation[2].

L'appel d'une procédure est réalisé selon un schéma similaire à celui de l'appel d'une opération sur un objet, c'est à dire en deux étapes : constitution du bloc de paramètres et appel, ce dernier étant réalisé par un branchement statique au code de la procédure.

La passe 3 produit un certain nombre de fichiers de description de l'objet en cours de compilation. Lors de la compilation d'une classe qui utilise cet objet, le compilateur se servira de ces fichiers pour la mise en œuvre de la passe 3.

Production du code exécutable - Passe 4

La passe 4 n'est mise en œuvre que si l'objet en cours de compilation est une classe. Sa fonction consiste en la production d'un fichier contenant la description de la classe sous la forme requise par le système de gestion d'objets de Guide. Comme l'illustre la figure 2, l'état d'un objet classe se compose de quatre parties :

• une liste d'informations concernant les exemplaires de la classe,

• une liste d'informations concernant la classe elle-même,

• une table descriptive de l'ensemble des méthodes définies sur les exemplaires de la classe,

• le code des méthodes définies sur les exemplaires de la classe et réalisées par la classe elle-même (le code d'une méthode héritée n'est pas dupliqué).

Actuellement, la liste des informations concernant les exemplaires de la classe contient une unique information : leur taille. Elle doit être complétée d'un descripteur de leur état. Ce descripteur permettra notamment la mise en œuvre du ramasse-miettes.

---

[1] Les instructions <self.op (...)> sont notamment réalisées selon ce schéma.

[2] Pour plus d'efficacité, les appels aux objets internes ou de travail de classes élémentaires, de classe String et de classe Array ne sont pas réalisés de cette manière mais compilés.

La liste des informations concernant la classe contient les informations qui permettent de situer cette dernière dans l'arbre d'héritage. Elle donne, sous forme d'une table, la liste des noms de ses super-classes donnés par ordre d'apparition dans la hiérarchie. Lors d'une construction de classe par héritage, le code des méthodes héritées mais non redéfinies n'est pas dupliqué dans le corps de la nouvelle classe. Cette table qui conserve l'identification des super-classes permet de retrouver le code de ces méthodes.

La table descriptive de l'ensemble des méthodes définies sur les exemplaires de la classe donne pour chaque méthode les informations suivantes :

- nom de la méthode,
- index dans la table décrite précédemment de la classe qui réalise la méthode,
- adresse de la méthode,
- adresse de la fonction de synchronization associée à la méthode.

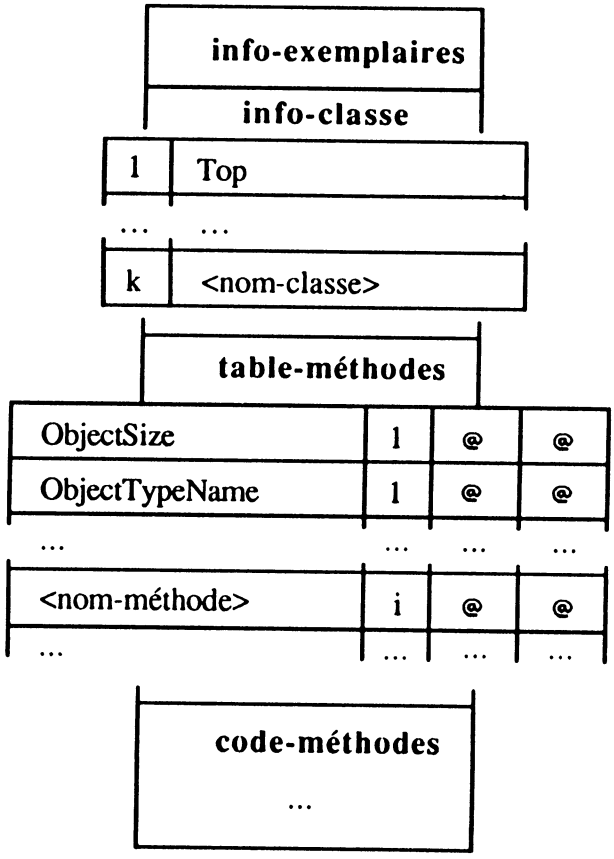


Figure 3. Structure de l'état d'un objet classe

Les deux tables contenues dans l'état d'un objet classe sont exploitées par les mécanismes de liaison dynamique mis en œuvre par le système. Leur contenu et leur utilisation sont détaillés dans la suite.

Les fonctions de synchronisation permettent l'évaluation des conditions d'exécution attachées aux opérations définies sur les objets synchronisés. Leur mise en œuvre est décrite dans l'annexe II.

La classe Top étant définie comme le sommet de l'arbre d'héritage des classes, on peut remarquer qu'elle apparaît, ainsi que les méthodes qu'elle définit, dans la description de l'état de toutes les autres classes.

Intégration du code exécutable dans le système Guide- Passe 5

La passe 5 permet de transférer la forme exécutable d'une classe de l'environnement de développement Unix, à l'environnement d'exécution Guide. Elle consiste à intégrer l'objet 'classe exécutable' dont l'état a été construit en passe 4 dans la Mémoire Permanente d'Objets de Guide. Cette intégration est faite par une application Guide dont l'exécution a l'effet suivant :

- création d'un nouvel objet classe,
- mise à jour de l'état de cet objet par recopie du contenu du fichier Unix produit en passe 4,
- association du nom de la classe à l'objet, par l'intermédiaire du service de désignation du système.

2.2 EXÉCUTION

Pour exécuter une application, l'utilisateur doit ouvrir une session de travail sous Guide. Dans l'état actuel du prototype, il a ensuite la possibilité de créer une application en donnant le nom d'une classe, par exemple *AppliClass*. Le système crée alors un nouveau domaine et son activité principale par laquelle il fait exécuter l'instruction suivante :

```
(AppliClass.new).Init ;
```

L'exécution de l'application consiste donc en l'appel de la méthode *Init* sur un nouvel exemplaire de la classe désignée. Dans l'environnement de développement d'Eiffel, les applications sont conçues de la même manière. Ce schéma est cependant un peu restrictif dans un système comme Guide qui possède une Mémoire Permanente d'Objets et dans lequel une application peut être définie plus généralement comme l'appel d'une opération sur un objet permanent. Les outils de développement qui sont en cours de réalisation supprimeront cette restriction. Ils sont décrits en 4.

3. LE SYSTEME D'EXÉCUTION

L'exécution d'une application donne lieu à la création d'un nouveau **domaine**. Chaque domaine est constitué d'un ensemble d'objets et d'activités qui évolue dynamiquement.

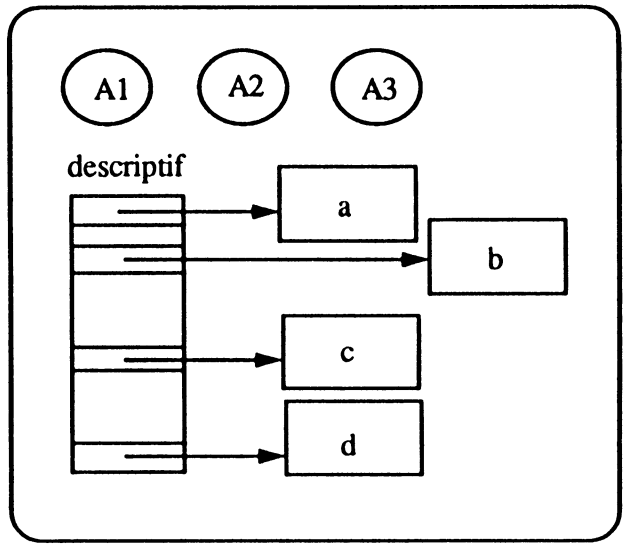


Figure 4. Représentation d'un domaine

Les objets accessibles aux activités du domaine constituent son **contexte** ; ils sont décrits par une table appelée **descriptif**. La figure ci-dessus représente un domaine qui possède trois activités. Une activité est toujours propre à un domaine. Le partage entre domaine ou entre activités se fait par le biais d'objets comme l'illustre la figure 4.

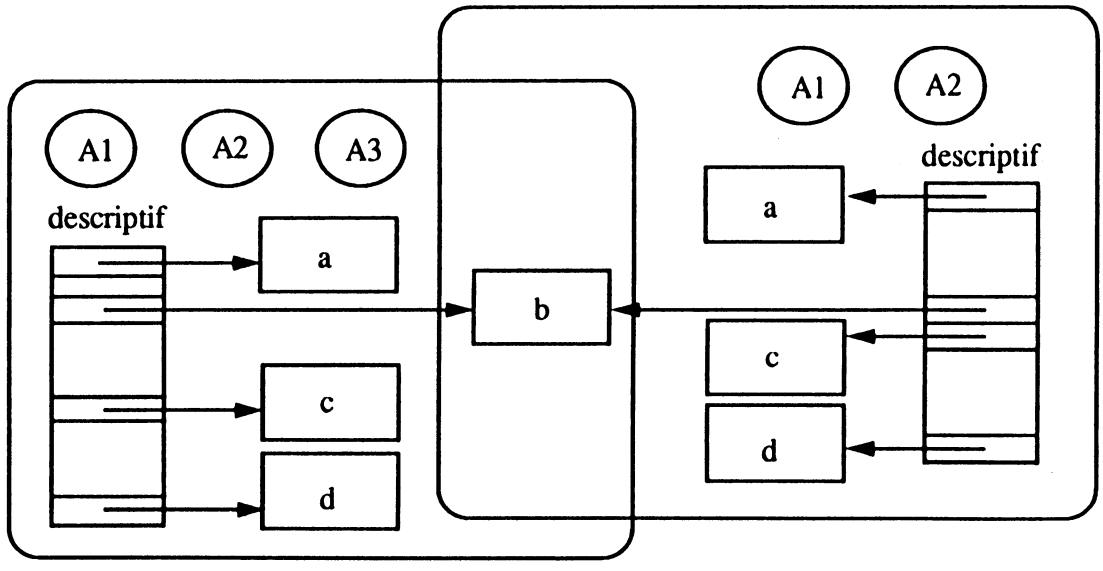


Figure 5. Partage d'objets entre deux domaines

Les domaines et les activités sont les seuls objets multi-sites de Guide. Le contexte d'un domaine peut en effet être constitué d'objets dispersés sur l'ensemble des sites du système, et une activité peut être amenée à s'étendre sur différents sites en fonction de la localisation des objets sur lesquels elle exécute des opérations.

