HAL
archives-ouvertes.fr

Shood : un modéle méta-circulaire de représentation de connaissances

Jose Guadalupe Escamilla de los Santos

▶ To cite this version:

Jose Guadalupe Escamilla de los Santos. Shood : un modéle méta-circulaire de représentation de connaissances. Interface homme-machine [cs.HC]. Institut National Polytechnique de Grenoble - INPG, 1993. Français. tel-00005126

HAL Id: tel-00005126
https://tel.archives-ouvertes.fr/tel-00005126

Submitted on 26 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par
ESCAMILLA DE LOS SANTOS José Guadalupe

pour obtenir le grade de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(arrêté ministériel du 30 mars 1992)

(Spécialité: Informatique)

==========

Shood: un modèle méta-circulaire de représentation de connaissances.

==========

Date de soutenance: septembre 1993

Composition du jury:	Président	Y. CHIARAMELLA
	Rapporteurs	A. NICOLLE
		C. ROCHE
	Examinateurs	T. NGUYEN
		R. MOHR

Thèse préparée au sein du Laboratoire de Génie Informatique.

A Carmen y Enrique.

Je tiens à remercier:

Monsieur Yves Chiaramella, Professeur à l'Université Joseph Fourier de Grenoble, de m'avoir fait l'honneur de présider le jury de cette thèse.

Madame Anne Nicolle, Professeur à l'Université de Caen, d'avoir acceptée d'être rapporteur de ma thèse. Je la remercie également pour l'intérêt qu'elle lui a portée et pour l'évaluation qu'elle en a faite.

Monsieur Christophe Roche, Professeur à l'Université de Savoie d'avoir accepté d'être raporteur de ce travail et pour sa lecture détaillée de mon memoire.

Monsieur Roger Mohr, professeur à l'Institut National Polytechnique de Grenoble, d'avoir bien voulu participer au jury de cette thèse et pour ses remarques pertinentes.

Nguyen Gia Toan, Directeur de Recherche INRIA et Docteur d'état, de m'avoir accepté dans son équipe et dirigé ma thèse. Je le remercie aussi pour tout le temps qu'il a bien voulu me consacrer.

Dominique Rieu pour son dynamisme contagieux, ses commentaires et ses critiques toujours justes. Je tiens à lui témoigner toute ma reconnaissance pour le soutien qu'elle m'a toujours manifesté.

François Rechenmann, Directeur de Recherche INRIA et responsable du projet Sherpa, commun à l'INRIA et à l'IMAG, pour m'avoir accepté dans son projet. Je le remercie également pour ses conseils qui m'ont aidé à finir cette thèse.

Je remercie le Conacyt (gouvernement mexicain), le CEFI (gouvernement français) et l'Instituto Tecnológico y de Estudios Superiores de Monterrey (Mexique) pour avoir apporté le soutien financier nécessaire au bon déroulement de cette thèse.

Je tiens à exprimer toute ma gratitude aux membres du projet Sherpa. A Jeff pour son sourire et ses corrections "théatrales". A Annie pour ses corrections toujours sur le bon "ton". A Jérôme pour la lecture attentive qu'il a faite du manuscrit. Je remercie également les autres membres de l'équipe Shood pour avoir fait régner une bonne ambiance dans l'équipe: Fethi, Chabane, Pierre ainsi que les "anciens" Valérie, Marie-Pierre, Miguel et David. Je remercie aussi les physiciens du projet du GSIP, spécialement Damien et Michel.

Je voudrais dire un grand merci à tous mes amis pour avoir fait de ces cinq ans en France une période "très zéspéciale".

A la comadre Nájera, Pierre, Olga, Pascal, Denis, Carlos, Arnie, Bixentxo et Sonia pour avoir accepté la lourde tâche d'être ma famille adoptive. A mes amis de la colonie Colombienne des "Andes": Rubby "la CasaLas", Rodrigo, Harold, Mary et Coquita pour toute la chaleur et l'amitié qu'ils m'ont offertes. A Horst, Johannes, Patrice, José Luis, Carmen, Frédéric, Arturo, Jérôme "Smiths", Pablo, Estela, Manolo, Ramón, Hugo, Fernando, Salvador, au Scénario et à tous ceux qui ont fait que cette période de ma vie soit aussi belle. A mes amis mexicains au Mexique et à ceux à l'étranger: Arturo, Enrique, Carlos et Silvia pour avoir continué à me témoigner leur amitié malgré la distance. Muchas gracias a todos.

A mon père pour m'avoir toujours encouragé par l'exemple à aller plus loin et, à ma mère pour avoir toujours supporté mes décisions. A ma "frangine" Lety et mes "frangins" Eduardo et Juán pour leur soutien pendant toutes ces années.

Table des matières

Introduction **1**
Cadre général du problème 1
Objectifs et motivations 1
Puissance de représentation 1
L'intégration des outils de calcul 4
Evolution des besoins 6
Contribution 8
Plan de la thèse 8

1. Les systèmes à Objets **11**
1.1 Introduction 11
1.2 Les aspects déclaratifs 12
1.2.1 Le partage structurel 12
1.2.1.1 L'approche ensemble-héritage 12
1.2.1.2 Une alternative: Prototype-Délégation 14
1.2.1.3 Réflexions 16
1.2.2 L'héritage multiple 17
1.2.2.1 Conflits d'héritage 17
1.2.2.2 La subsomption 18
1.2.2.3 Réflexions 20
1.2.3 Le traitement des exceptions 20
1.2.3.1 Réflexions 21
1.2.4 Les points de vue 21
1.2.4.1 Réflexions 23
1.2.5 L'accès aux objets 24
1.2.5.1 Encapsuler ou non? 24
1.2.5.2 Réflexions 26
1.2.6 Les attributs 26
1.2.6.1 Les connaissances déclaratives sur les attributs 27
1.2.6.2 Les connaissances procédurales sur les attributs 28
1.2.6.3 Réflexions 29
1.2.7 Récapitulatif des aspects déclaratifs 30
1.3 Les aspects procéduraux 31

1.3.1 Sélectionner la méthode la plus appropriée.....31
1.3.1.1 L'envoi de messages.....31
1.3.1.2 Les fonctions génériques.....33
1.3.2 La réutilisation des méthodes.....35
1.3.3 Réflexions.....37
1.3.4 Récapitulatif des aspects procéduraux.....38
1.4 L'extensibilité.....38
1.4.1 La méta-connaissance.....39
1.4.2 La réflexivité: les méta-classes.....39
1.4.3 L'uniformisation: le tout objet.....41
1.4.4 Réflexions.....42
1.4.5 Récapitulatif des aspects réflexifs.....43
1.5 Conclusions.....43

2. Shood: Les aspects déclaratifs.....45
2.1. Introduction.....45
2.2. Les concepts de base.....46
2.2.1. Classes, attributs et descripteurs.....46
2.2.2. La réutilisation: la spécialisation et l'héritage.....47
2.2.3. Plus de réutilisation: la spécialisation multiple.....48
2.2.4. L'instanciation.....49
2.2.5. L'identification des objets.....50
2.2.6. L'instanciation multiple.....50
2.2.7. La disjonction.....51
2.2.8. Récapitulatif.....52
2.3. Vers l'extensibilité.....53
2.3.1. Le niveau méta.....53
2.3.2. Le graphe de spécialisation.....55
2.4. Les conflits d'attributs.....56
2.4.1. Les noms des attributs.....56
2.4.2. Les attributs pré-déclarés.....59
2.4.3. Le type des attributs.....60
2.4.3.1. Les constructeurs.....61
2.4.3.2. La relation de sous-type.....61
2.5. Plus sur les attributs.....62
2.5.1. Les attributs des attributs.....62
2.5.2. Les attributs obligatoires.....63
2.5.3. Récapitulatif.....64
2.6. L'héritage, définition formelle.....64

2.6.1. Le mécanisme d'héritage de Shood 64
2.6.2. Les invariants dans l'évolution des schémas 64
2.6.2.1. Invariants du graphe d'héritage 65
2.6.2.2. Invariants des attributs 65
2.6.2.3. Invariants des descripteurs 66
2.6.2.4. Invariants des clefs 66
2.6.2.5. Invariant de Méta-classe 67
2.6.3. Principes de l'héritage 67
2.7. L'instanciation, définition formelle 69
2.7.1. Invariant du graphe d'instanciation 69
2.7.2. Invariants des attributs 69
2.8. Conclusions 71

3. Shood: aspects procéduraux 73
3.1. Introduction 73
3.2. Les méthodes 74
3.2.1. Les méthodes sont des classes 74
3.2.2. Spécialiseurs des arguments 75
3.2.3. Arguments obligatoires et optionnels 76
3.2.4. Sélection de méthodes 77
3.2.4.1. L'ordre des arguments 77
3.2.4.2. Sélection par classification 77
3.2.4.3. Exemple mécanique de multi-instanciation des méthodes 79
3.2.5. Spécialisation déclarative 81
3.2.5.1. Les traitements 82
3.2.5.2. Les traitements indépendants 83
3.2.5.3. Les traitements dépendants 84
3.2.5.4. Les contrôles de séquentialisation 85
3.2.5.5. Séquentialisation des traitements dépendants 86
3.2.5.6. Héritage dynamique des traitements 88
3.2.5.7. Récapitulatif 88
3.2.6. Spécialisation procédurale 89
3.2.7. Récapitulatif 91
3.3. Les inférences 92
3.3.1. Définition d'une inférence 93
3.3.2. Les arguments d'une inférence 93
3.3.3. Un exemple 94
3.4. Les contraintes 94
3.4.1. Un exemple 95

3.5. Conclusion....96

4. L'extensibilité....97

4.1. Introduction....97
4.2. Premier aperçu de Creer_objet....98
4.3. META: la définition minimale d'une classe....99
4.3.1. Comportement....101
4.4. Une extension: Les classes à clef....101
4.4.1. Définition des clefs....101
4.4.2. Représentation....102
4.4.3. Comportement....104
4.5. Modélisation des attributs....105
4.5.1. Représentation....105
4.5.1.1. Les classes-attributs....105
4.5.1.2. Spécialisation des attributs....106
4.5.1.3. Les classes-constructeurs....106
4.5.1.4. META_ATTRIBUT....108
4.5.2. Comportement....109
4.6. Les attributs des attributs....109
4.6.1. Représentation....109
4.6.2. Comportement....111
4.7. Les attributs obligatoires....111
4.7.1. Représentation....111
4.7.2. Comportement....112
4.8. La modélisation des méthodes....112
4.8.1. Représentation....112
4.8.2. Le comportement des méthodes....113
4.9. Modélisation des inférences....114
4.9.1. Représentation....114
4.9.2. Les arguments des inférences....114
4.9.3. Comportement des inférences....115
4.10. Modélisation des contraintes....115
4.10.1. Représentation....115
4.10.2. Les arguments des contraintes....116
4.10.3. Comportement des contraintes....116
4.11. L'amorce....116
4.11.1. Le graphe d'instanciation....117
4.11.2. Le graphe de spécialisation....117
4.11.3. Les classes-attributs de l'amorce....118

4.12. Récapitulatif des méta-classes et de Creer_objet 119
4.13. Conclusion 120

Conclusion 121

Bilan 121

Les aspects déclaratifs 121

La dynamique des points de vue 122

La réutilisation des structures 122

La sémantique ensembliste du modèle 122

Le pouvoir déclaratif des attributs 122

Les aspects procéduraux 122

La modélisation des outils de calcul 122

La réutilisation des procédures 123

Inférences et cohérence 123

L'évolution des besoins 124

L'extensibilité 124

Dynamique des applications 124

Réalisation et utilisation 124

Méthodologie 125

Niveaux d'interface 125

Perspectives 126

Les interactions procédurales-déclaratives 126

La persistance 127

Les performances 127

Bibliographie 129

Introduction

Cadre général du problème

A l'origine de ce travail de thèse se trouvent des études de modélisation en Conception Assistée par Ordinateur (CAO) de circuits [RIE85] ainsi que des constatations faites lors des collaborations menées avec des mécaniciens et des électrotechniciens afin de modéliser la conception de produits. Ces études montrent que les systèmes de représentation de connaissances (SRC) existants ne sont pas complètement adaptés aux besoins de modélisation en CAO.

Pour mieux illustrer les problèmes rencontrés ainsi que les solutions proposées nous utiliserons un exemple mécanique développé par le Laboratoire d'Etudes et des Procédés de Fabrication à l'INSA de Lyon [RIE91]. L'exemple concerne la conception d'un évacuateur destiné au sauvetage des personnes ou des biens à partir, par exemple, de ponts ou de téléphériques.

Objectifs et motivations

L'objectif de cette thèse est de définir un système de représentation de connaissances (SRC) adapté aux caractéristiques des objets manipulés dans les applications de CAO. Ce modèle doit donc servir de support pour la représentation des produits à concevoir. Lors de la conception de produits, sont utilisés des méthodes de calcul et des outils qui doivent eux aussi être modélisés et gérés par le SRC.

Les problèmes motivant ce travail sont issus de la modélisation en CAO et s'avèrent être des problèmes généraux de représentation de connaissances. Ceux-ci peuvent être classifiés en trois grandes familles de besoins: puissance de représentation, intégration des outils existants et évolution des besoins.

Puissance de représentation

Il n'existe pas aujourd'hui de processus de conception unique car le choix du processus dépend du produit que l'on conçoit. Par exemple, un produit innovant n'est pas conçu de la même manière qu'un produit non-innovant. En effet, la conception d'un produit non-innovant, aussi appelée reconception, est un processus dont les étapes et les moyens à mettre en œuvre sont

déjà connus. Par contre, la conception d'un produit innovant est un processus incomplet et moins précis car beaucoup d'informations proviennent d'hypothèses émises tout au long de la conception.

Si l'on ne s'accorde pas sur le processus de conception à utiliser, il existe au contraire un certain consensus en ce qui concerne l'utilisation des domaines d'expertise. En effet, au cours de ce processus, l'objet est examiné selon plusieurs domaines d'expertise ou *points de vue* [TOL92] [DAV91]. Un évacuateur, par exemple, peut être examiné selon un point de vue mécanique ou un point de vue technologique. Notre modèle doit permettre la modélisation des différents domaines d'expertise, qu'ils correspondent à plusieurs experts ou à un seul expert adoptant différents points de vue.

La conception d'un objet peut être perçue comme une activité nécessitant diverses étapes de transformation: cahier des charges, décomposition fonctionnelle, schéma cinématique, etc. Chacune de ces étapes demande la description de l'objet dans le formalisme du point de vue considéré. La figure 1 présente la première description de notre évacuateur correspondant au cahier des charges. Elle montre aussi une première transformation de celui-ci en un formalisme mieux adapté à l'étude de la cinématique de l'évacuateur.

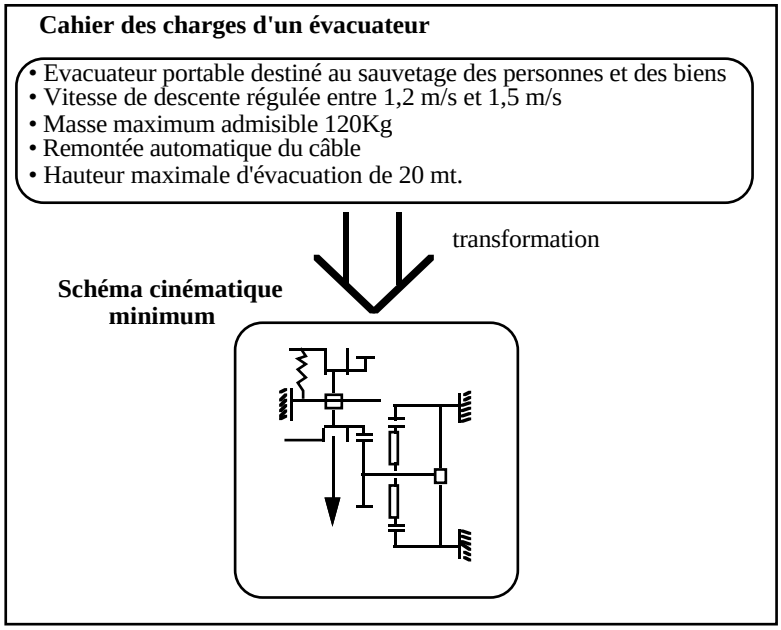


Figure 1. Transformation du cahier des charges en un schéma cinématique minimum.

Le passage d'un point de vue à l'autre peut être partiellement automatisé ou tout au moins assisté. Ceci est en partie dû au fait que les différents domaines d'expertise partagent des informations communes. Par exemple, les points de vue mécanique et technologique partagent des informations sur la forme des composants de l’évacuateur et les matériaux utilisés.

Finalement, les points de vue sur un objet peuvent être ajoutés dynamiquement. Par exemple, après le point de vue cinématique, l'évacuateur sera examiné du point de vue fonctionnel [CON92]. Notre modèle doit donc permettre la prise en compte **dynamique des points de vue**.

Un autre concept soulevé par la CAO concerne la réutilisation des objets. Par exemple, lors de la conception d'un produit, qu'il soit ou non innovant, des pièces existantes peuvent être utilisées. Ces pièces peuvent être soit des composants standards existant sur le marché, tels qu'une vis, soit des pièces préalablement fabriquées lors de conceptions antérieures. Nous devons permettre la réutilisation des *objets* existants.

Par ailleurs, la conception d'un nouveau produit peut réutiliser la description d'un produit déjà existant et ensuite l'ajuster en fonction du nouveau problème. Par exemple, l'introduction de l'esthétique du produit dans le cahier des charges peut amener à un changement superficiel comme sa couleur. Par contre, la structure de l'objet reste la même. Nous devons donc permettre la réutilisation des *structures*.

Finalement, l'expérience obtenue lors de la conception d'un objet doit aussi être réutilisable. Une méthode de calcul qui a donné de bons résultats lors de la conception d'un objet peut être adaptée et utilisée pour la conception d'un objet analogue. Nous devons permettre la réutilisation des *procédures*.

La figure 2 montre l'exemple de la conception d'un évacuateur permettant la descente de charges lourdes. Celle-ci est facilitée en utilisant l'expérience et les informations obtenues lors de la conception d'un évacuateur standard. Dans ce processus, dit de reconception, certains composants sont réutilisés (poulie), d'autres sont adaptés aux nouvelles données (câble).

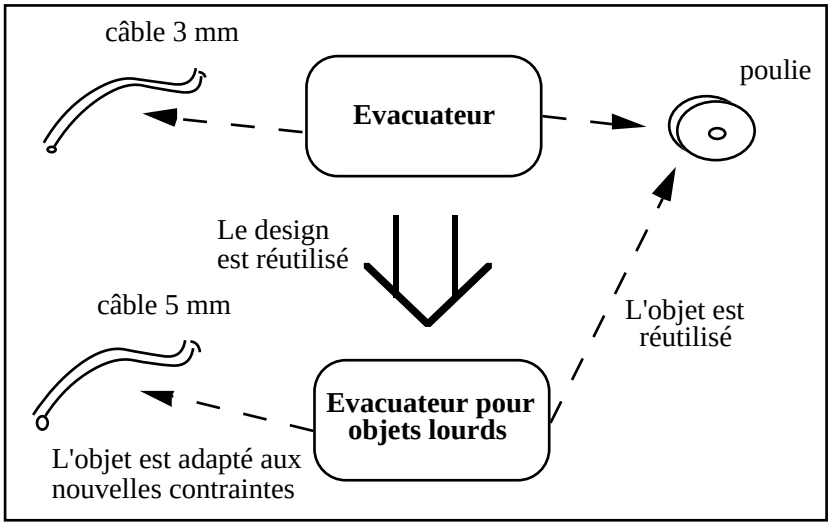


Figure 2. La réutilisation lors de la reconception.

De manière plus générale, notre modèle doit donc permettre la **réutilisation des concepts** (objets, structures et procédures).

Afin de gérer l'évolution des produits, notre modèle doit permettre la sélection de la structure la mieux adaptée à l'étape courante. Ce choix de structures se fait en fonction des informations disponibles sur le produit considéré. Un *mécanisme de classification* répond à ce besoin car il permet de trouver la représentation la plus adéquate à la spécification incomplète d'un objet en cours de conception [RIE91b].

Il est évident qu'un mécanisme de classification et le modèle de données le supportant sont intimement liés. Cependant, l'implantation d'un mécanisme de classification sort du cadre de cette thèse. Elle a été l'objet d'une étude menée en parallèle [LIO93] [RIE91a]. Nous nous contenterons donc de définir un modèle adapté à l'incorporation d'un **mécanisme de classification**.

Les applications de CAO requièrent un grand pouvoir d'expression en ce qui concerne les liens entre objets. En effet, la modélisation d'objets complexes nécessite divers types de liens. Un exemple typique est celui de la relation de composition (est_partie_de) pour laquelle il n'existe pas une sémantique unique. La sémantique peut en partie être fixée par l'expression des différents types de dépendances entre l'objet composite et ses différents composants. Par exemple, la relation entre une voiture et son châssis n'est pas la même que celle existant entre une voiture et ses roues; la destruction d'une voiture entraîne la destruction de son châssis (dépendance existentielle) mais elle n'entraîne pas la destruction de ses roues car ils peuvent être utilisés dans une autre voiture.

Le support des relations sémantiques ou dépendances inter-objets définies par l'utilisateur est couramment cité dans la littérature [KIM89] [RUM87] [WRI84]. L'expression de ces relations a également été étudiée dans le cadre de Shood [ESC90] [ESC90b] [ESC90c] [JEA89]. Celles-ci font actuellement l'objet d'une étude plus approfondie [DJE93a] [DJE93b]. Nous devons fournir un cadre de modélisation propice à leur expression. Notre modèle de représentation de connaissances doit permettre l'expression de **relations sémantiques** entre les objets.

L'intégration des outils de calcul

Nous devrons prendre en compte l'intégration des nouvelles méthodes de calcul ainsi que celles qui correspondent à des outils existants.

Les méthodes de calcul utilisées évoluent au fur et à mesure que des informations sur les objets en cours de conception deviennent disponibles. Ainsi, les méthodes utilisées pour calculer les déformations obtenues lorsqu'une pièce mécanique est soumise à des forces, peuvent être plus précises si l'on dispose d'informations supplémentaires. Dans une étape de prototypage, par exemple, la précision obtenue par un calcul utilisant les hypothèses de résistance des matériaux est suffisante alors que dans une étape finale la précision d'un calcul par éléments finis est nécessaire. Notre modèle doit permettre la **modélisation des méthodes de calcul**.

De plus, lors de la conception d'un objet, il n'est pas rare que le concepteur fasse appel à un certain nombre d'outils spécialisés. Par exemple, lors de la conception d'une pièce mécanique, le concepteur peut utiliser un modeleur 3D, effectuer des calculs par éléments finis et, en phase finale, déterminer les gammes d'usinage. Nous devons donc permettre à l'utilisateur d'avoir accès à toutes les fonctionnalités fournies par ces outils. Il est évident que la démarche consistant à développer ces logiciels en utilisant notre modèle comme support de représentation est impensable car l'investissement humain serait énorme. Nous devons donc utiliser les logiciels existants.

Deux manières d'intégrer des outils existants sont envisageables. La première consiste à utiliser des programmes de conversion entre les formats des différents outils (figure 3). L'intégration est alors faite par les traitements. Cette approche a l'avantage d'être facilement réalisable car il existe un certain nombre de formats dit neutres qui permettent l'échange de données entre outils de CAO tels que IGES [IGE81] et SET [SET91] ou encore le projet STEP [VIE92]. Ici, le rôle du système de représentation de connaissances se limite à celui d’une base de données sur laquelle des informations dont il ne connaît pas la sémantique sont déposées et retirées. Malheureusement, les formats neutres sont sémantiquement pauvres: leur utilisation provoque donc des pertes d'information [TOL92]. Ceci est très grave lors des retours en arrière, situation très fréquente dans la démarche de conception.

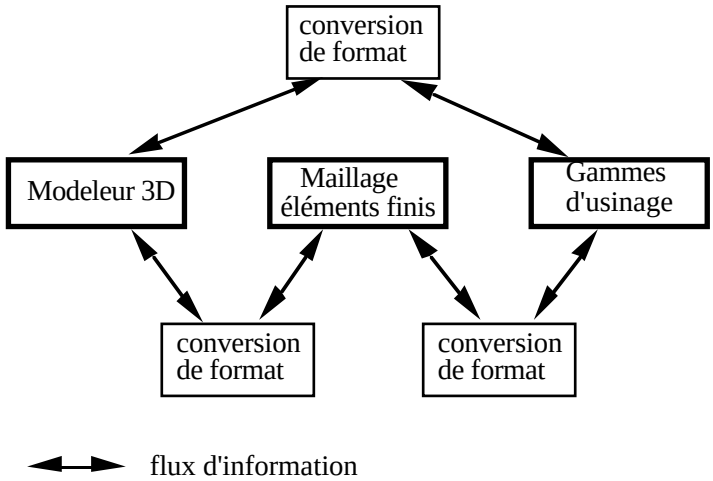


Figure 3. Intégration par les traitements

Une deuxième démarche consiste à donner au système de *représentation de connaissances* le rôle de dépositaire principal de l'information. Le système joue donc le rôle de relais de connaissances entre les différentes applications (figure 4). Ainsi, si un outil doit être utilisé, les informations nécessaires pour son utilisation peuvent être extraites du système de représentation de connaissances et converties dans un format reconnaissable par l'outil. Puis, une fois les calculs effectués, les résultats sont récupérés en transformant leur représentation du formalisme du logiciel en celui du système. L'intégration est alors faite par les données. Cette approche a l'avantage de permettre la conversion d'une représentation en une autre avec une perte minimale de sémantique. Par contre, elle requiert des méthodes de conversion du modèle de données de l'outil vers celui utilisé dans notre système et vice versa.

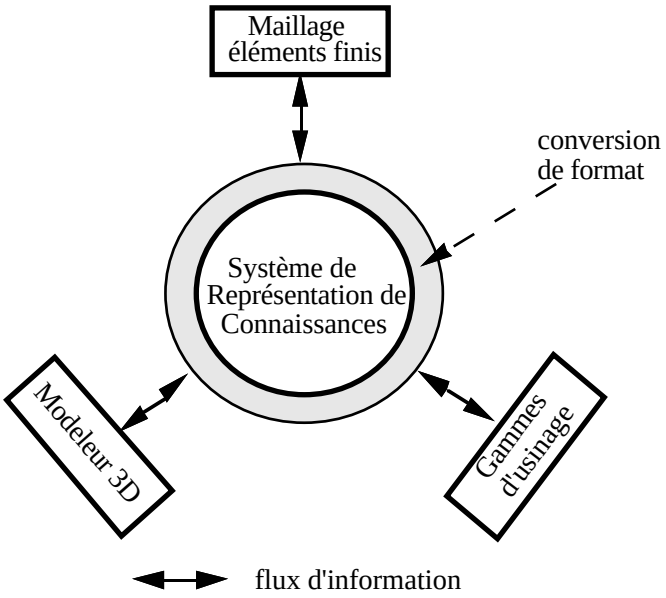


Figure 4. Intégration par les données.

Pour que cette approche soit réalisable, il faut faciliter cette conversion. Nous pensons qu'une manière d'automatiser dans une certaine mesure ces conversions est d'avoir une représentation interne des outils CAO dans notre modèle. Notre modèle doit permettre la **modélisation des outils de calcul** existants.

Evolution des besoins

La conception est une activité doublement évolutive. Elle est évolutive car, d'une part, la structure et les valeurs des objets en cours de conception sont en changement continuel. Mais elle l'est aussi, car les besoins de représentation des applications évoluent. On peut donc parler de deux types d'évolution:

• Les connaissances en CAO sont par nature très dynamiques. Elle changent au cours de la conception. Par exemple, une valeur approximative obtenue par une méthode de calcul est adéquate dans une étape de prototypage mais elle ne l'est plus dans une étape finale de conception; la valeur doit être recalculée par une méthode plus précise. Un modèle de représentation de connaissances doit donc prendre en compte cette dynamique due à la transformation des informations. Il doit supporter la **dynamique**: des objets dans leur processus de conception et celle des processus eux-mêmes.

• Il est impossible de prévoir tous les besoins de représentation des applications de conception. Ces besoins peuvent changer d'une application à l'autre ou évoluer avec le temps. Les relations sémantiques fournissent l'exemple typique d'évolution des besoins d'une application de conception. En effet, les relations entre les objets peuvent être très variées et il est impossible de toutes les répertorier. La figure 5 montre l'exemple d'un évacuateur dont le troisième composant nécessite l'expression des deux dépendances sur le même lien: exclusive et existentielle. Un système n'ayant pas pris en compte cette possibilité doit être étendu [DJE93b] [DJE93c].

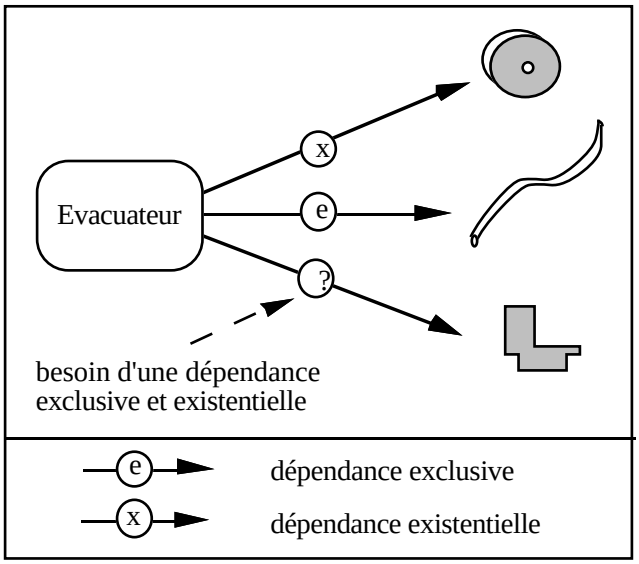


Figure 5. Une application nécessitant une dépendance à la fois existentielle et exclusive. Un système permettant seulement la représentation des liens exclusifs ou existentiels doit être étendu.

Nous devons donc fournir un modèle de représentation de connaissances capable d'évoluer. Il doit être **extensible**.

Contribution

Cette thèse concerne la définition et la réalisation d'un système de représentation de connaissances adapté aux besoins de la CAO. Le système, appelé Shood, propose des solutions aux besoins mentionnés dans la section précédente et classés en trois familles:

- La puissance de représentation
  - La réutilisation des objets
  - La réutilisation des structures
  - La réutilisation des procédures

- L'intégration des outils de calcul
  - La modélisation des nouvelles méthodes de calcul
  - La modélisation des outils de calcul existants

- L'évolution des besoins
  - La dynamique de la connaissance représentée
  - L'extension des concepts du système

Pour résoudre ces problèmes nous nous sommes inspirés principalement des modèles objets issus des Langages Orientés Objet (ObjVlisp [COI87], Clos [DEM87], CommonLoops [BOB85] entre autres) et de la Représentation de Connaissances Centrée Objet (Shirka [REC85], KRL [BRA85a], Mering [FER84] entre autres). L'utilisation des techniques issues de ces domaines nous a permis d'intégrer des caractéristiques que l'on trouve rarement rassemblées dans un même système.

Plan de la thèse

Cette étude comprend deux parties. La première (chapitre 1) est consacrée à l'étude de l'existant dans les domaines des langages de programmation orientés objet et des systèmes de représentation de connaissances. La seconde (chapitres 2, 3 et 4) présente Shood, un modèle de représentation de connaissances pour la CAO [NGU92] [NGU92a].

Dans le chapitre 1 les différentes approches proposées dans les systèmes pour implanter un concept donné sont présentées. Après chaque concept, une réflexion comparant les différentes approches est menée de manière à choisir la plus adéquate. Bien qu'un même concept puisse répondre à plusieurs besoins, nous les avons regroupés en trois grandes familles correspondant aux familles de besoins citées plus haut (puissance de représentation, intégration des outils de

calcul et évolution des besoins). La conclusion de ce chapitre récapitule les choix fondamentaux que nous avons faits. Elle constitue un cahier des charges pour la définition de Shood.

Le trois chapitres suivants correspondent aux trois parties de l'état de l'art: chacun des chapitres correspond donc à une famille de besoins.

Le chapitre 2 présente la partie déclarative de Shood. C'est ici que réside une grande partie de sa puissance de représentation. Les principes de base de Shood sont présentés: la représentation uniforme basée sur le "tout objet"; les concepts d'instance, classe et méta-classe; le pouvoir déclaratif des attributs à travers leurs descripteurs; le partage des structures grâce au mécanisme d'héritage multiple et le partage d'objets grâce à l'identité des objets; la dynamique de l'instanciation multiple.

Le chapitre 3 présente la partie procédurale de Shood. Les méthodes de Shood, inspirées des fonctions génériques de Clos [DEM87] sont présentées. Celles-ci permettent d'une part, de définir des nouveaux outils de calcul et d'autre part, de participer à l'intégration des outils existants. Les inférences et contraintes procédurales sont également présentées.

Le chapitre 4 présente la réflexivité du modèle, c'est-à-dire la modélisation dans Shood de ses propres concepts (attributs, méthodes, inférences et contraintes). La réflexivité facilite les extensions du modèle [MAE87]. En effet, des utilitaires et bibliothèques peuvent être écrits dans le langage lui-même afin de l'enrichir [MAS89]. Nous présentons comme exemple, l'ajout des classes à clef et l'ajout de nouvelles dépendances sémantiques.

Pour conclure, nous dressons un bilan des résultats de cette thèse et les extensions à envisager.

1. Les systèmes à Objets

1.1 Introduction

Le but de ce Chapitre est de présenter les caractéristiques des systèmes utilisant l'approche Objet qui peuvent être utilisés pour la CAO. Deux des principaux domaines d'utilisation du concept objet sont abordés: le génie logiciel (Langages Orientés Objets) et l'intelligence artificielle (Systèmes de Représentation des Connaissances). Seules les notions correspondant aux besoins mentionnés dans l'introduction de cette thèse sont abordées:

• Les aspects déclaratifs. Ces aspects donnent aux systèmes leur puissance de représentation. Nous examinons les différentes manières de réaliser:
-le partage structurel, permettant la réutilisation des structures,
-l'accès aux objets,
-les attributs et les connaissances que l'on peut exprimer sur eux,
-l'héritage multiple, augmentant le partage et
-les points de vue multiples sur un objet.

• Les aspects procéduraux. Ces aspects permettent de modéliser les outils de calcul, nouveaux ou existants. Nous examinons les différentes approches permettant de:
-sélectionner la meilleure méthode et de
-réutiliser les méthodes.

• L'extensibilité. Pour mieux suivre l'évolution des besoins des applications il faut définir un système extensible. La réflexivité s'avère un atout pour l'extensibilité, c'est pourquoi nous examinons donc:
-la méta-connaissance,
-la réflexivité à travers les méta-classes et
-les systèmes "tout objet".

Après chaque notion, une réflexion comparant les différentes approches permet de justifier les choix effectués pour notre modèle.

La conclusion est une compilation des options de modélisation que nous avons retenues. Elle constitue le cahier des charges du modèle Shood.

1.2 Les aspects déclaratifs

1.2.1 Le partage structurel

Les problèmes concernant la réutilisation de concepts structuraux peuvent être résumés par une proposition faite par Minsky en 1975. Celle-ci, bien qu'émanant du domaine de la vision par ordinateur, peut être généralisée car elle porte essentiellement sur la réutilisation des connaissances existantes. Dans sa proposition, il critique d'une part, la spécificité des systèmes de représentation de connaissances destinés à un domaine particulier et, d'autre part, leur manque de pouvoir structurel [MIN75]. Pour Minsky, un être humain se trouvant devant la nécessité de représenter une nouvelle situation ne crée pas une structure de représentation entièrement nouvelle mais au contraire, recherche dans sa mémoire la structure la plus similaire. Il n'a plus alors qu'à effectuer les changements nécessaires pour l'adapter à la nouvelle situation.

Les travaux sur la réutilisation des connaissances peuvent être divisés en deux écoles par rapport à cette proposition. La première correspond à une interprétation quasiment littérale de cette proposition. Elle donne naissance aux systèmes dits d'acteurs [LIB81], qui essaient d'individualiser les actions des objets. Ils utilisent comme mécanisme de base la *délégation*. Un acteur détient l'information nécessaire pour réagir à certains événements. Lorsqu'un acteur se trouve devant un événement auquel il ne sait pas répondre, il "délègue" cette tâche à un acteur capable de le faire. La deuxième école préconise une interprétation plus abstraite. Elle donne naissance aux systèmes de classes [STE86], qui regroupent les objets ayant des caractéristiques similaires. Leur mécanisme de base est *l'héritage* de propriétés. Les systèmes de classes structurent les objets car avant de définir un objet individuel on est obligé de définir le concept générique (classe) de cet objet.

Une discussion sur ces deux approches ainsi que sur leurs avantages et inconvénients est présentée ici afin de répondre à la question: pour quel type de problèmes sont-ils les plus adéquats? Cette présentation, inspirée de [LIB86], est faite d'un point de vue des Langages Orientés Objet (LOO). Dans ce paragraphe nous ne retiendrons de ces deux approches que les caractéristiques structurelles, par opposition aux procédurales.

1.2.1.1 L'approche ensemble-héritage

L'approche ensembliste est basée sur la théorie des ensembles abstraits [MAS89]. La description des ensembles est faite en intention. L'ensemble est décrit par un ensemble de propriétés caractérisant les objets modélisés. Ainsi, la description de l'ensemble des

évacuateurs se fait par énumération des propriétés telles que la masse maximale admissible ou la vitesse maximale de descente. La relation d'appartenance permet l'adjonction des objets individuels à un ensemble.

Lorsqu'on veut définir des objets ayant des propriétés additionnelles, comme des évacuateurs à vitesse réglable, on décrit un sous-ensemble qui partage les propriétés de l'ensemble original auquel sont ajoutées de nouvelles propriétés. Les évacuateurs à vitesse réglable partagent les propriétés communes des évacuateurs mais ils possèdent en propre des propriétés particulières telles que l'action correspondant à "régler la vitesse". Une fois l'ensemble des évacuateurs à vitesse réglable décrit, on peut créer des objets individuels.

L'approche ensembliste est mise en œuvre par les systèmes de classes. Une classe décrit le comportement commun à l'ensemble de ses objets: elle définit les attributs que les objets de l'ensemble ont le droit de valuer ainsi que les méthodes qui leur sont applicables.

Les classes définissent la structure des objets individuels, appelés instances, qui représentent des éléments de l'ensemble. Une instance d'une classe est un objet pour lequel chaque attribut de la classe a une valeur. Par exemple, l'évacuateur *standard-120* est une instance de la classe des Evacuateurs (figure 1.1), où les attributs "masse_max" et "vitesse_max" ont pour valeur respectivement 120 et 1,5. Si les instances d'une même classe partagent le comportement défini par la classe (les méthodes), en revanche chacune d'elles possède des valeurs propres constituant son état courant (les attributs).

Dans l'approche ensembliste, le partage des connaissances est réalisé par l'héritage. Pour affiner les concepts déjà créés, on peut définir des sous-classes d'une classe. Une classe hérite des attributs et des méthodes de sa super-classe, mais des attributs ou méthodes spécifiques peuvent aussi y être définis. La relation de spécialisation ou <classe, sous-classe> correspond intuitivement à la relation d'inclusion ou <ensemble, sous-ensemble>. Or la sémantique de celle-ci n'est pas toujours ensembliste car beaucoup d'implémantations se contentent d'hériter les attributs de la super-classe sans vérifier l'inclusion d'ensembles. Dans la figure 1.1, la classe "Evacuateurs à vitesse réglable", sous-classe de "Evacuateurs", admet une méthode supplémentaire "régler_vitesse".

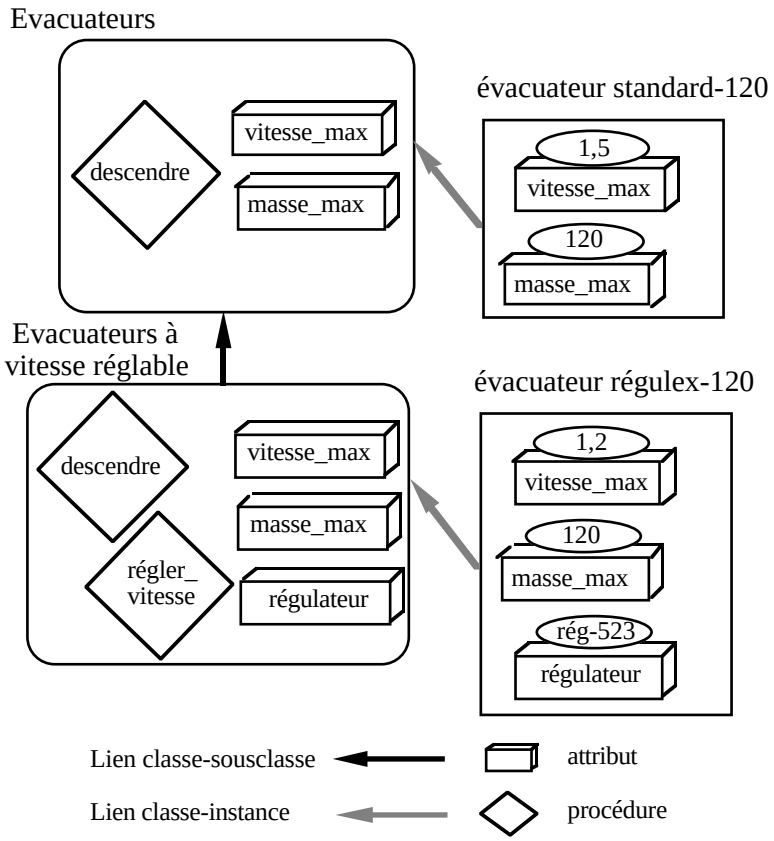


Figure 1.1. L'approche par l'héritage.

1.2.1.2 Une alternative: Prototype-Délégation.

L'approche de représentation des connaissances par prototypes est réputée plus proche de la représentation mentale des êtres humains [LIB86b]. Par exemple, si on demande à quelqu'un de penser à un évacuateur, il est probable qu'il se représente un évacuateur déjà connu de lui, avec ses caractéristiques particulières. Si la personne connaît déjà l'évacuateur *standard-120* (masse_max=120 et vitesse_max=1,5) la représentation d'un autre évacuateur, par exemple l'évacuateur *régulex-120* (masse_max=120, vitesse_max=1,2 et régulateur=rég-523), consiste à énumérer les différences avec l'évacuateur déjà connu (prototype). La représentation mentale consiste à dire que l'évacuateur *régulex-120* supporte la même charge que celle du *standard-120* mais qu'il a une vitesse de descente plus lente et un régulateur de vitesse.

Avec cette approche, on définit d'abord un objet individuel en décrivant son comportement (l'évacuateur *standard-120*). Cet objet peut ensuite devenir un prototype s'il partage son comportement avec des objets qui le signalent comme prototype (l'évacuateur *régulex-120*). Un objet qui signale un autre comme son prototype a le droit de changer localement la description initiale pour représenter son propre comportement (la capacité de régler la vitesse). Tout objet a le droit de devenir le prototype d'autres objets [MAS89].

L'approche des prototypes est implantée naturellement par la délégation [LIB86b]. La délégation est une notion dans laquelle les objets traitent certains messages eux-mêmes et transmettent ceux qu'ils ne peuvent pas traiter à d'autres objets appelés objets prototypes [STE86]. Dans la délégation, un objet a donc un ensemble de propriétés le caractérisant et une liste d'objets prototypes.

