Université de Nice-Sophia Antipolis

D.E.A. d'Informatique 1994-1995

Un système de vues pour *Eiffel*

Pierre Crescenzo
crescenz@unice.fr

Rapport final de stage : juin 1995

Laboratoire "Informatique, Signaux et Systèmes de Sophia-Antipolis"

Thème "Objets, Langages, Environnements, Modélisation"

Équipe "Objet et Composant Logiciel"

Responsables : Robert Chignoli et Philippe Lahire
chignoli@unice.fr et *lahire@unice.fr*

Rapporteur : Rose Dieng
dieng@sophia.inria.fr

Résumé

En tant qu'informaticiens, nous avons tous connu la désagréable impression que les langages de programmation dont nous disposions n'étaient pas tout à fait adaptés à nos désirs, voire même à nos besoins. C'est dans cette problématique que s'est inscrit ce stage de D.E.A. d'Informatique.

Nous avons, en effet, souhaité modéliser un système de méta-programmation le plus souple possible qui permette de générer des langages à la carte. Une démarche identique a été suivie par les concepteurs du langage *CLOS* sur-couche de *Common LISP*. Nous nous sommes inspirés de ce travail et d'autres en partant cependant de postulats différents : le langage *Eiffel 3* nous a servi de base, le génie logiciel a été notre fer de lance et les vues furent notre modèle de départ.

Pour cela, nous avons tout d'abord réalisé une étude bibliographique des principaux thèmes et concepts liés à cette démarche. Nous avons étudié le protocole méta-objets *MOP* de *CLOS*, l'extension d'*Eiffel 3 VBOOL*, les systèmes de gestion de bases de données ainsi que des documents sur les aspects version et protection des langages de programmation. Nous avons complété nos lectures par l'abord de certains concepts logiciels utiles (telles les continuations).

Nous avons poursuivi notre travail en échafaudant une modélisation de notre système de vues. Nous avons abouti à un modèle comportant deux pôles essentiels : un langage de base et un protocole de gestion de vues. Le langage de base constitue le langage minimal initial qui pourra être étendu. Le protocole de gestion de vues décrit les concepts et mécanimes nécessaires pour les extensions. Nous avons défini à ce propos la notion de lien.

Dans un dernier temps, nous avons ébauché une description opérationnelle du noyau de notre système de vues. Nous avons enfin conclu en mettant en évidence les différentes voies possibles de poursuite de ce projet.

Remerciements

Je remercie les responsables de ce stage, Messieurs Robert Chignoli et Philippe Lahire, pour leur disponibilité et leurs conseils avisés et le rapporteur Madame Rose Dieng pour son rôle actif.

Je remercie également Monsieur Roger Rousseau et tous les autres intervenants de l'unité de valeur "Approches Orientées Objets pour le Génie Logiciel, les Bases de Données et la Conception Assistée par Ordinateur : vers un modèle commun" du "D.E.A. d'Informatique" pour leurs très instructifs enseignements.

Je remercie aussi Madame Mireille Fornarino et tous les autres intervenants de l'unité de valeur "Fondements de la Programmation par Objets" du même D.E.A. pour leur aide et leurs conseils.

Je remercie enfin Mademoiselle Bérengère Gason et Monsieur Jean-Louis Paquelin pour leur soutien constant et leurs conseils tout au long de ce stage.

Table des matières

Table des figures

Chapitre 1

Introduction

Ce stage s'est déroulé au laboratoire I3S (Informatique, Signaux et Systèmes de Sophia-Antipolis) dans l'équipe OCL (Objet et Composant Logiciel) du thème OLEM (Objets, Langages, Environnements, Modélisation) sous la direction de Robert Chignoli et Philippe Lahire.

L'équipe OCL travaille essentiellement depuis quelques années *avec* et *sur* le langage *Eiffel* [Mey94], langage dédié au génie logiciel. L'un des objectifs de l'équipe est d'ajouter de nouvelles fonctionnalités au langage *Eiffel*, pour améliorer sa puissance d'expression [1].

Le projet *FLOO* [CFLR93b, CFLR93a, Mic94] a, par exemple, permis d'expérimenter un modèle et des mécanismes de persistance automatique pour *Eiffel* et a mis en lumière le besoin de dépasser le cadre strict du stockage d'instances persistantes. L'étude de ces besoins a été initiée par Éric Michaudel lors de son stage de D.E.A. d'Informatique 1993-1994 [Mic94].

L'objectif de ce stage 1994-1995 est de proposer, dans le cadre d'*Eiffel*, des solutions orthogonales aux problèmes de *protection*, d'*évolution* et de *points de vues* posés par les environnements de programmation "objet". L'idée centrale est de bâtir un modèle basé sur un concept de niveau supérieur aux *classes* appelé *vue*.

Ce rapport se décompose en huit grandes parties : cette introduction, une présentation du sujet, une étude préalable comportant un état de l'art et une première proposition de modélisation. Les trois chapitres suivants concernent la définition du langage de base et d'un protocole de gestion de vues ainsi qu'une ébauche de définition du noyau. Ce document se termine par une conclusion énonçant les principales perspectives.

1. A ce titre, *Eiffel* peut être simplement considéré comme un *bon* représentant des langages "objet" pour le Génie Logiciel

Chapitre 2

Le Sujet

2.1 Introduction

Le sujet consiste en l'étude d'un système de vues pour *Eiffel*. Par système de vues, nous entendons tout mécanisme qui permettrait de créer, gérer, voire détruire une vue. Le mot *vue* est utilisé en référence à la terminologie des systèmes de gestion de bases de données où il décrit une manière d'accéder à un ensemble d'objets. La suite de ce chapitre éclairera le lecteur sur les motivations et les objectifs qui ont présidé à ce stage.

2.2 Motivations

Nous allons, dans cette partie décrire les principales motivations qui nous ont amené à nous intéresser à ce sujet :

- Nous souhaitons gérer un mécanisme de persistance des objets et tenter une liaison entre les langages orientés objets [Mey90, MNC+91] et les systèmes de gestion de bases de données orientés objets. Cette motivation est la raison "historique" qui a engendré les travaux sur ce sujet. Aujourd'hui, nos motifs sont plus larges, comme le montre la suite de ce chapitre, mais il est intéressant de comprendre l'idée originelle. Les langages de programmation basent leur force sur la puissance de traitement. En ce qui concerne les systèmes de gestion de bases de données, l'accent est mis sur les données. Mais cette séparation est arbitraire. Un programme repose souvent autant sur les traitements que sur les données. Notre objectif principal est donc d'intégrer les vues (qui forment un des principes de base des systèmes de gestion de bases de données) dans les langages orientés objets. Par le biais de la gestion de la persistance, nous pensons également aux problèmes de l'évolution des composants et de la protection des objets.

- Par généralisation du point précédent, un autre de nos buts est d'intégrer les notions de réflexivité et d'introspection [1] à *Eiffel 3*. *Eiffel 3* est un langage orienté objets pur qui présente de très nombreux avantages sur le plan du génie logiciel mais il comporte cependant quelques faiblesses. Nous nous proposons, lors de ce stage, d'étudier le moyen d'ajouter à *Eiffel 3* une capacité à s'auto-décrire (réflexivité) et à s'auto-observer (introspection).

- Par le biais de ces différentes extensions, nous avons le souhait d'étendre la puissance d'*Eiffel 3* sans lui retirer ses aspects "génie logiciel". Retirer ce qui donne à *Eiffel 3* sa force, même pour étendre le langage, ne serait que de peu d'intérêt. Il nous faudra intégrer notre système de vues sans nuire aux grands principes fondateurs d'*Eiffel 3* [Mey90, Mey94, RBC+95, Swi95] et ainsi lui apporter une puissance encore supérieure.

1. Un travail portant sur le même thème (*IREC*) a été mené par l'équipe. Cet outil apporte de bonnes solutions. Notre propos est d'étudier une autre voie pour aller encore plus loin.

– L'objectif à long terme de ce stage est de mettre en place les prémisses d'un système permettant de traiter aisément certains points clefs de la programmation tels la persistance et l'évolution [Bou94] dans *Eiffel 3*. En partant d'un langage comme *Eiffel 3* qui présente déjà un grand nombre de qualités, nous voulons présenter un mécanisme (que nous appelons "système de vues") permettant de construire un langage à la carte dont une instance serait *Eiffel 3*.

2.3 Objectifs

Nous voulons donc définir un système de vues pour *Eiffel 3* avec les objectifs suivants :

– Rapprocher le monde des traitements (langages de programmation) et celui des données (systèmes de gestion de bases de données) par le biais de l'intégration des vues. Cela rejoint notre motivation "historique".

– Définir un système de vues qui puisse modéliser le langage dans son intégralité. Nous pourrons, par exemple, définir l'héritage comme un cas particulier de vue. Cet aspect de la modélisation est beaucoup plus important qu'il n'y paraît. Le fait que le langage puisse être dans sa totalité modélisé à l'aide du système de vues permettra une meilleure adéquation entre le modèle et le langage et assurera une plus grande orthogonalité des structures et des données. Nous donnerons une place particulière à ce problème de l'orthogonalité dans notre démarche.

– Proposer un système de vues complet, extensible et simple d'emploi : complet car l'on ne peut se contenter d'une maquette. Notre vœu n'est pas de réaliser un jouet mais un véritable langage de programmation auto-modifiable pour devenir un réservoir de langages à la carte. Il doit être extensible parce qu'il est illusoire de croire que nous pourrons tout prévoir et parce qu'il doit être un mécanisme de base. Il lui faudra également être simple d'emploi car un système obscur est rarement bien utilisé quand il n'est pas carrément mis aux oubliettes.

Chapitre 3

Étude préalable

Ce chapitre s'articule autour de deux sections principales. D'une part, nous présentons un état de l'art du domaine concerné par notre étude. D'autre part, nous proposons un choix de modélisation qui donnera au lecteur les éléments nécessaires pour comprendre aisément les chapitres suivants.

