THESE de DOCTORAT de l'UNIVERSITE PARIS 6

Spécialité :
Informatique

Présentée par
Gerson SUNYÉ

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITE PARIS 6

Sujet de la thèse :
Mise en œuvre de patterns de conception : un outil

soutenue le mardi 6 juillet 1999

devant le jury composé de :

M. Jean François PERROT	Directeur de thèse
M. Jean-Marc JÉZÉQUEL	Rapporteur
Mme Dominique RIEU	Rapporteur
M. Mehmet AKSIT	Examinateur
M. Gilles BLAIN	Examinateur
M. Etienne MOREL	Examinateur

Remerciements

En premier lieu, je tiens à remercier *Jean-François Perrot* d'avoir accepté de diriger cette thèse. Ses remarques pertinentes et éclairées m'ont permis de mieux structurer mes idées et, je l'espère, de mieux les décrire.

Je remercie également *Gilles Blain* pour l'accueil chaleureux qu'il m'a réservé dans son équipe, pour sa disponibilité, son dynamisme et sa grande créativité. Les nombreuses discussions que nous avons eues ensemble m'ont permis de mieux comprendre la méta-modélisation et d'enrichir ma culture informatique.

Je remercie Mme *Dominique Rieu* qui a accepté d'être mon rapporteur. Sa double compétence dans le domaine de la méta-modélisation et des patterns de conception fait d'elle un juge incontournable pour mes travaux.

Je remercie également M. *Jean-Marc Jézéquel* d'avoir accepté d'être lui aussi mon rapporteur. Son œil pragmatique et sa compétence dans le domaine des langages à objets lui ont permis de faire des critiques importantes et pertinentes sur mon travail, chose que j'ai beaucoup appréciée.

Je suis très honoré que M. *Mehmet Aksit* ait accepté de juger mon travail. J'apprécie beaucoup ses idées sur le développement par aspects et la conception logicielle en générale.

Je remercie M. *Etienne Morel* qui a également accepté de juger mon travail, malgré un emploi du temps très chargé.

Cette thèse doit beaucoup à mes collègues de l'équipe MétaFor qui ont permis au « nous », employé tout au long de cette thèse, d'être plus qu'un simple style de langage. Je remercie particulièrement *Bruno Lesueur* pour les longs moments que nous avons partagé durant le développement de MétaGen, *Houari Sahraoui* qui m'a fait connaître la méta-modélisation et guidé durant le début de mon doctorat et *Nicolas Revault*, qui m'a présenté les patterns de conception ainsi que les cadres d'application. Je n'oublie pas également *Mikal Ziane*, qui a su trouver le temps de corriger les différents chapitres de cette thèse et m'aider à les améliorer, ainsi que pour nos nombreux échanges.

Au cours de ces quatre années de thèse, j'ai eu le plaisir de partager salles, machines, idées et passions avec l'ensemble des thésards du LIP6, qui ont donné des couleurs à mon doctorat. Je pense particulièrement à ma collègue portoalegrense *Nara Martini Bigolin* pour le soutien qu'elle m'a témoigné au long de ces années ainsi qu'au montpelliérain *Christophe Marsala* qui, en galant ami, a toujours su trouver les mots pour m'encourager.

Je pense aussi à *Amel* (et sa bonne humeur perpétuelle), *Anne* (et ses passionnantes conversations avec ses machines), *Bruno*, *Fabrice*, *Geber* (l'autre brésilien), *Ilham*, *Isabelle* (et ses cheveux blonds oubliés sur mes chemises), *Maria* (et sa passion pour le foot), *Mohamed*, *Nathalie* (et ses petits éléphants), *Sergios*, *Tristan* ainsi qu'à tous les autres que j'ai sûrement eu l'indélicatesse d'oublier.

Ces dernières années ont été plus faciles à supporter grâce à l'amitié de *Fred*, *Inès*, *Marie* et *Rémi*. Je m'excuse auprès d'eux de mon indisponibilité de ces derniers mois et je les remercie de leurs encouragements.

Je remercie finalement mes parents qui, après quatre docteurs et un GMI, continuent à penser qu'ils n'ont fait que « leur obligation ».

Enfin, mon doctorat n'aurait pas vu la fin sans mon épouse Fabienne qui m'a encouragé et a crû en moi, quand moi-même je doutais.

A ma chère Fabienne,
pour son amour et son soutien

Table des Matières

Introduction

Les patterns de conception, en caractérisant des problèmes récurrents de conception et en spécifiant des solutions claires et élégantes à ces problèmes, représentent le vocabulaire commun de l'expertise des concepteurs de logiciels, [Coad, 1992 ; Coplien et Schmidt, 1995 ; Gamma *et al.*, 1995]. Ce vocabulaire leur fournit un niveau de description adéquat pour discuter les choix de conception ou de restructuration d'un système, au-delà des détails connus de conception. De plus, l'explicitation des patterns présents dans un système simplifie la compréhension de sa structure (voir par exemple [Jaczynski, 1998] et [Johnson, 1992]).

Ils apportent au développement au moins deux bénéfices. D'une part, ils rendent accessibles les connaissances des informaticiens plus expérimentés aux concepteurs novices. Grâce à ces connaissances ceux-ci peuvent identifier une situation où l'utilisation suggérée par un pattern de conception serait bénéfique.

D'autre part, ils guident les implémenteurs dans le choix de la mise en œuvre la plus adaptée à cette solution. En effet, un pattern regroupe sous une étiquette unique une variété de situations implémentatoires qui ont une logique de fonctionnement commune. Sa mise en œuvre est pilotée par différents compromis, et peut adopter différentes formes ou variations, sans pour autant déroger à la solution de conception qu'il définit.

La variation la plus évidente est celle qu'induit le contexte applicatif sur les noms des composants qui interviennent pour réaliser le pattern. Au-delà de cette « α-conversion », les choix de structures de données (e.g., pour représenter des collections : listes, tableaux, etc.) et l'organisation même des classes peuvent résulter de compromis liés au problème traité (voir pour plus de détails la partie « Implémentation » du canevas de description de patterns proposé par Gamma et al. [Gamma *et al.*, 1993]). Ceci pose évidemment un problème : il faut pour réutiliser vraiment un pattern aller plus loin que de la réutilisation directe de code exécutable [Budinsky *et al.*, 1996].

Les variantes d'implémentation constituent ainsi le principal intérêt et la principale difficulté de l'utilisation des patterns. Elles sont nécessaires car les patterns représentent les connaissances de conception et l'importance de celle-ci est d'être indépendante de l'implémentation. Cependant, elles rendent difficile l'implémentation et sa réutilisation. L'implémenteur est contraint de connaître toutes les variantes du pattern qu'il souhaite utiliser, leurs avantages et leurs limitations.

Certains travaux de recherche [Meijers, 1996 ; Soukup, 1995] proposent des techniques d'implémentation facilitant la réutilisation des patterns. En général, ces outils ont pour principal avantage d'améliorer la traçabilité de la présence de patterns dans les logiciels réalisés. Pour ce faire, ils imposent pour chacun une réalisation particulière (par exemple sous la forme d'une classe). Ils ne prennent donc pas en compte la pluralité des « variantes d'implémentation » qui fait la richesse de la notion de pattern, ce qui limite fortement leur champ d'application.

Dans cette thèse, nous présentons PatternGen, un prototype d'outil de génération automatique de code à l'aide de patterns de conception qui prend en compte les variantes d'implémentation. Pour passer de la description d'un pattern à son implémentation effective dans une application, PatternGen permet de définir des compromis d'implémentation spécifiques à chaque implémentation d'un pattern.

L'objectif de PatternGen est d'aider le concepteur à développer son application. Pour cela, il propose un éditeur de diagrammes permettant la construction d'un modèle de conception où les choix d'utilisation de patterns seront explicitement définis. Ce modèle est ensuite analysé par une base de règles, représentant les connaissances d'implémentation, qui génère le code exécutable correspondant. PatternGen adopte une approche qui ne réifie pas les patterns dans le code, mais qui génère la solution spécifique à chaque utilisation d'un pattern.

Cette approche a été vérifiée durant la réalisation de deux exemples d'utilisation, qui mettent en évidence deux approches différentes d'utilisation de patterns : la conception d'une nouvelle application et la modification d'une application existante.

L'implémentation d'un outil de mise en œuvre de patterns tel que PatternGen est une tâche complexe et coûteuse. Outre une représentation formelle des patterns, l'outil doit permettre l'utilisation de patterns durant la conception et l'implémentation d'une application. Pour ce faire, l'outil emprunte différentes fonctionnalités aux outils d'aide au développement, comme l'édition de diagrammes conceptuels (pour l'utilisation des patterns) et la génération de programmes (pour les implémenter). En d'autres termes, l'outil doit être lui-même un outil de développement.

Il existe actuellement plusieurs projets de recherche dont l'objectif est de créer un « méta-outil », i.e. outil capable de construire presque automatiquement des outils de développement. Un de ces projets est mené au sein de l'équipe de recherche, MétaFor, appartenant au thème OASIS, du Laboratoire d'Informatique de Paris VI (LIP6), où notre travail de thèse a été réalisé. Cette équipe maintient un prototype de méta-outil, appelé MétaGen. Ce dernier a été utilisé dans la mise en œuvre de différents environnements de développement, dont PatternGen.

L'emploi d'un méta-outil dans la mise en œuvre de PatternGen apporte deux avantages majeurs. Premièrement, une partie importante de notre outil, comme l'éditeur de diagrammes et les dialogues d'édition, a été développée rapidement. Deuxièmement, notre outil est évolutif. Il peut intégrer de nouveaux patterns au cours du développement d'une application.

L'objectif primordial de MétaGen est de supporter une démarche de développement basée sur des techniques empruntés à l'analyse du domaine et à la méta-modélisation. L'objectif de cette démarche est de permettre aux différents acteurs du processus de développement d'assumer pleinement leurs responsabilités. Cet objectif se traduit par la construction d'outils permettant de représenter indépendamment le point de vue des utilisateurs et de celui des informaticiens. Ces représentations sont exprimées dans différents langages de modélisation et sont liées aux domaines qu'elles représentent. La démarche est complétée par la traduction du langage des utilisateurs en celui des informaticiens et ensuite, en un langage compréhensible par la machine.

Cette traduction se fait grâce aux diverses connaissances présentes durant le développement d'une application, comme celles des experts métiers, des concepteurs ou des implémenteurs. Certaines de ces connaissances et plus spécifiquement celles concernant des problèmes précis de conception, sont regroupées sous un label unique et décrites suivant un canevas commun : les patterns de conception.

L'objectif secondaire de cette thèse est de proposer une meilleure intégration des patterns de conception à la démarche de développement de l'équipe MétaFor. Cette intégration a été réalisée grâce à PatternGen et aux facilités de génération de code qu'il propose.

Notre recherche nous a aussi permis de proposer et réaliser différentes modifications dans MétaGen, qui ont engendré le développement d'une nouvelle version de l'outil.

I. Plan de la thèse

Cette thèse est organisée en cinq chapitres. Le premier chapitre est consacré à l'état de l'art dans le domaine des patterns de conception. Le deuxième chapitre introduit PatternGen par des exemples d'utilisation. Le troisième chapitre est dédié à MétaGen et à son développement. Le quatrième chapitre présente l'implémentation de PatternGen grâce à MétaGen. Le cinquième chapitre présente la démarche de développement de l'équipe MétaFor et l'intégration de PatternGen à celle-ci. Finalement, la conclusion et les perspectives futures de nos travaux closent ce mémoire.

Dans le premier chapitre nous présentons les techniques et outils de mise en œuvre de patterns de conception. Cette présentation commence par un rappel de l'historique et des motivations de la notion de pattern. Ensuite, quelques exemples de patterns sont détaillés. Les difficultés principales de la mise en œuvre de patterns et une synthèse des principaux outils et techniques de mise en œuvre seront exposés.

Nous pensons que la manière la plus simple de présenter PatternGen est de commencer par des exemples concrets d'utilisation. C'est pourquoi le deuxième chapitre est dédié à la présentation de deux exemples de développement avec PatternGen.

Il serait difficile de dissocier PatternGen de MétaGen, l'outil employé pour le développer. Le troisième chapitre donc est consacrée à une description de l'architecture MétaGen et de l'implémentation de la version actuelle, à laquelle nous avons collaborée à la réalisation. La description commence par une introduction à la méta-modélisation dans le génie logiciel et par une synthèse des principaux travaux dans ce domaine. Ensuite, nous présentons les principales fonctionnalités de l'outil, illustrées par un exemple simple. Nous présentons, par ailleurs, le développement de la version actuelle de l'outil, en mettant en évidence deux aspects importants : l'intégration de divers cadres d'applications et l'utilisation de différents patterns de conception.

Dans le quatrième chapitre, nous présentons PatternGen. Nous détaillons son architecture et la comparons à celle des outils présentés dans le premier chapitre. Ensuite, nous illustrons l'utilisation de notre outil avec quatre exemples de patterns de conception, avant de détailler son implémentation. Nous concluons en expliquant comment un tel outil peut atténuer les difficultés exposées au premier chapitre.

Les principes la démarche de développement de l'équipe MétaFor, ainsi que notre contribution à cette démarche seront présentés dans le dernier chapitre. Cette présentation est précédée par une synthèse de plusieurs projets de recherche dans ce même domaine. Ensuite, nous expliquons comment sont traités des problèmes de communication entre les acteurs participant à un même projet de développement, par une technique qui combine de techniques de méta-modélisation et de l'analyse du domaine. Nous montrons de quelle manière notre approche permet d'offrir des bases pour supporter cette communication avec notamment changement de point de vue et de niveau de connaissances durant le développement d'une application. Cette présentation sera illustrée par un exemple d'utilisation de notre démarche dans le développement d'un éditeur de diagrammes de flots de données. Le chapitre se conclut par l'évaluation de notre démarche et par un exemple d'intégration de PatternGen à celle-ci.

II. Terminologie

La terminologie dans le domaine du génie logiciel est quelquefois confuse, ce qui nous motive à expliciter, par la suite, certains termes qui seront employés au long de ce mémoire. La liste de termes présentée ici n'est pas exhaustive, d'autres termes seront définis lors de leur apparition.

Tout d'abord, il n'existe pas de consensus sur la traduction appropriée du terme (*design*) *pattern*. Malgré les diverses propositions (e.g. patron, gabarit, schéma, modèle, micro-architecture), aucune traduction ne semble avoir la préférence de la communauté informatique francophone. Nous conserverons donc le terme anglais « pattern ».

En revanche, « cadre d'applications » semble s'affirmer comme la traduction du terme anglais *application framework*, en dépit d'autres propositions (e.g. cadriciel, canevas).

Le terme « méthode » (de développement de systèmes informatiques), est source de désaccord dans la communauté de recherche en génie logiciel. Une partie de cette communauté l'associe à une suite d'étapes qui culminent par la construction d'un diagramme, représentant un aspect du système, tandis qu'une autre partie l'associe aux étapes permettant la représentation de la totalité du système par un ensemble de diagrammes reliés entre eux.

Le terme « méthodologie » est aussi cause de désaccord, certains définissent une méthodologie comme un ensemble de méthodes [e.g. Kulmar et Welke, 1992], alors que d'autres [e.g. Harmsem *et al.*, 1994] conservent le sens étymologique du terme (i.e. étude de méthodes). Au long de ce mémoire nous employons le terme « méthode » selon la définition de [Smolander, 1991], i.e., un ensemble de règles et d'étapes à suivre pour réaliser un système informatique.

Une méthode peut être automatisée par un « environnement d'automatisation du processus de développement de systèmes informatiques » (ou simplement « environnement de développement logiciel »). M. Soberman [Soberman, 1992] définit ces environnements comme un « ensemble de composants logiciels organisés pour couvrir une partie du cycle de production et d'évolution logiciels ». Un environnement de développement logiciel est composé de différents « outils » de développement logiciel (en anglais CASE *tool*).

Une méthode dispose habituellement d'un ou de plusieurs « langages de modélisation », permettant de représenter différents aspects (statiques, dynamiques, etc.), de différentes phases (analyse, conception, etc.) du développement d'un système d'information. Un langage de modélisation peut avoir plusieurs « notations », comme un diagramme un tableau ou un texte.

Un « modèle » est la représentation, dans un langage de modélisation, d'un aspect d'un système d'information. Cette représentation est fréquemment réalisée grâce à un outil de développement qui outille ce langage. Par « outiller » un langage, nous entendons que l'outil contrôle la syntaxe du langage.

La description d'un langage de modélisation dans un autre langage de modélisation, dans le but de l'outiller ou de l'étudier, est appelée « méta-modélisation » [Brinkkemper, 1990 ; Premerlani, 1995 ; Verhoef et Hofstede, 1995]. Le langage utilisé pour réaliser la description est appelé métalangage ou « langage de méta-modélisation ». En analogie avec le terme métalangage, un langage de méta-modélisation est un langage qui sert à décrire, étudier ou analyser d'autres langages de modélisation.

III. Démarche suivie dans cette thèse

Avant de passer au premier chapitre de cette thèse, nous faisons un aparté pour présenter notre démarche de recherche. Cette démarche est conforme au cadre de recherche proposé par Nunamaker, Chen et Purdin [Nunamaker, 1992 ; Nunamaker *et al.*, 1991], qui est par ailleurs particulièrement applicable aux recherches menées dans le domaine du génie logiciel. La démarche suivie comprend trois cycles distincts qui seront détaillés ci-dessous. La proposition de Nunamaker et al. est présentée dans l'Annexe A de ce document.

Le premier cycle commence par l'analyse de la première version de MétaGen [Revault *et al.*, 1995] et par celle de différents outils similaires à MétaGen. L'observation d'une vaste gamme de méta-outils et de plusieurs analyses comparatives [comme Karrer et Scacchi, 1995 ; Kelly et Smolander, 1994] nous a permis de faire des propositions pour modifier MétaGen.

Ces observations ont été prises en compte lors du développement de la version actuelle (2.0) de MétaGen [Lesueur *et al.*, 1998]. Ce développement clôt le premier cycle de recherche et en relance un deuxième.

Le deuxième cycle commence par l'observation de différents projets de recherche concernés par l'analyse du domaine.

La démarche de développement de l'équipe MétaFor, basée sur la double méta-modélisation et son expérimentation dans la conception de bases de données [Sahraoui *et al.*, 1995] ainsi que sur l'utilisation de cadres d'applications [Revault, 1996] a servi de base théorique à ce deuxième cycle de recherche.

Grâce à cette base théorique, nous avons réalisé des expériences de développement d'applications à l'aide de cadres d'applications tels que HotDraw [Johnson, 1992] et Accounts [Keefer, 1994]. Ces deux expériences nous ont permis d'effectuer plusieurs observations sur notre démarche.

Ces observations ont donné origine à une première théorie : le nombre prédéfini (deux) de points de vue durant le processus de développement (utilisateur et informaticiens) imposé par MétaFor est une contrainte trop rigide, car d'autres points de vue sont présents pendant le processus de développement logiciel.

Un de ces points de vue, celui des implémenteurs, était parfaitement explicité par les patterns de conception. La vérification de cette théorie était dépendante d'une plus grande connaissance concernant les patterns de conception, ce qui a lancé un troisième cycle de recherche.

Ce troisième cycle commence par l'observation de différents outils et techniques d'implémentation de patterns de conception, ce qui nous a permis de dégager les caractéristiques souhaitables dans un tel outil.

Cette observation nous a permis de formuler une théorie : les méta-outils semblent être adaptés au développement d'outils de mise en œuvre de patterns. La présence, dans les méta-outils, d'un niveau d'abstraction supplémentaire, ou « méta », paraissait parfaitement adaptable aux deux facettes d'un pattern de conception, sa description abstraite et sa mise en œuvre concrète.

Cette théorie a été vérifiée par le développement, à l'aide de MétaGen, de PatternGen, un outil de mise en œuvre de patterns de conception. PatternGen est présenté dans le Chapitre 4.

Finalement, nous avons utilisé PatternGen pour concevoir et implémenter des applications qui utilisent des patterns de conception. Les expériences d'utilisation de PatternGen closent ce troisième cycle. Elles permettent aussi de clore le deuxième cycle (resté en suspens) en démontrant que d'autres points de vue (dans ce cas, celui des concepteurs), peuvent être pris en considération lors du développement d'une application.

La Figure 1 illustre notre démarche de recherche. Dans ce diagramme, les rectangles creux représentent les observations, les cercles représentent le développement et les rectangles pleins représentent les résultats.

Premièrement, l'étude de différents méta-outils nous a permis d'évaluer MétaGen et de préconiser son évolution. Ces idées ont été mises en œuvre lors du développement d'une nouvelle version de MétaGen. Celle-ci a fait appel à des technologies des langages à objets, comme les patterns de conception et les cadres d'applications.

L'expérience dans le développement avec des cadres d'applications nous a permis de les utiliser dans des expériences utilisation de MétaGen comme outillage support à la démarche de développement (dite de double méta-modélisation) de l'équipe MétaFor.

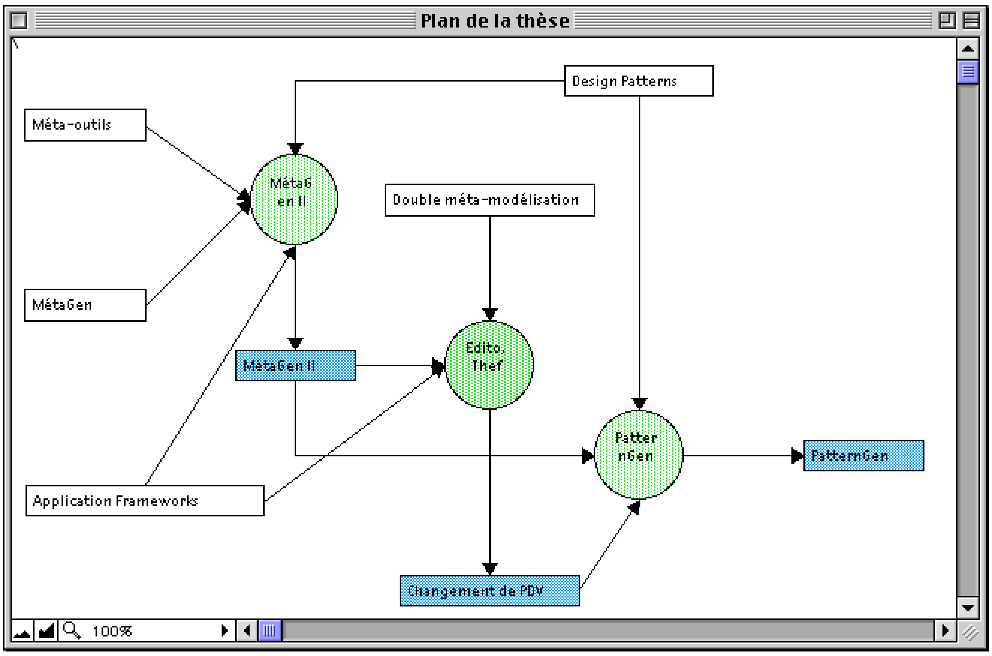

Figure 1 – Démarche suivie dans cette thèse

Ces expériences nous ont servi de base d'étude et d'évaluation de notre démarche de développement, et nous ont permis de proposer différentes évolutions. Une de ces propositions consistait à rendre souple le nombre de transformations entre points de vue (figé alors à deux : utilisateur et implémenteur), et de l'adapter aux besoins de chaque développement. Nous parlions alors de « changements multiples de point de vue ».

Cette proposition nous a motivé à considérer les patterns de conception (déjà utilisés dans le développement de MétaGen) comme une base à la modélisation des connaissances des concepteurs d'applications à objets. La mise en œuvre de ces considérations a donné origine à PatternGen.

Chapitre 1

Techniques et outils de mise en œuvre de patterns de conception

I. Introduction

En réponse à une question sur l'intégration des patterns de conception aux outils CASE, James Coplien a été catégorique : « Les patterns ne doivent pas, ne peuvent pas et ne vont pas remplacer les programmeurs » [Coplien, 1996] (page 30). Ceci car l'application d'un pattern comprend des connaissances diverses, des concepteurs et des implémenteurs, qui sont difficilement exprimables à l'aide d'un outil de développement quelconque.

Dans ce chapitre, nous analysons différents outils visant l'automatisation de la mise en œuvre de patterns de conception. En accord avec le point de vue de Coplien, l'objectif de ces outils n'est pas d'utiliser les patterns pour remplacer les concepteurs, mais d'aider ces derniers, en leur simplifiant l'accès à la description des patterns et en générant automatiquement l'implémentation des patterns choisis.

Comme les outils d'implémentation de patterns, ainsi que le concept même de pattern, sont récents, il existe peu d'efforts qui les analysent et qui décrivent les caractéristiques souhaitables des outils de mise en œuvre de patterns. Cependant, l'analyse des outils présentés ici nous sert de base pour énumérer cinq objectifs poursuivis actuellement par la recherche dans ce domaine, qui seront détaillés par la suite :

1. La gestion de patterns ;
2. La représentation d'instances de patterns ;
3. Le support aux variantes d'implémentation ;
4. La documentation de l'implémentation d'instances de patterns ;
5. La reconnaissance de patterns.

Le concept de pattern, ainsi que l'idée de les cataloguer sont très récents. La nomenclature actuelle des patterns existants est loin d'être exhaustive, de nouveaux patterns de conception apparaîtront ainsi que des nouvelles variantes d'implémentation de ceux déjà existants. L'outil doit alors permettre la gestion de patterns, acceptant l'ajout de nouveaux patterns et la modification des patterns existants.

Pour permettre la gestion de patterns, un outil doit fournir un moyen de représenter ces patterns. Cette représentation concerne au moins deux informations essentielles : (i) la description du pattern et (ii) son implémentation.

Bien que l'hypertexte semble être une bonne forme de représentation de la description d'un pattern [Budinsky *et al.*, 1996], il n'existe pas de consensus sur comment représenter les informations liées à l'implémentation. Quelques propositions ont été faites sur un langage de méta-modélisation [Smolárová *et al.*, 1998], les méta-patterns [Pagel et Winter, 1996] ou les langages formels [Eden *et al.*, 1997].

L'intégration des patterns dans la conception d'une application passe indubitablement par la représentation d'instances de patterns au sein d'un modèle de conception, à travers l'explicitation des classes participantes [Kim et Benner, 1995]. Cette représentation peut être guidée, soit par l'accès simplifié aux informations descriptives, soit par la suggestion de changements structurels. Bien qu'intéressante, cette option n'est abordée par aucun outil.

Dans la mesure où un pattern possède différentes variantes d'implémentation, il est important que l'outil dispose de la description de chacune d'entre elles. Il doit, à partir de ces descriptions, pouvoir générer le code intégrant le mieux les propriétés du pattern choisi et les contraintes propres au contexte. Ceci implique l'interprétation des choix d'implémentation, avant la génération du code, comme le fait le module Mapper de l'outil de Budinsky et al. [Budinsky *et al.*, 1996]. En raison des difficultés de la maîtrise de variantes d'implémentation (voir § IV), peu d'outils s'intéressent à leur implémentation.

La documentation des instances concrètes de patterns ou leur marquage constituent une voie peu explorée pour éviter que ces instances disparaissent dans le code. Une autre solution, celle de la réification de patterns dans le code [Soukup, 1995], semble aller à l'encontre d'une des caractéristiques importantes des patterns, celle de l'élégance des solutions d'implémentation proposées.

Finalement, la reconnaissance d'instances concrètes de patterns est une importante propriété complémentaire, encore peu exploitée, qui peut aider la documentation et l'étude d'applications existantes.

Ce chapitre est organisé comme suit. Après la présente introduction, nous présentons les patterns et leur utilisation dans le développement d'applications. Ensuite, nous présentons trois patterns qui serviront d'exemples à la section suivante, qui souligne les difficultés liées à la mise en œuvre de patterns. Dans la dernière section, nous énumérons des outils et des techniques qui s'intéressent à la mise en œuvre de patterns.

II. Les patterns de conception

Dans les années soixante-dix, l'architecte Christopher Alexander et quelques collègues, en réaction aux idées en vogue à l'époque, ont publié une trilogie [Alexander, 1979 ; Alexander *et al.*, 1975 ; Alexander *et al.*, 1977] mettant en question les techniques contemporaines de construction. Selon eux, les architectes n'étaient pas capables de satisfaire correctement les besoins individuels, sociaux et écologiques. L'idée centrale de ces livres était que les futurs habitants devaient concevoir eux-mêmes leurs propres maisons, rues et communes. La question était alors : « comment permettre à des gens sans compétences spécialisées de concevoir de telles œuvres sans l'aide d'architectes ? ».

Les auteurs ont remarqué que certaines solutions de conception, considérées comme bonnes, étaient récurrentes et s'appliquaient dans différentes situations. Ils ont recensé 253 solutions récurrentes et les ont décrites selon un même « canevas », pour des raisons de commodité et de clarté [Alexander *et al.*, 1977]. Ce canevas se compose essentiellement d'un ensemble de descriptions concernant un problème et sa solution, appelée « Pattern ». La combinaison de plusieurs patterns a été appelée « Langage de patterns ». Ce langage de patterns est conçu comme un outil pour élaborer la réponse à la question précédente.

Grâce à un tel langage, les gens pourraient concevoir leurs maisons, sans l'aide d'un spécialiste. Les auteurs attendaient une révolution des pratiques de leur métier grâce aux langages de patterns qui « tracent les bases d'une approche totalement nouvelle qui remplacera complètement les idées et les pratiques existantes ». La réponse à cette attente n'est pas venue de la communauté des architectes, mais d'une autre communauté plutôt inattendue : les concepteurs d'applications logicielles à objets.

1. Les patterns et le développement à objets

L'idée de l'application des patterns à la conception d'applications à objets est venue d'une hostilité partagée : Kent Beck a remarqué que tout ce qu'Alexander [Alexander, 1979] n'appréciait pas chez les architectes, lui-même ne l'appréciait pas chez les ingénieurs de systèmes [Coplien, 1996] (page 45).

Cette idée prend forme lorsque Beck rencontre Erich Gamma à ECOOP'90 qui s'est déroulé à Ottawa. Ce dernier avait remarqué l'existence de certaines structures récurrentes lors du développement du cadre d'applications ET++ [Weinand *et al.*, 1988 ; Weinand *et al.*, 1989] durant son doctorat, mais ne savait pas comment les partager avec d'autres développeurs. Trois ans après, à Kaiserslautern (Allemagne) dans ECOOP'93, Gamma, Richard Helm, Ralph Johnson et John Vlissides présentent la première esquisse d'un catalogue de patterns de conception à objets [Gamma *et al.*, 1993].

Le catalogue complet [Gamma *et al.*, 1995] a été publié deux ans plus tard. Il est composé de 23 patterns de conception, extraits de plusieurs cadres d'applications, comme ET++, HotDraw [Johnson, 1992], Unidraw [Vlissides et Linton, 1990], Interviews [Linton *et al.*, 1989], MacApp [Apple, 1989], NextStep [Next, 1994], Graphics [Vlissides et Linton, 1988], RTL [Johnson *et al.*, 1992], ObjectWindows [Borland, 1994] et ObjectWorks [ParcPlace, 1990]. Les patterns décrits dans ce catalogue diffèrent de plusieurs manières des patterns proposés en architecture.

D'une part, le catalogue n'a pas pour but de permettre à un utilisateur de concevoir sa propre application. L'objectif des patterns qu'il décrit est de permettre aux développeurs expérimentés de partager leurs expériences avec d'autres développeurs. Plus précisément, ils constituent une base de réutilisation de solutions de conception d'applications.

D'autre part, et comme le signalent les auteurs, ce catalogue ne forme pas (encore) un vrai langage de patterns tel que l'entendait Alexander : il est impossible de réaliser la totalité d'une application avec ces 23 patterns. Contrairement à l'architecture, le génie logiciel connaît encore peu de solutions reconnues par toute sa communauté. Toutefois, ce catalogue constitue un pas important vers la réutilisation des connaissances de conception en proposant, en plus des patterns, un canevas permettant la description de façon analogue, des nouveaux patterns de conception.

En publiant ce catalogue, les auteurs ne proposaient pas une approche totalement nouvelle de développement d'applications, comme le faisaient Alexander et al. en architecture, mais une manière de compléter les approches classiques de développement.

Leur proposition est toutefois convaincante : la communauté de recherche en génie logiciel a très bien reçu les patterns, qui ont été, depuis, intégrés à des méthodes de développement. De plus, plusieurs publications, comme [Buschmann *et al.*, 1996 ; Coplien et Schmidt, 1995 ; Martin *et al.*, 1998 ; Pree, 1995 ; Vlissides *et al.*, 1996] présentent de nouveaux patterns de conception, des patterns de code [Beck, 1996] ou encore, des patterns (ou de langages de patterns [e.g. Aarsten *et al.*, 1995]) spécifiques à un domaine d'application (aussi appelés d'analyse) [Coad *et al.*, 1995 ; Fowler, 1997] On parle alors de « software patterns », en opposition à « design patterns » pour désigner les patterns présents durant le processus de développement d'une application.

2. Patterns de conception à objets : variantes d'implémentation, instances de patterns

Les patterns de conception caractérisent des problèmes récurrents de conception et spécifient des solutions claires et élégantes à ces problèmes. Ils sont indépendants du domaine (i.e. de l'analyse) et du langage d'implémentation. Cependant, les efforts actuels de cataloguer les patterns ne concernent que le développement en langage à objets.

La description des patterns de conception est structurée par un canevas, de manière similaire à celles des patterns architecturaux. Actuellement, il existe trois canevas de description, proposés par Gamma et al. [Gamma *et al.*, 1995], Ward Cunnigham [Cunnigham, 1995] et James Coplien [Coplien, 1995]. Le plus répandu, celui de Gamma et al, décrit un pattern de conception selon 13 aspects :

Nom. Un label significatif qui traduit le principe du pattern.
Intention. Le problème de conception concerné.
Alias. Autres noms connus, s'il en existe.
Motivation. Un exemple d'application du problème de conception en cause et la solution proposée.
Indications d'utilisation. Le contexte dans lequel le pattern peut être appliqué.
Structure. Le diagramme de classes, suivant le formalisme OMT, représentant la solution.
Participants. Le rôle des différentes classes qui composent le diagramme.
Collaborations. La collaboration entre les participants.

Conséquences. Compromis et bénéfices de son utilisation.
Implémentation. Astuces et techniques pour l'implémenter.
Exemples de code. Extraits de code exemple en Smalltalk et C++.
Utilisations connues. Références à des systèmes réels (au moins deux) où le pattern est utilisé.
Patterns analogues. Les patterns en relation étroite avec celui-ci, leurs différences, la manière de les associer.

L'identification d'un pattern par une étiquette unique est sans doute la caractéristique la plus importante des patterns. Grâce à cette identification, un catalogue de patterns peut constituer un vocabulaire commun dénotant l'expertise des concepteurs de logiciels. Ce vocabulaire leur fournit un niveau de description adéquat pour discuter les choix de conception ou de restructuration d'un système, au-delà des détails connus de conception. De plus, ce vocabulaire peut aider à la compréhension rapide d'une application existante, sans rentrer dans les détails spécifiques d'implémentation.

Une autre caractéristique importante des patterns concerne l'implémentation de la solution. Les patterns décrivent des solutions de conception. Leur réalisation dépend de différents facteurs, comme le langage d'implémentation, l'expérience de l'implémenteur et les compromis d'implémentation. Ces derniers, décrits par l'aspect « Implémentation », peuvent modifier plusieurs aspects de l'implémentation, en suggérant, par exemple, la création d'une classe abstraite, l'utilisation d'une structure de données différente (liste, tableaux, etc.) ou l'utilisation d'un deuxième pattern.

Ainsi, un pattern peut être implémenté de différentes manières, appelées les variantes d'implémentation. Bien qu'implémentées différemment, les variantes respectent certaines contraintes implicites du pattern, qui assurent que chaque variante respecte la solution décrite par le pattern.

La variété des implémentations d'un seul pattern impose une distinction entre les termes « pattern » et « instance de pattern », qui sera utilisée tout au long de cette thèse. Un pattern décrit une solution abstraite à un problème de conception. Il s'énonce nécessairement en termes génériques, mettant en évidence les noms des classes qui composent un pattern, et les relations qu'elles entretiennent. Il faudra ensuite particulariser ce discours à plusieurs niveaux, avec des noms de classes spécifiques au contexte de réalisation et en tenant compte de plusieurs compromis d'implémentation, pour obtenir une réalisation concrète, instance de ce pattern.

3. Utilisation de patterns de conception

Nous détaillons ici le canevas de description de patterns présenté ci-dessus, avant de revenir à l'implémentation de patterns à la section IV. Un aspect intéressant de ce canevas, c'est qu'il peut être utilisé de différentes manières. Selon Marco Meijers [Meijers, 1996] la manière la plus simple de comprendre un pattern est d'avoir une « impression de déjà vu », i.e. de reconnaître le contexte du problème concerné par ce pattern. L'aspect « Motivation » d'un pattern offre un premier aperçu du contexte de chaque pattern. Par exemple, cet aspect du pattern **Itérateur** du catalogue de Gamma et al. [Gamma *et al.*, 1995] est décrit de la manière suivante :

> *Il est utile qu'un agrégat d'objets, comme une liste, offre un moyen d'accès à ses éléments, sans risque pour sa structure interne. De plus, on peut souhaiter parcourir la liste selon des itinéraires variés, au gré de ses projets. Mais on ne souhaite pas truffer l'interface de la liste d'opérations spécifiques des différents*

> *parcours, en admettant déjà que l'on puisse prévoir ceux que l'on utilisera. On peut également avoir besoin de plusieurs parcours simultanés sur la même liste.*

Cette introduction au problème suivie par une description générale de la solution :

> *Le pattern Itérateur permet de réaliser tout cela. L'idée de base de ce pattern est d'extraire de l'objet liste les fonctions en charge des accès et parcours pour les placer dans un objet itérateur. La classe Itérateur définit une interface pour accéder aux éléments de la liste. Un objet itérateur a à sa charge le suivi de l'évolution de l'élément en cours ; i.e. qu'il sait quels éléments ont déjà été atteints*

L'aspect est complété par un exemple concret d'utilisation du pattern, i.e. une instance. Cet exemple introduit les classes, méthodes et attributs pertinents dans le pattern. Il est accompagné par un modèle de classes OMT.

Meijers cite aussi l'aspect « Utilisations connues » comme source de « l'impression de déjà vu ». Cet aspect fait des références à des utilisations connues du pattern, sans détailler l'implémentation. Par exemple, les concepteurs habitués avec l'environnement Smalltalk peuvent reconnaître une implémentation du pattern **Itérateur** dans la classe 'PositionableStream'.

L'aspect « Indications d'utilisation » du pattern décrit des situations où l'application du pattern est souhaitable. On pourra, par exemple, utiliser le pattern **Itérateur** lorsque l'on souhaite :

> *(…) accéder au contenu d'un objet d'un agrégat sans en révéler la représentation interne.*

Le diagramme de classes OMT présenté dans l'aspect « Structure » est similaire à celui de l'exemple concret donné précédemment. La différence principale concerne le nom des composants du diagramme. Les noms spécifiques à l'exemple sont remplacés par des noms plus génériques, qui décrivent le rôle de chacun des participants.

Le rôle des classes participantes est présenté par l'aspect, « Participants ». Par exemple, le rôle de la classe Itérateur est de :

> *(…) définir une interface pour accéder aux éléments et de parcourir la structure.*

En plus de sa représentation statique, il est important de connaître le comportement dynamique d'un pattern. L'aspect « Collaborations » révèle l'interaction entre les différents participants, et est parfois illustré par un diagramme d'interaction OMT. Les collaborations des participants du pattern **Itérateur** sont résumées en une seule phrase :

> *Un itérateur concret assume le référencement de l'objet courant de l'agrégat et peut déterminer l'objet suivant dans le parcours de la structure.*

Une fois que le concepteur est familiarisé avec le problème concerné par le pattern et la solution proposée, il lui reste à l'utiliser effectivement.

La première manière d'utiliser un pattern est de l'employer pour améliorer la communication entre différents concepteurs. Dans ce cas, les aspects les plus importants à retenir sont le nom du pattern, ses alias et les rôles des participants. Ainsi, lorsqu'un concepteur fait référence à un « itérateur concret » du pattern **Itérateur**, les autres concepteurs en connaissent la signification.

La deuxième manière d'utiliser un pattern est de l'appliquer pour résoudre un problème donné. Les aspects « Intention » et « Indications d'utilisation », d'une part, aident à identifier le pattern qui s'applique au problème posé. L'aspect « conséquences », d'autre part, indique les bénéfices de l'utilisation et surtout certaines contraintes, qui peuvent remettre en cause l'utilisation d'un pattern. Une des conséquences de l'utilisation du pattern **Itérateur** est décrite comme suit :

> *Les itérateurs simplifient l'interface de l'agrégat : l'interface de parcours fourni par l'Itérateur dispense du besoin d'une interface similaire dans Agrégat, simplifiant de ce fait celle de l'agrégat.*

Un pattern peut disposer de plusieurs solutions d'implémentation, qui doivent respecter certains compromis. L'aspect « Implémentation » énumère plusieurs compromis d'implémentation du pattern. Ces compromis permettent l'adéquation du pattern à une situation spécifique, et peuvent engendrer différentes variantes. Un des compromis d'implémentation du pattern **Itérateur** propose des opérations additionnelles pour les itérateurs.