Parmi les variables qui constituent l'état des objets domaines et des objets activités on peut noter des références à des objets qui représentent des portes d'entrées-sortie. Chaque domaine et chaque activité possèdent notamment une porte d'entrée désignée par la référence ***input*** et une porte de sortie désignée par la référence ***output***. A la création d'un domaine, les variables d'état *input* et *output* du domaine désignent le terminal de l'utilisateur qui a provoqué sa création. Les variables *input* et *output* de chaque activité créée dans le domaine sont initialisées par copie de ces variables.

La création de domaines et la création d'activités sont présentées dans la section suivante. La section 3.2 présente les mécanismes de liaison dynamique d'objet dans un domaine.

3.1 Création de domaines et d'activités

Le système crée un nouveau domaine pour chaque application. A sa création, un nouveau domaine possède une activité, appelée activité principale du domaine, et un contexte vide.

La création de nouvelles activités dans un domaine résulte de l'exécution d'un bloc parallèle (*co_begin-co_end*, cf III-11). Le système crée une nouvelle activité pour l'exécution de chaque branche du bloc (le bloc parallèle est le seul outil fourni au programmeur pour la création explicite d'activités).

3.2 Liaison dynamique d'objets dans un domaine

La liaison dynamique d'objets dans un domaine est nécessaire à chaque fois qu'une activité requiert l'accès à un objet non présent dans le contexte du domaine.

La liaison d'un objet dans le contexte d'un domaine, consiste à allouer une entrée pour l'objet dans le descriptif du domaine. Elle nécessite la présence de l'objet dans la Mémoire Virtuelle d'Objets. Deux cas peuvent se présenter :

• l'objet est déjà lié dans le contexte d'un autre domaine ; il est donc déjà présent dans la MVO : il suffit de mettre à jour le descriptif du domaine demandeur,

• l'objet n'est pas présent dans la MVO : le système fait appel à la Mémoire Permanente d'Objets qui charge l'objet ; on est alors ramené au cas précédent.

Création d'un objet - Primitive GuideCreate

La création d'un objet est réalisée par l'appel de la primitive *GuideCreate* fournie par le système Guide.

En langage Guide, l'instruction de création s'écrit *<ref := aClass.new>* ; elle est traduite par le compilateur en un appel à la primitive *GuideCreate* : *<GuideCreate (@bloc-param)>*. Le bloc de paramètre dont l'adresse est transmise en paramètre à la primitive contient les informations suivantes :

• le nom de la classe dont un nouvel exemplaire doit être créé, c.-à-d. *aClass*,

• l'adresse de la référence *ref*.

La primitive *GuideCreate* effectue la suite d'actions suivante :

• liaison de la classe *aClass* dans le contexte du domaine demandeur (sauf si elle l'est déjà),

• demande d'un nouvel identificateur d'objet à la Mémoire Permanente d'Objets,

• création du nouvel objet,

• mise à jour de la référence dont l'adresse a été transmise dans le bloc de paramètres.

Appel d'une opération sur un objet - Primitive GuideCall

L'appel d'une opération sur un objet est réalisé par l'appel de la primitive *GuideCall* fournie par le système Guide.

En langage Guide, l'instruction d'appel d'une opération sur un objet s'écrit *<ref.op (...)>* ; elle est traduite par le compilateur en un appel à la primitive *GuideCall* : *<GuideCall (@bloc-param)>*. Le bloc de paramètre dont l'adresse est transmise en paramètre à la primitive contient les informations suivantes :

• la référence *ref* qui désigne l'objet sur lequel doit être exécutée l'opération,

• le nom de l'opération à exécuter,

• le nombre de paramètres, arguments et résultats, de cette opération,

• et pour chacun de ces paramètres, une information qui décrit sa nature (référence sur un objet externe ou valeur d'un objet interne) et donne sa valeur ou son adresse.

La description des paramètres de l'opération dans le bloc transmis à la primitive *GuideCall* permet au système d'effectuer les transferts entre opération appelante et

opération appelée. Elle est particulièrment utile dans les cas où le transfert doit être fait d'un espace virtuel Unix à un autre : appel d'une opération sur un objet distant ou exécution de l'opération par une activité différente de l'activité appelante (bloc parallèle).

La primitive *GuideCall* effectue la suite d'actions suivantes :

- liaison de l'objet dans le contexte du domaine demandeur (s'il n'y est pas déjà),
- liaison de la classe de l'objet dans le contexte du domaine demandeur (si elle n'y est pas déjà),
- recherche, dans l'état de l'objet classe, de l'adresse à laquelle est chargé le code de l'opération à exécuter (si l'opération est réalisée par une super-classe, la liaison de cette classe dans le contexte du domaine appelant est nécessaire, cf ci-dessous),
- exécution de l'opération.

Le système recherche le code de l'opération à exécuter en parcourant l'état de l'objet classe. Je décris ci-dessous le principe de recherche sur la base d'un exemple.

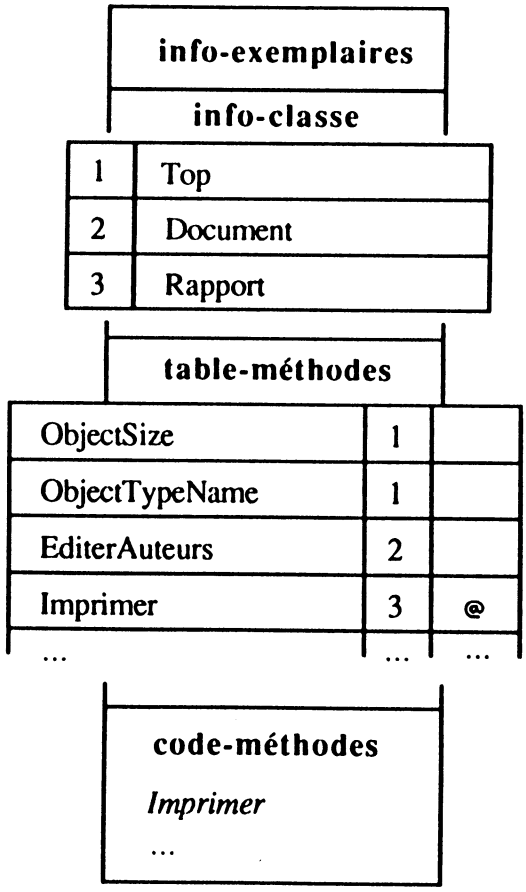


Figure 6. Structure de l'état de l'objet classe *Rapport*

Soit la classe *Rapport*construite par héritage de la classe *Document*, et les méthodes *EditerAuteurs* et *Imprimer* définies toutes les deux sur les rapports mais telles que la première soit réalisée par la classe *Document*(et héritée par la classe *Rapport*) et la seconde réalisée par la classe *Rapport*. La figure précédente représente l'état de la classe *Rapport*[1].

Pour trouver l'adresse du code de l'opération à exécuter, le système cherche dans la table des méthodes contenue dans l'état de l'objet classe, les informations relatives à cette opération. Deux cas peuvent se présenter :

• l'index de la classe qui réalise l'opération est celui de la classe courante (c'est le cas de l'opération *Imprimer* dans la figure 3) ; l'opération est donc réalisée par cette classe et son code, qui fait partie de l'état de la classe, est présent en mémoire ; son adresse est donnée dans la table ;

• l'index de la classe qui réalise l'opération est celui d'une super-classe de la classe courante (c'est le cas de l'opération *EditerAuteurs* dans la figure 3) ; l'opération est donc réalisée par cette super-classe qui doit en conséquence être liée dans le contexte d'exécution du domaine demandeur ; son nom est trouvé par le système dans la table qui décrit l'ensemble des super-classes de la classe courante ; l'adresse de l'opération est cherchée dans l'état de cette super-classe.

Les tables contenues dans l'état d'un objet classe sont construites par le compilateur. Ce dernier calcule les adresses données dans la table des méthodes de la façon suivante : l'ensemble des opérations réalisées par une classe est compilé en un bloc unique de code exécutable. L'adresse d'une opération dans la table est l'adresse relative de son code dans le bloc qui le contient. Ces adresses sont translatées par le système lorsque la classe est chargée en mémoire d'exécution ; la table contient alors des adresses absolues.

Le compilateur ne met à jour dans la table des méthodes d'une classe que les adresses des méthodes réalisées par la classe elle-même. La mise à jour par le système des adresses des méthodes réalisées par les super-classes peut éventuellement conduire à une réduction du coût moyen d'exécution de la primitive *GuideCall*. Des mesures sont en cours actuellement pour déterminer le gain qu'elle peut apporter.

---

[1] Par rapport à la description de l'état d'un objet classe donnée en 2.1, il manque ici les informations concernant les fonctions de synchronisation. Elles sont présentées en détail en annexe 2, et je suppose ici que les rapports sont des objets non synchronisés ce qui me permet de simplifier ma présentation.