3.1 Un état de l'art

Cette section présente donc un état de l'art du domaine étudié : les vues. Nous ne prétendons pas avoir fait un tour exhaustif des travaux sur ce sujet, les délais du stage étant trop courts pour cela. Nous espérons cependant avoir réunis les principales approches et pensons avoir étudié ainsi les éléments indispensables à une synthèse des concepts mis en jeu.

3.1.1 Utilité d'un système de vues

Avant de commencer notre étude sur les différents systèmes de vues proposés dans la littérature, nous avons voulu essayer de définir quels étaient les apports substantiels d'un système de vues dans un langage de programmation. Voici une liste qui, bien que non exhaustive, en donne un bon aperçu :

- Une vue permet de stocker les résultats de fonctions (ou requêtes) dans un but de réutilisation. En effet, pourquoi recalculer plusieurs fois un résultat quand celui-ci peut être stocké ?
- Bon nombre de langages de programmation permettent de définir aisément des ensembles d'objets. C'est notamment le cas pour tous les langages orientés objets dits de classes tel *Eiffel 3* : dans ce cas les ensembles d'objets sont les classes elles-mêmes. Mais peu d'entre eux permettent de gérer (créer, modifier ou détruire) ces ensembles (les classes : par opposition aux élements de ces ensembles : les objets) dynamiquement. Un système de vues peut apporter cette puissance supplémentaire.
- Dans les langages de programmation, le problème de la protection des objets est très peu souvent résolu car ils n'abordent pas le problème de la persistance des objets de manière complète. Comment faire lorsque certains objets doivent être modifiables et pas d'autres ? Comment gérer efficacement un système multi-utilisateurs avec différents degrés de responsabilités ? Les vues peuvent apporter une solution à ce genre de problème. [Mic94]
- Le problème de l'évolution est, lui aussi, une question d'actualité dans les langages de programmation. Un système de vues offre la possibilité d'une évolution simplifiée :
  - par la possibilité de définir un outil permettant la création de nouvelles vues avec un placement explicite ou implicite de la nouvelle vue dans la hiérarchie.

  - par les tests dynamiques des évolutions souhaitées du schéma (graphe d'héritage) : cela permet, par exemple, de faire évoluer un graphe d'héritage sans risque d'incohérence définitive de la base.
  - par la coexistence "pacifique" de plusieurs versions d'une même classe, d'un même objet, d'une même vue, ... On peut donc permettre l'accès à plusieurs versions d'une même entité ce qui réduit fortement les problèmes liés à l'évolution.

- Enfin, un système de vues offre la possibilite de voir un objet selon différents points de vue. C'est même, historiquement, son intérêt premier.

En conclusion de ces avantages apportés par un système de vues, nous pouvons dire que les vues donnent au langage qui les supporte une plus grande puissance grâce aux fonctionnalités supplémentaires de manipulation qu'elles apportent sur les données et les structures.

3.1.2 Caractéristiques nécessaires d'un système de vues

D'après l'étude bibliographique menée, nous définissons les grands principes qu'un bon système de vues doit satisfaire :

La simplicité : C'est un critère important. Le système que nous devons définir doit être simple aussi bien dans les concepts que dans l'utilisation que l'on peut en faire.

La richesse : Un nouveau système n'est intéressant que s'il apporte un large éventail de possibilités nouvelles. Il faudra donc que notre système de vues soit le plus riche possible.

Le génie logiciel : Aucun des principes qui font la puissance d'*Eiffel 3* ne devra être remis en cause. Nous pensons ici notamment aux grandes idées du génie logiciel : performance, efficacité, réutilisabilité, ... [Mey90]

3.1.3 Grands principes d'un système de vues

Plusieurs approches ont été étudiés dans le domaine des vues et des points de vues. Nous présentons trois de ces approches qui nous semblent être parmi les plus importantes : le domaine des systèmes de gestion de bases de données [CK94, BK93], le *MOP* de *CLOS* [Kee89, KdRB91] et *VBOOL* [MKC95, MEC95], une extension d'*Eiffel*.

Les vues dans les systèmes de gestion de bases de données

Les systèmes de vues ont été très étudiés dans les systèmes de gestion de bases de données orientés objets [CK94] et peu dans les langages orientés objets. En fait, le sujet des vues est assez ancien dans les systèmes de gestion de bases de données. Il a, en effet, été intégré assez tôt pour résoudre le problème épineux de la capacité des systèmes de gestion de bases de données à être multi-utilisateurs. La redondance de données étant un inconvénient majeur dans ces systèmes, les vues (qui permettent le partage d'objets) sont donc très vite apparues. Les langages orientés objets se tournent maintenant de plus en plus vers la gestion efficace de données, mais cette voie est assez récente et les vues n'ont pas encore vraiment trouvé leur place dans ces langages.

De cette intégration est né un objectif : intégrer les opérations de projection, sélection, agrégation, désagrégation et jointure sur les composants logiciels et les objets aux langages de programmation. Cela implique la gestion d'"ensembles d'objets partagés".

La plupart des propositions qui sont faites [BK93, CK94] ne sont que des propositions de *modus operendi*. On donne une méthode pour faire des vues mais aucun outil. Nous pouvons nous référer avec profit à la partie 4.3 de [BK93] pour voir, par exemple, les différentes étapes de la création d'une vue de sélection. On remarquera que toute la responsabilité de la cohérence de la vue avec sa classe d'origine est laissée à la seule charge du programmeur. Ce genre de méthode, pour intéressante qu'elle soit, ne peut favoriser les opérations sur les ensembles dans les langages orientés objets. Si c'était le cas, il faudrait admettre de recommencer le travail de garantie de la

cohérence pour chaque vue, ce qui est beaucoup trop lourd. Notre démarche nous amènera, si elle est menée à terme, à fournir des outils qui assisteront le programmeur dans toutes les tâches qui peuvent être automatisées dans la gestion de vues.

MOP **de** *CLOS*

MOP est un protocole méta-objets basé sur le langage de programmation *Common LISP* pour donner le langage orienté objets *CLOS*. Outre le fait de permettre une programmation orientée objets assez classique, *MOP* introduit le concept de méta-programmation au travers d'un protocole basé sur un graphe d'héritage mettant en jeu des classes particulières, les méta-classes. *CLOS* conserve toutes les particularités de *Common LISP* et offre donc, par l'intermédiaire de son *MOP*, une interface permettant de produire des langages à la carte selon les besoins des programmeurs.

Certains de nos objectifs sont donc compatibles avec ceux des protocoles méta-objets tel celui de *CLOS* [FC95, KdRB91], mais nous nous orientons plus vers l'intégration des langages de programmation et des systèmes de gestion de bases de données. *MOP* reste cependant pour nous un des objets d'étude les plus intéressants tant pour son modèle basé sur l'héritage et les méta-classes que pour sa réalisation qui a déjà montré toutes ses capacités. Nous présenterons plus tard dans ce rapport les différences importantes entre *MOP* et notre système de vues.

VBOOL, **une extension d'***Eiffel*

VBOOL [MKC95, MEC95] est une proposition d'extension d'*Eiffel* très récente intégrant la notion de point de vue. L'approche des auteurs est cependant assez éloignée de la notre. Il ne s'agit pas pour eux de créer un nouveau langage extensible mais d'ajouter à *Eiffel* la possibilité de gérer des points de vue, en s'appuyant sur l'héritage.

Ces nouvelles fonctionalités reposent sur une extension de la syntaxe d'*Eiffel 3* permettant de déclarer dans une classe l'ensemble de ses vues. Les auteurs de [MKC95] donnent ainsi les définitions suivantes :

Définition 1 (Vue de *VBOOL*) *Une vue d'une classe C est une autre classe déclarée dans une clause prévue à cet effet de C.*

Définition 2 (Point de vue de *VBOOL*) *Un point de vue sur une classe est une relation qui à cette classe associe un sous-ensemble de ses vues.*

Ainsi, il est possible de réunir plusieurs vues dans un point de vue. Une primitive est recherchée dans un point de vue (c'est-à-dire dans chacune des vues qui composent ce point de vue) à chaque fois que la recherche polymorphique traditionnelle a échoué. Les éventuels conflits entre primitives de plusieurs vues doivent être résolus à la compilation.

Pour conclure sur ce travail, précisons que *VBOOL* devrait être fourni sous la forme d'un environnement de programmation incluant un compilateur générant de l'*Eiffel 3*.

3.2 Le modèle retenu

Cette section présente d'une manière simple et synthétique les différents concepts qui régissent le modèle que nous avons choisi. Une description plus détaillée en sera donné dans les deux chapitres suivants.

Tout d'abord, rappelons notre but. Nous souhaitons créer un langage de génie logiciel qui puisse être étendu facilement. Ce langage reposera en grande partie sur *Eiffel 3*. Notre démarche est la suivante :

1. Définir aussi précisément que possible une modélisation qui permette de décrire un langage extensible et un protocole d'extension de ce langage.

2. Utiliser au mieux cette modélisation pour concevoir et décrire un premier langage réel muni de vues et conforme aux principes du génie logiciel.

Il nous faudra donc, bien entendu, faire des compromis entre une modélisation théorique parfaitement orthogonale et une réalisation qui tire parti au maximun de cette modélisation sans oublier le côté pratique.

Le modèle que nous avons retenu repose sur deux entités essentielles :

Le langage de base. C'est le point de départ, l'entité minimale qui représente le langage qui peut être étendu par la suite. Ce langage est inspiré d'*Eiffel 3* auquel on a retiré certains concepts : nous l'appellerons O^{fl} [1]. Ce langage sera décrit dans le chapitre 4 page 12.

Un protocole de gestion de vues. Ce protocole constitue le mécanisme qui permettra d'étendre O^{fl}. Il est défini dans le chapitre 5 page 17.

Pour comprendre facilement ces deux chapitres, il faut avoir à l'esprit quelques définitions préalables :

Vue. C'est une généralisation de la notion de classe.

Méta-vue. C'est une entité capable de créer et gérer des vues. Nous verrons que le concept de méta-vue se rapproche de celui de langage.

Lien. Il s'agit d'une relation existant entre deux vues (par exemple un relation de clientèle ou d'héritage simple).

Méta-lien. C'est une entité capable de créer et gérer des liens spécifiques.