Tout objet qui agit selon le principe de la délégation répond à un message X de la manière suivante: d'abord, il cherche la réponse dans son comportement individuel. S'il ne peut pas répondre au message, il envoie le message à un des objets dans sa liste d'objets prototypes en lui disant: "Essaie de répondre à ce message pour moi, mais si tu as besoin de quelque chose pour y répondre (comme la valeur d'un attribut ou une méthode) demande le moi". La figure 1.2 représente d'une manière graphique l'approche des prototypes. L'exemple choisi est celui des évacuateurs traité dans la figure 1.1 par une approche de classes.

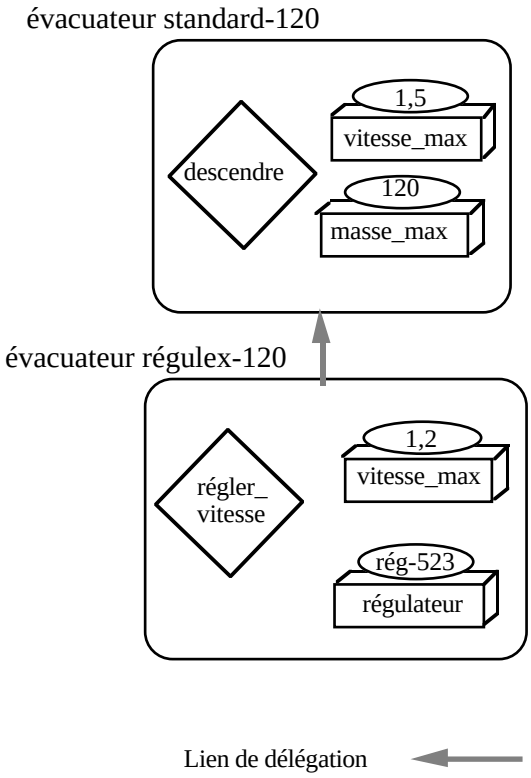


Figure 1.2. L'approche par la délégation.

Pour mieux comprendre le mécanisme, supposons qu'un message arrive à l'objet représentant l'évacuateur *régulex-120*. Ce message demande quelle est sa masse maximale supportée. L'objet évacuateur *régulex-120* commence alors à chercher dans ses propriétés spécifiques cette donnée; comme il ne la trouve pas, il demande à son prototype (l'évacuateur *standard-120*) d'y répondre à sa place. Finalement ce dernier lui donne la réponse cherchée: masse_max=120.

1.2.1.3 Réflexions

Différents points de vue ont été donnés pour montrer tant les avantages de la délégation [LIB86], [LIB86a], [BOR86] que la puissance de l'héritage [STE87].

Si on compare les deux approches par rapport à la *description des nouveaux concepts*, l'approche des ensembles requiert une description abstraite de l'ensemble avant de pouvoir y créer des éléments individuels. Ceci a l'avantage de permettre la gestion d'ensembles sans éléments mais présente aussi l'inconvénient de devoir définir de nouveaux ensembles (ou sous-ensembles) pour augmenter la spécificité des objets. Dans le pire des cas, cette approche nécessite la définition d'un nouvel ensemble même pour un seul élément (anomalie des classes à un seul élément). Dans cette optique, l'approche des prototypes permet la création d'un concept individuel, puis de sa généralisation en précisant quels aspects du concept vont varier. Ceci a l'avantage de permettre une meilleure expression des connaissances par défaut. Dans la délégation on trouve le dual de l'anomalie des classes à un seul élément, à savoir le problème de la représentation des connaissances sur un type d'individus quand on n'a pas d'exemplaire de ces individus.

Du point de vue de l'*implantation*, la délégation a l'avantage de la simplicité de concepts. Dans la délégation, il n'existe qu'un seul type d'objet, et un seul type de lien (le lien de délégation). Dans l'héritage, il existe deux types d'objets: les objets génériques (ou classes) et les objets individuels (ou instances). Entre les objets il existe deux types de liens: le lien d'une classe vers sa super-classe (spécialisation) et le lien d'une instance vers sa classe mère (instanciation).

En ce qui concerne la *puissance de représentation*, de la simplicité de la délégation découlent les inconvénients d'une sémantique faible: le lien de délégation ne supporte pas la classification taxinomique des instances [BOR86]. En revanche, l'héritage supporte une classification taxinomique des instances car, pour ajouter un comportement additionnel à un objet individuel, on doit d'abord créer une sous-classe (i.e un sous-ensemble). Les systèmes à héritage permettent donc l'incorporation d'algorithmes de classification automatique [CLA85] et des améliorations de ces algorithmes en utilisant un peu plus de sémantique (par exemple, le concept d'ensembles disjoints [BRA91]).

En résumé, l'héritage est un mécanisme de réutilisation de concepts plus structuré que la délégation. Cette propriété découle en effet de la représentation taxinomique du lien d'héritage. Cette représentation, absente dans la délégation, est indispensable au processus de classification. C'est cette caractéristique qui fait de l'héritage l'approche la plus adaptée à des applications qui demandent la représentation des produits en cours de conception. De plus, la structuration donnée par le graphe d'héritage donne une vue d'ensemble de la base. Ceci permet

une meilleure compréhension de la base de connaissances.

1.2.2 L'héritage multiple

L'héritage multiple autorise qu'une classe admette plusieurs super-classes. L'héritage multiple permet d'augmenter le partage de propriétés grâce à la combinaison des descriptions de plusieurs classes. La figure 1.3 présente un exemple d'héritage multiple pour les évacuateurs. La classe des évacuateurs de norme franco-allemande hérite de l'union des propriétés de toutes ses super-classes (Evac_norm_française et Evac_norm_allemande).

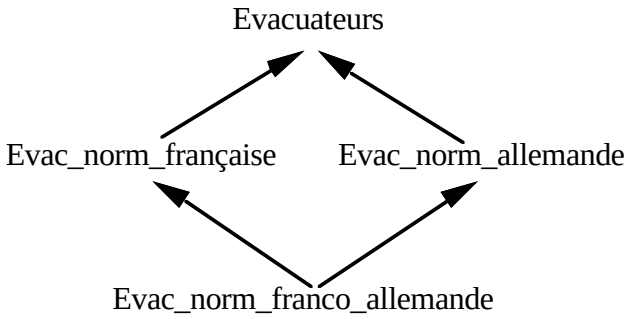


Figure 1.3. Exemple d'héritage multiple. Les évacuateurs de norme franco-allemande héritent des caractéristiques des évacuateurs de norme française et allemande.

1.2.2.1 Conflits d'héritage

Malheureusement l'héritage multiple est à la source des conflits que l'on peut classer en conflit de noms et conflits de valeurs [DUC92]. Il y a conflit de noms quand deux propriétés de même nom sont héritées dans une même classe. Pour résoudre ce conflit il faut déterminer s'il s'agit des propriétés différentes ou bien d'une seule propriété. Une fois le conflit de noms résolu, si l'on détermine que deux propriétés en conflit ont la même identité, il faut résoudre le conflit de valeurs. En effet, il faut choisir de quelle valeur on hérite pour cette propriété. Nous nous intéressons ici aux conflits de noms d'attributs et aux conflits de valeurs de leurs descripteurs (e.g définition du type des attributs).

Les problèmes posés par l'héritage restent encore sans solution définitive. Parmi les travaux les plus représentatifs, on peut citer le travail de Cardelli [CAR84, CAR85] et Danforth [DAN88] en théorie de types, Dugerdil [DUG86] et Ducournau [DUC87] [DUC89] en théorie des graphes ainsi que le travail de Tourezky [TOU84] concernant les exceptions.

Plusieurs critères plus ou moins arbitraires sont utilisés par les systèmes actuels pour résoudre ces conflits (de noms ou de valeurs). Loops [BOB81] utilise des listes de précédence sur les

classes pour choisir la propriété héritée en cas de conflit. Smalltalk-80 [GOL83], C++ [STR86] et Eiffel [MEY89] utilisent l'héritage explicite des propriétés: l'usager doit décider de quelle classe il veut hériter la propriété en conflit ou bien il doit systématiquement renommer les propriétés en conflit. ORION [KIM87] et O2 [BAN88] [BAN89] permettent d'hériter les différentes propriétés en conflit en les renommant. OBJLOG [DUG88] et ROME [CAR89] résolvent les conflits de noms par des techniques de points de vue. Dans l'exemple de la figure 1.4, la classe Evac_norm_franco_allemande hérite de deux attributs homologation: l'un du point de vue d'un Evacuateur français (evac_norm_française.homologation) et l'autre du point de vue d'un évacuateur allemand (evac_norm_allemande.homologation).

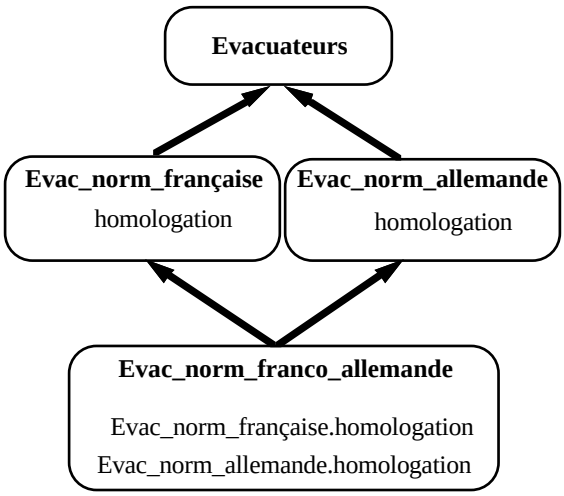


Figure 1.4. Conflit de noms résolu par points de vue. La classe des évacuateurs de norme franco-allemande hérite de deux attributs en conflit. Elle hérite d'un attribut homologation du point de vue français et un attribut homologation du point de vue allemand.

1.2.2.2 La subsomption

La solution donnée aux conflits de valeurs dans la famille de langages à subsomption (KL-ONE [BRA85a], Classic [BRA91], LOOM [MCG88], BACK [VAN87]) respecte l'inclusion d'ensembles:
"If a concept A subsumes concept B then every individual that can be described by B can also be described by A[1]*"* [BRA85a].

Un réseau formé par les relations de subsomption entre différents concepts génériques (classes) forme un graphe de spécialisation. Un graphe de ce type peut être exploité par le classificateur (classifier) [BRA85a], un mécanisme de classification permettant de prendre une

[1]"Si un concept A subsume un concept B alors tout individu pouvant être décrit par B peut aussi être décrit par A"

nouvelle description de concept et de l'insérer à l'endroit adéquat du graphe. Un concept est dit correctement inséré s'il est au-dessus de toutes les descriptions qu'il subsume et en dessous de toutes les descriptions le subsumant[1]. Dans la figure 1.5, la description du concept Evac_franco_allemand a été classée sous les concepts Evac_français et Evac_allemand.

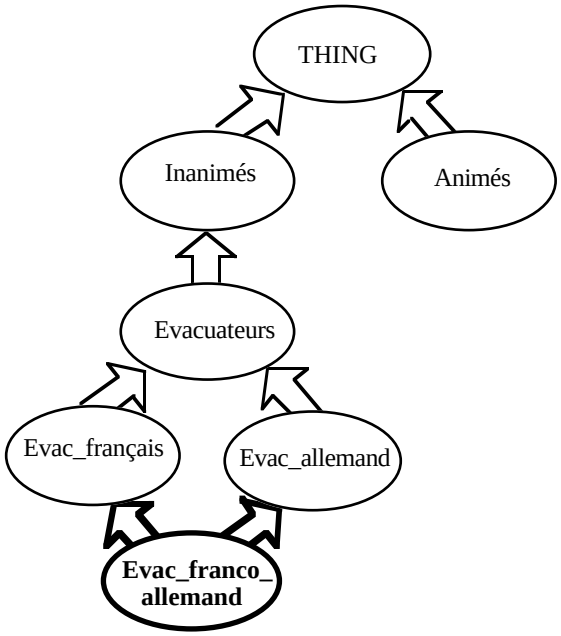


Figure 1.5. Un graphe de subsomption (KL-ONE) entre concepts génériques.

Le mécanisme d'héritage de KL-ONE [BRA85a] prend en compte la relation de subsomption. En effet, les restrictions applicables aux concepts subsumants (Inanimés) doivent aussi être applicables aux concepts subsumés (Evacuateurs). Dans le cas de l'héritage multiple, la conjonction des restrictions des concepts subsumants (Evac_français, Evac_allemand) sont appliquées au concept subsumé (Evac_franco_allemand).

La rigueur du mécanisme d'héritage induite par la relation de subsomption dans KL-ONE a entraîné le développement de systèmes ayant une trop faible expressivité. En effet, seul un nombre réduit de types de restrictions sur les attributs est offert: des restrictions de domaine (et leur intersection), des restrictions de cardinalité, des restrictions exprimant l'égalité ou l'inclusion d'ensembles ou encore un type plus général de restriction permettant d'exprimer une relation entre des attributs en termes d'un autre concept du graphe.

D'autre part, l'héritage multiple des contraintes procédurales sur un attribut pose un problème supplémentaire. La sémantique des procédures étant inconnue, le système ne peut pas garantir qu'une procédure soit plus restrictive qu'une autre. Dans ce cas, seul l'héritage systématique de

[1]La classification d'instances n'est pas implémentée dans KL-ONE. Un mécanisme de ce type tirant parti de la sémantique ensembliste de la subsomption pourrait être facilement implémenté.

toutes les contraintes procédurales relatives à un attribut, réalisé dans Shirka [REC88], permet d'assurer l'inclusion d'ensembles.

1.2.2.3 Réflexions

L'héritage multiple nous paraît indispensable car d'une part, il accroît la capacité d'expression du système et, d'autre part, il permet d'augmenter la réutilisation (structures, objets, procédures). Il augmente aussi le partage par la mise en commun des informations. Un autre avantage qui en découle est que les systèmes ainsi réalisés sont plus modulaires et requièrent des modifications plus localisées. Malheureusement, l'héritage multiple entraîne un certain nombre de problèmes bien connus auxquels il faut offrir de solutions: des conflits de noms et des conflits de valeurs.

Aucune solution des conflits de noms et de valeurs ne semble aujourd'hui définitive. Cependant certaines solutions paraissent être préférables dans un contexte CAO. L'approche de points de vue nous paraît la moins arbitraire pour résoudre les conflits de noms car elle prend en compte une certaine sémantique pour les noms des attributs. C'est cette approche que nous retiendrons pour notre système. Pour les conflits de valeurs, une approche ensembliste à la KL-ONE permettant le fonctionnement d'un mécanisme de classification est souhaitable.

L'accumulation systématique de contraintes procédurales à la manière de Shirka bien qu'assurant l'inclusion ensembliste, peut s'avérer coûteuse. En effet une nouvelle contrainte procédurale sur un attribut peut comprendre les vérifications faites par les contraintes procédurales héritées. Cette duplication entraîne un coût en temps d'exécution pouvant être très important dans des applications CAO. En effet, en CAO, une vérification peut demander l'exécution d'un logiciel existant exigeant un temps d'exécution très long (e.g calcul par éléments finis). De plus, si les contraintes ne sont pas héritées, le système est incapable d'assurer l'inclusion ensembliste. Le respect de celles-ci reste donc complètement à la charge de l'utilisateur. Nous ferons donc un compromis: les contraintes sont héritées par défaut et l'utilisateur peut, s'il le veut, demander de ne pas hériter des contraintes procédurales. Le respect de l'inclusion ensembliste, en ce qui concerne les contraintes procédurales, est assuré par convention.

1.2.3 Le traitement des exceptions

Certains mécanismes d'héritage autorisent les exceptions (e.g NETL [FAH79]). Il y a exception [TOU84] lorsque la définition d'une classe n'est pas applicable aux instances de sa super-classe. Reprenons l'exemple bien connu de la classe des éléphants où l'on affirme que

les éléphants sont gris, qu'ils ont quatre pattes et qu'ils ont une longue trompe. L'apparition d'un éléphant rose engendre une exception dans la classe des éléphants. Les systèmes à exceptions permettent la création d'une sous-classe de la classe des éléphants qui change la couleur grise en rose, mais qui hérite des autres caractéristiques telles que le nombre de pattes et la longueur de la trompe.

Il est évident qu'un système qui permet les exceptions est un système admettant une spécialisation non-stricte dans le sens de l'appartenance aux classes. Le mécanisme rétenu pour Shood est basé sur le point de vue des ensembles et l'utilisation des exceptions peut nous conduire à des situations qui n'ont aucun sens. Par exemple, si l'on permet les exceptions, il est possible de définir une classe des choses grises possédant une sous-classe des éléphants (gris) laquelle a aussi une sous-classe des éléphants roses (non-gris). On peut dire que les éléphants roses sont une spécialisation des éléphants gris et par conséquent aussi une spécialisation des choses grises. Dans ce cas, un éléphant rose est aussi une chose grise.

1.2.3.1 Réflexions

Dans le mécanisme d'héritage de Shood, l'occurrence d'une exception est interprétée comme une mauvaise modélisation par l'utilisateur. Par exemple, il ne permet pas de définir des éléphants gris qui sont aussi roses. L'une des justifications de ce mécanisme est évidente lors de l'utilisation des procédures de classification [BRA85]. Celles-ci bénéficient de la sémantique simple donnée par l'inclusion ensembliste pour améliorer ses performances: l'appartenance d'un objet à une classe implique automatiquement son appartenance à toutes les super-classes. Une modélisation sans exception de l'exemple précédent interdit donc que la classe des éléphants soit une sous-classe de la classe de choses grises. Pour pallier à cette restriction, nous offrons à l'utilisateur des moyens permettant de préciser que, par défaut, les éléphants sont gris. Par exemple, il peut définir dans la classe Eléphants un attribut *couleur* ayant la valeur par défaut "gris". Shood offre aussi des opérations de manipulation de schémas permettant à l'utilisateur de changer les définitions afin de trouver une modélisation plus appropriée (i.e sans exception).

1.2.4 Les points de vue

L'utilisation de points de vue multiples est fréquente dans le processus de conception. Par exemple, un évacuateur peut être analysé à travers différents domaines d'expertise. La figure 1.6 montre l'analyse d'un évacuateur selon deux points de vue différents. Le point de vue cinématique s'intéresse aux composants et à leurs liaisons. Le point de vue fonctionnel s'intéresse aux fonctions de l'objet. Les points de vue cinématique et fonctionnel partagent un

certain nombre d'informations: les composants du schéma cinématique et les composants réalisant les fonctions du schéma fonctionnel. Ils possédent aussi des informations propres (les liaisons entre les composants).

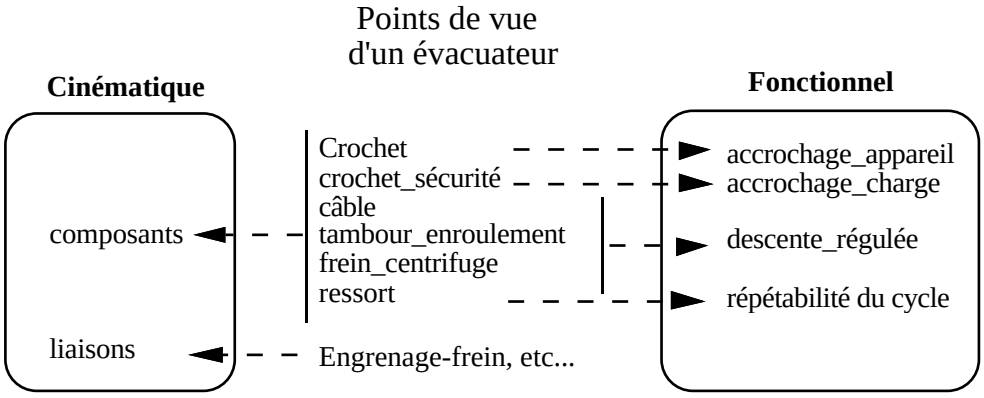


Figure 1.6. Exemple de l'analyse d'un évacuateur au travers de points de vue lors du processus de conception. Le point de vue cinématique s'intéresse aux composants de l'objet et aux liens entre eux. Le point de vue fonctionnel décrit les fonctions que l'objet doit réaliser.

Une première approche pour représenter les deux points de vue est de créer une sous-classe commune aux classes Evac_ciné et Evac_fonc. Un évacuateur particulier, vu de deux points de vue, est donc une instance de la classe Evac_ciné_fonc. Cette approche a au moins un inconvénient. La création systématique de sous-classes communes peut entraîner une explosion combinatoire. En effet, pour chaque combinaison de points de vue nous devons créer une classe.

Certains systèmes offrent une représentation explicite des points de vue. KRL [BOB77] est un langage de représentation de connaissances inspiré directement des idées de Minsky. Dans KRL, une unité individuelle (i.e une instance) peut être définie selon différents points de vue ou *perspectives*. Dans la figure 1.7 l'attribut SELF définit trois perspectives pour l'instance *standard-120*. Chaque perspective est définie par une unité "générique" équivalente à une classe. Dans notre exemple, Evacuateur est une unité générique de base (basic unit). Les unités génériques Evac_fonc et Evac_ciné sont des unités de spécialisation. Elles spécialisent l'unité de base Evacuateur. Une unité individuelle ne peut appartenir qu'à une seule unité de base.

```
[standard-120 UNIT Individual
  <SELF (an Evacuateur with
              masse_max            = 120
              vitesse_max          = 1,5)
          (an Evac_fonc with
             accrochage_apareil  = crochet_15
             accrochage_charge   = crochet_séc_34
             descente_régulée    = (câble_15 tambour_76 frein_120)
             répétabilité du cycle = ressort_61)
          (an Evac_ciné with
             composants          = (crochet_15 crochet_séc_34 ...)
             liaisons            = (engrenage-frein ...) )>]
```

Figure 1.7. Représentation par *perspectives* de l'évacuateur *standard-120* en KRL.

D'autre langages implantent des techniques similaires. Dans ROME [CAR90a] un objet a un lien d'instanciation vers une classe (Evacuateur) et des liens de représentation vers ses différents points de vue (Evac_fonc et Evac_ciné). La restriction ici est que la seule classe ayant le droit de créer des instances est la classe principale (Evacuateur). Les liens de représentation ne servent donc qu'à raffiner l'instance. Dans LOOPS [BOB83], un objet peut aussi avoir plusieurs "perspectives". Ici, un objet et ses différentes perspectives sont modélisés par des objets différents. En effet, chacune des perspectives est un objet ayant son propre lien d'instanciation. L'identité d'objet n'est donc pas conservée car à une entité du monde réel correspondent plusieurs objets.

1.2.4.1 Réflexions

La représentation de points de vue est indispensable pour le processus de conception. Or, les approches de points de vue analysées ici manquent de généralité. Elles obligent par exemple à avoir une classe "point de vue principal" et des classes "point de vue complémentaire" [CAR90]. La création directe d'instances est interdite dans les points de vue complémentaires. Elles doivent d'abord être créées dans leur point de vue principal.

Une certaine généralisation des points de vue peut être observée dans les langages terminologiques [BRA79] [MAR90] [NAP92]. Dans ces langages une instance peut appartenir à des concepts différents après l'exécution du mécanisme de classification. En effet, dans LOOM [MCG88], un concept individuel (instance) peut avoir des relations de subsomption (instanciation) vers plus d'un concept générique (classe). Ceci suggère que les points de vue peuvent être traités comme une généralisation du concept de l'instanciation.

Cette généralisation de l'instanciation est appelée *instanciation multiple* ou *multi-instanciation*. Dans la multi-instanciation une instance peut appartenir à plus d'une classe. Les restrictions sur

les points de vue des systèmes préalablement cités ne s'appliquent pas à la multi-instanciation. Par souci de cohérence, la sémantique de la multi-instanciation doit respecter la sémantique ensembliste choisie pour les classes et l'héritage. Une instance appartenant à plusieurs classes doit donc respecter la conjonction des restrictions de ses classes d'appartenance. En effet, un objet créé par multi-instanciation doit être idéntique à un objet obtenu par la création d'une sous-classe commune aux classes multi-instanciées, puis par la création de l'objet dans celle-ci[1].

Nous retiendrons pour notre modèle le concept d'instanciation multiple plus général que le concept explicite de point de vue. La sémantique de l'instanciation multiple devra respecter la sémantique ensembliste des classes et de l'héritage. Une étude concernant la modélisation explicite de points de vue est en cours [DJE93a]

1.2.5 L'accès aux objets

1.2.5.1 Encapsuler ou non?

L'encapsulation est un mécanisme bien connu des langages orientées objet implantant le principe de l'abstraction de données [MAS89]. Pour réaliser l'encapsulation, tout objet est composé de deux parties: une implantation et une interface. L'implantation d'un objet est composée de deux sous-parties: la partie données, chargée de garder son état, et la partie procédurale, implantant les opérations (méthodes) applicables à l'objet [STE86]. L'interface d'un objet est définie par la signature[2] de l'ensemble d'opérations qui lui sont applicables. L'accès à l'objet se faisant par l'interface, l'encapsulation permet une certaine modularité. En effet, les modifications effectuées sur l'implantation d'un objet ne modifient éventuellement pas son interface et, par conséquent, ne requièrent pas la modification des programmes l'utilisant.

Dans l'encapsulation stricte, dont Smalltalk-80 [GOL83], [STE86] est l'archétype, on ne peut accéder à un objet que par l'intermédiaire de son interface car elle est la seule partie visible d'un objet (figure 1.8). Toute opération de consultation ou de modification d'un attribut dans un système de ce type demande la définition d'une méthode de lecture ou d'écriture. Ainsi, on peut imaginer la situation extrême nécessitant la définition des méthodes de lecture et d'écriture pour

---

[1]Une variante de l'algorithme de classification de LOOM [MCG88] prend en compte cette propriété. Lorsqu'un concept individuel subsume plus d'un concept générique, il crée un concept générique subsumant les concepts en question. Puis, il rattache le concept individuel au concept générique créé.

[2]La signature d'une méthode spécifie le nom de la méthode, les noms et classes des arguments en entrée ainsi que la classe du résultat, s'il y en a un [BER91].

tous les attributs d'une classe. Tel peut être le cas d'un langage de requêtes de ba
orientée objet réalisant une encapsulation stricte: des méthodes de
pour tout attribut. L'inconvénient de cette approche est que l'
méthodes pour manipuler les attributs d'une classe, afin d nnées,
revient à s'en affranchir. Dans cette approche, on perd la aration
interface/implantation de l'encapsulation.

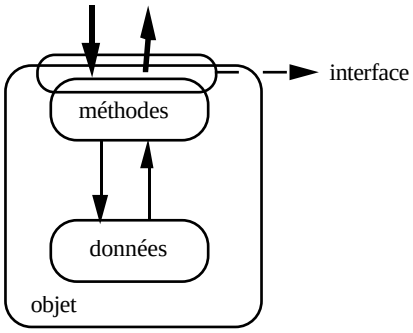


Figure 1.8. L'encapsulation stricte. On ne peut accéder aux attributs qu'à travers des méthodes.

Certains systèmes offrent les deux: une encapulation stricte pour certains attributs et un accès direct pour d'autres. Par exemple, dans C++ [STR89] il existe trois types d'attributs: les attributs privés (private), les attributs protégés (protected) et les attributs publics (public). Les attributs privés sont apparentés aux attributs de l'encapsulation stricte de Smalltalk-80 [GOL83] à la différence près que le caractère privé des données est restreint à la classe et non pas à l'objet. En effet, dans C++ [STR89] on ne peut pas cacher les données d'une instance à une autre instance de la même classe. Les attributs protégés sont visibles dans leurs classe de définition et dans leurs sous-classes. En revanche, les attributs publics sont accessibles à n'importe quelle instance de n'importe quelle classe: l'encapsulation est violée. Finalement, C++ offre un mécanisme permettant de déclarer des classes comme étant "amies" (friends). Ceci leur permet de partager les variables (et les méthodes) protégés et privées.

Malheureusement, certaines applications requièrent l'accès direct aux attributs. Ceci est le cas des systèmes comparant des objets. Par exemple O2, un système de bases de données orienté objet respectant l'encapsulation possède un langage de requêtes permettent une violation de l'encapsulation [BAN89]. En représentation de connaissances, des mécanismes couramment utilisés, tels que la classification [BRA85] et la sélection de frames par filtrage [RECH85] sont également basés sur la comparaison des objets. En effet, lors de la classification d'une instance dans un graphe de classes, l'accès direct aux attributs est nécessaire afin de vérifier les types et les contraintes. Des systèmes de représentation de connaissances possédant des mécanismes de

classification et/ou de filtrage, tels que Shirka [RECH85] ou KEE [FIK85], n'offrent pas mécanisme d'encapsulation.

Finalement, un compromis est fait par les systèmes implant stricte. Dans ces systèmes, un mécanisme réalisant l'enca interdire l'accès direct aux valeurs des attributs (figure n est faite par convention. Ceci est le cas de Loops [BOB83] et C net l'accès à un attribut n'ayant pas de méthode spécifique tion *slot-value*).

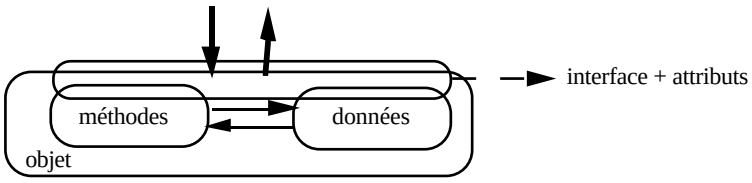


Figure 1.9. Encapsulation non-stricte. L'utilisateur choisit de la respecter en passant par l'interface ou de la violer en accédant directement aux attributs.

1.2.5.2 Réflexions

L'encapsulation stricte est un concept attrayant car elle permet une certaine modularité. Cependant, en CAO, l'accès aux valeurs des attributs est trop fréquent (comparaison, filtrage, classification). De plus, la plupart des logiciels existants de CAO que l'on doit intégrer ne respectent pas l'encapsulation.

L'encapsulation stricte est trop coûteuse pour un système manipulant des connaissances pour la CAO. Celle-ci reste néanmoins souhaitable dans la mesure où l'accès aux attributs reste possible. Notre position est donc de ne pas interdire l'accès aux attributs et de fournir un moyen (les méthodes) pour éventuellement réaliser une encapsulation par convention.

1.2.6 Les attributs

Les attributs modélisent les propriétés des objets correspondant à leur état. Pour assurer leur cohérence ou pour exprimer la manière de les calculer, il est intéressant d'exprimer des connaissances sur les attributs. Dans le monde objet, certains systèmes ne permettent l'expression d'aucune connaissance sur les attributs. Tel est le cas de Smalltalk-80 [GOL84], Flavors [MOO86] et Clos [DEM87] où les attributs sont de simples pointeurs vers des valeurs.

Ceci n'est pas le cas des systèmes de représentation de connaissances à base de frames. Dans ces systèmes, les connaissances sur les attributs permettent de préserver l'intégrité sémantique de la base de connaissances. De plus, ces connaissances, déclaratives ou procédurales, peuvent être exploitées par divers mécanismes (instanciation, classification, filtrage, etc.).

Rappelons rapidement quelques concepts des frames [MAS89]. Un frame représente une classe d'objets. Il est défini par des attributs appelés slots composés de facettes. Les facettes permettent de décrire un attribut, elles sont aussi appelées descripteurs d'attribut. Il existe des facettes *déclaratives* et des facettes *procédurales*. Les facettes *déclaratives* définissent la connaissance statique sur un attribut. Les définitions du type, de la cardinalité, des restrictions et des valeurs par défaut d'un attribut sont des facettes déclaratives [ROB77]. Les facettes *procédurales* définissent la connaissance dynamique sur un attribut. Les réflexes (attachements procéduraux) sont des facettes procédurales [REC88]. Ils permettent de calculer la valeur d'un attribut ou de définir le comportement des attributs lors de manipulations. La figure 1.10 montre un frame représentant la classe de Evacuateurs. Nous pouvons remarquer ici les trois niveaux de structure frame/slot/facette. Deux facettes définissent d'une part le domaine (mono-valué réel) et d'autre part l'intervalle de valeurs possibles (de 1,2 à 1,5) pour l'attribut vitesse_max.

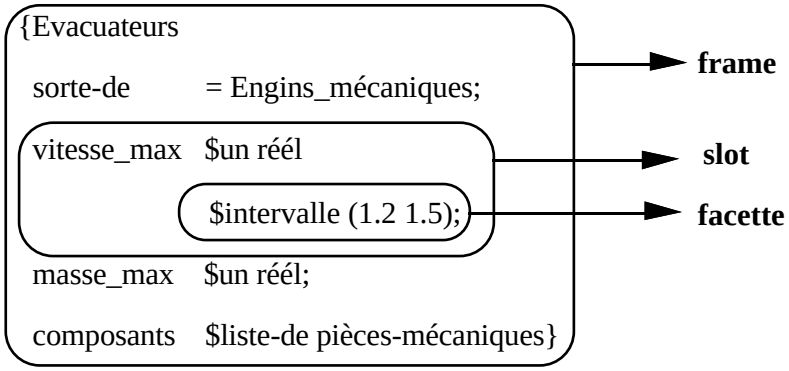


Figure 1.10. Représentation du frame de la classe des évacuateurs dans Shirka [REC85]. Dans Shirka, les frames sont appelés schémas et les slots sont appelés attributs.

1.2.6.1 Les connaissances déclaratives sur les attributs

KEE [FIK85] est un système à base de frames réalisé en Common Lisp. Les facettes déclaratives de KEE permettent de définir la cardinalité d'un attribut (CardinalityMin et CardinalityMax), son domaine de valeurs (ValueClass), ses valeurs par défaut (Values) et des contraintes procédurales (Constraints). La figure 1.11 présente la définition de deux slots du frame représentant la classe Véhicules.

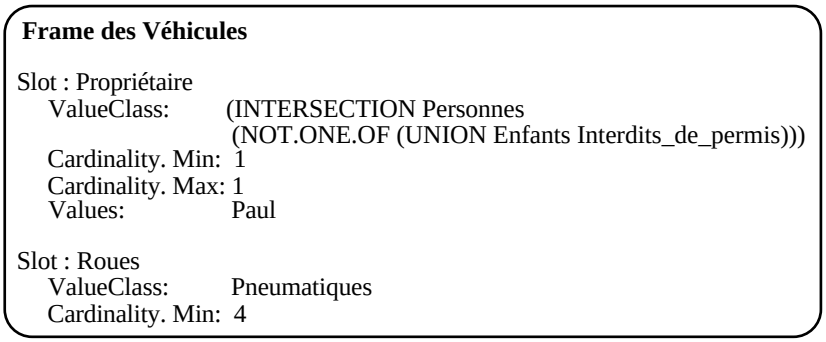


Figure 1.11. Description partielle du frame représentant la classe Voitures.

La facette ValueClass décrit l'ensemble de valeurs possibles du slot. Par exemple, la valeur de l'attribut roues doit être une instance du frame représentant la classe des Pneumatiques. Une originalité de cette facette de domaine par rapport à celles trouvées dans d'autres systèmes de frames tels que Shirka [REC85] ou SRL [WRI84] est d'inclure des opérateurs ensemblistes. Dans notre exemple, le propriétaire d'un véhicule est une personne qui n'est ni un enfant ni un interdit de permis de conduire.

La facette Values permet de définir des valeurs par défaut. Dans notre exemple, si au moment de créer un représentant (instance) du frame Véhicules on ne donne pas de valeur à l'attribut propriétaire celui-ci aura Paul comme valeur, celle étant déclarée comme valeur par défaut.

Finalement la facette Constraints de KEE [FIK85] permet, comme la facette $a-verifier de Shirka, d'attacher une restriction procédurale à un slot. Dans Shirka [ROU88] cette facette est un prédicat correspondant à une fonction Lisp dont l'évaluation retourne vrai ou faux.

1.2.6.2 Les connaissances procédurales sur les attributs

Les frames ne possèdent pas de comportement "propre" au sens des LOO, par exemple, ils ne peuvent pas répondre à des messages. Il sont donc manipulés par des primitives. En revanche, des procédures peuvent être attachées à leurs attributs grâce aux facettes procédurales. Elles sont appelées réflexes [ROB87] car elles sont exécutées en réaction à l'accès aux attributs. Les termes attachement procédural et démons [MAS89] sont aussi utilisés pour les réflexes.

A toute opération de manipulation d'un frame (lecture, écriture, modification ou suppression) correspond un ou plusieurs réflexes. Les langages des frames tels que Shirka [REC85] ou FRL [ROB77] ont toute une panoplie de facettes procédurales permettant de décrire des réactions, *a priori* et *a posteriori*, à des opérations de lecture, écriture, modification ou suppression. Ainsi, la figure 1.12 montre la définition de réflexes pour l'attribut masse-max. Le réflexe $sib-exec exprime que la procédure *détermine-vitesse* doit être utilisée pour calculer le vitesse maximale

de l'évacuateur. La facette $si-succes exprime qu'en cas de succès le résultat doit être rangé dans l'attribut (*vitesse-max*).

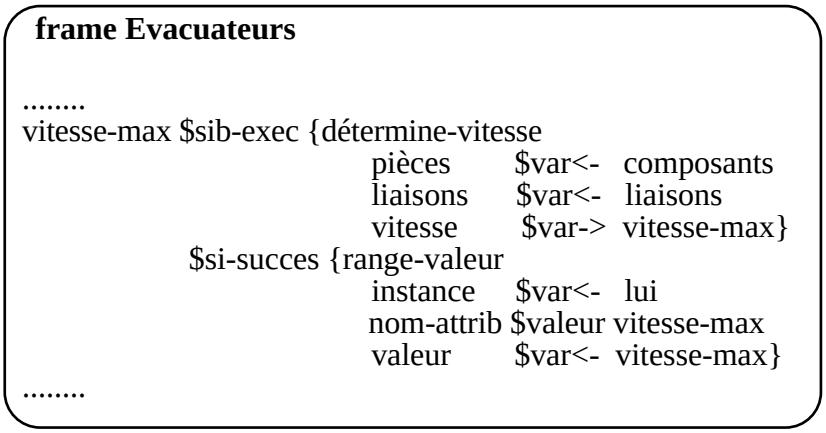


Figure 1.12. Description dans Shirka des facettes procédurales pour l'attribut *vitesse-max* du frame représentant la classe Evacuateurs.

La programmation avec des frames consiste donc à décrire les réactions qui correspondent aux opérations de manipulation. Ceci leur a donné droit à l'appellation de *programmation dirigée par l'accès* (ou par les données) [MAS89].

1.2.6.3 Réflexions

La complexité des applications de CAO requiert un modèle de représentation puissant. Si le modèle objet paraît répondre aux spécificités de ce type d'applications, c'est en partie dû au pouvoir d'expression des attributs. En effet, tout ce que l'on peut dire sur l'état d'un objet réside dans ses attributs. Les modèles objet les plus puissants sont donc ici les plus évolués au niveau de la sémantique des attributs.

Les connaissances déclaratives exprimées sur les attributs des frames permettent de gérer des objets complexes et sémantiquement riches. Elles ont un double usage car d'une part elles préservent l'intégrité sémantique des objets et donc de la base de connaissances, d'autre part elles peuvent être exploitées par des mécanismes tels que la classification de classes ou d'instances. Un exemple de ceci est la connaissance déclarative exprimée par une restriction de domaine. Elle est utilisée pour préserver l'intégrité des objets: seules les valeurs appartenant au domaine sont admises. Mais elle est aussi exploitée par le mécanisme de classification. Par exemple, lors de la classification d'instances, seul un objet vérifiant les restrictions de domaine d'une classe peut lui être ratachéé. L'expression des connaissances déclaratives sur les attributs nous paraît donc indispensable.

Les connaissances procédurales sur les attributs permettent d'enrichir leur sémantique. Elles peuvent aussi être utilisées pour maintenir la cohérence lors des modifications. Cependant dans les systèmes à frames, même un utilisateur averti est incapable de faire évoluer les facettes procédurales. En effet, celles-ci sont en nombre fixe par rapport aux primitives. Par exemple, si à l'opération de modification correspondent uniquement deux facettes procédurales: avant-modification et après-modification, l'ajout d'une troisième: vérifie-avant-modification est très difficile. De plus, la seule expression de réflexes s'avère souvent insuffisante car l'ordre de déclenchement de réflexes du même type sur différents attributs peut être très important. Il serait donc intéressant de pouvoir combiner des techniques d'attachement procédural (réflexes), des règles événementielles (événement, condition, action) et/ou des tâches.

Dans le cadre de ce travail nous définirons un système pouvant être étendu pour prendre en compte les notions des réflexes, règles et tâches (cf. § L'extensibilité). En ce qui concerne l'attachement procédural, deux types de réflexes seront pris en compte:

- Un réflexe permettant le calcul de valeurs des attributs lors des opérations de création.
- Un réflexe permettant l'expression de contraintes procédurales.

1.2.7 Récapitulatif des aspects déclaratifs

Le tableau suivant récapitule les aspects déclaratifs qui ont été abordés dans cette section. Pour chaque notion, on présente l'approche retenue pour notre modèle ainsi que le ou les systèmes qui nous ont inspiré.

Notion		Approche Shood	Origine	
			RCO	LOO
Partage structurel		héritage multiple	Shirka [REC88] KL-ONE [BRA85a]	Loops [BOB85]
Exceptions		interdites	KL-ONE [BRA85a]	
Attributs	Conflits de noms	points de vue	ROME [CAR89]	
Attributs	Conflits de valeurs	inclusion d'ensembles (sauf les contraintes procédurales)	Shirka [REC88] KL-ONE [BRA85a]	
Points de vue		multi-instanciation	ROME [CAR89] Loom [MCG88]	
Accès aux objets		pas d'encapsulation stricte (encapsulation par convention)		Clos [DEM87] Loops [BOB85]
Attributs	Connaissances déclaratives	facettes déclaratives	Shirka [REC88] KEE [FIK85] SRL [WRI84]	
Attributs	Connaissances procédurales	facettes procédurales	Shirka [REC88] FRL [ROB77]	

1.3 Les aspects procéduraux

Une fonctionnalité fondamentale des systèmes orientés objet concerne la gestion des connaissances procédurales. Ce type de connaissances occupe une place importante dans notre étude car elle intervient dans au moins trois des besoins évoqués dans les motivations, à savoir, la modélisation puissante de méthodes de calcul, la modélisation des outils de calcul existants et la réutilisation des concepts procéduraux. Les fonctionnalités que nous étudions dans cette section ont leurs origines dans les LOO. Ces fonctionnalités concernent l'organisation des procédures et ont donné naissance au concept de méthode [GOL84] [DEM87]. Les SRC se sont plutôt préoccupés d'un problème de niveau supérieur: l'organisation des méthodes de calcul [ORS90]. Ceci a donné lieu aux systèmes de gestion de tâches [CHA89] [CHA89a] ou de pilotage d'algorithmes (cf projet ORION). Ces derniers ne seront pas abordés dans cette thèse.

Nous examinons ici deux caractéristiques du partage procédural (méthodes) des LOO:
• Le *choix de la procédure la plus appropriée* pour la réalisation d'une opération. Une opération peut avoir une ou plusieurs implantations différentes appelées *méthodes*. Ainsi, lors de l'exécution, le choix de la meilleure méthode est fait en fonction des arguments effectifs de l'opération.
• La *réutilisation des méthodes*. Les méthodes implantant une même opération sont toujours organisées d'une manière hiérarchique allant de la plus générale aux plus spécifiques. Cette hiérarchisation permet à une méthode spécifique de réutiliser une méthode plus générale. Puis, selon le système, on peut choisir soit de réutiliser toute la méthode soit de réutiliser seulement une partie de celle-ci.