4. OUTILS DE DÉVELOPPEMENT

Les premiers outils destinés à l'aide au développement des applications sur le système Guide sont en cours de réalisation. Ils doivent permettre aux progammeurs de développer et de faire évoluer leurs applications dans l'environnement 'objet' de Guide.

Un programmeur souhaitant travailler dans Guide, ouvrira une session de travail Guide. Une telle session sera mise en œuvre par un objet comparable à un interpréteur de commandes dont l'interface pourra être aussi bien graphique que textuelle. Cet interpréteur lui permettra par exemple de consulter la liste des objets catalogués par le service de désignation, de créer une application en demandant l'exécution d'une opération sur un de ces objets, de programmer et de compiler des types et des classes.

La mise en œuvre d'un gestionnaire de types et de classes doit permettre de maintenir la cohérence entre les objets permanents du système (types, classes et exemplaires) tout en donnant aux utilisateurs la possibilités de modifier un type ou une classe déjà intégré dans la Mémoire Permanente d'Objets. Il permettra de fournir aux programmeurs des outils d'aide au développement d'application tels que par exemple un éditeur syntaxique et des outils de déboguage.

L'espace d'objets doit également être partitionné en sous-espaces dont la visibilité et l'accès seront définis en fonction de propriétés propres à chaque utilisateur. Ce partitionnement sera réalisé par l'intermédiaire d'un système de droits d'accès et pris en compte par les règles de recherche du gestionnaire de types et de classes.

CHAPITRE VI

CONCLUSION

1. RAPPELS DES OBJECTIFS

Le projet Guide a pour vocation d'être le support d'un ensemble de recherches sur la programmation d'applications réparties. Son objectif principal est le développement d'un système d'exploitation grandeur nature à haut degré d'intégration, et qui doit servir de terrain d'expérimentation à ces recherches. Le degré d'intégration souhaité fait de la propriété de transparence un aspect clé de ce système ; elle peut être considérée à différents niveaux :

• *Transparence de la localisation* : l'utilisateur peut être déchargé de la nécessité de connaître la localisation des ressources qu'il demande ou le détail des protocoles de communication permettant d'y accéder.

• *Transparence de la réalisation* : les ressources du système peuvent être spécifiées sous la forme abstraite d'une interface d'accès. Ce type de spécification permet de rendre les programmes indépendants de la représentation des données qu'ils traitent et favorise la réutilisation et l'évolution des composants logiciels qui constituent une application.

• *Transparence de la conservation* : les utilisateurs peuvent être déchargés de la prise en compte de la sauvegarde et du maintien de la cohérence en mémoire permanente des informations qu'ils gèrent.

Le développement du système Guide a nécessité la conception d'un modèle de structuration et d'exécution des applications et d'un modèle de conservation de l'information qui fournissent le degré d'intégration souhaité. Pour ma part, je me suis

surtout intéressée à la conception du modèle de programmation et à la définition du langage et du compilateur pour sa mise en œuvre sur le système. L'objet de ma thèse était notamment de faire une première évaluation de ce modèle, sur la base d'une étude bibliographique et d'une première expérience de programmation.

2. RAPPELS DES RÉSULTATS OBTENUS

Un premier prototype du système est disponible depuis fin 1988. Le choix d'un modèle à objets comme structure de base de ce système et comme modèle pour la programmation des applications a permis d'obtenir le niveau d'abstraction souhaité. Le prototype permet la programmation d'applications réparties. Il est constitué des éléments suivants :

• Un *système de gestion d'objets* conçu comme un espace universel d'objets permanents.

• Un *système d'exécution* qui permet la réalisation des applications sous la forme d'un ensemble d'objets passifs et de processus opérant sur ces objets.

• Un *langage* qui met en œuvre le modèle à objets et sert de support à la programmation.

Le choix d'un modèle à objets pour la programmation des applications Guide a constitué une décision majeure en permettant l'exploitation des avantages que ce type de modèle apporte : un niveau d'abstraction élevé qui permet une programmation modulaire en termes de composants réutilisables et extensibles, un mécanisme d'héritage qui permet la réutilisation de composants logiciels par extension et spécialisation, et un contrôle statique qui permet d'allier sécurité et puissance d'expression.

Le modèle d'objets de Guide présente de plus un ensemble de caractéristiques intéressantes qui font son originalité :

• La définition des classes comme des objets particuliers ce qui permet la définition d'un modèle uniforme ; cette solution est de plus représentative de la façon dont les classes sont réalisées.

• La séparation explicite entre la spécification de l'interface des objets (leurs types) et la spécification de leur réalisation (leurs classes) qui permet la réalisation simultanée d'un même type par plusieurs classes et la mise en œuvre de la notion de vue.

• L'organisation hiérarchique de l'ensemble des types et de l'ensemble des classes qui permet une définition propre et cohérente des propriétés communes à tous les objets.

• L'expression des contraintes d'accès aux objets partagés sous la forme de conditions d'exécution qui s'intègrent bien dans les schémas de définition et d'exécution des objets.

• La répartition du système support qui peut être ou non prise en compte par les programmeurs en fonction des besoins de leurs applications.

• La persistance implicite des objets qui décharge les programmeurs de la prise en compte de leur sauvegarde.

Ces deux derniers points mettent en évidence l'importance des caractéristiques du système d'exploitation sous-jacent. La mise en œuvre du modèle d'objets de Guide sur un système réparti à objets permet en effet d'intégrer naturellement la répartition et la persistance dans le modèle.

L'expérience de programmation a permis d'exploiter et de mettre en valeur les éléments caractéristiques du modèle cités précédemment. Les résultats de cette expérience sont rappelés ci-dessous :

• Le niveau d'abstraction mis en œuvre permet de cacher la réalisation physique des objets : cette transparence permet la modification de la réalisation d'un objet indépendamment de la réalisation des objets qui l'utilisent.

• Le mécanisme d'héritage permet la réutilisation de code existant et la construction d'objets par spécialisation.

• La propriété de conformité entre les interfaces permet de réaliser la notion de vue et d'utiliser indifféremment des objets de différentes classes. Elle permet également de conserver un niveau d'abstraction élevé ce qui favorise l'évolution des applications. Combinée au mécanisme d'héritage, elle permet de remplacer dynamiquement des objets par d'autres plus spécialisés.

• Le contrôle statique de conformité permet d'exploiter la propriété de partage d'interface en toute sécurité.

• Le mécanisme de composition par références permet le partage d'objets notamment entre différentes applications.

• Les conditions d'exécution permettent une spécification claire des contraintes d'accès aux objets partagés.

• La transparence de la répartition permet un découplage entre la structuration des applications et la prise en compte des problèmes liés à la localisation des objets. La répartition des objets sur les différents sites du système peut notamment être commandée indépendamment des applications par des considérations de commodité, de sécurité, de disponibilité ou de performances. La programmation d'applications réparties sur Guide se

fait aussi simplement que la programmation d'applications centralisées : le fait de renvoyer au système d'exécution le travail de localisation et de liaison simplifie le travail de conception et permet notamment la mise au point des applications en environnement centralisé.

La mise en œuvre du modèle Guide sur un système à objets persistants est perçue comme un progrès mais nécessite cependant une attention particulière de la part des programmeurs peu habitués à travailler dans ce type d'environnement : croyant tester une nouvelle version d'une application, certains d'entre eux continuent parfois sans s'en rendre compte à travailler sur les objets persistants créés par une version antérieure. Deux contreparties importantes à cette persistance explicite sont à noter également : la nécessité d'un mécanisme de ramasse-miettes et d'un mécanisme de gestion de versions.

3. PERSPECTIVES

Perspectives à court terme

L'étude bibliographique des modèles de programmation de Trellis/Owl, d'Emerald et d'Eiffel, ainsi que l'expérience de programmation réalisée sur le système Guide a mis en évidence différents aspects du modèle d'objets qui nécessitent un travail d'approfondissement. Les plus importants sont rappelés ci-dessous :

• La définition de l'opération de destruction explicite d'un objet (*destroy*) : il n'est pas cohérent qu'elle soit définie comme une opération de l'objet qui doit être détruit.

• La définition des règles de recherche des blocs de traitement dans le mécanisme de gestion des exceptions : un objet qui a provoqué le déclenchement d'une exception doit pouvoir la traiter lui-même.

• L'expression des conditions d'exécution qui permettent la spécification du contrôle d'accès aux objets partagés : les compteurs de synchronisation sont difficiles à maîtriser et des outils de plus haut niveau pourraient être fournis pour la spécification des schémas de synchronisation classiques tels que le schéma des producteurs/consommateurs ou celui des lecteurs/rédacteurs.

• L'intérêt des notions de classes différées, de déclaration par association, d'héritage caché et d'héritage non conforme doit être évalué précisément lors de la définition du mécanisme d'héritage multiple de Guide.

Le travail de programmation effectué sur le premier prototype a également mis en évidence la nécessité de disposer rapidement dans Guide d'un environnement de développement à la hauteur des possibilités du système. Cet environnement doit fournir notamment des outils d'aide à l'édition (éditeur guidé par la syntaxe, interpréteur de conformité), de gestion de versions (pour la gestion des versions successives des types et des classes en particulier) et de déboguage.

Perspectives à long terme

L'expérience de programmation a mis en valeur l'importance d'une période d'apprentissage aux techniques de la programmation par objets. La tendance naturelle de chacun est en effet de revenir sans cesse à des schémas de programmation connus et les avantages qu'apporte la notion d'objet (abstraction, réutilisabilité) sont très souvent peu ou mal exploités. En conséquence, il me semble essentiel de fournir aux programmeurs qui découvrent la notion d'objet des moyens d'appréhender les techniques de programmation originales qu'elle permet de mettre en œuvre.