Ces définitions sont évidemment incomplètes et simplifiées. Elles seront, bien sûr, précisées et complétées lors de cette étude.

1. O^{fl} signifie "*Open Eiffel*".

Chapitre 4

Le langage de base

Ce chapitre, comme nous venons de le mentionner, propose une définition du premier élément essentiel de notre système de vues : le langage de base (que nous nommerons O^{fl}). Ce chapitre se présente en trois parties. Tout d'abord une présentation générale d'O^{fl} est proposée. Elle est suivie d'une description de la structure de ce langage. La dernière partie décrit les principes que nous souhaitons intégrer à notre langage de base.

4.1 Généralités sur O^{fl}

La philosophie qui a présidé à la création de ce langage relève de la démarche suivante. Nous possédons un bon langage de programmation qui met l'accent sur les principes du génie logiciel [RBC+95] : ce langage est *Eiffel 3* [Mey94, Swi95]. Ce langage est cependant trop avancé pour fournir une base minimale à un protocole de gestion de vues. Nous proposons donc de reprendre ce qui nous semble être le plus intéressant dans *Eiffel 3* (sa substantifique moelle) pour constituer un langage minimal qui contienne pourtant tous les principes du génie logiciel.

Nous avons étudiés trois mécanismes fondamentaux d'*Eiffel 3* :

La clientèle. C'est-à-dire le mécanisme de contrôle de la visibilité entre objets,

L'héritage. *i. e.* la relation de sous-typage et

L'envoi de message.

Nous souhaitions retirer ces trois mécanismes d'O^{fl}, avec l'objectif de les réintégrer plus tard sous forme de liens. Pour les deux premiers, la clientèle et l'héritage, cela ne pose aucun problème. En ce qui concerne l'envoi de message, nous avons finalenemt choisi de le conserver. Voici la démarche qui a amené cette décision.

Nous faisons la différence, au sein de l'envoi de message, entre deux activités :

Le déclenchement de l'exécution. Un objet peut exécuter une des ses primitives. Il peut, par ce système, gérer ses champs expansés car ceux-ci lui appartiennent. Cette fonctionnalité est nécessaire sinon le langage ne peut être que déclaratif.

Le protocole de communication. Un objet peut envoyer un message à un autre objet pour lui demander d'exécuter une de ses primitives. Le protocole de communication utilise donc le déclenchement de l'exécution qui est un mécanisme de plus bas niveau. Le protocole de communication est utile pour :

- l'asynchronisme,
- l'échange entre entités distinctes,
- assurer l'indépendance des entités.

Il faudra alors gérer un système d'adressage. On peut s'adresser à une entité locale comme si elle était distante (par le protocole de communication), mais on peut aussi s'adresser à elle directement par le mécanisme de déclenchement de l'exécution.

Si l'on retire l'envoi de message, on peut donc tout de même exécuter une primitive locale. Mais cela entraine plusieurs conséquences directes :

1. Il ne peut y avoir qu'un seul véritable objet (une référence sur un objet) dans chaque application.
2. S'il y a d'autres références, elles ne peuvent être que déclarées et jamais rattachées à un objet.
3. Tout les autres objets sont de type expansé.

Outre l'aspect peu logique d'un langage orienté objets ne pouvant gérer qu'un seul objet, il est apparu que le retrait posait aussi le problème des types récursifs. Imaginons une classe `PERSONNE` définissant le champ `Époux`. Rien ne nous empêche de faire de cet époux un objet expansé. Mais dans ce cas, il nous sera impossible d'instancier l'époux de l'époux. En d'autres termes, si l'objet `A` contient l'objet `B` alors `B` ne peut contenir `A`. Le retrait de l'envoi de message a donc, pour ces raisons, été abandonné.

4.2 Structure d'O^{fl}

O^{fl} permet donc la gestion d'objets complexes, c'est-à-dire d'objets ayant pour champs des objets. O^{fl} lui-même est constitué de deux grandes parties :

Le moteur du langage. C'est le noyau. Nous ne donnerons pas, pour l'instant de syntaxe à O^{fl}. Le lecteur pourra cependant trouver dans le chapitre 6 page 27 une ébauche de définition des primitives de ce noyau. Lorsqu'il nous faudra donner un exemple, nous utiliserons une syntaxe "à la *Eiffel 3*" pour clarifier les choses. Mais ce choix n'a rien de définitif et O^{fl} se veut, pour le moment, indépendant des problèmes syntaxiques. Les concepts d'expansion (qui décrit les objets expansés) et de référence (pour les objets référencés) explicités ci-dessous sont les liens initiaux sur lesquels repose O^{fl}.

La bibliothèque de base. Elle doit comporter tout ce qui peut être utile au programmeur. À cette occasion, nous rappelons qu'O^{fl} est un véritable langage de programmation : il est donc évidemment possible d'écrire des applications en O^{fl} même si, de notre point de vue, il s'agit d'un support d'extensions. O^{fl} est un langage orienté objets donc sa bibliothèque de base est constituée de vues (ou modèles). Mais O^{fl} ne possède pas l'héritage : il n'y aura donc pas de graphe de vues dans cette bibliothèque. Pour cette raison, celle-ci est restreinte aux seules vues d'intérêt général.

4.2.1 Vues de la bibliothèque de base

Précisons tout d'abord que, dans notre approche, une vue est non seulement un ensemble d'objets associée à des primitives mais aussi un objet à part entière.

Comme nous l'avons précisé ci-dessus, la bibliothèque de base est constituée de vues. Voici celles qui nous paraissent devoir tout particulièrement y figurer :

- `ARRAY` (tableaux),
- `BOOLEAN` (booléens),
- `CHARACTER` (caractères),
- `DOUBLE` (réels en double précision),

– `EXCEPTION` (exceptions),

– `FILE` (fichiers),

– `INTEGER` (entiers),

– `POINTER` (adresses),

– `REAL` (réels en simple précision) et

– `STRING` (chaines de caractères).

Ces classes formeront donc une bibliothèque de composants logiciels qui seront tous sur un pied d'égalité, *i. e.* non liés par une hiérarchie.

4.3 Grands principes d'O^{fl}

4.3.1 Apports directs de *Eiffel 3*

O^{fl} conserve en l'état les particularités suivantes de *Eiffel 3*:

Les structures de contrôle :

– Le composé,
– L'instruction nulle (;),
– La conditionnelle (**if**...),
– La sélection à choix multiple (**inspect**...),
– La boucle (**from**...) et
– L'instruction débogage (**debug**...).

Les genres de types :

– Type de référence : C'est le genre du type des objets traditionnels souvent décrit par une référence sur une entité objet.
– Type expansé : C'est le genre du type de l'entité objet elle-même. On peut y accéder sans l'indirection due à la référence.
– Type ancré : Cela décrit un type contraint à être égal un autre type. Il est mis en œuvre dans *Eiffel 3* par le mot-clef **like**.
– Type générique formel : C'est le genre de type des paramètres génériques formels [1].

Les assertions :

– La précondition (**require**...),
– La postcondition (**ensure**...),
– L'invariant de classe (**invariant**...) qui devient un invariant de vue,
– L'invariant de boucle (**invariant**...) et
– L'instruction de vérification (**check**...).

Les exceptions :

– La violation d'une assertion,
– L'échec d'une routine appelée,
– L'utilisation d'une entité de valeur nulle,

1. La généricité dans O^{fl} est présentée section 4.3.2 page 15

- L'opération impossible,
- L'émission d'un signal d'interruption et
- L'exception levée par l'application.

Ces quatres concepts sont en effet totalement indépendants des notions de clientèle et d'héritage qui ont été retirées.

Le retardement est, pour sa part, abandonné. En effet, le mot-clef **deferred** de *Eiffel 3* semble n'avoir qu'une valeur méthodologique. En fait, techniquement, la définition d'une classe retardée revient à dire que :

- Cette classe ne peut avoir d'instance, ce qui peut être résolu par une interdiction de l'opérateur de création.
- Au moins une primitive de cette classe n'a pas d'implémentation, ce qui peut être résolu par une primitive vide associée à des clauses d'adaptation de type.

4.3.2 Généricité

La généricité d'O^{fl} est une généralisation de celle d'*Eiffel 3*. Nous rappelons que si nous donnons les exemples dans un langage "à la *Eiffel 3*" cela n'est pas significatif d'un choix de syntaxe pour O^{fl}. Il s'agit seulement d'un moyen commode de décrire la représentation interne donnée par O^{fl}.

En *Eiffel 3*, on peut écrire : `class T1 [T2 → T3]` ce qui signifie : `T1` est un type générique sur `T2` mais `T2` doit être `T3` ou un de ses descendants. Par exemple, si l'on veut créer le type "vecteur d'objets qui acceptent la sommation", on écrira : `Class VECTEUR [T → SOMMABLE]`.

En O^{fl}, on pourra écrire : `view T1 [T2 → (T3,T4)]` ce qui signifiera : `T1` est un type générique sur `T2` mais `T2` doit être `T3` ou `T4`. Par exemple, si l'on veut créer le type "vecteur d'objets qui acceptent la sommation ou la comparaison" – Il s'agit bien d'une disjonction et non d'une conjonction.– on écrira : `Class VECTEUR [T → (SOMMABLE,COMPARABLE)]`. Cette fonctionnalité développera évidemment toute sa puissance dans le cadre d'un lien d'héritage mais elle est déjà très utile dans O^{fl}.

4.3.3 Clauses d'adaptation de type

Les clauses d'adaptation de type de l'héritage d'*Eiffel 3* nous semble tout à fait généralisable à un lien quelconque. Ces clauses ne sont donc pas spécifiques à chaque lien et leur définition peut être factorisée dans les méta-liens.

Rappelons que les clauses d'adaptation de type de l'héritage d'*Eiffel 3* sont :

- le renommage (**rename**. . .),
- la redéfinition (**redefine**. . .),
- la sélection (**select**. . .) et
- l'annulation (**undefine**. . .).