1.3.1 Sélectionner la méthode la plus appropriée

Le partage procédural est analysé ici sous deux angles: celui des systèmes à envoi de messages et celui des systèmes à fonctions génériques.

1.3.1.1 L'envoi de messages

Dans les systèmes à envoi de messages, les opérations (i.e les méthodes) sont des propriétés des classes qui peuvent être surchargées (redéfinies) dans une sous-classe. Par exemple, nous pouvons définir une méthode d'impression appelée *print* applicable à toutes les personnes dans la classe de Personnes. Puis, nous pouvons redéfinir la méthode *print* dans la classe de

Parents, sous-classe de Personnes, de manière à ce qu'elle imprime aussi leurs enfants.

Cette organisation des méthodes est exploitée par le mécanisme d'envoi de messages. Dans ce mécanisme, pour exécuter une opération sur un objet, l'utilisateur doit lui envoyer un message dans lequel il spécifie l'opération (sélecteur de la méthode) ainsi que les arguments nécessaires à son exécution. Face à un message, le système réagit en exécutant la méthode définie dans la classe d'appartenance de l'objet. Ainsi, l'envoi d'un message ayant comme sélecteur *print* à *dupont*, une instance de la classe de Parents, déclenche l'exécution de la méthode *print* définie dans cette classe. Si nous assimilons l'objet sur lequel on effectue l'opération au premier argument d'une fonction, l'envoi de messages est une approche où seul le premier argument (i.e le receveur) est pris en compte pour la sélection de la méthode la plus appropriée.

L'envoi de messages a l'avantage d'être une approche intuitive. Par exemple, si nous voulons changer la définition d'une méthode héritée, il suffit de la surcharger (overloading) dans cette classe. Malheureusement, cet avantage devient un inconvénient lorsque l'on traite des méthodes multi-classes [DEM87]. Une *méthode multi-classe* est une méthode dont le choix du code dépend de plus d'un argument. Dans ce cas on ne peut pas placer les méthodes dans une classe car elles dépendent de plusieurs classes. Ceci est le cas de la méthode ProgrammeContrôle qui, pour déterminer le programme de contrôle d'exécution d'un robot, dépend des tâches à réaliser et des Robots utilisées pour accomplir la tâche. On ne peut pas définir la méthode dans la hiérarchie des classes ayant Robots comme racine car la méthode ProgrammeContrôle dépend aussi des tâches à réaliser. La figure 1.13 montre les hiérarchies des classes Robots et Tâches.

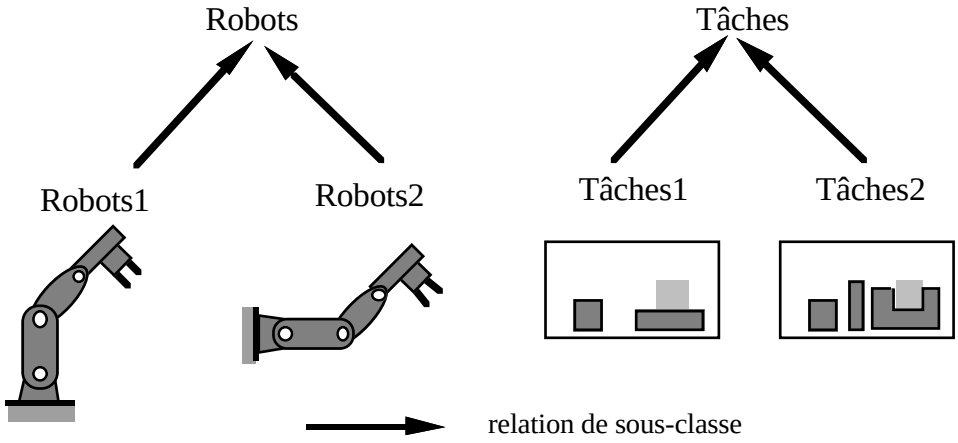


Figure 1.13. Les hiérarchies des classes Robots et Tâches. Avec envoi de messages, la méthode ProgrammeContrôle ne peut pas être définie d'une manière satisfaisante.

DeMichiel & al. [DEM87] énumèrent trois techniques générales qui peuvent être utilisées pour pallier au problème de l'"ambiguïté" dans l'envoi de messages due aux opérations multi-classes: la currification, la délégation et la distribution.

La *currification* est une technique qui consiste à décomposer la méthode multi-classe en une série d'opérations mono-classe. Ici, les objets s'envoient des messages entre eux et accumulent au fur et à mesure leurs contributions individuelles. Pour l'addition d'une séquence de nombres ceci consisterait à demander à chaque nombre de s'additionner lui-même à une somme partielle et puis de demander au nombre suivant de faire la même chose. Chaque objet faisant partie d'un processus de currification doit donc savoir comment commencer et comment finir l'opération. La currification peut amener à un système d'envoi de messages très compliqué.

Dans la *délégation* un objet est chargé d'effectuer des opérations sur un certain groupe d'objets (cf. § Une alternative: Prototype-délégation). Pour l'addition d'une séquence de nombres, on peut définir un objet acceptant un message contenant les identités d'une séquence de nombres qui effectuerait la somme. Ainsi, si un objet reçoit un message demandant l'addition d'une séquence de nombres, il doit déléguer cette opération à l'objet capable de l'effectuer. Par conséquent tout objet faisant partie d'un processus de délégation doit savoir comment et à qui il doit déléguer une opération.

La *distribution de méthodes* est une technique dans laquelle, à tout objet susceptible de participer dans une opération multi-classes, peut être fournie toute l'information nécessaire pour réaliser cette opération. Pour l'addition d'une séquence de nombres, tout objet susceptible de réaliser cette opération doit être muni d'une méthode permettant de répondre au message. Les méthodes doivent donc être dupliquées.

Ces trois techniques offrent des moyens pour contourner le problème des méthodes multi-classes dans les systèmes à envoi de messages. Aucune des trois n'offre une solution satisfaisante et naturelle. En effet, ces trois techniques peuvent être vues comme des astuces de programmation qui permettent de pallier à des faiblesses conceptuelles.

1.3.1.2 Les fonctions génériques

Une fonction générique est une fonction dont le comportement dépend des classes d'appartenance des arguments [STE86]. Cette propriété permet aux fonctions génériques de traiter correctement les méthodes multi-classes. Dans cette approche, les classes et les méthodes sont des entités autonomes. Les méthodes ne sont pas des propriétés des objets ni l'inverse. Les fonctions génériques permettent donc de séparer les méthodes des classes.

Une implantation bien connue des fonctions génériques est celle de Clos [DEM87]: une fonction générique est formée d'un ensemble de *méthodes*. Chacune des méthodes définit une

implantation pour un certain type d'arguments effectifs. L'en-tête d'une méthode est composé du nom de la fonction générique dont la méthode fait partie, d'une séquence de spécialiseurs des arguments qui spécifient quand la méthode est applicable et d'arguments optionnels.

Clos n'autorise que deux types de spécialiseurs pour les arguments des méthodes. L'un permet de restreindre le domaine de l'argument à une classe particulière, tel que l'appartenance à la classe Evacuateurs. L'autre permet de tester l'égalité avec un autre objet telle que l'égalité avec l'objet *standard-120*. Dans notre exemple des programmes de contrôle d'exécution nous pouvons écrire la fonction générique ProgrammeContrôle composée des méthodes montrées dans la figure 1.14. M1 est la méthode la plus générale, elle s'applique à tout type de robots et à tout type de tâches. M2 et M3 sont plus spécifiques. M2 s'applique à tout type de robots et aux tâches de type Tâches2. M3 s'applique aux robots de type Robots1 et à tout type de tâches.

```
M1 : (DefMethod ProgrammeContrôle ((r Robots)   (t Tâches)) ...)
M2 : (DefMethod ProgrammeContrôle ((r Robots)   (t Tâches2)) ...)
M3 : (DefMethod ProgrammeContrôle ((r Robots1) (t Tâches)) ...)
```

Figure 1.14. Définition des méthodes de la fonction générique ProgrammeContrôle.

Lorsque l'on appelle une fonction générique, celle-ci exécute un sous-ensemble de ses méthodes en fonction des classes d'appartenance des arguments effectifs. Un mécanisme de sélection de méthodes contrôle la sélection et l'ordre dans lequel les méthodes sont exécutées. Ce mécanisme sélectionne les méthodes applicables, puis les ordonne de la plus spécifique à la plus générale pour finalement les combiner [KEE88].

Une méthode est applicable si les arguments effectifs fournis lors de l'appel satisfont les spécialiseurs de ses arguments. Les arguments optionnels ne peuvent pas être spécialisés et ne sont pas utilisés pour la sélection des méthodes applicables.

Pour ordonner les méthodes applicables lors d'un appel, Clos utilise les listes de précédence de super-classes directes ou indirectes que toute classe possède. Dans cette liste toute classe précède ses super-classes. La spécialisation multiple pose un problème pour l'ordonnancement des super-classes directes d'une classe. Dans ce cas, Clos utilise l'ordre de définition des super-classes donné par le programmeur.

L'ordonnancement des méthodes applicables de Clos privilégie les arguments définis les premiers. En effet, il utilise d'abord le premier argument pour ordonner les méthodes. Si une ambiguïté est trouvée, c'est à dire si deux méthodes ont le même spécialiseur d'argument pour le premier argument, il utilise alors le deuxième et ainsi de suite. Pour notre exemple de programme de contrôle d'exécution, l'appel (ProgrammeContrôle r1 t2) où r1 est une instance

de la classe Robots1 et t2 est une instance de la classe Tâches2, donne l'ordonnancement de méthodes (M3 M2 M1). Nous pouvons remarquer que l'ordre des arguments est déterminant pour l'ordonnancement des méthodes. En effet, l'inversion des arguments r et t produirait l'ordre (M2 M3 M1).

Une fois la liste obtenue, Clos procède à la combinaison des méthodes. Le programmeur peut spécifier lors de la définition de la fonction générique, le type de combinaison de méthodes à utiliser [DEM87]. Par exemple, il peut demander d'exécuter la méthode la plus spécifique ou bien de combiner l'exécution de toutes les méthodes applicables par un mécanisme de combinaison déclarative de méthodes tel que celui de CommonLoops [BOB85] ou Flavors [KEE88] (cf. § La réutilisation de méthodes).

C++ est un système qui réalise également les fonctions génériques[1]. La sélection de la méthode à exécuter dans C++ diffère de celle de Clos. Cette sélection est faite un trois étapes. D'abord, une méthode ayant une correspondance exacte avec les type des arguments est recherchée. Si aucune méthode n'est trouvée, on recherche ensuite une correspondance en utilisant les méthodes de conversion standard (e.g conversion entre entier et réel) et on applique la première trouvée. Finalement, une correspondance entre les arguments effectifs et les arguments formels d'une méthode est recherchée en utilisant les conversions définies par l'utilisateur (e.g conversion entre les nombre complexes -définie par l'utilisateur- et les réels). Pour notre exemple du programme de contrôle d'exécution, la réponse à l'appel (ProgrammeContrôle r1 t2) peut changer d'un compilateur à l'autre. Par exemple, le compilateur C++ de GNU (gcc) [TIE90] exécute aléatoirement M2 ou M3. La méthode exécutée est en fait celle ayant été déclarée la première.

1.3.2 La réutilisation des méthodes

La réutilisation ou spécialisation des méthodes [STE86] est un processus qui autorise la modification incrémentale des méthodes. Nous distinguons la spécialisation procédurale de la spécialisation déclarative.

La spécialisation procédurale des méthodes est une approche dans laquelle la combinaison des méthodes est sous la responsabilité du consommateur, c'est à dire de la classe utilisant les méthodes. En Loops [BOB81] cette spécialisation est réalisée en utilisant une collection ésotérique d'appels de procédures: l'appel d'une méthode locale (<--), d'une méthode d'une super-classe (<--super), l'appel combiné de toutes les méthodes de toutes les super-classes les

[1]A ne pas confondre avec fonctions paramétrées appelées modèles.

plus directes (<--superfringe) et finalement l'invocation explicite des méthodes (domethod). L'utilisation de l'appel (<--super) permet de ne pas réécrire de code et de ne pas être obligé de faire des appels explicites. L'utilisation de l'appel (domethod) doit être faite avec soin, parce que ce type d'appel pose des problèmes quand le graphe d'héritage évolue.

Dans la spécialisation déclarative des méthodes, la combinaison des méthodes est sous la responsabilité du producteur, c'est à dire de la classe où les méthodes sont définies. En Flavors [WEI81], un système utilisant l'envoi de messages, on sépare la définition d'une méthode en trois parties: la partie initiale d'une méthode (before), la partie principale d'une méthode (primary) et la partie finale d'une méthode (after). En Flavors, l'exécution d'une méthode consiste à exécuter une série de parties initiales, puis une seule partie principale et finalement une série de parties finales. Les parties initiales et finales d'une méthode sont héritées d'une manière imbriquée. La première partie initiale à exécuter dans une méthode est la partie initiale de la classe la plus spécifique. La partie initiale suivante est celle de la super-classe qui suit dans l'ordre ascendant jusqu'à l'exécution de la partie initiale de la classe la plus éloignée du graphe d'héritage. Pour les parties finales l'ordre d'exécution est exactement l'inverse de celui des parties initiales: on exécute d'abord la partie finale de la classe la plus éloignée puis celle héritée de sa sous-classe et ainsi de suite jusqu'à celle de la classe la plus spécifique. La seule partie principale exécutée est héritée de la classe la plus spécifique. La figure 1.15 montre les définitions d'une méthode M dans une classe A et dans sa sous-classe B ainsi que le résultat de l'exécution de la méthode M dans la classe B.

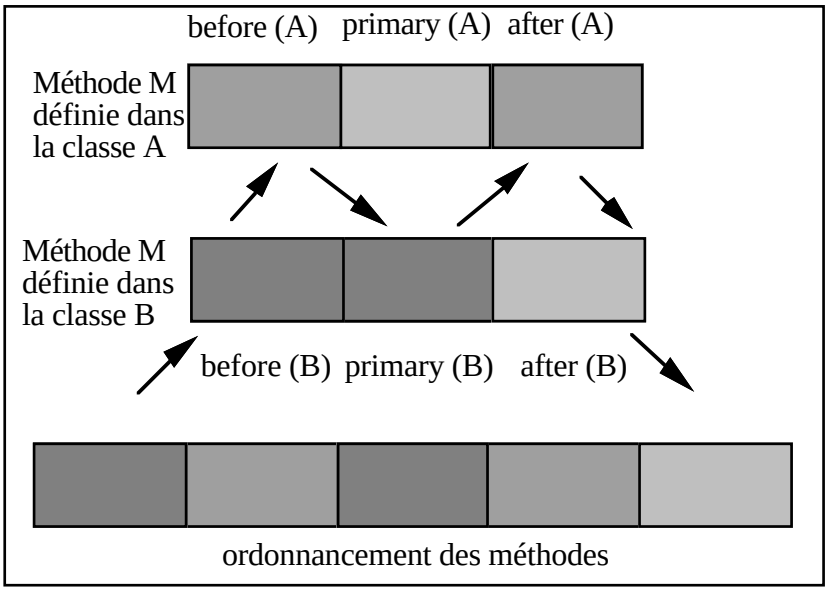


Figure 1.15. Exemple de la spécialisation déclarative des méthodes de Flavors.

CommonLoops [BOB85] et Clos [DEM87] sont des systèmes à fonctions génériques implantant la combinaison déclarative et procédurale de méthodes. Ceci permet l'utilisation des techniques de Loops (la combinaison des méthodes est sous la responsabilité des

consommateurs) et de Flavors (la combinaison des méthodes est sous la responsabilité des producteurs).

1.3.3 Réflexions

Les systèmes à envoi de messages ont l'avantage d'offrir une approche plus intuitive car il existe une relation directe entre l'objet recevant un message et la méthode, propre ou héritée, de la classe d'appartenance de cet objet. Cependant, l'envoi de messages, ne prenant en compte qu'un des arguments de la méthode (i.e l'objet recevant le message), n'offre pas une solution satisfaisante aux problèmes posés par les méthodes multi-classes.

L'approche des fonctions génériques de Clos [DEM87] ou de C++ [STR89] offre une solution qui est une généralisation de celle de l'envoi de messages car tous les arguments sont pris en compte pour sélectionner la méthode la plus appropriée.

L'implantation des fonctions génériques de Clos résout certains des problèmes des méthodes multi-classes. Cependant, on peut regretter sa faible capacité d'expression au niveau des restrictions des arguments car seuls deux types de spécialiseurs sont permis. De plus les arguments optionnels ne sont pas pris en compte pour la sélection des méthodes. Ils ne représentent que des information additionnelles.

Une modélisation de méthodes de calcul adaptée à la CAO doit être très précise en ce qui concerne la sélection des méthodes à exécuter. En effet, certains calculs pouvant demander plusieurs heures, le choix de la méthode la plus appropriée est décisif. Pour cela, nous croyons que les restrictions des arguments d'une fonction générique méritent plus d'importance. On envisage, par exemple, la spécialisation des arguments par des contraintes procédurales et la prise en compte des arguments optionnels lors de la sélection de méthodes.

D'autre part, dans les fonctions génériques de Clos [DEM87], le traitement des arguments est inégal, les premiers ont en effet plus importance. De plus, l'ordre donné aux super-classes d'une classe afin de trouver une linéarisation est arbitraire. Un changement d'ordre de super-classes ou des arguments d'une méthode peut entraîner un changement dans l'ordonnancement des méthodes applicables.

Les deux types de spécialisation de méthodes, déclaratif et procédural, sont des notions orthogonales qui offrent chacune des avantages. A l'instar de CommonLoops[BOB85] et Clos [DEM87], nous retiendrons ces deux notions.

1.3.4 Récapitulatif des aspects procéduraux

Le tableau suivant récapitule les aspects procéduraux qui ont été abordés dans cette section. Pour chaque notion, on présente l'approche retenue pour notre modèle ainsi que les systèmes qui nous ont inspiré cette approche.

Notion		Approche Shood	Origine
			LOO
Sélection de la meilleure méthode		fonctions génériques	Clos [DEM87] C++ [STR86]
Réutilisation des Méthodes	Spécialisation déclarative	Séparation des méthodes en traitements	Flavors [MOO86] Clos [DEM87] CommonLoops [BOB85]
	Spécialisation procédurale	appels procéduraux de méthodes (super, domethod, next-method, etc.)	Smalltalk-80 [GOL83] Clos [DEM87] Loops [BOB81] CommonLoops [BOB85]

1.4 L'extensibilité

Un des défis des applications de CAO est le support de l'évolution de leurs besoins. En effet, il n'est pas rare que de nouveaux types de classes ou de relations entre attributs s'avèrent nécessaires. Un modèle supportant des applications de CAO doit pouvoir évoluer au fur et à mesure que les besoins évoluent. La modélisation d'un type d'évacuateur à série limitée fournit un bon exemple d'un besoin nécessitant une extension du système. La figure 1.16, en montre une solution. Il s'agit d'une classe ayant un nombre maximum d'instances. Si l'expression d'une cardinalité maximale de l'ensemble des instances d'une classe n'a pas été prévue, le système nécessite alors une extension.

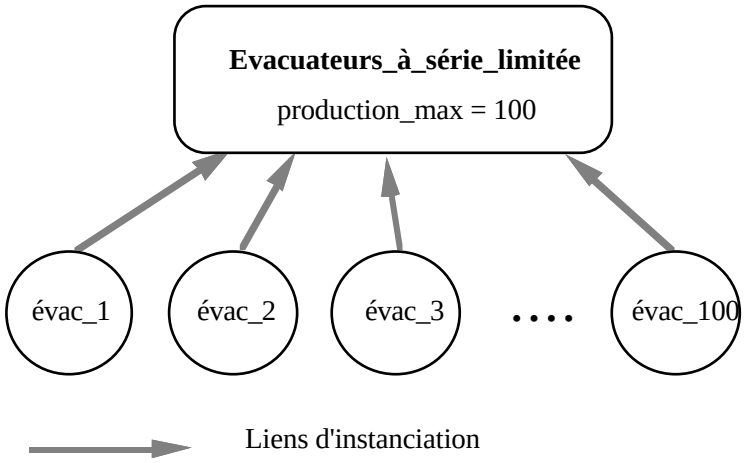


Figure 1.16. Besoin de modéliser une classe ayant un nombre maximum d'instances. La production de ce type d'évacuateurs est limitée à 100 unités.

1.4.1 La méta-connaissance

La méta-connaissance est une connaissance qui porte sur des connaissances [PIT90]. Dans SRL [FOX84], la méta-connaissance (appelée méta-information) sur une classe peut être une description de la classe en langage naturel (figure 1.17). Cette information, aussi banale qu'elle paraisse, n'est pas seulement utilisée pour documenter la classe mais aussi comme une donnée d'entrée pour un système d'explications. Cette information souligne déjà l'un des intérêts de la méta-connaissance: le système est capable de répondre à des questions sur lui-même.

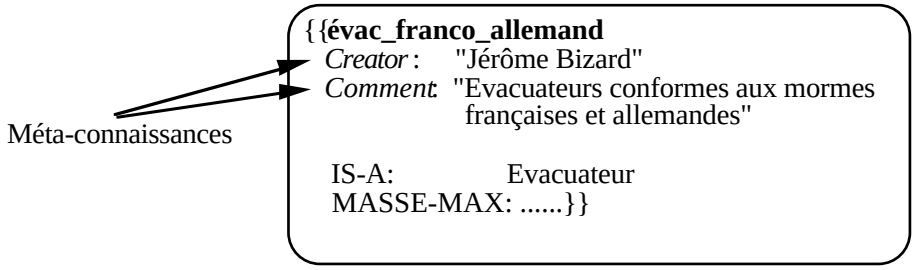


Figure 1.17. La méta-information en SRL. Elle permet de documenter les classes. Elle peut aussi être utilisée pour des explications.

Par contre, la simple expression de méta-connaissances sous la forme de commentaires ne permet pas l'extension des concepts du modèle. Un système qui permet l'expression de méta-connaissances n'est donc pas extensible per se. Pour qu'un modèle soit facilement extensible, il doit être réflexif.

1.4.2 La réflexivité: les méta-classes

La réflexivité est un cas particulier de méta-connaissance. Un modèle réflexif est un modèle qui possède une représentation de lui-même [MAS89]. Les modèles réflexifs proposent un cadre idéal pour les extensions car il peuvent modifier leur propre comportement. Il peuvent aussi répondre à des questions sur eux-mêmes. En IA, la réflexivité permet d'augmenter le pouvoir d'expression et de déduction des systèmes. Dans le monde objet, de plus en plus de systèmes ont recours à la réflexivité. Smalltalk-80 [GOL83], KRS [STE85] [MAE86], Mering [FER84] et ObjVlisp [COI87] en sont des exemples.

Un des premiers langages orientés objet permettant l'expression de méta-connaissances est Smalltalk-80. Dans Smalltalk-80 [GOL83], toute classe est une instance d'une classe d'ordre supérieur appelée méta-classe. Les méta-classes détiennent donc la structure et les comportement des classes. Elles en définissent donc les attributs (nom_classe, super, attributs,

etc.) ainsi que la méthode permettant la création d'un objet (new). Dans la figure 1.18 la méta-classe Evac_à_série_limitée Class modélise le comportement des classes ayant un nombre limité d'instances. Les méta-classes en Smalltalk-80 ne peuvent avoir plus d'une classe comme instance. Ceci nous oblige à définir une méta-classe (Poulie_à_série_limitée Class) pour la classes des poulies à production limitée. De plus, la profondeur du graphe d'instanciation est restreinte à 3 niveaux: méta-classes, classes et instances. Loops [BOB81] est un système hybride écrit en Lisp doté d'un méta-niveau. Les méta-classes de Loops ne connaissent pas la première restriction sur les méta-classes de Smalltalk: une même méta-classe peut avoir plusieurs classes comme instances. Par contre le graphe d'instanciation de Loops, comme celui de Smalltalk, est restreint à 3 niveaux.

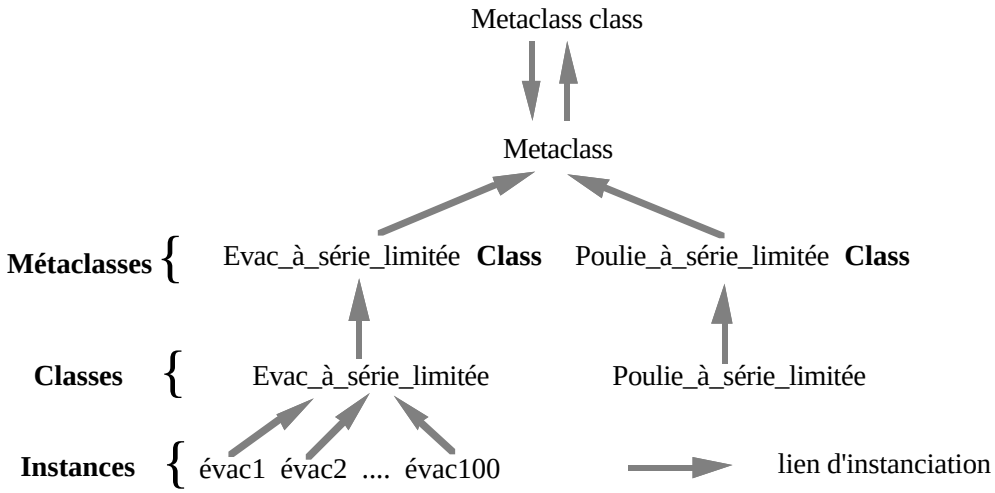


Figure 1.18. Dans Smalltalk-80, une méta-classe ne peut avoir plus d'une classe comme instance. Ceci interdit la création d'une seule méta-classe Méta_à_maximum_d'instances commune aux évacuateurs et aux poulies.

ObjVlisp [COI87] est un modèle minimal écrit en Vlisp conçu dans le but d'expérimenter l'implantation des systèmes à objets. ObjVlisp résout d'une manière élégante les limitations de Smalltalk-80 et Loops en uniformisant la représentation des méta-classes, classes et instances. La différence entre méta-classes, classes et instances est faite par les liens de spécialisation et d'instanciation (figure 1.19):

• Toute classe est instance de Class, la méta-classe primitive. Class contient la définition de la structure (les attributs) des classes.

• Toute classe génératrice d'instance terminales hérite d'Objet, la racine du graphe de spécialisation. La méthode new définie dans Objet permet la création d'instances terminales.

• Toute méta-classe hérite de Class. La méthode new définie dans Class permet la création des

classes.

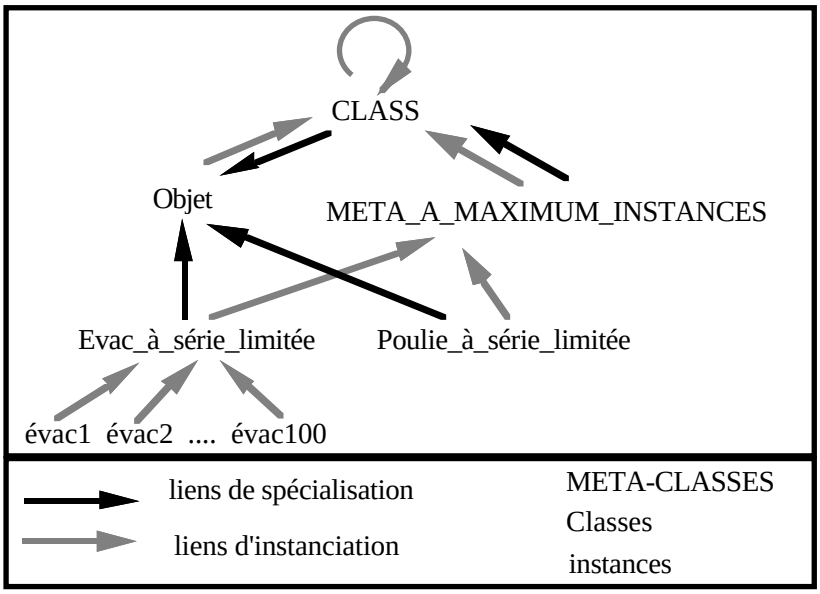


Figure 1.19. Les méta-classes dans ObjVlisp. META_A_MAXIMUM_INSTANCES modifie la méthode new héritée de CLASS, la méta-classe primitive. La nouvelle méthode new limite le nombre d'instances que ses classes peuvent avoir.

Le graphe d'instanciation d'ObjVlisp n'est donc pas limité à trois niveaux comme Loops ou Smalltalk-80. L'intérêt d'un nombre de niveaux supérieur à trois est cependant limité. En effet, d'après [PIT90]:
"Il est parfois utile de considérer deux métaniveaux, rare de devoir considérer trois et exceptionnel d'aller au quatrième métaniveau: il est important d'expliquer pourquoi un résultat à été obtenu, parfois intéressant d'expliquer comment une explication a été obtenue, mais faut-il expliquer pourquoi un système a trouvé telle explication d'une explication?"

1.4.3 L'uniformisation: le tout objet

Dans le monde objet, les systèmes ayant une représentation uniforme de leurs concepts ont l'appellation "tout objet". Shirka [REC88] est, comme ObjVlisp, un système "tout objet" mais allant plus loin que ce dernier car les définitions des attributs et des méthodes sont aussi représentées par des objets. Mering II [FER84] et PtitLoo [FER87] poussent l'idée du tout objet encore plus loin. Dans ces systèmes, les définitions d'attributs sont représentées par des classes instances d'une méta-classe particulière MetaSlot (figure 1.20). Les valeurs d'un attribut sont donc représentées par des instances de cette classe. FROME [CAR91], Yafool [DUC88] et KOOL [KOO87] développent des idées similaires.

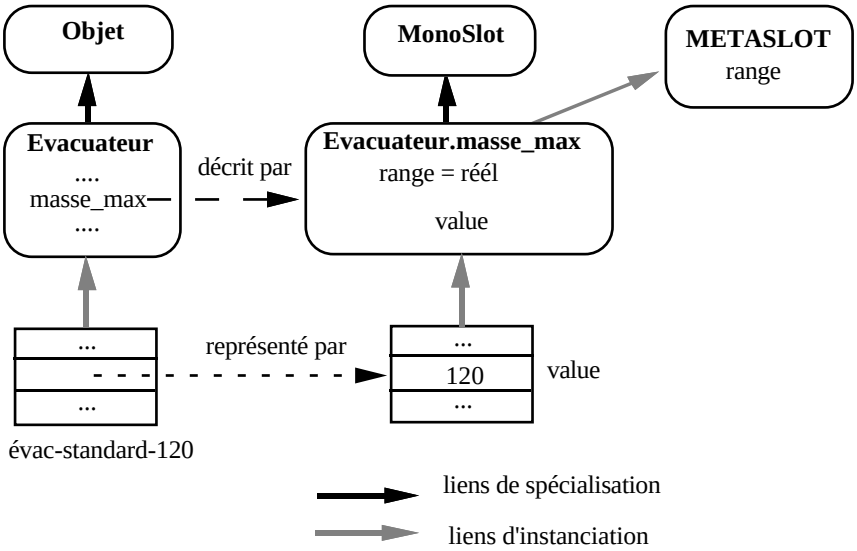


Figure 1.20. Dans PtitLoo [FER88] les attributs (masse_max de la classe Evacuateur) sont décrites par des classes (Evacuateur.masse_max) dont ses instances contiennent les valeurs (value=120).

1.4.4 Réflexions

Les systèmes réflexifs relèvent le défi de l'évolution des applications CAO. Leurs caractéristiques sont résumées dans [MAS89]:
"Un système réflexif est "ouvert" dans le sens où il peut être enrichi avec des utilitaires et des bibliothèques écrits dans le langage lui-même. Il est convivial, car chaque utilisateur peut à son tour modifier ces outils ou en créer d'autres, selon ses besoins. Enfin, il est dynamique, car il peut évoluer au cours du temps, en changeant sa propre description."

La réflexivité permet aussi d'augmenter le pouvoir d'expression et de déduction des systèmes. Elle leur permet également de répondre à des questions sur eux-mêmes, ce qui facilite l'implantation des systèmes d'explications.

Les systèmes réflexifs uniformisés ("tout objet") facilitent l'accès au méta-niveau (i.e la définition des concepts du modèle). En effet, les concepts du modèle peuvent être manipulés plus aisément car il sont représentés de la même manière que les connaissances relatives aux applications: par des objets.

Le prix à payer pour un système réflexif peut être résumé en deux points. D'une part, le temps d'exécution se voit augmenté par l'utilisation des méta-niveaux. Par exemple, l'accès aux attributs représentés par des classes, implique un coût en temps d'exécution: le système doit monter au niveau méta pour obtenir une valeur. D'autre part, l'utilisation des méta-classes par l'utilisateur final n'est pas évidente car les méta-classes ne sont pas faciles à assimiler

[BOR86].

Finalement, les méta-classes ne sont pas souhaitables dans les langages de programmation destinés au génie logiciel. Ceci est le cas des langages compilés tels que C++ [STR89], Simula [DAH66] et Eiffel [MEY89] dans lesquels toutes les classes doivent être connues lors de la compilation. En effet, les méta-classes permettent la création de classes lors de l'exécution. Elles offrent donc une solution au besoin d'évolution des applications. Pour pallier aux risques d'incohérence que cette approche peut entraîner, nous fournirons un cadre dans lequel l'évolution des schémas et des instances est gérée.

Nous retiendrons donc un modèle réflexif, “tout objet” décrit et implanté par des méta-classes.

1.4.5 Récapitulatif des aspects réflexifs

Le tableau suivant récapitule les aspects réflexifs qui ont été abordés dans cette section. Pour chaque notion, on mentionne l'approche retenue pour notre modèle ainsi que les systèmes qui nous ont inspiré.

Notion	**Approche Shood**	**Origine**	
		RCO	**LOO**
Réflexivité	Méta-classes (n niveaux)	KRS [STE85]	ObjVlisp [COI87]
Uniformisation	Le "tout objet"	Mering [FER84] FROME [CAR91] Yafool [DUC88] KOOL [KOO87] SRL [WRI84]	Clos [DEM87]

1.5 Conclusions

Nous avons examiné certaines caractéristiques des systèmes orientés objet pour tester leur adéquation aux applications CAO. Au cours de cet examen nous avons fait un certain nombre de choix que nous pouvons utiliser comme cahier des charges et que nous rappelons brièvement.

Le modèle fait une distinction entre classes et instances. Les premières représentent des abstractions ou descriptions génériques. Les secondes représentent des réalisations ou des descriptions individuelles. Les classes sont organisées en un graphe de spécialisation. La spécialisation est réalisée par un mécanisme d'héritage multiple de sémantique ensembliste. La

portée des noms des attributs est déterminée par leur classe d'origine. Ceci permet la solution des conflits de noms dus à l'héritage multiple.

Il n'existe pas un mécanisme d'encapsulation stricte. Celle-ci peut éventuellement être réalisée par convention en utilisant les méthodes. Les attributs sont définis par des descripteurs déclaratifs et procéduraux. Les descripteurs déclaratifs permettent de restreindre les valeurs qu'un attribut peut prendre. Lors de la redéfinition des attributs hérités (affinage), le système est capable d'assurer que les nouvelles définitions des descripteurs déclaratifs correspondent à un sous-domaine de l'attribut hérité. Deux descripteurs procéduraux sont prévus. L'un permet de restreindre les valeurs qu'un attribut peut prendre (contrainte procédurale). Le système ne peut pas assurer l'inclusion d'ensembles pour ce descripteur. L'inclusion ensembliste pour les contraintes procédurales est donc faite par convention sous la responsabilité de l'utilisateur. L'autre descripteur procédural permet de calculer la valeur d'un attribut au moment de la création d'un objet (inférence de valeur).

Les méthodes sont définies hors des classes. Elle sont multi-classes et s'inspirent du concept de fonction générique. Tous les arguments des méthodes ont la même importance. Des descripteurs déclaratifs et des contraintes procédurales peuvent être attachés aux arguments d'une méthode. Ceci permet de mieux sélectionner la méthode appropriée. Les méthodes peuvent être réutilisées (spécialisées) de façon déclarative ou procédurale.

Le système est méta-réflexif. Tout objet est instance d'une classe et toute classe est instance d'une méta-classe. Tous les concepts du système sont modélisés par des objets. Ces concepts peuvent donc être facilement étendus par l'adjonction de nouvelles méta-classes. Une attention spéciale est portée à la modélisation des attributs. Ceux-ci doivent pouvoir être étendus pour prendre en compte le concept de relation sémantique.

Le graphe d'instanciation n'a pas de limitation en nombre des niveaux. L'instanciation est multiple. Tout objet peut donc être instance de plus d'une classe à la fois. La sémantique de la multi-instanciation respecte la sémantique ensembliste du mécanisme d'héritage.

2. Shood: Les aspects déclaratifs

2.1. Introduction

Ce chapitre introduit les aspects déclaratifs du modèle Shood [ESC90a] [NGU91] [NGU91a] [RIE92a]. Le point est fait sur leur utilité pour résoudre des problèmes de CAO [RIE91] [RIE92].

Les concepts de base de Shood sont d'abord présentés. Leur prise en compte a déjà été justifiée dans l'état de l'art. L'héritage permet la réutilisation de concepts. L'héritage multiple augmente cette réutilisation en permettant de combiner les descriptions de plusieurs classes. L'instanciation multiple permet d'exprimer qu'un objet est instance de plus d'une classe à la fois. Le lien de disjonction entre classes permet d'accroître la puissance de représentation du modèle. Nous montrons ici que la fusion de toutes ces caractéristiques dans un même modèle n'est pas contradictoire.

Puis, nous abordons le niveau méta qui sera présenté plus en détail au chapitre 4. Les graphes de spécialisation et d'instanciation sont également montrés.

Nous montrons ensuite les solutions données aux problèmes techniques de l'héritage multiple (et de la multi-instanciation). Les conflits de noms sont résolus par la notion de nom-complet d'attribut et d'attribut pré-déclaré. Les conflits de descripteurs de type sont résolus par une relation de sous-type.

Puis, nous montrons que l'extensibilité du modèle n'est pas sans retombées sur la puissance de représentation: la modélisation des attributs par des classes permet, par exemple, d'enrichir facilement la sémantique des attributs. Les notions d'attribut d'attribut[1] et d'attribut obligatoire sont présentées au travers d'exemples.

Finalement, nous présentons la définition formelle des mécanismes d'héritage et d'instanciation.

---

[1] ce n'est pas une répétition accidentelle!

2.2. Les concepts de base

2.2.1. Classes, attributs et descripteurs

Un des buts de Shood est de fournir un modèle permettant de représenter des connaissances d'une manière organisée, c'est-à-dire en regroupant les connaissances similaires. Ainsi, les objets de mêmes caractéristiques sont regroupés dans des *classes* [STE86]. Les classes définissent la structure des objets à l'aide d'attributs. Les *attributs* représentent les caractéristiques des objets. Ces caractéristiques peuvent modéliser une fonctionnalité de l'objet ("descendre" pour un évacuateur), un composant de l'objet (une poulie), une propriété qualitative (a des ressorts) ou quantitative (nombre de ressorts). Les attributs sont définis par des descripteurs. La structure à trois niveaux (classe/attribut/descripteur) de Shood (figure 2.1) s'inspire donc de celle des systèmes à frames (schéma/attribut/facette) [REC85].

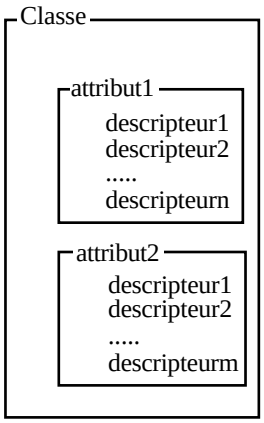


Figure 2.1. La structure des classes.

Au moment de définir un attribut, en plus de son nom, nous précisons s'il est mono ou multi-valué[1] et le domaine où il peut prendre ses valeurs. Ceci est fait grâce au descripteur de *type* (figure 2.2).

```
(nom_classe   Evacuateurs
 attributs    (masse_max   (type (un Réel))
               vitesse_max  (type (un Réel))
               )
)
```

Figure 2.2. La classe des Evacuateurs

[1]Le constructeur des attributs mono-valués est *un*. Il existe plusieurs types de constructeurs multi-valués; *ens* est celui qui modélise les ensembles.

2.2.2. La réutilisation: la spécialisation et l'héritage

Shood permet la réutilisation des structures (i.e des descriptions de classes). En effet, il autorise la définition d'un nouveau concept par raffinement d'un concept existant. Par exemple, le concept des Evacuateurs conformes aux normes françaises (Evac_français) peut être défini en raffinant le concept Evacuateurs.

Les systèmes utilisant les objets donnent des sémantiques différentes à leurs liens de spécialisation [BRA83]. Cette sémantique dépend en partie du rôle joué par les classes. Dans Shood, nous rattachons aux classes deux rôles différents: celui du "moule" définissant la structure (attributs) de ses instances et celui du "conteneur" (ensemble) de ses instances. La spécialisation définit donc deux aspects:

• Structurel
Une classe hérite de la définition de sa super-classe. Elle peut affiner cette définition soit par raffinement d'un attribut (raffinement du domaine par exemple) soit par l'ajout d'un nouvel attribut. Ces deux opérations ne sont pas exclusives.

• Ensembliste
L'inclusion d'ensembles doit être respectée. Dans Shood une classe modélise donc un sous-ensemble de l'intersection des ensembles représentés par toutes ses super-classes: la spécialisation est stricte.

Le mécanisme d'héritage de Shood correspond au moins à trois des quatres aspects de l'héritage soulévés dans [ATK89a]. Ces trois aspects sont basés sur la structure (i.e les attributs) et non pas sur le comportement (i.e les méthodes):

- *nclusion*, tout objet d'une classe est aussi un objet de sa super-classe;
- *contrainte*, une classe peut restreindre les attribus hérités;
- *spécialisation*, une classe peut ajouter de nouvelles informations (i.e des attributs).

Dans la figure 2.3 nous représentons la spécialisation par une flèche noire allant de la classe Evac_français vers sa super-classe Evacuateurs. La classe Evac_français est raffinée par l'ajout de l'attribut *homologation*.

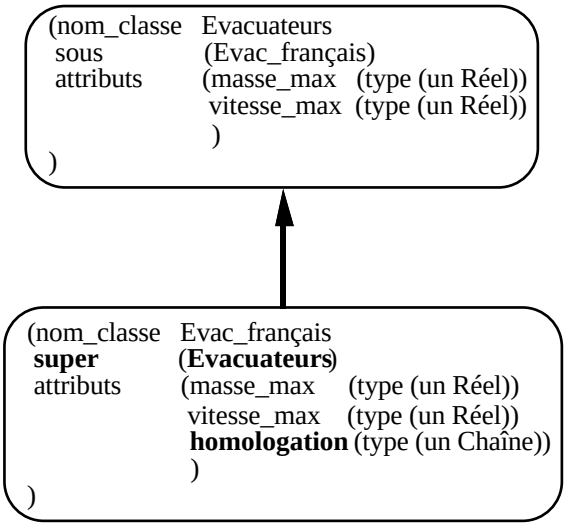


Figure 2.3. Evac_français réutilise la structure des Evacuateurs par le biais du lien de spécialisation.