> *L'interface minimale des itérateurs est formée par les opérations « Premier », « Suivant », « Terminé » et « Elément courant ». Certaines opérations supplémentaires peuvent s'avérer utiles. Par exemple, les agrégats ordonnés peuvent avoir une opération « Précédent », qui positionne l'itérateur sur l'élément précédent. Une opération « Aller à » est utile pour les collections triées ou indexées. Cette opération positionne l'itérateur sur l'objet répondant à certains critères spécifiques.*

Finalement, l'aspect « Exemples de code » donne quelques fragments de code qui montrent des exemples d'implémentation du pattern en C++ et en Smalltalk.

4. Patterns, cadres d'applications et idiomes

Une manière alternative de présenter les patterns de conception consiste à les comparer avec des concepts voisins, comme les idiomes et les cadres d'applications. Comme la différence entre ces concepts n'est pas évidente, une comparaison entre ces concepts procure une vision plus précise des patterns.

Les patterns se différencient des cadres d'applications de plusieurs manières [Lajoie et Keller, 1994]. Premièrement, ils ne sont pas une structure de classes qui peut être étendue pour réaliser une application. Ils sont une micro-architecture, i.e. ils sont composés par plus d'une classe, mais ne concernent qu'une partie réduite de l'application. Deuxièmement, les patterns sont abstraits, i.e. ils décrivent une solution générale qui s'appliquera à des cas précis, tandis que les cadres se composent de collections concrètes qui collaborent entre elles pour accomplir une tâche spécifique.

Les « Idiomes » de James Coplien [Coplien, 1992] décrivent des solutions d'implémentation. Cependant, contrairement aux patterns de conception, les idiomes sont spécifiques à un langage de programmation. Cette différence est très subtile, car certains idiomes induisent l'implémentation d'une solution similaire à celle suggérée par un pattern. L'idiome « compteurs de référence » (p 53), par exemple, décrit une manière simple de libérer la mémoire allouée à un objet qui est partagé par d'autres objets, par le remplacement des pointeurs par un mécanisme capable de compter le nombre de références à cet objet. Cette solution est une implémentation du pattern **Proxy (**ou Procuration) décrit par Gamma et al. Voici un extrait de la description de cet idiome :

> *Utilisez les compteurs de référence pour vous affranchir de la charge de devoir libérer manuellement la mémoire allouée dynamiquement. Ils peuvent aussi être utilisés lors du prototypage pour éviter de dupliquer des changements de la signature de la classe corps dans la classe gérant.*

Ce même principe, i.e. l'utilisation d'un objet « ambassadeur » à la place de l'objet souhaité, est aussi utilisé dans l'implémentation des Smart Pointers de Edelson [Edelson, 1992] et des classes 'NXProxy' de NextStep [Next, 1994], qui permettent l'accès à un objet lointain. Le pattern **Procuration** décrit ce principe de manière générique et englobe ces trois exemples.

III. Exemples de patterns de conception

Nous présentons dans cette section trois exemples de patterns de conception, extraits du catalogue de Gamma et al. [Gamma *et al.*, 1995]. Ils ont été choisis pour leur notoriété et leur simplicité et serviront de support à la section suivante. Nous ne présentons pas la description complète de ces patterns, mais seulement quelques caractéristiques qui nous permettront de les utiliser, dans la section suivante, comme exemples des difficultés d'implémentation de patterns de conception.

Itérateur, le premier pattern présenté ici est utilisé comme moyen d'accès séquentiel aux éléments d'un agrégat d'objets sans mettre à découvert la représentation interne de l'agrégat. La solution préconisée par ce pattern consiste en enlever les opérations de parcours de l'agrégat et de les transférer dans une classe séparée, appelée 'Itérateur' (Figure 2).

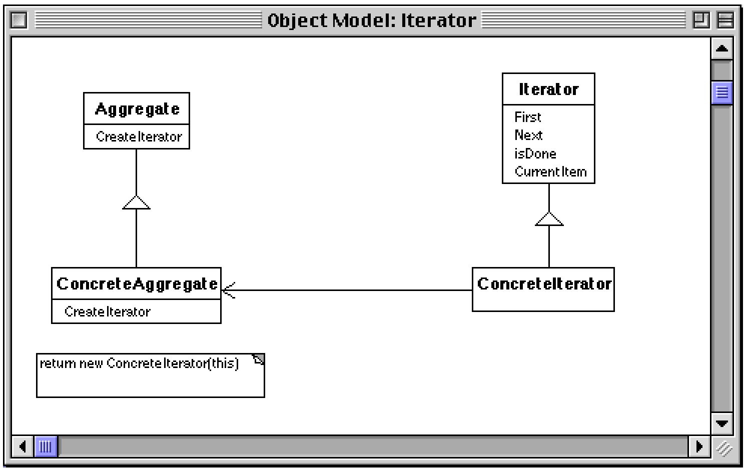

Figure 2 - Structure du pattern Itérateur

Le deuxième pattern, **Pont** (Bridge), découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre. Deux classes participantes composent le pattern, 'Abstraction' et 'Implémenteur'. La première définit l'interface de l'abstraction et gère une référence à un objet implémenteur. La deuxième réalise concrètement l'implémentation de l'interface définie par la première.

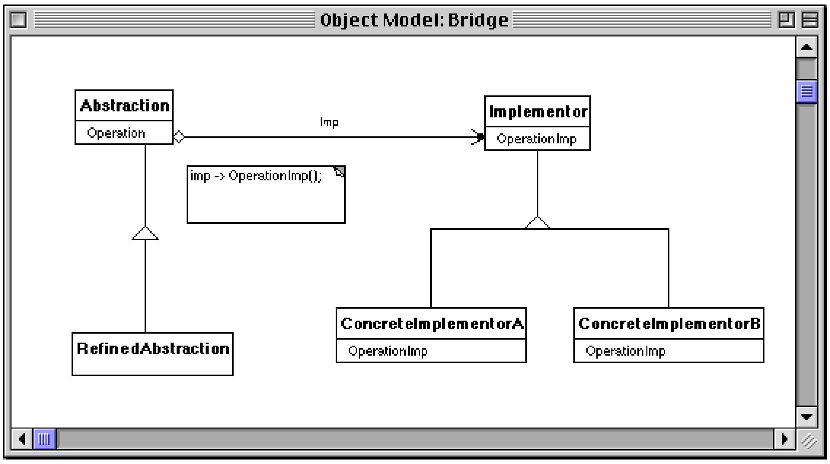

Figure 3 - Structure du pattern Bridge

Ce pattern est utile, par exemple, pour implémenter une interface graphique capable de fonctionner dans différents systèmes de gestion de fenêtres (X-Windows, Windows, etc.). Dans ce cas, une abstraction définit les opérations de manipulation d'une fenêtre et garde une référence vers une implémentation concrète. On définira autant d'implémentations concrètes que de systèmes de gestion d'interface.

Le pattern **Observateur** définit une interdépendance de type « un à plusieurs », de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour. Ce pattern est formé par deux classes participantes principales, le 'Sujet' et l' 'Observateur' (Figure 4).

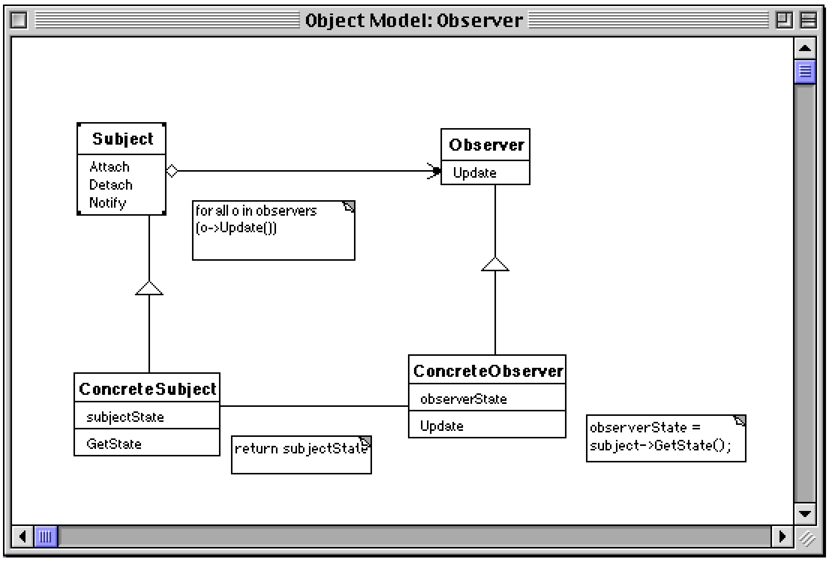

Figure 4 - Structure du pattern Observateur

La solution proposée par ce pattern est la suivante : le sujet implémente une collection d'observateurs et se charge, lorsqu'il est modifié, de notifier ceux-ci de cette modification.

IV. Difficultés de l'implémentation de patterns

Dans cette section, nous présentons les difficultés d'une utilisation particulière des patterns : la résolution d'un problème donné, durant le développement d'une application. La mise en œuvre d'une solution est une tâche complexe, qui affecte la conception et la mise en œuvre d'une application. Pour les utiliser, le concepteur doit les connaître suffisamment pour détecter, durant la conception, les problèmes qui peuvent être résolus par l'application d'un ou de plusieurs patterns. Une fois le besoin détecté et le pattern choisi, le pattern peut être appliqué.

Pour appliquer un pattern, le concepteur doit d'abord identifier quelles sont les classes (et éventuellement les méthodes et variables) participantes à l'instance d'un pattern. Il est important de signaler que les classes peuvent participer à plusieurs instances de patterns et que l'application d'un pattern peut engendrer des changements dans le modèle de conception. Une fois que les participants sont identifiés, le pattern peut être implémenté.

Pour implémenter une instance d'un pattern, l'implémenteur doit connaître ses diverses variantes d'implémentation et choisir la plus adaptée à cette situation spécifique. Ce choix est dirigé par différents facteurs, comme le modèle de conception, le langage, l'environnement applicatif ou les choix personnels de l'implémenteur.

1. Le code spécifique à une instance de pattern est difficilement réutilisable.

En effet, d'une part, l'objectif des patterns est de permettre la réutilisation de solutions de conception et non d'implémentation. D'autre part, l'implémentation est pilotée par différents compromis et peut adopter différentes formes ou variations, sans pour autant déroger à la solution de conception qu'il définit. La variation la plus évidente est celle qu'induit le contexte applicatif sur les noms des composants qui interviennent pour réaliser le pattern. Au-delà, les choix de structures de données (e.g.. pour représenter des collections : listes, tableaux, etc.) et l'organisation même des classes peuvent résulter de compromis liés au problème traité. L'implémentation d'un pattern est alors trop spécifique à un problème pour être réutilisée ailleurs [Soukup, 1995].

Le pattern Bridge, par exemple, est utilisé dans AppKit [Apple, 1997] pour implémenter la gestion d'images graphiques (Figure 5). Une image graphique ('NSImage', l'abstraction) stocke une liste des implémentations concrètes sachant la traiter (les sous-classes de 'NSImageRep').

Ce même pattern a été utilisé pour implémenter des listes agrégations dans la bibliothèque Libg++ de GNU. L'agrégation est stockée par la classe 'Set' (qui joue le rôle d'abstraction), et le contrôle de l'agrégation est réalisé par les classes 'LinkedList' et 'Hash', les implémentations concrètes.

Les classes qui composent ces deux implémentations du pattern Bridge pourraient difficilement partager leurs comportements. Il ne serait pas possible, par exemple, de généraliser les classes qui implémentent l'agrégation et l'image graphique : ces classes ne possèdent pas d'attributs communs, et n'ont pas un comportement similaire.

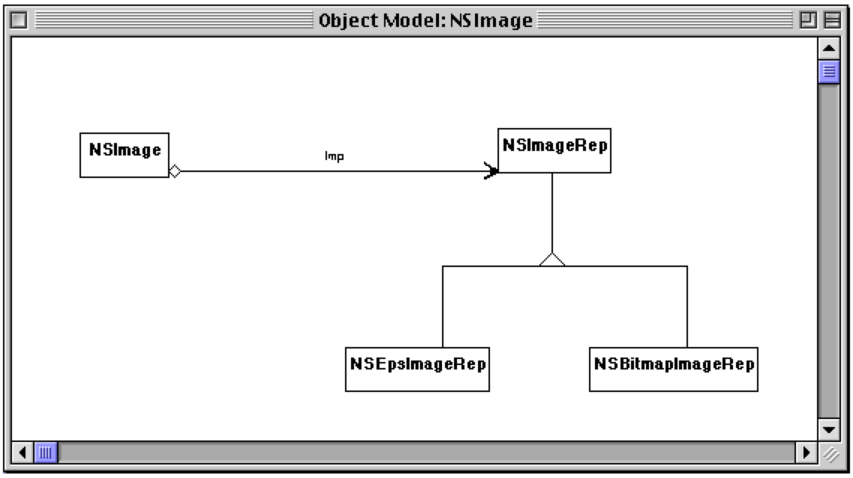

Figure 5 - Implémentation du pattern Bridge dans AppKit

Du point de vue de l'implémentation, le corps des méthodes n'est pas réutilisable, non plus. Quand les méthodes de la classe 'Abstraction' du pattern **Bridge** sont appelées, elles envoient un message à un implémenteur concret. Bien que l'implémentation de ces méthodes soit très simple, elle est spécifique à un message de la classe 'Implémenteur concret'.

2. Les patterns se perdent dans le code.

Selon Jiri Soukup [Soukup, 1995] et Jan Bosch [Bosch, 1996], les implémenteurs auraient tendance à perdre de vue les patterns après leur implémentation. En effet, en adoptant une dénomination spécifique à l'application en dépit des noms fictifs employés pour le décrire, le code spécifique à une instance de pattern peut être confondu avec le reste du code de cette application. De plus, un pattern n'étant pas toujours implémenté de la même manière, son identification à partir du code exécutable d'une application est difficile.

Premièrement, la structure d'un pattern peut varier selon les compromis implémentation suivis, ce qui rend difficile l'identification d'un pattern par sa structure. Le pattern **Bridge**, par exemple, peut être implémenté en C++ par héritage multiple, ce qui diffère de la structure originale du pattern. En plus, l'utilisation d'une classe abstraite 'Implémentation' est facultative, s'il n'existe qu'un seul implémenteur concret.

Deuxièmement, les méthodes qui composent un pattern adoptent des noms spécifiques à la leur classe. Les méthodes de la classe 'Abstraction' du pattern **Bridge** utilisent la logique suivante :

```
operationA
      self implementeur operationB
```

Lorsque le pattern est appliqué à une situation concrète, comme dans une interface graphique, le code d'implémentation ressemblera au code suivant :

```
drawRectangleFrom: aPoint: to: anExtent
      (self imp) drawRectangle: aPoint to: anExtent
```

Bien qu'il soit possible de déterminer que cette méthode ne fait qu'appeler une méthode similaire appartenant à une autre classe, son identification comme composante du pattern **Bridge** n'est pas instantanée.

3. L'évolution de l'application doit respecter les contraintes des patterns.

Comme le rappellent Florijn et al [Florijn *et al.*, 1997], l'implémentation d'un pattern respecte certaines contraintes, énoncées dans sa définition. Si l'application est modifiée après l'implémentation des patterns, ces modifications doivent aussi respecter ces contraintes.

Quand on applique le pattern Itérateur, il existe une contrainte implicite qui dit que lorsqu'un itérateur est utilisé pour parcourir un agrégat, celui-ci ne doit pas être modifié. Cette contrainte doit rester valide après l'inclusion de nouvelles méthodes susceptibles de modifier l'agrégat.

4. L'application des patterns surcharge l'implémentation

L'implémentation d'une instance de pattern engendre la création de nombreuses méthodes, qui expriment un comportement presque trivial, comme le simple envoi de messages entre objets, ce qui surcharge la tâche de l'implémenteur et nuit à la lisibilité du code [Bosch, 1996].

Le pattern **Bridge** ne prévoit aucune méthode propre. Cependant, l'application de ce pattern comprend la création de méthodes de la classe 'Abstraction' qui se chargent d'appeler des méthodes des classes « implémenteur concret ». Ces méthodes ne sont pas complexes mais, en grande quantité, surchargent l'implémentation.

5. Les compromis d'implémentation sont difficilement maîtrisables.

Les différents compromis d'implémentation ne sont pas complémentaires, les modifications qu'ils suggèrent ne s'appliquent pas les unes à la suite des autres. Au contraire, ils peuvent être combinés entre eux et dans ce cas, l'implémentation de chacun des compromis est affectée.

L'implémentation de la méthode de notification du pattern Observateur est un bon exemple de la variété d'implémentations induite par les différents compromis d'implémentation. Le rôle de cette méthode est d'envoyer un message de mise à jour à chacun des observateurs du sujet. Le code de cette méthode est simple :

```
notify
      self dependants do: [:each | each update]
```

Une deuxième manière d'implémenter cette méthode consiste à envoyer des informations additionnelles concernant le changement du sujet :

```
self dependants do: [:each | each update: self with: aParameter]
```

Parfois, les observateurs ne s'intéressent qu'à certains aspects du sujet. La méthode de notification doit alors sélectionner les observateurs avant d'envoyer le message de mise à jour :

```
(self dependants at: anAspect ) do: [:each | each notify]
```

Finalement, ces deux dernières variantes d'implémentations peuvent être combinées :

```
(self dependants at: anAspect ) do: [:each | each notifyWith:
     anAspect]
```

Ces exemples ne constituent pas les seules manières d'implémentation de la méthode de notification. Elle peut aussi changer dans le cas de l'utilisation d'un gestionnaire de changements (la mise à jour est envoyée au gestionnaire). Bien évidemment, cette méthode peut être combinée avec les variantes présentées précédemment :

```
self changeManager notify
```

Cette « concurrence » entre différents compromis d'implémentation est aussi vérifiée lors de l'implémentation des autres méthodes qui composent ce pattern, et dans d'autres patterns

L'implémentation de la méthode de parcours d'une liste dans le cas du pattern Itérateur est un autre exemple. Cette méthode peut être implémentée soit par l'agrégat, soit par l'itérateur (le contenu de la méthode est évidemment différent). Le contenu de cette méthode peut varier, si l'itérateur gradue la position relative au parcours (itérateur interne). Il peut encore varier si l'on souhaite implémenter un itérateur robuste (i.e. un itérateur qui accepte les changements de l'agrégat).

6. L'implémentation d'un pattern n'est pas isolée

D'une part, la description des patterns ne définit pas de contraintes concernant l'application multiple d'un même pattern dans une application. En effet, un pattern peut être implémenté plusieurs fois et de manière indépendante au sein d'un même programme.

Il est important de vérifier si ces implémentations ne peuvent pas être combinées ou généralisées. Deux classes jouant le rôle d'observateur pourraient engendrer, par exemple, la création d'une super-classe générique. Si une de ces classes est une spécialisation de l'autre, elles pourraient partager, par exemple, le contrôle de la collection d'observateurs.

D'autre part, certains patterns peuvent être combinés lors de l'implémentation. Le pattern **Itérateur** peut être utilisé, par exemple, pour parcourir une agrégation implémentée par le pattern **Composite**. Dans ce cas, le parcours de l'agrégation sera spécifique à la structure Composite / Composant et celle-ci peut être modifiée pour permettre le parcours. L'utilisation d'un pattern pour en implémenter un autre est un autre exemple de combinaison de patterns. Le pattern **Proxy** peut, par exemple, être utilisé dans l'implémentation d'une classe itérateur.

V. Techniques et outils de mise en œuvre de patterns

Selon Florijn et al. [Florijn *et al.*, 1997], l'outillage des patterns de conception peut suivre trois approches principales.

La première approche, dite générative ou descendante, consiste à générer le code d'implémentation d'un pattern à partir d'un choix explicite de conception. Dans cette approche, le concepteur choisit un pattern, les classes participantes et certains compromis d'implémentation et reçoit le code correspondant.

Dans la deuxième approche, de reconnaissance ou ascendante, l'outil reconnaît qu'un ensemble de classes, méthodes et attributs correspond à une instance d'un pattern et l'indique au concepteur. Cette approche est employée actuellement par deux outils : Pat [Krämer et Prechelt, 1996] et KT [Brown, 1995].

Ces deux approches peuvent être combinées pour en former une troisième, dite de reconstruction ou mixte. Dans cette approche, l'outil suggère la transformation d'un ensemble de classes ressemblant à l'instance d'un pattern en une instance effective du pattern. La transformation comprend l'ajout ou la modification de classes, de méthodes ou d'attributs.

Indépendamment de l'approche adoptée par un outil, il doit résoudre impérativement trois problèmes :

1. La représentation des patterns : l'outil doit mémoriser un catalogue de patterns, qui lui servira de référence lors de la reconnaissance d'un pattern ou de la génération du code d'implémentation. Cette représentation doit comprendre la description de la structure du pattern et des différentes variantes d'implémentation.

2. La représentation d'instances de patterns : l'outil doit permettre la représentation de l'existence d'une instance de pattern dans le modèle de conception d'une application. Cette représentation doit permettre au concepteur d'indiquer quelles sont les classes participantes à une instance de pattern et la variante d'implémentation choisie. Elle peut servir à l'outil pour indiquer la reconnaissance d'une instance de pattern.

3. Le lien entre une instance et son implémentation : l'outil doit garder un lien entre l'instance abstraite d'un pattern, représentée lors de la conception et son implémentation concrète. Ce lien est important car il permet la reconnaissance du code d'une instance de pattern dans le code de l'application et assure le passage entre les niveaux d'implémentation et de conception d'une application.

Nous présentons par la suite des outils suivant l'approche générative et certaines techniques d'implémentation de patterns. Une comparaison similaire a été proposée récemment par Isabelle Borne et Nicolas Revault [Borne et Revault, 1999].

1. CodeFarms

La plupart des techniques actuelles d'implémentation de patterns s'intéressent à leur réification sous forme de classes et aux moyens de partager leur code. Jiri Soukup [Soukup, 1995], par exemple, propose la réification du comportement et de la logique d'un pattern sous la forme d'une classe, appelée « Pattern Class », qui ne contiendrait seulement que le code d'implémentation du pattern, sans les variables concernées. Cette classe est ensuite utilisée comme une classe amie[1] des classes contribuant à l'instance d'un pattern. Cette technique d'implémentation fait appel à un pré-compilateur C du commerce (C/C++ Data Object Library de CodeFarms Inc) qui développe des macro-commandes signalées dans le code source.

Selon l'auteur, la concrétisation des patterns est une manière de pallier la perte de référence des patterns pendant le développement et de restreindre l'interdépendance entre les classes qui participent au pattern. Cependant, cette solution est une technique de codage de patterns ou plus précisément, un pattern d'implémentation de patterns (sa description selon le canevas de [Gamma *et al.*, 1993] est fournie en annexe), elle ne permet pas d'exprimer les choix de conception de l'instance à partir du pattern.

2. Pattern tool

Marco Meijers présente dans sa thèse [Meijers, 1996] le développement en Smalltalk VisualWorks 2.5 [ParcPlace, 1998] de PatternTool, un outil qui permet l'intégration des patterns à du code Smalltalk. Cet outil utilise un modèle en fragments, capable de représenter des patterns de conception (chaque pattern étant composé par plusieurs fragments). Ces fragments sont, par l'intermédiaire de rôles, liés aux classes qui vont réaliser concrètement une d'instance du pattern. Ainsi, ces classes peuvent partager, pendant leur exécution, le code appartenant au pattern. PatternTool utilise un éditeur graphique de fragments, qui permet d'inclure de nouveaux patterns dans l'outil.

Du point de vue de l'implémentation, les fragments sont une hiérarchie d'objets, où chaque objet est une agrégation d'un ou plusieurs *slots*, par analogie avec le système Self [Ungar et Smith, 1987]. Il existe trois sortes de slots : fragment, primitif et code. Les slots de fragments maintiennent une référence à un fragment d'un type particulier, et servent ainsi à représenter des rôles. Les slots primitifs peuvent contenir une référence à un objet Smalltalk quelconque, permettant aux instances de patterns d'avoir leurs propres variables. Enfin, les slots de code contiennent un bloc de 'byte code' correspondant à un comportement.

Cette approche étend la notion de Pattern Classes de Jiri Soukup, dans le sens où le modèle de fragments est plus détaillé et permet une meilleure manipulation des composants d'un pattern. Cependant, même dans cette perspective étendue, les patterns gardent un aspect figé : un pattern est codé d'une seule manière, et comme Meijers l'indique lui-même dans sa conclusion, les variantes d'un même pattern sont vues comme des patterns différents.

3. FACE

De leur côté, Meijler et Engel [Meijler et Engel, 1996] proposent FACE, qui est un environnement de développement « orienté composants », dont l'objectif principal est de simplifier l'utilisation des cadres d'application (frameworks). Il utilise des modèles de conception formés de composants (en réalité, un modèle de classes OMT étendu par des constructeurs de modélisation spécifiques) pour combiner plusieurs cadres d'applications et générer le code nécessaire à l'utilisation de chacun.

---

[1] *Friend*, dans le sens de C++

Les patterns de conception sont vus dans FACE comme des collections de composants de conception. Ils sont représentés dans un méta-modèle, qui définit la syntaxe des modèles de classes et par conséquent, des instances de patterns de conception. L'instance d'un pattern est indiquée dans le modèle de conception grâce à l'expression des rôles joués par les classes, les opérations et les relations qui la composent.

Le modèle de conception est interprété durant l'exécution de l'application, les concepts modélisés représentent directement les classes de l'application. Une approche générative n'est pas envisagée. Rien n'est dit sur la manière d'instancier un pattern par rapport aux variantes d'implémentation.

4. L'outil de Kim et Benner

Kim et Benner [Kim et Benner, 1995] proposent de séparer la conception en deux niveaux (tous deux décrits en OMT), le « niveau pattern » se situant au-dessus du « niveau de conception ». Le niveau de conception contient des classes, leurs composants et leurs associations, qui représentent ensemble le résultat de la conception. Au-dessus de lui, le niveau des patterns définit une sémantique et une structure supplémentaires. L'idée centrale est de lier les éléments de conception qui participent à une instance d'un pattern, à un modèle structurel représentant le pattern lui-même, grâce à trois types de rôles (classe, opération et relation).

Une certaine flexibilité est admise dans cette mise en relation. Les instances d'un pattern ne sont pas obligatoirement isomorphes, grâce à la notion de « chemin généralisé », qui permet la prise en compte des liens de spécialisation présents dans le niveau de conception. La question des variantes d'implémentation n'est pas posée.

5. L'outil de Budinsky et al.

Une solution assez complète est fournie par Budinsky et al. [Budinsky *et al.*, 1996]. Ils proposent un outil qui donne un accès rapide à la description des patterns de conception, organisé sous la forme de pages HTML. Grâce à un browser, le concepteur navigue entre ces pages jusqu'à ce qu'il trouve la solution qui lui convient, et il peut ensuite choisir certains compromis d'implémentation prédéfinis. L'intégration entre informations de description et compromis d'implémentation se révèle très complète. Cependant, la génération de code n'est intégrée à aucun environnement de conception : en effet, les morceaux de code C++ engendrés apparaissent dans une page HTML et doivent être copiés « à la main » et insérés à l'intérieur du code source de l'application en développement.

L'outil est composé par trois modules indépendants : le *Presenter*, un browser Web qui sert d'interface entre l'utilisateur et l'outil ; le Générateur de Code, un interpréteur *COGENT (Code Generator Template*), qui interprète les spécifications et génère le code C++ ; et le Mapper, qui détermine la coopération entre les deux premiers. Plus précisément, ce dernier reçoit des informations du browser Web par des commandes CGI (Common Gateway Interface), les traite et les transmet au générateur de code.

6. Patterns Wizard

Eden, Yhudai et Gil [Eden *et al.*, 1997] proposent Patterns Wizard, un outil d'aide à la programmation, capable de lire un programme Eiffel, de construire une représentation interne de leur structure de classes, de permettre au développeur de définir les instances de patterns souhaitées et d'ajouter au programme original, le code correspondant aux instances des patterns sélectionnés. Le choix de Eiffel est dû aux facilités liées au typage statique.

L'approche introduite par Patterns Wizard repose sur la représentation des « treillis » (*lattices*) d'un pattern, qui correspondent à la structure et au comportement spécifiés par la partie solution de ce pattern. La représentation des treillis se fait grâce à un métalangage appelé PSL (*Pattern Specification Language*), une combinaison entre un langage à objets (Smalltalk V) et une syntaxe abstraite. Le rôle de celle-ci est de représenter plus rapidement les constructeurs communs aux langages à objets (classes, héritage, etc.).

La séquence d'actions nécessaires à la réalisation d'un treillis est spécifiée par un constructeur de modélisation appelé « astuce » (*trick*). Celui-ci est défini comme un algorithme qui manipule des éléments de programmes, dont l'exécution implémente les treillis définis. Les astuces acceptent plusieurs arguments qui définissent les classes participantes et les compromis d'implémentation d'une instance de pattern.

La notion d'astuce peut être comparée avec celle de clichés du projet Programmer's Apprentice [Rich et Waters, 1990] (voir aussi § D.3, page 182), qui encapsulent la généralisation d'opérations particulières et les spécialisent quand cela est nécessaire. Toutefois, comme les astuces manipulent directement le code, elles doivent être définies par un métalangage.

7. LayOM

De son côté, Jan Bosch propose d'intégrer les patterns de conception à un modèle à objets étendu, appelé LayOM (*Layered Object Model*) [Bosch, 1996], grâce à la notion de « couches » supportée par ce modèle. Plus précisément, le rôle des couches est d'encapsuler un ensemble d'objets et, ensuite, d'intercepter et de traiter les messages qui leur sont envoyés.

LayOM intègre aussi deux autres notions, appelés « catégorie » et « état ». Une catégorie est une expression qui décrit les caractéristiques discriminatoires d'un sous-ensemble de clients possibles, qui seront traités également par une même classe. Un état est l'abstraction de la situation interne (ou concrète) d'un objet.

L'interception de messages, proposée par les couches, s'avère très appropriée à l'implémentation de certains patterns, comme le montre l'exemple présenté dans [Bosch, 1996], qui décrit l'utilisation d'une couche « Adapter » (décrite au préalable) pour implémenter le pattern Adaptateur. Cette couche se chargera, à la place d'un objet d'adaptation, de convertir les messages envoyés par un client en des messages compréhensibles par la classe adaptée :

```
class adapter
      layers
      adapt : Adapter (accept mess1 as newMessA, accept mess2, mess3
      as newMessB)
      inh : Inherit(Adaptee);
      end;
```

LayOM dispose d'un traducteur, capable de générer le code C++ correspondant à un modèle. Le traducteur est basé sur les objets de délégation de compilation (DCO [voir Bosch, 1995] pour plus de détails). Un DCO est un objet doté d'un ou plusieurs *parser* et d'un graphe de *parsing*, et qui est capable de compiler une partie de la syntaxe d'un modèle LayOM. Grâce à l'implémentation des DCO, LayOM est un modèle, extensible : des constructeurs peuvent être ajoutés par la création de nouveaux DCOs.

8. L'outil de Pagel et Winter

Bernd-Uwe Pagel et Mario Winter [Pagel et Winter, 1996] utilisent un des méta-patterns de Pree [voir Pree, 1995], « Hook & Template », pour représenter les patterns de conception et les intégrer ainsi à un outil de développement basé sur le langage de modélisation OOA (*Object-Oriented Analysis*) de Coad et Yourdon [Coad et Yourdon, 1990]. L'idée est de coupler l'outil avec un référentiel de patterns, et de permettre l'application d'un pattern par simple copie.

Le méta-pattern représente un pattern sous la forme de classes et de méthodes classées comme « template » ou « hook ». Les classes template sont celles couplées à au moins une classe participante du pattern. Ce couplage est souvent réalisé par une méthode, appelée template. Les méthodes hooks sont des méthodes complètes ou partielles utilisées en au moins une méthode template. Elles résident dans des classes hook, qui sont des classes abstraites complètes ou partielles et couplés à au moins une classe template.

Rien n'est dit sur les variantes d'implémentation. Cependant, la nature de l'application des patterns (par copie) nous laisse supposer que seule la solution type d'un pattern est acceptée.

9. La technique de Smolárová et al.

En revanche, Smolárová et al. [Smolárová *et al.*, 1998] abordent le problème des variantes d'implémentation. Ils proposent une technique de représentation des patterns de conception en un méta-schéma, un modèle Entité/Association enrichi par des contraintes, où les classes et les méthodes qui composent un pattern sont représentés par des entités. Le rôle des contraintes est de restreindre les variantes acceptées lors de l'application d'un pattern. Actuellement, cette technique n'est pas implémentée et elle ne comprend pas la génération de code.

10. Synthèse

	Nature	Représentation de patterns	Représentation d'instances de patterns	Lien instance/ code	Support aux variantes
CodeFarms	Outil	Classes C/ compilateur	Dans le code	(non applicable)	non
Pattern Tool	Outil	Fragments/ slots	Dans le code	(non applicable)	non
FACE	Outil	Méta-modèle	Modèle OMT		
Kim et Benner	Outil	Modèle OMT	Lien entre modèles	?	?
Budinsky et al.	Outil	Pages HTML et Cogent	Par copie	-	oui
Patterns Wizard	Outil	PSL	Dans un modèle de conception	-	?
LayOM	Outil	Méta-modèle	Dans un modèle LayOM	-	?
Pagel et Winter	Outil	Méta-patterns	Par copie	-	?
Smolárová et al.	Technique	Méta-modèle	-	-	?

Figure 6 – Synthèse des outils de mise en œuvre de patterns

La Figure 6 présente une synthèse des outils et des techniques de mise en œuvre de patterns. Comme ce tableau le démontre, peu d'outils s'intéressent aux variantes d'implémentation et à conserver des liens entre la représentation d'une instance de pattern et le code généré. Toutefois, il est possible de remarquer que la plupart des outils utilisent un méta-modèle pour représenter les pattern et représentent explicitement les instances de pattern dans un modèle de conception.

VI. Conclusion

Malgré l'avènement assez récent des patterns, il existe actuellement plusieurs techniques et outils de mise en œuvre de patterns, qui proposent des solutions diverses aux difficultés d'implémentation énoncées dans ce chapitre (voir § IV).

La plupart de ces outils ne proposent pas de support aux différentes variantes d'implémentation préconisées pour un même pattern. En effet, la multiplicité d'implémentations possibles d'un même pattern est souvent oubliée. Les patterns sont alors perçus comme ayant une implémentation unique et figée [e.g. Meijers, 1996 ; Soukup, 1995].

Cette perspective présente au moins deux inconvénients. D'une part, en limitant les variantes d'implémentation, elle restreint le nombre de cas où le pattern peut être utilisable. D'autre part, elle limite la souhaitable indépendance entre la conception et l'implémentation préconisée par le concept de pattern lui-même.

Le problème des variantes d'implémentation est considérablement abordé par l'outil de Budinsky et al. [Budinsky *et al.*, 1996]. L'avantage principal de notre système PatternGen par rapport à l'outil de Budinsky et al. réside dans sa capacité de modéliser les patterns en tant que tels. Comme nous le verrons, cette modélisation engendre un éditeur de diagrammes qui permet d'une manière intégrée de choisir les classes participantes et d'engendrer le code de l'application.

Les outils et techniques présentés ici proposent des fonctionnalités diverses et nous permettent de percevoir les caractéristiques souhaitables d'un outil de mise en œuvre de patterns. Cependant, aucun des outils présentés ne met en œuvre la totalité de ces caractéristiques. Ceci car la réalisation d'un environnement de développement évolutif est une tâche complexe, comme le rappellent les auteurs d'outils de création d'environnement de développement, les méta-outils (voir la section II du Chapitre 3, page 58). Ces méta-outils peuvent constituer une aide importante à la construction d'un outil de mise en œuvre de patterns.

Chapitre 2

Exemples d'utilisation de PatternGen

I. Introduction

Décrire PatternGen est une tâche relativement complexe car une partie importante des fonctionnalités cet outil dépendent intrinsèquement d'un autre outil, MétaGen, dont la description est tout aussi complexe. Nous croyons donc utile de commencer la description de notre outil par des exemples d'utilisation qui nous permettront de démontrer ce qui fait PatternGen avant d'entrer dans les détails de son implémentation.

Il existe trois approches de mise en œuvre de patterns (voir Chapitre 1 § V) : de reconnaissance, générative ou de reconstruction. Comme notre objectif n'est pas de reconnaître les instances de patterns, mais de les implémenter, nous nous intéresserons aux deux dernières approches.

Nous présentons dans ce chapitre deux exemples qui suivent deux approches différentes de développement avec les patterns de conception. La mise en œuvre des exemples est réalisée en Smalltalk et concerne quatre patterns extraits du catalogue de Gamma et al. [Gamma *et al.*, 1995] : **Observateur**, **Proxy**, **Composite** et **Stratégie** . La représentation de ces patterns à l'intérieur de PatternGen est détaillée dans la section III du Chapitre 4, page 106.

Le premier exemple illustre l'approche générative : le développement d'une application, en occurrence le classique MVC [Krasner et Pope, 1988]. Ce développement commence par la définition, grâce à l'éditeur de diagrammes de PatternGen, des classes qui composent MVC et des rôles que ces classes jouent dans différentes instances de patterns.

Cet exemple met en œuvre trois patterns et présente le code généré par PatternGen à partir des instances de ces patterns et des compromis d'implémentation spécifiques à ces instances. Nous profitons de cet exemple pour montrer quel serait le comportement de PatternGen si des compromis d'implémentation différents avaient été spécifiés.

Le deuxième exemple adopte l'approche de reconstruction : la transformation d'un ensemble de classes en une instance effective d'un pattern. Plus précisément, le pattern de conception Proxy sera employé pour modifier une application existante, MétaGen

L'exemple est réalisé en trois étapes : (i) une partie de cette application sera chargée dans PatternGen ; (ii) l'éditeur de digrammes sera utilisé pour indiquer les classes participantes de l'instance du pattern que l'on souhaite appliquer ; (iii) le code de l'application sera modifié par PatternGen.

Cet exemple est similaire au premier car, dans les deux cas, PatternGen est utilisé pour générer l'implémentation d'un pattern (classes, méthodes et variables) à partir de la définition explicite des instances de pattern. Cependant, il existe une différence importante entre ces deux exemples. Dans le premier, l'utilisation d'un pattern est décidée en même temps que la création des classes participantes. Dans le deuxième exemple, ces classes existent au préalable et sont modifiées par l'application des patterns.

II. MVC

Nous présentons, dans cette section, l'utilisation de PatternGen pour la mise en œuvre du cadre d'applications MVC. Cette utilisation s'appuiera sur un modèle de conception du cadre en question, où la participation des classes à une ou plusieurs instances de patterns est explicitement représentée.

La description de l'utilisation des patterns, dans le modèle de conception défini, se fait par le choix des classes participantes de chaque instance de pattern. A partir de ces choix, PatternGen génère les méthodes et les attributs d'implémentation des instances utilisées.

Le code généré ne couvre pas toutes les fonctionnalités de ce cadre d'applications. Ceci car comme le signalent Gamma et al. [Gamma *et al.*, 1995], les patterns ne permettent pas (encore) le développement complet d'une application. Cependant, ce code implémente la solution décrite par les patterns présents dans le modèle de conception, ce qui constitue une partie non négligeable du cadre.

Cette présentation commence par la description succincte de MVC et des patterns employés dans son implémentation originale. Ensuite, nous décrivons sa conception dans l'éditeur de diagrammes de PatternGen. Nous détaillons ensuite, chacune des instantiations de pattern présente dans ce modèle. A cette occasion, les compromis d'implémentation propres à chacune d'elles sont discutés. Enfin, le code engendré par ces instances de pattern en Smalltalk est montré.

1. Description du cadre d'applications MVC et spécification du problème posé

L'objectif du cadre d'applications MVC [Krasner et Pope, 1988] est de servir de base au développement d'interfaces graphiques. Il propose une architecture de développement basée sur une triade, les classes 'Modèle', 'Vue' et 'Contrôleur'. Chacune de ses classes joue un rôle très précis au cœur de l'architecture :

1. Le « modèle » stocke les données présentées par l'interface ;
2. La « vue » constitue la partie visible de cette interface ;
3. Le « contrôleur » englobe les actions qui peuvent modifier le 'Modèle'.

Dans MVC, plusieurs vues peuvent représenter, en même temps, un même modèle. Cette caractéristique importante permet notamment qu'une même donnée soit représentée par différentes sortes de graphes (camembert, barres, etc.).

Une autre caractéristique de MVC est d'actualiser les vues lorsque le modèle est mis à jour. L'information présentée est alors toujours cohérente, entre une vue et le modèle et entre différentes vues d'un même modèle.

Cette relation de dépendance entre modèles et vues est une application du pattern Observateur. Plus précisément, la classe 'Modèle' joue le rôle de sujet, tandis que la classe 'Vue' joue le rôle d'observateur.