4.3.4 Indexation

La clause d'indexation d'*Eiffel 3* se présente sous la forme d'une clause de commentaires structurés décrivant la classe. Nous pensons qu'il est utile de conserver cette particularité dans les vues d'O^{fl}. Nous gardons en effet à l'esprit l'idée de créer, par la suite, un lien "Documentation" qui tirerait parti de cette clause. On peut enfin remarquer que cette clause pourrait être sans dommage remplacée par des variables de vue dans les vues (le pendant des variables de classe dans les classes).

4.3.5 Méthodes auxiliaires

Sur le modèle des méthodes *before*, *around* et *after* de *CLOS* [Kee89, KdRB91], O^{fl} disposera de méthodes auxiliaires permettant de réaliser des opérations avant, autour ou aprés une méthode principale. Nous appelerons ces méthodes auxiliaires les pré-méthodes, les sur-méthodes et les post-méthodes.

Tout l'intérêt de ce type de méthode repose sur la puissance d'incrémentalité de ce système. Inutile de réécrire du code, il suffit de l'"encapsuler" par une ou plusieurs méthodes auxiliaires.

4.3.6 Autres caractéristiques

La description d'O^{fl} que nous donnons étant encore améliorable, certains choix n'ont pas encore été fait concernant certaines caractéristiques. Ces caractéristiques sont surement intéressantes mais il nous restera à définir ce qu'elles peuvent précisément apporter au langage de base et dans quelle mesure elles pourront être intégrées à O^{fl}. Nous pensons notamment au nommage des paramètres [AKDM+94] et aux continuations [Dal95, SF90, CP94].

Nous ne faisons ici qu'une rapide description des concepts de nommage des paramètres et de continuations (car les expliquer n'est pas notre propos) mais le lecteur intéressé ou curieux pourra consulter avec profit les ouvrages cités.

Le nommage des paramètres

Le nommage des paramètres permet d'identifier chaque paramètre exactement. Il apporte une plus grande souplesse de programmation, les paramètres pouvant être cités dans un ordre quelconque, voire même être omis. Ce point nous semble d'autant plus important qu'il permet une gestion simple et efficace d'un mécanisme de valeur par défaut des paramêtres. Il est cependant difficile d'étudier précisemment ce point sans parler des aspects syntaxiques, aussi ne donnons-nous pas d'avis définitif sur ce sujet.

Les continuations

Les continuations[2] sont une structure de données permettant de "capturer le futur d'un calcul" dans une variable pour pouvoir l'appeler explicitement plus tard. Ce concept étant très important mais risquant de déroger aux règles d'encapsulation des objets dans O^{fl}, nous le gardons à l'esprit sans idée préconçue de notre décision finale. Il est cependant important de mettre en évidence que la discussion sur l'ajout d'un tel mécanisme devra s'accompagner, selon nous, d'une proposition de créer un type de base décrivant des routines. L'existence d'un tel type est en effet une condition préalable nécessaire pour les continuations qui peuvent être assimilées à des routines dont on peut "générer" le corps en cours d'exécution.

2. Nous employons le terme continuation au sens qui est donné à ce mot dans les langages *Scheme* et *SML* et pas au sens des langages d'acteurs (bien que ces deux définitions ne soient pas sans rapport, mais cela est une autre histoire).

Chapitre 5

Un protocole de gestion de vues

5.1 Introduction

La seconde partie essentielle de notre système de vues est le protocole de gestion de vues [FC95]. Ce chapitre donne une description de ce protocole. Nous rappelons à cette occasion que ce protocole de gestion de vues que nous nommerons *VMP* (*View Management Protocol*) décrit les mécanismes qui permettent d'étendre O^{fl} pour faire des langages de plus haut niveau. Comme le montre la figure 5.1 page 18, O^{fl} peut lui-même être considéré comme une méta-vue de base comportant les liens d'expansion et de référence (cf. 5.6 page 21).

5.2 Généralités sur le protocole de gestion de vues

Avant de présenter *VMP*, il nous faut donner quelques définitions :

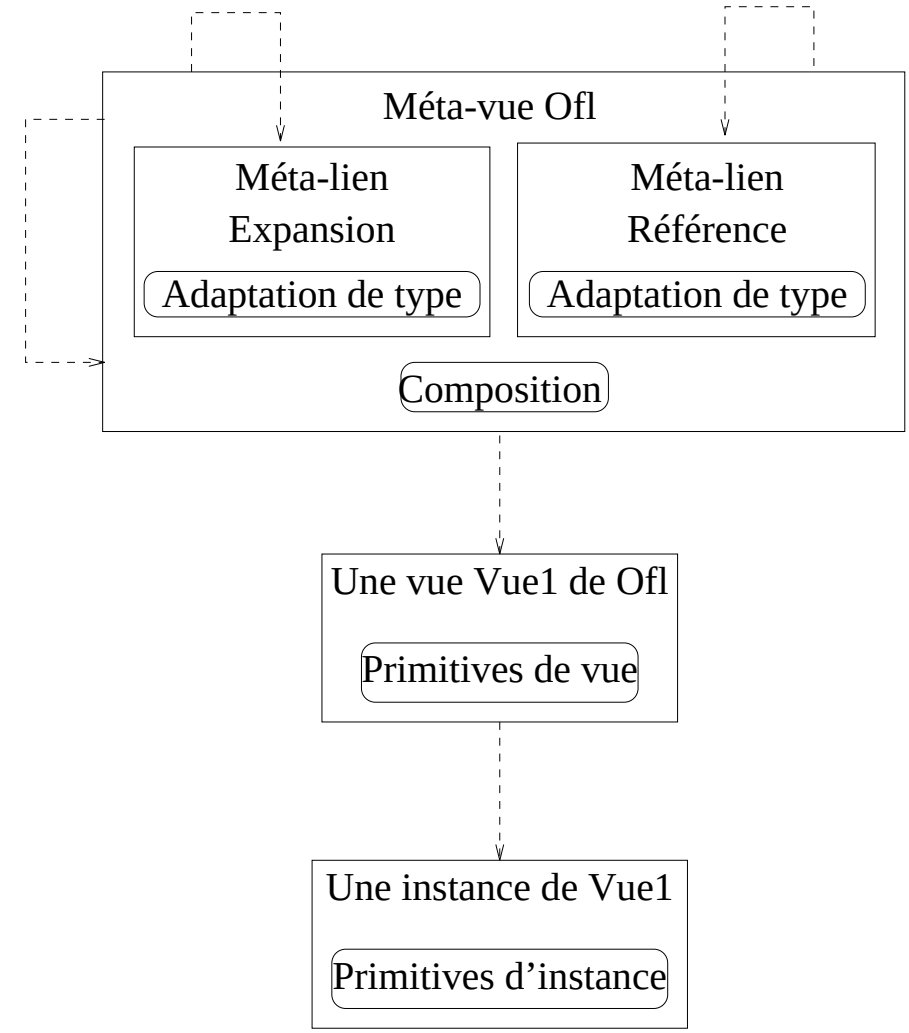
Définition 3 (Un lien) *La définition du Larousse*[1] *pour "lien" est "ce qui attache, unit, établit un rapport logique ou de dépendance". Pour nous, un lien est une instance d'un méta-lien : c'est un méta-lien dont les éventuels paramètres sont fixés. Un lien définit une relation existant entre plusieurs entités du langage (deux vues par exemple). Un lien possède un certain nombre de caractéristiques associées telles les clauses de adaptation de type. Une description en est donnée dans la section 5.4 page 20.*

Définition 4 (Un méta-lien) *Il s'agit d'un modèle de lien éventuellement paramétrique. Chaque méta-lien, s'il n'est pas paramétrique, peut engendrer une seule sorte de lien et, s'il est paramétrique, plusieurs sortes de liens. Les sections 5.6 page 21 et 5.7.2 page 24 présentent quelques méta-liens courants.*

Définition 5 (Une vue) *La définition du Larousse pour "vue" est "manière de voir, d'interpréter, de concevoir quelque chose". Pour nous, une vue est une classe qui admet une composition particulière de liens. En ce sens, c'est une instance de méta-vue pour laquelle on a spécifié l'ordre et la méthode de composition des liens. La composition de liens est toujours possible (à condition de la décrire), mais elle n'a jamais que la sémantique donnée par l'opérateur de composition. Nous donnons une définition plus complète des composants d'une vue dans la section 5.5 page 21.*

Définition 6 (Une méta-vue) *C'est un ensemble de méta-liens sélectionnés pour décrire un modèle de relation entre deux vues. Par extension, nous parlons parfois de langage en employant ce mot comme synonyme de "méta-vue bien structurée". On peut en effet considérer qu'une méta-vue décrit un langage de programmation comme la méta-vue O^{fl} décrit le langage O^{fl}.*

1. *Le petit Larousse Illustré 1995*, édité par Larousse.



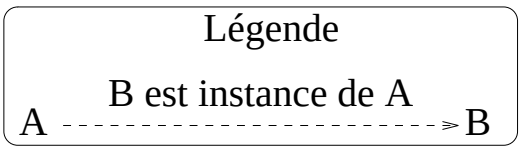


FIG. 5.1 - *Le langage de base :* O^{fl}

5.3 Exemple commenté

Nous présentons, sur la figure 5.2 page 19 un exemple qui va nous permettre de mettre en situation les différents concepts qui forment la proposition que nous faisons.