Le lien de spécialisation est transitif. La super-classe de la super-classe d'une classe C est aussi une super-classe de C. Les super-classes définies explicitement comme étant super-classes d'une classe sont appelées *super-classes directes*. Inversement les super-classes obtenues par transitivité sur le lien de super-classe sont appelées *super-classes indirectes*.

2.2.3. Plus de réutilisation: la spécialisation multiple

Pour augmenter le pouvoir de réutilisation Shood, permet la *spécialisation multiple*. Celle-ci autorise une classe à avoir plusieurs super-classes. Les possibilités de partage de propriétés sont augmentées car les descriptions de plusieurs classes peuvent être combinées.

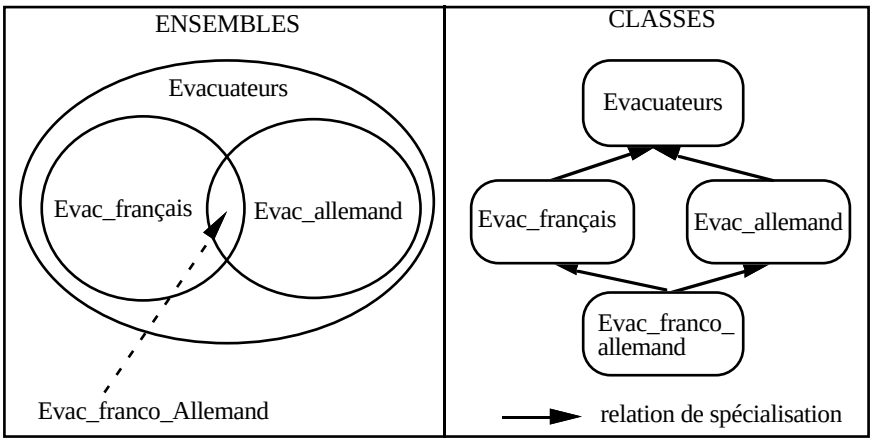


Figure 2.4. La spécialisation multiple. Une classe modélise un sous-ensemble de l'intersection des ensembles représentés par toutes ses super-classes.

Afin de respecter l'inclusion d'ensembles, une classe modélise un sous-ensemble de l'intersection des ensembles représentés par toutes ses super-classes directes. Dans la figure 2.4 nous définissons la classe Evac_franco_allemand qui représente, d'un point de vue ensembliste, l'intersection de Evac_français et de Evac_allemand. Dans certains systèmes tel que Flavors [MOO86], l'ordre de définition des super-classes détermine un ordre de priorité pour la résolution de conflits d'héritage (cf § état de l'art). Etant donné que dans Shood les classes représentent des ensembles, leur ordre de définition n'est pas important.

La spécialisation multiple augmente en contrepartie les possibilités de conflits de noms et de conflits de valeurs d'attributs. La section § Les conflits d'attributs est consacrée à leur solution. D'autre part, une définition formelle de l'héritage à l'aide d'invariants, inspirée des travaux de [BAN87] et [PEN87], est présentée dans la section § Les invariants d'héritage.

2.2.4. L'instanciation

Les objets appartenant à une classe sont appelés *instances* de la classe. La relation qui lie un objet à sa classe est appelée *relation d'instanciation* [WOO75]. Nous la représentons par une flèche claire allant de l'objet vers son objet-moule dans la figure 2.5. Nous appellerons cette relation le lien d'instanciation ou le lien instance_de. En plus de l'appartenance à la classe, ce lien à deux implications:

• L'instance value les attributs (*masse_max, vitesse_max*) définis par sa classe.

• Ces valeurs doivent respecter les définitions relatives aux attributs de la classe d'appartenance de l'objet.

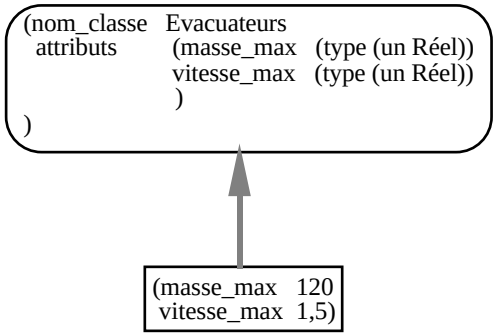


Figure 2.5. Lien d'instanciation

Dans Shood tout, objet est instance d'au moins une classe et il ne peut instancier que les attributs de sa (ses) classe(s). Par transitivité sur le lien de spécialisation et par inclusion ensembliste, un objet appartenant à une classe appartient aussi aux super-classes de cette classe.

2.2.5. L'identification des objets

Dans les systèmes orientés objet, tous les objets du système ont un identificateur unique appelé identificateur de l'objet (oid). Toutes les relations entre les objets sont représentées en stockant les oids des objets en jeu. Ainsi un attribut qui référence un objet a comme valeur l'oid de cet objet. Cette notion d'identité d'objet est plus forte que la notion de clef des bases de données [PAT88] car elle garantit l'intégrité de référence aux objets. En effet, la modification de la valeur d'un attribut d'un objet n'a aucun effet sur son oid. Puis, si l'objet est supprimé, c'est à la responsabilité du système d'assurer qu'aucun objet existant n'y fait plus référence.

La notion d'identité d'objet est retenue dans le modèle de Shood mais elle coexiste avec une notion de clef définie par l'utilisateur. En effet, l'introduction des clefs dans un système orienté objet facilite, comme nous allons le voir, les opérations d'ajout et de modification des objets du système. Par ailleurs, dans Shood un oid peut être soit défini par l'utilisateur ("mon-oid") soit alloué automatiquement par le système ("g2132").

2.2.6. L'instanciation multiple

Supposons maintenant qu'un évacuateur particulier, l'évacuateur standard 120 soit conforme aux normes françaises et allemandes. Nous pouvons représenter la vérification de ces deux normes en faisant de l'objet *standard-120* une instance des deux classes Evac_français et Evac_allemand (figure 2.6).

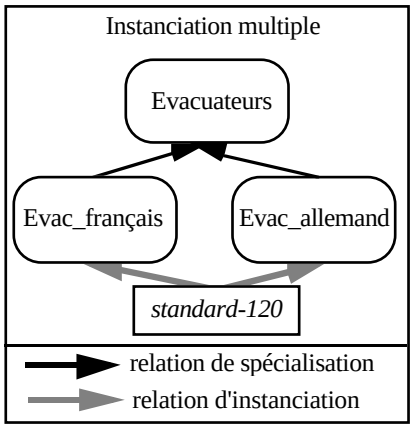


Figure 2.6. Multi-instanciation: *standard-120* est à la fois instance de Evac_français et de Evac_allemand.

Il s'agit ici d'un cas de *multi-instanciation* car un objet est instance de plus d'une classe à la fois. L'objet multi-instancié doit respecter les définitions de toutes ses classes d'appartenance.

Quels sont les rapports entre la spécialisation multiple et l'instanciation multiple? La spécialisation multiple est plus générale que l'instanciation multiple car elle modélise un sous-ensemble de l'intersection des ensembles représentés par les classes en question. Par exemple, Evac_rapide_franco_allemand, la classe des évacuateur rapides de norme franco-allemande, obtenue en spécialisant les classes Evac_français et Evac_allemand et en contraignant l'attribut *vitesse_descente* à plus de 1,4 m/s, représente un sous-ensemble de l'intersection des classes Evac_français et Evac_allemand. L'instanciation multiple se contente de modéliser l'appartenance à une "simple" intersection d'ensembles.

2.2.7. La disjonction

Parfois, il est possible de déterminer a priori que les ensembles représentés par deux classes sont disjoints, c'est-à-dire que ces deux classes n'ont aucune instance en commun. La possibilité de représenter cette connaissance augmente la puissance de représentation du système. Par exemple, cette connaissance déclarative peut être utilisée par des algorithmes de classification pour augmenter leurs performances. En effet, s'il est connu qu'une instance appartient à une classe donnée, l'algorithme n'essaie pas de la classifier dans une classe disjointe [BRA91] [LIO93] [MAR89].

Toute classe peut exprimer un lien de disjonction. La figure 2.7 montre l'utilisation de cette relation pour les classes Evacuateurs et Vis. Cette relation est symétrique: A est disjointe de B et vice versa. Il suffit donc d'exprimer dans une classe qu'elle est disjointe d'une autre; l'attribut *disjoint* de la classe en disjonction est automatiquement mis à jour. Mais elle n'est pas transitive: A disjointe de B et B disjointe de C n'implique pas A disjointe de C, la relation doit être explicitée.

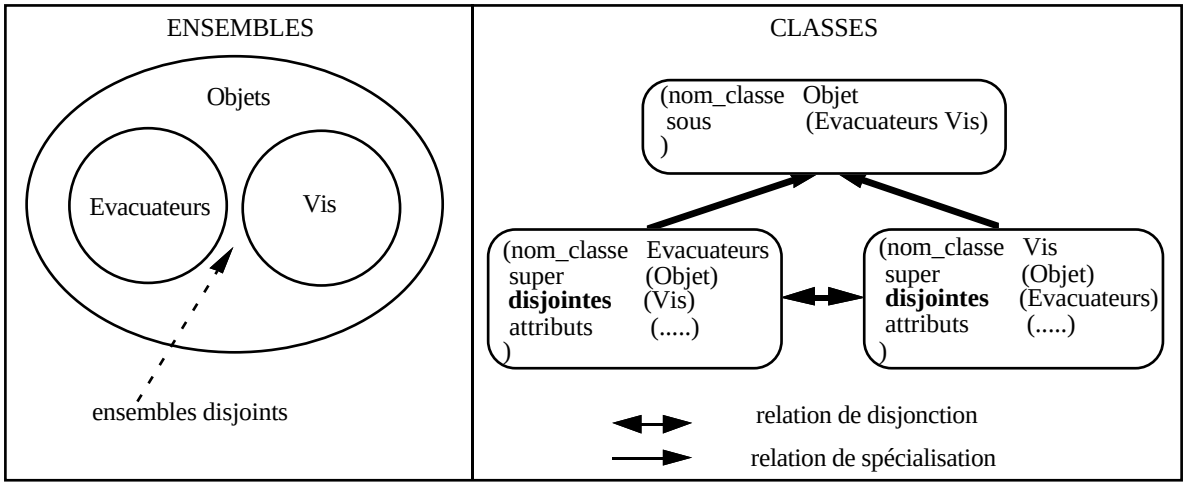


Figure 2.7. La relation de disjonction est exprimée par l'attribut *disjointes*.

Deux classes qui maintiennent une relation de disjonction n'admettent pas d'instances en commun et donc, n'admettent pas de spécialisation commune [ESC89]. Remarquons que deux classes qui sont chacune spécialisation directe d'une même classe représentent par défaut des ensembles non disjoints; une instance peut appartenir aux deux classes à la fois.

2.2.8. Récapitulatif

Dans l'état de l'art de cette thèse, nous avons dégagé les concepts des systèmes orientés objet qui nous ont parus nécessaires de prendre en compte dans un modèle de représentation de connaissances pour la CAO. Ces concepts ont leur origine dans des systèmes dont les notions ne sont pas forcément compatibles. Entre autres, nous pouvons mentionner des Langages Orientés Objet: ObjVlisp [COI87], Clos [DEM87], CommonLoops [BOB85] et de la Représentation de Connaissances Centrée Objet: Shirka [REC85], KRL [BRA85a], Mering [FER84].

Nous avons montré ici que la fusion de ces caractéristiques dans un même modèle n'est pas contradictoire. Pour définir un ensemble cohérent de concepts, nous avons utilisé la théorie des ensembles. Le rapprochement des concepts Shood avec les notions ensemblistes est montré dans le tableau récapitulatif suivant:

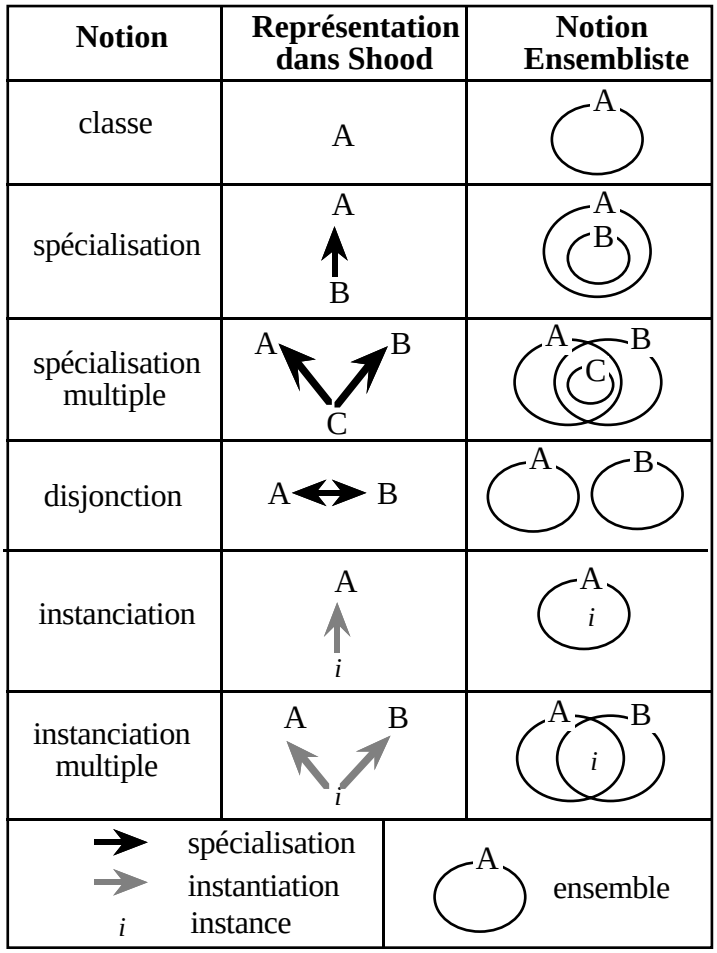


Figure 2.8. Ce tableau récapitule les notions de base de Shood. Nous montrons ici la représentation graphique de chaque concept ainsi que la notion de la théorie des ensembles à laquelle il se rapporte.

2.3. Vers l'extensibilité

2.3.1. Le niveau méta

L'extensibilité étant l'un des buts de Shood, nous avons choisi d'implanter un modèle réflexif [MAS89]. Toute connaissance est donc représentée à partir d'un concept unificateur: l'objet. Une connaissance élémentaire est représentée par un objet (l'instance *dupont*) mais aussi une connaissance générique (la classe "Personnes"), une connaissance sur un groupe d'objets ("toute personne marche"), une méthode ("la différence entre deux entiers"), une inférence ("l'âge des Personnes est calculé par la différence entre l'année courante et leur année de naissance"), une contrainte ("l'âge des Personnes doit être égal à la différence entre l'année courante et leur année de naissance"). La connaissance sur le modèle est donc modélisée en utilisant ses propres concepts, ce qui permet de la modifier et de l'étendre [PIT90].

Dans Shood, les classes étant des objets, elles sont elles-mêmes instances de classes de plus haut niveau appelées *méta-classes* [STE86]. Une méta-classe, tout comme une classe, définit la structure et le comportement de ses instances. Une *classe-feuille* est une classe qui ne pourra jamais fabriquer de classes (e.g Personnes). Un objet qui ne pourra jamais créer des objets est appelée une *instance-terminale*. Par exemple, *dupont*, objet appartenant à la classe Personnes, est une instance-terminale.

Les classes-feuilles, telles que Personnes, sont instances de la méta-classe appelée META. Mais quelle est la classe d'instanciation de META? META est instance d'elle-même car elle est la classe d'instanciation de toute classe. A l'instar de ObjVlisp [COI87], META est aussi la méta-classe primitive et par conséquent, la racine du *graphe d'instanciation*. Tous les objets (instances-terminales, classes-feuilles et méta-classes) valuent un attribut *instance_de*. La définition de *instance_de* doit donc exister dans toutes les classes (et les méta-classes). Dans la figure 2.9 seuls les attributs *nom_classe*, *instance_de* et *attributs* de META sont montrés.

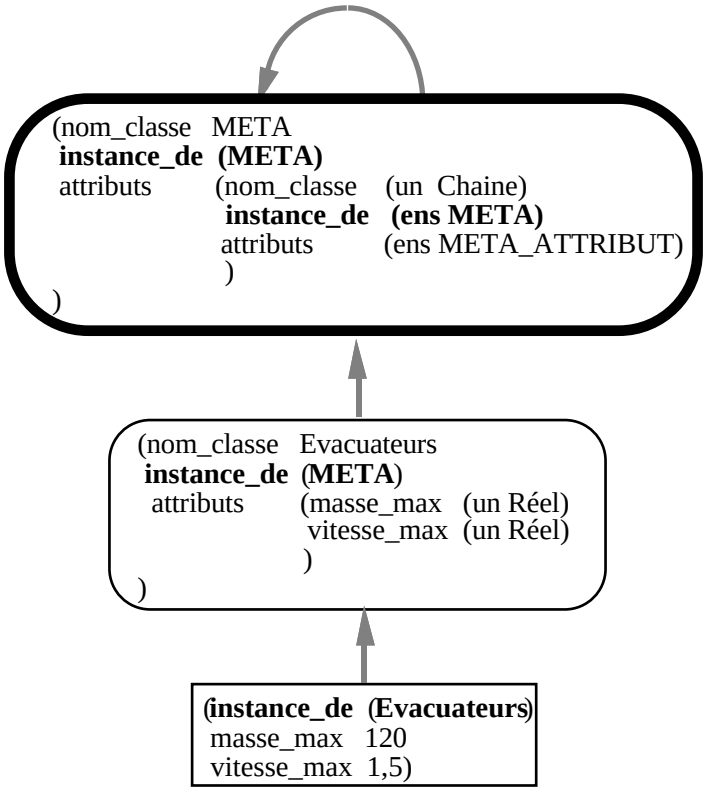


Figure 2.9. Tous les objets Shood sont instances d'une classe.

Graphiquement, les méta-classes seront représentées par des boîtes arrondies en trait gras, les classes-feuilles par des boîtes arrondies en trait simple et les instances terminales par des boîtes rectangulaires (figure 2.9).

Shood offre donc trois niveaux d'objets: les méta-classes, les classes-feuilles et les instances-

terminales. Par la suite les noms des méta-classes sont écrits en majuscules, les noms des classes-feuilles avec la première lettre en majuscule et les noms des instances-terminales en minuscules et en italiques (figure 2.10).

Figure 2.10. Les trois niveaux d'objets: méta-classes, classes-feuilles et instances-terminales.

Les méta-classes et les classes-feuilles sont donc des instances des méta-classes. Les instances-terminales sont instances des classes-feuilles.

2.3.2. Le graphe de spécialisation

Dans Shood, les liens de spécialisation forment un graphe acyclique connexe et à racine unique. Toute classe est donc sous-classe d'au moins une classe à la seule exception de la racine du graphe de spécialisation appelée Univers (figure 2.11).

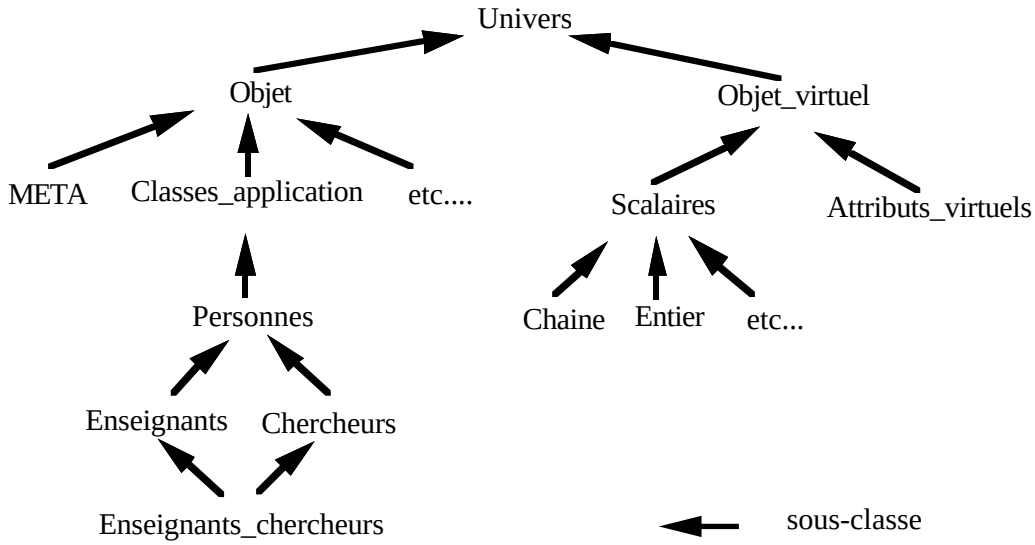


Figure 2.11. Les objets et les objets virtuels

Le graphe de spécialisation se subdivise en deux graphes dénotés par les classes: Objet et Objet_virtuel (figure 2.11).

La classe Objet est la racine du graphe de spécialisation des *classes instanciables*. Une classe est instanciable si elle est définie en intention: pour qu'une personne appartienne à la classe Personnes elle doit être explicitement créée dans cette classe. Objet admet un certain nombre de sous-classes directes pré-définies qui nous permettent de distinguer les classes selon leur rôle. Classes_application et META en sont des exemples. Toute classe-feuille définie par l'utilisateur est une sous-classe de Classes_application. Toute méta-classe est une sous-classe de META.

Par ailleurs, Objet_virtuel est la racine de spécialisation des *classes virtuelles*. Les classes virtuelles sont des classes non-instanciables (i.e sans instances). Les classes virtuelles sont utilisées dans Shood pour représenter les types de base (scalaires) et les attributs virtuels. Scalaires est la classe racine des classes représentant les types scalaires. Entier, Chaîne, Bool, Réel et Code, sous-classes de Scalaires, représentent des types de base. Des nouvelles classes de base peuvent être définies par l'utilisateur en fonction des besoins des applications. Attribut_virtuel est la racine de spécialisation des attributs virtuels. Dans Shood, à l'instar de PtitLoo [FER88], la valeur d'un attribut normal (non virtuel) est modélisée par une instance (chapitre 4 § Modélisation d'attributs). Par contre, les valeurs des attributs virtuels ne sont pas modélisées par des instances. Ceci restreint les manipulations que l'on peut effectuer sur eux. Par exemple, un attribut virtuel ne peut pas avoir d'attributs d'attribut car il n'est pas représenté par une instance (cf. § Les attributs d'attribut).

2.4. Les conflits d'attributs

L'héritage multiple (et la multi-instanciation) permettent d'augmenter le partage de propriétés car les différentes descriptions de classes peuvent être combinées. Le prix de ce partage est l'augmentation de conflits entre les descriptions de classes impliquées. Les différentes solutions données à ces conflits par les systèmes existants ont été présentées dans l'état de l'art de cette thèse (cf. § Conflits d'héritage). Dans ce paragraphe, nous présentons et justifions les solutions données dans Shood aux conflits de noms (sémantique des noms d'attributs) et aux conflits de valeurs (conflits des descripteurs de type).

2.4.1. Les noms des attributs

Dans Shood les attributs d'un objet représentent son état. Pour pouvoir référencer chacune des parties de l'état d'un objet, les attributs sont nommés et tous les attributs définis dans une même classe ont un nom différent.

Les classes d'appartenance d'un objet peuvent être interprétées comme les différents points de vue détenus sur lui. En effet, un évacuateur respectant les normes françaises et allemandes peut être observé selon deux points de vue: comme un évacuateur français ou comme un évacuateur allemand. Cet évacuateur peut donc être modélisé comme un objet appartenant aux deux classes Evac_français et Evac_allemand représentant les deux points de vue. Ainsi, lorsque l'on définit un attribut dans une classe nous pouvons penser que cet attribut représente une caractéristique des objets du point de vue de cette classe. Par exemple l'attribut *homologation* défini dans la classe Evac_allemand représente la connaissance sur l'homologation des objets vus comme des

Evac_allemand. Mais il est aussi possible de définir un attribut *homologation* dans la classe Evac_français représentant l'homologation des objets vus comme des Evac_français.

Un conflit entre les deux attributs *homologation* se présente dans la classe Evac_franco_allemand (figure 2.12), sous-classe commune à Evac_français et Evac_allemand. Quel attribut *homologation* doit être hérité?

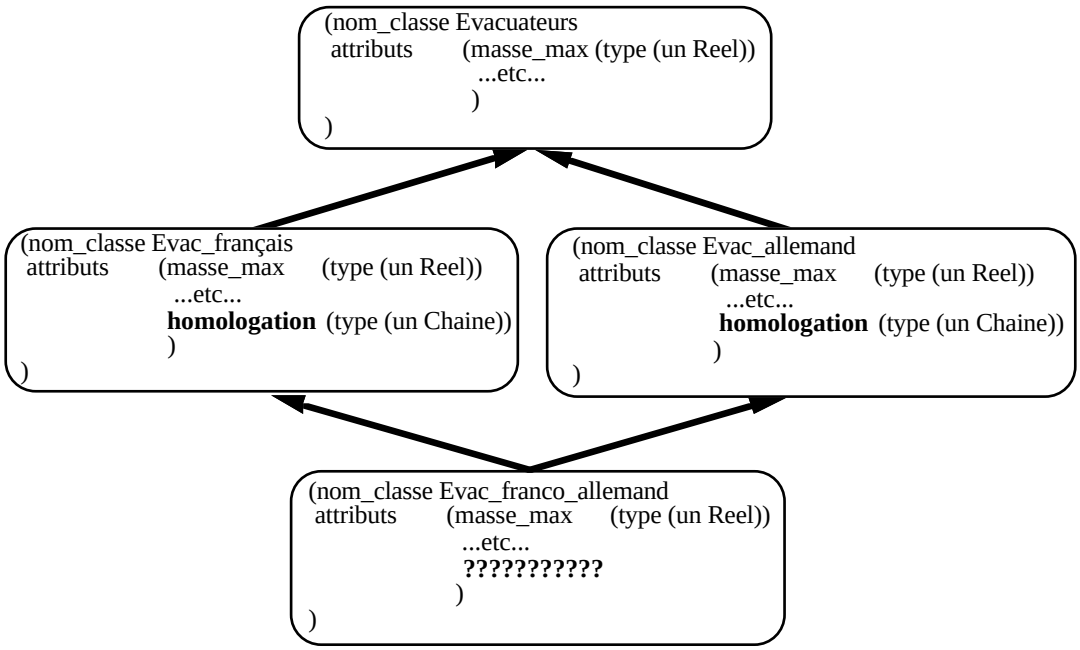


Figure 2.12. Conflit des attributs ayant des classes d'origine différentes

Dans la figure précédente, nous pouvons aussi observer un conflit d'héritage entre deux attributs *masse_max* dans la classe Evac_franco_allemand hérités respectivement de la classe Evac_allemand et de la classe Evac_français. D'un point de vue sémantique, la solution pour ce conflit d'héritage paraît claire: il s'agit toujours du même attribut puisqu'ils ont la même classe d'origine. La *classe d'origine* d'un attribut est la classe où il est défini pour la première fois. Pour *masse_max* la classe d'origine est Evacuateurs. Comment faire la différence entre deux attributs de même nom et de classes d'origine différentes?

Dans Shood, nous partons de l'hypothèse que deux attributs de même nom qui proviennent de classes d'origine différentes n'ont pas la même sémantique. Par conséquent si un concepteur d'une base de connaissances définit deux attributs dans deux classes différentes, c'est parce qu'il veut exprimer que ces attributs sont des attributs sémantiquement différents. Ceci est le cas pour les attributs *homologation* des classes d'évacuateurs français et allemands. L'un exprime la vérification des normes françaises. L'autre exprime la vérification des normes allemandes. Chacun d'eux exprime donc une caractéristique différente de l'objet. Pour déterminer le point de vue pour lequel l'attribut a été défini nous utilisons sa classe d'origine.

Nous définissons pour cela le concept du nom complet d'un attribut.

Le *nom complet* d'un attribut est le nom formé du nom de la classe d'origine et du nom donné à l'attribut dans cette classe. Ainsi, le nom complet de l'attribut *homologation*, défini pour la première fois dans la classe Evac_français, est *evac_français.homologation* et le nom complet de l'attribut *homologation*, défini dans la classe Evac_allemand, est *evac_allemand.homologation*. En utilisant le nom complet de l'attribut, les conflits d'héritage des attributs de même nom mais provenant de classes d'origine différentes sont résolus. Ainsi dans la figure 2.13 on peut voir que le schéma de la classe des Evac_franco_allemand contient la définition des attributs *evac_français.homologation* et *evac_allemand.homologation*.

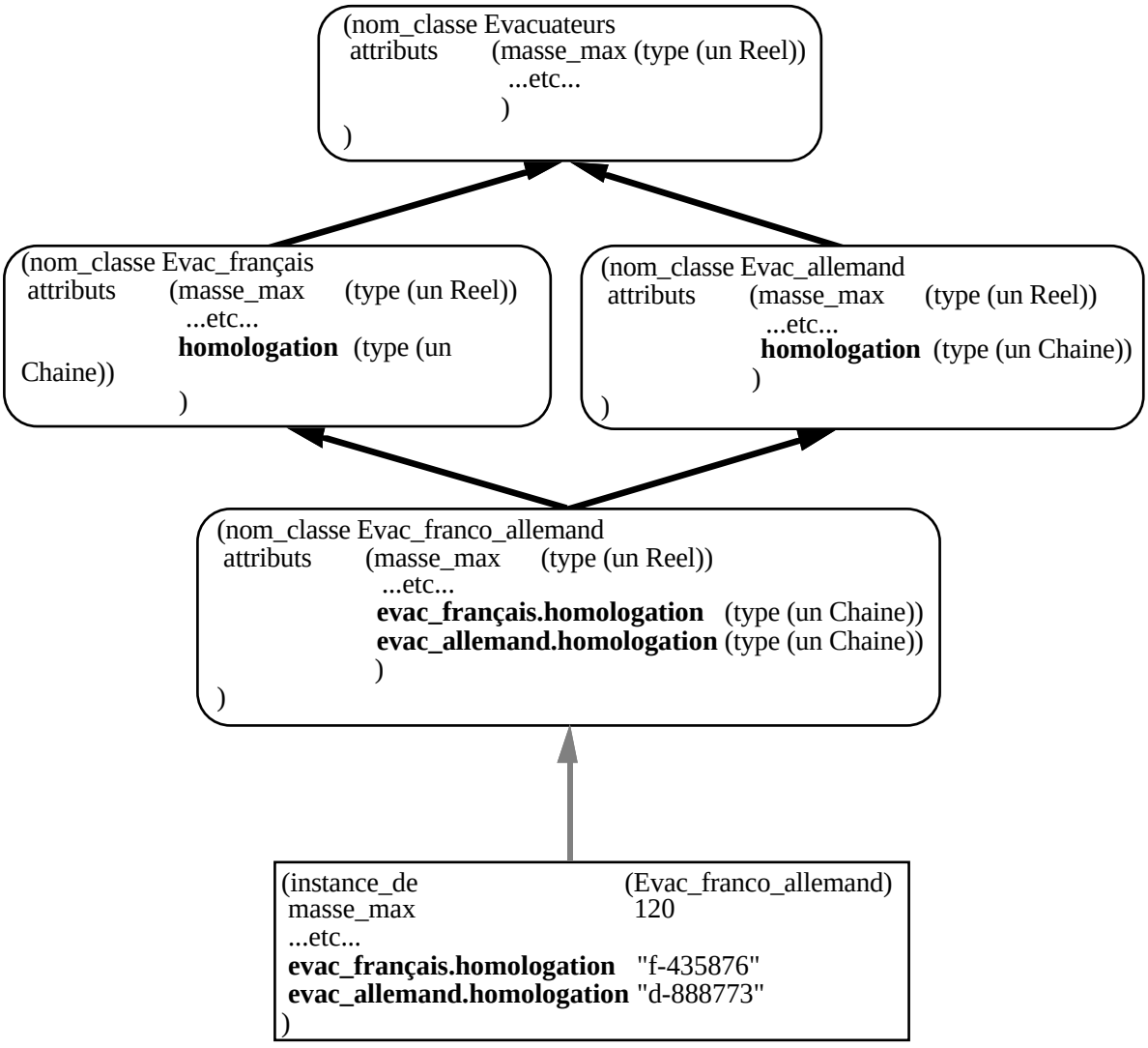


Figure 2.13. L'utilisation du nom complet des attributs.

Etant donné que le nom d'une classe est unique dans le graphe d'héritage et que le nom d'un attribut est unique dans une classe, le nom complet d'un attribut est unique dans le graphe d'héritage. S'il n'y a pas d'ambiguïté de noms, une référence à un attribut peut être faite par

son nom. Dans le cas contraire, le nom complet de l'attribut doit être obligatoirement fourni.

2.4.2. Les attributs pré-déclarés

Nous venons donc de voir que deux attributs ont la même sémantique si et seulement si ils ont le même nom complet. Dans l'exemple précédent, si l'on veut définir un attribut pour la date de la première homologation, identique pour les classes Evac_français et Evac_allemand il est donc nécessaire de définir un attribut *date_1ère_homo* dans leur super-classe commune à savoir Evacuateurs. Le nom de l'attribut est alors *evacuateurs.date_1ère_homo*. Or cet attribut n'est pas une caractéristique pertinente de tous les Evacuateurs. Par exemple, la classe Evac_rapides, sous-classe de Evacuateurs, en hérite, même si la valuation de cet attribut n'a aucun sens pour tous les évacuateurs rapides.

La solution proposée consiste à pré-déclarer l'attribut *evacuateurs.date_1ère_homo*. Pré-déclarer un attribut dans une classe C consiste à définir un nom d'attribut dans la classe C qui ne peut pas être valué par les instances de C. La pré-déclaration d'un attribut définit donc son nom complet: la classe de pré-déclaration est la classe d'origine de l'attribut. Cette notion de pré-déclaration permet de définir la sémantique d'un nom complet d'attribut dans tout le sous-graphe de la classe où il est pré-déclaré car il est hérité. Si un attribut pré-déclaré est ultérieurement défini dans une sous-classe, c-à-d si on lui donne un type, il devient instanciable.

Dans la figure 2.14, l'attribut *evacuateurs.date_1ère_homo* est pré-déclaré dans la classe Evacuateurs. Ceci permet de "fixer" sa sémantique. Il s'agit du même attribut qui est ensuite défini dans les classes Evac_français et Evac_allemand puis hérité dans Evac_franco_allemand. L'attribut *date_1ère_homo* n'est pas instanciable dans Evacuateurs mais il l'est dans Evac_allemand, Evac_français et Evac_franco_allemand. Il n'est pas instanciable non plus dans la classe Evac_rapides (non-montrée), sous-classe de Evacuateurs.

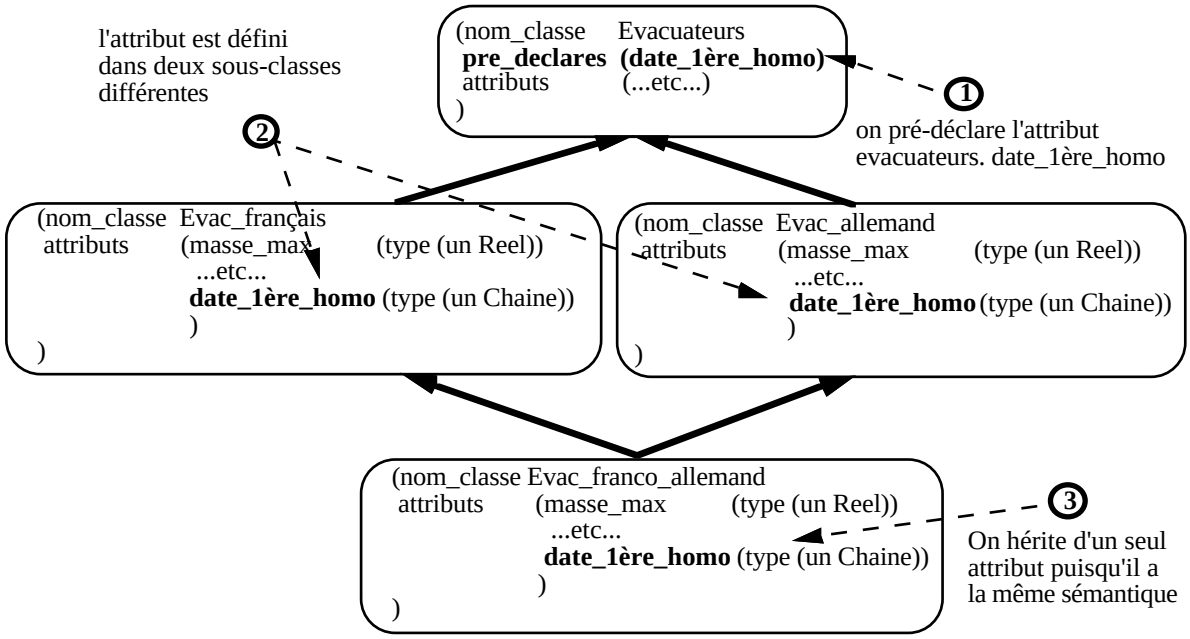


Figure 2.14. La pré-déclaration de l'attribut *date_1ère_homo*.

Les attributs pré-déclarés permettent aussi de maintenir la cohérence de la base. Par exemple, lorsqu'on élimine la définition d'un attribut de sa classe d'origine, celui-ci devient automatiquement un attribut pré-déclaré. Ceci permet de conserver la même sémantique pour cet attribut dans le sous-graphe dont la racine est la classe d'origine. Cette notion n'est pas sans rappeler celle de caractéristique différée ("deferred feature") de Eiffel [MEY89] et celle de méthodes virtuelles "pures" de C++ [STR89].

2.4.3. Le type des attributs

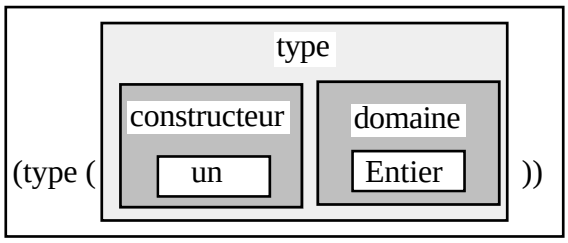
Le descripteur de type permet de spécifier les valeurs qu'un attribut peut prendre. Ces valeurs peuvent être de deux types:

• des scalaires; par exemple l'attribut *masse_max* de la classe Evacuateurs prend comme valeur un réel, ceci étant exprimé par le descripteur "(type (un Réel))".

• des instances; par exemple l'attribut *composants* de la classe Evacuateurs prend comme valeurs un ensemble de composants, ceci est exprimé par le descripteur "(type (ens Composant))".

Dans la suite nous utilisons le terme *type* pour le descripteur de même nom permettant de définir les valeurs possibles pour un attribut. Un descripteur de type est toujours composé de deux parties: un *constructeur* et un *domaine*. Le domaine est composé d'une partie obligatoire, le *domaine de base* et d'une partie optionnelle, les *restrictions de domaine*. La figure suivante

illustre un descripteur de type ayant un constructeur *un* et un domaine Entier.



Les restrictions permises sur le domaine de valeurs sont celles dont le système connaît la sémantique de façon à être capable de gérer les spécialisations. On exclut ainsi de la notion de type toutes les contraintes procédurales et toutes les inférences parce que leur résultat ne peut pas être connu du système. Deux attributs ayant le même type n'acceptent pas forcément les mêmes valeurs. Dans ce cas l'acceptation d'une valeur est conditionnée par les contraintes procédurales définies sur les attributs.

2.4.3.1. Les constructeurs

Les constructeurs permettent de définir si un attribut est mono-valué ou multi-valué. Le seul constructeur mono-valué est le constructeur *un*. Seul un cosntructeur multi-valué est implémanté au démarrage du système: le constructeur ensembliste (ens). L'ensemble n'admet pas de répétition des éléments et il ne conserve pas leur ordre.

2.4.3.2. La relation de sous-type

Nous définissons une relation de sous-type qui respecte notre sémantique de l'héritage, c'est à dire, l'inclusion des ensembles.

Axiome. Soit T_1 un type composé d'un constructeur con_1 et d'une classe C_1. Soit T_2 un type composé d'un constructeur con_2 et d'une classe C_2. T_1 est un sous-type de T_2 si et seulement si con_1 est égal a con_2 et si C_1 est une sous-classe de C_2. Cette relation est dénotée : $T_1 \leq T_2$

Cet axiome peut être écrit aussi de la forme suivante:

con1 C1 ≤ con2 C2 <==> (con1=con2) et ((C1 sous-classe de C2)

2.5. Plus sur les attributs

Dans Shood, les descriptions d'attributs sont modélisées par des classes, appelées classes-attributs. Comme dans Mering II [FER88], celles-ci sont instances d'une méta-classe particulière (META_ATTRIBUT) ou d'une ou plusieurs de ses sous-classes. Ces méta-classes définissent les caractéristiques propres aux attributs telles que le domaine, restriction de domaine, inférences et contraintes. Les descripteurs d'un attribut sont en effet des attributs de classe des classes-attributs. L'ensemble des descripteurs qu'un attribut peut valuer varie donc en fonction des méta-classes de la classe-attribut.

La modélisation des attributs par des classes permet de les enrichir plus facilement. En effet, la manipulation d'un attribut est équivalente à la manipulation de sa classe. Il est alors possible de lui ajouter des attributs, de changer sa classe d'instanciation ou sa super-classes. Cette section montre deux exemples d'extensions possibles d'attributs.

2.5.1. Les attributs des attributs

La modélisation des attributs par l'intermédiaire de classes permet la définition d'attributs pour les attributs. Supposons que nous voulions définir une classe de segments tolérants, sous-classe de la classe de Segments, ayant un coefficient de tolérance pour la dimension. Ceci veut dire que pour un segment donné, nous voulons exprimer la tolérance admise sur sa dimension. Une première solution consiste à définir un attribut *tolerance_dimension* dans la classe Segments_tolerants. Cette solution n'est pas satisfaisante car la tolérance que l'on veut exprimer est une caractéristique de la dimension et non pas des segments en général. C'est pourquoi le mot "dimension" fait partie du nom de l'attribut.

Pour que la tolérance soit une caractéristique de la dimension nous devons donc définir un attribut *coefficient_tolerance* pour l'attribut dimension. Pour ce faire, on définit Segments_tolerants, une sous-classe de la classe Segments (figure 2.15), dans laquelle on affine l'attribut dimension en lui rajoutant l'attribut *coefficient_tolerance*. Une instance de la classe de Segments_tolerants est également montrée. Son origine et son extrémité ont comme valeurs des instances de la classe Points. Sa dimension est de 30 et la tolérance sur la dimension est de 0,07.

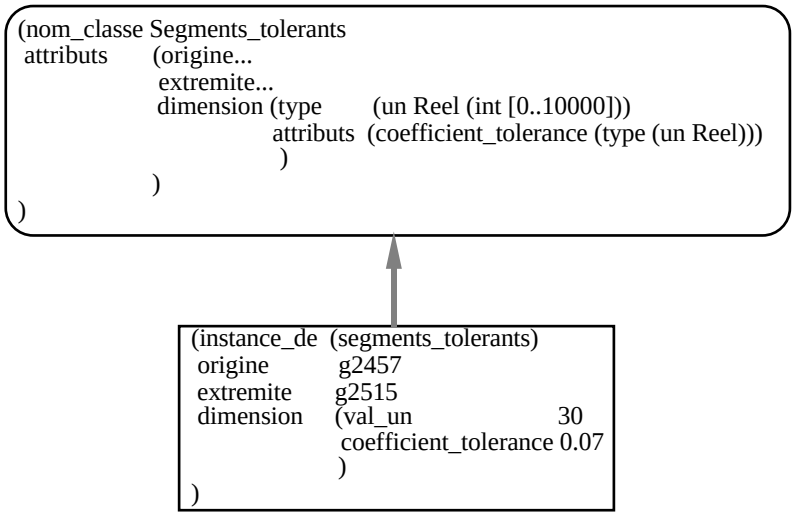


Figure 2.15. La classe Segments_tolerants et un de ses représentants. L'attribut *dimension* a un attribut d'attribut (*coefficient_tolerance*).