Les différentes sortes de représentation de données sont implémentées par différentes spécialisations de la classe 'Vue'. Une de ces spécialisations, la vue « composite » permet l'utilisation de vues récursives, i.e. des vues contenant d'autres vues. L'implémentation de cette vue composite, la classe 'CompositeVue', est une application du pattern Composite, où le rôle de composant est joué par la classe 'Vue'.

Les vues peuvent effectuer des modifications sur le modèle, grâce aux contrôleurs. En effet, chaque vue dispose d'un contrôleur qui lui est attribué lors de sa création. Cependant, la diversité de sortes de vue rend difficile la maîtrise, par le contrôleur, des actions réalisables par chaque sorte de vue. Pour faire face à ce problème, MVC fait appel au pattern Stratégie : les classes 'Vue' et 'Contrôleur' participent à une instance de ce pattern, jouant les rôles de contexte et stratégie abstraite, respectivement.

D'autres patterns sont présents au développement de MVC, comme Fabrication, qui permet aux vues de spécifier les classes contrôleur par défaut ; ou Décorateur, pour assurer l'ajout dynamique de propriétés à une vue. Toutefois, les trois premiers patterns (Observateur, Composite et Stratégie) assurent les caractéristiques essentielles de ce cadre.

2. Solution du problème posé avec PatternGen

Dans PatternGen, le développement d'une application commence par la construction d'un modèle de conception sous la forme d'un diagramme de classes. La structure du diagramme construit est similaire à celle produite pour le diagramme de classes en OMT. Ce modèle est enrichi de choix d'implémentation induits par la spécification d'instances de patterns.

La spécification d'une instance de pattern comprend : (i) sa représentation dans le modèle, de manière similaire à celle d'une classe ; (ii) la définition des classes participantes à cette instance, grâce à des flèches liant l'instance à ces classes ; et (iii) la spécification de compromis d'implémentation spécifiques à cette instance, grâce à l'édition de ses propriétés.

La conception du cadre d'applications MVC suit ce principe. La Figure 7 présente le diagramme correspondant au modèle de conception de ce cadre. Ce diagramme comprend les trois classes principales qui caractérisent le cadre ('Modèle', 'Vue' et 'Contrôleur'), représentées par des cubes. Les différentes instances de patterns sont représentées par des icônes spécifiques.

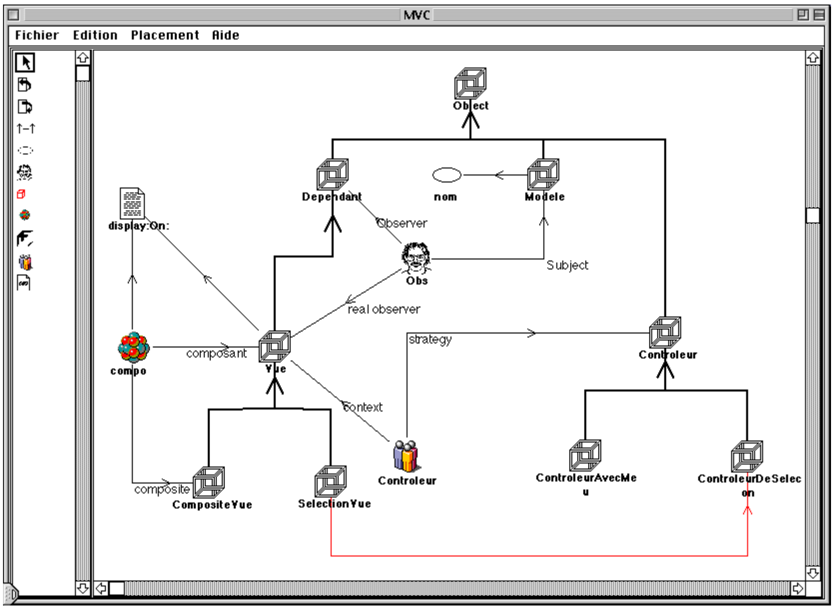

Figure 7 – Le modèle de conception de MVC

Une instance du pattern Observateur est représentée au centre du modèle (icône d'un visage). Trois flèches partent de l'icône vers des classes, indiquant les classes participantes à cette instance : 'Modele' (le sujet) ; 'Dependant' (l'observateur) ; et 'Vue' (l'observateur réel).

La classe 'Vue' participe aussi à une instance du pattern Composite, qui est représentée à gauche du modèle (icône d'une molécule). La classe 'CompositeVue' est la deuxième classe participante de cette instance de pattern.

Une troisième instance de pattern est présente dans ce modèle. Il s'agit d'une instance du pattern Stratégie (icône représentant trois personnes), qui indique deux classes participantes : 'Vue' (le contexte) ; et 'Contrôleur' (la stratégie). Cette instance de pattern est complétée par une flèche entre un contexte (la classe 'SelectionVue') et sa stratégie (la classe 'ControleurDeSelection').

Chacune de ces instances de pattern possède un ensemble de compromis d'implémentation propres, qui pilotent le choix de la variante d'implémentation du pattern la plus adaptée. Ces compromis sont établis par le concepteur, en fonction de l'implémentation qu'il souhaite réaliser.

La représentation explicite des instances de pattern dans le modèle de conception permet à PatternGen de générer automatiquement les attributs et les méthodes de leur implémentation. Dans les sections qui suivent, nous détaillons ces implémentations.

3. La mise en œuvre du pattern Observateur

La mise en œuvre de l'instance du pattern Observateur dans MVC fait appel à la variante la plus simple de ce pattern. Elle n'utilise pas de gestionnaire de changements (une classe à part, responsable par les associations entre sujets et observateurs), les sujets stockent une référence à leurs observateurs et sont responsables de les notifier lors d'un changement. L'opération de notification envoie seulement un message de changement aux observateurs, sans le détailler. Les observateurs s'intéressent indifféremment aux changements qui affectent n'importe quel aspect du sujet.

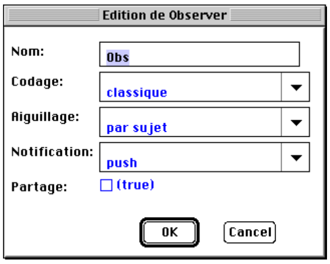

Figure 8 – Edition de l'instance du pattern Observateur

Une partie de ces compromis d'implémentation est spécifiée directement dans la boîte de dialogues présentée dans la Figure 8. Celle-ci est accessible à partir du menu contextuel de la représentation de cette instance de pattern dans l'éditeur de diagrammes.

Dans ce dialogue, la propriété 'codage' établie que l'implémentation sera normale, sans utiliser un gestionnaire de changements La propriété 'aiguillage' spécifie que le déclenchement de la mise à jour sera effectué par le sujet lui-même et non par les classes clientes. La propriété 'notification' établie le protocole de mise à jour qui sera utilisé. Dans ce cas, la notification ne sera pas détaillée. La propriété 'partage' établie si l'observation de plusieurs sujets est ou non possible.

Un dernier compromis d'implémentation, la spécification explicite de l'intérêt de l'observateur sur un aspect du sujet, sera spécifié à travers une deuxième boîte de dialogues, qui permet l'édition des propriétés de la relation libellée 'observateur concret'. Cette boîte ne dispose que d'une propriété, 'aspects', une liste de choix multiples. Les choix proposés initialement par cette propriété sont les attributs de la classe 'sujet'.

Dans le cas où plusieurs classes 'observateur réel' participent à une instance de ce pattern, chacune peut, à travers cette même boîte de dialogues, spécifier quels sont les aspects de son intérêt. Si aucun intérêt n'est spécifié, les observateurs seront notifiés de toutes les modifications du sujet.

3.1 Un observateur concret : la classe 'Vue'

La classe 'Vue' joue le rôle d'observateur concret dans cette instance du pattern. Elle est responsable par la gestion de la référence au sujet observé. Pour cela, PatternGen lui a ajouté une variable, appelée 'modele', et deux méthodes d'accès à cette variable. Le nom des méthodes et des variables sont extraits des classes participantes à l'instance du pattern.

i) Les méthodes 'modele:' et 'modele'

La première méthode présentée ici, permet l'affectation de la variable 'modele'. Cette méthode accomplie trois fonctions : (i) elle détache l'observateur de son ancien sujet, s'il existe ; (ii) elle affecte la variable 'modele' ; et (iii) elle attache l'observateur à son nouveau sujet.

Le code présenté ci-dessous a été généré. Le nom des variables a été choisi en fonction des noms des classes participantes :

```
modele: aModele
      modele isNil ifFalse: [modele detachDependant: self].
      modele := aModele.
      modele attachDependant: self
```

La rupture de la relation de dépendance entre l'observateur et son ancien sujet est important lorsqu'un seul sujet est observé à la fois. Dans ce cas, le lien entre l'observateur et son ancien sujet doit être rompu. L'attachement de l'observateur au sujet se fait de manière simple, car l'observateur ne spécifie pas d'intérêt explicite.

Si un intérêt spécifique avait été spécifié, ainsi que l'observation de plusieurs sujets, le code généré aurait été sensiblement différent :

```
modele: aModele aspect: anAspect
      modele isNil ifFalse: [ modele detachDependant: self aspect:
      anAspect].
      modele := aModele.
      modele attachDependant: self aspect: anAspect.
```

La deuxième méthode permet l'accès à la variable 'modele'. Son implémentation est triviale :

```
modele
      ^modele
```

3.2 Un sujet : la classe 'Modèle'

La classe 'Modele' joue le rôle de sujet dans l'instance du pattern. Elle est responsable par la gestion (i.e. attachement et détachement) des observateurs et par leur notification, lors d'un changement.

Les observateurs sont stockés dans une collection qui est référencée par la variable 'dependants', ajoutée par PatternGen.

i) La méthode 'dependants'

Cette méthode permet l'accès à la variable 'dependants' et, si celle-ci est nulle, lui affecte une instance d'une collection simple (Array).

```
dependants
      dependants isNil
      ifTrue: [^dependants := Array new]
      ifFalse: [^dependants]
```

La collection simple est suffisante pour gérer les différents observateurs. Cependant, si l'intérêt explicite avait été spécifié, les observateurs devraient être stockés par une collection indexée de collections. Dans Smalltalk, la classe 'Dictionary' met en œuvre une collection indexée :

```
dependants
      dependants isNil
      ifTrue: [^dependants := Dictionary new]
      ifFalse: [^dependants]
```

ii) La méthode 'notify'

Cette méthode est appelée dans l'éventualité d'un changement dans le sujet. Elle envoie le message 'update' à chaque observateur :

```
notify
      self dependants do: [:each | each update]
```

Cette implémentation est simple. Si l'intérêt explicite et le protocole de notification détaillé avaient été spécifiés, la méthode devrait sélectionner d'abord les observateurs concernés par l'aspect en question, et ensuite envoyer un message de mise à jour accompagné par un détail du changement :

```
notify: anAspect with: aParameter
      (self dependants at: anAspect) do: [:each | each update:
      aParameter].
```

iii) La méthode 'attachDependant:'

Le but de la méthode 'attachDependant:' est d'ajouter un élément à la liste d'observateurs stockée par le sujet. Le code correspondant est le suivant :

```
attachDependant: aDependant
      self dependants: (self dependants copyWith: aDependant)
```

Si l'intérêt explicite avait été spécifié, la méthode d'attachement devrait trouver (grâce à l'aspect) une collection dans la collection indexée, et ensuite insérer l'observateur :

```
attachDependant: aDependant aspect: anAspect
      self dependants at: anAspect put: ((self dependants at:
      anAspect) copyWith: aDependant)
```

iv) La méthode 'detachDependant:'

Cette méthode est complémentaire à la précédente, elle retire un élément de la liste d'observateurs. Leur code est presque similaire :

```
detachDependant: aDependant
      self dependants: (self dependants copyWithout: aDependant)
```

La spécification explicite de l'intérêt, ici aussi, accroît la complexité de la méthode :

```
detachDependant: aDependant aspect: anAspect
    self dependants at: anAspect put: ((self dependants at:
    anAspect) copyWithout: aDependant).
```

4. La mise en œuvre du pattern Composite

L'implémentation du pattern Composite dans MVC utilise une collection simple pour stocker les figures composantes d'une figure composite. Les composants ne stockent pas de référence à leur parent et ne sont pas partagés (à priori). L'accès aux informations des figures composites n'est pas optimisé par une l'utilisation d'une variable de « cache ».

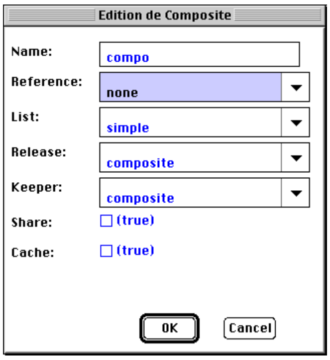

Figure 9 – Edition de l'instance du pattern Composite

Certains compromis d'implémentation ont été spécifiés directement à partir du dialogue d'édition présenté par la Figure 9. Dans ce dialogue, la propriété 'reference' spécifie que les composants n'ont pas de référence vers leur parent. La propriété 'list' établie que l'ordre des composants n'est pas important. La propriété 'release' spécifie que le composite doit s'occuper de la suppression des composants. La propriété 'keeper' établie que le composite doit déclarer les opérations de gestion des enfants. La propriété 'share' établie que les composants ne peuvent pas être partagés. La propriété 'cache' spécifique que le composite ne stockera pas d'informations permettant d'accélérer l'accès aux informations de ses composants.

D'autres compromis sont définis par des relations. Par exemple, l'instance du pattern composite précise dans le modèle, à travers une flèche, que la méthode 'displayOn:' de la classe composant sera utilisée pour créer une méthode de même nom pour la classe composite. Le rôle de cette nouvelle méthode sera de renvoyer tout message d'appel aux composants.

4.1 Un composant : la classe 'Vue'

La classe 'Vue' joue aussi le rôle de composant dans une instance du pattern Composite. Cette classe peut déclarer, éventuellement, l'interface des méthodes définissant le comportement de la classe, l'accès aux enfants et l'interface d'accès au parent.

Dans notre cas, les composants ne stockent pas une référence vers leur classe parent, et l'interface d'accès aux enfants est définie directement pas la classe composite.

Si cette classe devait maintenir une référence à la classe parent, elle devrait implémenter des méthodes de gestion de cette référence, comme la méthode d'affectation d'un parent, présentée ci-dessous.

i) La méthode 'compositevue :'

Cette méthode est générée dans l'éventualité de la spécification de la référence explicite à la classe parent, sans partage de composants. Elle (i) vérifie si le composant possède déjà un parent, et si oui, se retire de sa liste de composants et (ii) affecte la variable 'compositevue' avec la référence à un parent. Si cette référence n'est pas nulle, elle s'ajoute à la liste de composants du nouveau parent :

```
compositevue: aCompositevue
      compositevue isNil ifTrue: [compositevue vues remove: self
      ifAbsent: [nil]].
      (compositevue := aCompositevue) notNil ifTrue:[compositevue
      vues add: self].
```

4.2 Un composite : la classe 'CompositeVue'

Le rôle de la classe composite est de gérer une liste de composants et définit le comportement des composites dotés d'enfants. Plus précisément, elle définit quelles méthodes doivent être renvoyées aux enfants et lesquelles doivent être traitées par le composite lui-même.

Pour lui permettre de gérer la liste de composants, PatternGen lui a ajouté la variable d'instance 'vues', qui stockera une liste de vues.

i) La méthode 'vues'

Cette méthode permet l'accès à la variable 'vues' et, si celle-ci est nulle, lui attribue une instance d'une collection simple (Array). Si l'utilisation d'une liste ordonnée avait été spécifiée, le code serait similaire, mais la variable serait initialisée avec une collection ordonnée (OrderedCollection). Le code de la méthode est le suivant :

```
vues
      vues isNil
      ifTrue: [^vues := Array new]
      ifFalse: [^vues]
```

ii) Ajouter un composant

La méthode 'addVue:' utilise la méthode précédente pour insérer un élément à la liste de composants. Le code généré est le suivant :

```
addVue: aVue
      self vues add: aVue
```

Si le composant devait stocker une référence explicite à son parent, l'insertion d'un élément à la liste serait différente. En réalité, l'insertion serait effectuée par le nouveau composant lui-même, cette méthode ne faisant qu'appeler la méthode 'compositevue:'. Cette procédure permet au composant de défaire la référence à son ancien parent (si elle existe) et à la mettre à jour.

```
addVue: aVue
      aVue compositevue: self
```

iii) Supprimer un composant

Cette méthode, complémentaire à la précédente, supprime un élément de la liste de composants. La suppression se fait par l'envoie la méthode 'remove' à la collection de vues.

```
removeVue: aVue
      self vues remove: aVue
```

Si les composants devait stocker une référence à leur parent, la suppression se ferait par l'appel de la méthode 'compositevue:', avec le paramètre 'nil'. De cette manière, le composant se retire de la liste de son ancien parent et ne s'attribue pas de nouveau parent :

```
removeVue: aVue
      aVue compositevue: nil
```

iv) Les méthodes de renvoi

Le but des méthodes de renvoie est de transmettre le message reçu par un composite à tous ses composants. Une seule méthode a été spécifiée comme étant participante à cette instance du pattern Composite : la méthode 'display:On:', de la classe 'Vue'.

A partir de cette méthode, PatternGen a créé une méthode de même nom pour la classe 'CompositeVue'. La nouvelle méthode renvoie le message reçu à tous les composants :

```
display: aGC On: aPoint
      (self vues) do: [:each | each display: aGC On: aPoint ].
```

5. La mise en œuvre du pattern Stratégie

Une instance du pattern Stratégie régie la relation entre contrôleurs et vues dans MVC. En effet, une vue (i.e. un contexte) a une partie de son comportement piloté par un contrôleur (i.e. une stratégie).

Une vue peut posséder différentes sortes de contrôleurs, ce qui lui permet de changer « à la volée » son comportement. Cependant, chaque vue possède un contrôleur par défaut qui lui est assigné lors de sa création.

Les contrôleurs maintiennent une référence vers les vues. Celle-ci leur permet d'avoir accès à certaines informations des vues, nécessaires à l'exécution de certaines méthodes (i.e. algorithmes).

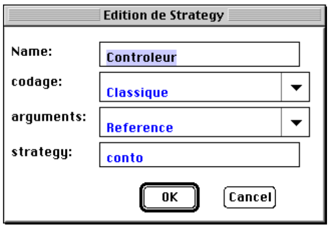

Figure 10 - Edition de l'instance du pattern Stratégie

Le choix des compromis d'implémentation se fait à l'aide du dialogue d'édition présenté par Figure 10. Dans ce dialogue, les propriétés 'codage' et 'arguments' spécifient respectivement que l'implémentation sera normale (et non par l'utilisation de templates) et que la stratégie stockera une référence vers le contexte.

De plus, une flèche partant de la classe 'SelectionVue' vers la classe 'ControleurDeSelection' désigne celle-ci comme la stratégie (i.e. le contrôleur) par défaut de la première.

5.1 La classe 'Vue'

La classe 'Vue' joue le rôle de contexte dans l'instance du pattern. Elle gère la référence au contrôleur qui lui sert de stratégie. Pour cela, PatternGen lui ajoute la variable 'controleur' et deux méthodes d'affectation et accès à cette variable.

i) La méthode 'controleur:'

Cette première méthode permet l'affectation de la variable 'controleur'. En même temps, elle affecte la variable 'vue' du contrôleur et assure la relation entre les deux classes :

```
controleur: aControleur
      controleur isNil ifFalse: [controleur vue: nil].
      controleur := aControleur.
      aControleur vue: self.
```

Si les arguments nécessaires aux méthodes du contrôleur étaient passés par valeur, le contrôleur n'aurait pas besoin de connaître sa vue. Dans ce cas, le code de la méthode serait plus simple :

```
controleur: aControleur
      controleur := aControleur
```

ii) Les méthodes 'controleur'

Cette méthode permet l'accès à la variable 'controleur' :

```
controleur
      ^controleur
```

iii) La méthode 'controleur' de la classe 'SelectionVue'

La classe 'SelectionVue' joue aussi le rôle de contexte de pattern et implémente la méthode 'controleur'. Celle-ci, qui est similaire à la précédente, permet l'accès à la variable 'controleur'.

Mais avant de retourner sa valeur, elle vérifie si cette variable a déjà été initialisée et, le cas échéant, lui attribue une instance de la classe 'ControleurDeSelection' (sa stratégie par défaut).

```
controleur
     controleur isNil
         ifTrue: [^controleur := ControleurDeSelection new]
         ifFalse: [^controleur]
```

5.2 La classe 'Controleur'

Cette classe joue le rôle de stratégie dans l'instance du pattern. Elle définit l'interface aux algorithmes des autres contrôleurs et gère la référence à une vue. La référence est stockée par la variable 'vue'

i) Les méthodes 'vue' et 'vue:'

Ces deux méthodes permettent l'accès à la variable 'vue' et son affectation :

```
vue
     ^vue


vue: aVue
     ^vue := aVue
```

6. Discussion

Cet exemple nous a permis de présenter le code généré par PatternGen et dans certains cas, le code qui aurait été généré si différents compromis d'implémentation avaient été spécifiés. Nous avons adopté ici une approche générative de mise en œuvre de patterns : les patterns, les classes participantes et les compromis d'implémentation étaient connus au préalable. Le rôle de notre outil a été de rendre le code correspondant à chacune des instances de pattern définies.

Les patterns **Observateur** et **Composite**, en raison de leur grand nombre de compromis d'implémentation, s'adaptent parfaitement à cette approche et permettent la génération de plusieurs variantes d'implémentation. De plus, ces patterns ont engendré la génération d'un nombre important de méthodes.

Le pattern **Stratégie** propose un nombre restreint de variantes, et son implémentation n'a pas été aussi importante que celle des patterns précédents. Ceci car son utilisation est plus intéressante dans la modification structurelle d'une situation initiale. Par ailleurs, ces modifications sont clairement spécifiées par sa description. D'autres patterns présentent cette même caractéristique. C'est le cas de **Proxy**, dont l'utilisation sera décrite dans l'exemple suivant.

III. Des procurations d'individus dans MétaGen

L'exemple précédent montre le développement d'une application à partir d'un modèle de conception et des instances de pattern présentes dans ce modèle. Ces instances ont engendré la génération du code de plusieurs méthodes.

Cependant, ces instances n'ont pas effectué de changements dans la structure du modèle de conception. Ceci car il s'agissait d'une nouvelle application où toutes les classes participantes des différentes instances de patterns étaient précisément définies.

Dans cette section, nous présentons une approche différente d'utilisation des patterns : la modification d'une application déjà existante. Plus précisément, les directives d'utilisation d'un pattern seront utilisées pour modifier une partie du code de l'application de manière à le transformer en une instance de ce pattern.

L'application en question est MétaGen, un outil permettant le développement rapide d'éditeurs de diagrammes et qui a été employé pour développer l'éditeur de diagrammes de PatternGen. MétaGen est décrit au Chapitre 3 et son implémentation en Smalltalk à la section IV, page 82 de ce même chapitre.

1. Des objets partagés dans MétaGen

MétaGen permet l'édition simultanée de plusieurs modèles décrits en différents langages de modélisation. Cependant, il ne dispose pas de référentiel de données (i.e. site capable de stocker les concepts modélisés) et en conséquence, ne permet pas le partage de modèles ou de « fragments » de modèles.

Partager les éléments d'un modèle MétaGen signifie, en termes d'implémentation, partager un ensemble d'instances de la classe 'MGIndividual'. Cette classe correspond « grosso modo » à la partie non visible de tous les éléments présents dans un modèle. Pour être partageables, les instances doivent être stockées dans un référentiel distant et accessible par différentes instances de MétaGen.

L'inconvénient de l'utilisation d'un référentiel est le coût d'accès aux informations, ce qui n'est pas négligeable dans un modèle composé par un grand nombre d'éléments. En plus, seulement une partie des informations d'un élément - plus précisément celles qui sont visibles dans l'éditeur - est nécessaire dans un premier temps, les autres informations ne sont nécessaires que lorsqu'un élément est consulté ou édité.

Le pattern de conception **Proxy** propose une solution originale à ce problème spécifique : créer une procuration (dite 'proxy') de l'objet d'accès coûteux, responsable de la création, en temps voulu, de celui-ci. La procuration est aussi chargée de transmettre les messages qui lui sont envoyés à l'objet coûteux (dit 'sujet réel').

2. La mise en œuvre du pattern Proxy

L'implémentation du pattern **Proxy** dans PatternGen se fait en trois étapes. Premièrement, le code Smalltalk des classes concernées a été chargé dans un modèle de l'éditeur de diagrammes. Ensuite, une instance de ce pattern a été créée dans ce modèle et la classe 'MGIndividual' a été indiquée comme participante à cette instance en tant que 'sujet réel'. Finalement, le modèle a été adapté à la structure du pattern et le code a été généré.

Il est important de souligner que le code généré correspond à une partie importante du code nécessaire pour implémenter le partage de modèles, mais non à sa totalité. Ceci car certaines informations (e.g. la manière d'accéder à l'objet coûteux) ne peuvent pas être déduites automatiquement.

La Figure 11 présente le modèle de conception de la partie concernée de l'application. Ce modèle est composé par cinq classes et par une instance du pattern Proxy, représentée par l'icône d'un dossier. Une seule flèche part de cette icône vers une classe, 'real subject', qui détermine la seule classe participante à cette instance du pattern.

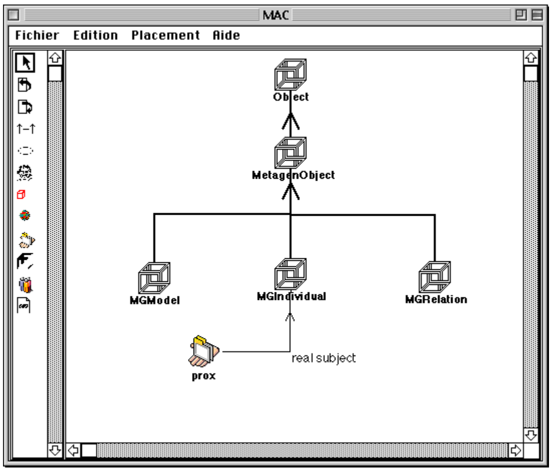

Figure 11 – La classe 'MGIndividual' et le pattern Proxy

Une fois que l'utilisation du pattern a été décidée, une question importante doit être abordée : comment implémenter le référentiel ? Au moins deux solutions sont possibles, l'utilisation d'un système de gestion de bases de données (e.g. Sybase, Oracle) ; et l'utilisation d'un système permettant l'accès à des objets distants (e.g. Corba).

Le pattern Proxy est approprié à ces deux solutions et à d'autres comme l'accès à un objet protégé ou à un fichier contenant une image. Les mises en œuvre de ces différentes solutions diffère sur plusieurs points, sans pour autant modifier le concept initial (une procuration gère l'accès à un objet coûteux).

Dans les sections suivantes, nous détaillerons deux mises en œuvre du pattern Proxy à l'aide de PatternGen. La première permet le chargement d'un objet stocké dans une base de données. La deuxième, l'accès à un objet distant.

3. Des procurations d'objets coûteux

La première solution consiste à utiliser une procuration capable d'accéder à une base de données et de récupérer les informations permettant la création d'une instance de la classe 'MGIndivudal', le 'sujet réel'. En plus, cette procuration doit être capable de vérifier si le 'sujet réel' existe, avant de lui renvoyer les messages reçus, et de le créer lorsqu'il est nécessaire.

Pour implémenter cette instance spécifique du pattern Proxy, certains compromis d'implémentation sont définis grâce à un dialogue d'édition (Figure 12). La propriété 'implementation' établie que la mise en œuvre sera classique (i.e. pas d'astuce pour implémenter le renvoie de messages) et que la nature du pattern sera, elle aussi, classique (i.e. le 'sujet réel' n'est pas un objet distant ou objet protégé).

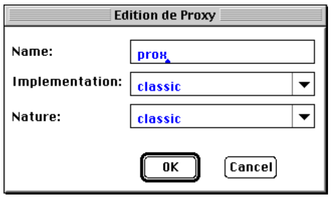

Figure 12 - Edition de l'instance du pattern Proxy

La Figure 13 donne un premier aperçu de l'implémentation de cette instance du pattern Proxy, pilotée par les compromis établis précédemment. Dans un premier temps, le modèle est modifié : le sujet réel change de nom et deux nouvelles classes sont créées.

Plus précisément, la classe 'MGIndividual' (le sujet réel) devient 'RealMGIndividual' et une nouvelle classe 'MGIndividual' (le sujet) est créée. Celle-ci devient la super-classe de la classe renommée et d'une nouvelle classe, appelée 'MGIndividualProxy' (la procuration). L'instance du pattern possède, à présent, toutes ses classes participantes.

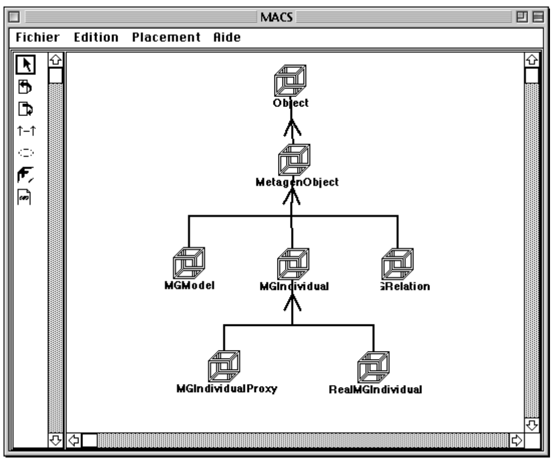

Figure 13 – Création des classes participantes au pattern Proxy

Cette « mise en forme » du modèle peut être effectuée par les instances d'autres patterns, comme Stratégie et Composite. Par exemple, si une instance du pattern Composite ne possède pas de classe participante jouant le rôle de 'composite', celle-ci est automatiquement créée.

Une fois que les deux nouvelles classes sont introduites dans le modèle, PatternGen crée les méthodes et les attributs de mise en œuvre du pattern, qui seront présentés dans les sections suivantes.

3.1 Un sujet, la classe 'MGIndividual'

Cette classe joue le rôle de 'sujet' dans l'instance du pattern. Dans la description originale du pattern Proxy, cette classe devrait définir une interface commune pour les classes 'RealMGIndividual' et 'MGIndividualProxy'. Cependant, cette définition n'est pas nécessaire dans le langage Smalltalk.

Elle implémente une seule méthode.

i) La méthode de classe 'new'

Cette méthode crée une instance de la classe procuration, qui remplace ainsi l'instance la classe 'MGIndividual' qui était attendue.

```
new
      MGIndividualProxy new
```

3.2 Une procuration, la classe 'MGInividualProxy'

La classe 'MGIndividualProxy' joue le rôle de procuration. Cette classe doit gérer une référence vers le sujet réel et lui renvoyer les messages reçus. Pour cela, PatternGen lui ajoute une variable, appelée 'mgindividual'.

i) La méthode d'accès à 'mgindividual'

Cette méthode permet l'accès à la référence au sujet réel. Avant de renvoyer la référence, elle vérifie dans un premier temps si l'objet référence existe et si tel n'est pas le cas, le crée.

```
mgindividual
      mgindividual isNil
      ifTrue: [^mgindividual := self loadMGIndividual]
      ifFalse [^mgindividual]
```

ii) La méthode de chargement du sujet réel

Cette méthode crée le sujet réel. Elle doit être modifiée pour permettre le changement du sujet à partir d'un accès à la base de données.

```
loadMGIndividual
      ^RealMGIndividual new
```

iii) Les méthodes de renvoi

La classe 'MGIndividual' initiale possède plusieurs dizaines de méthodes. PatternGen crée, pour chacune de ces méthodes, une méthode de renvoie similaire à celle présentée ci-dessous.

```
addRelation: aMGRelation to: aMGIndividu withDrawing: aBoolean
      ^self mgindividual
      addRelation: aMGRelation to: aMGIndividu withDrawing: aBoolean
```

3.3 Un sujet réel, la classe 'RealMGIndividual'

La dernière classe participante à cette instance de pattern, le sujet réel, n'est pas modifiée. Ses méthodes et ses attributs restent identiques à ceux de la classe originale ('MGIndividual') avant son changement de nom.

4. Des procurations d'objets distants

Cette deuxième solution consiste à utiliser une procuration capable d'encapusler les messages reçus et de les renvoyer à un objet distant. Elle diffère de la solution précédente car la procuration ne crée pas une instance de la classe 'MGIndivudal'. Celle-ci existe déjà, mais n'est pas stockée localement.

L'implémentation de la procuration présente aussi des différences. Premièrement, la mise en œuvre de Corba en Smalltalk que nous avons choisie (Orbix) propose une classe abstraite ('CorbaObjRef') permettant l'implémentation de procurations. Celle-ci doit être prise en considération lors de l'implémentation. Deuxièmement, la procuration n'a pas besoin de stocker une référence vers le sujet réel, Corba gère une table de correspondance entre procurations et sujets réels.

Les compromis d'implémentation sont définis de la même manière (Figure 14). Les propriétés établissent que le renvoi de messages sera fait par la surcharge de la méthode 'doesNotUnderstand' et que le sujet réel sera un objet distant.

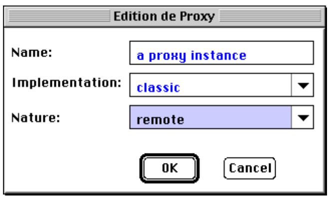

Figure 14 - Edition de l'instance du pattern Proxy

La Figure 15 explicite les différences existantes entre l'implémentation actuelle et la précédente. En effet, la classe 'MGIndividual' n'a pas été modifiée et une seule classe a été créée, 'MGIndividualProxy', sous-classe de 'CorbaObjRef'.

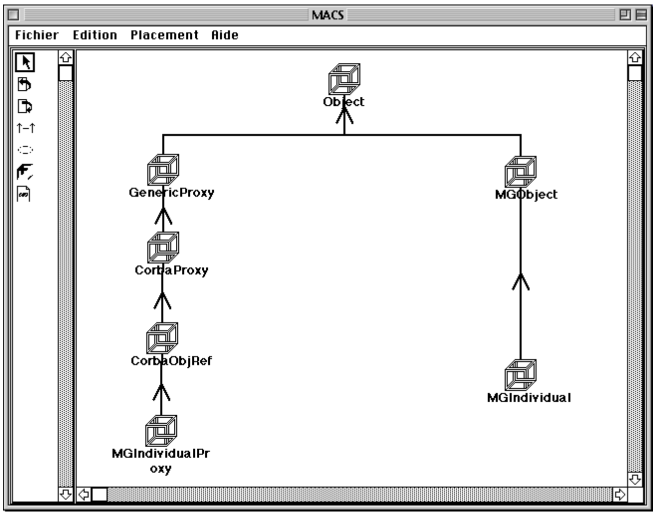

Figure 15 – Les procurations dans CORBA

La mise en œuvre de plusieurs patterns est simplifiée dans certains environnements d'implémentation. Par exemple, les classes 'NXProxy' de NextStep [Next, 1994] et 'NSProxy' de AppKit [Apple, 1997] sont similaires à 'CorbaObjRef'. Dans ces cas et si possible, l'implémentation du pattern doit tenir compte de ces classes.

Après l'introduction de la nouvelle classe dans le modèle, PatternGen crée ses méthodes et ses attributs, présentés dans la section suivante.

4.1 La classe 'MGIndividualProxy'

La procuration est la seule classe générée par cette variante d'implémentation du pattern. Son rôle est de renvoyer les méthodes reçues au sujet distant. Dans l'implémentation choisie par Orbix, le renvoi de messages est fait par la méthode 'doesNotUnderstand:' Tout message qui n'est pas compris par le proxy est renvoyée au sujet.

L'implémentation de notre procuration n'est pas totalement conforme aux indications du pattern et peut être vue comme une variante d'implémentation supplémentaire. Plus précisément, la classe 'MGIndividualProxy' possède les même variables que le sujet réel et les utilise comme cache. Elles sont affectées la première fois que l'accès à la variable est demandé.

i) Les méthodes d'accès au variables

Pour chaque variable du sujet réel, PatternGen a ajouté une variable de même nom à la procuration et créé une méthode d'accès à cette variable. Nous présentons ci-dessous la méthode d'accès à la variable 'model'.

```
model
        model isNil ifTrue: [model := super model ].
        ^model
```

Lorsque la procuration reçoit le message 'model', elle vérifie si la variable de même nom est nulle. Si tel est le cas, elle renvoie le même message à sa superclasse. Celle-ci ne le comprendra pas, ce qui déclenchera le mécanisme de 'doesNotUnderstand' et le renvoie du message au sujet réel.

5. Discussion

Cet exemple illustre une deuxième approche (dite de reconstruction) de mise en œuvre de patterns. Pour adapter les classes de MétaGen à la structure proposée par le pattern **Proxy**, PatternGen a réalisé plusieurs modifications dans le modèle de conception de MétaGen.

Ces modifications engendrent le déplacement de méthodes d'une classe à une autre et la création de méthodes, d'attributs et de classes permettant la transformation d'un ensemble de classes en une instance effective du pattern **Proxy**.

La mise en œuvre de Proxy n'a pas engendre la même diversité de méthodes que celle des patterns de l'exemple précédent. Cependant, l'automatisation de la mise en œuvre de ce pattern n'est pour autant moins intéressante car elle a libéré l'implémenteur d'une tâche fastidieuse : la création des méthodes de renvoi.

L'exemple illustre aussi la mise en œuvre de deux variantes d'implémentation du pattern **Proxy**. Dans le cas spécifique de la deuxième variante, PatternGen a dû prendre en compte la présence de l'implémentation d'un protocole de communication (Orbix) dans l'environnement de développement.

IV. Conclusion

Les deux exemples d'utilisation de PatternGen présentés ici illustrent deux approches différentes : le développement d'une nouvelle application et la modification d'une application déjà existante.

Dans le premier exemple, le développement d'un cadre d'applications, PatternGen a généré une partie importante du code du cadre MVC. Cette génération est possible grâce à la représentation explicite d'instances de patterns et de leurs compromis d'implémentation à l'intérieur du modèle de conception.

Dans PatternGen, les instances de patterns intègrent le modèle de conception d'une application. Plus précisément ils s'affichent comme des constructeurs de modélisation ayant des relations avec d'autres constructeurs (e.g. classes, méthodes, etc.), ce qui permet au concepteur de spécifier clairement quels sont les participants de cette instance. Les compromis d'implémentation de ces instances sont spécifiés à l'aide d'un dialogue d'édition.

Les compromis d'implémentation sont également mis en évidence dans le deuxième exemple, où deux solutions structurellement différentes d'un même pattern ont été proposées. Pour adapter une application déjà existante à la structure proposée par le pattern, PatternGen lui a ajouté de nouvelles classes et de nouvelles méthodes. Ainsi, PatternGen diminue la surcharge d'implémentation induite par l'application d'un pattern (voir § IV, Chapitre 1).

Dans le cas spécifique de la deuxième solution, l'implémentation a dû prendre en compte l'environnement d'implémentation car, comme nous l'avons vu précédemment (voir § IV, Chapitre 1), l'implémentation d'un pattern n'est pas isolée, elle peut dépendre de l'environnement d'implémentation et de l'existence d'autres instances de patterns.

En plus de simplifier l'implémentation de patterns illustrée par ces deux exemples, PatternGen simplifie aussi la compréhension d'une partie du comportement des classes d'une application grâce à la représentation explicite de leur participation dans une instance de pattern. Cependant, le développement de ces deux exemples mène à une question plus générale portant sur l'implémentation du reste de l'application : les patterns ne permettant pas le développement complet d'une application, comment développer le reste d'une application ?

Notre réponse va dans le sens d'une plus grande intégration des patterns dans le processus de développement. Notamment, lors du difficile passage entre l'analyse et la conception, l'intégration d'outils tel que PatternGen dans des environnements de développement explicitant les choix entre les modèles de conception et l'implémentation, doit permettre d'intégrer la réutilisation raisonnée des solutions prônées par les patterns de conception.

Nous allons, dans le Chapitre 3, présenter MétaGen, l'outil utilisé pour mettre en œuvre PatternGen. L'implémentation de PatternGen et la représentation de patterns seront présentés au Chapitre 4. Enfin, l'intégration de PatternGen au processus de développement d'une application sera présentée dans le Chapitre 5.

Chapitre 3

MétaGen, un environnement de méta-modélisation

I. Introduction

MétaGen est le méta-outil de développement logiciel développé au sein de l'équipe MétaFor. Un méta-outil est un outil capable de (i) permettre la représentation d'un environnement de développement logiciel dans un « modèle de construction » et de (ii) générer automatiquement, à partir de ce modèle, l'environnement souhaité. MétaGen, a été utilisé pour mettre en œuvre PatternGen, qui est présenté dans le Chapitre 4.

Dans MétaGen, à l'instar de la plupart des méta-outils (voir § II.2), le modèle de construction est en réalité un « méta-modèle » représentant la syntaxe d'un langage de modélisation. Le méta-modèle d'un langage de modélisation permet à MétaGen de générer son outillage et une « structure sous-jacente » capable de stocker des modèles issus de ce langage. L'outillage généré est composé par deux sortes d'éditeurs, de diagrammes et de listes (*browsers*), par un système d'aide hypertexte et par un générateur de rapports. Les éditeurs de diagrammes générés par MétaGen offrent plusieurs possibilités de masquage et de « zooming », qui permettent de cacher ou de mettre en évidence certains aspects d'un modèle.