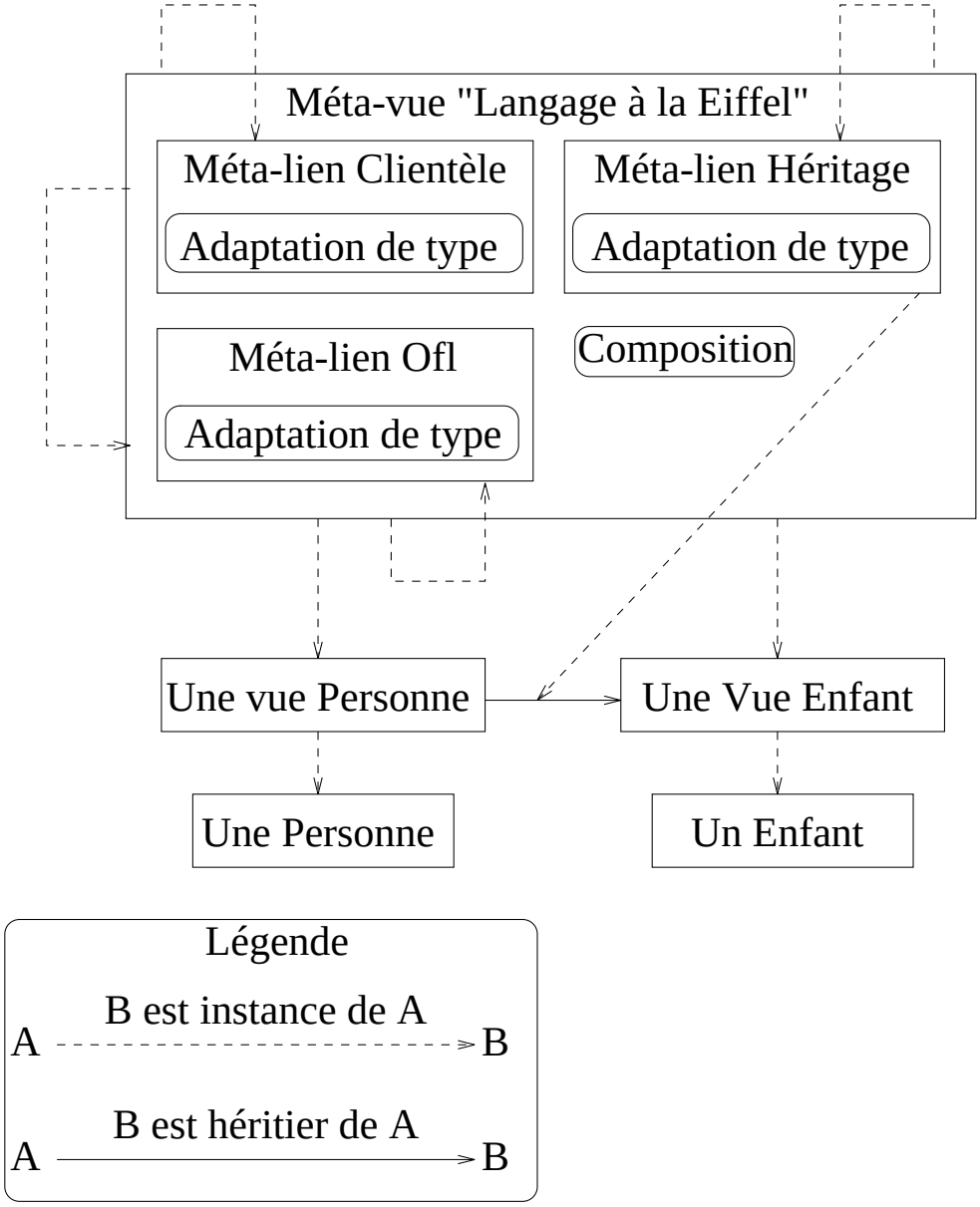


FIG. 5.2 - *Un exemple mettant en œuvre les concepts du modèle*

Le but de cet exemple est d'essayer de montrer comment l'on pourra modéliser le langage *Eiffel 3* (ou plutôt un sur-ensemble d'*Eiffel 3*) dans notre système de vues.

Quels sont les liens présents dans *Eiffel 3*? Nous avons evidemment le lien de clientèle: ce lien est en fait présent dans de nombreux langages de haut niveau. Nous ne pouvons non plus laisser de côté le lien d'héritage. Nous utiliserons donc les méta-liens "Clientèle" et "Héritage".

Il nous faut donc créer ces deux méta-liens en renseignant toutes les clauses nécessaires (la description précise d'un méta-lien est donnée dans la section 5.4 page 20). Nous la faisons donc en choisissant de les décrire pour qu'ils aient le même comportement que les liens présents dans

Eiffel 3. Nous voilà donc en possession de deux méta-liens qui décrivent un langage à la *Eiffel 3*. Nous allons les *combiner* avec la méta-vue "Ofl" pour former une nouvelle méta-vue. Cette combinaison correspond , en fait, à une composition des méta-liens "Expansion" et "Référence" du langage O^{fl} avec les méta-liens "Clientèle" et "Héritage". Sur la figure, la méta-vue générée est représentée par un rectangle rassemblant les méta-liens "Clientèle", "Héritage" et "Ofl". Nous pouvons remarquer que chacun des méta-liens possède des mécanismes de adaptation de type et que la méta-vue formée décrit la composition de ces méta-liens. Nous pouvons aussi remarquer que les trois méta-liens ainsi que la méta-vue sont des vues instances de la méta-vue ce qui permet de conserver une grande orthogonalité des concepts et de ne pas "laisser de trou" dans le graphe d'intanciation.

Grâce à cette méta-vue, nous allons pouvoir générer des vues qui décriront un graphe d'héritage et de clientèle. Nous voyons ainsi que la méta-vue "Langage à la *Eiffel*" peut générer les vues Personne et Enfant et gérer un lien d'héritage (issu du méta-lien "Héritage") entre ces vues. Les vues Personne et Enfant peuvent elles-même créer des instances terminales (ı. e. qui ne sont pas des vues) qui finalisent le schéma.

Voilà donc comment nous pouvons modéliser l'extension du langage de base O^{fl} en un langage à la *Eiffel 3*.

5.4 Description d'un méta-lien

Nous donnons, dans cette section, la description de la structure d'un méta-lien. Nous donnerons, tout au long de cettre description, des exemples tirés du méta-lien "Héritage" (à la *Eiffel*) pour que le lecteur puisse avoir une bonne idée des concepts décrits. Mais posons tout d'abord deux définitions nécessaires :

Définition 7 (La source) *La source, dans un méta-lien, est la vue qui fournit les services. Par exemple, dans le méta-lien "Héritage", la source est la vue héritée. Un lien peut avoir plusieurs sources.*

Définition 8 (La destination) *La destination est, dans le méta-lien, la vue qui reçoit les services. Par exemple, dans un lien d'héritage, la destination est la vue héritière. Un lien ne peut avoir qu'une seule destination.*

La(es) source(s) et la destination forment donc les deux partis d'une relation contractuelle.
Quatre clauses composent un méta-lien :

Visibilité. Elle définit la liste des services que la(es) source(s) fournit(ssent) à la destination. Dans le cas du méta-lien "Héritage", cette clause décrit la liste des primitives héritables.

Exécution. Elle décrit le comportement des opérateurs génériques (envoi de message, affectation et création)[2] pour ce méta-lien. Une description de cette clause du méta-lien "Héritage" est donnée dans la section 5.7.2 page 25.

Adaptation de type. Il s'agit de la définition des clauses de adaptation de type entre deux liens issus de ce méta-lien. Deux sortes de clauses sont définies :

Les clauses d'adaptation. redéfinition, renommage et annulation.

La clause de résolution de conflit. sélection.

Dans le cadre d'un méta-lien "Héritage", la sémantique des clauses de adaptation de type est bien définie, il s'agit de celle donnée par Bertrand Meyer dans [Mey94].

2. Il n'est pas exclu que cette liste s'allonge dans le futur. Nous pensons notamment aux opérateurs de copie et de comparaison.

Collection. Il s'agit de toutes les informations nécessaires pour la gestion de l'ensemble des instances. Pour le méta-lien "Héritage", il pourra s'agir de la gestion des instances de la source en fonction de la collection de la destination. Cela pourra se faire au travers de mécanismes décrivant des règles telle "Toute instance de la destination est instance de toute source".

Pour conclure cette description, nous précisons que deux liens issus du même méta-lien ne pourront être composés et que les éventuels conflits seront résolus grâce aux clauses d'adaptation et à la clause de sélection.

5.5 Description d'une vue

Nous quittons ici la couche "méta" pour spécifier plus complètement le concept de base du langage : la vue. Une vue est une entité un peu plus complexe qu'une simple composition de liens. Elle sert de modèle de description à ces instances et renferme des objets. Dans cette section, nous nous servirons d'une classe imaginaire *Eiffel 3* Personne que nous utiliserons pour présenter ce que pourrait être le squelette d'une vue. Une vue peut être composée de :

un nom. Pour la classe *Eiffel 3* Personne, c'est tout simplement l'identificateur "Personne".

des primitives. Elles peuvent être importées (avec redéfinition, rennomage, ...) à travers les liens ou ajoutées explicitement. Ces primitives constituent le parallèle des champs et méthodes de la classe *Eiffel 3* Personne.

une collection. Elle est définie à la création par un critère ou en extension. Elle est gérée au travers des liens. On peut faire le parallèle avec l'ensemble des instances de la classe *Eiffel 3* Personne. Remarquons que dans notre modèle toute vue connaît l'ensemble de ces instances, ce qui n'est pas le cas des classes *Eiffel*.

une sémantique des opérateurs génériques. L'envoi de messages ("."), la création ("!!") et l'affectation (":=") obtiennent leur sémantique de la composition des liens dont la vue est destination. Pour la classe Personne en *Eiffel 3*, la sémantique de ces opérateurs est fonction des clientèles et héritages mis en jeu dans cette classe.

un ensemble de liens. C'est le pendant de l'ensemble des relations de clientèle et d'héritage dans la classe Personne. En O^{fl}, les liens ne se limiteront évidemment pas à ces deux liens traditionnels d'*Eiffel 3*.

un ensemble d'informations. Elles serviront à la gestion des liens. La structure de ces informations n'est pas encore formalisée.

5.6 Exemples de méta-liens

Les méta-liens "Expansion" et "Référence" sont les méta-liens de base qui décrivent les relations entre vues. Il peuvent être enrichis et composés avec d'autres méta-liens pour former un langage à la carte.

La description que nous donnons ici est uniquement celle de la clause "Exécution" (seule cette clause est réllement utile pour ces méta-liens) des méta-liens de base. Les préconditions et postconditions présentées ici pourront être considérées en tant que telles (comme des assertions) ou faire l'objet d'une condition directement dans le corps de l'opérateur (comme une simple vérification explicite à l'exécution).