Ma deuxième et dernière proposition de travail à long terme concerne le langage de programmation Guide. L'objectif poursuivi pendant la première phase de conception du système était de pouvoir disposer rapidement d'un support d'expérimentation des modèles de programmation et d'exécution. Le langage Guide a été défini et mis en œuvre dans ce but. L'objectif visé a été atteint : tel qu'il est défini à ce jour, le langage permet l'exploitation de tous les éléments définis dans le modèle et la programmation d'applications sur le premier prototype. Je pense cependant qu'il est essentiel que dans la seconde phase de conception qui est en cours, le langage Guide fasse l'objet d'un travail de définition propre et soit notamment complété par une sémantique précise. Il me semble que ce n'est qu'à ce prix qu'on pourra réellement le qualifier de 'langage de programmation'.

ANNEXE I
Eléments de la bibliothèque Guide

Cette annexe n'a pas pour but de donner un aperçu général du contenu de la bibliothèque Guide. Seuls les types et classes dont la description est nécessaire à la compréhension des exemples donnés au chapitre IV sont décrits.

1. LE TYPE *TOP*

Le type Top décrit ci-dessous constitue le sommet de la hiérarchie des types dans Guide : il est ancêtre implicite de tous les autres. Les méthodes qu'il spécifie sont définies sur tous les objets.

```
type Top is

   method Equal (in ref Top) : Boolean ;
      // si l'objet désigné par la référence donnée en paramètre
      // est de la même classe que l'objet courant, et si de plus
      // la valeur de son état est identique à la valeur de l'état
      // de ce dernier, la méthode retourne vrai
      // sinon, elle retourne faux

   method Destroy ;
      // destruction de l'objet courant
      // la valeur de la référence qui le désignait
      // dans l'instruction d'appel devient incohérente
      // au retour de l'appel

   method Size : Integer ;
      // retourne la taille en octets de l'objet courant

   method TypeName : String [20] ;
      // retourne le nom du type de l'objet courant

   method ClassName : String [20] ;
      // retourne le nom de la classe de l'objet courant

end Top.
```

2. LE TYPE *CATAL*

L'objet désigné par la référence constante *catal*, et qui fournit le service de désignation dans le système Guide est du type *Catal* décrit ci-dessous :

```
type Catal is

  method Insert (in name : String [80] ; object : ref Top) ;
    // mémorise sous le nom donnée en premier argument
    // la référence de l'objet donnée en second argument

  method Search (in name : String [80]) : ref Top ;
    // rend une référence sur l'objet catalogué
    // sous le nom donné en premier argument
    // rend 'nil' si le nom donné n'existe pas

  method Delete (in name : String [80]) ;
    // supprime du catalogue le nom donné en argument
    // et la référence associée

end Catal.
```

3. LE TYPE *FILE*

Le type *File* décrit ci-dessous est utilisé pour la déclaration d'objets de communication dans l'application de gestion de bibliothèque décrite au chapitre IV.

```
type File is

  method Writeln ;
    // rajout du caractère fin de ligne

  method WriteString (in String[80]) ;
    // rajout de la chaîne de caractères donnée en paramètre

  method WriteInteger (in Integer) ;
    // rajout de la chaîne de caractères qui représente
    // la valeur de l'entier donné en paramètre

  operator & (in ref File) : ref File ;
    // opérateur de concaténation
    // produit un fichier obtenu par concaténation du fichier
    // courant à celui transmis en paramètre

end File.
```

On peut remarquer ci-dessus la déclaration d'une méthode sous la forme d'un opérateur binaire. Ce type de déclaration signale au compilateur les symboles qu'il doit reconnaître comme étant des opérateurs ; il est réservé à la déclaration des types et des

classes prédéfinies et permet notamment l'appel des méthodes définis sur les objets nombres sous la forme classique d'appel d'opérateurs : <*i* + *j*>, <*i* * *j*>, etc.

4. LE CONSTRUCTEUR DE TYPES ET DE CLASSES *LIST*

Le constructeur de types décrit ci-dessous ainsi que le constructeur de classes qui le réalise (et qui est décrit plus loin) sont utilisés dans l'application de gestion de bibliothèque décrite au chapitre IV pour la définition des bases de références bibliographiques.

```
type constructor List OF [Item : Top] is

   method First ;
      // positionne le pointeur courant sur le premier élément
      // de la liste
   method Next : Item ;
      // retourne l'élément désigné par le pointeur courant et
      // incrémente ce dernier - si avant incrémentation le
      // pointeur courant désignait le dernier élément de la liste,
      // tout nouvel appel de Next retourne nil et est sans effet
      // sur la liste jusqu'à ce que le pointeur soit repositionné
      // en début de liste par un appel à l'opération First -
      // sur une liste vide, un appel de Next rend nil et est sans
      // effet
   method Append (in Item) ;
      // insère le nouvel élément en fin de liste sans modifier le
      // pointeur courant - sans effet si la capacité maximale de
la
      // liste est atteinte
   method Remove : Item ;
      // retourne l'élément désigné par le pointeur courant
      // et l'extrait de la liste - si cet élément était le dernier
      // de la liste, le pointeur courant désigne après l'appel
      // le nouvel élément de fin de liste -
      // l'opération Remove retourne nil et est sans effet dans les
      // mêmes cas que l'opération Next
end List.

-------

class constructor List OF [Item : Top]
implements List is

const MaxItems : Integer = 100 ;
contents : Array [MaxItems] OF Item;
ptr, nbItems : Integer = 0, 0;

method First ;
   begin
   ptr := 0;
   end First ;

method Next : Item ;
   begin
   if (nbItems = 0) or (ptr >= nbItems) then
```

```
        return nil
      else
        ptr := ptr + 1 ;
        return contents[ptr-1] ;
      end ;
      end Next ;

    method Append (in newItem : Item) ;
      begin
      if (nbItems # MaxItems) then
          contents[nbItems] := newItem ;
          nbItems := nbItems + 1 ;
      end ;
      end Append ;

    method Remove : Item ;
      i : Integer ;
      result : Item ;
      begin
      if (nbItems = 0) or (ptr >= nbItems) then
        return nil
      else
        result := contents[ptr] ;
        if (ptr = nbItems-1) then
          ptr := ptr - 1 ;
        else
          for i := ptr to (nbItems-2) do
          contents[i] := contents[i+1] end ;
        end ;
        nbItems := nbItems - 1 ;
        return result ;
      end ;
      end Remove ;

    end List.
```

ANNEXE II
Compléments sur les objets synchronisés

La première partie de cette annexe donne quelques exemples d'utilisation des conditions d'exécution pour l'expression du contrôle d'accès aux objets partagés. La seconde partie décrit le principe de la mise en œuvre du contrôle.

1. EXEMPLES DE CLASSES SYNCHRONISÉES

Les trois exemples présentés dans cette partie ont été réalisés et testés sur le premier prototype du système Guide.

Le constructeur Buffer

Le constructeur de classe *Buffer* décrit ci-dessous permet de créer des objets boîtes pouvant contenir k éléments, et dont le contenu est géré selon la méthode FIFO.

```
class constructor Buffer [size] OF [Element : Top]
implements Buffer OF [Element : Top] is

contents : Array [size] OF Element;
first, last : Integer = 0, 0;

method Put (in m: Element);
begin
  contents[last] := m;
  last := last+1 mod size;
end Put;
method Get (out m: Element);
begin
  m := buffer[first];
  first := (first + 1) mod size;
end Get;

control
  Put  :
    current(Put)=0 and (completed(Put)-completed(Get)) < size
  Get  :
    current(Get)=0 and (completed(Put) > completed(Get))

end Buffer.
```

Pour garantir l'intégrité de l'état d'un objet boîte, les méthodes *Put* et *Get* sont respectivement exclusives avec elles-mêmes. La seconde partie des conditions d'exécution spécifie qu'un nouvel élément peut être mis dans la boîte si elle n'est pas pleine, et qu'un élément peut en être extrait si elle n'est pas vide.

La classe RWro_File

La classe *RWro_File* permet de créer des objets fichiers pour lesquels les demandes de lecture et d'écriture sont servies par ordre d'arrivée ('request-order'). Ce schéma de synchronisation nécessite la définition sur ces objets de méthodes destinées uniquement à l'ordonnancement des requètes. Les objets fichiers de classe *RWro_File* doivent cependant pouvoir être utilisés de la même façon que tout autre objet fichier. Leur type est donc construit par sous-typage du type *File* et leur classe par héritage de la classe de même nom.