MétaGen intègre deux mécanismes de manipulation de modèles. Le premier de ces mécanismes est un moteur d'inférences en chaînage avant. L'utilisation d'un moteur d'inférences dans un méta-outil est aussi proposée par le projet DSF [Mi et Scacchi, 1991]. Le deuxième mécanisme, à l'instar du méta-outil SMART [Garg *et al.*, 1994], est un moteur d'interprétation ou de parcours de graphes.

Ces deux mécanismes permettent la création d'instruments de manipulation automatique, comme la transformation de modèles et leur validation Il est possible, par exemple, de créer un outil de conception de bases de données [voir Sahraoui et Revault, 1994], capable de lire un modèle Entité/Association et de générer le code SQL de création des tables relationnelles correspondantes.

Contrairement aux autres méta-outils, MétaGen ne déduit pas directement l'outillage d'un langage de modélisation à partir de son méta-modèle. Il évite ainsi de surcharger la représentation de la syntaxe d'un langage (i.e. son méta-modèle) avec des détails propres à l'implémentation d'outils de modélisation (e.g. symboles de représentation, couleurs, dialogues d'édition, etc.).

Dans MétaGen la génération de l'outillage d'un langage de modélisation est totalement pilotée par un module, appelé le « méta-éditeur ». Celui-ci s'appuie sur le méta-modèle de ce langage et sur des définitions propres aux caractéristiques des outils souhaités pour générer l'outillage de modélisation de ce langage. Le résultat est un ensemble d'outils qui permettent la création de modèles obéissant à la syntaxe définie dans le méta-modèle. Ces outils peuvent être différents sous plusieurs aspects, comme la notation employée, la représentation graphique ou l'édition des concepts modélisés.

MétaGen sert comme outil de support aux expériences menées dans l'équipe MétaFor et à sa démarche de développement d'applications, présentée dans le Chapitre 5 (page 133). Il a été présenté dans le mémoire de doctorat de Houari Sahraoui [Sahraoui, 1995], qui dédie un chapitre à la description de ses fonctionnalités. Le mémoire de Nicolas Revault [Revault, 1996] complète cette description, par la présentation du processus d'utilisation de MétaGen, illustré par la mise en œuvre d'un outillage de modélisation de bases de données, basé sur le langage Morse.

Cependant, comme nous verrons par la suite, MétaGen a subi d'importants changements lors du développement de la version actuelle, réalisée par Bruno Lesueur en collaboration avec l'auteur [Sunyé *et al.*, 1997]. Dans ce chapitre, nous présentons les nouvelles fonctionnalités de l'outil et complétons ainsi les présentations de Sahraoui et Revault.

Ce chapitre est divisé en cinq parties. Après cette introduction, nous énumérons différents projets de recherche dans ce même domaine. Ensuite, nous présentons les principales fonctionnalités de MétaGen. Dans la quatrième partie, nous décrirons son implémentation, dans le but de simplifier sa compréhension par de nouveaux développeurs. Nous terminerons par une conclusion dans laquelle nous énumérerons les leçons tirées de la version actuelle de MétaGen et les évolutions prévues.

II. Intégration d'outils, méta-outils et modélisation du processus

Depuis les années 80, une vaste gamme d'outils d'automatisation du processus de développement (en anglais Computer-Aided Software Engineering tools) a été proposée. L'objectif commun de ces outils était d'augmenter l'efficacité et la qualité de la production de logiciels par la réalisation automatique des tâches fastidieuses réalisées durant le développement.

Depuis leur début, les techniques d'automatisation ont suivi deux directions distinctes, guidées par les deux cycles de vie logiciel les plus populaires : en cascade [Royce, 1970] ou en spirale [Bohem, 1986]. Il est alors possible de classer les outils en deux familles différentes[1] [Schefström, 1993], selon le cycle de vie supporté.

Les outils de la première famille, aussi appelés des *Upper CASE tools* ont comme objectif l'automatisation des phases initiales du développement. Cette famille s'intéresse à l'automatisation des méthodes de développement structuré [Gane et Sarson, 1979 ; Jackson, 1983 ; Yourdon, 1989] et met en valeur les phases d'analyse et de conception des systèmes, au détriment des techniques de mise en œuvre. Les tâches les plus communément automatisées sont :

1. L'édition, l'analyse et la vérification de diagrammes conceptuels ;
2. L'interrogation, l'importation et l'exportation des données d'un référentiel ou d'un dictionnaire de données ;
3. La génération de canevas de programmes ou de documentation ; et
4. La définition de formats de présentation.

Les outils de la deuxième famille, ou *Lower CASE tools*, s'intéressent aux phases finales du développement, comme l'implémentation, les essais et la maintenance. Les exemples les plus connus de cette famille sont les environnements de programmation comme InterLisp [Teitelman et Masinter, 1981] ou Smalltalk [Goldberg et Robson, 1983], aussi bien que les langages de quatrième génération (L4G), largement utilisés dans le développement de systèmes commerciaux et de gestion. Les fonctions automatisées sont principalement :

1. Le contrôle de l'exécution d'un programme (débogage interactif) ;
2. L'analyse dynamique d'un programme (statistique d'exécution) ;
3. L'analyse statique d'anomalies (variables non déclarées, fonctions non appelées, etc.) ;
4. L'édition structurée de programmes ;
5. La simulation d'utilisateurs (génération de données d'essai) ;
6. L'édition de renvois (de programmes, de variables, etc.) ;

[1] La frontière entre ces deux familles est subtile car les deux cycles de vie s'intéressent à la totalité du processus de développement. Toutefois, le cycle de vie en cascade attache une importance particulière aux premières phase (analyse et conception) du développement alors que le cycle en spirale paraît s'intéresser surtout aux phases finales (implémentation et tests).

7. La génération de rapports ;

8. L'interrogation de bases de données (SQL).

Une classification, alternative à celle présentée ici, a été proposée par [Fugetta, 1993]. Elle consiste en classer les outils d'automatisation du processus de développement en trois groupes, selon l'ampleur de l'automatisation : « outils », « workbenchs » et « environnements ». Ainsi, les outils automatisent une tâche dans une des phases du développement, les workbenchs automatisent totalement une phase et les environnements intègrent plusieurs workbenchs afin d'automatiser plusieurs phases.

Des études récentes sur les outils d'aide à l'automatisation du processus de développement logiciel [voir Stobart *et al.*, 1993], mettent en évidence plusieurs défaillances des outils actuellement commercialisés. Parmi ces défaillances, (i) l'impossibilité d'utilisation simultanée de différentes méthodes, (ii) la pauvreté d'intégration entre les outils s'intéressant aux différentes étapes du développement, (iii) le faible couplage entre les modèles et les référentiels de données et (iv) l'absence d'une communication effective entre outils, permettant à différents développeurs de travailler en même temps.

Des techniques d'intégration d'outils ont été proposées en réponse à la pauvreté d'intégration entre différents outils. Actuellement, deux approches sont envisageables : (i) les *canevas d'environnements*, qui proposent des services de bas niveaux permettant de combiner les outils basés sur le même canevas et (ii), les *standards*, qui proposent des formats communs d'échange d'informations.

Les méta-outils sont construits autour d'un méta-modèle qui, combiné avec d'autres spécifications, permettent la description d'un environnement de développement propre à une méthode de développement. Le noyau de ces outils dispose de capacités de haut niveau et des moyens de personnaliser ces capacités.

Les outils centrés « processus de développement », proposent une approche alternative d'automatisation du développement. Cette approche consiste à offrir aux développeurs des moyens d'assistance et de guidage. Certains outils sont munis d'un langage de description du processus de développement, apte à décrire un modèle du processus. Ce modèle permet la génération d'un environnement spécifique à chaque projet de développement.

Par la suite, nous présenterons plusieurs projets de recherches, classés en trois groupes, selon la nature des environnements de développement qu'ils mettent en œuvre. Bien que la plupart de ces projets utilisent, pour des objectifs distincts ou non, une notion de métalangage et de méta-modèle, notre classification est indépendante de la nature du métalangage.

La liste des projets présentés ici n'est pas exhaustive. Les travaux de Kelly et Smolander [Kelly, 1995] sur les méta-outils et de Karrer et Scacchi [Karrer et Scacchi, 1995] sur l'intégration d'outils et la modélisation du processus de développement constituent un répertoire complémentaire de projets présentés.

1. Intégration d'outils

Une tendance commune aux outils de développement actuels est la possibilité de les combiner pour automatiser complètement le processus de développement. Dans cette optique, il serait possible de créer un environnement de développement par la combinaison d'outils de conception, d'un référentiel et d'un moteur d'interprétation avec un modèle du processus de développement. L'intégration d'outils peut être réalisée en cinq niveaux [Sommerville, 1995], qui concernent la plate-forme, les données, la présentation, le contrôle et le processus de développement :

1. Le premier niveau d'intégration concerne l'utilisation d'une plate-forme logiciel commune et est le niveau de base d'intégration ;

2. L'intégration de données permet aux outils d'échanger des informations, par l'intermèdiaire d'un format de fichier commun ou d'un référentiel de données ;

3. L'intégration de la présentation implique l'utilisation d'une interface commune d'interaction avec les concepteurs ;

4. L'intégration du contrôle permet aux outils d'activer et de contrôler l'exécution d'autres outils. Ce genre d'intégration profite de certaines facilités du système d'exploitation, comme les enchaînements d'entrées et de sorties (*pipe*) d'Unix ou le pilotage d'applications (AppleScript) de MacOS ;

5. Le dernier niveau d'intégration concerne le guidage des outils par un modèle explicite du processus de développement et par son moteur de processus associé.

Les efforts d'intégration d'outils ont suivi deux directions différentes. La première suit l'établissement de standards, auxquels les outils doivent adhérer. La seconde direction, celles des canevas d'environnement, cherche à fournir des services de base d'un environnement de développement, comme la gestion d'objets (SGO), d'interface utilisateur (SGIU) et du processus de développement (SGPD). Ces services fournissent des moyens de changer l'état courant des éléments du développement.

Les SGO constituent une base d'objets persistants, complémentaire aux systèmes de fichiers des systèmes d'exploitation. Les SGIU fournissent des mécanismes de base pour définir des interfaces utilisateur et associer des objets décrits graphiquement avec les actions et les objets contenus dans un environnement. La plupart des SGPD se servent de la gestion de tâches des systèmes d'exploitation. Toutefois, les SGBD fournissent des mécanismes plus puissants de contrôle, grâce à des services de modelage.

Contrairement aux méta-outils, qui demandent la traduction d'une spécification en un environnement, aucune transformation n'est nécessaire dans les canevas d'environnements. Ici, les spécifications font directement partie de l'architecture d'un environnement, elles ne sont pas transformées.

1.1 Standards

IRDS [Goldfine et Konig, 1988] est un standard ISO de définition de référentiels de données sous la forme de schémas relationnels, accessible par des requêtes SQL. Un référentiel IRDS est divisé en un niveau de définition et un niveau IRD. Ce dernier stocke les données du développement d'une application, sous la forme de tables. Le format de ces tables est représenté par un modèle Entité/Association et stockés dans le niveau de définition.

CDIF [EIA, 1990] est une proposition d'un format commun d'échange de données entre outils de développement. CDIF utilise un métalangage permettant de décrire le format de stockage d'un outil et d'importer ses données, ce qui lui confère une grande flexibilité. Le métalangage est une variante du langage Entité/Association.

L'acceptation de ces standards, ou d'autres comme le AD/Cycle de IBM [IBM, 1990], est encore très réduite. D'une part, la possibilité d'échange de données entre différents outils n'est pas particulièrement appréciée par l'industrie, qui craint de voir sa clientèle captive migrer vers un outil concurrent. D'autre part, l'existence de différents standards ne favorise pas l'adoption d'un standard universel.

1.2 Canevas d'environnements

L'utilisation d'un canevas disposant de plusieurs services de base est une option d'intégration des outils non conformes à un standard. SoftBench [Cagan, 1990] de Hewlett-Packard, par exemple, est un cadre d'intégration d'environnements de développement fonctionnant sous Unix. SoftBench préconise l'intégration d'outils par l'utilisation d'un bus logiciel, contrôlé par un serveur d'échange de messages, BMS (Broadcast Message Service).

Les outils étrangers sont amenés au standard SoftBench grâce au langage EDL, qui permet la description d'interfaces utilisateur et les messages envoyés à BMS lors du déclenchement d'un événement par l'utilisateur. SoftBench dispose aussi d'un moteur d'exécution de processus, SynerVision.

PCTE (Portable Common Tool Environment) [Campbell, 1988 ; Thomas, 1989] est un canevas d'environnement qui propose plusieurs outils d'accès à un référentiel de données commun. Le référentiel met en œuvre plusieurs tables permettant de stocker des schémas de définition d'objets. En plus, PCTE dispose d'un système de gestion de processus qui permet l'activation et le monitorage d'un processus. PCTE dispose aussi d'un ensemble de primitives de gestion de l'interface utilisateur.

2. Méta-outils

Les méta-outils [Kelly, 1995 ; Loucopoulos et Theodoulidis, 1992 ; Martiin *et al.*, 1993] proposent des moyens rapides de génération d'un environnement de développement servant d'outillage à une méthode de développement. Plus précisément, ils intègrent :

(1) Un langage de méta-modélisation qui, combiné avec d'autres spécifications, permettent la création d'un modèle de construction, i.e. la description de l'environnement souhaité ;

(2) Un canevas d'environnement, spécifiques au méta-outil ou disponibles sur le marché (voir § 1.2) ;

(3) Un mécanisme de transformation, permettant la génération de cet environnement. Généralement, la transformation consiste en l'adaptation (ou personnalisation) du canevas, plutôt que la génération effective des outils.

Les premières propositions de méta-outils ont émergé au début des années 80, avec SEM [Teichrorew *et al.*, 1980], qui proposait une version textuelle du langage Entité/Association comme la base de son environnement. Les modèles de construction gardés dans le ISLDS (Information Systems Language Definition System) de SEM représentaient les langages de modélisation, dont les modèles étaient stockés dans le SEMS (Systems Encyclopaedia

Management System). Cette distinction entre un niveau de méta-modélisation (ISLDS) et un niveau d'instances (SEMS) est une caractéristique qui a influencé tous les environnements actuels de méta-modélisation.

Les méta-outils introduisent un nouvel acteur dans le processus de développement, l'ingénieur de méthodes, dit aussi « méta-concepteur ». C'est lui qui se charge de la conception et de la modélisation des langages de modélisation qui seront supportés par l'environnement généré.

Des efforts récents d'évaluation de méta-outils [Karrer et Scacchi, 1993 ; Kelly, 1995 ; Loucopoulos et Theodoulidis, 1992 ; Martiin *et al.*, 1993] énumèrent plusieurs outils et proposent des caractéristiques souhaitables dans l'architecture d'un tel outil, que nous énumérons ci-dessous :

1. **Langage de méta-modélisation.** L'ingénierie du méta-modèle de construction doit être rapide et de haut niveau. La plupart des outils actuels répondent à ce besoin par l'utilisation d'interfaces graphiques et d'un langage de méta-modélisation connu des informaticiens, et donc d'apprentissage rapide.

2. **Multiplicité de notations.** L'outil de méta-modélisation, ainsi que les environnements générés, sont utilisés par des utilisateurs de différents niveaux de connaissances. Il doit d'une part supporter différents paradigmes d'édition de modèles et de méta-modèles (i.e. permettre des différentes vues) comme les éditeurs de réseaux sémantiques, de textes ou de matrices. D'autre part, il doit supporter plusieurs niveaux d'utilisateurs, offrant des différentes possibilités à des utilisateurs de différents niveaux de connaissances.

3. **Support à l'évolution.** D'une part, l'outil doit supporter la méta-modélisation incrémentale, i.e. la prise en compte immédiate, par l'environnement de modélisation, des changements effectués dans le modèle de construction. D'autre part, les environnements de modélisation doivent supporter l'évolution du système informatique développé (*versionning*).

4. **Simplification de modèles.** Les outils composants l'environnement de modélisation généré doivent supporter des techniques de simplification de l'édition, comme la composition (un objet contenant d'autres) ou l'explosion d'objets (des objets liés à des modèles).

5. **Intégration de modèles.** L'outil doit permettre la vérification de la consistance horizontale et verticale des différents modèles. Par consistance verticale, on entend la vérification entre les différents modèles d'un même niveau technologique, indépendamment des langages utilisés pour les exprimer. La consistance horizontale est vérifiée entre des modèles de différents niveaux technologiques, par exemple, entre un modèle Entité/Association et son modèle relationnel correspondant.

6. **Adaptation à l'environnement existant.** L'environnement de modélisation généré doit s'adapter aux autres environnements déjà existants.

Dans les sections suivantes, nous présentons plusieurs projets de recherche mettant en œuvre des méta-outils. Cette présentation est précédée par une introduction aux langages de méta-modélisation.

2.1 Langages de méta-modélisation

La similitude entre la méta-modélisation et la modélisation de systèmes [Harmsem *et al.*, 1994] a motivé l'utilisation des langages de modélisation déjà existants. Par ailleurs, Sjaak Brinkkemper

[Brinkkemper, 1990] défend que n'importe quel langage de modélisation conceptuel peut aussi servir comme langage de méta-modélisation. Les travaux actuels sur la méta-modélisation se divisent en deux principales approches (les études de [Harmsen et Saeki, 1996] et [Tolvanen *et al.*, 1996] utilisent une classification semblable) :

1. La modélisation des outils de modélisation (orienté données);

2. La modélisation du raisonnement de conception (orienté processus).

Les langages utilisant la première approche sont, pour la plupart, basés sur le langage de modélisation Entité/Association [Chen, 1976], étendu par des contraintes d'intégrité propres aux langages de modélisation. C'est le cas, par exemple, de ISLDS [Teichrorew *et al.*, 1980], Infokit [Atzeni et Torlone, 1993], GOPRR [Kelly *et al.*, 1996 ; Smolander, 1991], EARA [Sorenson *et al.*, 1988] et plus récemment de MOF [OMG, 1997a].

D'autres, comme PSM [Bommel *et al.*, 1991] utilisent NIAM [Nijssen et Halpin, 1989], un langage de modélisation plus formel. Ce formalisme plus important simplifie la création de langages de manipulation de modèles, comme Lisa-D [Hofstede, 1993].

Finalement, quelques travaux utilisent aussi des langages à objets, comme Object-Z [Saeki et Wenyin, 1994] ou Telos [Mylopoulos *et al.*, 1990], ou des réseaux de Petri [Bézivin, 1995].

Les langages de la deuxième approche essayent de modéliser le processus de développement logiciel. Task Structured Diagrams [Wijers, 1991], HFSP [Katayama, 1989], Merlin [Emmerich *et al.*, 1991], MASP [Benali *et al.*, 1989 ; Oquendo *et al.*, 1992] et Articulator [Mi et Scacchi, 1990] sont des exemples de langages de méta-modélisation orientés processus.

Quelques langages de méta-modélisation échappent à cette classification. MEL [Harmsen et Brinkkemper, 1995], par exemple, utilise une approche hybride. A. Jordan [Jordan et Davis, 1991] propose une approche par spécialisation, utilisant un langage de méta-modélisation extrêmement riche, composé de plusieurs constructeurs.

2.2 MetaEdit+

MetaEdit+ [Smolander *et al.*, 1991] est un outil de création et de modélisation de méthodes de développement. Il fait partie du projet MetaPHOR des Universités de Jyväskylä et Oulu en Finlande, associées à MetaCase Consulting Oy.

MetaEdit+ a été construit autour du langage de méta-modélisation GOPRR *(Graph, Objects, Property, Role and Relationship*), une variante du langage Entité/Association étendue par le constructeur « Graph », un sous-graphe d'un méta-modèle, qui s'attache à un objet, un rôle ou une association. Ce constructeur permet plusieurs niveaux d'édition d'un constructeur.

Les méta-modèles, ainsi que les concepts modélisés sont stockés dans un système de gestion d'objets, le Repository, implémenté avec ArtBase Object Manager. Ce système permet l'édition simultanée d'un modèle, bien que l'absence d'un bus logiciel limite la prise en compte des modifications externes à un outil.

Le cœur de l'environnement généré est le MetaEngine qui sert d'interface entre le Repository et les outils d'édition, permettant une indépendance de représentation. En effet, la multiplicité de notations supportées est un apport important de MetaEdit+. Un modèle peut être représenté et modifié sous la forme d'un diagramme, d'une matrice ou d'un ensemble de listes (browser).

MetaEdit+ a été développé en Smalltalk VisualWorks.

2.3 MetaView

MetaView [Sorenson *et al.*, 1988] de l'Université du Saskatchewan (Canada), peut être vu comme une version restreinte de MetaEdit+. Ici, un environnement de développement est décrit grâce à EDL (Environment Definition Language). EDL utilise une notation textuelle, bien qu'une extension graphique, EARA [Boloix *et al.*, 1992], ait été proposée. L'environnement généré consiste en un éditeur de diagrammes.

2.4 PHEDIAS

Le méta-outil PHEDIAS [UMIST, 1995 ; Wang et Loucopoulos, 1995], de l'Université de Manchester, dispose de deux composants permettant la description d'un langage de modélisation, le CSC (*Concept Specification Component*) et le PSC (*Presentation Specification Component*). Le premier met en œuvre une notation graphique d'un langage de méta-modélisation composé par 5 constructeurs : objet, relation binaire, rôle, libellé et spécialisation. Le deuxième, PSC, permet la description des outils de modélisation qui composeront l'environnement de développement généré. Comme cet environnement consiste en un éditeur de diagrammes, le PSC est utilisé pour décrire les symboles et les menus de cet éditeur.

PHEDIAS dispose aussi d'un système de gestion d'objets, mis en œuvre avec le SGBD à objets Ode et chargé de stocker les informations du CSC, du PSC et des environnements de développement. Ces derniers peuvent être construits de deux manières différentes :

1. Version portable : PHEDIAS crée un programme source en C ANSI ; et
2. Version non portable : un canevas d'environnement utilise les informations stockées dans le SGO pour constituer un environnement spécifique.

PHEDIAS a été mis en œuvre en langage C. Grâce à l'utilisation de la bibliothèque XVT (Extensible Virtual Toolkit), ce méta-outil est disponible en plusieurs plates-formes (Unix, Windows, MacOS, OS/2).

2.5 ToolBuilder

ToolBuilder [Alderson, 1991] de l'Université de Sunderland, est un produit commercial de IPSYS Software. Ce méta-outil est composé par un outil de spécification de méthodes, METHS, par un éditeur générique, DEASEL et par un système de gestion d'objets.

La spécification d'un environnement dans METHS se fait en plusieurs étapes. D'abord, un langage de modélisation est représenté par un méta-modèle (appelé data model). Le langage de méta-modélisation utilisé est une extension du langage Entité/Association, étendu par la notion d'attribut calculé (à partir d'autres attributs). Puis, la description des opérations réalisées sur les plusieurs constructeurs du langage est faite dans un modèle de cadres (frame model). Dans la dernière étape, la présentation des constructeurs et le format d'édition d'attributs sont définis dans des outils de présentations.

Après ces spécifications, METHS génère des scripts EASEL. EASEL est un langage interprété de configuration de DEASEL, un éditeur générique de diagrammes. En plus d'une fenêtre d'édition de diagrammes, DEASEL fournit une fenêtre de texte, et permet la navigation hypertexte entre les deux.

D'après le guide d'utilisation de ToolBuilder1, les spécifications sont introduites dans METHS par une notation textuelle, en quatre langages différents : DDL (Data Definition Language), FDL (Format Definition Language), GDL (Graphical Definition Language) et LL (Layout Language). Ces spécifications ne sont probablement pas exhaustives car les scripts générés de configuration de l'éditeur de diagrammes et de l'éditeur textuel doivent être complétés par des commandes EASEL.

2.6 ABACO

L'objectif de ABACO de l'Université de Paris IX [Linval *et al.*, 1992] est de générer des éditeurs de diagrammes spécifiques à un langage de modélisation, à partir du méta-modèle de ce dernier. Les méta-modèles sont décrits dans un éditeur graphique en M2SL (Meta Model Specification Language), une variante du langage OOA [Coad et Yourdon, 1990]. Cet éditeur traduit les modèles M2SL en code du langage SGEN (Synthesizer Generator), qui sont ensuite compilées pour générer un éditeur de diagrammes.

Le code SGEN généré peut être complété par des déclarations qui utilisent le paradigme des grammaires attribuées de Knuth. Grâce à ces déclarations, ABACO peut transformer un modèle en un autre, décrit par un langage de modélisation différent. Les transformations sont réalisées de manière interactive, i.e. toute modification du modèle source affecte automatiquement les concepts concernés du modèle de destination. Lorsque des changements sont effectués sur ce dernier, ils seront respectés lors d'une nouvelle transformation.

2.7 GraphOR

GraphOR [Morejon *et al.*, 1991] utilise un méta-modèle d'un langage de modélisation pour configurer l'éditeur graphique SYLVA [CADWARE, 1988]. Le langage de méta-modélisation utilisé est une variante du langage Entité/Association. Les éditeurs générés peuvent être couplés avec un générateur de code, créés à partir d'une description du langage de programmation cible en BNF.

2.8 Atelier de Méthodes

L'Atelier de Méthodes [Chaplais *et al.*, 1993] d'EDIAT combine un langage de méta-modélisation graphique, appelé graphe de méthode, et des prédicats de premier ordre pour décrire des langages de modélisation et générer leur éditeur de diagrammes en langage Lisp. Les prédicats sont utilisés pour définir la représentation graphique des constructeurs et des contraintes, qui serviront à valider l'intégrité des modèles.

L'Atelier de Méthodes a été développé à l'aide des outils Le Lisp, Aïda et Massaï de Ilog.

2.9 OSMOSIS

OSMOSIS a été développé par le groupe ERTO (Équipe de Recherche en Technologie Objet) de l'Université de Nantes. Il permet de créer des outils destinés à intégrer les différentes étapes du

[1] http://osiris.sunderland.ac.uk/rif/bgtbk/wmwork/www/tbk_1.html

processus de développement d'une application (de l'analyse des besoins à l'implémentation) ainsi que permettre le suivi des transformations entre modèles.

Le langage de méta-modélisation utilisé est une variante des réseaux sémantiques, les « SNETS ». OSMOSIS configure un éditeur de listes, le SBROWSER, à partir des définitions d'un méta-modèle. SBROWSER est aussi l'outil utilisé pour construire un méta-modèle.

OSMOSIS a été développé en Smalltalk VisualWorks.

2.10 TRAMIS

TRAMIS, de l'Université de Namur [Hainaut *et al.*, 1992], diffère des autres méta-outils en deux aspects, son champs d'application et la nature de son langage de méta-modélisation. En effet, il s'intéresse à la création d'outils couvrant les trois phases standards de la conception d'une base de données : l'analyse conceptuelle et les conceptions logique et physique ; ainsi que le processus de développement associé.

Les langages de modélisation sont représentés dans TRAMIS grâce à un langage générique. Ce langage, contrairement aux langages de méta-modélisation, ne sert pas à décrire d'autres langages. Son rôle est de les dépasser, de disposer d'un ensemble extensif de constructeurs de modélisation qui sera spécialisé (ou plutôt restreint) pour définir un autre langage. Le langage générique comprend les constructeurs du langage Entité/Association, et des constructeurs informels, qui permettront de décrire certaines statistiques d'une donnée comme sa population ou son utilisation. Les spécialisations du langage générique sont utilisées pour configurer un éditeur de listes.

TRAMIS propose également un ensemble d'outils de transformation entre modèles à trois niveaux : élémentaire, général et orienté modèle. Le premier concerne individuellement les concepts modélisés, ce qui permet de résoudre les ambiguïtés. Le deuxième concerne les concepts modélisés par un même constructeur. Le dernier s'intéresse aux constructeur qui n'ont pas de correspondance dans un des langages.

TRAMIS a été développé en C pour Windows.

3. Modélisation du processus

Grâce à un langage de méta-modélisation de haut niveau, un méta-outil permet la création rapide et peu onéreuse d'un outil de développement. Toutefois, il ne permet pas le changement du processus de développement qui sera total ou partialement automatisé par les outils générés. Par processus de développement, nous entendons l'ensemble d'activités qui permettent la transformation des besoins de l'utilisateur en un système informatique.

Une alternative aux méta-outils est l'utilisation d'outils permettant la modélisation et l'exécution du processus de développement. Un modèle du processus décrit les objets actifs et inactifs présents durant le développement. L'exécution de ce modèle peut informer un développeur sur ce qu'il doit faire après l'accomplissement d'une tâche, coordonner le travail de plusieurs développeurs et garder une trace des tâches réalisées. L'analyse des ces dernières et du modèle de processus permet aux outils de déterminer quel est le meilleur ordre d'exécution de tâches pendant le développement et aussi quelles sont les tâches qui peuvent être exécutés en parallèle.

Les outils de modélisation du processus utilisent souvent des règles de production et des moteurs d'inférence comme moyens de spécification et exécution d'un processus de développement. Dans la plupart des cas, ils génèrent ou personnalisent des environnements de développement basés sur la connaissance.

Lors du développement de MétaGen, nous avons étudié quelques outils centrés sur la modélisation de processus. Ces outils sont présentés dans l'Annexe B de ce document. Une étude similaire a été effectuée par Patrick Yung [Young, 1994] (page 54) dans sa thèse de doctorat.

III. Architecture et fonctionnalités

La conception de MétaGen ressemble à celle des méta-outils présentés précédemment. En effet, il peut être divisé en trois modules distincts : (i) des outils de méta-modélisation ; (ii) un canevas d'environnement et (iii) un mécanisme de transformation. Nous détaillons ces trois modules par la suite.

Les outils de méta-modélisation permettent la création d'un modèle de construction, qui représente l'environnement de développement souhaité. Actuellement, le modèle de construction est composé à partir de quatre outils : l'éditeur PIR3, le méta-éditeur, l'éditeur de règles et l'éditeur de parcours. Ces outils seront décrits dans les sections suivantes.

Le canevas d'environnement est composé par un éditeur de diagrammes personnalisable (appelé Editeur MG), par un éditeur de listes (browser), par un moteur d'interprétation, par un moteur d'inférences et par une structure sous-jacente générique.

Le mécanisme de transformation est responsable de la configuration du canevas d'environnement, à partir des définitions contenues dans le modèle de construction, de manière à créer l'environnement souhaité.

L'élaboration d'un modèle de construction (Figure 16) commence par la création d'un ou de plusieurs méta-modèles, chacun représentant la syntaxe d'un langage de modélisation. Les méta-modèles sont créés par un éditeur graphique ou par un browser, basés sur le langage de méta-modélisation PIR3, qui est présenté au § 1, ci-dessous.

Ensuite, des règles de production et des parcours d'interprétation sont définis, en fonction des méta-modèles, au sein d'éditeurs spécialisés. Ces définitions permettront, plus tard, la manipulation de modèles ou plus précisément, leur validation, leur transformation ou leur exécution.

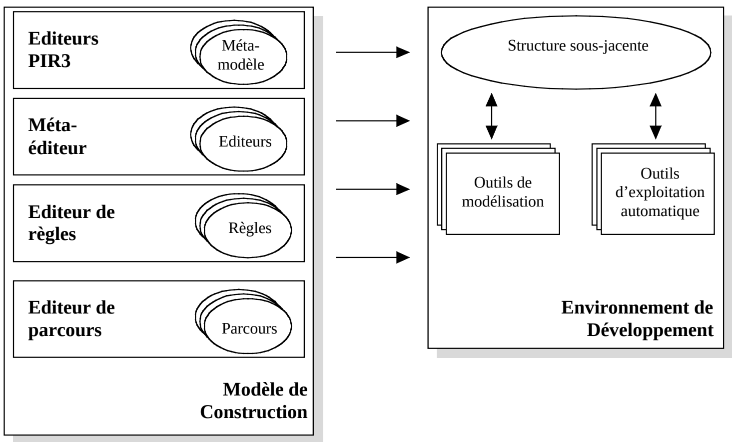

Figure 16 - Architecture de MétaGen

La génération des outils de modélisation d'un langage est pilotée par un outil de définition d'éditeurs, appelé « méta-éditeur ». Le méta-éditeur permet aussi de créer des dialogues d'édition des concepts modélisés par les constructeurs de ce langage. Les outils de modélisation serviront d'interface entre les concepteurs et la structure sous-jacente, responsable du stockage des concepts modélisés. Celle-ci est spécifique à chaque langage de modélisation et est générée automatiquement à partir du méta-modèle de ce langage.

Les outils de modélisation serviront aussi à piloter les outils d'exploitation automatique de modèles, définis par les règles et les parcours. Ces outils interagissent avec la structure sous-jacente.

La Figure 17 représente le schéma d'utilisation de MétaGen, décrivant les actions de composition du modèle de construction (de 1 à 4) et d'utilisation d'un environnement de développement (5-8), les outils nécessaires à la réalisation de chaque action et les relations de dépendance entre elles.

La construction d'un environnement commence par la méta-modélisation (1), i.e., la création d'un modèle PIR3 de chacun des langages de modélisation qui seront proposés par cet environnement. Après cette étape, trois actions sont réalisables : la création de son outillage (2), la création des bases de règles de transformation et de validation (3) ou la définition d'un parcours d'interprétation (4).

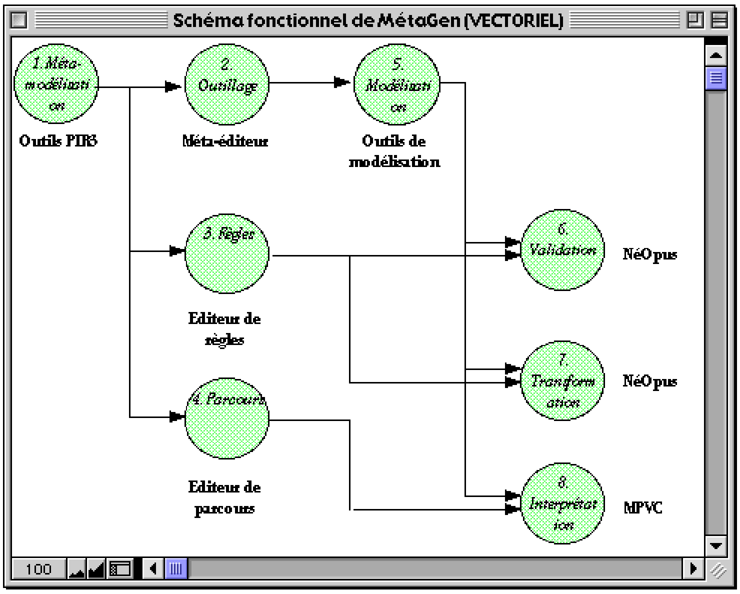

Figure 17 - Schéma fonctionnel de MétaGen

L'utilisation de l'environnement généré commence par la modélisation (5), en employant l'outillage créé par (2). Trois actions peuvent être appliquées à un modèle : la validation, la transformation ou l'interprétation. Comme la définition des bases de règles et des parcours d'interprétation dépend uniquement de l'existence d'un méta-modèle et qu'elles s'appliquent uniquement sur des modèles, leur définition est souvent réalisée après la création de quelques modèles en exemple.

Par la suite, nous détaillerons les fonctions principales de MétaGen en quatre sections. Dans la première, nous présentons PIR3. Son outillage sera et la génération d'une structure sous-jacente seront présentés à la deuxième section. Ensuite, nous présenterons les règles de production et les définitions de parcours. Le méta-éditeur et la génération des outils de modélisation seront présentées ensemble dans la troisième section. L'environnement de développement généré sera présenté dans la quatrième et dernière section.

1. Le langage de méta-modélisation PIR3

PIR3 est une extension de IR3 décrit par [Sahraoui, 1995], un langage de méta-modélisation dont l'objectif est de permettre la description des aspects structuraux (vocabulaire et syntaxe) d'un langage de modélisation (la sémantique des constructeurs est définie ultérieurement, par les transformations). Contrairement aux autres langages qui mettent en œuvre une approche de méta-modélisation orienté données, il ne permet pas la spécification de détails des outils de support du langage de modélisation décrit, comme les symboles de représentation ou les libellés utilisés par un éditeur graphique.

PIR3 est formé par cinq constructeurs de base : Plan, Méta-individu, Méta-relation, Rôle et Rubrique. Par la suite, nous présenterons ces constructeurs et un exemple de méta-modélisation avec PIR3. Le constructeur Plan, défini en fonction des autres constructeurs, sera présenté en dernier.

Un méta-individu est un élément de représentation identifiable (constructeur) d'un langage de modélisation. Chaque méta-individu se caractérise par une seule propriété, son nom.

Si les méta-individus représentent les constructeur d'un langage, les méta-relations, elles, représentent les combinaisons possibles entre ces constructeur. Elles sont binaires (ne lient que deux méta-individus) et toujours orientées.

Un rôle décrit le comportement d'une méta-relation, contrairement aux rôles du langage Entité/Association, qui décrivent la participation d'une entité dans une association. Il peut être partagé entre plusieurs méta-relations.

Les rubriques sont les seuls constructeurs typés de PIR3. Les méta-modèles, les méta-individus, les méta-relations et les plans possèdent une liste de rubriques. Chaque rubrique est caractérisée par un nom, un type et, en fonction de ce dernier, une taille. Une dernière propriété, appelée nature, permet aux rubriques d'assumer plusieurs valeurs au même temps.

Un plan est un ensemble de méta-relations et de méta-individus, un sous graphe d'un méta-modèle. Il peut être associé à un méta-individu ou à une méta-relation.

Les Individus et les Relations sont des occurrence des méta-individus et des méta-relations dans un modèle représenté dans le langage décrit par le méta-modèle contenant ces méta-individus et ces méta-relations. La relation existante entre un méta-individu et un individu, et entre une méta-relation et une relation, est la même que celle entre un Type d'entité et une Entité [Chen, 1976]. On dira qu'un individu est de type MI(n) ou qu'une relation est de type MR(n) s'ils sont une occurrence de MI(n) ou de MR(n).

Finalement, il est possible d'établir trois sortes de contraintes d'intégrité dans un méta-modèle. Chaque méta-relation possède deux cardinalités (minimale et maximale) d'entrée et de sortie, qui servent à contraindre son utilisation. Des contraintes d'inclusion et d'exclusion (prohibant ou exigeant l'existence d'une relation en fonction d'une autre) peuvent être établies entre plusieurs méta-relations. Il est aussi possible de définir une contrainte référentielle entre une rubrique d'un méta-individu et une méta-relation.

1.1 Construction d'un méta-modèle

Prenons l'exemple d'un modèle simplifié du jeu d'échec (présenté dans [Rumbaugh *et al.*, 1991]), représenté comme un diagramme d'état. Dans la Figure 18, 'Début,' 'Les noirs jouent', 'Victoire des Blancs', etc. sont des états, et les flèches entre eux, des transitions. Chaque transition est associée à un événement, qui est son libellé ('mouvement des noirs', 'échec et mat', etc.). Notons qu'il existe trois types d'état : initial, intermédiaire et final ; et que chaque type a ses propres contraintes (état de début, par exemple, n'a aucune transition d'entré, inversement au état final, qui n'a aucune transition de sortie).

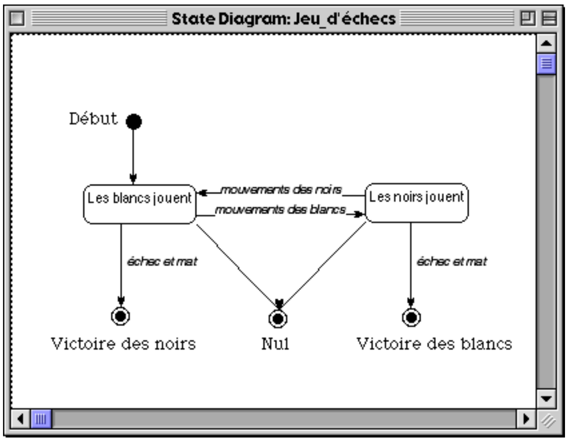

Figure 18 - Diagramme d'état simplifié d'un jeu d'échecs

Le méta-modèle présenté dans la Figure 19 est la représentation de ce langage de modélisation (le diagramme d'état) par PIR3. Il est composé par deux constructeurs de modélisation (méta-individus) principaux : état et transition. Etat et Transition sont liés par deux méta-relations. Chacune porte un libellé (e.g. 'input') qui représente son rôle, et deux contraintes de multiplicité qui indiquent chacune le nombre maximal et minimal d'occurrences de cette méta-relation dans un modèle d'une application (par exemple, un état peut avoir entre aucune ou plusieurs transitions de sortie). 'Evénement' est représenté comme une rubrique de 'Transition', aussi bien que 'nom' et 'type' sont des rubriques de 'Etat'. 'Type' est une rubrique particulière, qui peut seulement assumer trois valeurs possibles ('initial', 'intermédiaire' et 'final').