5.6.1 Méta-lien "Expansion"

Ce méta-lien nommé "Expansion" est le premier des deux méta-liens de base d'O^{fl}. Il permet la gestion des objets expansés. Rappelons que cette formalisation est réalisée dans un but d'orthogonalité avec les méta-liens qui seront ajoutés par la suite. Il ne s'agit pas seulement de décrire le langage mais bien de le formaliser de manière homogène. Du point de vue de l'exécution, nous pouvons dire que le méta-lien "Expansion" gère la partie de l'envoi de message que nous avons appelée "déclenchement de l'exécution" dans la section 4.1 page 12.

L'envoi de message "." : Objet.Primitive

Précondition : ∃ Objet (expansé) *et* Primitive ∈ vue de Objet

Rôle : Si Primitive est une méthode, on l'exécute ; si c'est un champ, on y accède.

Postcondition : *Si* Primitive = Méthode *alors* Primitive exécutée sur Objet *sinon* (Primitive = Champ) Champ de Objet accédé

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : ∃ Objet (expansé) *et* Champ ∈ vue de Objet *et* ∃ AutreObjet (expansé)

Rôle : Copie de AutreObjet dans Champ de Objet.

Postcondition : AutreObjet copié dans Champ de Objet

La création "!!" : ![Type]!Objet[.Initialiseur]

Précondition : ∃ Type (si spécifié) *et* ∃ Objet (expansé) *et* Initialiseur ∈ vue de Objet (si spécifié)

Rôle : Réservation mémoire pour Objet *puis* liaison de Objet avec sa vue *puis* initialisation de Objet par Initialiseur (si spécifié).

Postcondition : Objet créé ; *Si* Type spécifié *alors* Objet a pour type Type *sinon* Objet a pour type celui de sa déclaration ; *Si* Initialiseur spécifié *alors* Objet initialisé par Initialiseur *sinon* Objet contient ValeurParDéfaut

5.6.2 Méta-lien "Référence"

Ce second méta-lien "Référence" du langage de base est chargé, pour sa part, de la gestion des objets référencés. Comme pour le méta-lien "Expansion", nous ne donnons que la définition des opérateurs génériques. Ajoutons enfin que le méta-lien "Référence" assure la fonction de "protocole de communication" décrite section 4.1 page 12.

L'envoi de message "." : Objet.Primitive

Précondition : ∃ Objet (référencé) *et* Primitive ∈ vue de Objet

Rôle : Si Primitive est une méthode, on l'exécute et on récupère le résultat éventuel ; si c'est un champ, on y accède.

Postcondition : *Si* Primitive = Méthode *alors* Primitive exécutée sur Objet et éventuel résultat récupéré *sinon* (Primitive = Champ) Champ de Objet accédé

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : ∃ Objet (référencé) *et* Champ ∈ vue de Objet *et* ∃ AutreObjet (référencé)

Rôle : Copie de AutreObjet dans Champ de Objet.

Postcondition : AutreObjet copié dans Champ de Objet

La création "!!" :![Type]!Objet[.Initialiseur]

Précondition : ∃ Type (si spécifié) *et* ∃ Objet (référencé) *et* Initialiseur ∈ vue de Objet (si spécifié)

Rôle : Réservation mémoire pour Objet *puis* liaison de Objet avec sa vue *puis* initialisation de Objet par Initialiseur (si spécifié).

Postcondition : Objet créé ; *Si* Type spécifié *alors* Objet a pour type Type *sinon* Objet a pour type celui de sa déclaration ; *Si* Initialiseur spécifié *alors* Objet initialisé par Initialiseur *sinon* Objet contient ValeurParDéfaut

5.7 Fonction de composition

5.7.1 Composition de méta-liens

Nous noterons la fonction de composition de méta-liens $\oplus$. Commençons par donner un exemple.

Soient E le méta-lien de base "Expansion", R le méta-lien de base "Référence", C le méta-lien "Clientèle" et H le méta-lien "Héritage" [3]. Nous pouvons alors écrire : $Ofl = \oplus(E, R)$, $Ofl1 = \oplus(E, R, C)$ et $EIFFEL = \oplus(E, R, C, H)$.

Mais cette notation pose *a priori* un problème : le travail effectué dans O^{fl} n'est pas repris dans $Ofl1$, il faut tout recommencer. Pour résoudre ce problème, nous déclarons que l'opérateur $\oplus$ est associatif et que son résultat est du même type que ses opérandes. Nous pouvons donc désormais écrire les égalités suivantes : $Ofl = \oplus(E, R)$, $Ofl1 = \oplus(\oplus(E, R), C)$ et $EIFFEL = \oplus(\oplus(\oplus(E, R), C), H)$.

Par extension nous arrivons à la notation $Ofl = \oplus(E, R)$, $Ofl1 = \oplus(Ofl, C)$ et $EIFFEL = \oplus(Ofl1, H)$

Il est à noter que l'opérateur $\oplus$ n'a d'existence que du point de vue modélisation. Du point de vue de l'exécution, il est mis en œuvre par les opérateurs de composition des différentes clauses des méta-liens (Visibilité, Exécution, Adaptation de type et Collection).

5.7.2 Composition d'opérateurs génériques et exemples

Un méta-lien est, comme nous l'avons vu, structuré en plusieurs clauses. Il nous faut donc, si l'on ne veut pas rester trop général et imprécis, préciser ce qui se passe pour les éléments de chacune de ces clauses lors de la composition de méta-liens. Précisons tout d'abord qu'il s'agit d'une proposition et non d'une décision ferme. En effet, la composition est un des points clefs les plus importants de notre étude et elle est encore sujette à modifications.

Nous nous intéressons ici à la composition des éléments de la clause "Exécution" : les opérateurs génériques. Nous noterons $\odot$ l'opérateur de composition des opérateurs génériques.

Soit AE l'opérateur d'affectation du méta-lien "Expansion" et AR celui du méta-lien "Référence". Nous pouvons écrire : $AOfl = \odot(AE, AR)$ avec $AOfl$ l'opérateur (issu de la composition) de la méta-vue O^{fl}.

L'opérateur $\odot$ présente les mêmes caractéristiques que l'opérateur $\oplus$: il est associatif et son résultat est du même type que ses opérandes. Pour être plus précis, en voici une description globale. $\odot$ représente une fonction logique (ou, et, non, ...) simple. Ainsi nous pouvons préciser : $AOfl = \text{ou}(AE, AR)$ ce qui exprime que l'affectation et celle d'"Expansion" ou de "Référence" selon la nature de ses arguments.

Cette description logique de la composition des opérateurs génériques est simple mais il n'est pas sûr qu'elle soit toujours adaptée. Aussi l'associerons-nous peut-être au mécanisme des méthodes auxiliaires présenté dans le chapitre précédent (section 4.3.5 page 16). Ainsi, lors de la

3. Les méta-liens "Clientèle" et "Héritage" cités s'entendent comme "Clientèle à la *Eiffel*" et "Héritage à la *Eiffel*"

composition, on pourra décrire aussi bien des opérations logiques que des opérations temporelles ce qui offrira une plus grande souplesse.

Cette denière option n'est pas définitive. En effet, les exemples que nous présentons dans la section 5.7.2 page 24 pourraient être entièrement résolus en remplaçant les méthodes auxiliaires par des opérateurs logiques (en l'occurrence un *andthen* pourrait aisément remplacer les pré-méthodes). Mais le fait de donner plusieurs exemples n'ayant jamais rien démontré, l'utilité de l'usage des méthodes auxiliaires n'est pas totalement écarté.

Cette dernière section présente maintenant la clause "Exécution" de quatre méta-liens qui pourraient être ajoutés à O^{fl}. Nous donnons non seulement la description des opérateurs génériques mais aussi des indices sur la manière dont ils pourraient être composés avec O^{fl}. Nous précisons que le symbole $\mathcal{N}$ représente pour nous l'élément neutre de la fonction de composition d'opérateurs génériques $\odot$.

Méta-lien "Clientèle"

Le méta-lien "Clientèle" que nous présentons est celui d'*Eiffel 3*. Nous pouvons remarquer que ce méta-lien n'apporte rien aux méta-liens de base en ce qui concerne les opérateurs génériques. En effet, la modification se fait au niveau de la clause "Visibilité" du méta-lien et non au niveau de la clause "Exécution". Ceci est un choix que nous aurions pu remplacer par la redéfinition de l'opérateur "." dans la clause "Exécution". La composition de ces opérateurs avec ceux de la méta-vue O^{fl} est donc évidente: aucune modification n'est apportée à ceux de O^{fl}.

L'envoi de message "." : Objet.Primitive

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

La création "!!" :![Type]!Objet[.Initialiseur]

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

Méta-lien "Protection"

Le méta-lien que nous appelons "Protection" [Lah92] est celui qui permet de définir des droits d'accès sur les primitives d'un objet. Cela rejoint assez bien la notion de sélection dans le domaine des systèmes de gestion de bases de données. Ici la composition est principalement intéressante du point de vue des préconditions et postconditions. En effet, ce méta-lien n'apporte rien au niveau de la sémantique des rôles des opérateurs génériques. On peut assez intuitivement déclarer que la composition consistera en une conjonction des préconditions et postconditions respectives des différents opérateurs. Par exemple la précondition de l'opérateur d'affectation de la méta-vue issue de la composition de "Ofl" avec "Protection" sera: Précondition de l'affectation de "Ofl" *et* Précondition de l'affectation de "Protection". Le lecteur pourra remarquer que nous utilisons les

notions d'exécutabilité, d'accessibilité et d'autorisation. Ces concepts pourraient être mis en place grâce à une extension des fonctionnalités de la clause "Visibilité" du méta-lien.