Je rappelle ci-dessous la déclaration du type *File* et de la classe *File* :

```
type File is
  method read ...;
  method write ...;
end File.

class File
implements File is
  <déclaration des variables d'état>
  method read ...;
    <code de la méthode read>
  method write ...;
    <code de la méthode write>
end File.
```

Le type *RWro_File* et la classe qui le réalise sont décrits ci-dessous :

```
type RWro_File subtype of File is
  // method read ...;
  // method write ...;
  method GetOrder (in lect : Boolean) : Integer;
    // méthode qui permet d'ordonner les requètes
  method AskRead (in request-order : Integer);
    // méthode qui permet de bloquer les activités
    // qui demandent une lecture
  method AskWrite (in request-order : Integer);
    // méthode qui permet de bloquer les activités
    // qui demandent une écriture
end RWro_File.
```

```
class RWro_File subclass of File
implements RWro_File is

Nblec, Nbecr : Integer = 0,0 ;

method  GetOrder (in lect : Boolean) : Integer;
// cette méthode permet d'ordonner les requètes
// elle attribue un numéro d'ordre à chacune en considérant
// le fait que plusieurs lectures peuvent être simultanées
result : Integer;
begin
  if lect then
    // demande de lecture
    result := Nbecr;
    Nblec := Nblec + 1;
  else
    // demande d'écriture
    result := Nbecr + Nblec;
    Nbecr := Nbecr + 1;
  end ;
  return result;
end GetOrder ;

method  AskRead (in request-order : Integer);
// méthode 'bidon' dont la condition d'exécution
// permet de bloquer les activités en demande de lecture
begin
end AskRead;
method  AskWrite (in request-order : Integer);
// méthode 'bidon' dont la condition d'exécution
// permet de bloquer les activités en demande d'écriture
begin
end AskWrite;

method  Read (...);
// redéfinition de la méthode Read héritée
// pour la prise en compte du nouveau schéma de synchronisation
begin
  // l'activité appelante 'prend place' dans la file d'attente
  // et se bloque sur la condition de la méthode AskRead
  self.AskRead (self.GetOrder(true)); // (1)
  super'Read(...);
end Read;

method  Write (...);
// redéfinition de la méthode Write héritée
// pour la prise en compte du nouveau schéma de synchronisation
begin
  // l'activité appelante 'prend place' dans la file d'attente
  // et se bloque sur la condition de la méthode AskWrite
  self.AskWrite (self.GetOrder(false)); // (2)
  super'Write (...) ;
end Write;

control

  GetOrder  : current(GetOrder) = 0 ;
  AskRead   : completed(Write) = request-order ;
  AskWrite  : completed(Write)+completed(Read) = request-order;

end RWro_File.
```

Les conditions d'exécution associées aux méthodes *GetOrder*, *AskRead* et *AskWrite* spécifient que :

• La méthode *GetOrder* est exclusive avec elle-même : cette condition est nécessaire pour garantir la validité de l'ordonnancement des requètes.

• La méthode *AskRead* ne peut être exécutée que si la valeur de son argument est supérieure au nombre total d'écriture déjà effectuées sur l'objet. Pour respecter le schéma de synchronisation choisi, la valeur de l'argument doit correspondre au nombre de demandes d'écriture qui ont précédées la demande de lecture en cours de traitement : c'est justement cette valeur que rend la méthode *GetOrder* appelée pour le calcul de l'argument (cf instruction (1)).

• La méthode *AskWrite* ne peut être exécutée que si la valeur de son argument est supérieure au nombre total d'écriture et de lecture déjà effectuées sur l'objet. Pour respecter le schéma de synchronisation choisi, la valeur de l'argument doit correspondre au nombre de demandes d'écriture et de lecture qui ont précédées la demande d'écriture en cours de traitement : c'est justement cette valeur que rend la méthode *GetOrder* appelée pour le calcul de l'argument (cf instruction (2)).

Les philosophes aux spaghetti

Le cas des philosophes est spécifié dans [Dijkstra 71]. Avant de décrire les types et les classes mis en œuvre, je rappelle les données du problème.

Un certain nombre de philosophes sont conviés à un repas. Le plat est unique, il s'agit de spaghetti. Chacun ne trouve malheureusement qu'une seule fourchette à la place qui lui est affectée et doit donc emprunter une fourchette à un des ses voisins pour pouvoir manger. D'un commun accord, les philosophes décident de passer chacun une partie du temps à penser, de façon à laisser la possibilité de manger à ses voisins. Le problème consiste à représenter le dîner par un ensemble d'objets en prennant garde à ce que chaque convive puisse manger et penser à son rythme sans risque d'interblocage.

Trois types d'objets sont mis en œuvre :

• chaque philosophe est représenté par un objet de type *Philosopher*,

```
type Philosopher is
  method Init (in tableId : ref ForkAllocator; placeId :
Integer);
  method Active;
end  Philosopher.
```

• la table est représentée par un objet de type *ForkAllocator*,

```
type constructor ForkAllocator is
  method GetForks (in philosopherName : Integer);
  method FreeForks (in philosopherName : Integer);
  method Init;
end ForkAllocator.
```

• le diner est mis en œuvre par un objet de type *Application.*

```
type Application is
  method Init ;
end Application.
```

Une table peut être vue comme un allocateur de fourchettes. Elle possède autant de fourchettes que de convives et deux opérations *GetForks* et *FreeForks* permettant à ces derniers de prendre les deux fourchettes qui leur sont nécessaires et de les rendre. La condition d'exécution associée à l'opération *GetForks* constitue la garantie anti-interblocage de l'application : elle spécifie que les fourchettes ne peuvent être prises que par couple et évite ainsi que plusieurs convives se retrouvent avec une seule fourchette et attendent indéfiniment la seconde.

```
class constructor ForkAllocator[philosopherNbr]
implements ForkAllocator is

forks : Array [philosopherNbr] of Boolean;

method GetForks (in philosopherName : Integer);
begin
  forks[(philosopherName + 1) Mod philosopherNbr] := false ;
  forks[philosopherName] := false ;
end GetForks ;

method FreeForks (in philosopherName : Integer);
begin
  forks[(philosopherName + 1) Mod philosopherNbr] := true ;
  forks[philosopherName] := true ;
end FreeForks ;

method Init;
i : Integer;
begin
  for i := 1 to philosopherNbr do
    forks[philosopherNbr - 1] := true ;
  end ;
end Init;

control

  GetForks :
       (current(GetForks) = 0) And (current(FreeForks) = 0)
       And forks[philosopherName]
       And forks[(philosopherName + 1) Mod philosopherNbr]
  FreeForks:
       (current(GetForks) = 0) And (current(FreeForks) = 0);

end ForkAllocator.
```

A chaque philosophe est attribuée une place à table : l'objet philosophe possède une variable d'état qui désigne l'objet qui représente la table, et une autre qui contient le numéro de sa place à cette table. Au cours du repas le philosophe pense et mange alternativement. Lorsqu'il souhaite manger, il requiert l'accès aux fourchettes qui lui sont nécessaires par un appel à l'objet table auquel il transmet l'identification de sa place.

```
class Philosopher
implements Philosopher is

table  : ref ForkAllocator;
place  : Integer;

method Init (in tableId : ref ForkAllocator; placeId : Integer)
;
begin
   table   := tableId ;
   place   := placeId ;
   mind    := Clock.new ;
   stomach := Clock.new ;
end Init;

method Active;
begin
   while true do
        // le philosophe pense
      ...
        // il souhaite manger et attend ses fourchettes
      table.GetForks(place);
        // il mange
      ...
        // il arrête de manger et se remet à penser
      table.FreeForks(place);
   end ;
end Active;

end  Philosopher.
```

Le diner est mis en œuvre par un objet de type Application et de classe *Diner [k]*, où *k* représente le nombre de convives. L'opération *Init* définie sur cet objet constitue le programme principal de l'application et consiste en la création d'un objet table, d'un objet allocateur de places, et de *k* objets philosophes, en l'initialisation de ces objets si besoin, puis en l'invitation simultanée des philosophes à commencer le repas. Cette invitation est matérialisée par un bloc parallèle.

```
class constructor Diner[philosopherNbr]
implements Application is

method Init ;
table  : ref ForkAllocator;
place-alloc : ref PlaceAllocator;
guest : Array [philosopherNbr] of ref Philosopher;

i : Integer;
begin
   table := ForkAllocator[philosopherNbr].new ;
```

```
    table.Init ;

    place-alloc := PlaceAllocator.new ;

    for i := 0 to philosopherNbr do
      guest[i] := Philosopher.new;
      guest[i].Init (table, place-alloc.GetPlace);
    end ;

    co_begin
      guest-1 : guest[0].Active;
      guest-2 : guest[1].Active;
        ...
      guest-k : guest[philosopherNbr].Active;
    co_end ;

end Init;

end Diner.

-------

class PlaceAllocator ... is

place : Integer = -1;

method GetPlace : Integer ;
begin
  place := place + 1 ;
  return place ;
end Getplace ;

end PlaceAllocator.
```

Je n'ai pas pris en compte ici le problème de la terminaison du repas. Sa résolution nécessiterait la mise en œuvre d'un protocole d'accord entre les différents philosophes.

2. Mise en œuvre des fonctions de synchronisation

On distingue dans Guide les objets synchronisés de ceux qui ne le sont pas. Les premiers sont les exemplaires des classes qui associent des conditions d'exécution aux méthodes qu'elles définissent. La prise en compte de ces conditions nécessite le complément de l'état des exemplaires de la classe par un ensemble de données (compteurs, liste des activités bloquées) et la mise en œuvre d'un protocole particulier pour l'exécution d'une opération sur ces exemplaires (mise à jour des compteurs, test de la condition d'exécution).

Remarque. Un objet synchronisé peut être utilisé sans jamais être partagé : le contrôle d'accès mis en œuvre par le système apparaît alors comme une charge inutile. De façon à permetttre une utilisation rationnelle des objets synchronisés non partagés, le

contrôle d'accès n'est pas mis en œuvre si le programmeur le demande explicitement à leur création :

```
anNonSharedSynchonizedObject := aClass.new (not_shared) ;
```

Le principe d'exécution d'une opération sur un objet a été présenté au chapitre V. Il est basé sur l'organisation des éléments qui constituent l'état des objets classes que je rappelle dans la figure 1 donnée plus loin.