2.5.2. Les attributs obligatoires

La construction d'un objet est souvent une tâche longue qui demande une réalisation par étapes. Ainsi par exemple, la conception d'un avion est une tâche qui demande des années d'efforts au cours desquelles l'objet conçu reste la plupart du temps incomplet. Par exemple, lors de la conception d'un avion, l'attribut *ailes* ne sera pas instancié avant de connaître le poids du fuselage. Des applications de conception constituent pour Shood un domaine privilégié d'expérimentation. Elle sont caractérisées par une forte évolution des objets en cours d'élaboration. Ces objets ne sont complets qu'à l'issue de leur processus d'élaboration [NGU89]. La gestion des objets incomplets mais aussi incohérents a déjà été étudiée [RIE90] [FAV89] [FAV90]. Nous prenons en compte cette étude afin de gérer des attributs qui peuvent être éventuellement non-valués à un moment donné.

Il existe dans Shood des attributs obligatoires qui doivent toujours avoir une valeur et des attributs facultatifs qui peuvent ou non être valués lors de la création de l'objet. L'ensemble des attributs obligatoires d'une classe correspond en fait à la définition minimale des objets de cette classe. Les attributs instances de META_ATTRIBUT, sont des attributs facultatifs et META_ATTRIBUT est la classe par défaut des attributs.

META_ATTRIBUT admet une sous-classe META_ATT_OBLIGATOIRE modélisant les attributs obligatoires. Pour définir un attribut obligatoire, il suffit donc d'utiliser le descripteur de classe *instance_de* pour dire que la classe-attribut est instance de META_ATT_OBLIGATOIRE.

2.5.3. Récapitulatif

La modélisation des attributs par des classes permet d'augmenter facilement le pouvoir de représentation des attributs. Nous avons montré ici le résultat de deux extensions des attributs. La mise en œuvre de ces extensions est présentée dans le chapitre 4. Les avantages de ces extensions ainsi que les concepts Shood utilisés sont présentés dans le tableau de la figure 2.16

Extension sur les attributs	Avantage	Notion permettant la manipulation
Attribut des attributs	Permet d'attacher une propriété à la valeur d'un attribut (e.g la tolérance de la dimension)	Attribut des classes-attributs
Attribut obligatoire/facultatif	Permet de mieux modéliser l'incomplétude d'un produit en cours de conception: certains attributs restent longtemps sans valeur; d'autres doivent être valués pour que l'objet puisse exister.	Classe d'instanciation de la classe-attribut (META_ATTRIBUT et META_ATT_OBLIGATOIRE)

Figure 2.16. Ce tableau compile les avantages apportés par les extensions des attributs présentées dans cette section. Pour chaque extension, nous énonçons également la ou les notions de Shood permettant leur expression.

2.6. L'héritage, définition formelle

2.6.1. Le mécanisme d'héritage de Shood

Le définition du mécanisme d'héritage de Shood a été inspirée du travail de [BAN87]. Il est défini en utilisant 9 invariants et 3 principes [ESC89]. Un invariant est une condition que l'ensemble des schémas de classes doivent respecter à tout moment. Par contre, un principe représente seulement une indication lors d'une modification du graphe d'héritage. Les invariants sont classifiés par rapport à la partie du modèle qu'ils traitent.

2.6.2. Les invariants dans l'évolution des schémas

D'abord définissons un invariant qui nous assure la forme du graphe d'héritage en prenant en compte aussi les relations de classes disjointes.

2.6.2.1. Invariants du graphe d'héritage

[**GRAPHE]** Le graphe d'héritage est un graphe acyclique orienté connexe et à racine unique. Les classes sont représentées comme des nœuds. Chaque nœud a une étiquette unique représentant le nom de la classe. Les relations de <classe, sous-classe> sont représentées par des arcs unidirectionnels qui vont des classes vers leurs super-classes et les relations *disjointe-à* sont représentées par des arcs bi-directionnels entre les nœuds représentant les classes disjointes. La racine est le nœud représentant la classe Univers. Tous les nœuds sont accessibles à partir de la racine en suivant la relation de <classe, sous-classe>. Cette relation ne forme pas de cycles.

L'invariant du graphe prend en compte l'existence de la relation *disjointe-à* comme une relation coexistant avec la relation de <classe, sous-classe>. L'invariant suivant définit la sémantique d'une relation *disjointe-à*.

[SOUS-GRAPHES DISJOINTS] Deux sous-graphes à racine unique dont les nœuds-racines sont liés par la relation *disjointe-à* ne sont pas connectés par le lien de <classe, sous-classe>.

L'invariant suivant garantit l'unicité des noms des attributs définis ou pré-déclarés dans une classe.

2.6.2.2. Invariants des attributs

[NOM DES ATTRIBUTS] Tous les attributs définis ou pré-déclarés dans une classe ont des noms différents. Cet invariant garantit l'unicité des noms des attributs définis ou pré-déclarés dans une classe. Comme l'unicité des noms de classes est assurée par l'invariant du graphe, l'unicité du nom complet des attributs est assurée également.

L'invariant suivant met "en marche" le premier pas du mécanisme d'héritage. Cet invariant est chargé de garantir l'héritage des noms complets des attributs. Il garantit qu'une classe hérite des noms complets des attributs définis et pré-déclarés dans ses super-classes.

[HERITAGE COMPLET DES NOMS D'ATTRIBUTS] Une classe hérite de tous les noms complets des attributs définis dans ses super-classes. Une classe hérite de tous les noms complets des attributs pré-déclarés dans ses super-classes, sauf s'il est défini dans la classe (i.e s'il a un type).

L'invariant d'héritage complet garantit l'héritage des noms complets des attributs. Ceci résout

l'héritage des attributs pré-déclarés, mais ne résout pas le problème du type des attributs non pré-déclarés. L'invariant suivant maintient la relation définie précédemment sur les types des attributs non pré-déclarés.

2.6.2.3. Invariants des descripteurs

[RESTRICTION DES TYPES] Le type de tout attribut est celui défini dans son descripteur de type. Le type d'un attribut dans une classe doit être un sous-type des définitions de types données dans ses super-classes (si ces définitions existent).

On hérite ensuite des autres descripteurs.

[HERITAGE DES DESCRIPTEURS] Chacun des attributs d'une classe hérite de tous les descripteurs définis pour ce même attribut dans les super-classes de celle-ci. Cet invariant ne s'applique pas au descripteur de type.

Grâce à cet invariant on finit de résoudre les problèmes d'héritage des attributs. L'invariant suivant garantit l'héritage de toutes les clefs des super-classes d'une classe.

2.6.2.4. Invariants des clefs

Les clefs d'une classe sont aussi valides pour ses sous-classes. En effet, la clef (nom prénom) de la classe Personnes est aussi valide dans Les_dupont, une sous-classe de Personnes car elle détermine aussi de manière unique les objets de la classe Les_dupont. Par conséquent, dans Shood les clefs sont héritables. En effet, de la même manière qu'une classe hérite des attributs de ses super-classes, elle hérite aussi de ses clefs. La clef (nom prénom) est héritée dans Les_dupont. L'invariant suivant résout le problème de l'héritage des clefs.

[HERITAGE COMPLET DES CLEFS] Une classe hérite de toutes les clefs de ses super-classes.

Toute définition de clef dans Shood est supposée minimale. C'est à dire que lorsque l'on définit une clef, aucun des attributs de la clef n'est redondant. Si l'on supprime un attribut de la clef, celle-ci ne peut plus déterminer de manière unique les objets de sa classe. Par exemple, la clef (*nom prénom*) de la classe Personnes est une clef minimale. Aucun de ses attributs n'est redondant car les clefs (*nom*) et (*prénom*) ne déterminent pas de manière unique les objets de la classe Personnes.

Certaines définitions de nouvelles clefs non-minimales par rapport aux clefs héritées peuvent être détectées. En effet lorsqu'une clef est un sur-ensemble des attributs d'une clef héritée, la nouvelle clef n'est pas minimale dans cette classe. Par exemple, la clef (*nom prénom age*) de la classe Les_dupont n'est pas minimale. En effet, l'attribut *age* de la clef est redondant car la clef (*nom prénom*), héritée de Personnes, détermine de manière unique les objets de la classe Les_dupont.

L'invariant suivant se charge de garantir que les nouvelles définitions des clefs soient correctes. D'une part, les attributs qui constituent une clef dans une classe doivent avoir un type défini dans cette classe, car un attribut sans type n'est pas instancié (pas d'attributs pré-déclarés dans les clefs). Il doivent aussi avoir une valeur. Les attribut d'une clef doivent donc être obligatoires. D'autre part, les définitions des clefs non-minimales par rapport aux clefs héritées doivent être interdites.

[RESTRICTION DES CLEFS] Une clef dans une classe doit être composée d'attributs à caractère obligatoire et ayant un type défini. Toute nouvelle clef définie dans une classe doit être minimale par rapport aux clefs héritées.

2.6.2.5. Invariant de Méta-classe

Une classe doit hériter de toutes les fonctionnalités de ses super-classes. Ainsi si une classe possède une définition des clefs, ses sous-classes doivent en hériter. Pour accomplir cet objectif, toute classe doit instancier au minimum les mêmes méta-classes que ses super-classes, ceci afin d'hériter des structures nécessaires (par exemple les structures des clefs). Définissons l'invariant correspondant.

[META-CLASSE D'UNE CLASSE] Toute classe instancie, de manière directe ou indirecte, toutes les méta-classes de toutes ses super-classes.

2.6.3. Principes de l'héritage

Les principes sont des règles reposant sur la sémantique intuitive de l'héritage et relatives aux modifications que l'on peut apporter sur les schémas de classes. Ces principes permettent de choisir la meilleure politique pour conserver les invariants au moment d'une modification si plusieurs solutions sont envisageables.

Les modifications opérées sur des schémas peuvent ou non être propagées dans le graphe d'héritage. Par exemple, le changement de nom d'un attribut doit être obligatoirement propagé

pour conserver l'invariant d'héritage complet. Par contre, au moment d'éliminer la définition d'un attribut dans une classe, on peut soit uniquement l'éliminer de cette classe, soit l'éliminer de toutes les sous-classes. Le principe appliqué ici consiste à localiser le plus possible les modifications. On énonce le principe dit de propagation des modifications.

[PROPAGATION DES MODIFICATIONS] Toute modification faite dans une classe est propagée dans les sous-classes si cette propagation n'entraîne pas de perte d'information.

En appliquant ce principe, l'élimination d'un attribut d'une classe ne doit pas être propagée parce que sa propagation entraîne une perte d'information.

Au moment d'éliminer une classe, plusieurs formes valides peuvent se présenter pour maintenir les invariants, le principe suivant définit celles qui entraînent le moins de pertes d'information.

[ELIMINATION DES CLASSES] On ne peut pas éliminer une classe s'il existe au moins un attribut dont elle est la classe d'origine. Une solution possible est de changer la classe d'origine de l'attribut. On ne peut pas éliminer une classe définie par le système telle que Objet, Entier ou Chaîne. On ne peut pas éliminer une classe qui est utilisée dans le domaine d'un attribut. Lorsqu'une classe est éliminée, ses super-classes deviennent des super-classes de toutes ses sous-classes et ses sous-classes deviennent des sous-classes de toutes ses super-classes.

Ce principe peut être vu comme une extension du principe de propagation des modifications puisque l'élimination d'une classe n'entraîne pas de modification dans les schémas de ses sous-classes.

Lorsque l'on veut éliminer un lien de sous-classe et que ce lien est le seul qui permette l'héritage d'un attribut, on suppose que l'utilisateur est conscient que le fait d'enlever ce lien implique une perte d'information. Voici la définition du principe d'élimination d'un lien de super-classe, celui-ci prédominant sur le principe de propagation des modifications.

[ELIMINATION DES LIENS DE SUPER-CLASSE] Si le lien de <classe, sous-classe> éliminé est le seul lien par lequel un nom complet d'attribut (pré-déclaré ou défini) est hérité, la suppression de ce lien entraîne l'élimination de l'attribut en question. Ce principe prédomine sur le principe de propagation des modifications.

Une autre solution possible serait de renommer l'attribut hérité par ce lien. Cependant, le fait de renommer un attribut implique un changement de sa sémantique. Ceci peut conduire à une partition de l'attribut dans le sous-graphe dont la racine est la classe perdant le lien. Par

exemple, la classe C de la figure 2.17 (partie gauche) perd le lien de super-classe vers la classe A. Si l'attribut A.a est renommé dans la classe C par C.a (partie droite), le changement de sémantique implique une partition de l'attribut dans la classe D; elle hérite deux attributs, A.a et C.a, de ses super-classes.

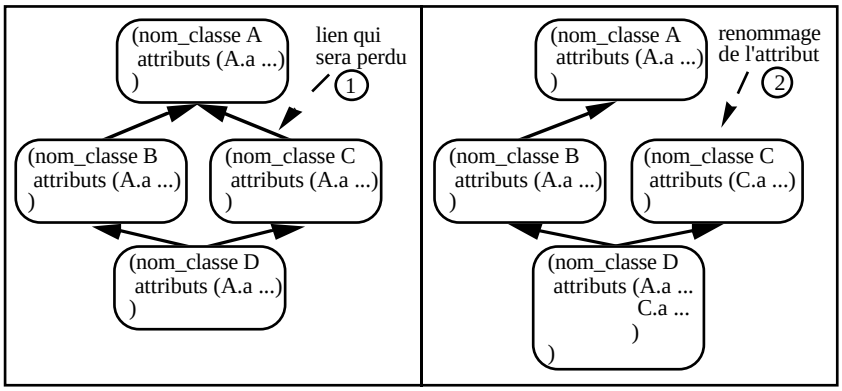


Figure 2.17. Le renommage de l'attribut A.a dans la classe C produit une partition de l'attribut en question dans la classe D.

2.7. L'instanciation, définition formelle

2.7.1. Invariant du graphe d'instanciation

[**GRAPHE]** Le graphe d'instanciation est un graphe cyclique orienté connexe, les nœuds de ce graphe représentant des objets. Tous les nœuds ont une étiquette unique: leur oid. Les liens d'instanciation sont représentés par des arcs uni-directionnels allant des objets vers leurs classes d'instanciation. Le seul cycle existant dans le graphe est le lien d'instanciation allant de META_A_CLEF vers elle-même. Seuls les liens d'instanciation d'un objet vers ses classes d'appartenance les plus spécifiques sont représentés.

[DISJONCTION] Deux nœuds représentant des classes disjointes ne sont pas référencés par un même objet.

2.7.2. Invariants des attributs

[NOMS DES ATTRIBUTS] Tous les attributs valués par un objet ont des noms distincts. Ceci garantit l'unicité des noms des attributs d'un objet.

[INSTANCIATION DES ATTRIBUTS] Tout objet value tous les attributs obligatoires

définis dans ses classes d'appartenance. Il peut ou non valuer les attributs non-obligatoires. Un objet ne peut valuer que les attributs définis dans ses classes d'appartenance.

[RESTRICTION DES TYPES] Toute valeur d'attribut d'un objet vérifie toutes les restrictions des types définies pour cet attribut dans les classes d'appartenance les plus spécifiques de l'objet. Les attributs facultatifs n'ayant pas de valeur ne sont pas contraints de vérifier cet invariant.

Certaines contraintes peuvent être temporairement violées. C'est le cas d'un objet en cours de conception. Des valeurs provisoires peuvent être calculées pour certains attributs qui ne vérifient pas toutes le contraintes. Souvent, des contraintes ne sont vérifiées qu'à la fin du processus de conception. Prenons l'exemple d'un objet représentant un avion en cours de conception. Les contraintes portant sur la résistance de l'avion à l'air peuvent rester violées longtemps au cours de sa conception car elles ne peuvent pas être vérifiées avant que la forme des ailes de l'avion soit complétée. Pour supporter de telles applications Shood gère des contraintes qui peuvent être violées.

La gestion des objets incohérents mais aussi des objets incomplets a fait l'objet d'une étude approfondie dans Shood [RIE90]. Nous prenons en compte ici les résultats de cette étude. Il existe donc deux types de contraintes: les contraintes faibles et les contraintes fortes. Les contraintes faibles sont des contraintes qui peuvent être violées. Les contraintes fortes sont des contraintes qui doivent être respectées dès qu'elles sont évaluables. Une contrainte est évaluable lorsque ses arguments en entrée sont connus. L'invariant suivant est chargé de gérer la vérification des contraintes faibles et fortes.

[**CONTRAINTES]** Toute valeur d'attribut d'un objet vérifie toutes les contraintes fortes définies pour cet attribut dans les classes d'appartenance les plus spécifiques de l'objet. Les contraintes faibles peuvent être violées. Les attributs facultatifs qui n'ont pas de valeur ne sont pas contraints de vérifier cet invariant.

Les inférences d'un attribut standard ont toutes la même spécificité. Elles sont toutes équivalentes car n'importe laquelle des inférences définies sur lui produit le même résultat et donc une valeur correcte. Ceci est le cas de l'attribut *age* défini pour la première fois dans la classe Personnes. Dans Personnes le seul moyen dont nous disposons pour calculer l'attribut est de demander une valeur à l'utilisateur: il s'agit de l'inférence User. Cependant dans les sous-classes de Personnes nous pouvons avoir des informations supplémentaires qui nous permettent de déduire automatiquement l'âge. Ceci est le cas de la classe Employes, sous-classe de Personnes. Dans Employes l'attribut supplémentaire *noss*, représentant le numéro de sécurité sociale, nous permet de calculer l'âge par l'intermédiaire de l'inférence

Calcule_age_par_noss. Dans cet exemple, toutes les inférences produisent une valeur correcte. Elles ne sont que des alternatives de calcul choisies en fonction de la connaissance détenue sur l'objet.

[INFERENCES] La valeur d'un attribut standard peut être calculée par une inférence de l'attribut définie dans n'importe quelle classe. En effet, l'inférence utilisée peut être définie même dans une classe qui n'est pas dans la liste des classes d'appartenance directes de l'objet.

2.8. Conclusions

La puissance d'un système de représentation réside dans sa capacité à représenter les problèmes pour lesquels il a été conçu. Dans ce chapitre, nous avons montré que les concepts déclaratifs de Shood permettent de satisfaire certains besoins des applications CAO tels que:

- La réutilisation des objets existants (instances)
- La réutilisation des structures (classes)
- La puissance de représentation

Au premier abord, nous avons présenté les concepts de base de Shood, à savoir:

- L'héritage
- L'héritage multiple
- L'instanciation
- L'instanciation multiple
- Le lien de disjonction

Nous avons montré que la fusion de toutes ces caractéristiques dans un même modèle n'est pas contradictoire.

Puis, nous avons montré un premier aperçu du niveau méta ainsi que les graphes de spécialisation et d'instanciation.

Ensuite, nous avons donné une solution aux conflits d'attributs dus à l'héritage multiple. Les conflits de noms sont résolus par la notion de nom-complet d'attribut et d'attribut pré-déclaré. Les conflits de valeurs dus aux descripteurs de type sont résolus par une relation de sous-type.

Puis, nous avons montré trois extensions du modèle et leur utilité pour la représentation de connaissances en CAO, à savoir:

- Les attributs d'attribut
- Les attributs obligatoires/facultatifs

Finalement, les définitions formelles des mécanismes d'héritage et d'instanciation ont été présentées. Ceux-ci sont définis par des invariants.

3. Shood: aspects procéduraux

3.1. Introduction

Ce chapitre présente les aspects procéduraux de Shood [MIL92] [RIE92b] [RIE92c]. Ces aspects répondent au besoin d'intégration des outils de calcul de CAO et augmentent la puissance de représentation de Shood [NGU92b] [NGU92c].

Pour réaliser ces concepts nous nous sommes fixés comme but de réutiliser un maximum de concepts existants. Ce but a été atteint grâce à la réflexivité. Les nouveaux concepts ont été définis en termes du système lui même. Ceci facilite l'application de mécanismes généraux de raisonnement (e.g instanciation, spécialisation, classification) aux nouveaux concepts.

Trois grands sujets sont abordés dans ce chapitre.

La première partie présente les méthodes, basées sur le concept de fonction générique. Les méthodes de Shood sont originales à plusieurs titres. D'une part, elles sont définies de manière réflexive: elles sont modélisées par des classes et leurs arguments par des attributs. Ceci permet de récupérer toute la puissance du descripteur de type pour la spécialisation des arguments des méthodes. D'autre part, l'implantation réflexive permet également que la sélection de méthodes soit faite à l'aide d'un mécanisme de classification qui ne donne pas de priorité à un argument particulier. Finalement, les mécanismes de spécialisation déclarative et procédurale gèrent la coopération de méthodes à l'aide de contrôles de séquentialisation.

Les idées sur les méthodes présentées ici étant issues de réflexions collégiales, il est difficile de départager l'apport personnel de chaque membre de l'équipe [RIE92b] [RIE92c]. Mon plus grand apport se situe sur le plan de l'implantation réflexive des méthodes et du contrôle de traitements dépendants (i.e contrôles de séquentialisation). J'ai également participé à l'encadrement d'un étudiant de DEA. Le résultat de son travail permettant l'implantation réflexive de contrôles et l'utilisation des arguments de méthodes pour la communication de résultats partiels n'est pas présenté ici. Pour plus de détails sur son travail, nous envoyons le lecteur à son rapport de DEA [MIL92].

Nous présentons ensuite les inférences. Elles permettent de définir la manière de calculer la valeur d'un attribut lors de la création d'un objet. Elles sont modélisées par des classes et utilisent des méthodes. Plusieurs inférences peuvent être définies pour un même attribut. Celles-ci sont organisées selon un ordre total.

Finalement, les contraintes procédurales sont présentées. Elle permettent de restreindre les valeurs qu'un attribut peut prendre par l'utilisation d'une méthode retournant un résultat booléen. Plusieurs contraintes peuvent être définies pour un attribut.

3.2. Les méthodes

Shood, comme Clos [DEM87] et CommonLoops [BOB85], est basé sur l'utilisation de fonctions génériques et non pas sur l'envoi de messages (cf état de l'art § Les aspects procéduraux). Une fonction générique est une fonction dont le comportement dépend du type des arguments: le choix du code à exécuter est déterminé par le type des arguments effectifs. Par exemple, la fonction générique Différence est réalisée par deux méthodes (figure 3.1). L'une d'elles a des arguments de type réel et l'autre de type entier. Cette fonction peut donc accepter comme arguments en entrée soit une paire de réels soit une paire d'entiers.

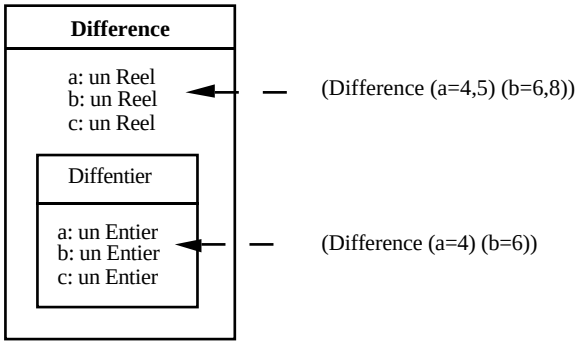


Figure 3.1. La fonction générique *Différence* est constituée de deux méthodes traitant les cas des entiers et des réels.

3.2.1. Les méthodes sont des classes

A partir de l'exemple précédent, nous pouvons déduire qu'une fonction générique est constituée d'un ensemble de procédures que nous appellerons méthodes. Celles-ci possèdent un ensemble d'arguments typés et une implantation (code). Nous pouvons remarquer dans la figure précédente que la forme externe d'une méthode ressemble beaucoup à celle des classes: les arguments typés d'une méthode rappellent les attributs typés d'une classe.

Dans Shood, les méthodes sont représentées par des classes, appelées *classes-méthodes*, dont

les attributs représentent les arguments de la méthode. Les classes-méthodes sont instances d'une méta-classe particulière META_METHODE qui spécialise META par ajout des attributs supplémentaires permettant de stocker le code à exécuter ainsi que des informations relatives à ce code. Le nom de la méthode est donné par le nom de la classe. Nous donnons une définition complète de META_METHODE dans le paragraphe 4.8. relatif à la modélisation des méthodes.

Dans les sections suivantes, nous verrons que l'utilisation de classes pour la modélisation de méthodes permet de récupérer des mécanismes généraux de Shood: la spécialisation pour la surcharge de méthodes, l'instanciation pour l'exécution de méthodes, la classification pour la sélection des méthodes et les types et les contraintes procédurales des attributs pour les spécialiseurs des arguments.

3.2.2. Spécialiseurs des arguments

Nous rappelons que Clos [DEM87], qui réalise également des fonctions génériques, n'admet que deux façons de spécialiser les arguments d'une méthode. L'une permet de restreindre le domaine de l'argument à une classe (e.g "Evacuateurs"); l'autre permet de tester l'égalité d'un argument avec un autre objet par exemple (eql standard-120).

Dans Shood, les classes-méthodes constituant une fonction générique sont organisées dans un graphe de spécialisation appelé *graphe de méthodes*. Le graphe des méthodes d'une fonction générique est organisé en fonction du type et du nombre des arguments. La méthode racine du graphe représente donc la méthode la plus générale et les méthodes feuilles représentent les méthodes les plus spécifiques.

La *surcharge* de méthodes est donc modélisée par la relation de sous-classe. Une classe-méthode M surcharge une classe-méthode M', ssi M est une sous-classe de M'. Cette surcharge consiste en effet en une spécialisation du type des arguments ainsi qu'en une redéfinition du code. La spécialisation est donc "réutilisée" pour modéliser un nouveau concept: la surcharge des méthodes.

Un autre exemple de réutilisation est celui des arguments formels des méthodes: ils sont modélisés par des attributs. Ceci permet de récupérer tout le système de types et de contraintes procédurales des attributs existant dans Shood pour spécialiser les arguments d'une méthode.

Les spécialiseurs des arguments de Shood sont donc plus puissants que ceux de Clos [KEE88] car un argument peut être spécialisé soit par la définition d'un nouveau type, soit par l'ajout d'une contrainte procédurale, soit par l'ajout d'un nouvel attribut.

3.2.3. Arguments obligatoires et optionnels

Les fonctions génériques ont un nombre variable d'arguments. En effet, l'attachement d'une nouvelle méthode à une fonction générique provient de la volonté d'exprimer un calcul nouveau. Cette nouveauté peut fort bien induire la nécessité d'un nouvel argument. Nous remarquerons que dans Clos [DEM87] les arguments optionnels existent mais ils ne sont pas pris en compte lors de la détermination de la fonction générique la plus spécifique.

Dans Shood, une fonction générique est désignée par un classe-méthode racine d'un graphe. Les attributs de cette classe racine sont les *arguments obligatoires* de la fonction générique. Les attributs propres aux sous-classes de la classe racine sont les *arguments optionnels*. Par exemple, une fonction générique A (figure 3.2) ayant un argument *a* peut avoir deux sous-méthodes B et C ayant chacune un attribut propre *b* et *c* respectivement. *a* est le seul argument obligatoire de la fonction générique A. Elle peut être appelée de trois manières différentes: avec *a* comme argument, avec *a* et *b* ou avec *a* et *c*.

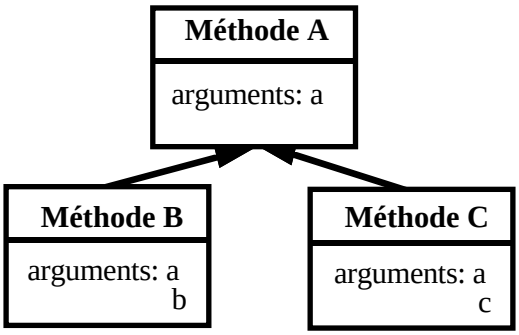


Figure 3.2. Les arguments b et c de la fonction générique A sont optionnels.

L'existence des arguments optionnels implique donc que les arguments d'une fonction générique ne peuvent pas être complètement ordonnés.

Dans Shood, les arguments formels et les arguments effectifs correspondants doivent être fournis à chaque appel. Ainsi, l'exécution de la fonction générique ayant comme racine la méthode *Différence* avec 4 comme valeur pour *a* et 6 pour *b*, est déclenchée par `(exec_meth Difference (a b) (4 6))`; où

- *exec_meth* est la commande demandant l'exécution de la fonction générique,
- *Différence* est le nom de la classe racine de la fonction générique appelée,
- *(a b)* est la liste des arguments formels de la fonction générique et

• *(4 6)* est la liste des arguments effectifs.

3.2.4. Sélection de méthodes

3.2.4.1. L'ordre des arguments

Un problème qui a été soulevé dans le premier chapitre de cette thèse est la sélection de la méthode la plus spécifique applicable. Nous rappelons que Clos [DEM87] réalise cette sélection en fonction du premier argument, et prend en compte les suivants seulement en cas d'ambiguïté. L'ordre des arguments est donc déterminant dans Clos: un changement d'ordre peut entraîner un changement dans la spécificité des méthodes. Par exemple, soit F une fonction générique comportant deux arguments en entrée et composée de deux méthodes (M1 a:integer b:float) et (M2 a:float b:integer). Si F est exécutée avec 5 comme valeur pour a et 8 comme valeur pour b la méthode M1 sera sélectionnée. L'inversion de l'ordre des arguments de F entraîne la sélection de M2 pour la même paire de valeurs. Nous rappelons que dans GNU C++ [TIE90], qui réalise aussi les fonction génériques, l'exécution de M1 ou M2 est déterminée par l'ordre de déclaration des méthodes. La première à avoir été déclarée est exécutée.

Dans Shood, le choix de la méthode la plus spécifique applicable est fait par classification. En effet, les méthodes étant organisées en graphe de spécialisation, la méthode la plus spécifique correspond à la classe-méthode la plus spécialisée. Le types et les contraintes ainsi que le nombre d'arguments (optionnels) sont toujours pris en compte lors de la sélection des méthodes.

A la différence de CLOS [DEM87] ou C++ [STR89], pour l'exemple précédent (F avec a=5 b=8), Shood retient (et fait coopérer) les deux méthodes M1 et M2.

3.2.4.2. Sélection par classification

L'utilisation de la classification pour la sélection de méthodes a l'avantage de réutiliser un mécanisme existant. Le mécanisme de classification utilisé est une variante du mécanisme de classification général de Shood appelé MIC [LIO93] [RIE91a].

Nous présentons ici un exemple de l'application de la classification à l'exécution d'une fonction générique. Celle-ci est réalisée en trois étapes: recherche de la méthode la plus spécifique vérifiant les paramètres, exécution de cette méthode et valuation des arguments en sortie. Par

exemple, pour l'appel de méthode `(exec_meth Difference (a b) (4 6))` MIC génère une instance dans la classe-méthode racine de la fonction générique invoquée (pour notre exemple une instance de *Différence*) en valuant les arguments en entrée de la méthode (voir figure 3.3). Cette instance est appelée *instance-exécution*.

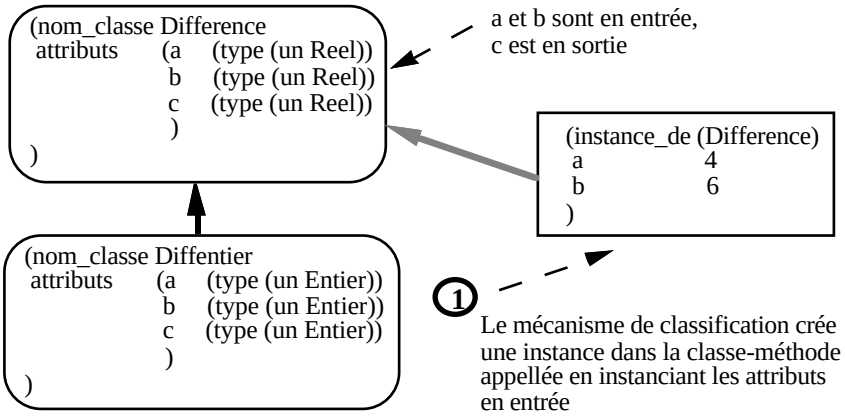


Figure 3.3. Première étape de la classification dans l'exécution d'une méthode

Ensuite, MIC classifie l'instance dans le graphe en fonction des types et des contraintes des arguments. C'est ainsi que l'instance correspondant à l'exécution `(exec_meth Difference (a b) (4 6))` sera automatiquement rattachée à la classe-méthode la plus spécialisée pour effectuer le calcul: Diffentier (figure 3.4).

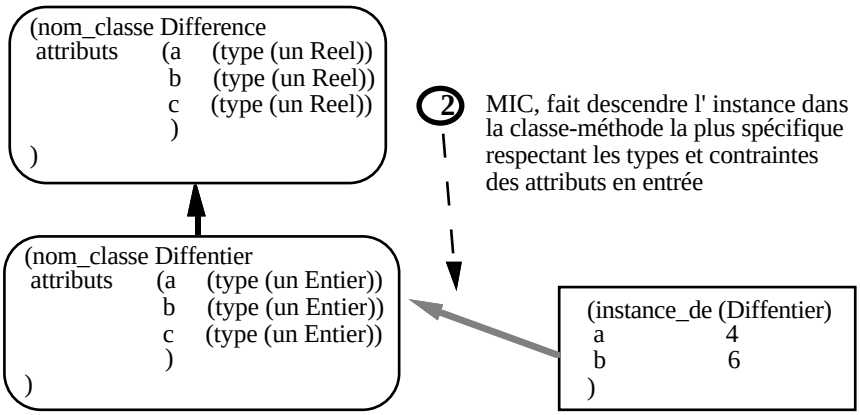


Figure 3.4. Deuxième étape: MIC trouve la classe-méthode la plus spécifique.

Ensuite, l'algorithme de la méthode la plus spécifique (i.e Diffentier) est exécuté. Après l'exécution de l'algorithme, les arguments en sortie de la méthode sont valués (voir figure 3.5).

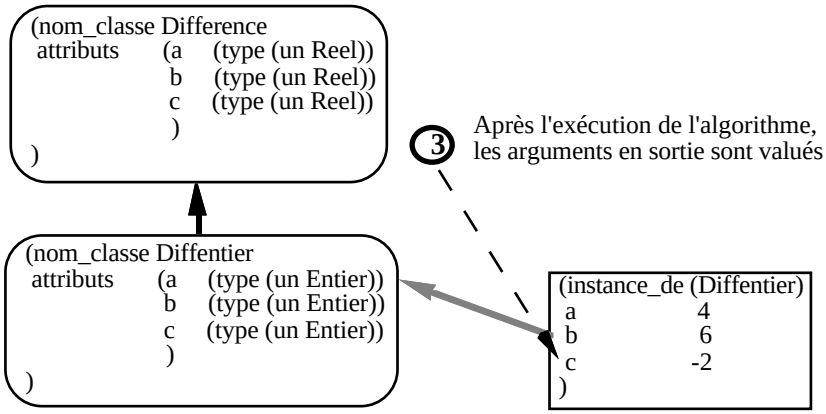


Figure 3.5 Troisième étape: l'instance-exécution après l'exécution de l'algorithme.

3.2.4.3. Exemple mécanique de multi-instanciation des méthodes

Dans certains cas, le mécanisme de classification peut délivrer plusieurs classes-méthodes les plus spécifiques. C'est le cas, par exemple, de la fonction F (cf. § L'ordre des arguments) et aussi de la fonction générique qui calcule la masse d'une barre. Cette dernière a été développée avec l'aide de mécaniciens du L3S [GSI93]. Dans cette application, nous pouvons modéliser 4 types de barres: des barres simples, des barres avec un trou, des barres avec un axe et des barres avec un axe et un trou. Le graphe 3.6 montre les trois classes nécessaires pour cette application avec 4 instances, une pour chaque type de barre que l'on peut représenter. Remarquons qu'une barre avec un axe et un trou est représentée comme une instance (*barre4*) à la fois de Barres_avec_axe et Barres_avec_trou.

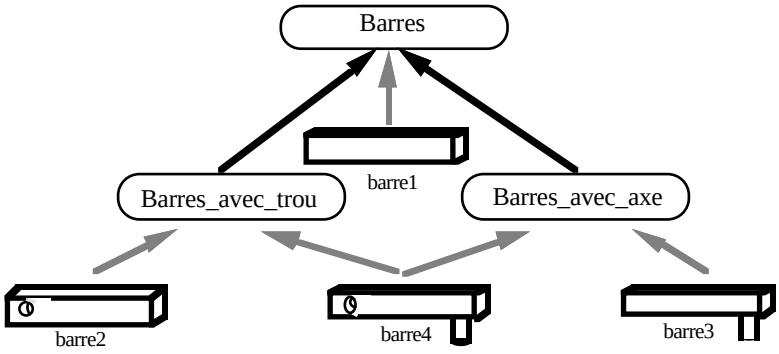


Figure 3.6. Trois classes sont nécessaires pour représenter les quatre types de barres.

La méthode qui nous permet de calculer la masse de n'importe quelle barre admet un attribut en entrée *barre* et un en sortie *masse*. Les différents types de barres, c'est à dire la spécialisation de l'argument en entrée *barre*, servent de fil conducteur pour la création du graphe de méthodes. Nous définissons donc trois méthodes, une pour chaque classe de barres. Ces trois méthodes (figure 3.7) définissent la fonction générique Calcule_masse_barre.

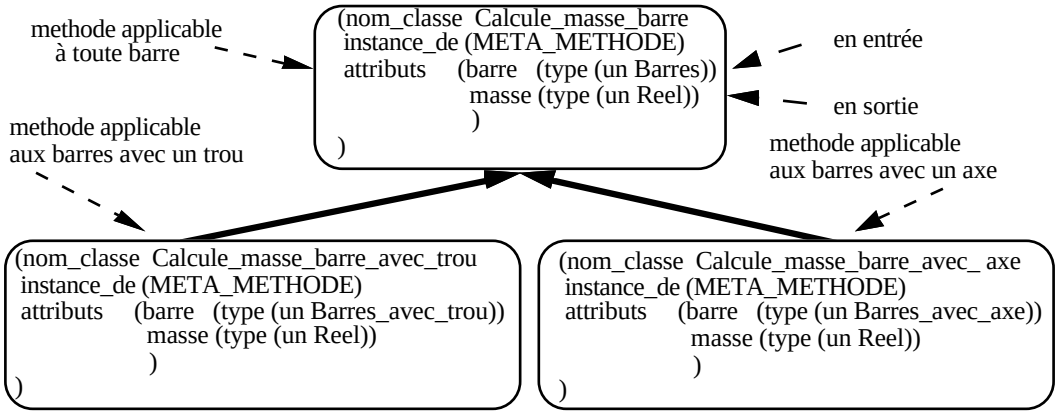


Figure 3.7. Un graphe de méthodes pour le calcul de la masse d'une barre quelconque.

Ainsi, pour l'appel de fonction générique (exec_meth Calcul_masse_barre (barre) (barre2)) le mécanisme de classification trouve Calcule_masse_barre_avec_trou comme étant la méthode la plus spécifique. Maintenant si l'on appelle la fonction générique avec *barre4*, l'instance représentant une barre avec un trou et un axe, le mécanisme de classification trouve les *deux méthodes* les plus spécifiques Calcule_masse_barre_avec_axe et Calcule_masse_barre_avec_trou. Quelles sont les actions possibles devant une telle situation? Trois possibilités sont envisageables, chacune excluant les autres:

• *Interdire*. Cela signifie que le cas où le mécanisme de classification délivre plus d'une méthode spécifique est interprété comme une erreur de modélisation et le système ne décide pas quelle méthode doit être exécutée. On ne peut donc pas calculer la masse d'une barre avec un axe et un trou. Ceci rappelle le fonctionnement du compilateur de C++ de Bell [XXX]. Pour l'exemple de la fonction générique F (cf. § L'ordre des arguments), si F est exécutée avec (a=5 b=8) une erreur "bad argument list" est déclenchée car il ne peut pas décider laquelle de deux méthodes (M1 ou M2) exécuter.

• *Choisir par un critère arbitraire*. Le système choisit, soit de manière aléatoire soit à l'aide d'une liste de précédence, une méthode parmi les méthodes applicables et il l'exécute. C'est le choix qui a été fait dans Clos. Dans Clos [DEM87] une liste de précédence est faite à partir du graphe des classes pour résoudre ce problème. Les classes sœurs sont ordonnées de gauche à droite ce qui permet d'établir un ordre totale entre elles. Les arguments des méthodes sont également ordonnés. Les premiers arguments sont plus importants. Par exemple nous pouvons dire que Barres_avec_trou est avant Barres_avec_axe. Dans ce cas l'exécution de la méthode avec *barre4* nous donne Calcule_masse_barre_avec_axe comme la méthode la plus spécifique applicable. L'inconvénient de cette approche est son côté arbitraire; en effet rien ne justifie qu'une méthode soit meilleure qu'une autre uniquement parce qu'elle a été définie avant.

• *Gérer*. Le système gère l'exécution de toutes les méthodes sélectionnées. Pour ce faire

il faut fournir un mécanisme qui permette une *coopération* des méthodes les plus spécifiques délivrées par le mécanisme de classification.

Nous avons choisi de faire coopérer les méthodes. Nous rappelons que, dans Shood, exécuter une méthode revient à créer une instance dans la classe modélisant la méthode. Envisager une coopération de méthodes implique la multi-instanciation des classes-méthodes impliquées.

Dans l'exemple précédent, pour pouvoir faire coopérer les méthodes Calcule_masse_barre_avec_axe et Calcule_masse_barre_avec_trou, afin de calculer la masse d'une barre ayant un axe et un trou, il faut créer une instance-exécution commune aux deux classes-méthodes.

Notre justification philosophique pour gérer la multi-instanciation des méthodes est donnée par la sémantique des classes et de la spécialisation. Dans Shood, les super-classes d'une classe ne sont pas ordonnées puisqu'elles représentent des ensembles (cf. § Les concepts de base). La multi-instanciation de méthodes nous permet donc d'être cohérents avec notre modèle: une classe n'est pas meilleure qu'une autre parce qu'elle a été définie avant.

D'un point de vue pratique, ce choix permet d'éviter la prolifération des sous-classes de méthodes due à la spécialisation multiple. Cet argument a été aussi invoqué pour justifier la multi-instanciation des classes (cf. Les points de vue). Dans l'exemple du calcul de la masse de la barre nous évitons la création d'une méthode Calcule_masse_barre_avec_axe_trou. Nous pouvons imaginer le nombre des classes-méthodes qui seraient nécessaires s'il existait d'autres types de barres (e.g des barres avec un axe triangulaire).

Dans la section suivante, nous verrons que dans Shood la coopération des méthodes est contrôlée de manière déclarative.

3.2.5. Spécialisation déclarative

Dans la spécialisation déclarative des méthodes, la combinaison des méthodes est sous la responsabilité des producteurs (cf. chapitre I. La réutilisation des méthodes). C'est donc au moment de la définition des méthodes et de manière externe (pas dans le code) que l'on exprime comment les méthodes vont se combiner. Cette section présente les problèmes rencontrés ainsi que les choix techniques qui ont été faits pour réaliser la spécialisation déclarative. D'abord les méthodes ont été découpées en traitements. Deux types de traitements sont ensuite identifiés: indépendants et dépendants. Les traitements dépendants ont motivé la définition de *contrôles de séquentialisation*. Ces contrôles permettent d'exprimer les dépendances entre les traitements.