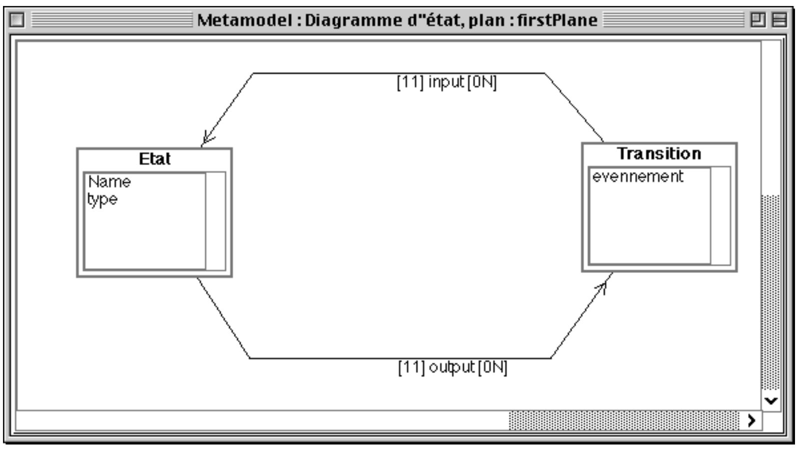

Figure 19 - Méta-modèle PIR3 d'un diagramme d'état

2. L'outillage PIR3

Comme vu précédemment, la composition du modèle de construction commence par la création des méta-modèles. Ceux-ci représentent la syntaxe des langages de modélisation qui serviront de base à l'environnement généré. Pour cela, MétaGen propose deux outils de méta-modélisation, basés sur PIR3 : un éditeur graphique et un browser (Figure 20).

L'éditeur graphique fournit une vision globale d'un méta-modèle, permettant ainsi une lecture rapide. Il permet de diviser un méta-modèle en plusieurs plans, qui représentent ses multiples facettes, ce qui peut simplifier son interprétation. Toutefois, cet outil ne permet pas l'affichage direct des rubriques des méta-modèles, des méta-individus ou des méta-relations.

Le browser fournit un accès rapide à tous les méta-individus présents dans un méta-modèle, et permet la visualisation, en même temps, de toutes les méta-relations et rubriques d'un même méta-individu. C'est un outil de modélisation rapide, qui toutefois ne permet qu'un aperçu limité des méta-relations, ce qui limite la compréhension globale d'un méta-modèle. Tout changement opéré sur un méta-modèle est automatiquement reporté à tous les outils de méta-modélisation ouverts.

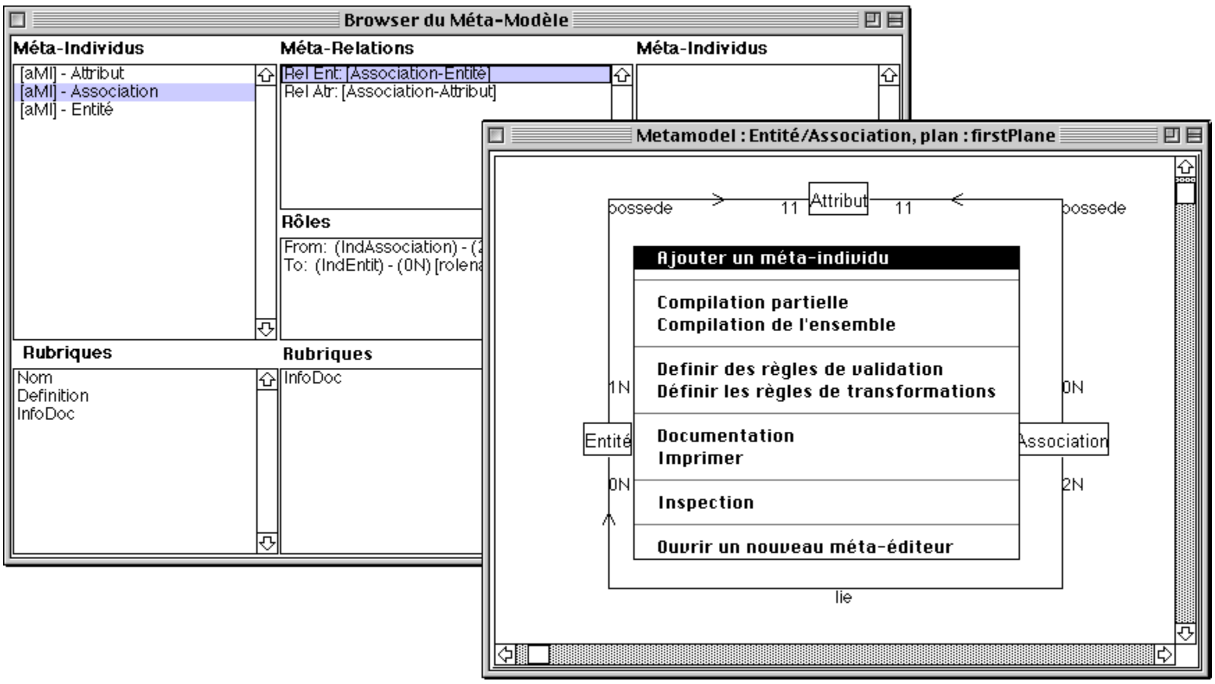

Figure 20 - Le browser et l'éditeur PIR3

Le browser de méta-modèles (Figure 20, en haut et à gauche) affiche une liste contenant tous les méta-individus d'un méta-modèle (dans ce cas, du langage Entité/Association). Lorsqu'un élément de cette liste est sélectionné (dans la figure, Association), le browser affiche la liste de ses rubriques (nom, définition, etc.) et énumère ses méta-relations. Lorsqu'une méta-relation est sélectionnée, le browser affiche ses rôles, ses rubriques et le deuxième méta-individu de cette méta-relation.

L'éditeur graphique (Figure 20, à droite), se présente sous la forme d'un éditeur de diagrammes. Les méta-individus sont représentés par des rectangles, et les méta-relations par des arcs orientés. Ces deux outils disposent de plusieurs menus contextuels, qui permettent le déclenchement de différentes actions, comme la création d'un méta-individu ou la sauvegarde d'un modèle. Ces actions seront expliquées dans la section suivante.

2.1 Méta-modèles, méta-individus et méta-relations

Lors de la création (ou de la modification) d'un constructeur de modélisation (méta-individu, d'une méta-relation ou d'un méta-modèle), indépendamment de l'outil de modélisation utilisé, des boîtes de dialogues sont affiché sur l'écran, permettant l'édition des propriétés du constructeur créé (Figure 21). Comme les trois boîtes de dialogues utilisées sont similaires, elles seront présentées en même temps.

Tout d'abord, tous les dialogues d'édition affichent un bouton de documentation (en bas, à gauche). Il donne accès à certaines informations descriptives des constructeurs de modélisation représentés en PIR3, qui peuvent être utilisées pour documenter un projet. En effet, MétaGen peut générer un document Word à partir de ces informations.

Les dialogues ne possèdent que deux champs d'édition communs (qui sont, par ailleurs, les seules propriétés d'un méta-individu) : nom et rubriques. Le premier sert d'identifiant du constructeur. Le deuxième est en réalité une liste, qui dispose d'un menu contextuel permettant l'ajout ou la suppression de rubriques. Les rubriques serviront, lors de la modélisation, à décrire les concepts représentés par les constructeurs.

Les méta-modèles possèdent deux autres propriétés, abréviation et description. La première sera utilisée lors de la transformation du méta-modèle (voir § 2.4). La deuxième garde une description succincte des caractéristiques du langage de modélisation représenté.

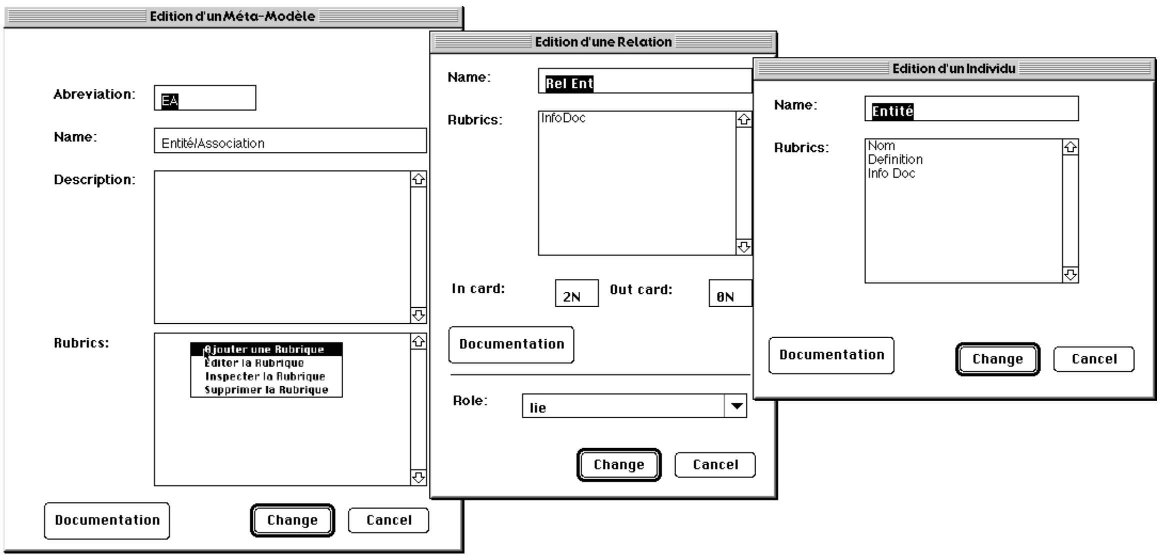

Figure 21 - Edition d'un méta-modèle, d'une méta-relation et d'un méta-individu

Les méta-relations disposent de trois propriétés supplémentaires : les cardinalités d'entrée et de sortie et le rôle. La cardinalité d'entrée établie le nombre (minimum et maximum) d'individus de type cible (ou de type méta-individu cible) qui pourront être liés à un individu de type source. Une association, par exemple, devra être liée à au moins deux entités.

La cardinalité de sortie établie l'inverse, i.e., le nombre d'individus de type source qui pourront être liés à un individu de type cible. Le rôle d'une méta-relation est choisi à partir d'une liste prédéfinie de rôles. Cette liste dispose d'un menu contextuel permettant l'inclusion de nouveaux rôles. Les rôles sont communs à la totalité d'un projet.

2.2 Rubriques

La liste des rubriques d'un constructeur de modélisation est accessible à partir des dialogues d'édition et du browser de méta-modèles. Le dialogue d'édition (ou de création) d'une rubrique est toujours le même, indépendamment du constructeur auquel elle est liée (méta-modèle, méta-individu ou méta-relation).

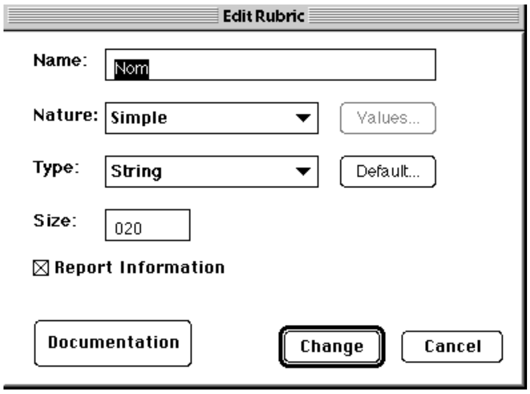

Figure 22 - Edition d'une rubrique

Le dialogue d'édition d'une rubrique est présenté par la Figure 22. Trois propriétés sont directement modifiables par ce dialogue : le nom, la nature, et le type. Le nom est l'identifiant de la rubrique. La nature est effectivement une liste d'options, qui permet d'établir si la propriété décrite par la rubrique est une liste, un sous-ensemble d'une liste, une option d'une liste ou une valeur élémentaire. Le type est aussi une liste d'options permettant de décrire le type de la propriété décrite. Les types disponibles sont : symbole, numéro, date, texte, heure, objet, chaîne et booléen.

2.3 Contraintes

En plus des contraintes de cardinalité, définies au niveau d'une méta-relation, l'outillage PIR3 permet la définition de deux autres types de contraintes, référentielles et fonctionnelles, définies au niveau d'un méta-individu. Elles s'appliqueront donc aux individus, lors de la modélisation. Les dialogues de définition de ces contraintes (Figure 23) sont accessibles à partir des menus contextuels des méta-individus. Nous décrirons par la suite ces deux types de contraintes.

L'objectif d'une contrainte référentielle d'un individu est d'interdire ou d'obliger un type relation en fonction de la valeur d'une de ses propriétés. La définition d'une contrainte référentielle est réalisée en deux parties : condition et action. La partie condition s'établie en fonction d'une rubrique, à laquelle un prédicat (=, >, <) et une valeur sont appliqués. Lorsque cette condition s'avère vraie, elle déclenche une action : l'interdiction ou l'obligation d'un type de relation.

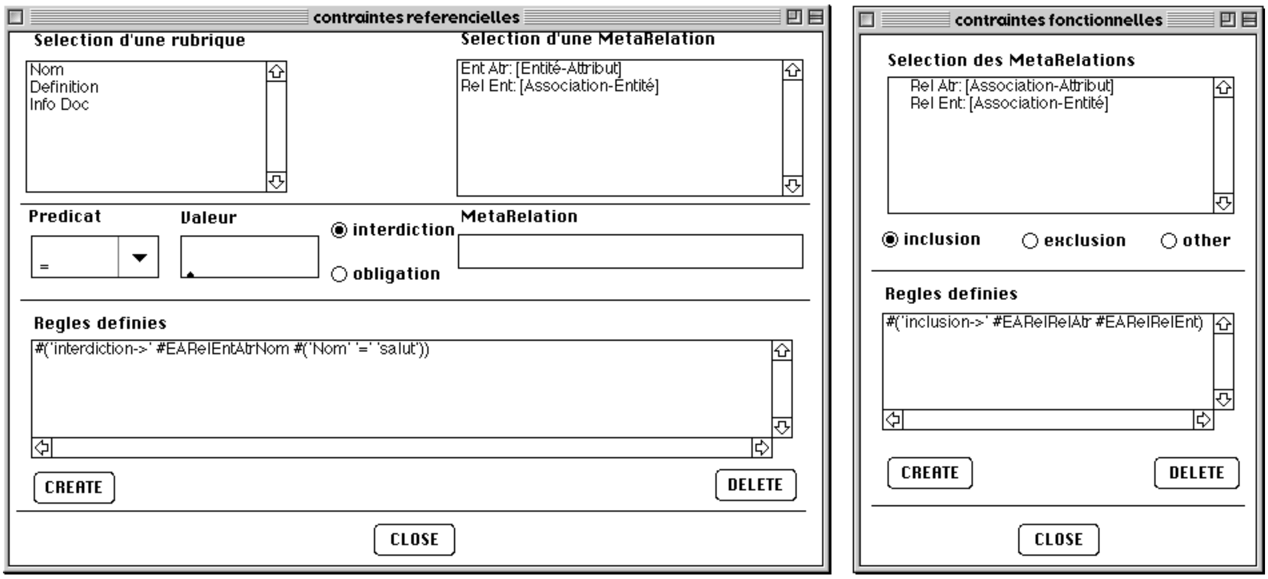

Figure 23 - Edition de contraintes référentielles et fonctionnelles

Les contraintes fonctionnelles (ou d'inclusion et d'exclusion) d'un individu ont aussi comme objectif d'interdire ou obliger un type de relation, mais en fonction d'un ou de plusieurs types de relation. Elles sont définies par un ensemble de méta-relations et par une condition d'inclusion ou d'exclusion entre elles.

2.4 Génération de la structure sous-jacente

Après la création d'un méta-modèle d'un langage de modélisation, il est nécessaire de le compiler. La « compilation » est en réalité la création d'une structure sous-jacente propre à ce langage. Cette structure est utilisée pour stocker les concepts qui seront représentés par les constructeurs du langage de modélisation. En même temps, elle permet que ces concepts soient indépendants de leur représentation, et donc, utilisables par différents outils.

Pour créer une structure sous-jacente spécifique à chaque méta-modèle, MétaGen spécialise une structure générique déjà existante, appelée SGM, qui assemble les caractéristiques communes à toute structure sous-jacente. La SGM est composée par trois classes principales ('MGModel', 'MGIndividual' et 'MGRelation') et par d'autres classes complémentaires. Elle fournit les fonctions de base à toute structure sous-jacente.

Les fonctions de base comprennent la sauvegarde et la restauration, des méthodes booléennes de liaison d'individus (pour construire des prémisses de règles de production), des méthodes de fabrication de modèles (création, suppression, etc.) et de génération de rapports.

La Figure 24 représente les classes qui composent la structure sous-jacente correspondante au méta-modèle du langage Entité/Association (présenté par la Figure 20). Le méta-modèle, les méta-individus et les méta-relations ont été transformés en des sous-classes de 'MGModel', 'MGIndividual' et 'MGRelation', respectivement. Les trois méta-individus sont transformés en des sous-classes de 'MGIndividu'. Les rubriques sont devenues des variables d'instance des classes générées.

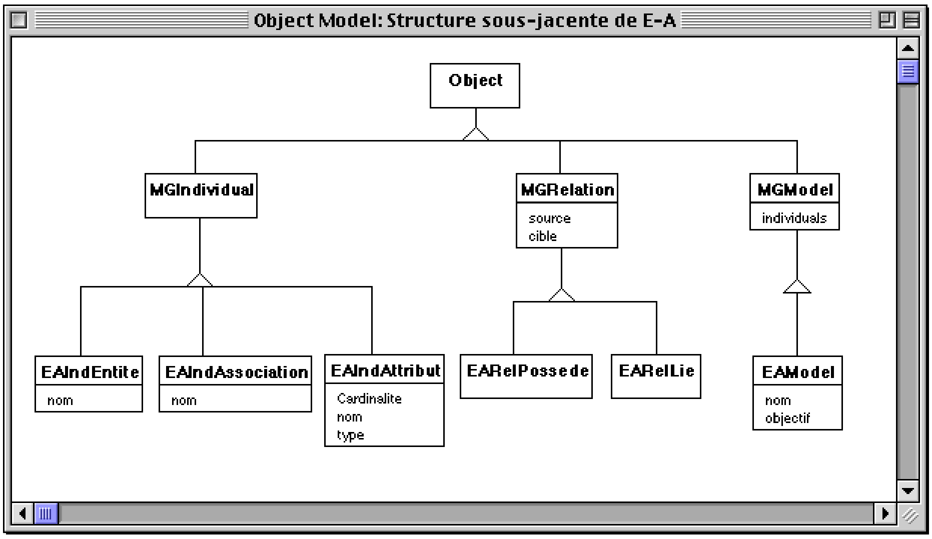

Figure 24 – Structure sous-jacente de Entité/Association

Les noms des classes de la structure sous-jacente et de leurs variables, sont générés à partir de quelques informations présentes dans le méta-modèle, tout en respectant les contraintes de Smalltalk. Les noms des classes sont composés par un préfixe (l'abréviation du méta-modèle) et par les noms des méta-individus ou des méta-relations, précédés par « Ind » et « Rel », respectivement.

MétaGen supporte la méta-modélisation incrémentale, i.e., il accepte des modifications dans le méta-modèle après la génération de l'environnement de développement, et les rend disponibles promptement. Plus précisément, lorsqu'un méta-individu, une méta-relation ou une rubrique sont créés, les constructeurs seront disponibles dans l'environnement de développement. Lors qu'ils sont supprimés, MétaGen les garde dans la structure sous-jacente et n'efface pas les concepts modélisés.

3. La définition de règles et de parcours d'interprétation

Les langages de modélisation décrits dans MétaGen permettent de générer des éditeurs de modèles. Ces modèles sont, *grosso modo*, des graphes orientés dont les nœuds sont des occurrences des individus et les arcs des occurrences des relations. Cette généralité nous a permis de définir un ensemble de primitives de parcours de manipulation des éléments de ces modèles.

Grâce à ces primitives, diverses utilisations des modèles sont possibles. La première est la possibilité d'écrire des règles de validation de modèles. Ces règles sont définies en fonction de méta-modèles et dépendent des constructeurs du langage représenté dans celui-ci. De cette manière, elles peuvent être utilisées par tous les modèles issus de ce langage.

Une seconde utilisation possible est de définir des règles de transformation de modèles. De même que les règles de validation, ces règles sont écrites en utilisant les primitives de parcours et de manipulation. La transformation d'un modèle peut avoir deux types de résultats. Le premier est un autre modèle conforme au même langage ou à un langage différent, comme transformer un modèle décrit en Entité/Association en un modèle décrit en Relationnel. Le second type de résultats peut être du code comme, par exemple, transformer un modèle[1] relationnel en un ensemble de requêtes SQL permettant de créer les tables correspondantes.

Une dernière utilisation possible d'un modèle est de lui associer un parcours d'interprétation. Sur celui-ci on définira des actions à effectuer pour faire des simulations [Heroux et Desevedavy, 1992]. Cette définition de parcours se fait dans MétaGen grâce à un moteur de parcours développé par Philippe Krief [Krief, 1990] : MPVC (Modèle, Point de Vue, Contrôleur). A partir d'un ensemble de primitives, MPVC permet de définir un parcours sur un graphe : à chaque nœud pouvant être parcouru, on associe le nom d'une action. On définit par la suite les « Points de Vue » qui permettent de donner des implémentations aux actions. Ainsi, lors d'un même parcours, on pourra exécuter différentes actions suivant les points de vue que l'on aura choisis. On pourra, par exemple, faire exécuter en même temps le point de vue permettant la visualisation du parcours et un point de vue simulant le modèle.

4. Le méta-éditeur

Le méta-éditeur permet la génération de l'outillage d'un langage de modélisation. Le rôle de cet outillage est de servir d'interface entre les concepteurs et les concepts stockés par la structure sous-jacente. Il permet aussi l'introduction de contraintes supplémentaires à celles présentées précédemment, ainsi que des mécanismes de simplification de modèles. Il se présente sous la forme d'un éditeur graphique (Figure 25, à gauche). Son menu principal propose plusieurs actions, dont l'inclusion des deux éléments principaux, les décorateurs et les esquisses, qui seront présentés dans les paragraphes suivants. Actuellement, le méta-éditeur ne permet que la création d'éditeurs graphiques.

Un décorateur est un composant qui associe certaines propriétés aux constructeurs d'un méta-modèle. Ces propriétés sont spécifiques à un outil de modélisation. Par exemple, lors de la création d'un éditeur de diagrammes Entité/Association, un décorateur est associé au méta-individu 'Entité'. Ce décorateur établira qu'une entité est représentée par un rectangle, contenant son nom à l'intérieur.

[1] Le modèle est employé ici comme synonyme de « schéma », et non comme de « langage », contrairement au sens qui lui est habituellement accordé par la communauté de bases de données.

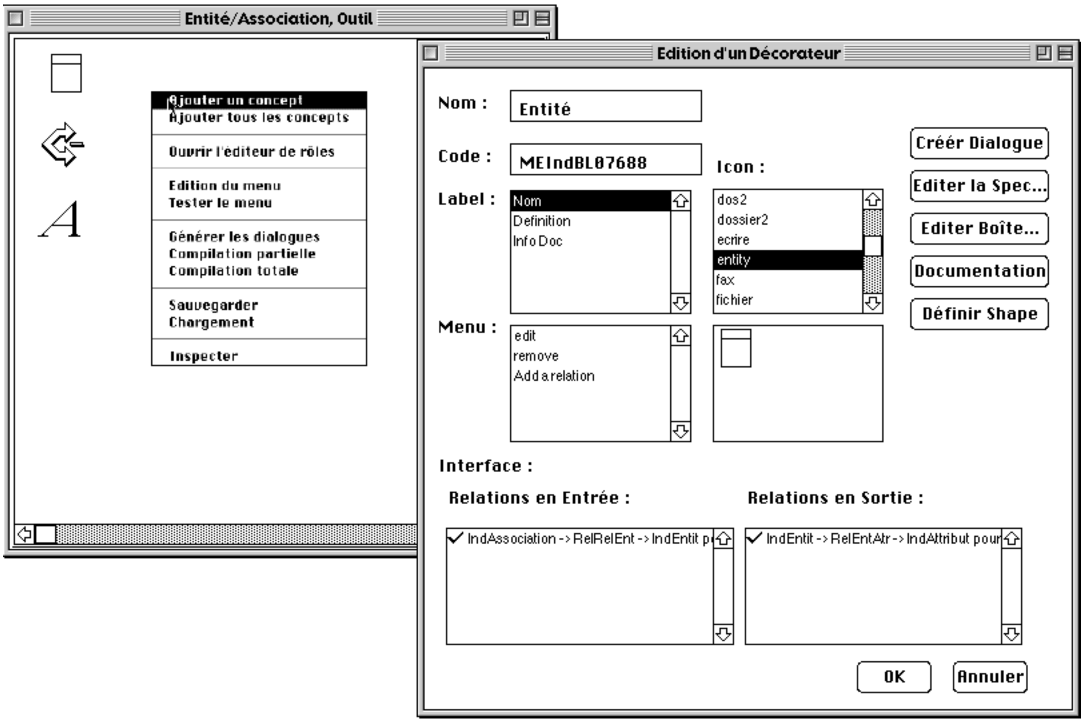

Figure 25 - Le méta-éditeur

La Figure 25 présente le dialogue d'édition qui permet la définition d'un décorateur. Cette définition comprend le choix d'un libellé, parmi les rubriques du méta-individu de référence, le choix d'une icône, l'ajout d'options au menu contextuel, et la sélection des types de relations permis. En même temps, ce dialogue permet la création et la modification d'une boîte d'édition (voir § 4.1). Finalement, il est aussi possible de choisir une figure de représentation, utilisée à la place de l'icône.

Les esquisses sont des raccourcis de construction d'un sous-graphe de modèle. Elles sont composées par un méta-individu maître et par un ou plusieurs méta-individus fils. Lorsqu'on utilisera cette esquisse dans un modèle, l'outil créera, au niveau de la structure sous-jacente, un ensemble d'individus et relations correspondant à la définition de l'esquisse. Toutefois, au niveau de l'outil de modélisation, on ne verra qu'un seul élément, qui regroupera toutes les propriétés des composants de l'esquisse.

Une esquisse est un décorateur capable de contenir d'autres décorateurs. Pour définir cette structure hiérarchique, les esquisses disposent d'un éditeur graphique. En plus de la structure de l'esquisse, l'éditeur permet de définir pour chacun des décorateurs composant l'esquisse, des valeurs qui seront prises par défaut dans l'outil généré.

4.1 Mécanismes de simplification de modèles

Un problème qui apparaît très rapidement lors de la modélisation est l'explosion des concepts qui perturbe la visualisation d'un modèle et empêche une interprétation claire. Une manière d'atténuer ce problème, dans le cas des éditeurs de diagrammes, est d'utiliser des mécanismes permettant de mettre en valeur les concepts les plus importants, selon un point de vue spécifique.

Les éditeurs générés par le méta-éditeur peuvent disposer de quatre mécanismes différents de simplification, qui seront présentés par la suite : les calques, la rétractation, le regroupement et le filtrage. Ces mécanismes sont définis dans le méta-éditeur et appliqués par les éditeurs, et donc, aux individus et relations.

Un calque est une extraction d'un sous-graphe d'un modèle, sélectionné grâce à des règles définies dans le méta-éditeur. La définition d'un calque est faite par le choix d'un ensemble de types de relation autorisés. L'extraction est réalisée par la sélection des relations appartenant à un de ces types, et des individus liés à celles-ci. Il est possible de définir plusieurs calques pour un même modèle, qui peuvent se superposer.

Le mécanisme de rétractation permet de cacher une partie d'un modèle, à partir de la définition des types de relations qui ne seront pas visibles. Quand la rétractation est activée, l'éditeur occulte les relations non visibles et les individus non liés à une relation visible.

Plusieurs individus peuvent être regroupés en un regroupement hiérarchique. Ceci consiste en l'agglutination, par un individu maître, d'autres individus, tout en conservant une représentation des relations vers des individus non agglutinés. Les regroupements ne nécessitent pas de définitions dans le méta-éditeur, il s'applique à tout individu sélectionné à l'intérieur de l'éditeur graphique.

Finalement, les filtres fonctionnent de manière inverse aux calques : on définit les relations non autorisées, qui sont exclues du modèle.

4.2 Autres fonctionnalités

Le méta-éditeur dispose de quelques fonctions complémentaires, qui permettent une meilleure maîtrise de la génération de l'environnement de développement. Nous présentons par la suite deux de ces fonctions, la génération des dialogues d'édition et la configuration des menus contextuels.

i) Génération des boîtes de dialogue

Le méta-éditeur permet la définition d'une boîte de dialogues pour chacun des ses éléments (décorateurs, esquisses). Ces dialogues sont générés automatiquement, en fonction des rubriques du constructeur associé (méta-modèle, méta-individu ou méta-relation), et sont spécifiques à un outil de modélisation.

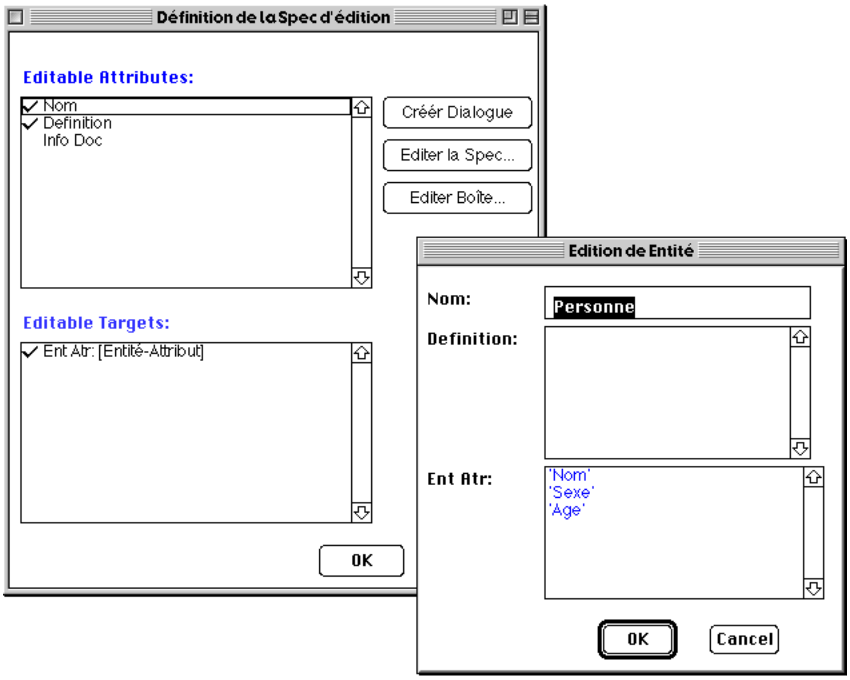

Figure 26 - Génération d'un dialogue d'édition

La Figure 26 présente la définition d'un dialogue d'édition, et le dialogue généré. La définition consiste à choisir un sous-ensemble des rubriques du méta-individu associé, puis les méta-individus en relation directe avec celui-ci. Dans cet exemple, le dialogue généré permet l'édition de deux propriétés d'une entité (nom et définition), et de lui ajouter des attributs (qui sont en réalité d'autres individus).

ii) Les menus contextuels

La dernière fonction du méta-éditeur est d'associer des menus contextuels aux décorateurs et aux éditeurs. Ces menus sont composés par des options générées automatiquement (création et édition des concepts) et par des options définies par le méta-concepteur. Ces dernières sont définies dans le méta-éditeur, et associées à des blocs de commande Smalltalk.

5. L'environnement de développement

L'environnement de développement généré est composé par un ensemble d'outils de modélisation - correspondant aux divers langages de modélisation – et par des mécanismes permettant d'exploiter automatiquement les modèles créés par l'outillage.

Le principal outil de l'environnement de modélisation est l'éditeur de diagrammes, dont les qualités sont définies dans le méta-éditeur. L'éditeur se compose d'un menu, d'une barre d'outils et d'une surface d'édition (Figure 27, à gauche). Grâce aux menus de l'éditeur, un modèle peut être sauvegardé, mis à jour (pour prendre en compte les changements du méta-éditeur), replacé automatiquement ou présenté sous forme d'un browser. La barre d'outils est composée par plusieurs petites icônes, qui représentent la commande de création des individus - chaque décorateur ou esquisse définis dans le méta-éditeur y sont présents.

Chaque individu de l'éditeur dispose d'une poignée de connexion placée en son centre (représentée par un petit carré). En glissant cette poignée vers un autre individu, il est possible de créer une relation entre les deux individus. Si plusieurs types de relation sont acceptables entre les deux individus, l'éditeur demandera un choix. Les relations peuvent aussi être créées en choisissant une option du menu des individus.

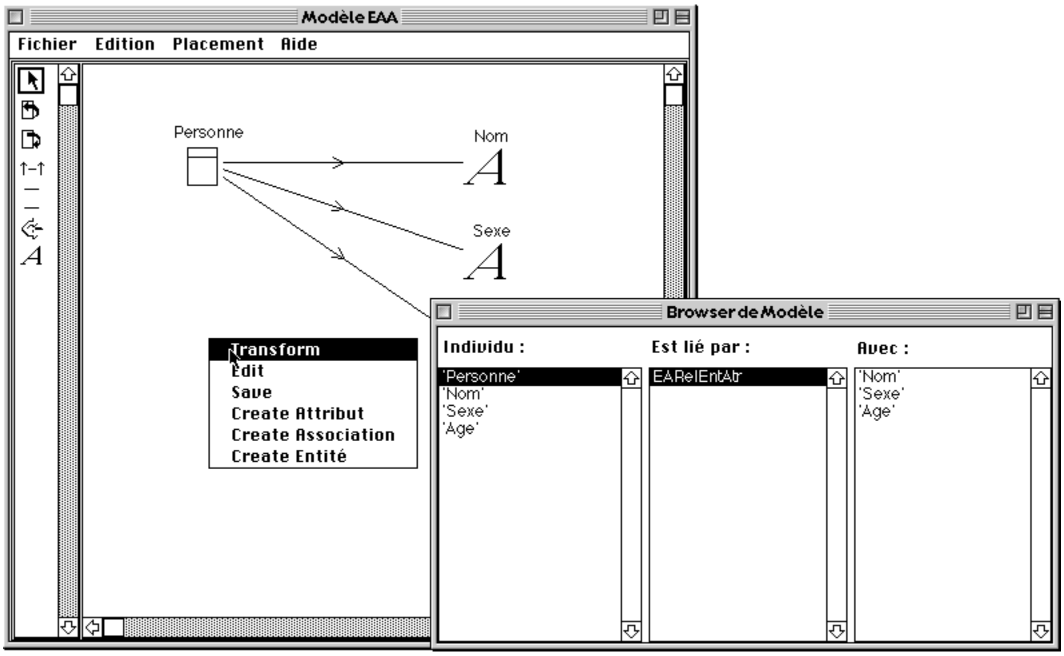

Figure 27 – L'environnement de modélisation

L'éditeur de diagrammes propose un mécanisme de replacement automatique, grâce auquel les modèles générés automatiquement deviennent aussi lisibles qu'un modèle dessiné « à la main » par le concepteur. Deux algorithmes de replacement sont disponibles : aléatoire et par quadrillage. Le premier algorithme s'applique directement à un modèle, contrairement au deuxième qui demande la définition d'une distance minimale de placement, ainsi que la définition des types d'individus majeurs (placés dans un premier temps).

La deuxième manière d'éditer un modèle est l'utilisation d'un browser générique (Figure 27, à droite). Contrairement à l'éditeur graphique, qui peut cacher certains éléments d'un modèle en raison des mécanismes de simplification (esquisse, calques, etc.), le browser générique présente toujours tous les éléments qui composent un modèle. Le browser est formé par trois listes. La première, « Individu », placé à gauche, comprend tous les individus présents dans un modèle. La deuxième, « Est lié par », est formée par les types de relation qui peuvent avoir l'individu comme origine. La dernière liste, « Avec », contient une liste des individus liés par le type de relation sélectionné.

Les bases de règles et les interprétations sont lancées par les options des menus contextuels des outils. Ces options sont définies au méta-éditeur.

IV. Implémentation

MétaGen a été mis en œuvre en Smalltalk VisualWorks 2.5. Le choix de ce langage d'implémentation s'est fait de manière naturelle. En effet, contrairement aux autres langages disponibles sur le marché, Smalltalk permet de générer des classes, de les compiler et de les exécuter au sein d'une même exécution. Cette capacité est particulièrement souhaitable dans notre cas car la génération d'un environnement de développement implique la génération d'un ensemble de classes qui met en œuvre cet outil. Cependant, cette capacité de Smalltalk ne résout pas tous les problèmes posés par l'implémentation d'un environnement tel que MétaGen. Un premier de ces problèmes concerne le nombre de niveaux d'abstraction à représenter (au moins trois : langage de méta-modélisation, méta-modèles et modèles).

Un deuxième problème concerne la nature et la complexité des composants qui doivent constituer le système (générateur d'éditeurs graphiques, générateur de boîtes de dialogue, générateur de bases de règles, moteur d'inférence, etc.). Un troisième problème concerne la méta-modélisation incrémentale : le méta-modèle n'est pas fixe, il demeure modifiable après la génération de son outillage support. Ainsi, toutes les modifications de celui-ci doivent être prises en compte par les outils générés et ensuite répercutées sur les modèles existants.

La version actuelle de MétaGen a été réalisée grâce à l'intégration de plusieurs cadres d'applications implémentés en Smalltalk 80, comme HotDraw [Johnson, 1992], MPVC [Krief, 1990] et l'interface graphique de VisualWorks [ParcPlace, 1998]. Cette intégration a été réalisée grâce au développement d'un cadre d'applications supplémentaire, dont l'objectif est d'aider la construction de structures de représentation de modèles, spécifiques aux langages de modélisation. Ce cadre a été appelé SGM (structure générique de modèles). De plus, plusieurs patterns de conception ont été employés pour réaliser l'intégration. Ces patterns permettent, par exemple, d'assurer la propagation des changements au cœur de MétaGen sans modifier la structure des cadres d'applications ou de contourner les limitations des méta-classes de Smalltalk lors de la mise en œuvre d'un niveau de méta-modélisation.

Nous décrirons par la suite l'implémentation de MétaGen sous trois aspects : la mise en œuvre de plusieurs niveaux d'abstraction, l'intégration de plusieurs cadres d'application et l'utilisation de patterns de conception. Mais avant de commencer, nous rappellerons rapidement les versions antérieures de MétaGen et les raisons qui ont incité le développement de la version actuelle.

1. Historique

MétaGen a connu plusieurs changements depuis la création de son ancêtre AckPro en 1991 [Krief, 1991], jusqu'à la version actuelle (une description plus détaillée est disponible dans [Revault, 1996] page 234). Ces changements ont été motivés soit par l'inclusion de nouvelles fonctions, soit par le besoin d'un portage vers un environnement de développement plus récent. MétaGen a connu deux grands portages, qui ont coïncidé avec deux changements majeurs du langage d'implémentation et qui ont engendré la réécriture complète de son code : de AckPro vers AckPro GLA [Bonne et Razavi, 1993] (de ObjectWorks 2.5 à ObjectWorks 4.1) et de MétaGen à MétaGen II [Sunyé *et al.*, 1997] (de ObjectWorks 4.1 à VisualWorks 2.5). Nous décrirons par la suite ce dernier portage, auquel nous sommes particulièrement concernés.

La dernière version de MétaGen intégrait le cadre d'applications Editalk, utilisé pour créer des éditeurs graphiques, la bibliothèque AckiaToolBox d'utilisation de boîtes de dialogue et le moteur d'inférences NéOpus. Lors de l'actualisation du langage Smalltalk de Parc Place (de OW 4.1 à VW 2.0), Editalk et AckiaToolBox se sont montrés totalement incompatibles avec la nouvelle version. Comme ces deux composants étaient fortement présents dans l'implémentation de MétaGen et comme leur mise à jour était impossible (d'une part, par l'abandon de leur développement et d'autre part, par la difficulté de les rendre compatibles), nous nous sommes vus confrontés à un dilemme : conserver OW 4.1 et ne pas profiter des nombreux avantages de VW 2.0 (boîtes de dialogue, interface avec SGBD, etc.) ou réécrire complètement MétaGen. Nous avons choisi la deuxième solution.

2. Niveaux d'abstraction

Pour implémenter les trois niveaux d'abstraction souhaités, le premier réflexe consiste à utiliser les niveaux d'abstraction de Smalltalk (méta-classe, classe et instance). Cependant, compte tenu des restrictions d'utilisation des méta-classes, cette solution est d'emblée rejetée. En effet, une méta-classe ne peut avoir qu'une instance unique. Or, les éléments de modélisation de PIR3 doivent être instanciés plusieurs fois. Par exemple, si l'on modélise le langage permettant de décrire des diagrammes de flots de données, la méta-classe individu serait instanciée trois fois ('Processus', 'Acteur Externe' et 'Dépôt de Données'). Pour cela nous avons envisagé une autre solution qui consiste à « zigzaguer »[1] entre les deux niveaux classe et instance, tel qu'il est montré dans la Figure 28 :

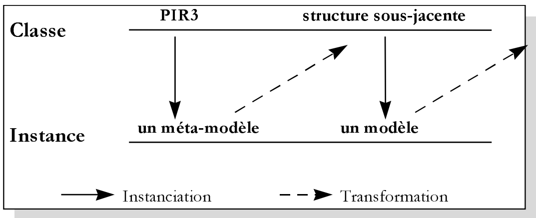

Figure 28 - Les trois niveaux d'abstraction

Cette même technique par d'autres implémentations de méta-outils, comme MetaEdit [Kelly, 1998]. Cet outil utilise des variables de classe pour stocker les propriétés des constructeurs et tire parti de la singularité du statut des classes en Smalltalk. Lorsqu'un nouveau méta-constructeurs est défini, MetaEdit+ crée une nouvelle classe et utilise les variables de classe pour stocker les attributs de méta-constructeurs. La transformation se passe alors entre les niveaux de classe et instance

Pour illustrer notre technique, nous allons présenter un exemple d'un méta-modèle décrivant une abstraction particulière d'un programme dans le langage C.