Les fonctions de synchronisation sont générées par le compilateur sur la base des conditions d'exécution définies dans le corps de la classe. Elles effectuent la mise à jour des compteurs et permettent l'évaluation par le système des conditions d'exécution. La forme générale d'une fonction de synchronisation est décrite ci-dessous :

```
f_synchro (in state : Integer) : Boolean ;
// fonction de synchronisation associée à une opération op
// dont la condition d'exécution est cond_op
begin
  case state of
    0 :
      // première évaluation de la condition [1]
      pending(op) := pending(op) + 1 ;
      if cond_op then
        pending(op) := pending(op) - 1 ;
        current(op) := current(op) + 1 ;
        return true ;
      else
        return false ;
      end ;
    end ;
    1 :
      // réévaluation de la condition
      if cond_op then
        pending(op) := pending(op) - 1 ;
        current(op) := current(op) + 1 ;
        return true ;
      else
        return false ;
      end ;
    end ;
    2 :
      // épilogue
      current(op) := current(op) - 1 ;
      completed(op) := completed(op) + 1 ;
    end ;
  end ;

end f_synchro ;
```

[1] L'activité courante étant considérée comme potentiellement bloquée dans l'attente de l'autorisation d'exécuter l'opération, la mise à jour du compteur pending est nécessaire avant la première évaluation de la condition.

Lorsqu'un appel d'opération est fait sur un objet synchronisé, le système met en œuvre, par le biais de la primitive *GuideCall*, le protocole de contrôle d'accès décrit par la séquence de code suivante :

```
if f_synchro(0) then
  <exécution de l'opération>
else
  <blocage de l'activité courante>
  while not f_synchro(1) do <blocage de l'activité courante> ;
  <exécution de l'opération>
end ;
f_synchro(2) ;
<réveil des activités bloquées sur l'objet>
```

Les activités bloquées sur un objet sont celles qui ont demandé l'exécution d'une opération sur cet objet et qui attendent que cette condition soit vérifiée. A chaque objet synchronisé est affecté une liste des activités bloquées sur lui. Le blocage de l'activité courante consiste à insérer l'activité dans la liste des activités bloquées sur l'objet et à suspendre son exécution (ce schéma d'exécution est similaire à celui proposé par Hoare dans [Hoare 74] pour la mise en œuvre des moniteurs).

L'exécution d'une activité suspendue ne peut être reprise qu'après un réveil explicite. Lorsqu'une activité termine l'exécution d'une opération sur un objet synchronisé, toutes les activités bloquées sur lui sont réveillées.

La recherche de l'adresse du code d'une fonction de synchronisation dans l'état d'un objet classe est faite selon le même principe que la recherche de l'adresse du code d'une méthode (cf V).

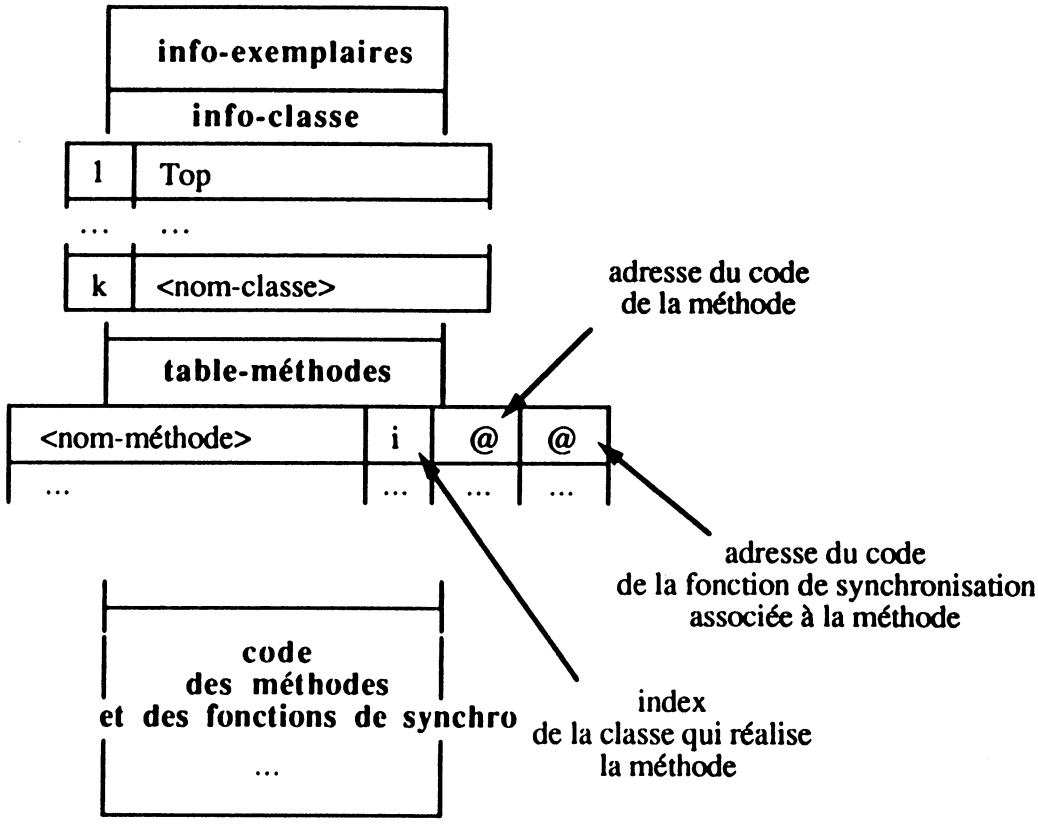


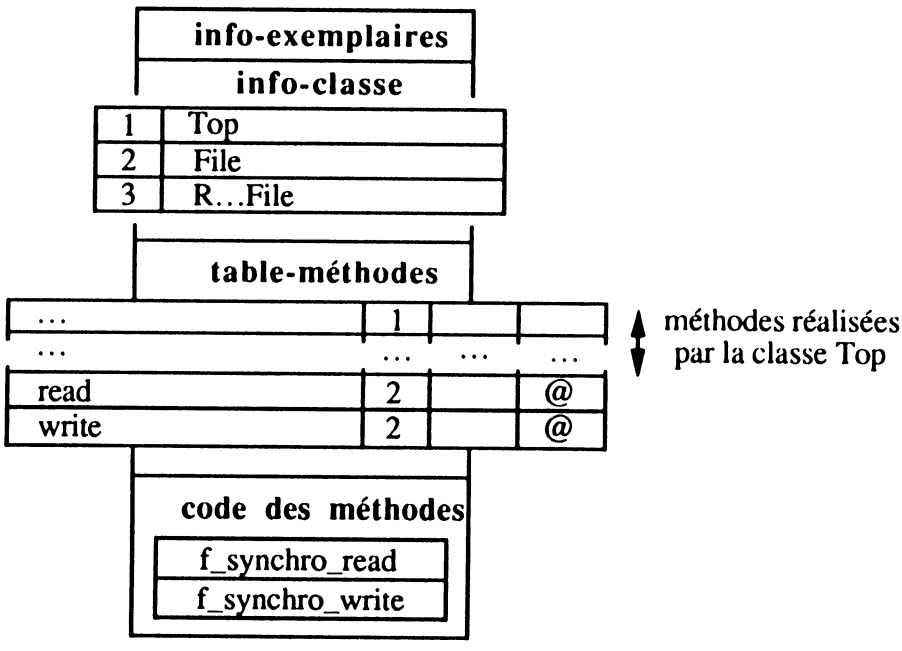
Figure 1. Etat d'un objet classe

On peut cependant remarquer que lors de la construction d'une classe par héritage, la possibilité de redéfinition des conditions d'exécution attachées aux opérations indépendamment de la redéfinition des opérations elles-mêmes, peut conduire à des cas où la fonction de synchronisation associée à une opération et l'opération elle-même sont réalisées par des classes différentes. C'est le cas par exemple des classes *RW_File*, *RpW_File* et *RWp_File* définies en 11.2 au chapitre III.

Ces trois classes sont construites par héritage de la classe *File* et spécifient chacune un schéma de synchronisation d'accès différents pour les objets fichiers qu'elles permettent de créer. Pour ce faire, elles complètent simplement la définition de la classe *File* par un jeu de conditions d'exécution :

```
class File
implements File is
...
method read ... ;
begin
   <code de la méthode read>
end read ;
method write ... ;
begin
   <code de la méthode write>
end write ;

end File.

-----------------------------

class R...File subclass of File
implements File is

control
   read : ... ;
   write : ... ;
end R...File.
```

Chacune de ses trois classes à un état conforme à celui décrit sur la figure suivante.



ANNEXE III
Intégration de logiciels existants

Cette annexe décrit les possibilités d'intégration dans des applications Guide de logiciels écrits dans d'autres langages. Elle est basée sur une comparaison entre les possibilités fournies dans l'environnement de développement d'Eiffel et celles fournies dans Guide et peut être dans ce sens considérée comme un appendice au chapitre III.

Le langage Eiffel est supporté par Unix, alors que le langage Guide est mis en œuvre sur un système d'exploitation propre basé sur la gestion d'objets. Les possibilités d'intégration de logiciels existants sont en conséquence très différentes dans les deux langages et cette différence permet de mettre en évidence et de justifier la spécificité des solutions envisageables dans Guide.

Une synthèse est donnée en conclusion.

1. EIFFEL

L'intégration de code écrits dans d'autres langages est faite par encapsulation de ce code dans une procédure qualifiée par le mot clé *external*. Une telle procédure est toujours locale à une opération définie sur un objet et après la compilation son code reste externe au code Eiffel. Cette technique assure une interface claire entre le monde Eiffel et les autres mondes.