Finalement, nous montrons comment les traitements sont hérités dynamiquement en l'absence d'une redéfinition (surcharge).

3.2.5.1. Les traitements

La granularité de code la moins fine que l'on puisse associer à une méthode est celle correspondant à une procédure indivisible. Malheureusement, une procédure indivisible limite les possibilités de réutilisation puisque cela oblige à réutiliser la totalité du code. Si nous affinons la granularité, nous pouvons donc imaginer la réutilisation de morceaux de code plus ciblés.

Heureusement, certaines méthodes ont un code qui peut être découpé en un nombre arbitraire de "morceaux" réalisant un pas ou un traitement bien spécifique d'une méthode. Nous appellerons ces morceaux des *traitements*.

Le découpage du code en traitements offre un maximum de souplesse pour la récupération du code au détriment, peut être, de la facilité d'utilisation. En effet, la réutilisation impose un adressage précis du traitement à réutiliser. Pour cette raison nous avons choisi, comme Clos [KEE88] et Loops [BOB83], de limiter le découpage du code à trois traitements et de leur donner un nom qui ait une signification intuitive.

Ce découpage du code en trois parties peut être considéré comme le prologue, développement et épilogue d'une procédure. Les trois parties sont appelées respectivement *pré* (-traitement), (traitement-) *principal* et *post* (-traitement).

Pour découper une méthode en traitements, des heuristiques sont utilisées:

- le traitement *principal* représente le code qui peut être commun à un certain nombre de méthodes du graphe de la fonction générique et

- les *pré* et *post* traitements représentent le comportement particulier apporté par chacune des méthodes. C'est souvent ce code qui différencie une méthode d'une autre.

La figure 3.8 montre la décomposition en traitements des méthodes pour le calcul de la masse d'une barre.

Méthodes	Type Traitement	Action du traitement
Calcule_masse_barre	principal	calculer la masse d'une barre simple
Calcule_masse_barre_avec_trou	principal	calculer la masse d'une barre simple
	post	retrancher la masse correspondant au volume du trou
Calcule_masse_avec_axe	principal	calculer la masse d'une barre simple
	post	ajouter la masse de l'axe

Figure 3.8. La décomposition en traitements du graphe de méthodes *Calcule_masse_barre*.

Les traitements sont modélisés comme des attributs de classe des classes-méthodes. Remarquons que dans le monde objet, si une méthode n'est pas définie pour une classe on recherche sa définition dans une super-classe. Ceci est une forme d'héritage puisque l'on hérite d'une méthode d'une super-classe. Les traitements sont donc des attributs héritables. Ainsi, si on définit le traitement *principal* "obtenir la masse d'une barre simple" dans la méthode Calcule_masse_barre, il n'est plus nécessaire de le définir dans les sous-classes: le traitement est hérité. L'héritage de traitements est dynamique; le traitement à exécuter est recherché dans les super-classes lors de l'exécution (section 3.2.5.6. Héritage dynamique des traitements). La figure 3.9 montre de manière schématique la définition des traitements des classes-méthodes pour le calcul de la masse d'une barre.

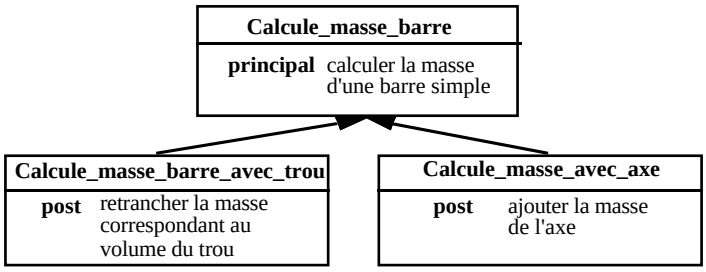


Figure 3.9. Le traitement *principal* de la méthode Calcule_masse_barre est hérité dynamiquement.

3.2.5.2. Les traitements indépendants

La granularité du code ayant été définie, nous allons aborder la coopération des traitements des méthodes. Nous devons donc trouver quels traitements exécuter et dans quel ordre.

La sémantique implicite dans les noms des traitements suggère trois étapes d'exécution: d'abord tous les traitements *pré* des méthodes concernées, puis l'un des traitements de type *principal* et finalement tous les traitements *post*. Seul un traitement *principal* est exécuté car le traitement *principal* est censé regrouper le code commun aux méthodes. Tous les traitements *pré* et *post*

doivent être exécutés car ils représentent le traitement particulier de chaque méthode. Il reste encore à choisir l'ordre le plus adéquat à l'intérieur des étapes *pré* et *post*.

Le code de certains traitements d'une même étape peut être indépendant du code des autres. Un exemple de ce cas est celui du calcul de la masse pour la barre *barre4* traité dans le paragraphe précédent. Regardons de près l'exécution des traitements dans cet exemple: il n'y a pas de traitements *pré*. Pour les traitements de type *principal*, les deux sont identiques car hérités ("calculer la masse d'une barre simple"); un seul est donc exécuté. Finalement, pour les traitements *post* nous pouvons d'abord retrancher le trou puis calculer la masse de l'axe ou l'inverse: leur ordre est indifférent.

Dans cet exemple, les post-traitements sont *indépendants*. Leur ordre d'exécution est indifférent. Ceci est dû à la commutativité des opérations arithmétiques. Assumer cette indépendance de traitements est le comportement par défaut dans Shood.

3.2.5.3. Les traitements dépendants

Dans certains cas les traitements ne sont pas indépendants. Prenons comme exemple, celui des méthodes commandant le façonnage de lentilles. La classe Lentilles (figure 3.10) possède deux sous-classes. Chacune de ses classes permet de définir une lentille ayant subi un traitement particulier. La classe Lentilles_teintées modélise les lentilles ayant subi un processus de pigmentation. La classe Lentilles_antireflet modélise les lentilles ayant subi un traitement antireflet.

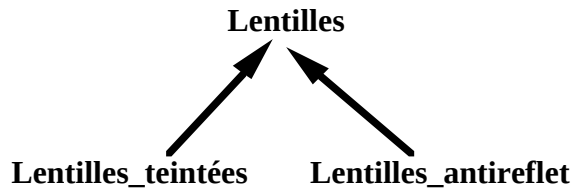


Figure 3.10. Les différentes classes de Lentilles.

Un graphe de méthodes capable de contrôler le façonnage de tout type de lentilles est présenté dans la figure 3.11. Chaque type de lentille nécessitant un traitement particulier, une méthode par classe a été définie. La racine du graphe des méthodes est Façonnage_lentille. Cette méthode nous permet de façonner une lentille modélisée par une instance de la classe Lentilles. Elle possède un traitement *principal* P1 donnant forme au verre.

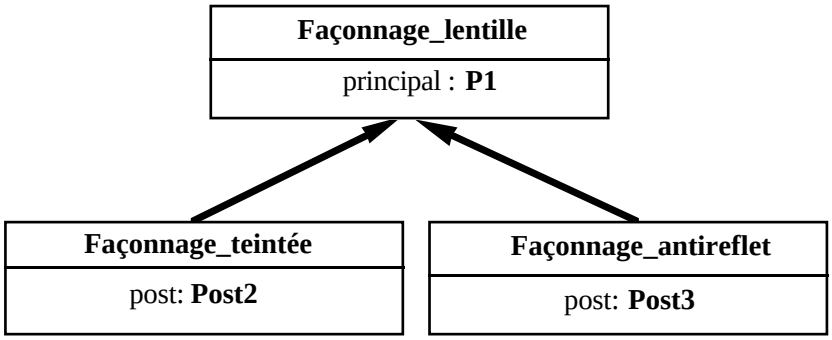


Figure 3.11. Le graphe de méthodes pour afficher une fenêtre

Chaque sous-classe de Façonnage_lentille possède un post-traitement réalisant le traitement supplémentaire correspondant au type de lentille. Dans Façonnage_teintée, Post2 donne une pigmentation à la lentille. Dans Façonnage_antireflet, Post3 applique un traitement antireflet à la lentille. On notera que ces deux méthodes hériteront dynamiquement du traitement *principal* P1 de Façonnage_lentille.

Une lentille peut avoir comme caractéristiques une combinaison de traitements de pigmentation et antireflet. Ceci implique que les méthodes sont appelées à coopérer pour produire une lentille ayant toutes ces caractéristiques: le traitement antireflet doit être réalisé après celui de pigmentation. Pour cela, nous devons pouvoir spécifier que le *post-traitement* de Façonnage_antireflet doit être exécuté après le *post-traitement* de Façonnage_teintée.

Dans cet exemple, nous pouvons remarquer que certains *post-traitements* sont *dépendants*. En effet, leur ordre d'exécution a son importance. Dans Shood, nous exprimons les dépendances entre traitements à l'aide de contrôles de séquentialisation.

3.2.5.4. Les contrôles de séquentialisation

Pour ordonner les traitements dépendants lors de la coopération de méthodes, Shood permet de préciser des *contrôles de séquentialisation*. Un contrôle de séquentialisation spécifie si le traitement doit être exécuté avant ou après un autre ou bien si l'exécution d'un traitement implique qu'il n'est plus nécessaire d'exécuter un autre.

Les 4 types de séquentialiseurs: *est_avant*, *est_après*, *renonce_si* et *inhibe*. sont présentés dans la figure 3.12. A tout traitement peut donc être associé un ensemble de contrôles de séquentialisation. Pour les *pré* et *post* traitements, ces contrôles peuvent être utilisés pour établir un ordre en éliminant des traitements (*renonce_si*, *inhibe*) ou pour les ordonner (*est_avant*, *est_après*). Par contre, pour les traitements de type *principal*, seul un contrôle de type *renonce_si* (ou *inhibe*) peut être utilisé. En effet, comme un seul traitement *principal* est

exécuté, les ordonner n'a aucun sens, nous pouvons uniquement en exclure.

type de contrôle	utilisation dans la méthode M1	Résultat	contrôle inverse
est_avant	est_avant M2	Le traitement de M1 est exécuté avant celui de M2	est_après
est_après	est_après M2	Le traitement de M1 est exécuté après celui de M2	est_avant
renonce_si	renonce_si M2	Le traitement de M1 n'est pas exécuté si celui de M2 est présent	inhibe
inhibe	inhibe M2	Le traitement de M2 n'est pas exécuté si celui de M1 est présent	renonce_si

Figure 3.12. Les contrôles de séquentialisation.

Après l'application des contrôles sur les traitements, les *pré* et *post* traitements sont ordonnés en un ordre partiel et un seul traitement *principal* est exécutable. Un autre choix aurait pu être d'accepter que plusieurs traitements de type *principal* soient exécutables après avoir analysé les contrôles. Ceci aurait voulu dire qu'aucun traitement n'est "meilleur" que les autres et qu'il fallait donc exécuter n'importe lequel. Nous avons choisi de signaler une erreur afin d'obliger l'utilisateur à spécifier exactement le traitement qu'il veut exécuter. Ainsi, après une erreur de ce type, il peut ajouter des contrôles de séquentialisation de façon à choisir le bon traitement *principal*, afin d'éviter que l'erreur ne se reproduise. De façon analogue, le fait qu'aucun traitement principal ne puisse être exécuté, est interprété comme une mauvaise modélisation de l'utilisateur: un message d'erreur est envoyé. Remarquons que l'exécution d'une fonction générique échoue si l'exécution d'un des traitements échoue.

L'approche par contrôles de séquentialisation utilisée lors de la sélection de méthodes joue le même rôle que la priorité donnée aux premiers arguments et l'ordre des super-classes de Clos [DEM87]. Cependant, elle est plus générale que celle de Clos (cf Chapitre 1 § Les aspects procéduraux) dans le sens où la réutilisation des différents types de traitements -*pré*, *principal* et *post*- est indépendante. En effet, l'ordre établi entre des éléments de *pré* traitements peut être différent de celui des *post*. D'autre part, les contrôles de séquentialisation ne sont utilisés qu'entre les traitements dépendants. Les traitements indépendants peuvent être exécutés dans un ordre quelconque.

3.2.5.5. Séquentialisation des traitements dépendants

Nous allons à présent montrer l'utilisation des contrôles de séquentialisation à l'aide de la méthode Façonnage_antireflet (figure 3.13). Une dépendance est exprimée dans son *post* traitement. Celui-ci est réalisé par un code d'adresse code_antireflet lequel doit être exécuté

après le *post* traitement de la méthode Façonnage_teintée.

```
(nom_methode Façonnage_antireflet
 super          (Façonnage_lentille)
 post           (algo       code_antireflet
                 est_après (Façonnage_teintée)
                 )
 ...
)
```

Figure 3.13. La définition du post-traitement de la méthode Façonnage_antireflet. Ce traitement doit être exécuté après le *post* traitement de la méthode Façonnage_teintée.

Afin de bien montrer le fonctionnement des contrôles de séquentialisation lors de l'exécution, nous allons utiliser l'appel de fonction générique suivant: (exec_meth Façonnage_lentille (len) (l1)). Où *l1* est une instance des classes Lentilles_antireflet et Lentilles_teintées.

L'exécution fait appel à la classification laquelle retourne l'ensemble des méthodes les plus spécialisées suivant: {Façonnage_antireflet, Façonnage_teintée}. Une instance-exécution multi-instanciant ces deux classes-méthodes est créée avec *l1* comme valeur pour *len*, l'argument en entrée.

Les trois étapes de l'exécution peuvent alors commencer.

L'étape *pré*, comme dans l'exemple de la barre, est sans conséquence: il n'y a aucun traitement *pré*.

L'étape *principal* est aussi très simple. Le traitement *principal* hérité par chacune des méthodes sélectionnées est celui défini dans leur super-méthode Façonnage_lentille. L'analyse des contrôles est alors inutile. C'est donc le traitement *principal* de la méthode Façonnage_lentille qui sera exécuté.

L'étape *post* collecte les *post* traitements des trois méthodes sélectionnées. Une analyse des contrôles nous fournit l’ordre total : <Post2 Post3>. En effet, la seule contrainte est "Post3 *est_après* Post2".

L'ordre d'exécution des traitements aboutissant aux façonnage de la lentille *l1* est le suivant: <P1, Post2, Post3>.

3.2.5.6. Héritage dynamique des traitements

Afin de traiter de manière similaire la multi-instanciation et la spécialisation multiple des classes-méthodes, l'héritage des attributs traitement est traité dynamiquement. Modifions l'exemple de la section précédente pour inclure la spécialisation multiple de classes-méthodes. Supposons à présent que l'on désire modéliser la méthode Façonnage_teintée_antireflet façonnant une lentille teintée et traitée antireflet (figure 3.14). La classification de l'instance-exécution créée pour afficher la lentille *l1* donne comme résultat à présent la classe-méthode Façonnage_teintée_antireflet. En l'absence de traitements plus spécifiques, aucun traitement n'ayant été redéfini dans Façonnage_teintée_antireflet, il semble naturel que l'ordre d'exécution reste le même que précédemment: <P1, Post2, Post3>.

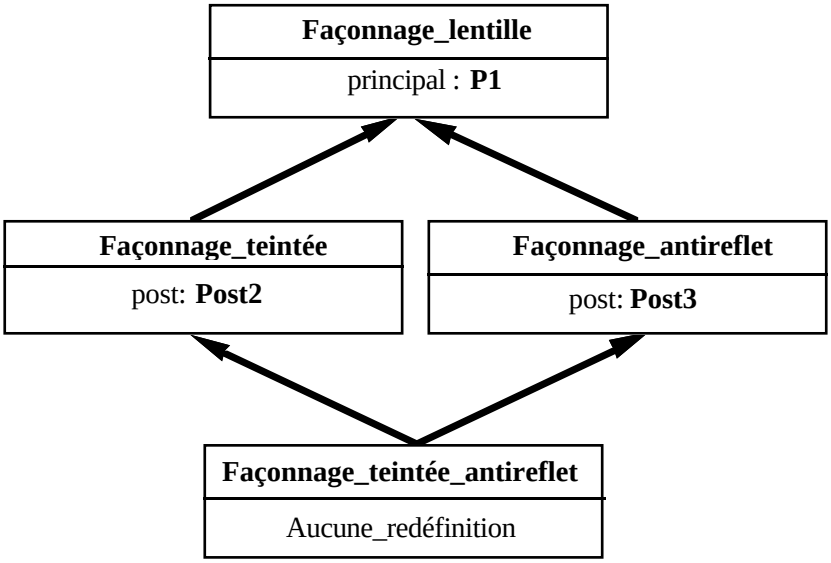


Figure 3.14. Aucun traitement n'est redéfini dans la méthode Façonnage_teintée_antireflet.

Pour ce faire, si les traitements n'ont pas été redéfinis, le mécanisme d'héritage recherche dans les super-classes les différents traitements et utilise les contrôles de séquentialisation pour les faire coopérer. Le processus est bien sûr itératif. La coopération des *pré* traitements, traitements *principaux* et*post* traitements des super-méthodes directes de Façonnage_teintée_antireflet donne le même résultat que celui de l'exemple précédent. L'ordre d'exécution est donc le même: <P1, Post2, Post3>.

3.2.5.7. Récapitulatif

Nous récapitulons ici les choix qui ont été faits pour réaliser la spécialisation déclarative des méthodes. D'abord, nous avons séparé le code d'une méthode en traitements pour affiner la granularité. Ceci nous permet de réutiliser des morceaux de code plus ciblés (figure 3.15, critère de granularité). Puis, nous avons identifié les traitements indépendants (leur exécution

est indépendante des autres traitements) et les traitements dépendants. Pour les traitements dépendants, nous avons défini des contrôles de séquentialisation qui permettent d'exprimer les dépendances entre eux (figure 3.15, critère de dépendance). Finalement, les traitements sont des attributs héritables. Leur héritage est dynamique: lors de l'exécution d'une méthode si un traitement n'a pas de valeur, on le recherche dans sa ou ses super-classes en le faisant éventuellement coopérer.

Critère de classification	Appelation du traitement	Caractéristique
Par sa Granularité	pré	prologue
	principal	développement
	post	épilogue
Par sa dépendance	indépendant	comportement par défaut d'un traitement
	dépendant	Caractéristique exprimée par les contrôles de séquentialisation

Figure 3.15. Tableau résumant les types de traitements lors de la spécialisation déclarative.

3.2.6. Spécialisation procédurale

Shood permet également d'utiliser une combinaison procédurale des méthodes. Cette réutilisation permet de demander directement, dans le code d'une méthode, l'exécution du code d'autres méthodes. La spécialisation procédurale est plus classique dans le sens où elle est toujours présente dans les langages à envoi de messages tels que Smalltalk. Elle s'oppose à la combinaison déclarative car cette dernière n'est pas spécifiée dans le code des méthodes mais dans les traitements contenant ce code. Les combinaisons déclarative et procédurale coexistent et se complètent. Ceci est le cas de l'appel procédural de super-méthodes (*super*) qui sera présenté plus bas. Cet appel correspond en fait à la coopération (déclarative) de super-méthodes.

Pour la combinaison procédurale, Shood offre une collection de primitives d'appel de méthodes. En particulier :

• *(exec_meth nom_methode l_param_formels l_param_effectifs)* est un appel de méthode avec classification.

C'est l'appel de méthode le plus général. Il correspond à un envoi de message classique ou un appel de fonction générique qui est dans Clos [KEE88] similaire à un appel de fonction Lisp. L'instance modélisant l'appel devient instance de la ou des méthodes les plus spécifiques

retenues.

• *(do_meth nom_methode l_param_formels l_param_effectifs)* qui permet l'exécution directe de la méthode nom_methode. Les arguments effectifs doivent satisfaire la définition des arguments formels mais le mécanisme de classification est court-circuité: l'instance modélisant l'appel est directement rattachée à la classe-méthode nom_methode. Cette primitive est similaire au *domethod* de Loops [BOB83].

• *(super)* correspond à un appel de super-méthodes.
Comme le “call-next-method” de Clos [DEM87], cet appel n'a de sens qu'entre méthodes d'une même fonction générique. Cependant, ce type d'appel s'apparente plus au “superfringe” de Loops dans le sens où plusieurs super-méthodes peuvent être sélectionnées puisqu'une classe-méthode peut être sous-classe de plusieurs autres classes-méthodes. Remarquons que les super-méthodes correspondent réellement ici aux super-classes de la classe-méthode propriétaire du traitement à partir duquel on fait l'appel. En effet un appel *super* dans le traitement *principal* de Façonnage_antireflet, hérité de Façonnage_lentille, correspond toujours à un appel aux super-classes de Façonnage_lentille car elle est propriétaire du traitement.

• *(super-traitements)* correspond à un appel de traitements de super-classes. Cet appel n'a de sens que s'il est effectué depuis un traitement. Il déclenche la coopération des super-traitements du même type que le traitement d'où l'appel est fait. Il se différencie de l'appel *(super)* car ce dernier fait coopérer les super-méthodes et non pas un seul type de traitement (*pré, post* ou *principal*).

Par exemple, supposons maintenant que la classe-méthode Façonnage_teintée_antireflet (figure 3.16) définit un *post* traitement propre. Un appel de type *(super-traitements)* dans le code du *post* traitement de cette méthode provoque la sélection de ses deux *post* traitements des super-classes: Façonnage_antireflet et Façonnage_teintée. Nous nous retrouvons alors de nouveau dans un cas de multi-instanciation de traitements: les deux traitements doivent coopérer.

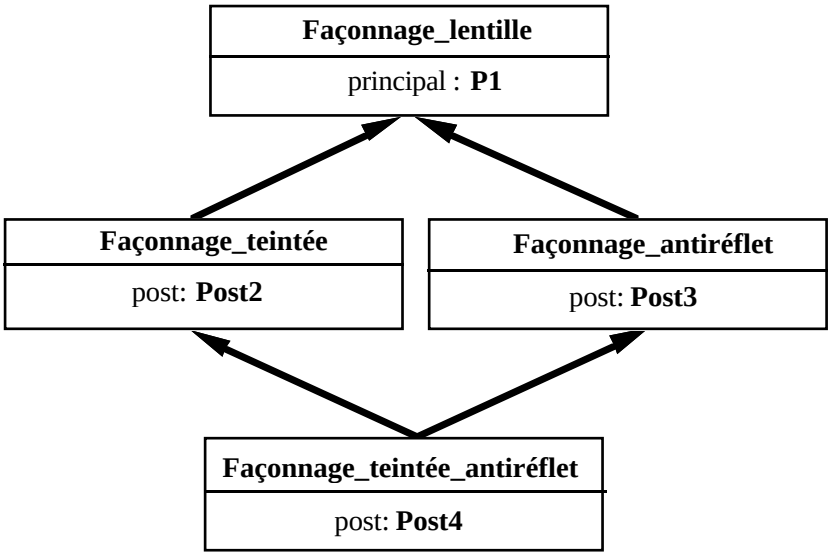


Figure 3.16. Le traitement Post4 fait un appel de super-traitements.

3.2.7. Récapitulatif

L'un de buts que nous nous sommes fixés avant de définir des fonctions génériques dans Shood est de réutiliser un maximum de concepts existants. Ce but a été atteint grâce à la réflexivité du modèle. Les notions concernant les fonction génériques ont été définies en termes du système lui même. Ce qui nous a permis la réutilisation de mécanismes généraux. Le tableaux de la figure 3.17 récapitule la réutilisation de concepts de Shood pour la modélisation de méthodes. La mise en œuvre de ces concepts est présentée dans le chapitre 4 de cette thèse.

Notion des méthodes	Modélisation dans Shood
Méthode	Classe-méthode
Fonction générique	Graphe des classes-méthodes
Surcharge	Spécialisation des classes-méthodes
Arguments	Attributs des classes-méthodes
Spécialiseur des arguments	Types et contraintes procédurales des attributs des classes-méthodes
Arguments optionnels d'une fonction générique	Attributs propres aux sous-classes de la classe racine du graphe de classes-méthodes
Type des arguments (entrée, sortie, communication)	Sous-classes de META_ATTRIBUT
Exécution d'une méthode	Instanciation (création d'une instance-exécution)
Sélection de méthodes	Classification (de l'instance exécution)
Coopération de méthodes	Multi-instanciation des méthodes concernées

Figure 3.18. Ce tableau récapitule les concepts de Shood qui ont été réutilisés pour la définition de méthodes.

3.3. Les inférences

Les inférences permettent de calculer les valeurs des attributs. Les *inférences intra-attribut* calculent un attribut à la fois et les *inter-attributs* calculent plus d'un attribut à la fois. Ce calcul peut être fonction des attributs de l'objet en cours de manipulation ou des attributs d'objets appartenant à d'autres classes.

Une inférence peut être exécutée à des moments différents. En effet, il peut être nécessaire de calculer la valeur d'un attribut d'un objet lors de sa création, modification ou accès en lecture (cf état de l'art § Les connaissances procédurales sur les attributs). Pour certains attributs, il est aussi important de maintenir des liens de dépendance: l'attribut doit évoluer si la valeur dont il est fonction évolue. La définition des inférences faite ici laisse la porte ouverte à des extensions car une étude approfondie des problèmes liés au calcul des valeurs des attributs fait actuellement l'objet d'une thèse [BOU94].

Comme nous l'avons dit dans l'état de l'art de cette thèse, nous ne nous intéressons ici qu'aux inférences intra-attribut déclenchées lors de la création. Ainsi, à un attribut peut être associés un

groupe d'inférences organisées dans un ordre total. Au moment de la création d'un objet, elles sont exécutées dans cet ordre jusqu'à ce que l'une d'entre elles réussisse.

3.3.1. Définition d'une inférence

Toute inférence est réalisée par un appel direct de méthode (do_meth) ou par un appel de fonction générique (exec_meth). Le résultat de cet appel doit être une instance-exécution ayant exactement un argument en sortie. Cet argument sera récupéré automatiquement pour valuer l'attribut sur lequel l'inférence porte. Pour cela, l'appel de méthodes dans une inférence a la même syntaxe que les appels exec_meth et do_meth présentés dans ce chapitre (cf. § La spécialisation procédurale). Par exemple, soit **Calculage** une méthode qui, à partir de l'année de création d'un objet, calcule l'âge que cet objet a atteint ou atteindra dans l'année courante. L'appel: `(exec_meth Calculage (an) (86))` définit donc une inférence qui pour l'année 1993, renvoie systématiquement 7 ans.

Dans Shood, les inférences sont représentées par des classes instances de la méta-classe META_INFERENCE ou d'une de ses sous-classes (cf. chapitre 4). Ceci laisse la porte ouverte à la modélisation des inférences ayant un comportement différent de celui par défaut. Un exemple de ceci est la création d'une méta-classe, sous-classe de META_INFERENCE, modélisant les inférences se déclenchant au moment de la lecture (et non pas à la création).

3.3.2. Les arguments d'une inférence

Toute inférence est réalisée par un appel de méthode. Les arguments effectifs d'une inférence utilisés pour valuer les arguments de la méthode la réalisant peuvent être de 3 types:

• le résultat d'une inférence. Shood permet l'imbrication des inférences. Dans ce cas, la valeur en sortie de l'appel de méthode réalisant l'inférence imbriquée est automatiquement récupérée pour valuer l'argument de l'inférence la contenant. Les inférences imbriquées sont exécutées au moment d'exécuter l'inférence les contenant.

• la valeur d'une variable. Shood admet la définition de variables qui sont évaluées au moment de l'exécution de l'inférence. La variable *self* désigne l'instance en cours de manipulation.

• Une constante. Shood reconnaît toute autre définition d'argument effectif comme étant une constante. Par exemple, *5* est interprété comme une constante.

3.3.3. Un exemple

Soient *age* et *annéenaiss* deux attributs de la classe Personnes. Nous associons à l'attribut *age* deux inférences (figure 3.18): Calculage et User. La première calcule l'âge de la personne en fonction de son année de naissance (les arguments ne sont pas montrés). Si cette inférence échoue, parce que la date de naissance n'est pas encore connue par exemple, alors l'inférence User s'exécute et l'utilisateur peut alors donner l'âge de la personne. Nous rappelons que les inférences sont totalement ordonnées.

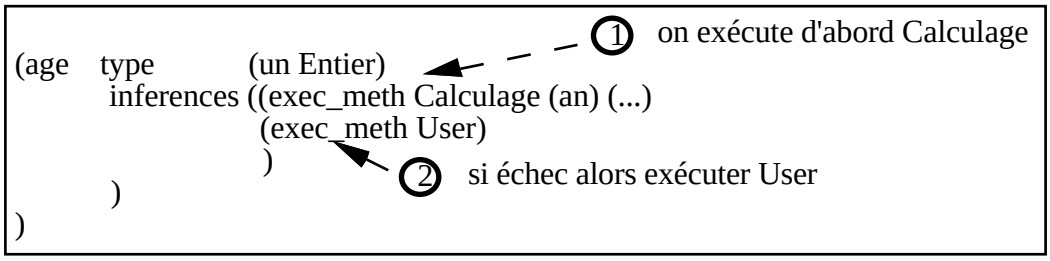


Figure 3.18. Deux inférences sont associées à l'attribut *age*.

3.4. Les contraintes

Les contraintes permettent de restreindre, de façon procédurale, les valeurs admises par les attributs. Les contraintes *intra-attribut* s'expriment sur un attribut; leur non vérification correspond à la mise en cause de l'attribut sur lequel elles portent. Elles restreignent donc un attribut à la fois. Les contraintes *inter-attributs* s'expriment au niveau d'une classe et concernent plusieurs attributs; leur non-vérification correspond à une mise en cause de tous les attributs sur lesquels elles portent. Elles restreignent donc plusieurs attributs à la fois.

Les contraintes sont vérifiées après toute modification de valeur d'un attribut. En effet, lors de la modification d'un attribut on doit procéder à une vérification de type pour ensuite vérifier les contraintes concernant l'attribut.

Une contrainte est réalisée par un appel de méthode (do_meth ou exec_meth). Une contrainte est vérifiée lorsque l'appel de méthode la réalisant retourne la valeur vrai. Plusieurs contraintes peuvent être associées à un attribut. Dans ce cas la relation qui les relie est un *et* logique implicite: toutes les contraintes doivent être vérifiées.

Plusieurs extensions sont possibles pour les contraintes. Une extension serait de pouvoir ordonner les inférences et les contraintes entre elles. Il faudrait pouvoir exprimer quelles contraintes doivent être évaluées pour quelles inférences. Cette extension serait accompagnée d'un langage d'inférences et de contraintes plus riche incluant des opérations telles que le *ou*

logique.

Dans ce travail, nous nous intéressons seulement aux contraintes intra-attribut. Elles sont ordonnées par un *et* logique. Leur ordre est total. Elle sont évaluées après chaque modification de valeur de l'attribut sur lequel elles portent.

A l'instar des inférences, les contraintes sont représentées par des classes appelées classes-contraintes (cf. chapitre 4). Celles-ci sont des instances de META_CONTRAINTE, la méta-classe primitive des contraintes. Pour cela, la syntaxe des contraintes et des inférences est quasi identique (cf. § Les inférences). Par exemple l'expression:

```
(exec_meth Plus_petit_que (a b) (5 6))
```

représente une contrainte réalisée par un appel exec_meth sur la méthode Plus_petit_que. Cette contrainte rend toujours vrai car ses arguments effectifs sont des constantes.

Les arguments effectifs acceptés par les contraintes sont identiques à ceux acceptés par une inférence: des constantes, des variables et des inférences. Nous renvoyons le lecteur aux inférences (cf. § Les arguments d'une inférence) pour la définition des différents arguments effectifs d'une contrainte.

3.4.1. Un exemple

Soient *père* et *age* des attributs de la classe Personnes avec la définition suivante:

```
(père (type          (un Personnes))
 age  (type          (un Entier)
       contraintes ((exec_meth Inférieur (a b) (...)))
       )
)
```

L'attribut *père* prend ses valeurs dans la classe Personnes. Aucune inférence n'ayant été définie par l'utilisateur, le système lui associe donc l'inférence par défaut: User. L'attribut *age* prend ses valeurs dans la classe Entier et il possède une contrainte exprimée par l'intermédiaire de la méthode *Inférieur* (les arguments ne sont pas montrés). Cette contrainte vérifie que la personne en cours de manipulation n'a pas un âge supérieur à celui de son père

Nous rappelons que la violation d'une contrainte intra-attribut (définie sur un attribut) ne remet en cause que la valeur de cet attribut même si d'autres attributs de la classe sont utilisés. C'est ainsi qu'une violation de la contrainte précédente ne provoque pas de remise en cause de l'attribut *age* du père de la personne en cours de manipulation.

3.5. Conclusion

Nous avons présenté les aspects procéduraux de Shood. Ces aspects répondent au besoin de l'intégration des outils de calcul de CAO et augmentent la puissance de représentation de Shood.

Les concepts procéduraux introduits (méthodes, inférences et contraintes) ont été définis en termes du système lui même. Le chapitre suivant montre leur mise en œuvre.

Les méthodes sont basées sur le concept de fonction générique. Elles sont modélisées par des classes. Les arguments sont modélisés par des attributs. La sélection repose sur un mécanisme de classification. Des mécanismes de spécialisation déclarative et procédurale sont fournis.

Les inférences définissent la manière de calculer la valeur d'un attribut lors de la création d'un objet. Elles sont modélisées par des classes. Plusieurs inférences, organisées dans un ordre total, peuvent être définies pour un même attribut.

Finalement, les contraintes permettent de restreindre les valeurs qu'un attribut peut prendre par l'utilisation d'une méthode retournant un résultat booléen. Plusieurs contraintes peuvent être définies pour un attribut.

4. L'extensibilité

4.1. Introduction

Ce chapitre est dédié à l'étude du méta-niveau de Shood. Shood est en effet un système méta-circulaire, il se définit lui-même. Les différents concepts du modèle sont modélisés par des méta-classes. Par exemple, META_ATTRIBUT est la méta-classe des attributs, META_METHODE celle des méthodes, META_INFERENCE celle des inférences et META_CONTRAINTE celles des contraintes.

Les méta-classes permettent de définir une structure commune au type de classes qu'elles modélisent. Par exemple, nous verrons que pour les classes à clef, cette structure commune est l'attribut *clef* (défini par META_A_CLEF) .

Mais la définition des méta-classes n'est pas suffisante pour définir un comportement. En effet, au moment de la manipulation d'une classe à clef, il faut que les caractéristiques des clefs soient respectées. Lors de la création d'une instance d'une classe à clef, il faut vérifier l'unicité des attributs de la clef et il faut faire une entrée dans la table de clefs de la classe.

Shood est un modèle *extensible*, il doit donc fournir un moyen ouvert pour représenter ce comportement. Le comportement étant une notion intrinsèquement procédurale, il est défini par des *méthodes*.

Il existe dans Shood un graphe de méthodes correspondant à chacune des manipulations système que nous pouvons effectuer sur les objets. En effet, les opérations système telles que la création d'objets et la modification sont modélisées par des graphes de méthodes. Grâce à cette modélisation, l'extensibilité du système est rendue plus facile. Ainsi, pour ajouter un nouveau concept nous ajoutons des méta-classes modélisant la structure du concept et les méthodes modélisant son comportement. Nous verrons par exemple que l'ajout des classes à clefs correspond à l'ajout d'une méta-classe pour les classes à clef plus l'ajout, dans le graphe de création d'objets, d'une méthode applicable aux classes à clef faisant les vérifications et mises à jour nécessaires.

Nous commençons ce chapitre par un aperçu global de la modélisation du comportement des fonctions système dans Shood. Celui-ci est illustré par la fonction générique *Creer_objet*, la méthode système de création d'objets.

Puis, nous montrons la modélisation des concepts système présentés dans les chapitres 2 et 3: les classes standards, les classes à clef, les attributs, les attributs des attributs, les attributs obligatoires, les méthodes, les inférences et les contraintes. Si certains de ces concepts existent lors du démarrage du système, d'autres sont en fait des extensions. Pour chacun des concepts nous présentons les méta-classes ainsi que la méthode correspondant à l'opération de création.

Nous présentons ensuite l'amorce-strap, définition minimale du modèle qui permet d'amorcer sa création. Les méta-classes et classes de l'amorce sont présentées et justifiées.

Finalement, nous résumons les méta-classes et les méthodes du graphe de Creer_objet présentées dans ce chapitre.

4.2. Premier aperçu de Creer_objet

Nous avons déjà énoncé que, dans Shood, les concepts du système sont représentés par des objets. Certains de ces concepts sont modélisés par des classes. Par exemple les attributs, les méthodes, les inférences et les contraintes. Cette modélisation offre des avantages qui sont expliqués dans ce chapitre. Avec l'introduction d'une méthode Shood pour la création d'objets, un avantage additionnel devient apparent: la standardisation. En effet, valuer un attribut, exécuter une méthode ou une inférence ou vérifier une contrainte revient à créer une instance dans la classe modélisant l'un de ces concepts. Cette création n'a pas toujours le même sens: pour un attribut elle équivaut à une valuation, pour une méthode à une exécution. La création d'objets de Shood peut être modélisée par une fonction générique.

Creer_objet est la racine du graphe de méthodes système permettant la création d'objets. Il existe une méthode dans le graphe pour chaque type d'objet. Les méthodes Creer_objet prennent en entrée deux attributs :

- classe; la classe dont l'objet sera instance, ainsi que

- l_att_val; une liste de couples (attribut-valeur) qui seront utilisés lors de la valuation de l'objet.

Le type de l'attribut *classe* sert de fil conducteur pour la définition du graphe de méthodes de Creer_objet. Les classes étant toutes des instances des méta-classes nous nous intéressons au graphe des méta-classes. META, la méta-classe primitive, est la racine du graphe ayant comme sous-classes les méta-classes modélisant les autres concepts du système. A chacune de ces

méta-classes, c'est-à-dire à chacun de ces concepts, correspond une méthode du graphe Creer_objet. La figure 4.1 montre les deux graphes en parallèle.

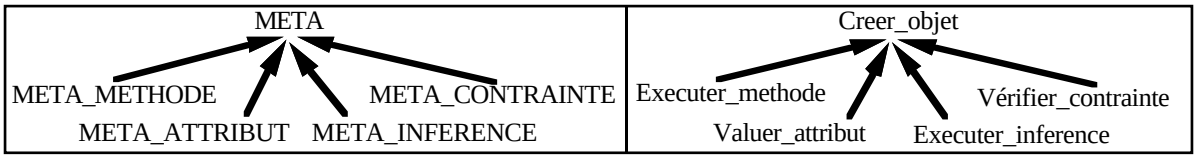


Figure 4.1. Le graphe de méta-classes et le graphe de Creer_objet.

Le but de ce premier aperçu de Creer_objet est de donner une approche intuitive du graphe. Nous présenterons tout au long du chapitre les méta-classes et leurs méthodes associées modélisant le comportement . Nous verrons par exemple que l'introduction des classes à clef correspond à:

• la création d'une méta-classe META_A_CLEF modélisant la structure des classes à clef,

• la création d'une méthode Creer_objet_a_clef dans le graphe Creer_objet modélisant le comportement des classes à clef.

Nous verrons dans ce chapitre que le même raisonnement peut être appliqué aux arguments des méthodes, des inférences et des contraintes.

4.3. META: la définition minimale d'une classe

Revenons sur la méta-classe META, présentée brièvement dans le chapitre 2 (cf. § Vers l'extensibilité). META étant la plus générale des méta-classes, elle définit la structure commune à toutes les classes. Les attributs de cette classe sont donc les attributs minimaux qu'une classe peut instancier. La figure 4.2 montre la définition de META.

```
(nom_classe   META
 instance_de  (META)
 super        (Objet)
 attributs    (nom_classe   (un  Chaine)
               instance_de  (ens META)
               super        (ens META)
               sous         (ens META)
               attributs    (ens META_ATTRIBUT)
               instances    (ens Objet)
               )
)
```

Figure 4.2. Définition complète de META, la méta-classe primitive.

Nous donnons une définition intuitive des attributs de META.

nom_classe

Les classes sont identifiées par une clef constituée du nom de la classe: l'attribut *nom_classe*. Les classes, comme tous les objets, sont aussi identifiées par des identificateurs système.

instance_de/*instances*

L'attribut *instance_de* représente les classes d'appartenance de tout objet. La valuation de cet attribut pour une instance (par exemple *dupont*) permet donc de la lier à sa classe d'appartenance (Personnes) et c'est grâce à ce lien qu'une instance peut valuer les attributs (*nom*, *prenom*) définis par sa classe. Afin de permettre la multi-instanciation, *instance_de* est multi-valué. L'attribut *instance_de* est donc de type ensembliste car les classes d'appartenance d'une instance ne sont pas ordonnées.
Toutes les classes existantes sont des instances directes ou indirectes de META car elle est la méta-classe primitive. Le domaine de valeurs de l'attribut *instance_de* est donc META. Objet est la classe des objets instanciables (par opposition aux objets virtuels, non instanciables). L'attribut *instance_de* est donc défini dans la classe Objet afin de le faire hériter dans toutes les classes instanciables.
Toutes les classes valuent aussi un attribut *instances* permettant de connaître les instances de la classe. Cet attribut est l'inverse de l'attribut *instance_de*. Ainsi l'attribut *instances* de la classe Personnes contient l'oid *dupont* et l'attribut *instance_de* de l'objet d'oid *dupont* contient l'oid de la classe Personnes. Les attributs *instances* et *instance_de* sont automatiquement mis à jour par le système.

super/*sous*

Toutes les classes valuent un attribut *super* permettant de spécifier les super-classes d'une classe à l’exception de la classe Univers qui est la racine du graphe de spécialisation). Le modèle Shood permet la spécialisation multiple c'est à dire qu'une classe peut avoir plusieurs super-classes. L'attribut *super* est donc multi-valué et prend ses valeurs dans META. Il est de type ensembliste car les super-classes d'une même classe ne sont pas ordonnées.
L'attribut inverse de super est *sous*. Il permet de connaître les sous-classes d'une classe. Ainsi l'attribut *sous* de la classe Personnes contient l'oid de la classe Enseignants et l'attribut *super* de la classe Enseignants contient l'oid de la classe Personnes. Les attributs *super* et *sous* sont automatiquement mis à jour par le système.

attributs

Finalement toutes les classes valuent un attribut *attributs* permettant de définir l'ensemble des attributs de cette classe. Cet attribut est donc multi-valué et de type ensembliste car les attributs d'une classe ne sont pas ordonnés. Son type est META_ATTRIBUT. Dans Shood, les attributs sont modélisés par des classes appelées classes-attributs. META_ATTRIBUT est la méta-classe par défaut des classes-attributs.

4.3.1. Comportement

Creer_objet, la racine du graphe, modélise le comportement des classes standard. L'attribut *classe* de cette méthode à comme type (un META).

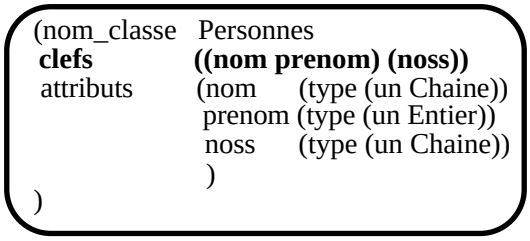
Creer_objet est constituée d'un traitement *principal*. Elle n'a pas de pré ou post traitement. Dans le traitement principal, la création de l'objet se fait en deux étapes. D'abord un nouvel oid est alloué à l'objet à créer. Puis, la méthode Creer_objet est récursivement appelée pour chacun des attributs de la classe à instancier.