2.1 Implémentation de méta-modèles

Du point de vue de l'implantation, un méta-modèle est un ensemble d'instances des classes qui composent PIR3. Par exemple, tout individu est une instance de la classe 'MetaIndividual'. Cette

[1] Ce terme a été introduit lors d'un groupe de travail sur la méta-modélisation et les objets, dans le cadre de la conférence OOPSLA'95.

classe possède trois variables d'instance principales : 'nom', 'liens' et 'rubriques'. La variable 'liens' doit contenir la liste des liens qui a un individu avec de différentes relations. Un lien est lui-même un objet complexe représenté par un triplet : le lien proprement dit, la relation, et les cardinalités maximale et minimale. La variable d'instance 'rubriques' contient la liste des rubriques d'un individu.

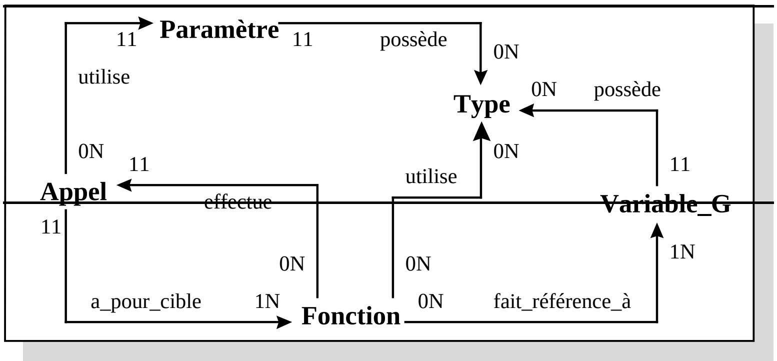

Figure 29 - Méta-modèle d'un programme C

Comme exemple d'implémentation, la Figure 29 présente un méta-modèle partiel d'un programme en langage C, composé par cinq méta-individus : 'Appel', 'Paramètre', 'Fonction', 'Type' et 'Variables'. Le pseudo-code suivant décrie une instance de la classe 'MetaIndividual' correspondant au méta-individu Fonction. Cet individu possède trois rubriques : le nom de la fonction, sa définition et le fichier dans lequel elle est définie. Une 'Fonction' possède, en outre, différents liens, comme celui qui le permet d'utiliser des types par la relation 'utilise'.

```
un MetaIndividual Fonction
      nom: 'Fonction'
      liens: {[utilise 0N],[fait_reference_a 0N], ...}
      composants: {nom, definition, fichier}
      ...
```

Par définition, le méta-modèle est destiné lui-même à être « instancié » pour créer différents modèles. Ceci nécessite une transformation d'une instance telle que celle décrivant une fonction en une classe. Pour cela, nous employons une structure générique de modèles, la SGM, et un mécanisme de transformation qui traduit des instances (un méta-modèle) en classes, sous-classes des classes abstraites qui composent cette structure générique. Cette spécialisation est pilotée par le contenu du méta-modèle.

Plus spécifiquement, les méta-modèles, les méta-individus et les méta-relations sont respectivement transformés en des sous-classes des classes 'MGModel', 'MGIndividual' et 'MGRelation' qui composent la SGM. La classe 'MGIndividual', en particulier, englobe la structure et le comportement générique de tout individu. Ainsi, la variable d'instance relations doit contenir la liste des relations (occurrences) que possède toute occurrence d'un individu. L'instance Fonction décrite précédemment est transformée en une classe 'CIAIndFonction'. Le préfixe « CIA » sert à identifier le méta-modèle :

```
MGIndividual subclass: #CIAIndFonction
      instanceVariableNames: 'nom definition fichier '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'CIACM - CIA - Modele'
```

On notera que les rubriques associées au méta-individu 'Fonction' ('nom', 'définition' et 'fichier') deviennent des variables d'instance de cette classe. Des méthodes d'accès pour ces variables sont générées automatiquement comme les deux suivantes pour la variable fichier :

```
fichier
      "Generee par MetaGen"
^fichier isNil
      ifTrue:[fichier := (MGValue with: nil) type: #IR3TypeString]
      ifFalse:[fichier];
```

```
fichier: valeur
      "Generee par MetaGen"
fichier := MGValue with: valeur
```

Parallèlement, les méta-relations deviennent des méthodes de la sous-classe générée. Par exemple, les deux méthodes suivantes permettent respectivement de rajouter une occurrence de la relation 'utilise', et de retourner la liste des occurrences de cette relation. En effet :

```
addCIARelUtiliseTo: anIndividual
      "Generee par MetaGen"
CIARelUtilise new in: self; out: anIndividual
```

```
relutilise
      "Generee par MetaGen"
^self relations at:#CIARelFontyp ifAbsent: [OrderedCollection new]
```

Les caractéristiques graphiques des méta-modèles, enfin, ne sont pas transformées : elles restent des instances de la structure de classes qui composent le méta-éditeur. Ainsi, tout changement des spécifications du méta-éditeur affecte dynamiquement l'éditeur de modèles.

3. Intégration de cadres d'applications

D'un point de vue conceptuel, MétaGen est divisé en trois grands modules : (i) L'outillage PIR3, où sont exprimés les méta-modèles, (ii) le Méta-éditeur, qui permet d'établir les caractéristiques de l'outillage généré pour chacun d'eux et finalement (iii) la structure générique de modèles, qui sert de base à l'outillage généré.

Ces trois modules, au-delà de leurs objectifs, possèdent une structure similaire : chacun d'eux est formé par un éditeur graphique permettant la création de modèles exprimés dans un formalisme, une structure permettant de stocker les données du modèle représenté et un ensemble de boîtes de dialogue servant à éditer les éléments de ce modèle.

La mise en œuvre de MétaGen présente deux caractéristiques principales, (i) l'utilisation de composants logiciels existants et (ii) une conception modulaire capable de garder une indépendance de ces composants. L'objectif était de minimiser l'effort de développement, tout en gardant un niveau d'indépendance tel qu'il serait possible de remplacer un des composants sans incidence notable sur les autres.

Le résultat final, comme nous verrons par la suite, est un système qui intègre et coordonne plusieurs sous-systèmes : un générateur d'éditeurs graphiques capable de représenter des modèles de type réseaux sémantiques, un générateur de boîtes de dialogues, un interprète d'expertises représentées par des systèmes de règles et un moteur de parcours.

3.1 HotDraw

Pour créer l'éditeur graphique, nous avons choisi HotDraw, qui est un cadre d'applications d'éditeurs graphiques à deux dimensions, développé autour de MVC [Krasner et Pope, 1988] et présent dans la bibliothèque de classes de Smalltalk (Figure 30). De plus, l'ensemble des patterns de conception utilisé pour sa construction fait de HotDraw un cadre d'applications relativement simple à utiliser. HotDraw est décrit dans l'Annexe C de ce document.

HotDraw est intégré de deux façons dans MétaGen. Tout d'abord, il est utilisé pour construire l'outillage de PIR3 : l'éditeur graphique qui permet la représentation du méta-modèle et le Méta-éditeur. Ensuite, HotDraw a été utilisé dans la construction de l'Editeur MG.

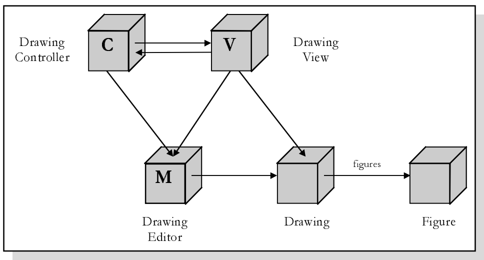

Figure 30 - MVC au cœur de HotDraw

Il existe une dépendance entre le noyau de MétaGen et les éditeurs graphiques (Figure 31). En effet, quand un élément d'un modèle est changé, il envoie un message de changement à tous ses dépendants, qui effectuent les actions nécessaires. Cette solution, inspirée du pattern de conception **Dépendance** [ParcPlace, 1990] ou **Observateur** [Gamma *et al.*, 1995] , permet la représentation de plusieurs plans d'un modèle.

Dans une même logique, l'appel à des opérations ne se fait pas directement. En effet, l'appel d'un dialogue d'édition d'un élément s'effectue par l'intermédiaire d'un message envoyé au noyau, qui appelle la boîte d'édition et ensuite communique le changement de l'élément à l'éditeur graphique.

Une autre solution, inspirée du pattern de conception **Décorateur** [Gamma *et al.*, 1995] ou **Wrapper** [ParcPlace, 1990], permet que certaines caractéristiques d'édition, normalement définies dans une sous-classe de la classe abstraite 'Figure', soient effectivement gardées dans des instances du méta-éditeur. Ainsi, des changements du méta-éditeur, comme l'icône (ou dessin) de représentation d'un individu, sa couleur, son libellé, ses relations (sous-ensemble des relations permises par le méta-modèle) ou son menu, sont affectés dynamiquement aux éditeurs existants (parmi ceux-ci, les éditeurs ouverts auront besoin d'une actualisation).

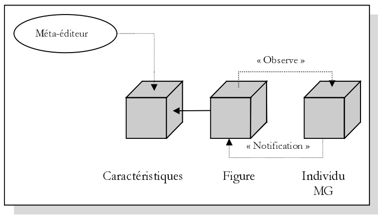

Figure 31 - Représentation graphique d'un individu

3.2 VisualWorks GUI

A partir de sa version 2.0, VisualWorks apporte des canevas de conception de boîtes de dialogue. Ces canevas sont capables d'utiliser une structure générique d'application présente dans la bibliothèque de classes. Cette structure fait aussi appel au pattern de conception **Décorateur**, qui permet l'attachement dynamique de nouvelles caractéristiques à un objet.

Cette structure est construite autour de la classe 'ApplicationModel', qui représente l'objet de domaine à éditer. Les spécifications de l'interface d'édition sont gardées dans une méthode de classe. Lors de l'édition, un builder (par ailleurs, une utilisation du pattern **Builder**) lit ces spécifications et crée les composants de l'interface. Ainsi, la programmation interactive permise par les canevas, ne fait que créer ou surcharger des méthodes.

Dans MétaGen, cette facilité est utilisée de deux façons : comme programmation interactive, pour construire toutes ses boîtes de dialogue et pendant la compilation, pour la génération des boîtes de dialogue de l'outillage généré. Ceci permet un avantage en plus : les boîtes de dialogue générées sont facilement modifiées grâce aux canevas de conception.

Les spécifications de l'interface d'édition sont, en effet, des caractéristiques de la fenêtre et un ensemble de composants graphiques ou *widgets* (des champs d'édition et des libellés), générés par les rubriques (et par leurs types). Une fois cet ensemble constitué, il est possible de disposer les composants sur la fenêtre et d'y ajouter des boutons d'annulation et de confirmation.

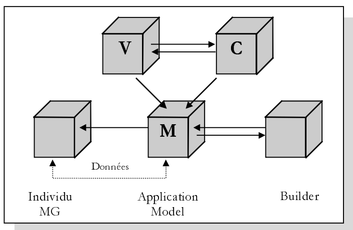

Figure 32 - VisualWorks GUI

Lors de l'édition, les sous-classes de 'ApplicationModel' font une copie de l'élément de la SGM à modifier, pour permettre l'annulation des changements effectués. Dans le cas des dialogues d'édition non modale (notamment les browsers), il existe un lien implicite de dépendance inspiré aussi du pattern de conception **Observateur**.

3.3 Génération d'outils de transformation et de validation

Les règles de production sont utilisées à deux niveaux dans MétaGen. Une première utilisation consiste en la vérification des contraintes d'un modèle, notamment pour la vérification des cardinalités minimales (les maximales étant vérifiées à la volé). La seconde utilisation des règles est la transformation d'un modèle en autre chose qui pourra être un modèle ou bien une implémentation de ce modèle. Ces règles sont exécutées par NéOpus [Pachet et Dojat, 1992] [Pachet, 1995].

NéOpus est un moteur d'inférence en chaînage avant, complètement intégré à Smalltalk et qui fournit un mécanisme de contrôle de règles via des méta-règles [Pachet et Perrot, 1994]. Les bases de règles de NéOpus sont effectivement des classes Smalltalk (sous-classes de 'NeOpusRuleSet'), et les règles de production, des méthodes. Contrairement aux méthodes classiques, les règles sont divisées en trois parties : prédicats, actions et état final. Les prédicats servent à filtrer les objets présents dans le contexte, les actions sont en réalité du code Smalltalk et l'état final permet d'établir des post conditions, à la suite de l'exécution d'une règle. Lors de la compilation du méta-modèle, MétaGen crée deux classes (i.e. deux bases de règles), qui contiendront les règles de transformation et validation d'un modèle.

Dans MétaGen, les bases de règles utilisées sont définies au niveau des méta-modèles et appliquées à tous les modèles issus d'un même méta-modèle. Les prédicats, les actions et les post conditions des règles utilisent généralement des méthodes génériques existantes dans la SGM (e.g. si deux nœuds sont liés) ou des méthodes spécifiques au méta-modèle, créées pendant la transformation (e. g. les associations liées à une entité donnée). Comme nous avons vu ultérieurement dans la section 2.4, ces méthodes (génériques ou spécifiques) sont soit de nature comparative, soit de nature constructive.

La partie action des règles profite des méthodes de création et manipulation de modèles proposées par la SGM et des méthodes spécifiques au méta-modèle (e.g. créer une entité). L'exécution d'une base de règles NéOpus, est précédée par la définition des objets du contexte (potentiellement tous les objets Smalltalk). Dans MétaGen, le contexte d'exécution d'une base de règles est habituellement un modèle, ses individus et relations, et éventuellement le modèle cible de la traduction. Actuellement, il n'existe pas d'éditeur spécifique de règles, elles sont saisies directement dans l'environnement Smalltalk.

Deux méthodes de création d'une relation de type 'Lie' entre une entité et une association sont présentées ci-dessous. La première est générique à la SGM et la deuxième spécifique au langage Entité/Association :

```
<un individu> addRelation: <symboleRelation>
      to: <autre individu>
```

```
<une entité> addEAARelLieTo: <une association>
```

Nous présentons ci-dessous cinq méthodes permettant de vérifier si deux individus sont liés. Les trois premières s'appliquent à un individu et listent, respectivement, tous les individus liés, tous les individus liés par un type de relation et tous les individus liés par la le type de relation Lie. Cette dernière est spécifique au langage Entité/Association. Les deux méthodes suivantes vérifient, respectivement, si un individu est lié à un autre par un type de relation et si une entité est liée à une relation :

```
<un individu> allTargets


<un individu> allTargetsAt: <symboleRelation>


<une entité> targetsAtLie


<un individu> binds: <autre individu>
      at: <symboleRelation>


< une entité > binds: <une association>
```

Les méthodes présentées ci-dessous ne constituent qu'une partie de celles disponibles dans la structure sous-jacente. Le code suivant est un exemple d'une règle de transformation :

```
creerTableAssociation
| RELModele dest. EAIndAssociation ass. EAIndEntite ent1,ent2 |
      ent1 ~= ent2.
      ass attributs size > 0
      ass binds: ent1.
      ass targetsAtRelLie includes: ent2.

actions
| table colonne |
table:= RELIndTable new; nom: (ass nom).
dest addNode: table.
ass attributs do:
      [:each |colonne := RELIndColonne new; nom: (each nom value).
      dest addNode: colonne.
      Table addRelTblColTo: colonne].
(ent1 keys, ent1 keys) do: [:each |colonne := RELIndColonne new;
      nom: (each nom value).
      colonne estCle value: true.
      dest addNode: colonne.
table addRelTblColTo: colonne].
```

Les prédicats de cette règle filtrent les instances de la classe 'EAIndAssociation' de façon à trouver une association liant deux entités et possédant des attributs. La méthode 'attributs' a été générée par MétaGen et rend une collection contenant toutes les instances de 'EAIndAttribut' liées à une association. Les deux derniers prédicats sont équivalents : la méthode 'targetsAtRelLie' rend toutes les entités liées à une association et la méthode 'binds:' vérifie si l'association est liée à une entité, indépendamment de la relation qui les lie.

Les actions permettent de créer, dans un modèle relationnel, une instance de 'RELTable' et de lui ajouter des colonnes : les attributs de l'association en question, et les attributs clefs des deux entités liées par cette association. Ces dernières seront les clefs de la nouvelle table.

3.4 Génération d'outils d'interprétation de modèles

Outre les règles, il existe une seconde méthode pour exploiter les modèles construits : le parcourir. Pour cela, nous avons employé le moteur de parcours MPVC [Krief, 1990] (Figure 33) qui nous permet de définir, sur un graphe, un parcours (cette définition est appelée un contrôleur), et d'associer à chaque élément parcouru un ou plusieurs points de vue, où chaque point de vue correspond à une action à réaliser. Ainsi, à partir d'un individu d'un modèle, il est possible de suivre toutes les relations qui sortent de cet individu. Dans le cas où plusieurs relations sortent d'un même individu, l'utilisateur est invité à choisir lesquelles devront être parcourues.

Il est important de signaler que l'initialisation du parcours se fait à l'aide de plusieurs méthodes, définies au niveau de la classe supportant le contrôleur du parcours. Toute modification apportée à ces méthodes doit être validée non seulement en modifiant la méthode, mais également par la redémarrage de l'exécution.

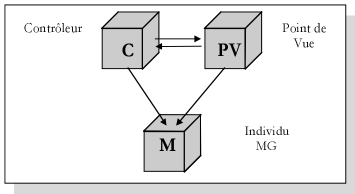

Figure 33 - MPVC

Deux points de vue sont créés en même temps que le contrôleur. Le premier servira à visualiser le parcours suivi à l'aide d'une petite sphère. Le second n'a pas d'actions associées mais des méthodes sont créées pour servir de réceptacle au point de vue choisi.

4. Patterns de conception utilisés

L'implémentation de MétaGen a utilisé plusieurs patterns de conception pour réaliser, par exemple, la structure générique de modèles ou l'intégration des cadres d'applications présentés dans la section précédente. Nous décrirons par la suite quelques exemples d'utilisation de patterns de conception dans MétaGen.

La totalité des implémentations de patterns décrites ici fait référence aux descriptions du catalogue de Gamma et al. [Gamma *et al.*, 1995].

Le pattern **Décorateur** a été employé dans l'implémentation des décorateurs du méta-éditeur (classe 'MEIndividual'). Ceci permet que certaines caractéristiques d'édition, normalement définies dans une sous-classe de la classe abstraite 'Figure' de HotDraw, soient effectivement stockées par des instances. Ainsi, les changements des décorateurs, comme l'icône (ou dessin) de représentation d'un individu, sa couleur, son libellé, ses relations (sous-ensemble des relations permises par le méta-modèle) et son menu, sont affectés dynamiquement aux éditeurs existants (parmi ceux-ci, les éditeurs ouverts auront besoin d'une actualisation).

Les esquisses du méta-éditeur sont des structures complexes car elles rassemblent plusieurs décorateurs et doivent, en même temps, se comporter comme un décorateur unique. Pour faire face à cette difficulté, nous avons utilisé le pattern **Composite**. Plus précisément, une instance de la classe 'MEEsquisse' stocke une référence de tous les décorateurs qui composent l'esquisse et transmet, si nécessaires, les messages qui lui sont envoyées.

Les rubriques de PIR3 sont des objets complexes représentés par un ensemble d'objets de trois classes : 'IR3Rubrique', 'IR3Nature' et 'IR3Type'. De ce fait, l'accès à certaines valeurs d'une rubrique est complexe. Par exemple, la valeur par défaut d'une rubrique est stockée dans un objet de la classe 'IR3Nature', tandis que la taille d'une rubrique est gardée par un objet de la classe 'IR3Type'. Pour simplifier l'accès à cette structure, nous faisons appel au pattern de conception **Façade**, dont l'objectif est de fournir une interface unifiée à l'ensemble d'interfaces d'un sous-système, le rendant plus simple à utiliser. Dans notre cas, classe 'IR3Rubrique' regroupe toutes les méthodes d'accès aux valeurs d'une rubrique.

La représentation d'une méta-relation PIR3 comprend plusieurs objets, instances de cinq classes principales : 'MetaRelationBinaire', 'IR3Role', 'IR3Cardinalite', 'IR3Lien' et 'IR3Rubrique'. En principe, des instances de ces classes sont créées en même temps que la méta-relation. Cependant, ceci n'est pas une règle générale car dans certain cas, notamment lors de la restauration d'un méta-modèle, où ces instances doivent être créées séparément. Dans d'autres cas, comme lors de l'utilisation de méta-modèles créés par d'autres versions de MétaGen, les méta-relations sont représentées par d'autres classes, ce qui empêche l'utilisation d'une méthode unique de création. Pour résoudre ce problème, nous avons créé des « kits » de création de méta-relations. Ceci est une implémentation du pattern **Fabrique Abstraite** (*Abstract Factory*), dont le propos est de fournir une interface pour créer des familles d'objets liés ou dépendants, sans préciser leurs classes concrètes. Ce même pattern est aussi employé lors de la création d'un méta-individu.

La structure sous-jacente d'un langage de modélisation est représentée par un ensemble de classes, sous-classes de celles qui composent la SGM. La forme de cette structure est fortement dépendante du méta-modèle qui représente ce même langage, et est donc variable. Ceci rend complexe la création d'un procédé commun de sauvegarde de modèles. En plus, les structures ne sont pas fixes, elles peuvent être modifiées en fonction des changements effectués sur le méta-modèle. Pour résoudre ce problème, nous avons choisi d'utiliser une approche constructive de la sauvegarde et la restauration de modèles, au détriment de l'utilisation d'un format spécifique de fichier. Par approche constructive, nous entendons la création d'un fichier contenant les diverses méthodes qui doivent être appelées pour créer une structure complexe d'objets. Ce fichier est en réalité une application du pattern **Monteur**, dont l'objectif est de séparer la construction d'un objet complexe de sa représentation.

Ce même pattern est employé par les dialogues d'édition présents dans MétaGen. Lors de la création des outils de modélisation, le méta-éditeur génère des boîtes de dialogue spécifiques aux décorateurs, qui s'adaptent à l'interface graphique utilisée et peuvent être modifiés. Cette génération est en réalité la création d'une méthode de construction d'un dialogue, compatible avec les méthodes générées par les outils de dessin de dialogues de VisualWorks. Le procédé de création du fichier de sauvegarde et des dialogues d'édition est une application du pattern **Chaîne de Responsabilités**.

Finalement, le pattern **Singleton** a été employé pour créer un référentiel de messages (implémenté par la classe 'MetagenLanguage'). L'assurance de l'unicité de ce référentiel nous a permis d'utiliser les mêmes boîtes de dialogues en plusieurs langues. Ce même pattern a été utilisé dans l'implémentation de la classe 'MetagenBrowser', qui stocke les variables globales de MétaGen.

V. Conclusion

Dans ce chapitre, nous avons présenté MétaGen, un outil de génération d'environnements de développement logiciel. La version actuelle de MétaGen a amené plusieurs améliorations, concernant l'outillage PIR3, les éditeurs de diagrammes générés et la génération des dialogues d'édition. De nouvelles fonctions ont été introduites, comme les mécanismes de simplification de modèles, la génération de boîtes de dialogue ou l'édition de contraintes.

MétaGen a été développé Smalltalk 80. Sa mise en œuvre a été marquée par l'intégration de plusieurs cadres d'applications, et par la cohabitation de trois niveaux d'abstraction. Les restrictions d'utilisation des méta-classes dans Smalltalk posent un problème par rapport à l'implantation des concepts méta. Nous avons contourné ce problème en définissant et en utilisant le pattern Zigzag, qui s'applique entre les niveaux classe et instance. D'un autre point de vue, Smalltalk s'est avéré très efficace quant à la disponibilité et l'utilisation de composants existants.

L'utilisation de cadres d'applications diminue le codage et oblige une étude préalable des composants déjà existants. Même si cette étude est longue, elle est très importante puisqu'elle oblige le programmeur à comprendre un code en général bien écrit. Ceci explique une caractéristique assez particulière à Smalltalk où les conventions de codage et de présentation jouent un rôle aussi important que la syntaxe du langage.

L'utilisation de plusieurs cadres d'applications dans le développement de MétaGen a nécessité une étape de développement supplémentaire, celle de leur intégration. Dans notre cas, le développement d'un nouveau cadre d'applications - le noyau MétaGen (SGM) - était nécessaire pour représenter les objets du domaine. Cependant, l'intégration ne dépendait pas de ce nouveau cadre d'applications, mais seulement des relations qui font appel à des techniques de conception bien définies et documentées : les patterns de conception.

1. Perspectives

Actuellement, MétaGen peut évoluer en deux directions distinctes. Premièrement, sous le point de vue de sa conception, la structure générique de modèles peut évoluer, de manière à devenir complètement indépendante de MétaGen. Elle constituerait ainsi un véritable cadre pour la réalisation de structures sous-jacentes de langages de modélisation. Plus précisément, elle serait utilisée pour stocker de modèles et de méta-modèles, ainsi que pour gérer l'accès à un futur système de gestion d'objets (SGO).

Deuxièmement, plusieurs fonctionnalités peuvent être ajoutées à MétaGen, comme un éditeur de règles ou des nouveaux outils génériques. Nous présenterons par la suite quelques propositions sur l'évolution de MétaGen Nous commençons par le support de différentes notations par le canevas d'environnement existant, suivie d'un éditeur de règles.

1.1 Vues multiples d'un modèle

L'objectif principal de la démarche de développement de l'équipe MétaFor est de donner aux utilisateurs des moyens de modélisation capables de comprendre le jargon propre à leur domaine de connaissances. Actuellement, MétaGen ne satisfait que partiellement cet objectif : il permet de générer un outillage de modélisation spécifique à un langage de modélisation, mais cet outillage est restreint en ce qui concerne les notations supportées.

Il nous paraît important que l'évolution de MétaGen passe par le support de plusieurs formes de notation, complémentaires à l'éditeur de diagrammes déjà existants. Nous proposons ici de faire progresser le méta-éditeur de manière à permettre de définir deux nouvelles manières d'interagir avec la structure sous-jacente, i.e. de visualiser et modifier des modèles créés dans MétaGen : un éditeur matriciel et un browser (Figure 34).

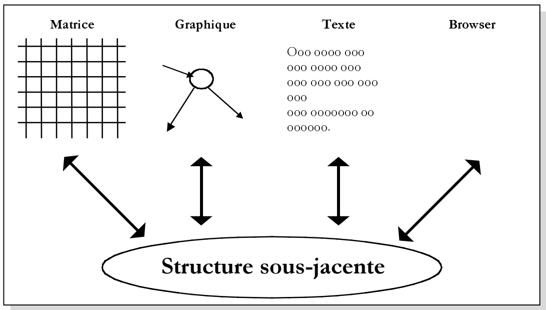

Figure 34 - Des vues multiples d'un modèle

Dans ce même sens, l'éditeur de diagrammes nécessite plusieurs améliorations, en ce qui concerne la représentation des nœuds et des arcs graphiques.

1. Points d'ancrage : les nœuds ne possèdent qu'un seul point d'ancrage (le centre), ce qui empêche la présence de symboles comme un cercle ou une flèche au bout d'un arc. Le méta-éditeur devrait permettre la définition de plusieurs points d'ancrage ;

2. La représentation des nœuds est limitée aux icônes et aux symboles de base (cercle, rectangle, etc.), les derniers étant disponibles dans un outil d'édition de symboles, présent dans le méta-éditeur. Il serait intéressant de faire évoluer cet éditeur de façon à permettre de combiner plusieurs figures graphiques de façon à mieux habiller les nœuds et arcs graphiques ;

3. Finalement, il serait souhaitable de pouvoir associer plusieurs formes de représentation à un élément (individu ou relation), dépendantes de la valeur d'une propriété. Cette caractéristique permettrait de connaître, partiellement, les valeurs des propriétés des concepts modélisés.

Nous présenterons par la suite la proposition de deux nouveaux outils de modélisation, un éditeur matriciel et un browser (spécifique au langage). Ces outils permettront aux concepteurs de mettre en valeur de différents concepts d'un modèle, selon leurs besoins.

i) Un éditeur matriciel

Un aspect intéressant d'un éditeur matriciel est qu'il met en évidence les relations d'un modèle, contrairement à l'éditeur de diagrammes. Une vue matricielle est sans doute plus limitée qu'un diagramme, car elle n'est composée que de deux axes, et ne peut présenter plus que deux types d'individus en même temps (bien que la scission d'un axe soit une alternative à cette limitation). Toutefois, une matrice permet la manipulation simultanée d'une plus grande quantité d'informations, ce qui peut s'avérer très pratique pour la manipulation de modèles complexes.

La définition d'un éditeur matriciel serait réalisée par le choix de deux axes qui le composent. Tous les types d'individus d'un langage sont des candidats à composer un des axes de l'éditeur de matrices, s'il existe au moins un type de relation permettant de les relier ou de les lier à un troisième type d'individu. Le deuxième choix à réaliser concerne le contenu des cellules, qui peut être une case cochée (il existe une relation entre les deux individus), une propriété (de la relation ou de l'individu de liaison). L'éditeur matriciel pourrait aussi être utilisé pour éditer rapidement les propriétés des individus issus d'un même type. Pour cela, le deuxième axe doit être formé par les propriétés du type d'individu du premier. Les cellules présenteraient alors les valeurs des propriétés des individus.

Entité/ Entité	Personne	Voiture
Personne		Possède
Voiture	Possède	

Figure 35 – Edition matricielle d'un modèle Entité/Association

La Figure 35 est un exemple de l'édition matricielle d'un modèle Entité/Association. Les deux axes utilisent le même type d'individu (Entité) et une propriété du type d'individu de liaison (de nom de l'Association) comme valeur de la cellule.

Une alternative à limitation de deux axes consiste en simuler un troisième axe par la création d'un second niveaux. La Figure 36 est un exemple d'utilisation de trois axes pour représenter un diagramme de flots de données (le diagramme original est présenté par la Figure 65, page 150). Le premier niveau de l'axe horizontal représente les Dépôts de Données, le deuxième niveau les Processus qui sont directement liés à ces Dépôts (ils peuvent se répéter). L'axe vertical représente tous les Acteurs Externes et les cellules de la matrice représentent l'existence d'une relation entre les Processus et les Acteurs.

Dépôts	Client	Produit	Commande		Facture	
Acteurs/ Processus	Contrôle Commande	Contrôle Commande	Contrôle Commande	Livraison et Facturation	Livraison et Facturation	Contrôle Facture
Client	X		X			X

Figure 36 – Edition matricielle d'un diagramme de flots de données.

Une deuxième alternative de simulation d'un troisième axe est présentée par la Figure 37. Elle consiste en scinder un axe et de lui permettre de contenir des individus issus de deux types différents.

	Entités		Associations
Attributs	Personne	Voiture	Possède
Nom	X		
Age	X		
Code		X	

Figure 37 - Edition matricielle d'un modèle Entité/Association

ii) Un browser

Un browser peut être vu comme une succession de listes ou le contenu de chacune dépend de la sélection de la liste précédente. Cette chaîne de dépendances s'arrête à la première liste, dotée d'un contenu fixe. Notre proposition consiste en permettre la définition, dans le méta-éditeur, de la quantité de listes, leur placement et du contenu de chacune.

Dans notre cas précis, le contenu des listes devrait être soit un ensemble de constructeurs (des types d'individu ou de relation), soit un ensemble de concepts (des individus ou des relations). La dépendance entre deux listes sera définie en fonction du contenu des deux listes. La dépendance peut être directe, comme entre les constructeurs et leurs concepts ou entre individus et relations. La dépendance entre deux listes contenant des individus doit être définie en fonction d'un ensemble de types de relation.

Une liste peut donc contenir des concepts issus de différents types, bien que les relations et les individus ne puissent pas cohabiter dans une même liste. Les propriétés peuvent être utilisées comme filtre, et constituer un complément au sous-ensemble de types qui déterminent le contenu d'une liste.

Une alternative intéressante à la présentation par liste serait de réserver une zone du browser aux propriétés et permettre ainsi l'édition des concepts.

1.2 Edition de règles

Malgré les améliorations de la version actuelle, la génération d'un environnement de développement reste une tâche complexe. L'utilisation d'une variante d'un langage de modélisation connu, Entité/Association, comme langage de méta-modélisation permet la représentation rapide de la syntaxe d'un langage de modélisation et la génération de ses outils de modélisation. Cependant, cette rapidité n'est pas présente lors de l'écriture des règles de transformation de modèles ou des parcours d'interprétation. En effet, l'écriture des règles de transformations de modèles et en particulier celle concernant la génération de code, s'avère très complexe. Cette complexité concerne aussi bien les prédicats que les actions d'une règle.

L'objectif des prédicats d'une règle est de filtrer les objets contenus dans le contexte. Dans le cas spécifique de la manipulation d'un modèle, ces objets correspondent aux concepts modélisés par les constructeurs d'un langage. Comme les constructeurs sont spécifiés dans un méta-modèle, et que plusieurs opérations de vérification sont disponibles dans la structure générique de modèles, il est possible de créer un éditeur de règles capable de proposer une interface simplifiée de création des prédicats d'une règle.

Toutefois, l'édition des actions d'une règle combine des opérations qui concernent les constructeurs des langages source et cible, ce qui augmente la complexité du développement de cette partie spécifique de l'éditeur de règles. Pour réaliser un tel éditeur, la complexité des règles doit être atténuée.

Chapitre 4

PatternGen, un outil de mise en œuvre de patterns de conception

I. Introduction

Dans ce chapitre, nous présentons PatternGen [Sunyé, 1999], un prototype d'outil de génération automatique de code à l'aide de patterns de conception qui prend en compte les variantes d'implémentation. PatternGen adopte donc une approche générative de l'outillage de patterns de conception. Comme nous avons vu antérieurement (§ Chapitre 1V page 26), le principe de cette approche est de fournir aux concepteurs des moyens de sélectionner le pattern de conception souhaité, de lui permettre de choisir sur qu'elles classes il sera appliqué et de générer automatiquement le code correspondant à cette instance spécifique du pattern.

Avant d'introduire les fonctionnalités de PatternGen, il est important de rappeler la signification du terme « instance de pattern. Comme vu précédemment à la page 14, une instance de pattern est la réalisation concrète d'un pattern, c'est à dire, la particularisation de la description d'un pattern à plusieurs niveaux, avec des noms spécifiques au contexte de réalisation et en tenant compte de plusieurs compromis d'implémentation. Un pattern décrit une solution abstraite à un problème de conception, et s'énonce nécessairement en termes génériques.

Pour passer de la description d'un pattern à une de ses instances dans une application, PatternGen explicite la définition des compromis d'implémentation propres à cette instance. Il utilise pour cela une représentation des connaissances des implémenteurs sur la mise en œuvre du pattern. En effet, l'utilisation des patterns de conception requiert l'expertise d'au moins deux acteurs du développement d'applications : le concepteur et l'implémenteur. Le premier identifie une situation d'utilisation d'un pattern de conception. Le deuxième développe le code le plus approprié pour cette situation spécifique.

Ces connaissances se traduisent à différents niveaux, depuis un éditeur de diagrammes, où les modèles de conception utilisables pour la réalisation d'une application sont explicités, jusqu'à une base de règles qui se charge de générer le code exécutable correspondant à l'ensemble des choix retenus et qui représente le savoir-faire des implémenteurs. L'objectif principal de PatternGen n'est pas de remplacer l'expertise du concepteur, mais de l'aider à modéliser son application, en lui fournissant l'information nécessaire et en suggérant, éventuellement, des modifications à son modèle. Le modèle de l'application est ensuite analysé par une base de règles, qui génère le code exécutable correspondant. Ce faisant, PatternGen adopte une approche qui ne réifie pas les patterns dans le code, mais qui génère la solution spécifique à chaque instance d'un pattern.

Ce chapitre est divisé en 5 sections. La section suivante présente l'architecture de PatternGen et de ses trois modules principaux, et explique les choix effectués pour les réaliser. L'inclusion de quatre patterns de conception dans PatternGen sera expliquée par la troisième section. La quatrième section explique l'implémentation de PatternGen et l'écriture de la base de règles. Nous concluons par une discussion sur les avantages et limites de l'utilisation de PatternGen et sur les axes d'évolutions que nous retenons pour la suite de nos travaux.

II. Architecture

PatternGen est composé par trois modules principaux (Figure 38). Le plus important d'entre eux est un éditeur de diagrammes qui permet la modélisation d'une application sous la forme d'une hiérarchie de classes (correspondant grosso modo au diagramme de classes d'OMT). En même temps, il permet de choisir les classes qui composent une instance d'un pattern. Ce choix peut être comparé à la mise en relation entre les modèles de conception et de patterns, proposée par Kim et Benner [Kim et Benner, 1995], bien que, PatternGen permette la description des compromis d'implémentation spécifiques à cette instance, ce qui n'est le cas de leur proposition.

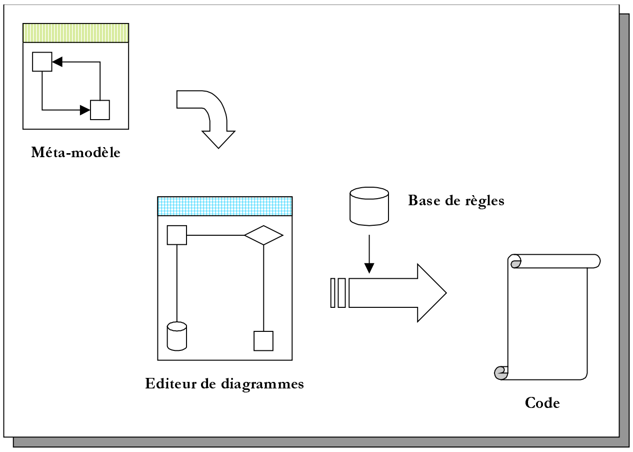

Figure 38 - Architecture de PatternGen

L'éditeur de diagrammes supporte le langage de modélisation MAC (pour Méthode, Attribut et Classe), où les patterns apparaissent comme des entités atomiques. Ils jouent le rôle de constructeurs, opérant sur des classes, sur des attributs ou sur des méthodes d'un modèle de classes. De cette manière, les patterns intègrent le langage MAC, de manière similaire à l'inclusion des patterns dans LayOM [Bosch, 1996] (§7, page 29).

L'éditeur de diagrammes est couplé avec une *base de règles* de production, exprimant des connaissances de conception et d'implémentation. Grâce aux connaissances exprimées par cette base de règles, PatternGen peut analyser les compromis et les participants de chaque instance d'un pattern, suggérer des modifications dans la hiérarchie de classes et générer le code spécifique aux différentes variantes d'implémentation d'un pattern. L'éditeur est aussi capable de lire le code d'une application et de la présenter comme un modèle, permettant ainsi la modification d'applications existantes, comme le propose Patterns Wizard [Eden *et al.*, 1997] (§6, page 29) avec les applications écrites en Eiffel.

PatternGen accepte la modification des caractéristiques d'un pattern, ainsi que l'inclusion de nouveaux patterns grâce à un *méta-modèle*, qui représente (réifie) le langage MAC. Lorsqu'un nouveau pattern est ajouté au méta-modèle, il est aussitôt disponible dans l'éditeur de diagrammes.

Finalement, PatternGen gère les diverses variantes d'implémentation, comme le fait l'outil proposé par Budinsky et al. [Budinsky *et al.*, 1996] (§ 5, page 28). Toutefois, contrairement à leur outil, le code généré par PatternGen est intégré au code de l'application modélisée. Il ne s'agit pas ici de réifier les patterns comme le fait CodeFarms Jiri Soukup [Soukup, 1995] (§1, page 26), mais de générer le code propre à chaque instance d'un pattern.

1. L'éditeur de diagrammes

Avant de développer PatternGen, et plus précisément son éditeur de diagrammes, il était nécessaire de concevoir un langage de modélisation. Ce langage devrait permettre la représentation d'instances de patterns de conception. Il servirait de support à l'éditeur de diagrammes.

Deux approches étaient envisageables pour créer ce langage :

1. La création d'un véritable langage de patterns, i.e. un langage de modélisation composé par des constructeurs extraits directement de la description des patterns (e.g. Observateur, Proxy, Sujet, etc.) ;

2. L'extension d'un langage de conception existant, comme OMT et UML.

Cette première solution a été rejetée, pour deux raisons principales. Premièrement, les patterns de Gamma et al. [Gamma *et al.*, 1995] sont décrits en termes de classes, méthodes et attributs. Ils s'appliquent à un modèle à objets et aident les concepteurs non expérimentés à employer des solutions de conception préconisées par des concepteurs expérimentés. Donc, ils ne constituent pas une alternative à la conception à objets et il est difficile de les dissocier d'un langage de modélisation à objets.