La classe *Array [T]*, élément de la bibliothèque déjà partiellement décrit dans les parties précédentes, fait appel à des fonctionnalités du langage C. Son code est donné ci-dessous.

```
class Array [T] export
  lower, size, upper, entry, enter
feature

  lower, size, upper : INTEGER ;
  area : INTEGER ;

  Create (min, max : INTEGER) is
```

```
      external
        allocate (n : INTEGER) : INTEGER
        name "allocate" language "C" ;
                -- réserve une zone mémoire de n entiers
                -- retourne son adresse ou 0 si problème
      do
        lower := min ;
        upper := max ;
        if min <= max then
          size := max - min + 1 ;
          area := allocate (size)
        else
          size := 0
        end ;
    end ; -- Create

    entry (i : INTEGER) : T is
      require
        lower <= i ; i <= upper ; area > 0 ;
      external
        dynget (ad, i : INTEGER) : T
        name "dynget" language "C" ;
                -- valeur du i-ème élément
                -- dans la zone d'adresse ad
      do
        Result := dynget (area, i)
    end ; -- entry

    enter (i : INTEGER, value : T) is
      require
        lower <= i ; i <= upper ; area > 0 ;
      external
        dynput (ad, i : INTEGER, val : T)
        name "dynput" language "C" ;
                -- met à jour la valeur du i-ème élément
                -- dans la zone d'adresse ad
      do
        dynput (area, i, value)
    end ; -- enter

  invariant
    size = upper - lower + 1 ;
    size >= 0 ;

  end -- class Array [T]
```

L'accès à des fonctions écrites en C est requis ici dans le but d'obtenir une implémentation efficace de la classe *Array*.

Un autre exemple est la programmation de fonctions mathématiques :

```
sqrt (x : Real) : Real is
        -- racine carrée de x
  require
    x >= 0 ;
  external
    c_sqrt (x : Real) : Real
    name "c_sqrt" language "C" ;
```

```
  do
    Result := c_sqrt (x)
end ; -- enter
```

L'ensemble des langages connus d'Eiffel est un paramètre d'implémentation.

Dans certains cas, une application peut avoir besoin d'accéder à un objet dans une séquence de code écrite dans un langage autre que Eiffel. La conception d'une interface avec un système de gestion de bases de données pour le stockage et la récupération d'objets en est un bon exemple. La classe prédéfinie *Internal* est fournie par la bibliothèque d'Eiffel à cet effet.

La classe *Internal* définit un ensemble d'opérations permettant de récupérer le nom de la classe d'un objet, le nombre de ses arguments, le nom de la classe et la valeur de chacun de ses arguments, etc. La classe des objets sur lesquels de telles fonctionnalités sont requises, doit être déclarée sous-classe de *Internal* . Les informations relatives à la structure interne et à l'état d'un exemplaire d'une telle classe sont alors accessibles et peuvent être transmises à des procédures externes.

Un extrait de la classe *Internal* est donné ci-dessous :

```
class Internal export
  class_name, field_nb, field_name, ...
feature

  class_name (obj : INTERNAL) : STRING
       -- nom de la classe de l'objet désigné par obj
    require
      not obj.Void  -- obj doit être initialisée

  field_nb (obj : INTERNAL) : INTEGER
       -- nombre d'arguments de l'objet désigné par obj
    require
      not obj.Void

  field_name (i : INTEGER, obj : INTERNAL) : STRING
       -- nom du i-ème argument de l'objet désigné par obj
    require
      not obj.Void ;
      1 <= i ; i <= field_nb (obj) ;
  ...

end ; -- class Internal
```

Les opérations fournies par la classe *Internal* rendent possible l'écriture d'applications qui requièrent l'accès à la structure interne d'un objet. Plusieurs outils intégrés à l'environnement Eiffel utilisent ces fonctionnalités. On peut citer en particulier

les outils de débogue qui permettent de visualiser l'état d'un objet pendant l'exécution d'une application, les outils qui permettent la recopie de l'état d'un objet dans un fichier Unix (classe *Storable* - cf 6.1) et le ramasse-miettes.

2. GUIDE

Différentes possibilités sont envisageables dans Guide pour l'intégration de logiciels existants (celles décrites ici sont basées sur une implémentation du système Guide sur Unix).

Une séquence de code, écrite dans un langage différent de Guide, est suceptible d'être intégrée dans un programme Guide si elle est exécutable sur Unix. Elle peut être intégrée de différentes manières en fonction notamment de sa nature : application ou fonction.

Intégration d'une application

L'intégration d'une application externe dans Guide consiste à provoquer la création d'un processus Unix pour l'exécution de cette dernière. Un tel processus est créé par l'appel d'une primitive du noyau Guide qui permet de transmettre une commande au système Unix sous-jacent. En l'occurrence la commande transmise est la commande *system* d'Unix dont le paramètre indique le nom et éventuellement les arguments de l'application externe à exécuter.

Cette solution permet une intégration rapide d'applications indépendantes de Guide et dont notamment les sources ne sont pas disponibles. Elle peut permettre par exemple de lancer la commande *mail* d'Unix depuis une application Guide.

Une solution différente doit être envisagée pour l'intégration des applications dont on veut pouvoir appliquer les fonctionnalités à des objets Guide. Dans ce cas, l'application doit pouvoir accéder à ces derniers. La méthode d'intégration proposée est similaire à la précédente mais suppose une modification du code de l'application : le processus Unix créé par la commande *system* doit être reconnu comme une activité par le système Guide de façon à avoir accès au système de gestion d'objets de ce dernier. Une telle solution n'est donc envisageable que si les sources de l'application à intégrer sont disponibles.

Cette deuxième solution a été retenue pour une première intégration du logiciel d'édition Grif [Quint 86], utilisé sous la forme d'une application externe pour la

réalisation des opérations d'édition (édition, impression, import, export) définies sur les objets documents. Lors de l'exécution de Grif sur un objet document de Guide, la cohérence entre l'état de cet objet et les données qui le représente dans le code de l'application externe est maintenue par cette dernière dont la version originale a été modifiée en conséquence.

Intégration d'une fonction

La solution proposée pour l'intégration d'une fonction externe consiste à la définir sous la forme d'une primitive du noyau Guide. C'est notamment la solution retenue pour la définition des objets chaînes de caractères pour lesquels les opérations de comparaison et de recopie sont réalisées par appel des primitives corrrespondantes d'Unix. Toute fonction externe intégrée par le biais d'une primitive du noyau Guide peut ainsi être rendue accessible par encapsulation dans une opération définie dans une classe. Cette encapsulation permet d'offrir aux programmeurs une interface conforme au modèle de programmation par objets préconisé dans Guide.

Le logiciel de gestion de fenêtres X-Window [Scheifler 86] peut être intégré dans Guide selon ce principe. Il est en effet conçu sous la forme d'une bibliothèque. Un premier prototype de X dans Guide a ainsi été réalisé par intégration d'un sous-ensemble des fonctionnalités de la version 10.

La définition des fonctions externes sous la forme de primitives du noyau Guide est directement applicable à l'intégration de toute bibliothèque Unix dont l'interface est connue. Elle doit cependant être réservée à un usage limité. La taille trop importante de la bibliothèque provoquerait sinon des pertes non acceptables de place et d'efficacité.

Plutôt que de définir une fonction externe sous la forme d'une primitive du noyau Guide, son code peut aussi être directement utilisé pour l'écriture d'une opération déclarée dans le corps d'une classe Guide. Cette solution est intéressante car elle représente une intégration réelle, mais sa mise en œuvre est longue. Le code intégré doit en effet être entièrement adapté au support d'exécution sous-jacent qui n'est plus Unix mais Guide, notamment en ce qui concerne les éventuels appels à la bibliothèque d'Unix.

Cette deuxième solution n'est envisageable que dans la mesure où les sources de la fonction à intégrer sont disponibles. Elle est actuellement utilisée pour l'intégration dans Guide des fonctionnalités de la boîte à outils de la version 11 du logiciel X-Window. Les

fonctions de base de cette nouvelle version ont été intégrées sous la forme de primitives du noyau comme cela avait été fait pour l'intégration de la version précédente.

Le schéma suivant présente d'une manière synthétique les différentes possibilités d'intégration de logiciels dans Guide, en fonction des critères à prendre en compte pour le choix de l'une ou l'autre des solutions (chaque solution est numérotée en fonction de son ordre de présentation).

conditions requises	**intégration d'une application**	**intégration d'une fonction**	conditions requises
	solution - 1	solution - 3	
application qui ne requiert pas l'accès à des objets	exécution par un processus non reconnu par Guide	définition sous la forme d'une primitive du système	binaire de petite taille
	solution - 2	solution - 4	
sources disponibles	exécution par une activité Guide	intégration dans le corps d'une méthode	sources disponibles

3. SYNTHESE

L'intégration dans une application Guide de code écrit dans un autre langage ne peut être faite de la même façon que dans Eiffel pour les deux raisons suivantes :

• Une application Eiffel s'exécute dans un unique espace d'adressage ce qui permet une gestion dynamique de la mémoire par le biais d'outils indépendants des mécanismes de gestion d'objets (cf description de la classe *Array*). Une telle possibilité n'est pas envisageable dans le système Guide où toute application est potentiellement répartie et parallèle et tout objet potentiellement partageable par différentes activités.

• Eiffel est supporté par Unix et peut en conséquence offrir la visibilité des fichiers gérés par ce système (cf possibilité de recopie de l'état d'un objet dans un fichier et de mise à jour de l'état d'un objet à partir du contenu d'un fichier). Bien que le système Guide soit bâti sur Unix, il ne gère que des objets et ne connaît pas la notion de fichier ; le langage Guide ne peut donc raisonnablement offrir la visibilité du système de fichiers d'Unix aux programmeurs d'applications.