4.4. Une extension: Les classes à clef

La manipulation des objets par des clefs définies par l'utilisateur est un concept ajouté dans Shood. Une discussion sur les avantages de l'introduction de cette notion est présentée dans [ESC89].

4.4.1. Définition des clefs

Dans Shood, toute classe peut être munie d'une ou de plusieurs clefs. Une clef est formée d'un ensemble d'attributs qui déterminent totalement l'identité d'un objet, c'est à dire qu'il n'y a pas deux objets d'une même classe ayant la même valeur pour une clef. Un exemple de définition de clefs pour la classe des Personnes est montré dans la figure suivante. En effet, le numéro de sécurité sociale (l'attribut *noss*) ainsi que le nom et le prénom (*nom prenom*) sont des clefs pour la classes des Personnes.



4.4.2. Représentation

Nous présentons ici l'extension du modèle qui permet la définition des classes à clef.

Le comportement des classes à clef est détenu par la méta-classe META_A_CLEF (figure 4.3) instance de META et sous-classe de META. En tant qu'instance de META elle value les propriétés minimales des classes (figure 4.3). En tant que sous-classe de META elle hérite de ses propriétés minimales et est donc génératrice de classes (par exemple de Personnes). META_A_CLEF admet un attribut supplémentaire *clefs*. La valuation de cet attribut permet de définir la ou les clefs des classes à clef.

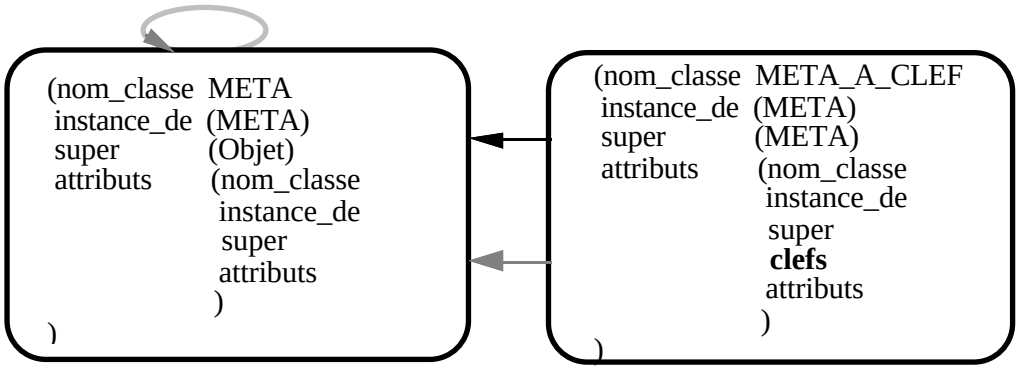


Figure 4.3. Premier pas dans la définition des classes à clef

Seule META, instance d'elle-même, fait partie de l'amorce du système Shood (cf chapitre 4 § L'amorce). L'ajout de META_A_CLEF est fait postérieurement en utilisant les primitives du système.

Cependant dans Shood, toute classe (ou méta-classe) est identifiée par son nom qui est donc unique dans la base. L'attribut *nom_classe* défini dans méta est donc un attribut clef. Pour pouvoir définir *nom_classe* comme étant un attribut clef il est nécessaire que META soit elle-même une instance de META_A_CLEF (figure 4.4). Nous avons dit précédemment que META est instance d'elle-même. Par conséquent META est instance de META et de META_A_CLEF.

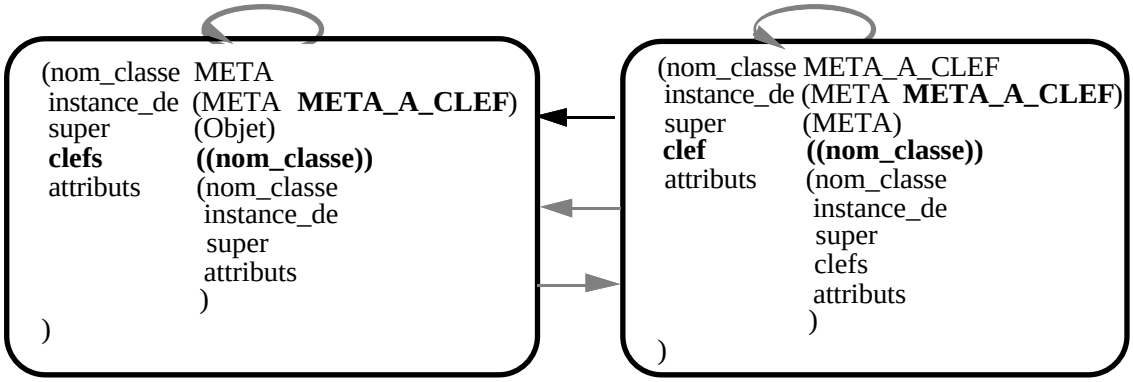


Figure 4.4. Définition dans META de la clef formée par l'attribut *nom_classe*.

De plus, si l'on veut que META_A_CLEF, sous-classe de META, hérite de toutes les fonctionnalités définies dans META (telle que l'attribut *clefs*) il faut que META_A_CLEF instancie au minimum le même ensemble de classes que META, sa super-classe (cf. l'invariant d'héritage de méta-classes au chapitre 2). META_A_CLEF doit donc être instance de META et d'elle-même.

Finalement, nous pouvons observer dans la figure précédente que la définition des liens d'instanciation est redondante. META instancie les attributs de META, mais aussi ceux de META_A_CLEF, pourtant META_A_CLEF est une sous-classe de META, c'est-à-dire, une classe plus spécifique. Le même raisonnement peut être fait sur le lien d'instanciation de META_A_CLEF vers META. Pour éliminer la redondance, nous éliminons META, la classe la moins spécifique, des liens d'instanciation de META et de META_A_CLEF (figure 4.5).

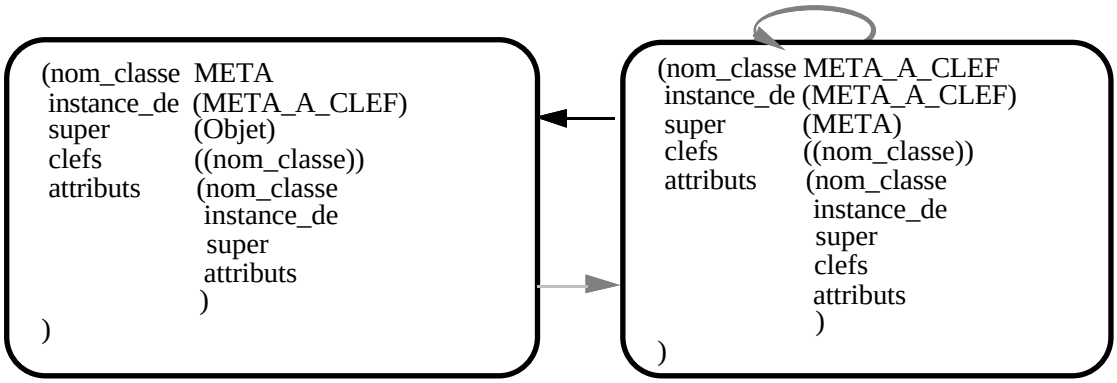


Figure 4.5. Modélisation finale: (*nom_classe*) est une clef pour toutes les instances de META.

L'utilité de ces deux méta-classes est facilement justifiable. META est directement ou indirectement la méta-classe des classes dont les instances ne sont pas identifiées par des clés. META_A_CLEF est directement ou indirectement la méta-classe des classes dont les instances sont identifiées par des clés.

Deux exemples de l'utilité de META et META_A_CLEF sont les classes Adresses, une classe sans clef, et Personnes, une classe ayant deux clefs, présentées dans la figure 4.6.

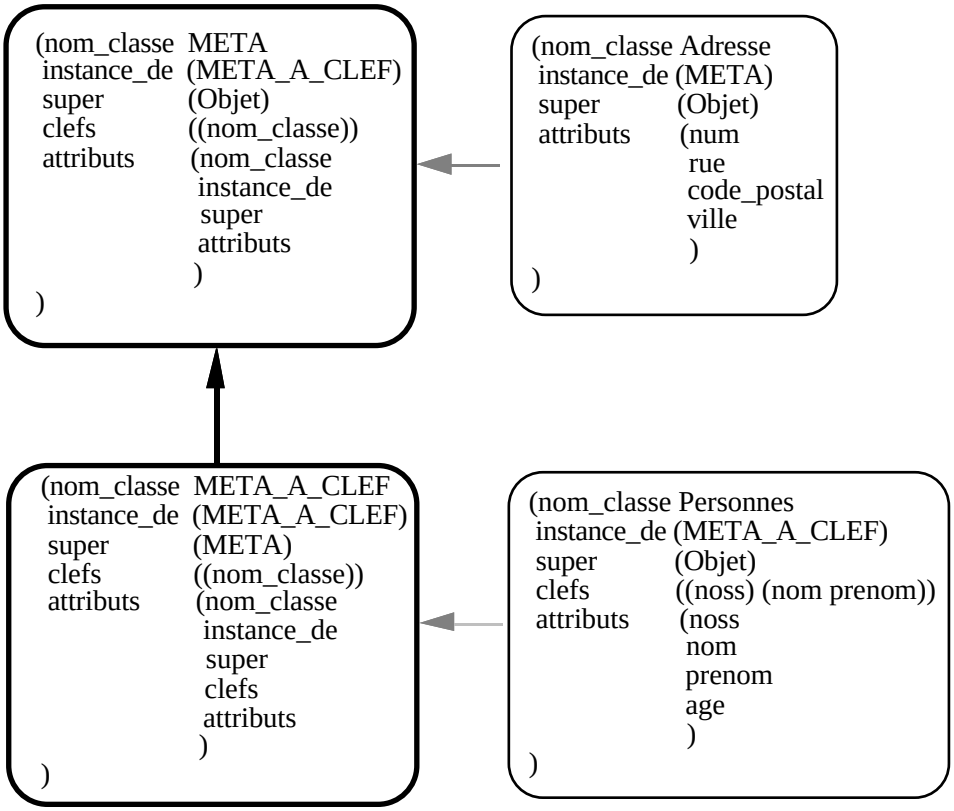


Figure 4.6. La classe Adresses, instance de META n'admet pas de clef. La classe Personnes, instance de META_A_CLEF, admet deux clefs

4.4.3. Comportement

Un fois la structure nécessaire pour définir des classes à clef créée nous pouvons nous pencher sur le comportement. Les classes à clef diffèrent des classes normales, instances de META, par la mise à jour des clefs. Nous devons donc définir une nouvelle méthode dans le graphe de méthodes de Creer_objet qui modélise le comportement des classes à clef.

Pour modéliser le comportement des classes à clef nous créons donc une méthode Creer_objet_a_clef, sous-classe de Creer_objet, qui spécialise l'attribut *classe* à (un META_A_CLEF). Ce comportement peut être divisé en deux parties correspondant à chacun deux traitements de la méthode:

Le traitement *principal*. D'abord il faut créer l'objet lui même. Cette tâche est accomplie en récupérant du code soit de la super-méthode en cas d'instanciation simple (héritage dynamique du traitement *principal*) soit d'une des méthodes sœurs en cas de multi-instanciation

(coopération des traitements de type *principal*).

Le *post* traitement. Avant de plébisciter la création de l'objet, il faut certifier que ses attributs vérifient l'unicité de la ou les clefs de la classe. Finalement, nous pouvons créer une entrée dans la ou les tables des clefs de la classe.

4.5. Modélisation des attributs

4.5.1. Représentation

4.5.1.1. Les classes-attributs

A l'instar de Mering [FER84], toute définition d'attribut est modélisée dans Shood par une classe appelée *classe-attribut*. C'est par l'intermédiaire de cette méta-classe que toutes les propriétés des attributs sont définies. La classe-attribut Segments.dimension.segments modélisant l'attribut *dimension* (nom complet *segments.dimension*) de la classe Segments est présentée dans la figure 4.7. META_ATTRIBUT est la classe d'instanciation des classes-attributs.

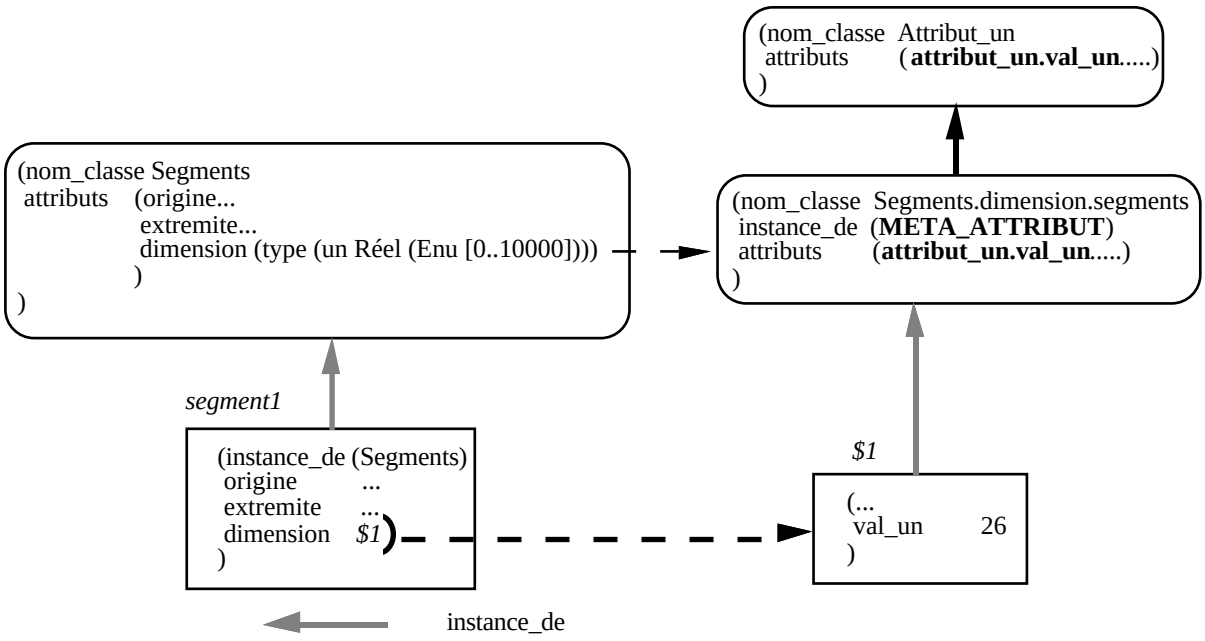


Figure 4.7. Les attributs sont modélisés par des classes-attributs (Segments.dimension.segments), les valeurs par des instances-valeurs ($1).

Nous pouvons observer que la valuation de *dimension* pour l'instance de la classe Segments

d'oid *segment1* correspond à la création de l'instance d'oid *$1* dans la classe Segments.dimension.segments. Cette dernière est donc la classe dont les instances modélisent les valeurs de l'attribut *dimension* pour les instances de Segments (par exemple pour *segment1*). Les instances des classes-attributs, telles que $1, sont appelées des *instances-valeurs* parce qu'elles contiennent les valeurs des attributs.

Pour la valuation des attributs, Shood fait appel à l'instanciation. En effet, valuer un attribut revient à créer une instance dans la classe le modélisant.

4.5.1.2. Spécialisation des attributs

Comme toute classe, une classe-attribut est identifiée par son nom grâce à l'attribut *nom_classe*. Le nom d'une classe-attribut pourrait correspondre au nom complet d'un attribut si les attributs n'étaient pas hérités puis modifiés. Le nom complet d'un attribut, par exemple *segments.dimension*, n'est pas suffisant parce que pour un attribut donné il existe tout un sous-graphe de classes-attributs représentant ses différents affinements.

Pour l'affinage des attributs, Shood fait appel à la spécialisation. En effet, affiner un attribut revient à créer une sous-classe de la classe le modélisant.

La racine d'un graphe des classes-attributs correspond donc à la définition de l'attribut dans sa classe d'origine. Les sous-classes du graphe correspondent aux définitions de l'attribut dans les sous-classes de sa classe d'origine. Ainsi, pour trouver une unicité de nom des classes-attributs le nom de la classe est constitué du nom complet de l'attribut suffixé du nom de sa classe de définition, c'est-à-dire soit sa classe d'origine, soit la classe où l'attribut est affiné. Ainsi la classe Segments.dimension.segments modélise l'attribut *segments.dimension* dans sa classe d'origine tandis que la classe Segments.dimension.segments_tolerants modélise l'attribut *segments.dimension* dans la classe Segments_tolerants.

4.5.1.3. Les classes-constructeurs

Une classe-attribut hérite toujours d'un attribut représentant un constructeur. Par exemple, l'attribut *val_un* dans l'instance $1 (figure 4.7) permet de modéliser le constructeur mono-valué *un*. Il permet de conserver la valeur (26) pour l'attribut *dimension* de l'instance *segment1*. L'attribut *val_un* est hérité de la classe Attribut_un dont Segments.dimension.segments est une sous-classe. Attribut_un correspond à un type de classes appelées *classes-constructeurs*.

Des classes-constructeurs existent pour chaque constructeur du modèle. Elles sont regroupées

sous la classe Attributs (figure 4.8).

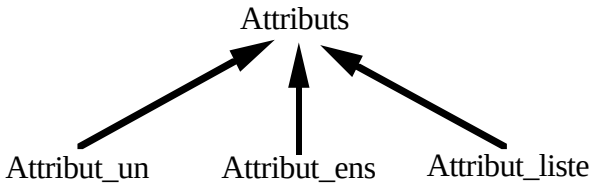


Figure 4.8. Les classes des constructeurs.

Les attributs des classes-constructeurs ne peuvent pas être représentés par des classes-attributs normales. En effet, si nous définissons les classes des attributs-constructeurs, tels que celle de l'attribut *val_un*, comme des classes-attributs normales, elles auraient elles-mêmes des attributs-constructeurs, ce qui ferait une boucle infinie! Par exemple si la classe Attribut_un.val_un.attribut_un était une classe-attribut normale, elle serait une sous-classe d'une classe-constructeur, disons Attribut_un, ce qui ferait une boucle au niveau du graphe de spécialisation impossible à résoudre. La récursivité due au "tout objet est instance d'une classe" de Shood doit bien être arrêtée quelque part.

La récursivité doit être arrêtée à l'endroit où elle commence: au niveau des classes-attributs des attributs-constructeurs (e.g Attribut_un.val_un.attribut_un). Par exemple, la figure 4.9 montre la définition de la classe-constructeur Attribut_un. Nous pouvons observer que l'attribut *val_un* est sous-classe de Attribut_virtuel, la super-classe des classes-attributs non-instanciables.

```
(nom_classe  Attribut_un
 attributs       (val_un    (type   ()
                                 super (Attribut_virtuel)
                                 )
                 )
)
```

Figure 4.9. La classe-constructeur Attribut_un (constructeur mono-valué).

Les valeurs d'attributs dont les classes-attributs sont une sous-classe de Attribut_virtuel sont représentées dans Shood de manière directe en utilisant les primitives du système hôte (Le_Lisp dans notre cas). Ceci veut dire que la classe Attribut_un.val_un.attribut_un n'a pas d'instances. Les valeurs de cet attribut sont représentées par des primitives lisp. Les classes-attributs des attributs-constructeurs (e.g Attribut_un.val_un.attribut_un) sont des classes spéciales dans le sens où elles modélisent un attribut n'ayant ni un constructeur ni un type à vérifier.

La modélisation des constructeurs par des classes permet de les modéliser de manière uniforme et de faciliter leur extension. Pour ajouter un constructeur nous devons donc ajouter une classe-constructeur et faire de sa classe-attribut une sous-classe de Attribut_virtuel.

Cette extensibilité est également utilisée lors du démarrage du système. En effet, dans l'amorce seules Attribut_un et Attribut_ens existent, les autres classes-constructeurs telle que Attribut_liste (figure 4.8) peuvent ajoutées par la suite grâce aux primitives Shood.

4.5.1.4. META_ATTRIBUT

La figure 4.10 montre la définition complète de META_ATTRIBUT et de la classe-constructeur Attribut_un. Les attributs hérités de META par META_ATTRIBUT ne sont pas montrés.

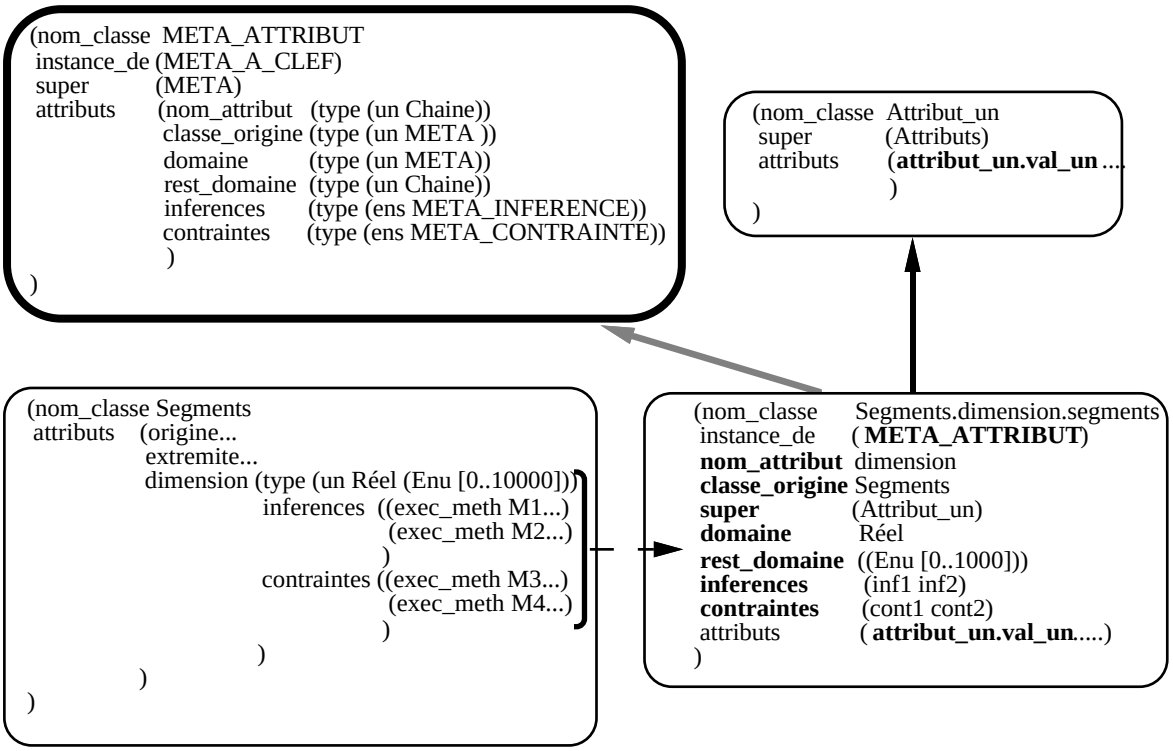


Figure 4.10. Définition complète de META_ATTRIBUT.

Les attributs propres à META_ATTRIBUT valués par la classe-attribut Segments.dimension.segments (en gras dans la figure 4.10) permettent la modélisation des propriétés des attributs: leur nom, type, inférences et contraintes. Nous allons les examiner en détail:

Le nom complet de l'attribut est stocké par les attributs *classe_origine* et *nom_attribut*. Pour Segments.dimension.segments leur valeurs sont "Segment" et "dimension" respectivement.

La modélisation du type d'un attribut par la classe-attribut est faite par trois parties différentes: constructeur, domaine et restrictions de domaine. Le constructeur mono-valué est modélisé en faisant de la classe-attribut une sous-classe de Attribut_un. Le domaine et la restriction de

domaine sont modélisés par les attributs *domaine* et *rest_domaine*. Pour Segments.dimension.segments leurs valeurs sont respectivement Réel et (Enu [0..1000]).

Les attributs *inférence* et *contrainte* permettent la définition des deux inférences et des deux contraintes sur *dimension*. Pour Segments.dimension.segments ils ont comme valeur (inf_1 inf_2) et ($cont_1$ $cont_2$) respectivement. Les inf_i et $cont_i$ sont les oids des classes modélisant les contraintes et les inférences définies sur *dimension*. La modélisation des inférences et des contraintes est présentée dans ce chapitre (cf. § La modélisation des contraintes et § La modélisation des inférences).

4.5.2. Comportement

Pour modéliser le comportement des classes-attributs nous créons donc une méthode Valuer_attribut, sous-classe de Creer_objet, qui spécialise l'attribut *classe* à (un META_ATTRIBUT). Cette classe-méthode a un argument supplémentaire *obj_proprietaire* modélisant l'objet dont on value l'attribut (i.e *segment1* pour notre exemple).

Valuer_attribut est constituée d'un traitement *principal*. Les *post* et *pré* traitements de celle-ci n'effectuent aucune opération. Dans le traitement *principal*, la valuation de l'attribut se fait en deux étapes:

Boucle d'inférence et vérification. Cette boucle exécute les inférences de l'attribut, une à une, en leur appliquant les restrictions de type et les contraintes. La boucle se termine lorsqu'une des inférences a réussie ou lorsque toutes les inférences ont échouées. Une inférence réussit si elle retourne une valeur vérifiant les descripteurs de type et de contraintes.

Création de l'instance-valeur et valuation de l'attribut. Si la boucle d'inférence et vérification réussit, une instance-valeur est créée et l'attribut-constructeur est mis à jour. Finalement, l'oid de l'instance-valeur (*$1*) est stocké dans obj_proprietaire (*segment1*).

4.6. Les attributs des attributs

4.6.1. Représentation

La modélisation des attributs par l'intermédiaire de classes permet la définition des attributs des attributs. La figure 4.11 montre la modélisation des classes-attributs pour l'attribut *dimension* présenté dans le Chapitre 2 (cf. § Les attributs des attributs). L'attribut *dimension* est défini

dans la classe Segments, puis raffiné dans Segments_tolerants. Dans celle-ci, un attribut d'attribut *coefficient_tolerance* est défini. Cet attribut exprime un coefficient de tolérance pour une dimension donnée.

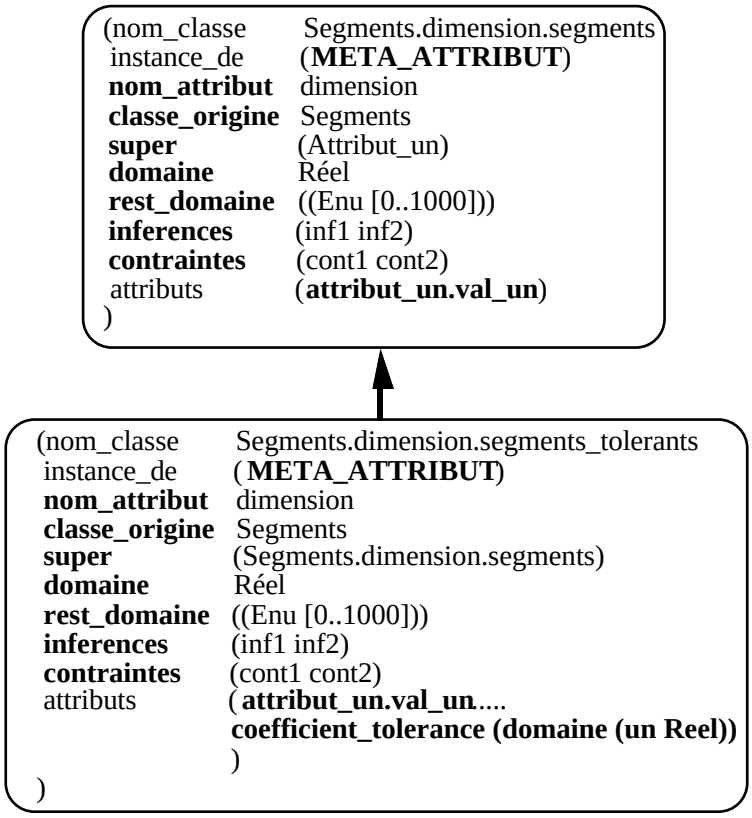


Figure 4.11. Modélisation de l'attribut *segments.dimension* des classes Segments et Segments_tolerants. Dans Segments_tolerants, *coefficient_tolerance*, un attribut d'attribut, a été défini.

Ainsi, pour toute dimension exprimée (i.e toute valuation de *val_un*) pour une instance de Segments_tolerants, nous pouvons valuer l'attribut *coefficient_tolerance*. La figure 4.12 montre la modélisation de l'instance du même exemple ayant une dimension de 30 et admettant un coefficient de tolérance de 0.07.

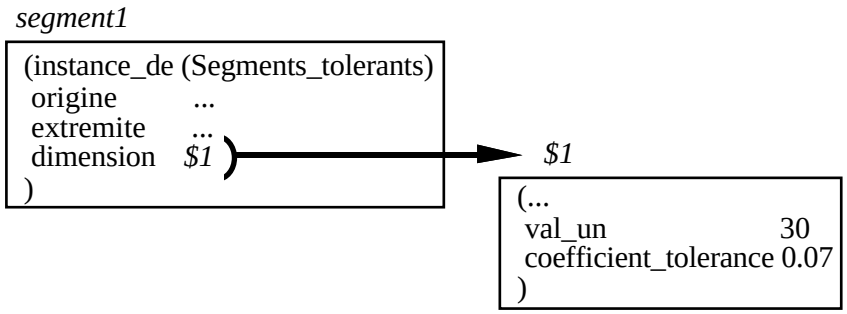


Figure 4.12. Les attributs des attributs au niveau des instances-valeurs.

Ce principe ne connaît pas de limitations à condition que l'attribut en question soit un attribut "instanciable" (i.e non virtuel). Par exemple le coefficient de tolérance peut être muni d'un attribut d'attribut tel que *saisi_par* modélisant la personne l'ayant saisi. Par contre, les attributs virtuels, tels que attribut_un.val_un, n'admettent pas d'attributs d'attribut car ils n'ont pas d'instances-valeurs, les valeurs étant représentées directement dans le système hôte.

4.6.2. Comportement

La modélisation des attributs d'attribut ne demande pas la création d'une nouvelle méta-classe. Ils sont modélisés par des classes-attributs standards.

Cependant, ces attributs doivent être valués lors de l'exécution de la méthode Valuer_attribut. Pour ce faire, nous ajouterons à son traitement *principal* un appel récursif de Valuer_attribut. Cet appel permettra éventuellement de valuer les attributs d'attributs. Pour notre exemple, nous devons valuer l'attribut coefficient_tolerance de l'instance d'oid $1. Pour ce faire, Valuer_attribut doit être appelée avec l'oid de la classe-attribut de *coefficient_tolerance* (valeur de *classe*) et avec *$1* (valeur de *obj_proprietaire*).

4.7. Les attributs obligatoires

4.7.1. Représentation

Il existe dans Shood des attributs obligatoires qui doivent toujours avoir une valeur et des attributs facultatifs qui peuvent ou non être valués lors de la création de l'objet (cf chapitre 2 § Les attributs obligatoires). Les attributs instances de META_ATTRIBUT, sont des attributs facultatifs. META_ATTRIBUT admet une sous-classe META_ATT_OBLIGATOIRE modélisant les attributs obligatoires (figure 4.13).

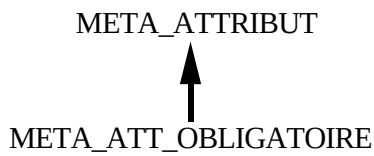


Figure 4.13. META_ATT_OBLIGATOIRE est une sous-classe de META_ATTRIBUT.

Cette méta-classe n'ajoute pas d'informations supplémentaires (i.e d'attributs). En effet un attribut obligatoire ne diffère d'un attribut standard (i.e facultatif) que dans son comportement.

4.7.2. Comportement

La modélisation du comportement des attributs obligatoires demande la création d'une nouvelle méthode dans le graphe de méthodes de Creer_objet. Nous ajoutons donc la méthode Valuer_obligatoire, sous-méthode de Valuer_attribut, spécialisant l'attribut *classe* à (un META_ATT_OBLIGATOIRE).

Cette méthode ne diffère de Valuer_attribut que dans sont *post* traitement. Dans celui-ci on vérifie que l'attribut a été valué par le traitement *principal* hérité dynamiquement de Valuer_attribut. Si ceci n'est pas le cas, une erreur est déclenchée.

4.8. La modélisation des méthodes

4.8.1. Représentation

Au chapitre 3, nous avons vue que dans Shood, les méthodes sont représentées par des classes, appelées *classes-méthodes*. Les classes-méthodes sont instances d'une méta-classe particulière, META_METHODE, laquelle spécialise META en ajoutant des attributs supplémentaires qui nous permettent de stocker les traitements qui se rapportent à la méthode. Le nom de la méthode est donné par le nom de la classe. La figure 4.14 montre une définition complète de META_METHODE. Les attributs hérités de méta ne sont pas montrés.

```
(nom_classe  META_METHODE
 instance_de (META_A_CLEF)
 super       (META)
 attributs    (pré        (type (un  Traitements))
               principal  (type (un  Traitements))
               post       (type (un  Traitements))
               attributs  (type (ens META_ARGUMENT_METHODE))
               )
)
```

Figure 4.14. META_METHODE, la méta-classe de classes-méthodes.

META_METHODE spécialise META de deux manières différentes:

• Elle a trois attributs supplémentaires: *pré, principal* et *post* qui modélisent les traitements de la méthode. Ces trois attributs prennent leurs valeurs dans la classe Traitements, la classe modélisant les traitements présentée dans le paragraphe suivant.

• Elle spécialise aussi l'attribut *attributs* hérité de META. Elle restreint les valeurs d'*attributs* a META_ARGUMENT_METHODE, une sous-classe de META_ATTRIBUT qui est la racine

des arguments de méthodes. Les arguments de méthodes sont expliqués dans ce chapitre (§ Les arguments de méthodes).

Suivant notre principe d'uniformisation "tout est objet", les traitements et les contrôles de séquentialisation sont modélisés par des objets. La modélisation de ceux-ci est présenté dans [MIL92].

4.8.2. Le comportement des méthodes

L'exécution des méthodes est modélisée par la méthode Executer_methode du graphe de méthodes Creer_objet. Cette méthode spécialise l'argument *classe* à (un META_METHODE).

Nous résumons les différentes étapes dans l'exécution des méthodes. Leur exécution a été montrée dans le chapitre 2 à l'aide d'un exemple (cf § Séquentialisation des traitements dépendants). Nous rappelons au lecteur que chacune de ces étapes correspond en fait à une méthode du graphe de méthodes Creer_objet.

Pour exécuter une méthode l'utilisateur fournit le nom de la méthode qu'il veut exécuter et les valeurs de ses arguments en entrée. S'il s'agit d'une exécution "do_meth" Shood sélectionne cette méthode et passe à l'étape suivante. S'il s'agit d'une exécution "exec_meth" Shood demande d'abord à l'algorithme de classification de trouver les classes-méthodes les plus spécifiques satisfaisant les arguments en entrée des méthodes.

Une fois que les méthodes à instancier sont trouvées, Shood exécute successivement les étapes *pré*, *principal* et *post*. Pour chacune de ces étapes, Shood collecte les traitements concernés. Si une méthode n'a pas de traitement défini, Shood cherche les traitements des super-méthodes. Cette opération est effectuée récursivement dans les super-méthodes. Une fois qu'il a collecté tous les traitements, Shood obtient un ordre valable d'exécution. Un ordre valable est un ordre qui respecte les contrôles de séquentialisation. Il exécute dans cet ordre les traitements *pré* et *post*. Pour les traitements de type principal, il en exécute un seul. S'il y en a plusieurs on déclenche une erreur.

4.9. Modélisation des inférences

4.9.1. Représentation

Nous avons vu dans le chapitre 3 (cf. § Les inférences) qu'une inférence permet de calculer la

valeur d'un attribut en utilisant une méthode. Dans Shood les inférences intra-attribut sont modélisées par des classes, appelées *classes-inférences*, qui ont une structure et un comportement spécifique. Cette structure est modélisée par META_INFERENCE, la méta-classe de toutes les inférences. Les classes-inférences sont générées automatiquement par le système au moment de la définition d'une inférence. Au niveau du graphe de spécialisation, toutes les inférences sont regroupées sous *Inferences*, la super-classe de toutes les classes-inférences.

La figure 4.15 montre la définition de META_INFERENCE, seuls les attributs qui diffèrent de ceux META sont montrés:

- *méthode*, modélisant la méthode à exécuter, ainsi que
- *attributs*, modélisant les arguments en entrée et en sortie de la méthode.

Le paragraphe suivant explique la modélisation des arguments des inférences.

```
(nom_classe  META_INFERENCE
 super       (META)
 attributs   (methode (type (un  META_METHODE))
              attributs  (type (ens META_ARGUMENT_INF))
             )
)
```

Figure 4.15. META_INFERENCE est la méta-classe des classes-inférences.

4.9.2. Les arguments des inférences

Comme nous l'avons dit dans le chapitre précédent, dans Shood il existe trois manières de fournir des arguments en entrée à une inférence. Les arguments étant modélisés par des attributs, nous définissons des sous-classes (indirectes) de META_ATTRIBUT qui modélisent le comportement spécifique de ces arguments:

- META_ARGUMENT_INF_CONT_ENTREE modélise le comportement des arguments en entrée calculés par une inférence. Ceci est le cas des compositions d'inférences. Le calcul de l'argument est fait à partir de l'attribut *inférences* définit dans META_ATTRIBUT.

- META_ARGUMENT_INF_CONT_FINAL modélise le comportement des arguments ayant comme valeur des constantes ou des variables.

- META_ARGUMENT_INF_SORTIE modélise le comportement des arguments en sortie des inférences. Ceux-ci permettent de valuer le ou les attributs sur lesquels l'inférence porte.

4.9.3. Comportement des inférences

L'exécution des inférences est modélisée par la méthode Executer_inference du graphe de méthodes Creer_objet. Cette méthode spécialise l'argument *classe* à (un META_INFERENCE).

Pour exécuter une méthode, l'inférence calcule d'abord les valeurs des arguments en entrée de la méthode. Ces arguments peuvent être fournis de trois manières: comme une constante, comme une variable ou comme une autre inférence. Une fois les arguments en entrée calculés, l'inférence fait exécuter la méthode: d'abord les arguments en entrée de la méthode sont valués, puis le code est exécuté. Une fois la méthode exécutée, l'inférence récupère ses sorties. Finalement l'inférence utilise ses sorties pour valuer le ou les attributs sur lesquels elle porte.

Dans une classe-inférence le passage des arguments est assuré par ses attributs. En effet, une classe-inférence a un attribut (argument en entrée) pour chacun des arguments en entrée de la méthode et un attribut (argument en sortie) pour chacun des attributs sur lesquels elle porte.

Exécuter une inférence revient donc à instancier la classe-inférence qui la modélise. L'instance créée représente l'exécution de l'inférence: arguments en entrée et arguments en sortie.

4.10. Modélisation des contraintes

4.10.1. Représentation

Les contraintes, comme les inférences sont modélisées par des classes appelées *classes-contraintes*. La structure des classes-contraintes est donnée par la méta-classe META_CONTRAINTE (figure 4.16). Elle définit un attribut supplémentaire *méthode* qui contient la méthode vérifiant la contrainte, et spécialise l'attribut *attributs*. Ce dernier voit ses valeurs restreintes aux classes des arguments des contraintes. Au niveau du graphe de spécialisation, toutes les contraintes sont regroupées sous *Contraintes*, la super-classe de toutes les classes-contraintes.

```
(nom_classe  META_CONTRAINTE
 super       (META)
 attributs   (methode (type (un  META_METHODE))
              attributs  (type (ens META_ARGUMENT_CONT))
              )
)
```

Figure 4.16. META_CONTRAINTE, la méta-classe des contraintes.

La modélisation des contraintes est similaire à celle des inférences. Le paragraphe suivant explique la modélisation des arguments des contraintes.

4.10.2. Les arguments des contraintes

Nous avons vu dans le chapitre 3 qu'une contrainte et une inférence ont beaucoup de points en commun. En ce qui concerne les arguments, si les inférences et les contraintes partagent les méta-classes des arguments en entrée, ce n'est pas le cas pour ceux en sortie. En effet, à la différence des arguments en sortie des inférences, ceux des contraintes ne doivent pas valuer un attribut. Pour cela META_ARGUMENT_CONT_SORTIE, la méta-classe des arguments en sortie de contraintes, ne définit pas d'attribut *attribut_sor* comme META_ARGUMENT_INF_SORTIE le fait.

4.10.3. Comportement des contraintes.

La vérification des contraintes est modélisée par la méthode Verifier_contrainte du graphe de méthodes Creer_objet. Cette méthode spécialise l'argument classe à (un META_CONTRAINTE).

Les opérations réalisées pour vérifier une contrainte sont similaires à celles réalisées pour exécuter une inférence. La seule différence est qu'une contrainte ne value pas d'attribut.

4.11. L'amorce

Shood est un système méta-circulaire car il se définit en ses propres termes. Probablement le cas le plus représentatif d'auto-définition dans Shood est celui des méta-classes. En effet, META, la méta-classe primitive dont toutes les classes sont instances, est instance d'elle-même. META s'auto-définit.

La mise en œuvre d'un système tel que Shood demande une construction par étapes. Cette construction est facilitée par l'utilisation des primitives du système telle que la primitive de création des classes. Cependant, cette primitive ne peut pas créer META. Cette classe doit exister pour que la primitive de création de classe puisse fonctionner correctement. Les primitives de Shood sont donc incapables d'amorcer le système. Elles ne peuvent fonctionner que sur une définition minimale de celui-ci.

La définition minimale du système sur laquelle les primitives peuvent fonctionner est appelée l'*amorce*. L'amorce de Shood comprend donc la définition d'objets minimaux: méta-classes, classes et instances terminales nécessaires au système.

Cette définition est ensuite augmentée par les primitives de manipulation.

Nous présentons ici la justification du choix des classes de l'amorce. L'amorce doit définir des graphes d'instanciation et de spécialisation minimaux. Nous examinons ici les classes nécessaires pour construire ces deux graphes. Ensuite, nous définissons les attributs minimaux de ces classes. Finalement, étant donné que les attributs sont modélisés par des classes, nous définissons les classes-attributs de l'amorce.

4.11.1. Le graphe d'instanciation

Seules deux classes sont nécessaires pour construire le graphe d'instanciation minimal:

• META, la méta-classe primitive. META est la classe d'instanciation de toutes les classes standards.

• META_ATTRIBUT, la méta-classe des classes-attributs. La création des classes de l'amorce demande aussi la création des attributs. Seule META_ATTRIBUT est nécessaire dans l'amorce, ses sous-classes sont ajoutées par la suite.

4.11.2. Le graphe de spécialisation

Le graphe de spécialisation minimal nécessite la définition de six classes. Toutes ces classes sont instances de META:

• Univers, la racine du graphe de spécialisation.

• Objet_virtuel, la super-classe des attributs virtuels et des classes scalaires. Les classes Attribut_virtuel et Scalaires ne sont pas définies dans l'amorce car elles peuvent être définies plus tard par des primitives système.

• Chaîne, la classe des chaînes. Dans l'amorce, Chaîne est sous-classe de Objet_virtuel, plus tard elle deviendra sous-classe de Scalaires.

• Objet, la super-classe de toutes les classes instanciables. Objet est la super-classe de META

et, par transitivité, de META_ATTRIBUT.