L'envoi de message "." : Objet.Primitive

Précondition : Primitive est exécutable[4]

Rôle : $\mathcal{N}$

Postcondition : Vrai

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : Objet.Champ accessible en écriture

Rôle : $\mathcal{N}$

Postcondition : Vrai

La création "!!" :![Type]!Objet[.Initialiseur]

Précondition : Création autorisée sur le type Type (si spécifié) ou sur le type de déclaration de Objet

Rôle : $\mathcal{N}$

Postcondition : Vrai

Méta-lien "Héritage"

Le méta-lien "Héritage" que nous présentons est celui d'*Eiffel 3*. Ici nous trouvons encore un exemple de composition différent. Du point de vue de l'exécution, l'héritage se résume au problème du polymorphisme. Ce mécanisme permet de mettre en œuvre un processus de choix de la primitive à déclencher. Nous le formalisons comme un recherche de la bonne primitive suivie d'une exécution de celle-ci. L'exécution est réalisée par l'envoi de message du méta-lien "Ofl", reste donc à la faire précéder du choix de la primitive. Nous résolvons ce problème en déclarant que le rôle de l'envoi de message du méta-lien "Héritage" constitue un pré-méthode de l'envoi de message du méta-lien "Ofl". Ainsi, on réalise les opérations dans le bon ordre. Rappelons que dans ce cas, l'usage d'une pré-méthode pourrait être remplacé par un opérateur logique de composition du type *andthen*.

L'envoi de message "." : Objet.Primitive

Précondition : Vrai

Rôle : Choix de la Primitive : polymorphisme

Postcondition : Vrai

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

La création "!!" :![Type]!Objet[.Initialiseur]

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

4. Si la Primitive est un champ, alors nous vérifions qu'il est lisible ; si c'est une méthode, alors nous contrôlons qu'elle est exécutable.

Méta-lien "Version"

Le méta-lien "Version" [Lah92, Zdo86, BDN88, BK87] permet de gérer plusieurs versions d'un objet. Cela signifie que selon qu'on s'adresse à une version ou à une autre l'objet pourra avoir des réactions différentes. Cette sélection s'opère de la même manière que le polymorphisme dans "Héritage" et peut donc être résolu de façon identique.

L'envoi de message "." : Objet.Primitive

Précondition : Vrai

Rôle : Choix de la Primitive

Postcondition : Vrai

L'affectation ":=" : Objet.Champ := AutreObjet

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

La création "!!" : ![Type]!Objet[.Initialiseur]

Précondition : Vrai

Rôle : $\mathcal{N}$

Postcondition : Vrai

Chapitre 6

Ébauche du noyau du système

6.1 Introduction

Ce chapitre décrit succinctement les *primitives* essentielles de l'environnement O^{fl}. Il ne s'agit pas ici de présenter une liste exhaustive et stable de ces primitives mais seulement de donner au lecteur une idée de celles que nous pensons devoir y intégrer. Ce chapitre ne doit donc pas être pris comme une proposition mais plutôt, comme le déclare son titre, comme une ébauche de la définition du noyau de notre système de vues. Le lecteur intéressé par cette partie pourra se référer avec profit aux travaux réalisés au sein de l'équipe O.C.L sur *IREC* [RBC+95] et à l'article [CCL96].

Nous sommes restés jusque là et resterons totalement indépendants des problèmes de compilation et interprétation. Ce chapitre entame toutefois une première approche de ces problèmes qui devront être étudiés par la suite.

6.2 Primitives de gestion des vues

Ces primitives décrivent la gestion de la structure de données de base d'O^{fl} : les vues. Ces primitives sont étrangères à celles qui pourraient exister dans *Eiffel 3*.

- CréerVueGénérique(NomVue, ParametresGeneriques) : Elle permet de créer une vue générique.
- CréerVue(NomVue) : Elle permet de créer une vue non générique. Bien que non indispensable (Il est possible de créer un vue non générique en utilisant CréerVueGénérique sans paramètre de généricité.), elle existe par souci d'orthogonalité et de simplicité.
- AjouterChamp(NomVue, NomChamp, TypeChamp) : Elle permet de créer et d'ajouter un attribut d'instance dans une vue.
- AjouterChampVue(NomVue, NomChamp, TypeChamp) : Elle permet d'ajouter un attribut de vue (partagé) dans une vue.
- RetirerChamp(NomVue, NomChamp) : Elle permet de retirer un attribut d'une vue.
- AjouterMéthode(NomVue, NomMéthode, Précondition, Local, Corps, Postcondition) : Elle permet d'ajouter une méthode d'instance dans une vue.
- AjouterMéthodeVue(NomVue, NomMéthode, Précondition, Local, Corps, Postcondition) : Elle permet d'ajouter une méthode de vue (partagée) dans une vue.
- PréciserMéthode(NomVue, NomMéthode, Précondition, Local, Corps, Postcondition) : Elle permet de redéfinir une méthode d'instance dans une vue.

- PréciserMéthodeVue(NomVue, NomMéthode, Précondition, Local, Corps, Postcondition) : Elle permet de redéfinir une méthode de vue (partagée) dans une vue.
- RetirerMéthode(NomVue, NomMéthode) : Elle permet de retirer une méthode d'une vue.
- RetirerCréateur(NomVue, NomCréateur) : Elle permet de retirer une méthode de création d'instance d'une vue.
- AjouterIndexationVue(NomVue, NomIndexation, Indexation) : Elle permet d'ajouter une clause dans la section d'indexation d'une vue (ou de créer un attribut y correspondant).
- RetirerIndexationVue(NomVue, NomIndexation) : Elle permet de de retirer une clause de la section d'indexation d'une vue (ou de retirer l'attribut correspondant).
- AjouterInvariantVue(NomVue, Condition) : Elle permet d'ajouter une condition à l'invariant d'une vue.
- VérifierInvariantVue(NomVue, Condition) : Elle permet de générer le code de vérification de l'invariant de vue.
- RetirerInvariantVue(NomVue, Condition) : Elle permet de retirer une condition de l'invariant d'une vue.

Une vue est un objet, elle est donc automatiquement détruite si elle n'est plus référencée. Il est donc inutile de prévoir une primitive de destruction.

6.3 Primitives de gestion d'instances

- CréerInstanceGénérique(NomVue, Créateur, ParametresGeneriques) : Elle permet de créer une instance d'une vue générique.
- CréerInstance(NomVue, Créateur) : Elle permet de créer une instance d'une vue non générique. Bien que non indispensable (Il est possible de créer un instance d'une vue non générique en utilisant CréerInstanceGénérique sans paramètre de généricité.), elle existe par souci d'orthogonalité et de simplicité.
- AffecterChamp(NomVue, Champ, Valeur) : Elle permet d'affecter une valeur à un attribut d'instance.
- AffecterChampVue(NomVue, Champ, Valeur) : Elle permet d'affecter une valeur à un attribut de vue.
- LireChamp(NomVue, Champ) : Elle permet de lire la valeur d'un attribut.
- ExécuterMéthode(NomVue, NomMéthode) : Elle permet d'exécuter une méthode.

6.4 Exemple

Pour donner corps à ces définitions, nous allons donner un premier aperçu de ce que pourrait être la traduction d'un programme écrit en O^{fl} (toujours en le décrivant avec une syntaxe "à la *Eiffel 3*", ce qui ne restreint rien) en ses primitives.

Voici un extrait d'un texte source d'une vue jouet avec sa traduction (l'intégralité du texte source et de la traduction sont donnés dans l'annexe A page 31). Les primitives décrites correspondront aux opérations activées pendant la *vie* d'une application O^{fl}.

view PERSON
...

```
feature

  p_name : STRING;
  f_name : STRING;
...
  create(n : STRING ; p : STRING ; a : INTEGER ; s : CHARACTER ; v : TOWN) is
  do
    p_name := n.duplicate;
    f_name := p.duplicate;
    age := a;
    sex := s;
    town := v;
  end; -- create

  SetSpouse(p : PERSON) is
  do
    spouse := p; p.spouse := current;
  end; -- SetSpouse
...
end -- feature
```

Voici la traduction :

```
CréerVue("PERSON");
AjouterChamp("PERSON", "p_name", STRING);
AjouterChamp("PERSON", "f_name", STRING);
...
AjouterCréateur("PERSON", "create", (n : STRING ; p : STRING ; a : INTEGER ;
  s : CHARACTER ; v : TOWN), (), (),(p_name := n.duplicate ;
  f_name := p.duplicate ; age := a ; sex := s ; town := v), ());
AjouterMéthode("PERSON", "SetSpouse", ( p : PERSON), (), (), (spouse := p ;
  p.spouse := current), ());
...
```

Chapitre 7

Conclusion

Ce stage, effectué au laboratoire I.3S. aura été l'occasion de découvrir le monde de la Recherche et d'effectuer un travail très intéressant et enrichissant qui nous l'espérons pourra être poursuivi. Un article [CCL96] sur le sujet sera dans ce but soumis à parution durant cet été.

Nous conclurons ce rapport selon deux axes. Tout d'abord, nous donnerons une vue des principales différences entre notre système de vues et *CLOS*. Cela n'a pas qu'un but informatif car le rappel ou la mise-à-jour de ces différences permettront au lecteur de mieux recentrer notre travail par rapport à un langage relativement connu. La seconde partie constitue une définition des perspectives d'avenir que nous voyons pour ce projet.

Voyons donc quelles sont les différences entre notre système de vues et *CLOS*.

- *CLOS* est basé sur le lien d'héritage alors que notre système de vues ne l'utilise pas.
- *MOP* n'introduit pas directement le concept de lien qui est un *plus* apporté par notre modélisation.
- Dans *CLOS*, les méthodes ne sont pas encapsulées et les fonctions sont des objets de premier niveau, ce qui n'est pas le cas pour nous.
- *CLOS* n'est pas un langage de génie logiciel, alors que nous avons toujours souhaité qu'O^{fl} le soit.
- *MOP* repose sur *Common LISP* alors que *VMP* repose sur O^{fl} (et en partie sur *Eiffel 3*), ce qui induit deux manières de programmer totalement différentes.

Enfin, nous dirons, avant de passer aux perspectives, que, globalement, ce travail peut être considéré comme la phase préliminaire (l'étude de faisabilité) d'un projet visant à réaliser un prototype d'environnement de programmation par vues, extensible, tourné vers le génie logiciel et basé sur *Eiffel 3*.

L'objectif de cette étude a été de poser les problèmes, de montrer les besoins, de considérer l'existant et enfin de proposer une modélisation offrant des perspectives intéressantes. La poursuite de ce travail pourra être réalisée suivant plusieurs axes.