Deuxièmement, le comportement d'une classe n'est défini que très partiellement par un pattern. Il dépend aussi du rôle joué par cette classe au sein de l'application, du comportement d'autres classes et même d'autres patterns (l'application d'un pattern n'est pas exclusive). Le groupement des ces différents comportements en un seul constructeur de modélisation (e.g. une figure-observateur-composite) est donc presque impossible.

La deuxième approche est d'une part plus simple à réaliser et d'autre part, plus simple à comprendre car le concepteur est confronté à un langage connu, disposant de constructeurs connus. Cette approche a été suivie lors de la création de MAC.

Il est important de signaler qu'il ne suffit pas d'enrichir les constructeurs de modélisation du langage, comme par exemple ajouter une rubrique « rôle » dans le constructeur classe. On doit permettre d'indiquer l'existence d'un pattern en dehors de la classe.

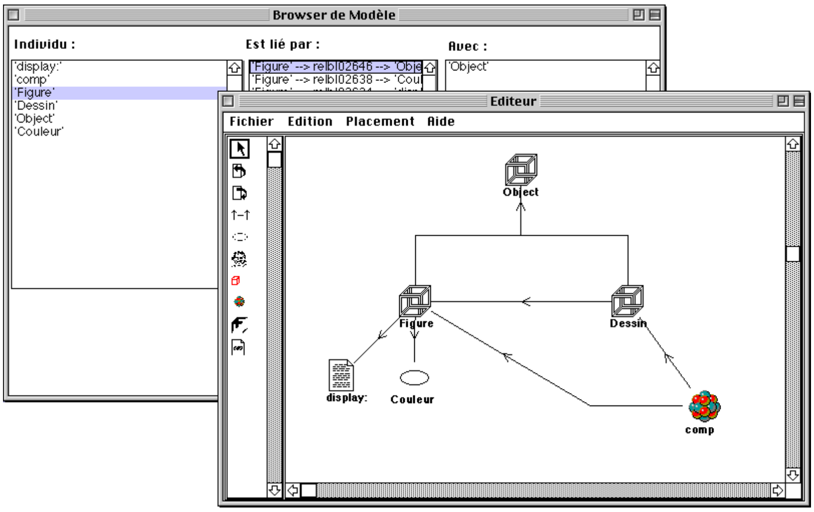

Figure 39 - L'éditeur de diagrammes

MAC est supporté par deux outils, un éditeur graphique de diagrammes et un browser (à la Smalltalk). Dans l'éditeur de diagrammes (Figure 39), les classes ont pour icône générique le cube. Leurs listes d'attributs (représentés par une ellipse) et leurs méthodes (les feuilles de texte) sont accessibles interactivement.

L'éditeur de diagrammes dispose de plusieurs mécanismes de simplification de modèles (pour une description plus précise, voir page 78). Ces mécanismes permettent l'occultation certains éléments d'un modèle pour mettre en évidence un aspect précis. Il est possible, par exemple, de cacher les relations d'agrégation, les méthodes et les attributs d'un modèle pour mettre en évidence la structure hiérarchique d'un modèle.

L'éditeur dispose aussi de boîtes de dialogue (Figure 40), qui permettent l'édition des propriétés du modèle et des classes, attributs et méthodes qui le composent. Un modèle MAC possède deux propriétés, son nom et le langage utilisé (Java ou Smalltalk).

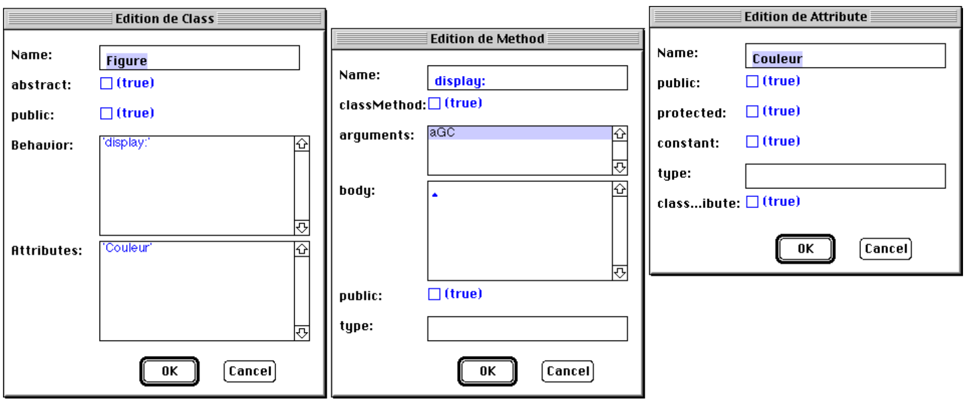

Figure 40 - Edition de méthodes, attributs et classes

Ces mêmes dialogues sont accessibles à partir du browser. Les modifications réalisées par l'intermède d'un des deux outils sont automatiquement prises en compte par l'autre.

Les instances de patterns ont des icônes spécifiques : une molécule pour Composite, un visage pour Observateur, etc. Les flèches déterminent quelles sont les classes qui participent à ces réalisations.

Un exemple d'une instance d'un pattern peut être distingué dans la Figure 39. L'icône d'une molécule représente une instance du pattern Composite. Deux flèches indiquent les classes participantes à cette instance ('Figure' et 'Dessin'). Ce pattern sera détaillé par la suite (§2, page 110).

Les choix d'implémentation spécifiques à chaque instance d'un pattern apparaissent comme des propriétés de cet objet, et comme tels ils sont accessibles par dialogue à travers l'éditeur.

2. Représentation de patterns

Dans PatternGen, les patterns sont représentés par un méta-modèle, conjointement avec le langage de modélisation MAC. La représentation d'un pattern dans le méta-modèle ne nécessite pas la description complète d'un pattern, tel qu'il est décrit dans un catalogue. L'objectif ici est de permettre au concepteur de décrire une instance d'un pattern qu'il souhaite utiliser.

En effet, l'objectif de notre outil n'est pas d'aider le concepteur à choisir l'application d'un pattern, ou suggérer un besoin, mais de lui permettre de décrire, une fois le pattern et les options d'implémentation dûment choisies, toutes les informations importantes concernant l'implémentation d'une instance de pattern. Le méta-modèle doit alors permettre la représentation de certains aspects précis d'un pattern : les participants et les compromis d'implémentation.

La question qui se pose alors, est de savoir comment représenter un pattern, i.e. quel est le langage de méta-modélisation le plus adapté pour « méta-modéliser » un pattern.

Une des réponses possibles a été étudiée par Pagel et Winter [Pagel et Winter, 1996] : l'utilisation des méta-patterns de Pree [Pree, 1995]. Dans leur cas, il s'agit du méta-pattern « Hook & Template ». Toutefois, cette solution est limitée en ce qui concerne l'intégration des patterns à un langage de modélisation car les méta-patterns ne sont pas adaptés à la méta-modélisation.

Une deuxième solution, (et celle qui a été adoptée par la plupart des outils de mise en œuvre de patterns) consiste à utiliser un langage de méta-modélisation quelconque, comme ceux présentés précédemment (page 70).

Dans notre cas, nous avons opté pour la deuxième solution, qui paraissait plus adapté à notre idée initiale : intégrer les patterns à un langage de modélisation. Il restait à déterminer comment cette intégration serait faite. Deux possibilités se sont présentées : la spécialisation des constructeurs d'un langage (à la LayOM [Bosch, 1996]) ou l'ajout d'un nouveau constructeur à un langage de modélisation.

Dans le premier cas, il s'agit d'ajouter une propriété 'rôle' à un ou plusieurs constructeurs de modélisation. L'objectif de cette propriété étant de déterminer, par exemple, le rôle d'une classe (ou méthode ou attribut) participante à une instance de pattern. Cette approche présente trois difficultés : (i) une classe peut participer à des différentes instances de pattern ; (ii) la détermination des classes participantes à une même instance d'un pattern n'est pas directe et (iii) les compromis d'implémentation s'appliquent à l'instance du pattern et ne peuvent pas être spécifiés au niveau des classes participantes.

Dans le deuxième cas, ils s'agit d'ajouter un ou plusieurs constructeurs à un langage de modélisation, ce qui permet de contourner les difficultés énumérées ci-dessus. Notre idée initiale était d'utiliser un seul constructeur, appelé « Pattern ». Cette solution a été rejetée, car les patterns sont trop spécifiques pour être regroupés dans un constructeur unique. Cette spécificité concerne notamment les compromis d'implémentation et le nombre de classes participantes. La solution adoptée a été alors de définir un constructeur de modélisation pour chaque pattern de cocneption.

La syntaxe du langage de modélisation MAC représentée dans un méta-modèle PIR3 (Figure 41). Comme nous avons décrit auparavant (page 70), le langage de méta-modélisation PIR3 est formé par deux constructeurs de modélisation principaux : méta-individu et méta-relation.

Les méta-individus sont les constructeurs identifiables d'un langage de modélisation (e. g. Entité, Processus, Classe) et sont liés par des méta-relations. Méta-individus et méta-relations possèdent des rubriques typées (par des classes standard Smalltalk).

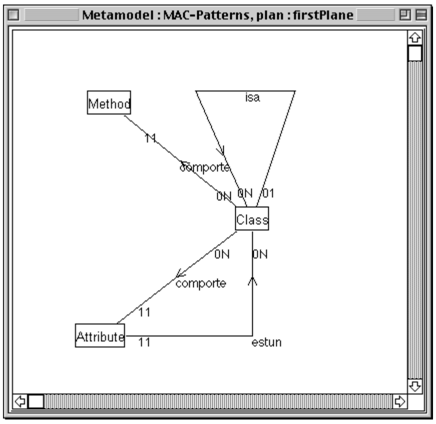

Figure 41 - Méta-modèle de MAC – plan de base

Le plan (i.e. sous-graphe) de base du méta-modèle de MAC (Figure 41) est formé par trois méta-individus, représentés par des rectangles : 'Class', 'Method' et 'Attribute' et par des liens (méta-relations) entre eux. Les cardinalités des méta-relations assurent que, par exemple, une classe ne peut avoir plus d'une super-classe.

Le méta-individu 'Class', aussi bien que les autres méta-individus, possède plusieurs rubriques (non visibles dans le dessin) : 'name', 'abstract' et 'public', qui correspondent aux propriétés du dialogue d'édition présenté auparavant (Figure 40).

La Figure 42 présente un plan différent de ce même méta-modèle. Ce plan contient un nouveau méta-individu, 'Composite', qui représente le pattern de même nom. En effet, un pattern est représenté dans le méta-modèle par un méta-individu et une ou plusieurs méta-relations. Les rubriques de ces objets permettront, respectivement, la création d'instances de ce pattern dans l'éditeur de diagrammes, l'indication des classes participantes à chaque instance et le choix des compromis d'implémentation.

Chaque pattern a une représentation particulière dans le méta-modèle. Leur seule caractéristique commune est d'entretenir des méta-relations avec le méta-individu « Class ». En effet, les patterns mettent en œuvre une collaboration entre plusieurs classes, mais la manière de réaliser cette collaboration, le nombre de classes participantes et les compromis d'implémentation sont spécifiques à chaque pattern.

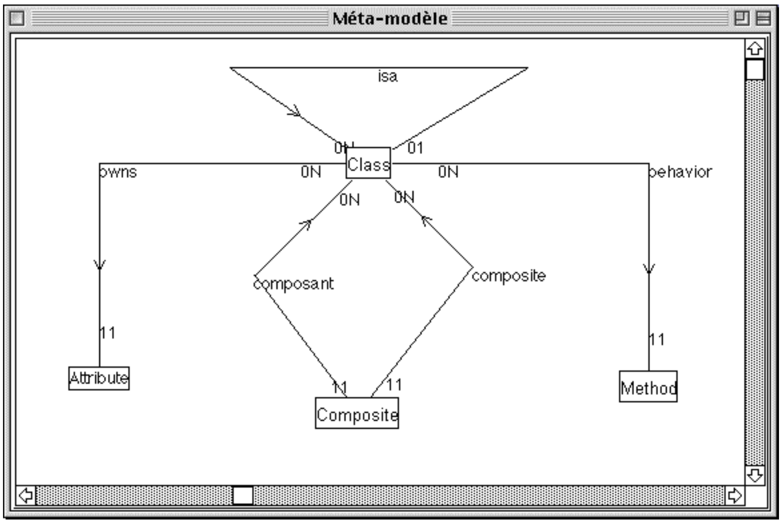

Figure 42 - Méta-modèle de MAC – plan « Composite »

Chaque classe participante à un pattern joue un rôle spécifique, signalé par son nom (observateur, sujet, composite, etc.). Ces rôles sont représentés dans le méta-modèle par des méta-relations. Dans la Figure 3, deux méta-relations (composite et composant) indiquent les deux participants du pattern Composite. Les méta-relations sont aussi utiliser pour indiquer des méthodes, si leur rôle est important au développement du pattern.

L'implémentation d'un pattern dépend de plusieurs compromis d'implémentation, qui peuvent modifier la structure du pattern (et éventuellement induire à l'application d'un second pattern) ou l'implémentation des méthodes d'une des classes participantes. Lorsqu'un compromis interfère avec la totalité de l'implémentation, il est représenté par une rubrique du méta-individu qui représente le pattern.

Lorsqu'un compromis modifie l'implémentation d'un des participants, il est aussi représenté par une rubrique, mais cette fois de la méta-relation correspondante à ce participant. Certains compromis sont déduits automatiquement soit par l'indication ou l'omission d'un participant, soit par l'existence d'une autre instance du même pattern.

La représentation d'un pattern est complétée par l'écriture d'une base de règles, qui exprime les connaissances nécessaires à son implémentation. Ces connaissances seront décrites dans la section suivante.

3. Représentation des connaissances des implémenteurs

L'utilisation de patterns de conception durant le développement d'une application implique la connaissance de ceux-ci par au moins deux acteurs : le concepteur et l'implémenteur. L'expertise du concepteur se base sur son expérience en conception de logiciels et sur sa connaissance, au moins partielle, des patterns de conception - les problèmes qu'ils décrivent et les solutions proposées. Celle de l'implémenteur comprend sa connaissance du ou des langages de programmation utilisés, avec ses techniques de mise en œuvre et l'usage de l'environnement de développement associé. C'est à dire, tout ce qui l'aide à choisir l'implémentation la plus appropriée à une instance d'un pattern.

Dans PatternGen, ces connaissances sont représentées par une base de règles de production, qui sont adaptées à la représentation de l'aspect concurrentiel des compromis d'implémentation (voir § Chapitre 1 : IV(5), page 24).

Les bases de règles sont implémentées avec NéOpus [Pachet, 1995]. Comme décrit au Chapitre 3 (§ III.3, page 76), NéOpus permet le contrôle du déclenchement de règles par des métarègles. Cette caractéristique est particulièrement souhaitable dans l'implémentation de patterns, car les règles peuvent être déclenchées en différentes étapes, comme par exemple : (i) détection d'une instance de pattern ; (ii) analyse des compromis d'implémentation ; et (iii) génération de la variante la plus adaptée.

L'objectif de la base de règles d'implémentation d'un pattern est de représenter les connaissances des implémenteurs : elle analyse les choix réalisés par le concepteur lors de l'instanciation de ce pattern et génère le code exécutable correspondant. Les bases de règles que nous avons écrites dans PatternGen analysent chaque instance d'un pattern en trois étapes :

Contexte. Analyse des potentialités du contexte par rapport aux patterns choisis. Par exemple, certains cadres d'application intègrent déjà certaines implémentations de patterns de conception, comme 'NxProxy' de OpenStep ou 'DependentPart' de VisualWorks, qui mettent en œuvre les patterns Procuration et Observateur, respectivement. Même si les classes proposées n'englobent pas toutes les variantes d'implémentation prévues, elles peuvent s'appliquer à certains cas et doivent être prises en considération lors du codage.

Interaction. Comme plusieurs instances d'un même pattern peuvent être présentes dans une même application, il est important de vérifier si elles ne peuvent pas partager une partie de leur implémentation. Cela peut être fait par la création de classes abstraites. Il est intéressant aussi d'éviter la génération de code redondant (e.g. une classe qui joue un rôle particulier dans une instance d'un pattern qui hérite d'une classe jouant le même rôle, mais qui ne participe pas à cette instance du pattern).

Compromis spécifiques. Dans cette étape, le code source est généré. Certains choix de génération sont spécifiques à une instance particulière d'un pattern. La combinaison de ces choix permet de choisir le code le plus adapté.

III. Exemples de représentation de patterns

Les exemples d'utilisation que nous allons de décrire montre la suite d'opérations permettant la représentation de quatre patterns dans PatternGen et leur utilisation au cœur d'une application.

L'utilisation de patterns dans PatternGen passe d'abord par l'extension du méta-modèle décrit précédemment, avec la spécification des patterns désirés. Ensuite, on fournit les règles de production analysant les modèles produits par le formalisme défini afin d'en déduire la génération de code impliquant les meilleurs compromis entre les variantes induites par la spécification des solutions correspondantes au pattern utilisé.

Les quatre patterns présentés ici ont été extraits du catalogue de Gamma et al. [Gamma *et al.*, 1995]. Il s'agit de deux patterns structurels, Composite et Proxy et de deux patterns de comportement, Observateur et Stratégie.

La présentation de l'utilisation de chacun des patterns suit une même structure. Elle commence par une description rapide du pattern en question. Ensuite, la partie du méta-modèle concernant la représentation du pattern sera présentée. La présentation est complétée par la description de la logique d'exécution des règles d'implémentation du pattern.

1. Observateur[1]

Le pattern **Observateur**, aussi connu sous le nom Dépendance, explicite la relation entre un objet quelconque dit Sujet et un mécanisme observant les changements d'état du premier appelé Observateur. L'interface graphique d'une application est un exemple simple d'application du pattern Observateur : elle affiche à l'écran des données de l'application mémorisées par les objets représentant l'état d'exécution. Pour assurer la cohérence de l'interface, tous les éléments qui exhibent une même donnée doivent être notifiés simultanément lors de sa modification.

Le célèbre cadre d'applications MVC [Krasner et Pope, 1988] de Smalltalk 80 répond de ce pattern. Il met en jeu une relation 1-N entre le modèle (Sujet) et ses dépendants (Observateurs) de façon à informer ces derniers de tout changement d'état du premier. Cette solution simple assure la cohérence entre les éléments de l'interface graphique et les objets de l'application.

[1] Ce pattern a été rapidement introduit précédemment (page 20). Toutefois, dans ce chapitre, il était nécessaire de détailler ses compromis d'implémentation, ce que nous a mené à le décrire à nouveau. Nous nous excusons pour cette petite redondance.

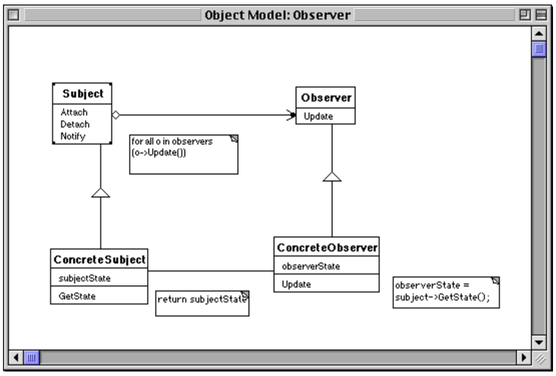

Figure 43 - Le pattern Observateur

Le pattern Observateur est formé par quatre classes, présentées dans la Figure 43 :

1. Sujet : garde la collection des observateurs et fournit des méthodes pour attacher et détacher un observateur ;

2. Observateur : fournit les méthodes qui seront appelées lors d'une notification de changement ;

3. Sujet Concret : mémorise l'état d'une donnée et notifie à ses observateurs les changements qui la concernent ;

4. Observateur Concret : garde une référence vers le sujet et met en œuvre, lors de chaque modification, les méthodes de prise en compte du changement de la donnée.

Cette structure évite de compliquer les méthodes de l'application définissant l'état des sujets. Les sujets gardent la référence de tous leurs observateurs et, lorsqu'un changement les affecte, ils déclenchent le mécanisme de notification.

En raison de la notoriété du cadre MVC, Observateur est un des patterns les plus connus et probablement celui qui présente le plus de variantes d'implémentation répertoriées (cf. sa description dans [Gamma *et al.*, 1995]). En voici quelques-unes :

Observation de plusieurs sujets. Dans certains cas, l'observateur dépend de plus d'un sujet. Il doit alors déterminer le sujet qui est à l'origine de la notification.

Déclenchement de la mise à jour. Le déclenchement de la notification peut se faire soit par le sujet, à travers les méthodes qui modifient son contenu, soit par les clients qui appellent ces mêmes méthodes. Ces deux options présentent des inconvénients : la première s'avère peut efficace lors de changements successifs (la notification est dans ce cas appelée plusieurs fois), et la deuxième exige que tous les clients déclenchent la notification, une contrainte facilement oubliée lors de la maintenance de l'application.

Protocoles de mise à jour. Le processus de notification peut simplement informer les observateurs d'un changement, et ceux-ci se chargent alors de chercher les informations qui les intéressent, ou d'envoyer aux observateurs, lors de la notification, des informations détaillées sur le changement.

Spécification explicite de l'intérêt. Les observateurs peuvent manifester un intérêt explicite sur un aspect du sujet, et ne souhaiter être informé d'un changement que lorsque cet aspect précis est concerné. Cette option peut améliorer l'efficacité de la mise à jour.

Combinaison des classes Sujet et Observateur. Une dernière option, utilise un tiers objet, le gestionnaire de changements (unique dans le système - par ailleurs une instance du pattern Médiateur), qui se charge des relations entre observateurs et sujet.

L'adoption de certains de ces compromis peut résulter en des variantes très différentes de l'implémentation originale de MVC. Mais, toutes ces variantes respecteront toujours la contrainte implicite dans ce pattern de conception : les observateurs doivent être notifiés lors de la modification du sujet.

1.1 Méta-modèle

La Figure 44 représente une extension du méta-modèle de base, prenant en compte le pattern Observateur (elle est similaire à la Figure 41, ces deux figures sont des plans différents du même méta-modèle).

Le méta-individu « Observer » a quatre méta-relations avec le méta-individu « Class ». Ces méta-relations correspondent à la désignation des quatre classes participantes. Les cardinalités indiquent que chaque instance de ce pattern doit avoir au moins une classe pour sujet et une classe pour observateur (une règle d'intégrité, spécifiée par ailleurs empêchera qu'une même classe joue le rôle de sujet et observateur). La participation des classes « sujet réel » et « observateur réel » est facultative, ce qui est noté par les cardinalités minimales 0.

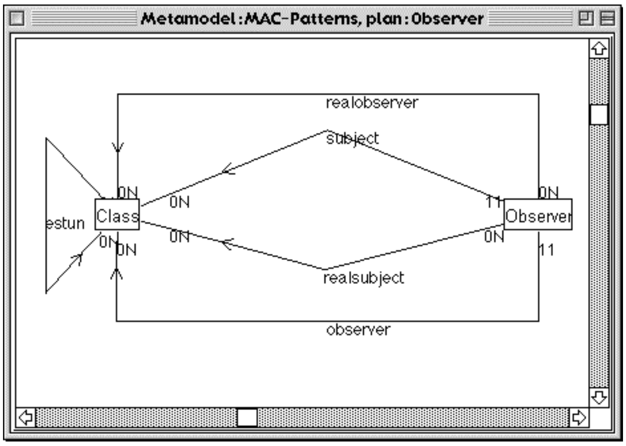

Figure 44 - Représentation du pattern Observateur

Le méta-individu 'Observer' possède trois rubriques (cachées sur la Figure 44) : 'codage', 'aiguillage' et 'notification' ; qui correspondent, respectivement, à trois compromis d'implémentation : combiner ou non le *Sujet* et *l'Observateur*, le mode de déclenchement de la mise à jour et le protocole de notification. Chacune de ces rubriques précise une liste d'options : respectivement *(Automatique, Classique, MVC et Gestionnaire de Changements)* ; *(Client* et *Sujet)* ; et *(Push* et *Pull)*. Elles pilotent le dialogue d'édition montré dans la Figure 7. La méta-relation 'Real Observer' possède aussi une rubrique, *Intérêt*, qui correspond au compromis d'implémentation « spécification explicite de l'intérêt » décrit antérieurement.

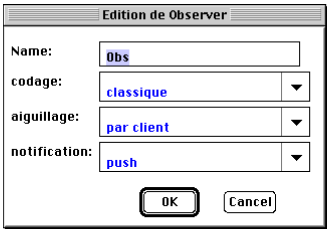

Figure 45 – Edition d'une instance du pattern Observateur

Lors de l'instanciation du pattern dans un modèle d'application, le choix des compromis d'implémentation sera effectué par le choix d'une seule option parmi les listes décrites précédemment et en suivant les flèches du diagramme, qui indiqueront les classes participantes à la réalisation du pattern.

1.2 Règles de mise en œuvre

Les règles qui concernent le pattern Observateur s'appliquent en trois temps, conformément aux indications du § II.3. D'abord, le contexte est analysé pour prendre en considération les classes de VisualWorks ('Model' et 'DependentPart') qui mettent déjà en œuvre ce pattern. Ensuite, une réutilisation possible des méthodes de plusieurs instances du pattern dans le modèle sera envisagée. Finalement, les compromis d'implémentation seront pris en compte, et le code sera généré.

i) Contexte

Les deux classes Smalltalk qui mettent en œuvre ce pattern ne résoudront pas tous les compromis d'implémentation prévus. Leur utilisation dépend de plusieurs facteurs, et ne sera envisagée que lorsque le choix de codage le spécifie ou laisse les décisions aux règles (codage automatique).

ii) Interaction

Quand il existe plusieurs instances du pattern Observateur, la création d'une classe abstraite peut simplifier le codage. Pour la créer, il est nécessaire que les participants du pattern répondent à certaines contraintes :

1. Deux sujets peuvent partager leurs méthodes s'ils ont une super-classe commune et s'ils implémentent de la même manière leur collection d'observateurs.

2. Deux observateurs peuvent partager leurs méthodes s'ils ont une super-classe commune et s'ils observent le même sujet (i.e. font partie du même pattern).

iii) Compromis spécifiques

La dernière étape consiste à prendre en compte les compromis d'implémentation, décrits précédemment et à générer le code Smalltalk :

Observation de plusieurs sujets. Ce premier compromis est vérifié lorsqu'une classe participe à plusieurs instances de ce pattern. Dans ce cas, il est nécessaire de remplacer la variable *sujet* des classes engendrées par les instances qui participent à '*Observer*', par une collection.

Déclenchement de la mise à jour. La méthode de notification est déterminée par la rubrique Aiguillage. Elle peut être déclenchée de deux manières possibles, par le sujet, lors d'un changement d'état ou par les classes clientes qui l'auraient provoqué. La notification d'un changement sera réalisée par les opérations d'accès aux variables d'instance de la classe sujet.

Au moins deux cas sont possibles :

1. Chaque méthode d'affectation de variable notifie le changement qu'elle provoque ;
2. Une seule méthode, décrivant un ensemble d'affectations spécifiant le changement génère une seule notification.

La notification peut aussi être déclenchée par les classes clientes qui utilisent l'une ou l'autre de ces méthodes d'affectation. Cependant, cette option exige que toutes les classes clientes notifient effectivement les changements, ce qui exige une plus grande discipline de l'ensemble des implémenteurs, ou encore, complexifie les règles de génération. Dans ce cas précis, les règles pourraient assurer que toutes les classes qui modifient le sujet fassent ensuite appel à la notification. En raison de certaines difficultés dues principalement à l'absence de types en Smalltalk, cette vérification n'a pas été mise en œuvre.

Protocoles de mise à jour. Ce compromis est déterminé par le choix parmi les options *Pull* et *Push* de la rubrique Notification. Dans le premier cas, seulement un message de changement sera envoyé. Dans le deuxième, une information détaillée sur le changement accompagnera le message.

Spécification explicite de l'intérêt. Un observateur peut manifester un intérêt particulier sur certains aspects du sujet. Cet intérêt peut être décrit grâce à la rubrique Intérêt de la méta-relation « Real Observer ». Dans ce cas, la liste d'observateurs utilisera dictionnaire pour notifier les observateurs à bon escient.

Combinaison des classes Sujet et Observateur. La rubrique Codage détermine l'utilisation d'un gestionnaire de changements. Elle peut aussi forcer l'utilisation du cadre d'applications MVC, présent dans Smalltalk ou induire la réalisation d'un mécanisme de contrôle complètement indépendant. L'option *Automatique* tranche parmi ces options

2. Composite

Le pattern **Composite** décrit la mise en œuvre de structures de représentation de hiérarchies parent/enfants. Il permet que de classes clientes traitent de la même manière objets simples et complexes, grâce à l'utilisation d'une interface de méthodes commune.

Ce pattern est appliqué, par exemple, dans la mise en œuvre de figures complexes (i.e. groupement de figures élémentaires) d'un éditeur graphique [voir Johnson, 1992]. Bien que composée par d'autres figures, une figure complexe est vue et manipulée comme étant une seule figure élémentaire. Ainsi, les classes clientes ne différentient pas ces figures des figures élémentaires et les traitent de manière identique.

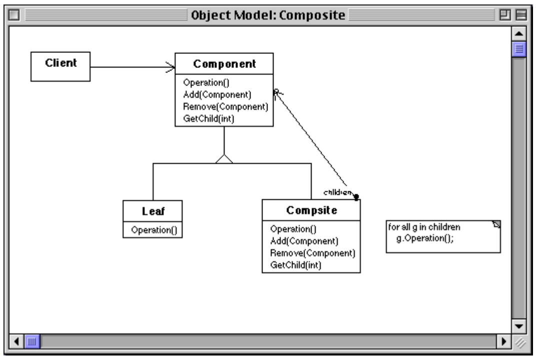

Figure 46 - Le pattern Composite

Le pattern Composite est formé par trois classes, présentées dans la Figure 46 :

1. Composant : cette classe déclare l'interface des méthodes définissant le comportement par défaut, l'accès aux enfants et éventuellement, l'interface d'accès au parent ;

2. Feuille : définit le comportement des objets élémentaires (enfants) dans la composition. Les feuilles ne possèdent pas d'enfants ;

3. Composite : définit le comportement des composites (dotés d'enfants), stocke les enfants et implémente les opérations liées à la manipulation d'enfants.

Cette structure permet qu'une classe cliente envoie un même message à différents objets composants, sans connaître la nature exacte de ceux-ci. Lorsqu'un objet composite reçoit ce message, il le retransmet à tous ses composants.

Le pattern Composite dispose de plusieurs compromis d'implémentation :

Référence explicite au parent. Dans certains cas, les enfants peuvent vouloir garder une référence vers leur parent, ce qui leur permet de remonter dans la hiérarchie. Cette option implique en une contrainte implicite, qui doit assurer qu'un enfant a comme parent un composite qui, en retour, l'a comme enfant.

Partage de composants. Le partage d'enfants entre parents peut permettre de réduire l'occupation de la mémoire nécessaire à leur stockage. Cependant, cela engendre une plus grande complexité lorsque les enfants gardent une référence explicite à leurs parents. Dans ce cas, la référence doit être stockée dans une collection, ce qui rend difficile de remonter dans la hiérarchie.

Ordonnancement des enfants. Certaines implémentations demandent un ordonnancement des enfants. Ceci est nécessaire lorsque l'ordre est important, comme dans le cas de plusieurs figures superposées.

Suppression des composants. Qui doit effacer les enfants d'un composite lorsque celui-ci est lui-même effacé ? Si le langage d'implémentation ne dispose pas de ramasse-miettes, il est préférable que le composite s'en occupe. Cette option est toutefois indésirable si le partage d'enfants est accepté.

Déclaration des opérations de gestion des enfants. S'il est claire que la classe Composite doit implémenter les opérations d'ajout et de suppression d'enfants, il est cependant difficile de déterminer si la classe Composant doit ou non déclarer l'interface de ces opérations. Si elle les déclare, elle assure une homogénéité de l'interface des enfants, mais permet en même temps l'utilisation d'opérations incohérentes, tel que l'ajout d'un enfant à une feuille.

Utilisation d'un cache pour améliorer des performances. Lorsque les enfants d'un composite sont fréquemment parcourus, celui-ci peut vouloir stocker un cache des informations recherchées. Ceci accélère l'accès à ces informations, mais exige une nouvelle contrainte : tout changement concernant les enfants doit être notifié au parent.

Maximisation de l'interface composant. Cette dernière option concerne l'interface de la classe composant. Un des objectifs du pattern est de permettre que des classes clientes traitent indifféremment parents et enfants. Pour cela, la classe composant doit déclarer un maximum d'opérations, même si certaines opérations n'ont pas de sens pour cette classe.

2.1 Méta-modèle

La Figure 47 représente l'extension du méta-modèle de base, prenant en compte le pattern Composite. Le méta-individu 'Composite' possède deux méta-relations avec le méta-individu 'Class', et une méta-relation avec le méta-individu 'Method'. Ces méta-relations correspondent à la désignation des deux classes participantes, 'composite' et 'composant' et des méthodes de 'interface'.

Chaque instance de ce pattern doit avoir une classe pour 'composant' et au moins une classe pour 'composite'. La participation de méthodes de 'interface' est facultative, ce qui est noté par la cardinalités '0N'. L'indication des classes 'feuilles' n'est pas nécessaire, d'une part parce qu'elles ne sont pas modifiées par l'application d'un pattern et d'autre part parce qu'elles sont facilement reconnaissables dans le modèle (les sous-classes de 'composant' qui ne sont pas des composites).

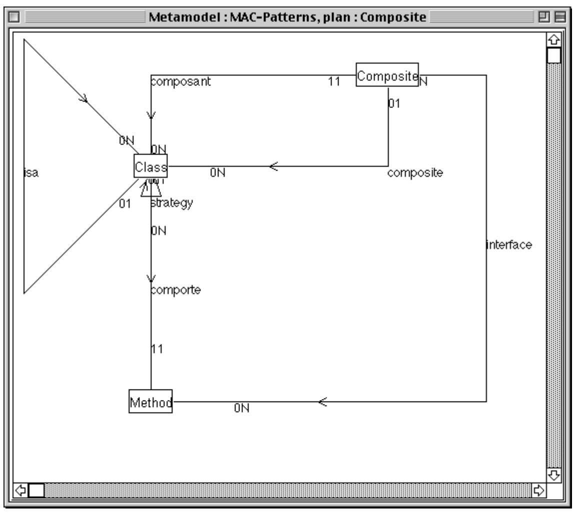

Figure 47 – Représentation du pattern Composite

Le méta-individu 'Composite' possède six rubriques : *Composite_Reference*, *Components_List*, *Components_Release*, *Components_Keeper*, *Share_Components* et *Maximize_Interface*.

Les quatre premières rubriques précisent des listes d'options. Respectivement : *Explicite* ou *None* ; *Ordered* ou *Simple* ; *Composite* ou *Components* ; et *Composite* ou *Components*. Ces rubriques correspondent à quatre compromis d'implémentation : « référence explicite au parent » ;

« ordonnancement des enfants » ; « suppression des composants » ; et « déclaration des opérations de gestion des enfants ».

Les deux dernières rubriques sont de type booléen et correspondent, respectivement, à deux compromis d'implémentation : « partage de composants » ; et « maximisation de l'interface de composant ».

Ces rubriques pilotent le dialogue d'édition montré dans la Figure 48.

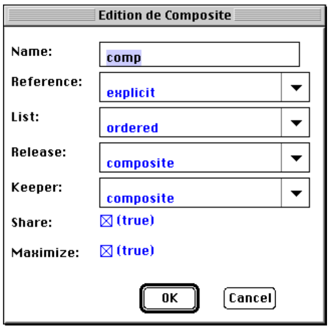

Figure 48 – Edition d'une instance du pattern Composite

Lors de l'instanciation du pattern Composite, les méthodes d'interface pourront être indiquées grâce à une relation (de type 'interface') entre l'instance du pattern et une méthode de la classe composant.

2.2 Règles de mise en œuvre

Les règles de mise en œuvre du pattern Composite s'appliquent en trois temps.

i) Contexte

La mise en œuvre de ce pattern n'est pas influencée par le contexte d'implémentation, conforme sa description.

L'absence de l'indication (par une relation) d'une classe participante jouant le rôle de 'Composite' dans le modèle implique la création de cette classe. Le nom de cette nouvelle classe est celui de la classe 'Composant', précédé par l'appellation « Composite ».

ii) Interaction

Contrairement au pattern Observateur, le pattern Composite n'offre pas de possibilité de réutilisation de code entre différentes instances de ce pattern. Ceci pour deux raisons. D'une part le lien de spécialisation entre les classes Composant et Composite rend impossible la création d'une classe abstraite permettant le partage de méthodes d'ajout et de suppression de composants. D'autre part, les méthodes d'interface sont spécifiques à la classe Composant et donc ne peuvent pas être partagées avec d'autres instances de ce pattern.

Toutefois, une instance du pattern Composite peut influencer l'implémentation d'autres patterns. Plus précisément, l'implémentation d'une instance du pattern Itérateur peut être guidée si le rôle de celui-ci est de parcourir une collection de composants.

iii) Compromis spécifiques

Six compromis sont pris en considération par cette étape, et pilotent la génération de code :

Référence explicite au parent. Ce premier compromis influence la mise en œuvre des méthodes d'ajout et de suppression d'un composant, et déclenche la création d'une variable dans la classe composant et des méthodes d'accès à cette variable.

Partage de composants. Ce compromis interfère dans l'implémentation de la référence explicite au parent. Plus précisément, le composant doit stocker une liste de parents et mettre en œuvre des méthodes permettant la gestion de cette liste.

Le partage de composants modifie aussi la manière de supprimer un composant, qui ne peut pas être réalisée par la classe Composite. La suppression doit être réalisée par le composant lui-même, lorsqu'il n'a plus de référence vers un parent.

Ordonnancement des enfants. Parfois, l'ordre d'inclusion des composants dans un composite est important. Dans ce cas, la liste de composants utilisera une collection ordonnée.

Suppression des composants. Il existe deux options de mettre en œuvre la suppression d'un composant. La première est de demander au composant lui-même de se retirer de la liste de composants. Cette option est particulièrement souhaitable lorsque le composant peut être présent dans différentes listes. La deuxième option est de permettre au composite de retirer les composants de sa liste et laisser le ramassage de miettes s'occuper de libérer sa place en mémoire.

Déclaration des opérations de gestion des enfants. La rubrique *Components_Keeper* détermine quelle classe doit mettre en œuvre les méthodes d'ajout et de suppression de composants. Les méthodes restent inchangées dans ces deux cas.

Maximisation de l'interface composant. Ce dernier compromis est vérifié de deux manières. Premièrement, l'existence d'une relation (de type 'interface') entre l'instance du pattern et une méthode permet d'indiquer quels messages l'objet composite doit renvoyer à ses composants.

Deuxièmement, la rubrique *Maximize_Interface* peut indiquer que le renvoie doit être effectué pour toutes les méthodes de la classe composant.

3. Stratégie

Le pattern Stratégie, aussi connu sous le nom de Politique, préconise une indépendance entre une classe cliente et une famille d'algorithmes (ensemble d'algorithmes ayant un objectif commun) utilisée par cette classe.

Cette indépendance est désirable d'une part car elle décharge la classe cliente, qui ne supporte qu'un algorithme à la fois. D'autre part, elle permet la modification d'un algorithme sans que cela affecte la classe client.

L'implémentation de la validation de champs de saisie dans ObjectWindows de Borland [Borland, 1994] est un exemple d'utilisation de ce pattern. En effet, les algorithmes de validation sont implémentés par des classes indépendantes et sont donc dissociés des champs de saisie. Lorsqu'un champ de saisie est créé, il est associé à un algorithme de validation, implémenté par une des sous-classes de la classe « Validator », qui implémentent les algorithmes les plus utilisés : valeur minimale, intervalle, etc.

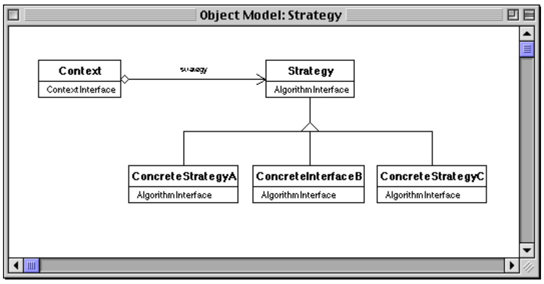

Figure 49 - Le pattern Stratégie

Le pattern Stratégie est formé par trois classes, présentées dans la Figure 49 :

1. Stratégie : déclare une interface commune à tous les algorithmes représentés.
2. Stratégie concrète : implémente l'algorithme, selon l'interface
3. Contexte : gère la référence à une stratégie

Définition des interfaces Stratégie et Contexte. La classe contexte doit assurer, aux stratégies, l'accès à ses données. Ceci peut être fait de deux manières : (i) les passer comme paramètre et (ii) passer une référence à l'objet client comme paramètre.