En conséquence, l'intégration dans Guide d'un logiciel externe doit toujours être faite avec précaution : une application ou une fonction conçue pour être exécutée sur Unix, peut faire appel aux primitives de gestion de fichiers, de gestion dynamique de mémoire ou de gestion dynamique de processus fournies par ce dernier. L'utilisation de ces primitives est faite en fonction des caractéristiques propres à l'organisation et au mode de fonctionnement du système Unix, qui sur ce plan est fondamentalement différent de Guide. Un logiciel construit sur ces bases ne peut donc être intégré dans Guide selon un schéma simple et systématique.

Cette raison explique le fait que le premier prototype du système n'offre pas d'outils qui permettent aux programmeurs l'intégration de code écrit dans des langages différents du langage Guide. La conception de ce type d'outils est cependant envisagée et doit être entreprise sur la base des expériences d'intégration réalisées.

BIBLIOGRAPHIE

[Almes 85] Almes G., Black A., Lazowska E., Noe J., *The Eden System: A Technical Review*, IEEE Trans. Softw. Eng. SE-11, n° 1, pp. 43-58, janvier 1985.

[Almes 84] Almes G., Black A., Bunje C. Wiebe D., *Edmas : A Locally Distributed Mail System*, Proc. of the 7th Int. Conf. en Software Eng., Orlando, march 1984.

[Andrews 82] Andrews G.R., *The distributed Programming Language SR Mechanisms, Design and Implementation*, Software - Practice and Experience, 12, pp 719-753, 1982.

[Behm 87] Behm P., *La paramétrisation en programmation*, Rapport Interne Bull, DSG/CRG/87017, juin 1987.

[Bert 83] Bert D., *Manuel de référence du langage LPG*, Rapport R-408, v 1.2, IMAG, Grenoble, décembre 1983.

[Black 85] Black A., *Supporting Distributed Applications: Experience with Eden*, Proc. of 10th ACM SIGOPS, Washington, dec 1985.

[Black 86a] Black A., Hutchinson N., *Object Structure in the Emerald System*, OOPSLA'86, Conference Proceedings, pp. 78-86.

[Black 86b] Black A., Hutchinson N., Jul E., Levy H., Carter L., *Distribution and Abstract Types in Emerald*, IEEE Trans. Softw. Eng., SE-12, december 1986.

[Cardelli 85] Cardelli L., Wegner P., *On Understanding Types, Data Abstraction and Polymorphism*, Computing Surveys, vol 17-4, pp 471-522, december 1985.

[Decouchant 88] Decouchant D., Duda A., Freyssinet A., Paire E., Riveill M., Rousset de Pina X., Vandôme G., *Guide: an implementation of the Comandos object-oriented architecture on Unix*, Proceedings EUUG Conference, Lisbonne, october 1988.

[Dijkstra 71] Dijkstra E.W., *Hierarchical ordering of sequential processes*, Acta Informatica 1,2, 1971.

[Freyssinet 87] Freyssinet A., *Désignation dans un système réparti, Application au système Guide*, Rapport de DEA, Grenoble, juin 87.

[Goldberg 83] Goldberg A., *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading (Mass.), 1983.

[Goldberg 85] Goldberg A., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading (Mass.), 1985.

[Hoare 74] Hoare C.A.R., *Monitors: an Operating System Structuring Concept*, Comm. ACM 17, 10, pp 549-557, october 1974.

[Ichbiah 83] *Reference Manual for the Ada Programming Language*, U.S. Department of Defense, Washington 1983.

[Jamrozik 89] Jamrozik H., *Gestion des types et des classes dans un système d'exploitation à objets, Application au système Guide*, Rapport de DEA, Grenoble, à paraître (1989).

[Johnson 86] Johnson R.E., *Type-Checking Smalltalk*, Conference Proceedings OOPSLA'86, Portland, pp 315-321.

[Lécluse 88] Lécluse C., Richard P., Velez F., *O2, Un Modèle de Données Orienté-Objet*, IVèmes journées BD avancées, Benodet, pp 75-88, mai 1988.

[Liskov 74] Liskov B., Zilles S., *Programming with Abstract Data Types*, Computation Structures Group. Memo n°99, MIT, Project MAC, Cambridge (Mass.), 1974. (SIGPLAN Notices, 9-4, pp 50-59, april 1974)

[Liskov 79] Liskov B., Snyder A., *Exception handling in CLU*, IEEE Trans. on Software Engineering, SE-5,6, november 1979.

[Liskov 83] Liskov B., Scheifler R., *Guardians and Actions: Linguictic Support for Robust, Distributed Programs*, ACM Trans. on Prog. Lang. and Syst., 5 - 3, pp 381-404, july 1983.

[Liskov 85] Liskov B., *The Argus language and system*, Lecture Notes in Computer Science, 190, Springer-Verlag, 1985.

[Lunati 88] Lunati J.M., *Migration de processus et partage de charge dans les systèmes répartis, Application au système Guide*, Rapport de DEA, Grenoble, septembre 1988.

[Meyer 88] Meyer B., *Object-oriented Software Construction*, Prentice Hall, 1988.

[News 1988] Discussion sur *comp.lang.eiffel*, décembre 1988.

[NGuyen Van 89] NGuyen Van H., *Propositions pour le ramasse-miettes de Guide*, Rapport de recherche Guide, Bull-LGI Grenoble, à paraître (1989).

[O'Brien 85a] O'Brien P., *Trellis Object-Based Environment, Language Tutorial*, DEC-TR-373, v1.1, November 1985.

[O'Brien 85b] O'Brien P., *Trellis Object-Based Environment, Language Reference Manual*, DEC-TR-372, v1.1, November 1985.

[O'Brien 87] O'Brien P., Halbert D., Kilian M., *The Trellis Programming Environment*, DEC-TR-503, April 1987.

[O'Brien 86] O'Brien P., Bullis B., Schaffert C., *Persistent and Shared Objects in Trellis/Owl*, Int. Workshop on Object-Oriented Database Systems, Asilomar Conference Center, Pacific Grove California, September 1986.

[O'Brien 88] O'Brien P., *Common Object-Oriented Repository System*, Lecture Notes in Computer Science, 334, pp 329-333, 1988.

[Quint 86] Quint V., Vatton I., *Grif : an Interactive System for Structured Document Manipulation, Text Processing and document Manipulation*, Proceedings of the International Conference (ed. J.C. van Vliet), Cambridge University Press, pp 200-213, 1986.

[Rovner 86] Rovner P., *Extending Modula-2 to build large, integrated systems*, IEEE Software, 1986.

[Schaffert 85] Schaffert C., Cooper T., Carrie W., *Trellis Object-Based Environment, Language Reference Manual*, DEC-TR-372, v1.1, November 1985.

[Scheifler 86] Scheifler R.W., Gettys J., *The X Window System*, ACM Trans. on Graphics, 5, 2, pp 79-109, april 1986.

[Scioville 89] Scioville R., *Gestion des informations persistantes dans un système réparti à objets*, Thèse de Docteur de l'Université J. Fourier de Grenoble, à paraître (1989).

[Stroustrup 86] Stroustrup B., *The C++ programming langage*, Addison Wesley, march 1986.

[Vandôme 88] Vandôme G., *Le service de désignation du système Guide*, Rapport de recherche Guide, Bull-LGI Grenoble, à paraître (1989).

Rapports Guide

[Guide R1] Balter R., Krakowiak S., Meysembourg M., Roisin C., Rousset de Pina X., Scioville R., Vandôme G., *Principes de conception du système d'exploitation réparti Guide*, Rapport de recherche Guide n°1, Bull-LGI Grenoble, avril 1987. (Version préliminaire dans BIGRE+Globule, n° 52, décembre 1986).

[Guide R2] Krakowiak S., Meysembourg M., Riveill M., Roisin C., *Modèle d'objets et langage du système Guide*, Rapport de recherche Guide n°2, Bull-LGI Grenoble, novembre 1987.

[Guide R3] Balter R., Bernadat J., Decouchant D., Krakowiak S., Riveill M., Rousset de Pina X., *Modèle d'exécution du système Guide*, Rapport de recherche Guide n°3, Bull-LGI Grenoble, décembre 1987.

[Guide R4] Freyssinet A., Scioville R., Vandôme G., *Gestion des objets dans le système Guide*, Rapport de recherche Guide n°4, Bull-LGI Grenoble, décembre 1987.

[Guide R5] Decouchant D., Rousset de Pina X., *Principe de réalisation du noyau d'exécution de Guide sous Unix System V*, Rapport de recherche Guide n°5, Bull-LGI Grenoble, juin 1988.

[Guide R6] Freyssinet A., Scioville R., Vandôme G., *Réalisation de la mémoire permanente d'objets dans le système Guide*, Rapport de recherche Guide n°6, Bull-LGI Grenoble, juin 1988.

TABLE DES MATIERES

CHAPITRE I
INTRODUCTION

CHAPITRE II
LA PROGRAMMATION PAR OBJETS
Présentation des concepts

CHAPITRE III
LE MODELE D'OBJETS DE GUIDE
Présentation Comparative

CHAPITRE IV
LE LANGAGE GUIDE
Exemples de programmation

CHAPITRE V
MISE EN ŒUVRE D'UNE APPLICATION GUIDE

CHAPITRE VI
CONCLUSION

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Madame ANDRE Françoise, Professeur
- Monsieur BEZIVIN Jean, Professeur

Madame MEYSEMBOURG Marie Laurence épouse MANNLEIN

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité " Informatique "

Fait à Grenoble, le 15 juin 1989

Pour le Président de l'I.N.P.G.
et par délégation
Le Vice-Président

C. GAUBERT