• Attribut_un et Attribut_ens, les classes des constructeurs *un* et *ens* respectivement. Tous les attributs de l'amorce sont définis soit par un constructeur *un* soit par un constructeur *ens*. Les autres constructeurs ne sont pas nécessaires dans l'amorce car ils peuvent être ajoutés par la suite. Dans l'amorce, Attribut_un et Attribut_ens sont sous-classes de Objet. Plus tard elles deviendront sous-classes de Attributs, la super-classe de toutes les classes-constructeurs.

La figure 4.17 montre les graphes de spécialisation et d'instanciation des classes de l'amorce que nous avons présentées. Toutes ses classes sont instances de META. Il ne nous reste qu'à examiner les classes-attributs.

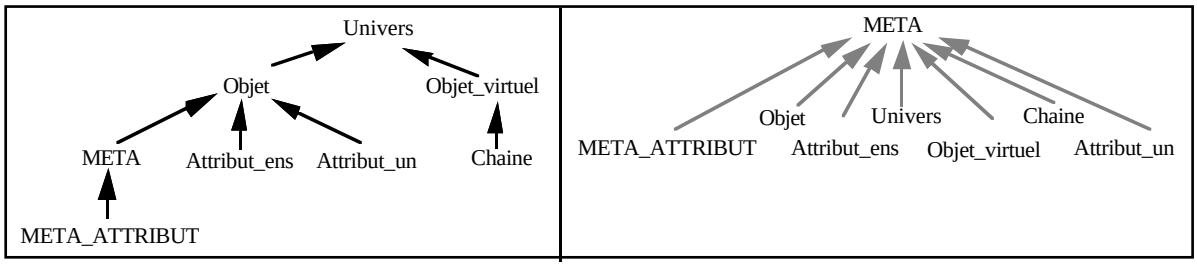


Figure 4.17. Les graphes de spécialisation et d'instanciation minimaux. Les classes-attributs ont été omises.

4.11.3. Les classes-attributs de l'amorce

Examinons maintenant les attributs des classes de l'amorce:

La classe Objet a besoin d'un seul attribut: *instance_de*. L'attribut *instance_de* est défini dans Objet car Objet est la super-classe de toutes les classes instanciables. Les objets virtuels n'héritent pas de *instance_de*. Afin d'éviter une boucle infinie, *instance_de* est modélisé par une sous-classe de Objet_virtuel. En effet si *instance_de* n'était pas un Objet_virtuel il hériterait de lui-même.

La méta-classe META doit définir les attributs que toute classe doit instancier: *nom_classe*, *super*, *sous*, *attributs*, *instances*. Les définitions de ces attributs sont présentées plus haut dans ce chapitre (cf. § META: La définition minimale d'une classe). META hérite de l'attribut *instance_de* de la classe Objet.

La méta-classe META_ATTRIBUT doit définir les attribut minimaux pour modéliser des attributs. Dans l'amorce, META_ATTRIBUT comporte les attributs: *nom_attribut*, *classe_origine*, *domaine*. Les attributs *rest_domaine*, *inferences* et *contraintes* sont ajoutés par

des primitives système.

Finalement, les classes-constructeurs Attribut_un et Attribut_ens définissent les attributs *val_un* et *val_ens* respectivement. Les attributs *val_un* et *val_ens* sont représentés par des sous-classes de Objet_virtuel afin d'éviter une boucle infinie.

L'amorce de Shood est compact: Seul dix-neuf classes sont nécessaires. Les classes de l'amorce modélisent un minimum de concepts: méta-classes, classes, attributs et types des attributs. Les autres concepts de base tels que les inférences, les contraintes et les méthodes sont ajoutés par la suite grâce au caractère réflexif du modèle

Dans le modèle ObjVlisp, seules deux classes, Objet et Class (la méta-classe primitive), sont définies à la fin de la procédure d'amorce [COI87]. Ceci est dû au fait que, dans ObjVlisp, les attributs ne sont ni modélisés par des classes, ni typés.

4.12. Récapitulatif des méta-classes et de Creer_objet

Nous récapitulons ici les méta-classes et méthodes de Creer_objet qui ont été présentées dans ce chapitre. La figure 4.18 montre à gauche le graphe de méta-classes et à droite le graphe de méthodes associées à chaque méta-classe.

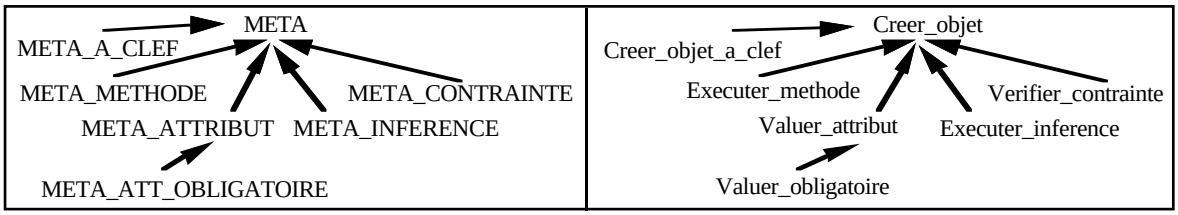


Figure 4.18. Graphe des méta-classes et de ses méthodes associées du graphe Creer_objet.

4.13. Conclusion

L'extensibilité de Shood réside dans la réflexivité du modèle: Shood modélise sa propre structure et son propre comportement. La structure de concepts de Shood est modélisée par des méta-classes. Leur comportement est modélisé par des graphes de méthodes. Ainsi, un graphe de méthodes (i.e fonction générique) est défini pour chaque type de manipulation d'objets (e.g création, modification, etc.)

Nous avons présenté ici la modélisation de la structure des concepts de base de Shood ainsi que

de leur comportement lors de la création d'objets. Ce comportement est modélisé par le graphe Creer_objet. Par exemple, nous avons présenté la modélisation des attributs standards. Leur structure est modélisée par META_ATTRIBUT. Leur comportement est modélisé par la méthode Valuer_attribut du graphe de méthodes Creer_objet.

La modélisation des nouveaux concepts a permis de montrer l'extensibilité du modèle. Par exemple, nous avons montré la modélisation des extensions sur les attributs introduites dans le chapitre 2 (cf. § Plus sur les attributs) à savoir les attributs des attributs et les attributs obligatoires. Nous avons montré que la modélisation des attributs par des classes permet d'augmenter facilement leur pouvoir d'expression. Ces extensions sont modélisées dans Shood d'une part, par l'ajout d'une méta-classe, sous-classe de META_ATTRIBUT, et d'autre part, par l'ajout, dans le graphe de méthodes de Creer_objet, d'une sous-méthode de Valuer_attribut.

Conclusion

Bilan

Cette thèse traite de la définition et de la réalisation de Shood [NGU92b] [RIE92a], un système de représentation de connaissances adapté aux besoins de la CAO. L'expression de ces besoins a permis de dégager trois points essentiels.

• En ce qui concerne la nécessité de doter le modèle d'une puissance de représentation:
- La réutilisation des objets
- La réutilisation des structures
- La réutilisation des procédures

• En ce qui concerne la possibilité d'y intégrer des outils de calcul:
- La modélisation des nouvelles méthodes de calcul
- La modélisation des outils de calcul existants

• En ce qui concerne la prise en compte de l'évolution des besoins des applications:
- La dynamique de la connaissance représentée
- L'extension des concepts du système

A partir de l'étude de l'existant dans les systèmes orientés objet, nous avons défini un cahier des charges s'inspirant principalement des modèles issus des Langages Orientés Objet (ObjVlisp [COI87], Clos [DEM87], CommonLoops [BOB85]) et des Systèmes de Représentation de Connaissances Centrée Objet (Shirka [REC85], KRL [BRA85a], Mering [FER84]) (cf. chapitre 1. Les systèmes à objets).

Ceci nous a permis de définir un modèle intégrant des caractéristiques que l'on trouve rarement rassemblés dans un même système.

Les aspects déclaratifs

Nous résumons ici comment les notions déclaratives de Shood permettent de combler les besoins de puissance de représentation des applications CAO.

La dynamique des points de vue

Les applications de CAO présentent des besoins différents de représentation multiple. Plusieurs types de problèmes de représentation multiple peuvent être modélisés par les concepts de base de Shood. Par exemple, l'instanciation multiple permet la modélisation des points de vue à forte dynamique [RIE91b].

La réutilisation des structures

Les structures sont définies en Shood par des classes. La réutilisation des structures est possible dans Shood grâce au mécanisme d'héritage. Celui-ci permet l'affinage ou la spécialisation des classes. Par ailleurs, l'héritage multiple augmente le partage de structures en permettant de combiner les descriptions de plusieurs classes [STE86].

La sémantique ensembliste du modèle

La sémantique des liens structurant la connaissance est fondée sur des relations ensemblistes. Ainsi, les liens de spécialisation, de disjonction et d'instanciation correspondent aux notions de sous-ensemble, d'intersection vide et d'appartenance à l'ensemble, respectivement [ESC89]. Par ailleurs, la sémantique ensembliste de ces liens a été utilisée par un mécanisme de classification incorporé dans le système [LIO93][RIE91a].

Le pouvoir déclaratif des attributs

Le pouvoir déclaratif des attributs de Shood réside dans leurs descripteurs de base et dans la possibilité de les enrichir. Les descripteurs de base de Shood permettent de définir le type d'un attribut, ses inférences et ses contraintes procédurales [ESC90a]. L'ensemble des descripteurs peut être enrichi. Par exemple, si nous voulons gérer la réutilisation d'objets, nous pouvons étendre les descripteurs pour définir des dépendances, telles que la dépendance exclusive [DJE93].

Les aspects procéduraux

La modélisation des outils de calcul

Les outils de calcul sont modélisés dans Shood par les méthodes. Le concept de méthode de Shood s'inspire de celui de fonction générique [BOB85][DEM87]. Beaucoup d'attention a été

portée dans Shood au mécanisme de sélection permettant de choisir la procédure la plus adaptée. Les critères de cette sélection se basent sur les caractéristiques des informations fournies en entrée aux outils. Ils sont au nombre de trois [MIL92][RIE92c]:

- Tous les arguments en entrée sont pris en compte et leur ordre n'est pas important.
- Les types et les contraintes procédurales des arguments sont pris en compte.
- Les arguments optionnels sont également pris en compte.

Ces critères permettent la sélection de la meilleure méthode de calcul en fonction des informations disponibles en entrée. En effet, un seul graphe de méthodes peut modéliser plusieurs outils de calcul ayant le même but mais des précisions différentes. Par exemple, si les informations en entrée ne sont pas suffisantes pour l'exécution d'une méthode de calcul de haute précision, on choisira d'exécuter une méthode aproximative plus adaptée à l'étape de prototypage.

La réutilisation des procédures

La réutilisation des procédures est rendue possible dans Shood grâce aux mécanismes de spécialisation déclarative et de spécialisation procédurale de méthodes. Ces deux notions orthogonales offrent chacune des approches différentes pour combiner (réutiliser) les méthodes:

- Dans la spécialisation procédurale, la combinaison de méthodes est sous la responsabilité des consommateurs [BOB81].
- Dans la spécialisation déclarative, la combinaison de méthodes est sous la responsabilité des producteurs [WEI81].

Inférences et cohérence

Les connaissances procédurales définies sur les attributs permettent d'une part, d'enrichir leur sémantique et d'autre part, de maintenir la cohérence de la base lors des modifications. Les connaissances procédurales sur les attributs sont définies dans Shood par les inférences et les contraintes [NGU91a]:

- Les inférences permettent le calcul de valeurs lors des opérations de création.
- Les contraintes procédurales permettent de contrôler la cohérence de valeurs lors des modifications.

Par ailleurs, contraintes, inférences et méthodes sont nécessaires à l'intégration des outils de

calcul, nouveaux ou existants. En effet, si un graphe de méthodes modélise un ou plusieurs outils de calcul, les inférences et les contraintes modélisent leur utilisation.

L'évolution des besoins

L'extensibilité

L'objectif d'extensibilité a été atteint dans Shood par la réflexivité: les concepts de Shood sont définis en termes d'eux-mêmes [RIE92b]. Ceci facilite les extensions car Shood peut être étendu par des utilitaires écrits dans le modèle lui-même [MAS89]. Ainsi, si des nouveaux besoins apparaissent, tels que la définition des classes à clef ou l'ajout de dépendances sémantiques, le modèle peut être étendu pour les supporter [DJE93]. Shood est donc un système ouvert et adaptable à de nouveaux besoins.

Dynamique des applications

Le méta-niveau de Shood permet de pallier au besoin dynamique des applications. En effet, la création de classes lors de l'exécution est rendue possible par les méta-classes. Ceci permet la gestion dynamique des changements de structure (classes) et des liens (spécialisation, instanciation). D'autre part les changement de valeurs sont gérés par la définition des inférences et des contraintes.

Réalisation et utilisation

Le modèle Shood a été implanté sur station de travail (d'abord Macintosh et ensuite SparcSation II, PC486 et Silicon Graphics Indigo) en Le_Lisp. Le prototype développé a été transféré au Laboratoire d'Electrotechnique de Grenoble et au laboratoire 3S (Sols, Solides et Structures) de l'Institut de Mécanique de Grenoble dans le cadre du projet du Groupement Scientifique Interdisciplinaire en Productique où il est utilisé pour la conception d'objets mécaniques et électrotechniques. Il est également utilisé comme un "cahier des charges vivant" dans une étude pour la gestion de projets de constructions immobilières [CUL93] menée avec le laboratoire CRISTO et l'UDEC, un groupement d'entreprises du second œuvre.

L'utilisation de Shood a mis en évidence des besoins qui correspondent enfin à deux étapes du processus de modélisation: la nécessité d'une méthodologie de modélisation et la nécessité de différents niveaux d'interface.

Méthodologie

Lors des réunions de travail avec les utilisateurs de Shood, la nécessité d'une méthodologie leur permettant une meilleure conception par objets est souvant apparue. Nous avons pu constater qu'il existe peu de travaux concernant les méthodologies de conception pour un modèle à objets. Du moins, la plupart des méthodologies objets existantes, telles que SYS_P_O [JAU92] ou OOA [COA93], provenant du génie logiciel, n'ont pas les mêmes préoccupations que celles des utilisateurs de Shood. En effet, ces méthodologies s'intéressent surtout à la réutilisation et à la maintenance.

Si une méthodologie existante devait être appliquée à Shood, celle-ci devrait d'abord être adaptée à ses concepts. En effet, étant donnée la richesse des concepts de Shood, la solution à une catégorie de problèmes ne correspond pas forcément à l'application directe d'un concept mais à l'application d'un concept choisi parmi plusieurs. Tel est le cas de la modélisation des points de vue dans Shood [RIE91b]. Le choix de la solution la plus adéquate parmi cet ensemble de concepts demande une bonne connaissance des fonctionnalités de Shood. Une première étude concernant la définition d'une méthodologie de conception d'applications pour Shood a fait l'objet d'un mémoire de DEA [GUI93].

Niveaux d'interface

Un autre problème constaté est celui de fournir une interface adaptée aux besoins des utilisateurs. Un moyen de pallier à ce problème est de fournir plusieurs niveaux d'utilisation. Au moins trois niveaux peuvent être préconisés.

Un premier niveau serait adapté aux utilisateurs-concepteurs souhaitant étendre le modèle. Ces utilisateurs doivent avoir accès à toutes les fonctionnalités de Shood. Ce niveau correspond en effet à l'interface programmable [SHO93].

Un deuxième niveau serait adapté aux utilisateurs du système ne souhaitant pas étendre le modèle. L'interface de ce niveau correspond à un "câblage" de concepts définis dans le premier niveau de manière à alléger leur utilisation. Un utilisateur du premier niveau qui fait une extension au modèle pourrait éventuellement étendre le deuxième niveau pour y ajouter de manière câblée les nouvelles fonctionnalités.

Finalement, un troisième niveau correspondrait au niveau des applications. Il s'agit ici d'une interface ad hoc. Chaque application Shood pourrait éventuellement posséder un niveau de ce type offrant une interface adaptée aux besoins de ses utilisateurs terminaux. Ceci est le cas de

l'interface du "cahier des charges vivant" pour la gestion de projets de construction [CUL93].

Perspectives

La représentation de connaissances pour la CAO est un problème complexe faisant appel à plusieurs domaines de l'informatique tels que l'intelligence artificielle, le génie logiciel et les bases de données. Pour cela, certains besoins ont été volontairement négligés dans l'étude présentée dans cette thèse. Nous citerons ici trois besoins concernant les interactions procédurales-déclaratives, la persistance et les performances.

Les interactions procédurales-déclaratives

Dans cette thèse, nous avons étudié ces interactions d'une position "très proche du procédural": la gestion de procédures. Ceci nous a amené au concept de méthodes [GOL84][DEM87]. Au moins trois autres approches "plus proches du déclaratif" méritent d'être étudiées.

Un niveau supérieur de gestion des connaissances procédurales qui n'a pas été traité ici est celui de l'organisation des méthodes de calcul par des tâches. Des techniques similaires à celles de méthodes Shood ont été utilisées dans les systèmes de tâches [CHA89][CHA89a][ORS90] pour organiser les méthodes de calcul. L'adaptation à Shood d'un mécanisme de ce type bénéficierait des techniques existantes concernant les méthodes.

Une autre approche est celle des règles événementielles du type <événement, condition, action> [DAY88]. Ce type de règles est bien adapté à l'expression de connaissances expertes. Une étude est en cours sur l'incorporation dans Shood de règles événementielles [BER93]

Une dernière approche est celle des réflexes qui à été présentée dans le premier chapitre de cette thèse (cf. § Les connaissances procédurales sur les attributs). Bien que cette technique ne soit pas suffisante car, comme nous l'avons dit, l'ordre de déclenchement des réflexes du même type sur différents attributs peut être très important, elle a l'avantage de permettre l'expression des connaissances procédurales à l'endroit concerné: les attributs.

Les problèmes pouvant être résolus par les techniques des réflexes, tâches et règles se recouvrent. Cependant, certaines techniques sont mieux adaptées pour résoudre un certain type de problèmes. Par exemple, les réflexes permettent de mieux exprimer des connaissances sur un attribut, alors que les tâches sont mieux adaptées à l'expression des connaissances sur un processus de décision. Il serait intéressant d'étudier leur coexistence et leur interaction.

La persistance

Le cahier des charges d'un gestionnaire de la persistance pour un modèle de représentation de connaissances tel que Shood recouvre partiellement celui des gestionnaires de bases de données classiques. Par exemple, nous retrouvons les besoins de fiabilité, de manipulation de grands volumes d'information et de bonnes performances. Nous devons ajouter à ce cahier des charges le support physique de notions non-classiques inhérentes au modèle à objets de Shood telles que la manipulation d'objets complexes (les objets ne sont pas en 1ère Forme Normal [DEL82] et il référencent d'autres objets), la multi-instanciation, l'héritage multiple, la disjonction et la réflexivité. Une étude est en cours sur l'incorporation de la persistance dans Shood [BOU93]. Cette étude définit également le modèle de persistance de Shood. Un *modèle de persistance* [ATK89] définit l'ensemble de règles d'attribution de la persistance et de manipulation des objets persistants du système.

Les performances

Nous avons dit au chapitre 1 (cf. § L'extensibilité) que l'utilisation d'un niveau méta diminuait les performances du système. Au cours des expérimentations réalisées avec le prototype du modèle, nous nous sommes vite aperçus que des améliorations étaient possibles. C'est ainsi que dans la version la plus récente du prototype nous avons intensifié l'utilisation des attributs virtuels, plus performants que les attributs standard. La modélisation des inférences et des contraintes a également été optimisée. En effet, leur modélisation par des classes ne semble pas assez justifiée car, à la différence des méthodes, les inférences et les contraintes ne tirent pas profit des liens de spécialisation, ni des types des attributs.

D'autres études visant à améliorer les performances des mécanismes de base peuvent être également envisagées. L'amélioration de l'un des mécanismes de base les plus utilisés, le mécanisme d'instanciation, aurait probablement les retombées les plus importantes sur le plan des performances.

Bibliographie

[ATK89] ATKINSON M., MORRISON R. Persistent System Architectures. In Persistent Object Systems, Workshops in Computing, dirigé par J. Rosenberg et D. Koch. Springer-Verlag, 1989.

[ATK89a] ATKINSON M., BANCILHON F., DEWITT D., DITTRICH K., MAIER D., ZDONIK S. The Object-Oriented Database System Manifesto. Dans Proceedings of the First International Conference on Deductive and Object-Oriented Databases. Kyoto, Japan, Decembre 1989.

[BAN87] BANERJEE J., W. Kim, H. Kim at H. Korth: "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", Proceedings ACM/SIGMOD annual Conference on Management of Data, San Francisco, (CA), Mai 1987.

[BAN88] BANCILHON F. et al: "The desing and implementation of O2, an objet oriented database system", OODBS Workshop, Badmunster, West Germany, 1988.

[BAN89] BANCILHON F. et al. The O2 book. GIP Altaïr. 1989.

[BAN89] BANCILHON F., CLUET S., DELOBEL C. A Query Language for an Object-Oriented Database System. Proceedings of the second Workshop on DataBase Programming Languages. Salishan, Oregon, USA. June 1989.

[BER91] BERTINO B., MARTINO L. Object-Oriented Database Management Systems: Concepts and Issues. IEE Computer, Avril 1991.

[BER93] BERGUES P. Les objets actifs dans un système de bases de connaissances. Mémoire d'ingénieur CNAM. A paraître 1993.

[BOB77] BOBROW D.G., WINOGRAD T.. An overview of KRL, a Knowledge representation language. Cognitive Science, vol. 1 n. 1. 1977.

[BOB81] BOBROW D. G., M. STEFIK. The Loops manual. Tech.Report KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center.

[BOB83] BOBROW D.G, STEFIK M. The Loops manual: a data and object-oriented programming system for InterLisp. Xerox PARC. Palo Alto. 1983.

[BOB85] BOBROW D. G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdibel. CommonLoops: Merging Common Lisp and Object-Oriented programming. Xerox Palo Alto Research Center, Intelligent Systems laboratory Series ISL-85-8, Août 85.

[BOR86] BORNING A. H.: "Classes versus Prototypes in Object-Oriented Languages", Fall joint Conf. IEEE, Dallas, Nov 1986.

[BOU93] BOULENGER J. Introduction de la Persistance dans Shood. Mémoire de DEA en Informatique INPG. Septembre 1993.

[BOU94] BOUNAAS F. Dynamique dans les systèmes de représentation de connaissances à objets. Thèse de l'Institut National Polytechnique de Grenoble. A paraître 1994.

[BRA79] BRACHMAN R. J. On the Epistemological Status of Semantic Networks. Dans Associative Networks: Representation and Use of Knowledge by Computers. Edité par Findler N.V. Academic Press, New York 1979.

[BRA83] BRACHMAN R. J. : "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", dans Computer of the IEEE, vol 16, n. 10, Octobre 1983.

[BRA85] BRACHMAN R. J. I lied about the trees, or Defaults and Definitions in Knowledge Representation. The AI Magazine. Fall 1985.

[BRA85a] BRACHMAN R. J., SCHMOLZE J.G. An Overview of the KL-ONE Knowledge Representation System. Cognitive Science, Vol 9 No. 2, 1985.

[BRA91] BRACHMAN R. J. & al Living with Classic: When and How to use a KL-ONE-Like Language. Dans Principles of Semantic Networks, Explorations in the Representation of Knowledge. Edité par Sowa J. Morgan Kaufmann Publ. 1991.

[CAR84] CARDELLI L. A Semantics of Multiple Inheritance. Lecture notes on Computer Science, Conf. Semantics of datatypes, vol 173, Springer-Verlag 1984.

[CAR85] CARDELLI L., WEGNER P. On understanding types, data abstraction and polymorphism. ACM Computing Surveys. 17,4. Décembre 1985.

[CAR89] CARRE B. Méthodologie orientée objet pour la représentation des connaissances. Thèse de Doctorat. Université des Sciences et Techniques de Lille Flandres Artois. Janvier 1989.

[CAR90] CARRE, B., GEIB J.M. The Point of View notion for Multiple Inheritance. ECOOP/OOPSLA '90. Octobre 1990.

[CAR90a] CARRE B., LENNEKE D., GEIB J.M. Multiple and Evolutive Representation in the ROME language. Proc. of the Second International Conference TOOLS. Paris (FR) 1990.

[CAR91] CARRE, B., LENNEKE D. Inheriting Object-Oriented Features through Meta-Programming A Frame extension to ROME. Publication ERA no. 93 Février 1991.

[CHA89] CHANDRASEKARAN B., Smith J.W., Sticklen J. Deep Models and their relation to diagnosis. Artificial Intelligence in Medicine 1 1989.

[CHA89a] CHANDRASEKARAN B., Tanner M., Josephson J.R. Explaining Control Strategies in Problem Solving. IEEE Expert. Spring 1989.

[CLA85] CLANCEY W.J. Heuristic Classification. Artificial Intelligence Journal. V. 27 No. 4, 1985.

[COA93] COAD P., YOURDON. E. Analyse Orientée Objets. Masson 1992.

[COI87] COINTE P. Metaclasses are first class: the ObjVlisp model. Proc. 2nd OOPSLA ACM. Orlando (Fl.). Sept. 87.

[CON92] CONSTANT D. Expérimentation du modèle de données Shood sur un exemple de conception mécanique. Rapport de DEA de mécanique. INP Grenoble. Juin 1992.

[CUL93] CULET A., GUFFOND J.L., LECONTE G., MILLASSEAU J.F. Modélisation et échange de données techniques dans un groupement de PME. Soumis au 4ème congrès International de Génie des Systèmes Industriels. Marseille, décembre 1993.

[DAH66] DAHL O., K. Nygaard. Simula an Algol-based simulation language. Comm. ACM, vol 9, 1966.

[DAN88] DANFORTH S., TOMLINSON C. Type theories and object-oriented programming. ACM Computing Surveys. Vol 20, n. 1. Mars 1988.

[DAV91] DAVID B.T. & al. Environement Informationnel pour le CIM: le projet CIM-ONE. 23ème CIRP Séminaire international sur les systèmes de production. Nancy, Juin 1991.

[DAY88] DAYAL U., BUCHMAN A., MCCARTHY D. Rules are objects too: a knowledge model for an active OODS. Advances in OODS, Septembre 1988.

[DEL82] DELOBEL C., ADIBA M. Bases de données et systèmes relationnels. Dunod, 1982.

[DEM87] DEMICHIEL L.G, Gabriel R.P The Common Lisp Object System : An Overview proc. ECCOP'87, Paris, 1987.

[DJE93] DJERABA C., NGUYEN G.T., RIEU D. Objet Composites et Liens de Dépendance dans un Système à Base de Connaissances. Inforsid '93. Lille, Mai 1993.

[DJE93a] DJERABA C. Les relations sémantiques dans un Système à Base de Connaissances. Thèse de Doctorat de l'Université Claude Bernard (Lyon 1). A paraître 1993.

[DJE93b] DJERABA C. Composite Objects and dependency relationships in a knowledge-based system. Proc. 8th Intl. Conf. AI in Engineering. Toulouse. Juin 1993

[DJE93c] DJERABA C. Production Information System Design. In proceedings of the International Conference on Industrial Engineering and Production Management (IEPM-93), Mons, Belgique, Juin 1993.

[DJE93d] DJERABA C., HSSAIN A., DECOTES-GENON. Composition and Dependency Relationships in Production Information Systems Design. In Proceedings of the Fourth International Conference on Data Base and Expert System Applications (DEXA-93). Septembre 1993.

[DUC87] DUCOURNAU R., M. Habib: "On some algorithms for multiple inheritance in object oriented programming", ECOOP'87, Paris, Juin 1987, dans BIGRE no. 54, Juin 87.

[DUC88] DUCOURNAU R. YAFOOL : manuel de référence. Sema-Matra. 1988.

[DUC89] DUCOURNAU R., HABIB M. La multiplicité de l'héritage dans les langages à objets. TSI. 8,1. Dunod Informatique. 1989.

[DUC92] DUCOURNAU R., et al. L'Héritage Multiple dans tous ses états. Soumis à TSI. 1992.

[DUG86] DUGERDIL P. A propos des mécanismes d'héritage dans un langage orienté objet. 2ème Colloque International d'Intelligence Artificielle. Marseille 1986.

[DUG88] DUGERDIL P. "Contribution à l'étude de la représentation des connaissances fondée sur les objets : le langage OBJLOG". Doctorat. Faculté d'Aix-Marseille II 1988

[ESC89] ESCAMILLA J. Mécanisme d'héritage dans une base de connaissances orientée objets. Rapport de D.E.A. d'informatique, INPG 1989.

[ESC90] ESCAMILLA J., JEAN P. Relations Verticales et Horizontales dans un modèle de représentation de connaissances. Actes Congrès Inforsid 90, Eyrolles Ed., Biarritz, France. Mai 1990.

[ESC90a] ESCAMILLA J., FAVIER V., JEAN P., NGUYEN G.T, RIEU D. Représentation de connaissances dynamiques dans SHERPA. Actes Congrès Inforsid 90, Eyrolles Ed., Biarritz, France. Mai 1990.

[ESC90b] ESCAMILLA J., JEAN P. Relations verticales et horizontales dans un modèle de représentation de connaissances. Actes 4e Journées INRIA Bases de Données Avancées. Montpellier. Septembre 1990.

[ESC90c] ESCAMILLA J.,JEAN P. Relationships in an Object Knowledge Representation Model. Proc. 2nd International Conf. Tools for Artificial Intelligence. Washington (DC). Novembre 1990.

[FAH79] FAHLMAN S.E NETL: A System for Representing and Using Real-World Knowledge. MIT Press. Cambrige, Massachusetts. 1979.

[FAV89] FAVIER V. Représentation et manipulation d'objets dynamiques dans les bases de connaissances. Mémoire d'ingénieur CNAM, INPG 1989.

[FAV90] FAVIER V., RIEU D. Manipulation d'objets dynamiques dans les bases de connaissances. Proc. MICAD '90. Paris. Hermès Ed. Février 1990.

[FER84] FERBER J. Quelques aspects du caractère self réflexif du langage MERING. Proc. 2° JLOO. Brest 1984.

[FER87] FERBER J. Approches Réflexives en Informatique. Proc. de Cognitiva 87. Paris, La Villete. Mai 1984.1987.

[FER88] FERBER J. PtitLoo, Manuel utilisateur. LAFORIA, Université de Paris 6. Paris 1988.

[FIK85] FIKES R., T. Kehler. The role of frame-based representation in reasoning. Communications of the ACM, vol.28, n. 9, 1985.

[FOX84] FOX. M.S., WRIGHT J.M., ADAM D. Experiences with SRL: An Analysis of a Frame-based Knowledge Representation. Expert database systems. September 1984.

[GUI93] GUINET P. Méthodologie de conception d'applications pour Shood. Mémoire de DEA de Génie Industriel. ENSGI de l'INPG. Grenoble 1993.

[GOL83] GOLDBERG A., D. Robson. Smalltalk-80: The language and its implementation. Reading, Massachusetts: Addison-Wesley, 1983.

[GOL84] GOLDBERG A. The Smalltalk-80 system release process. Smalltalk-80, Bits of History, Words of advice. Addison-Wesley. 1984.

[GSI93] Rapport du GSIP. XXX

[IGE81] IGES Initial Graphic Exchange Specification. ANSI Y14.26-M "Digital Representation for Communication of Product Definition Data". 1981

[JAU92] JAULENT P. Génie logiciel: les méthodes. Armand Colin 1992.

[JEA89] JEAN P. Manipulation d'objets composites dans les bases des connaissances. Rapport de D.E.A. d'informatique. ENSIMAG 1989.

[KEE88] KEENE S.E Object-oriented programming in Common Lisp. A programmer's guide to CLOS. Addison-Wesley. 1988.

[KIM87] KIM W., BALLOU N., BANERJEE J., CHOU HT, GARZA J.F, WOELK D. Features of the ORION Object-Oriented Database System. Rapport ACTA-ST-308-87, MCC. Austin (TX). Septembre 1987.

[KIM89] KIM W., BERTINO E., GARZA J. F. Composite objects revisited. SIGMOD 1989 Proceedings Volume 2.

[KOO87] KOOL Version 1.3, Manuel de l'utilisateur. Bull Cediag. 1987.

[LIB81] LIBERMAN H. A preview of Act 1. Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo No. 625, Juin 1981.

[LIB86] LIBERMAN H. Delegation and inheritance : Two Mechanisms for Sharing Knowledge in Object-Oriented Systems. Actes des Journées Afcet-Informatique Langages Orientés Objet Janvier 1986, dans Bigre + Globule No. 48.

[LIB86a] LIBERMAN H. Using Prototypical Objects to Implement Shared Behaivior in Object Oriented Systems. Proceedings of OOPSLA '86, ACM, Septembre 1986.

[LIO93] LIOTARD M.P. Mécanisme de classification pour un système de représentation de connaissances. Mémoire d'ingénieur CNAM en Informatique, centre régional associé de Grenoble. Mars 1993.

[MAE86] MAES P. Introspection in Knowledge Representation. ECAI-86, Brighton (GB)

[MAS89] MASINI G. et al. Les langages à objets. InterEditions, Paris 1989.

[MAR89] MARIÑO O. XXX. Rapport de DEA en Informatique. ENSIMAG-INPG. Juin 1989.

[MAR90] MARIÑO O., RECHENMANN F., UVIETA P. Multiple perspectives and classification mechanisms in object-oriented representations. Proc 9th ECAI. Stockholm, 1990.

[MCG88] McGregor. A Deductif Pattern Matcher. Proc. 7th AAAI. Saint Paul, Minnesota, 1988.

[MEY89] MEYER B. Conception et programmation par objets, pour du logiciel de qualité. InterEditions. Paris. 1989.

[MIL92] MILLASSEAU J.F. Les méthodes sont aussi des objets. Rapport de DEA en Informatique. ENSIMAG-INPG. Juin 1992.

[MIN75] MINSKY M. A framework for representing knowledge. The psychology of computer vision. Edité par Winston P.H. McGraw-Hill. 1975.

[MOO86] MOON D. Object-Oriented programming with Flavors. Proc. 1st OOPSLA Conf. Portland (Oregon). 1986.

[NAP92] NAPOLI A. Représentations à objets et raisonnement par classification en intelligence artificielle. Thèse de doctorat d'état. Université de Nancy 1. Janvier 1992.

[NGU89] NGUYEN G.T., RIEU D. Schema evolution in object-oriented database systems. Data & Knowledge Engineering. North Holland. Vol 4, n°1. Juillet 1989.

[NGU91] NGUYEN G.T, RIEU D. Database issues in Object Oriented Design. Proc. TOOLS '91. Paris. Mars 1991.

[NGU91a] NGUYEN G.T, RIEU D. Representing desing objects. Proc. 1th Intl. Conf. Artificial Intelligence in Desing, AID'91. Edinburgh (U.K.). Juin 1991.

[NGU92] NGUYEN G.T, RIEU D. Multiple object representations. Proc. 20th ACM Computer Science Conference. Kansas City (Missouri). Mars 1992.

[NGU92a] NGUYEN G.T, RIEU D. SHOOD : a design object model. Proc. 2nd Intl. Conf. Artificial Intelligence in Design. Carnegie Mellon University. Pittsburgh (Pennsylvania). Juin 1992.

[NGU92b] NGUYEN G.T, RIEU D. ESCAMILLA J. An object model for engineering design. Proc. 6th European Conf. on Object-Oriented Programming. ECOOP '92. Utrecht (NL). Juin 1992. Egalement Rapport de recherche INRIA n° 1653. Avril 1992.

[NGU92c] NGUYEN G.T, RIEU D. Un modèle à objets pour la conception assistée. Symposium International CNES "La conception en 2000 et au-delà". Strasbourg. Novembre 1992.

[ORS90] ORSIER B. Evolution de l'attachement procédural : intégration de tâches, méthodes et procédures dans une représentation de connaissances par objets. DEA d'informatique. INPG. Juin 1990.

[PAT88] PATON N.W., P. M. D. GRAY. Identification of Database Objects by Key. Advances in Object Oriented Database Systems, Springer-Verlag, 1988.

[PEN87] PENNEY D.J., J. Stein. Class Modification in the GemStone Object-Oriented DBMS. Proceedings of OOPSLA '87, ACM, Octobre 1987.

[PIT90] PITRAT J. Métaconnaissance, futur de l'intelligence artificielle. Hermès, Paris 1990.

[REC85] RECHENMANN F. Shirka: mécanismes d'inférences sur une base de connaissances centrée objets. Actes du 5ème congrès AFCET Reconnaissances des Formes et Intelligence Artificielle, Grenoble 1985.

[REC88] RECHENMANN F. Shirka: un système de gestion de bases de connaissances. Manuel de référence. IMAG. Laboratoire Artémis. juin 1988.

[RIE88] RIEU D., G.T. Nguyen. Dynamic schemas for engineering databases. Workshop Engineering Information Systems, 4th International Conference on Systems Research, Informatics and Cybernetics. Baden-Baden (RFA), juillet 1988.

[RIE90] RIEU D., FAVIER V., DAVID B., JEAN P. SHERPA : un support d'intégration pour le processus CEDF. CIM 90. Bordeaux Juin 1990

[RIE91] RIEU D., G.T NGUYEN. De l'Objet à l'Objet CFAO. Actes MICAD 91. Paris. Février 1991.

[RIE91a] RIEU D., CULET A. Classification et représentation d'objets. Congrès INFORSID 91. Paris. Juin 91.

[RIE91b] RIEU D., NGUYEN G.T., CULET A., ESCAMILLA J., DJERABA C. Représentations multiples d'objets évolutifs. Actes 8° Congrès RFIA 91. Lyon. Novembre 91.

[RIE92] RIEU D., NGUYEN G.T. Object views for engineering databases. Proc. 3rd Intl. Conf. Data & Knowlegde Systems for Manufacturing and Engineering. Lyon. Mars 1992.

[RIE92a] RIEU D., NGUYEN G.T., ESCAMILLA J. Shood: an Object Model for Design Applications. EC2 Journées Internationales d'Avignon. Avignon 1992.

[RIE92b] RIEU D., NGUYEN G.T., ESCAMILLA J. Méthodes et représentation des connaissances. Journées LIRMM-LERI-EC2 "Représentations par objets". La Grande Motte. Juin 1992.

[RIE92c] RIEU D., NGUYEN G.T., ESCAMILLA J. Représentation, sélection et combinaison de méthodes. Actes 8ème Journées bases de Données Avancées. Trégastel. Septembre 1992.

[ROB77] ROBERTS R.B, GOLDSTEIN I.P. The FRL Manual. AI memo 409, AI lab., MIT Cambridge, Massachusetts, 1977.

[ROU88] ROUSSEAU B. Vers un environnement de résolution de problèmes en biométrie, Apport des techniques de l'intelligence artificielle et de l'interaction graphique. Thèse de doctorat de l'université Claude Bernard Lyon I. Février 1988.

[RUM87] RUMBAUGH J. Relations as semantic constructs in an object oriented language. Proc. 2nd OOPSLA. Orlando (USA). Octobre 1987.

[SET91] SET Standard d'Echange et de Transfert. AFNOR Z68-300 version 08-85, et 06-89 révision B 09-91.

[SHO93] BOUNAAS F., DJERABA C., ESCAMILLA J., MILLASSEAU J.F. Interface Programmeur Shood. LGI-IRIMAG, Mars 1993.

[STE85] STEELS L. The KRS Concept System.Virje Universiteit Brussel. Artificial Intelligence Laboratoy. Technical Report 85-4. Bruxelles (BE). 1985.

[STE86] STEFIK M., BOBROW D.G. Object oriented programming: themes and variations. The AI magazine. Janvier 86.

[STE87] STEIN L. A. Delegation is Inheritance. Proceedings of OOPSLA '87, ACM, Octobre 1987.

[STR89] STROUSTRUP B. Le langage C++. InterEditions, Paris. 1989.

[TIE90] TIEMANN M.D. User's Guide to GNU C++, version 1.37.1. Free Software Fondation. Mars, 1990.

[TOL92] TOLLENAERE M. Quel modèle produit pour concevoir? Syposium International La conception en l'an 200 et au-delà: Outils et Technologies. Strasbourg, France. Novembre 1992.

[TOU84] TOURETZKI D.S The mathematics of inheritance systems. PhD Thesis. Carnegie Mellon University. Mai 1984.

[VAN87] VAN LUCK K., NEBEL B., PELTASON C., SCHMIEDEL A. The Anatomy of the BACK System. KIT-Report 41. Université Technique, Berlin. Berlin, 1987.

[VIE92] VIEL C. La modélisation dans STEP. Syposium International La conception en l'an 200 et au-delà: Outils et Technologies. Strasbourg, France. Novembre 1992.

[WEI81] WEINREB D., D. Moon. Lisp Machine Manual. Symbolics Inc, 1981.

[WOO75] WOODS W.A. What's in a Link: Foundations for Semantic Networks. Dans Representation and Understanding: Studies in Cognitive Science. Edité par Bobrow D.G. et Collins A.M. New York, Academic Press. 1975.

[WRI84] WRIGHT J.M, FOX M.S, ADAM D.L SRL User's manual. Carnegie Mellon University, 1984.

Résumé

Cette thèse définit un modèle de représentation de connaissances adapté aux besoins des applications de CAO. Le modèle proposé, appelé Shood, s'inspire de concepts issus des systèmes de représentation de connaissances et des langages orientés objet. Shood répond aux besoins de puissance de représentation (aspects déclaratifs), d'intégration des outils de calcul (aspects procéduraux) et d'évolution des besoins (extensibilité) des applications de CAO. Dans Shood, la sémantique des liens structurant la connaissance (héritage multiple, disjonction et instanciation multiple) est fondée sur des relations ensemblistes. Les attributs sont définis par des descripteurs qui peuvent être enrichis. Les descripteurs de base permettent de typer les attributs, de leur associer une méthode de calcul automatique ("inférence") et de restreindre leurs valeurs de façon procédurale ("contrainte"). Les méthodes Shood sont basées sur le concept de fonction générique. Un mécanisme de classification permet de choisir avec précision la méthode la plus adaptée en fonction des valeurs des arguments en entrée. Les méthodes peuvent être réutilisées grâce aux mécanismes de spécialisation déclarative et procédurale. L'extension du modèle est possible grâce à sa définition réflexive: les concepts de Shood sont définis en termes d'eux-mêmes grâce à un niveau méta.

Mots clefs: Représentation de connaissances, modèle orienté objet réflexif, modèle pour la CAO, extensibilité, niveau méta, fontion générique, héritage multiple, instanciation multiple

Abstract

This thesis defines Shood, a knowledge representation model suited for CAD applications. Knowledge Representation Systems and Object Oriented Languages are the inspiration sources of the Shood model. Shood meets three kinds of CAD application requirements: representation power (declarative aspects), integration of computation tools (procedural aspects) and requirement evolution (extensibility). The semantics of its knowledge structuring links (multiple inheritance, disjonction and multiple instantiation) is based on set relationships. Attributs are defined by an extensible set of descriptors. The initial set of descriptors allows the definition of typed attributs, methods for automatic computing attribut values ("inferences") and procedural constraints on attribut values ("constraints"). Shood methods allow the definition of generic fonctions. Input arguments are used by a classification mecanism to find the best suited method to execute. Methods can be reused in a declarative or procedural style. Extensibility is achieved by the reflexive definition of the model: Shood concepts are defined in terms of Shood concepts themselves using meta-classes.

Keywords: Knowledge representation, Reflexive object oriented model, CAD model, extensibility, meta level, generic fonction, multiple inheritance, multiple instantiation.