Tout d'abord, en ce qui concerne le langage de base O^{fl}, il reste quelques précisions à donner concernant en particulier les aspects syntaxiques. Cela constitue un premier objectif.

Ensuite, nous nous intéresserons à l'implantation de notre système de vues. Pour cela, il conviendra de compléter la description du noyau du moteur qui est actuellement à l'état d'ébauche et d'en assurer la mise en œuvre après avoir approfondi les aspects statiques et dynamiques.

La réalisation de ce deuxième objectif permettra, en particulier, de parfaire les choix de modélisation et en initialisera un troisième : l'écriture d'un langage à la *Eiffel 3*, basé sur notre modèle, qui constituera la validation pratique de notre modélisation.

Annexe A

Exemple d'une vue en O^{fl}

A.1 Texte source

view PERSON

– *La vue PERSON décrite avec le langage de base :*
– ** Utilisation des vues "noyau" STRING, INTEGER et CHARACTER pour définir des*
– *attributs,*
– ** Utilisation de la vue "application" TOWN,*
– ** Navigation libre (cf. SetSpouse et AddChild).*

```
feature

  p_name : STRING;
  f_name : STRING;
  age : INTEGER;
  sex : CHARACTER;
  town : TOWN;
  spouse : PERSON;
  children : LINKED_LIST [PERSON];

  create(n : STRING ; p : STRING ; a : INTEGER ; s : CHARACTER ; v : TOWN) is
  do
    p_name := n.duplicate;
    f_name := p.duplicate;
    age := a;
    sex := s;
    town := v;
  end; – create

  SetSpouse(p : PERSON) is
  do
    spouse := p; p.spouse := current;
  end; – SetSpouse

  AddChild(p : PERSON) is
  local
    l_children : LINKED_LIST [PERSON];
  do
```

```
    if
      children.void
    then
      children.create;
    end; -- if
    children.start;
    children.put_right(p);
    if
      not spouse.void
    then
      spouse.children := children;
    end; -- if
  end; -- AddChild

  DisplayNames is
  local
    str : STRING;
  do
    str := " ";
    f_name.print;
    str.print;
    p_name.print;
  end; -- DisplayNames

  DisplayTown is
  local
    str : STRING;
  do
    str := " ";
    town.town_name.print;
    str.print;
    town.nb_inhabitants.print;
  end; -- DisplayTown

end -- feature
```

A.2 Traduction

```
CréerVue("PERSON");
AjouterChamp("PERSON", "p_name", STRING);
AjouterChamp("PERSON", "f_name", STRING);
AjouterChamp("PERSON", "age", INTEGER);
AjouterChamp("PERSON", "sex", CHARACTER);
AjouterChamp("PERSON", "town", TOWN);
AjouterChamp("PERSON", "spouse", PERSON);
AjouterChamp("PERSON", "children", LINKED_LIST [PERSON]);
AjouterCréateur("PERSON", "create", (n : STRING ; p : STRING ; a : INTEGER ;
  s : CHARACTER ; v : TOWN), (), (),(p_name := n.duplicate ;
  f_name := p.duplicate ; age := a ; sex := s ; town := v), ());
AjouterMéthode("PERSON", "SetSpouse", ( p : PERSON), (), (), (spouse := p ;
  p.spouse := current), ());
AjouterMéthode("PERSON", "AddChild", (p : PERSON), (),
```

```
  (l_children: LINKED_LIST [PERSON]), (if children.void then children.create;
  end; children.start; children.put_right(p); if not spouse.void then
  spouse.children:= children; end), ());
AjouterMéthode("PERSON", "DisplayNames", (), (), (str: STRING), (str:= " ";
  f_name.print; str.print; p_name.print), ());
AjouterMéthode("PERSON", "DisplayTown", (), (), (str: STRING), (str:= " ";
  town.town_name.print; str.print; town.nb_inhabitants.print), ());
```

Bibliographie

[AKDM+94] Aït-Kaci (Hassan), Dumant (Bruno), Meyer (Richard), Podelski (Andreas) et Roy (Peter Van). – *The* Wild LIFE *Handbook*. – Paris Research Laboratory, `digital`, Mars 1994. Prépublication.

[BDN88] Benzaken (V.), Delobel (C.) et Ndala (J.B.). – Gestionnaires de mémoire et d'objets. *In : Quatrième journée bases de données avancées*. – Benodet, BD3, Mai 1988.

[BK87] Banerjee (J.) et Kim (W.). – Semantics and implementation of schema evolution in object-oriented databases. *In : International conference on management of data*, p. 311 à 322. – San Francisco, Californie, ACM SIGMOD, Mai 1987.

[BK93] Barclay (Peter J.) et Kennedy (Jessie B.). – Viewing Objects. *In : Proceedings of Advances in Databases*, p. 93 à 110. – Keele, Royaume Uni, 11^{th} British National Conference In Databases, juillet 1993.

[Bou94] Bourdin (Cyril). – *Points de vue et représentation multiple et évolutive dans les langages à objets*. – Rapport de stage de D.E.A. d'Informatique, Université de Montpellier II, juin 1994.

[Bro95] Browne (Roger). – *"comp.lang.eiffel" : Frequently Asked Questions*. – Everything Eiffel, avril 1995.

[CCL96] Chignoli (Robert), Crescenzo (Pierre) et Lahire (Philippe). – O^{fl} : Un environnement extensible pour gérer des vues en *Eiffel*. – Avril 1996. Soumis à *Fourth International Conference on Software Reuse*.

[CFLR93a] Chignoli (Robert), Farré (Jacques), Lahire (Philippe) et Rousseau (Roger). – *FLOO* : Principes et Illustration des mécanismes automatiques de persistance pour *Eiffel*. *In : 6th International Conference on Software Engineering and its Applications*, p. 181 à 190. – Paris-La Défense, Kluwer Academic Publishers, Novembre 1993.

[CFLR93b] Chignoli (Robert), Farré (Jacques), Lahire (Philippe) et Rousseau (Roger). – *FLOO* : Une implémentation de la persitance pour *Eiffel*. *In : RPO'93 - Représentation par Objets*, p. 131 à 142. – La Grande Motte, EC2 (Paris), Juin 1993.

[CK94] Chan (Daniel K. C.) et Kerr (David K.). – Improving One's Views of Object-Oriented Databases. *In : Actes du Colloque "Orientation objet en bases de données et génie logiciel"*, p. 123 à 137. – Montréal, Canada, Soixante-deuxième Congrès de l'Association Canadienne Française pour l'Avancement des Sciences, mai 1994.

[CP94] Crescenzo (Pierre) et Paquelin (Jean-Louis). – Scheme et les continuations. – Juillet 1994. Rapport de conférences.

[Cre95] Crescenzo (Pierre). – Un système de vues pour *Eiffel*. – Université de Nice-Sophia-Antipolis, avril 1995. Rapport intermédiaire de stage.

[Dal95] Dalmas (Stéphane). – Cours de "Programmation Fonctionnelle Avancée". – Cours de D.E.A. d'Informatique, 1994-1995.

[FC95] Fornarino (Mireille) et Cointe (Pierre). – Cours de "Fondements de la Programmation par Objets". – 1994-1995. Cours de D.E.A. d'Informatique.

[KdRB91] Kiczales (Gregor), des Rivières (Jim) et Bobrow (Daniel G.). – *The Art of the Metaobject Protocol.* – Cambridge, Massachusetts ; London, England, The MIT Press, 1991.

[Kee89] Keene (Sonya E.). – *Object-Oriented Programming in COMMON LISP : A Programmer's Guide to CLOS.* – Reading, Massachussetts ; Menlo Park, California, Addison-Wesley, 1989.

[Lah92] Lahire (Philippe). – *Conception et réalisation d'un modèle de persistance pour le langage Eiffel.* – Laboratoire I.3S., Sophia-Antipolis, Thèse de doctorat en informatique, Université de Nice-Sophia-Antipolis, Mai 1992.

[MEC95] Marcaillou-Ebersold (S.) et Coulette (B.). – Extension du modèle objet par introduction de la visibilité. – ENSEEIHT, Toulouse, Mai 1995.

[Mey90] Meyer (Bertrand). – *Conception et programmation par objets pour du logiciel de qualité.* – Paris, InterÉditions, 1990.

[Mey92] Meyer (Bertrand). – *Introduction à la théorie des langages de programmation.* – Paris, InterÉditions, 1992.

[Mey94] Meyer (Bertrand). – *Eiffel, le langage.* – Paris, InterÉditions, 1994.

[Mic94] Michaudel (Éric). – *Un système de vues pour FLOO : Modélisation et application aux protections.* – Université de Nice-Sophia-Antipolis, juin 1994.

[MKC95] Marcaillou (S.), Kriouile (A.) et Coulette (B.). – *VBOOL*, une extension d'*Eiffel* intégrant le concept de point de vue. – ARAMIIHS, UMR 115 du CNRS, Toulouse, Mai 1995.

[MNC+91] Masini (Gérald), Napoli (Amedeo), Colnet (Dominique), Léonard (Daniel) et Tombre (Karl). – *Les langages à objets : langages de classes, langages de frames, langages d'acteurs.* – InterÉditions, 1991.

[RBC+95] Rousseau (Roger), Boussard (Jean-Claude), Chignoli (Robert), Collet (Philippe), Ducasse (Stéphane), Gallésio (Éric) et Lahire (Philippe). – Cours de "Approches Orientées Objets pour le Génie Logiciel, les Bases de Données et la Conception Asssitée par Ordinateur : vers un modèle commun". – 1994-1995. Cours de D.E.A. d'Informatique.

[SF90] Springer (George) et Friedman (Daniel P.). – *Scheme and the Art of Programming.* – The MIT Press, 1990.

[Swi95] Switzer (Robert). – *Introduction à Eiffel.* – Paris, Masson, 1995, *Méthodologies du logiciel.*

[Zdo86] Zdonik (S.B.). – Maintening consistency in a database with changing types. *In : Object-oriented programming workshop.* – Yorkton Heights, NY, SIGPLAN Notice (USA), Juin 1986.