Des Stratégies arguments de templates. Dans le cas précis de C++, il est possible de composer les classes clientes et les stratégies à l'aide de templates.

Objets Stratégie optionnels. Une manière de simplifier la classe contexte est d'employer des stratégies facultatives. A défaut d'une référence vers une stratégie concrète, le contexte utilise un comportement par défaut.

3.1 Méta-modèle

La Figure 50 représente l'extension du méta-modèle de base, prenant en compte le pattern Stratégie. Le méta-individu 'Strategy' a deux méta-relations avec le méta-individu 'Class', et trois méta-relations avec le méta-individu 'Method'. Ces méta-relations correspondent à la désignation des deux classes participantes, et de trois méthodes, correspondant aux méthodes suivantes : 'ContextInterface()', 'AlgorithmeInterface()' et l'algorithme utilisé par défaut.

Les cardinalités indiquent que chaque instance de ce pattern doit avoir une classe pour contexte, alors que l'existence d'une classe stratégie est facultative. La participation des trois méthodes est aussi facultative.

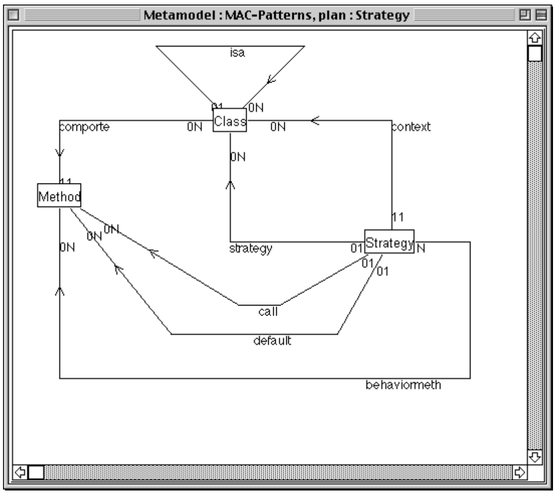

Figure 50 - Représentation du pattern Stratégie

Le méta-individu 'Strategy' possède trois rubriques : 'Codage', 'Arguments' et 'Strategy'. Les deux premières correspondent à deux compromis d'implémentation, l'utilisation de templates en C++ et la définition des interfaces entre Stratégie et Contexte. Elles précisent, chacune, une liste d'options : (*Classique et Templates*) ; (*Reference et Valeurs*) ; respectivement. La troisième rubrique correspond au sélecteur des méthodes 'AlgorithmInterface()', et est utilisée en absence de la définition de la classe Stratégie. Ces rubriques pilotent le dialogue présenté par la Figure 51.

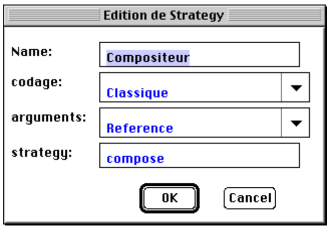

Figure 51 - Edition d'une instance du pattern Stratégie

Lors de l'instanciation du pattern dans un modèle, le compromis d'implémentation, l'utilisation d'un comportement par défaut, sera vérifié en fonction de l'existence d'une relation (de type 'default') entre l'instance du pattern et une méthode.

3.2 Règles de mise en œuvre

Les règles de mise en œuvre du pattern Stratégie s'appliquent également en trois temps.

i) Contexte

Une alternative à l'implémentation de ce pattern consiste à utiliser des templates. Cependant, cette alternative est propre au langage C++ et n'est pas exploitée par les règles.

L'absence de l'indication (par une relation) d'une classe participante jouant le rôle de 'Stratégie' dans le modèle implique en la création de la structure hiérarchique des classes 'Stratégie'. Cette facilité crée une classe fille correspondant à chacune des méthodes indiquées comme

'AlgorithmeInterface()' et transfert ces méthodes vers les classes créées. Le transfert implique en une modification de la méthode, en ce qui concerne l'envoi de messages et l'accès à de variables.

ii) Interaction

Le pattern Stratégie propose peu de possibilités de partage de classes et méthodes entre deux de ses instances. Le seul partage envisageable concerne la structure des classes Stratégie, qui pourrait implémenter différents algorithmes, de différents contextes.

Cette solution n'offre aucun avantage en ce qui concerne la réutilisation de méthodes et peu rendre difficile la compréhension de l'implémentation, en raison de l'appellation des classes. Elle n'a pas été envisagée.

iii) Compromis spécifiques.

Dans cette dernière étape, trois compromis sont pris en considération avant la génération du code Smalltalk :

Définition des interfaces Stratégie et Contexte. Ce compromis est déterminé par le choix parmi les options *Référence et Valeurs* de la rubrique 'Arguments'. Dans le premier cas, l'appel aux méthodes qui mettent en œuvre un algorithme ('AlgorithmeInterface()') auront comme argument une référence à l'objet client (le Contexte).

Des Stratégies arguments de templates. Ce compromis, bien que vérifié à partir de la rubrique 'Codage', n'est pas pris en considération lors de la génération de code en Smalltalk.

Objets Stratégie optionnels. Ce dernier compromis peut être vérifié par l'utilisation d'un relation de type ('algorithme par …) 'défaut' avec une méthode. Dans ce cas, la méthode d'appel à l'algorithme ('ContextInterface()') sera modifiée de manière à, lorsqu'il n'a pas d'objet Stratégie, faire appel à une méthode locale (établie par la relation) ou créer un objet Stratégie (établie par une relation de même type).

4. Proxy

Le pattern Proxy décrit une technique de remplacement d'un objet, dit « sujet », par un deuxième objet, sa procuration. Cette technique permet que l'objet sujet ne soit accédé que lors de son utilisation réel, et non pendant sa création.

Cette technique est recommandée lors de l'utilisation d'objets d'accès coûteux, tels que des images graphiques ou des objets distribués. Elle peut être employée aussi lors de l'utilisation d'objets protégés (i.e. des objets demandant des droits d'accès). Une troisième utilisation d'objets procureurs consiste à garder des références intelligentes à des objets partagés [voir Edelson, 1992].

Les classes NXProxy de NextStep [Next, 1994] constituent un exemple d'implémentation du pattern Proxy. Les objets issus de ces classes fonctionnent comme des représentants locaux d'objets distribués. Lorsqu'un objet NXProxy reçoit un message adressé à l'objet qu'il représente, il l'encapsule et l'envoi à l'objet concerné.

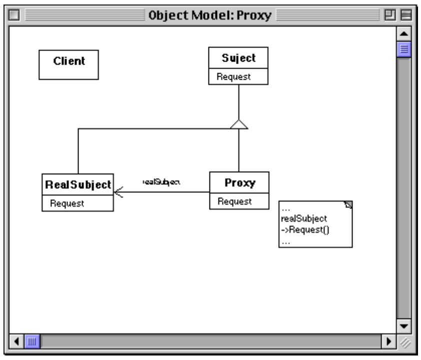

Figure 52 - Le pattern Proxy

Le pattern Proxy est composé de trois classes participantes, présentées dans la Figure 52 :

- Proxy : conserve une référence qui permet au proxy d'accéder au sujet réel, fourni une interface semblable à celle du sujet et contrôle l'accès au sujet réel ;

- Sujet : définit l'interface pour le sujet réel et pour le Proxy. De cette manière, le proxy peut être utilisé partout où le sujet réel est attendu ;

- Sujet Réel : l'objet réel représenté par le proxy.

Contrairement aux patterns présentés auparavant, peu de compromis d'implémentation pilotent l'implémentation de proxy. Toutefois, celle-ci est dirigée par la nature du sujet réel référencé par le proxy (objets coûteux, distants, protégés, etc.). Les compromis proposés sont :

Les procurations n'ont pas toujours besoin de connaître le type de sujets. Si le procureur n'a pas besoin de créer le sujet réel (i.e. de le connaître), alors il est possible de partager les classes procureur entre plusieurs sujets réels.

Surcharge de l'opération d'accès aux membres (C++). Le langage C++ permet la surcharge de l'opérateur d'accès aux membres ('->'). Grâce à cette technique, il est possible de créer des procureurs qui fonctionnent exactement comme des pointeurs. Cependant, elle ne permet pas au procureur de connaître les opérations qui sont renvoyées au sujet réel.

Utilisation de 'doesNotUnderstand'. Le langage Smalltalk dispose de mécanismes permettant le contrôle de l'envoi de messages. Ces mécanismes peuvent être utilisés dans l'implémentation d'une classe procureur. Plus précisément, cette classe n'implémente que la méthode 'doesNotUnderstand', actionnée lors de l'envoi de messages inconnus à un objet. Le rôle de cette méthode sera de renvoyer tout message envoyé à un objet proxy vers le sujet réel.

Référence au sujet. Ce dernier choix d'implémentation concerne la manière de faire référence au sujet réel, avant que celui-ci soit instancié. Cette référence dépend de la nature du sujet réel et peut être un nom de fichier (dans le cas d'une image), une adresse (objet distribué) ou un mode d'accès (objet protégé).

4.1 Méta-modèle

L'extension du méta-modèle de base, prenant en compte le pattern Proxy est présenté par la Figure 53. Trois méta-relations lient le méta-individu 'Proxy' au méta-individu 'Class', qui correspondent à la désignation des trois classes participantes.

Chaque instance du pattern Proxy doit avoir (cardinalité 11) une classe pour sujet réel et peut avoir (cardinalités 01) une classe pour sujet et une classe pour proxy.

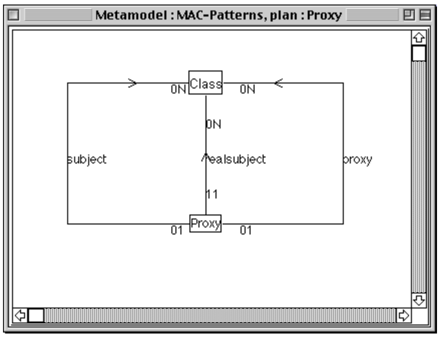

Figure 53 - Représentation du pattern Proxy

Le méta-individu 'Proxy' possède deux rubriques : 'Implementation' et 'Nature'. La première correspond à deux compromis d'implémentation, la surcharge de l'opérateur d'accès au membres et l'utilisation de 'doesNotUnderstand'. Elle précise une liste d'options : (*'Classique', 'doesNotUnderstand' et 'opérateur ->'*). La deuxième rubrique correspond à la nature de l'implémentation et est utilisée pour préciser ci le sujet réel est un objet distant, un objet protégé ou un objet d'accès coûteux. Ces deux rubriques pilotent le dialogue d'édition présenté par la Figure 54.

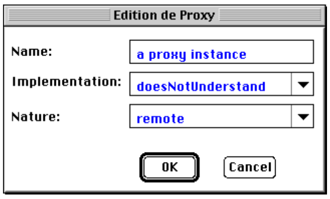

Figure 54 - Edition d'une instance du pattern Proxy

Comme pour les patterns précédents, le choix des compromis d'implémentation sera effectué en suivant les flèches du diagramme, indiquant les classes participantes et par le choix d'une option parmi les deux listes décrites auparavant.

4.2 Règles de mise en œuvre

En un premier temps, les règles de mise en œuvre du pattern Proxy analysent le contexte pour prendre en considération la présence de classes qui mettent à œuvre ce pattern (e.g. 'CorbaObjRef' de Corba). En un deuxième temps, une réutilisation possible de l'implémentation plusieurs instances du pattern dans le modèle sera envisagée. En un troisième temps, les compromis d'implémentation seront pris en compte, et le code sera généré.

i) Contexte

Dans le cas précis de Smalltalk VisualWorks, l'utilisation d'un proxy distant dépend de la présence d'un paquetage de communication, comme Orbix, qui implémente un proxy abstrait, la classe 'GenericProxy'. La spécialisation de cette classe doit suivre une implémentation très précise, qui doit respecter certaines contraintes, comme l'utilisation de la méthode 'doesNotUnderstand'.

En absence de l'indication d'une classe jouant le rôle de 'proxy' ou de 'sujet' implique en la création de celles-ci. Plus précisément, une classe abstraite (le sujet) sera introduite dans le modèle, comme super-classe du sujet réel. Celui-ci aura son nom précédé par le préfixe 'Real'. En même temps, une deuxième classe sera introduite dans le modèle (le proxy, sous-classe du sujet). Son nom sera celui du sujet, combiné avec le suffixe 'Proxy'.

ii) Interaction

La classe proxy peut être partagée par différentes instances de ce pattern. Le partage est possible si les différentes classes proxy implémentent le renvoie de méthodes par 'doesNotUnderstand' et ne connaissent pas leur sujet réel.

Le partage de la classe proxy représente le premier compromis d'implémentation vérifié ici : les procurations n'ont pas toujours besoin de connaître le type des sujets.

iii) Compromis spécifiques

La dernière étape consiste à prendre en compte les compromis d'implémentation décrits au préalable et à générer le code Smalltalk :

Surcharge de l'opération d'accès aux membres (C++). Ce compromis est propre au langage C++ et n'est pris en considération ici.

Utilisation de 'doesNotUnderstand'. Ce compromis peut être mis en œuvre en Smalltalk, ainsi qu'en Objective C. Si ce compromis est choisi, la classe proxy n'implémente que la méthode 'doesNotUnderstant' de manière à aiguiller les messages qui lui sont adressés à son sujet réel.

Dans le cas contraire, la classe proxy implémentera une méthode de renvoie pour chaque méthode du sujet réel.

Référence au sujet. Ce compromis est vérifié par la rubrique 'nature'. La référence au sujet (avant que celui-ci soit créé) peut être réalisé par un nom de fichier (objet d'accès coûteux), par une méthode de création ou par un gestionnaire de références (les objets distants de Corba).

IV. Implémentation

PatternGen a été réalisé avec MétaGen. Comme présenté dans le chapitre précédent, MétaGen est un méta-outil, i.e. un outil capable de (i) permettre la représentation d'un langage de modélisation dans un *méta-modèle* et de (ii) générer automatiquement, à partir de ce méta-modèle, l'outillage support de ce langage. L'utilisation de MétaGen dans le développement d'un outil de mise en œuvre de patterns présente plusieurs bénéfices :

Premièrement, MétaGen génère automatiquement l'outillage du langage de modélisation MAC à partir de son méta-modèle. L'outillage généré est composé par deux sortes d'éditeurs, de diagrammes et de listes (*browsers*), par un système d'aide hypertexte et par un générateur de rapports.

L'éditeur de diagrammes généré offre plusieurs possibilités de masquage et de « zooming », qui permettent de cacher ou de mettre en évidence certains aspects d'un modèle. Ainsi l'éditeur peut, par exemple, masquer les instances de pattern présentes dans un modèle de conception ou l'isolement d'une instance de pattern et de ses classes participantes.

Deuxièmement, MétaGen génère des boîtes de dialogue qui permettent l'édition des compromis spécifiques à chaque instance de pattern (e.g. Figure 45), et des autres concepts modélisés (classes, méthodes, etc.). La génération de ces boîtes de dialogue est aussi réalisée automatiquement à partir des informations du méta-modèle.

Troisièmement, dans la mesure où MétaGen supporte la *méta-modélisation incrémentale*, l'outillage support d'un langage évolue en fonction des modifications réalisées sur le méta-modèle. Ainsi, lorsqu'un nouveau pattern est ajouté au méta-modèle, il sera disponible pour les modèles déjà existants.

Finalement, MétaGen a été employé de diverses manières dans la le développement de plusieurs facilités qui permettent la simplification des bases des règles, qui seront présentés dans les sections suivantes. Cette présentation est suivie par plusieurs exemples de règles.

1. Construction des bases de règles

L'objectif des règles de génération est de produire le code source le plus adapté pour une situation donnée. En général, la partie '*prédicats*' des règles contient les choix d'implémentation d'un pattern et la partie '*actions*', le code source associé. Pour simplifier l'écriture des actions et plus précisément la génération des méthodes, on utilise un mécanisme de macro génération, à base de *fragments*, qui sera présenté par la suite.

Pour ne pas surcharger le modèle de conception de méthodes et d'attributs générés (à partir des instances de pattern), les règles sont appliquées à une copie du modèle original. Contrairement à ce dernier, la copie du modèle n'est pas décrite en langage MAC, mais grâce à une variation de celui-ci, adaptée à un langage de programmation. Les variations de MAC, adaptées au langages Smalltalk et Java, sont appelées MAC-S et MAC-J.

L'utilisation d'un langage de modélisation adapté au langage cible apporte deux bénéfices. Premièrement, elle assure l'indépendance entre le modèle de conception, qui contient les instances de patterns et les langages d'implémentation. Celle-ci est une des caractéristiques importantes des patterns de conception. Deuxièmement, elle permet de décharger la base de règle d'opérations spécifiques à ce langage, comme la création d'un fichier texte correspondant au modèle.

Les fragments, ainsi que les deux variations du langage MAC, MAC-S et MAC-J, seront détaillés dans les sections suivantes.

1.1 Les fragments

Les fragments sont des textes ayant vocation à représenter des morceaux de code répondant de la syntaxe du langage utilisé. Les unités syntaxiques caractéristiques du langage, ainsi que des marqueurs spécifiques, appelés paramètres, seront substituables lors de l'utilisation du fragment, par des chaînes de caractères spécifiques à l'usage particulier du fragment.

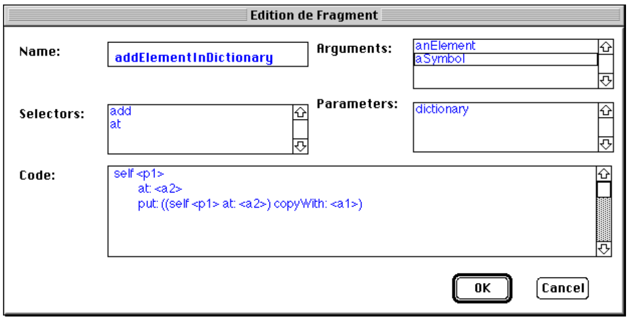

Figure 55 – Edition d'un fragment

Les fragments que nous présentons ici sont différents des fragments de PatternTool [Meijers, 1996]. Il n'y a pas de correspondance entre un fragment et une méthode composante d'un pattern. L'un est un instrument pour la génération de code, qui est aussi utilisé dans d'autres bases de règles de génération de code ; l'autre au contraire est spécifique à l'implémentation d'un pattern et peut être composée par plusieurs fragments

La Figure 55 présente un fragment élémentaire, l'ajout d'un élément à une liste indexée (un dictionnaire Smalltalk). Ce fragment permet la substitution de deux noms de sélecteurs, de deux noms d'arguments et du nom de la collection. A défaut d'une substitution, des valeurs préétablies seront employées. Respectivement : 'add' et 'at' ; 'anElement' et 'aSymbol' ; et 'dictionary'.

```
| frag newMeth|
frag := mod fragModel fragmentFor: 'addElementInDictionary'.
newMeth := frag asSmalltalkMethod:  #('ajouter' 'dans')
      arguments: #('unElement' 'unCode')
      parameters: #('dico).
```

Le code présenté ci-dessus est un exemple de l'utilisation du fragment présenté dans la génération de code Smalltalk. Le rôle de ce fragment est de créer une méthode permettant d'ajouter un élément à un dictionnaire, appelé « dico », dans une position spécifique, déterminée par la variable « unCode ».

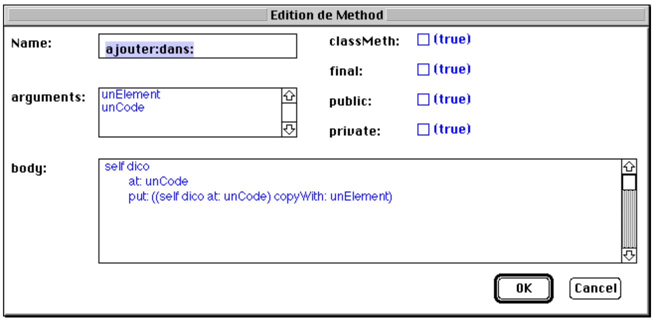

Figure 56 - Méthode d'ajout d'un élément à un dictionnaire

La Figure 56 présente la méthode Smalltalk générée par cet exemple. Il est important de remarquer que le dialogue présenté n'est pas le même que celui présenté précédemment (Figure 40). Le dialogue présenté ici est spécifique aux méthodes Smalltalk, et plus précisément au langage de modélisation MAC-S, qui sera présenté dans la section suivante.

1.2 MAC-S

La Figure 57 présente le méta-modèle du langage de modélisation MAC-S, qui est une variation de MAC spécifique à Smalltalk. Ce méta-modèle est couplé avec une base de règles qui génère, à partir du modèle d'une application, le code Smalltalk correspondant. Ce méta-modèle est similaire à celui présenté antérieurement (Figure 41), malgré quelques différences.

La première de ces différences (et la seule visible dans la figure) est l'absence du méta-individu 'Interface', concept inexistant dans Smalltalk. Le méta-modèle ne dispose que de 3 méta-individus : 'Class', 'Method' et 'Variable'.

Le premier méta-individu, 'Class', possède deux rubriques : 'name' et 'category', qui correspondent respectivement au nom de la classe et à sa catégorie. Cette dernière rubrique est utilisée pour créer des 'groupes' de classes.

Le méta-individu 'Method' possède 5 rubriques : 'name', 'arguments', 'body', 'classMethod' et 'protocol'. Les deux premières sont utilisées pour composer l'entête de la méthode, dont le corps est stocké par la troisième rubrique. La quatrième est une rubrique de type booléen, qui indique s'il s'agit d'une méthode de classe ou d'instance. La cinquième et dernière rubrique est utilisée pour grouper différentes méthodes en un 'protocole' commun.

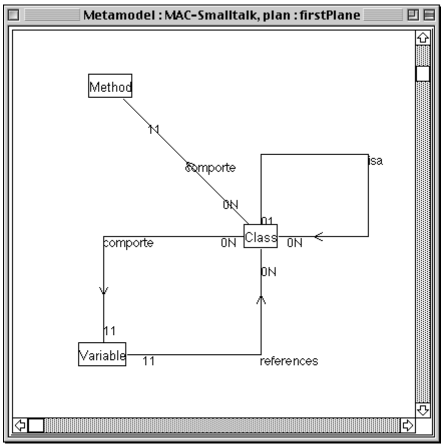

Figure 57 - Le méta-modèle de MAC-S

Le dernier méta-individu, 'Variable', dispose de trois rubriques : 'name', 'declaration' et 'public'. La première rubrique correspond au nom de l'attribut. La deuxième est de liste permettant d'indiquer s'il s'agit d'une variable de classe, d'instance ou un « pool dictionary ». La dernière rubrique est de type booléen et est utilisée pour créer automatiquement les méthodes d'accès à cet attribut.

Les cardinalités des méta-relations existantes entre les trois méta-individus indiquent qu'une classe peut posséder zéro ou plusieurs méthodes et variables et peut être la spécialisation d'une autre classe.

La structure sous-jacente de MAC-S, qui a été générée automatiquement par MétaGen, a été enrichie par plusieurs méthodes qui permettent : (i) la simplification de la création d'un modèle à partir d'un modèle MAC et (ii) la génération du code Smalltalk.

1.3 MAC-J

De son coté, le méta-modèle de MAC-J (Figure 58), est composé des mêmes quatre méta-individus que celui de MAC (Figure 41) : 'Method', 'Attribute', 'Class' et 'Interface'. Bien que ces deux méta-modèles soient similaires en apparence, les rubriques des méta-indivudus présentent plusieurs différences.

Le méta-individu 'Method', par exemple, possède 14 rubriques. Ces rubriques permettent de décrire, en plus du nom et du code d'une méthode, plusieurs propriétés spécifiques aux méthodes Java ('abstract', 'final', 'syncronized', 'private', etc.).

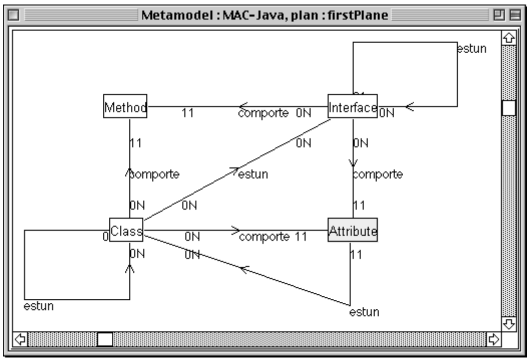

Figure 58 - Le méta-modèle MAC-J

Les rubriques du deuxième méta-individu, 'Class', permettent la description du nom de la classe, du paquetage auquel elle appartient et s'il s'agit d'une classe abstraite, statique, finale, etc.

Le méta-individu 'Attribute' possède plusieurs rubriques permettant de décrire le nom d'un attribut, son type, s'il s'agit d'un attribut statique, publique, privé, protégé, etc.

Finalement, le méta-individu 'Interface' possède trois rubriques : 'name', 'package' et 'public', correspondant respectivement au nom de l'interface, au paquetage auquel elle appartient et s'il s'agit d'une interface publique.

La structure sous-jacente de MAC-J, conformément à celle de MAC-S, dispose de plusieurs méthodes qui simplifient la création d'un modèle et la génération de code Java. Cependant, Contrairement au code Smalltalk, qui peut être chargé rapidement à partir de l'environnement, le code Java se présente sous la forme d'un fichier texte. Le chargement, dans PatternGen, d'un programme Java est plus complexe. Pour lire un fichier et retrouver les classes, leurs méthodes et attributs, nous avons fait appel au cadre d'applications « Parser » disponible dans VisualWorks.

2. Exemples de règles de production

La présentation des exemples de règles de production est divisée en deux parties. Dans la première, des règles permettant la traduction d'un modèle MAC en un modèle MAC-S seront exposées. Dans la deuxième, nous énumérons des règles de génération de la mise en œuvre du pattern Observateur.

2.1 De MAC à MAC-S

La traduction d'un modèle MAC en un modèle MAC-S (nous parlerons par la suite de modèles cible et source) est effectuée en trois étapes : (i) la création d'un modèle cible ; (ii) la copie d'individus (classes, méthodes et attributs) ; et (iii) la copie de relations (spécialisation, etc.).

Pendant l'exécution de ces trois étapes, de nouveaux objets sont ajoutés à la base de faits. Ces objets, des instances de la classe 'Association', permettront à d'autres règles de retrouver facilement des éléments générés (classes, méthodes, etc.) à partir des éléments d'origine.

Les règles sont indépendantes du modèle destination, car elles font appel à des méthodes implémentées aussi bien par la structure sous-jacente de MAC-S que par celle de MAC-J.

i) Création d'un modèle cible

La première étape consiste à créer un nouveau modèle, qui servira de cible. Celui-ci est créé par la méthode 'newModel', en fonction du langage cible choisi au préalable. Ensuite, un lien entre les deux modèles est créé et est introduit dans la base de faits (méthode 'go') :

```
createModel
| MACModel mod |
      mod notNil.


actions
|newModel |
      newModel := mod newModel.
      (Association key: mod value: newModel) go.
```

ii) Copie des classes, méthodes et attributs

La deuxième étape consiste en la copie d'individus (classes, méthodes, attributs et interface). Cette copie est réalisée par différentes règles, dont 'copyClass', présentée ci-dessous. Un objet « lien » est utilisé pour retrouver le modèle cible.

La nouvelle classe est créée à travers la méthode 'newClass', envoyée au modèle cible. Les propriétés de la nouvelle classe sont affectées par la méthode 'copyFrom'. Ensuite, la nouvelle classe est ajoutée au modèle cible (méthode 'addIndividual') et un lien entre les deux classes est créé. Finalement, la nouvelle classe et le lien sont ajoutées à la base de faits.

```
copyClass
| MACIndClass cl. MACModel mod Association link |
      cl notNil.        mod notNil. link notNil.
      link key = mod.


actions
|newCl rnd|
      newCl := link value newClass.
      newCl copyFrom: cl.
      link value addIndividual: newCl.
      newCl go.
      (Association key: cl value: newCl) go.
```

iii) Copie de relations

La copie des relations du modèle source vers le modèle cible est, elle aussi, réalisée par différentes règles. La règles présentée ci-dessous sélectionne deux individus (une classe et une méthode), le modèle source et trois liens. Les derniers permettront de retrouver, le modèle cible et dans celui-ci, la classe et la méthode qui seront liées par une relation 'Behavior' (comportement).

Cette relation est créée par la méthode 'newRelBehavior' du modèle cible. La mise en relation entre la classe et la méthode est faite par la méthode 'addRelation:to:'.

```
copyRelBehavior
|MACIndClass src. MACIndMethod tgt. MACModel mod. Association link1
      link2 link3|
src notNil. tgt notNil. mod notNil.
link1 notNil. link2 notNil.
src targetsAtRelBehavior includes: tgt.
link1 key = src.
link2 key = tgt.
link3 key = mod.

actions
link1 value addRelation: (link3 value newRelBehavior) to: link2
      value
```

2.2 La détection et l'implémentation des instances de patterns

Après la copie des individus et relations du modèle source vers le modèle cible, les règles peuvent s'intéresser aux instances de patterns. L'implémentation d'un pattern requiert la prise en compte de plusieurs compromis. Comme nous avons vu précédemment (voir § 2), une partie de ces compromis est choisie directement lors de la création d'une instance de pattern. Une autre partie doit être déduite par les règles d'implémentation du pattern, en fonction plusieurs paramètres, comme les relations entre l'instance du pattern et les participants ou les relations que les classes participantes entretiennent avec d'autres classes.

L'implémentation d'un pattern est alors divisée en deux étapes majeures : la déduction des compromis d'implémentation implicites et la génération de code.

La génération de code est réalisée par différentes règles. Ceci car un seul compromis d'implémentation peut influencer l'implémentation de différentes méthodes, et l'implémentation d'une méthode, de son coté, peut être pilotée par plusieurs compromis d'implémentation (voir page 24).

Pour générer le code d'une méthode, une règle doit connaître certaines informations sur l'instance du pattern, comme le nom des classes participantes ou des autres méthodes qui seront aussi générées. Pour éviter que chaque règle soit obligée de retrouver ces informations, celles-ci doivent être partagées. La meilleure manière de partager des informations entre règles de production est d'introduire de nouveaux faits dans la base de faits.

Pour ce faire, un ensemble de classes a été créé (Figure 59), en fonction des patterns introduits dans le méta-modèle. Ces classes ont une superclasse commune, PatternOption, qui possède 6 variables ('instance', 'participants', 'selectors', 'methods', 'attributes', 'arguments') destinées à stocker les informations concernant une instance de pattern (respectivement : l'individu représentant l'instance de pattern, les sélecteurs de messages, les méthodes à générer, le nom des attributs et le nom des arguments).

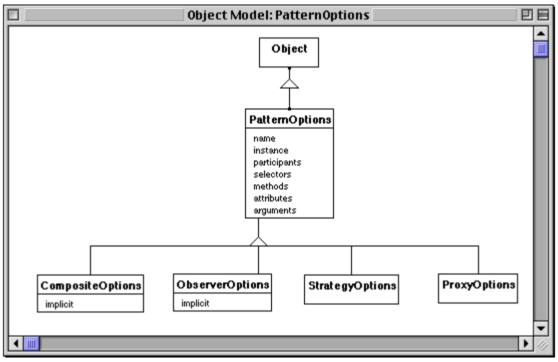

Figure 59 - La hiérarchie de classes PatternOptions

Nous présentons par la suite quelques règles de production utilisées pour mettre en œuvre le pattern Observateur. Il ne s'agit pas ici d'énumérer toutes les règles associées à ce pattern, mais d'introduire le fonctionnement général de ces règles à travers quelques règles choisies.

i) Initialisation du pattern observateur

Le rôle de cette première règle, 'observerInitialize', est de créer une instance de la classe ObserverOptions, d'affecter quelques unes de ses variables, et de l'introduire dans la base de faits.

Cette règles sélectionne deux instances de la classe 'MACIndClass' ('aSubject' et 'anObserver'), une instance de la classe 'MACIndObserver' ('pat') et deux instances de la classe 'Association'. En d'autres termes, deux individus de type 'Class', un individu de type 'Observer' et deux liens qui permettront de retrouver deux classes dans le modèle cible.

La partie 'prédicats' de cette règle vérifie que les deux classes sélectionnées sont effectivement les classes participantes de l'instance du pattern. Comme les méthodes 'tagetsAtRelXXX' retournent une collection de tous les individus liés par la relation 'XXX', il suffit de vérifier si ces collections incluent les individus en question. Les prédicats vérifient aussi que deux liens sélectionnés sont bien ceux qui lient les deux classes participantes à leurs homologues du modèle cible.

```
observerInitialize
| MACIndClass aSubject anObserver. MACIndObserver pat. Association
      link1 link2|
pat notNil. link1 notNil.  link2 notNil.
pat targetsAtRelSubject includes: aSubject.
pat targetsAtRelObserver includes: anObserver.
link1 key == aSubject.
link2 key == anObserver.

actions
| options |
“Creation d’une instance de ObserverOptions”
options := ObserverOptions subject: (link1 value) observer: (link2
      value) instance: pat.
“Affectation des selecteurs”
(options selectors) at: #observersList put: (Array with: (anObserver
      name value asLowercase, ‘s’)).
(options selectors) at: #subject put: (Array with: (aSubject name
      value asLowercase)).
(options selectors) at: #update put: (Array with: ‘update’).
“Affectation des arguments”
(options arguments) at: #observer put: (Array with: (‘a’, (link2
      value name value asLowercase firstCharToUpperCase))).
(options arguments) at: #subject put: (Array with: (‘a’, (link1
      value name value asLowercase firstCharToUpperCase))).
(options arguments) at: #aspect put: (Array with: (‘anAspect’)).
(options arguments) at: #parameter put: (Array with:
      (‘aParameter’)).
“Inclusion de l’instance dans la base de faits”
options implicit: pat isImplicit.
options go.
```

La deuxième partie de cette règle effectue plusieurs actions. Premièrement, une instance de la classe ‘ObserverOptions’ est créée et les variables ‘instances’ et ‘participants’ sont affectées (méthode ‘subject:observer:instance:’).

Ensuite, les appellations de quelques sélecteurs de méthodes et de quelques arguments sont choisies, en fonction du nom des classes participantes. Les appellations de sélecteurs et d’arguments présentes ici sont indépendantes des compromis d’implémentation.

Finalement, l’instance de ‘ObserverOptions’ est ajoutée à la base de faits.

L’appellation de sélecteurs qui dépendent de certains compromis d’implémentation est choisie par d’autres règles. Par exemple, la règle ‘observerInitializeExplicit’ présentée ci-dessous, choisie les sélecteurs des méthodes d’attachement et de détachement d’observateurs, dans le cas où la spécification explicite de l’intérêt est implémentée :

```
observerInitializeExplicit
|ObserverOptions options|
      options notNil.
      options implicit not.


actions
(options selectors) at: #attach put:
      (Array with: (‘attach’, (options observer name value
      asLowercase firstCharToUpperCase)) with: ‘aspect’).
(options selectors) at: #detach put:
      (Array with: (‘detach’, (options observer name value
      asLowercase firstCharToUpperCase)) with: ‘aspect’).
```

La règle ‘observerInitializeImplicit’ choisie l’appellation de ces mêmes sélecteurs dans le cas de l’implémentation d’une spécification implicite de l’intérêt :

```
observerInitializeImplicit
|ObserverOptions options|
      options notNil.
      options implicit.


actions
(options selectors) at: #attach put: (Array with: (‘attach’,
      (options observer name value asLowercase
      firstCharToUpperCase))).
(options selectors) at: #detach put: (Array with: (‘detach’,
      (options observer name value asLowercase
      firstCharToUpperCase))).
```

ii) Génération des méthodes

Une fois que les appellations des sélecteurs, des paramètres et des arguments ont été choisis, des règles de génération de méthodes sont activées. Par exemple, la règle ‘notifyExplicit’ présentée ci-dessous génère le code de la méthode de notification, pour une mise en œuvre ‘classique’ et une spécification explicite de l’intérêt.

La partie actions de cette règle génère le code de la méthode à partir du fragment ‘AdInArray’ et l’ajoute au modèle cible :

```
attachImplicitClassic
| ObserverOptions options. MACModel mod. Association link|
      mod notNil. options notNil. Link notNil.
      options instance codage selection = ‘classique’.
      options implicit.
      link key = mod.


actions
| arg sel par |
“Composition de la methode”
      par := options selectors at: #observersList.
      sel := options selectors at: #attach.
      arg := options arguments at:#observer.
      frag := mod fragModel fragmentFor: ‘addInArray’.
      newMeth := frag asSmalltalkMethod: sel arguments: arg
      parameters: par.
“Ajout de la methode au modele cible”
      link value addIndividual: newMeth.
      options subject addRelation: #MACSRelBehavior to: newMeth.
```

V. Conclusion

Lors de la présentation des patterns de conception (page 14), nous avons évoqué plusieurs difficultés relatives à leur implémentation (page 22). Dans les paragraphes suivants, nous revenons à ces difficultés et discutons les avantages de l'utilisation d'un outil de mise en œuvre, et plus précisément de PatternGen, dans l'atténuation de ces difficultés :

Le code spécifique à une instance de pattern est difficilement réutilisable. Le code généré par PatternGen n'est pas plus réutilisable que celui fait « à la main » car un de ses objectifs est justement de réaliser une mise en œuvre similaire à celle réalisée par un implémenteur. En effet, la réutilisation préconisée par PatternGen est réalisée en autre niveau, celui des connaissances d'implémentation. Celles-ci sont réutilisables, grâce à la base de règles d'implémentation.

Les patterns se perdent dans le code. D'une part, l'utilisation d'un outil permet la génération automatique d'un code clair et commenté, ce qui peut simplifier la détection d'une instance de pattern dans le code. D'autre part, la représentation d'instances de pattern directement dans l'éditeur de diagrammes permet l'identification immédiate des classes participantes à une instance de pattern.

L'évolution de l'application doit respecter les contraintes des patterns. PatternGen, à l'instar des outils de mise en œuvre de patterns adoptant une approche générative, ne peut pas assurer que les modifications opérées directement sur le code d'une application respectent les contraintes des instances de patterns déjà existantes. Cependant, il peut vérifier que les modifications opérées par l'intermède de l'éditeur de diagrammes respectent les contraintes existantes.

L'application des patterns surcharge l'implémentation. La génération automatique de code décharge l'implémenteur de la mise en œuvre de méthodes triviales. En même temps, comme les méthodes générées ne sont pas ajoutées à l'éditeur de diagrammes, la lecture du modèle de conception d'une application n'est pas surchargée par de telles méthodes.

Les compromis d'implémentation sont difficilement maîtrisables. La maîtrise des compromis d'implémentation est simplifiée de deux manières dans PatternGen. Premièrement, par l'utilisation, d'une base de règles capable de choisir la variante d'implémentation la plus adaptée aux compromis choisis. Deuxièmement, par l'utilisation de dialogues d'édition explicitant les compromis existants.

L'implémentation d'un pattern n'est pas isolée. Une fois encore, la base de règles d'implémentation permet d'atténuer une difficulté d'implémentation. Ici, elle peut trouver, en fonction de l'existence de différentes instances de patterns et des compromis propres à ces instances, l'implémentation la plus adéquate.

L'analyse de différents outils de mise en œuvre de patterns (page 31) nous a permit d'énumérer 5 caractéristiques souhaitables dans ce genre d'outil : la gestion de patterns, la représentation d'instances de patterns, le support aux variantes d'implémentation, la documentation de l'implémentation d'instances de patterns et la reconnaissance de patterns.

Parmi ces cinq caractéristiques, la seule qui n'est pas présente actuellement dans PatternGen est la dernière, la reconnaissance d'instances de patterns à partir du code d'une application. Bien que notre outil dispose des instruments nécessaires pour charger des applications écrites en Java et en

Smalltalk, il ne nous est pas encore sûr que l'utilisation de bases de règles sera adaptée à la reconnaissance de patterns.

La mise en œuvres des idées énoncées par Kyle Brown [Brown, 1995], lors du développement de KT peuvent constituer un important point de départ. Cet outil est capable de reconnaître des instances des patterns suivants : **Composite**, **Décorateur** et **Fabrique Abstraite**.

Chapitre 5

PatternGen et le projet MétaFor

I. Introduction

Dans ce chapitre nous présentons la démarche de développement de l'équipe MétaFor. Cette démarche est issue d'un projet de recherche mené dans notre laboratoire depuis le début des années 90 [Lesueur *et al.*, 1998 ; Lesueur *et al.*, 1999 ; Perrot, 1998 ; Revault *et al.*, 1995] sur l'acquisition, la représentation et l'utilisation des connaissances liées au processus de construction d'applications informatiques. Elle a fait l'objet d'une première présentation au colloque RPO'96 [Blain et Beugniet, 1996].

Les travaux de l'équipe MétaFor ont comme objectif de générer des applications, le plus mécaniquement possible, à partir de l'expression des besoins des utilisateurs. Pour aider ces derniers à mieux exprimer leurs besoins, des langages de modélisation fortement liés au domaine d'application sont spécifiés, représentant une application usuelle de l'analyse du domaine. Pour outiller les divers langages de domaines, l'équipe MétaFor a fait appel à des techniques de méta-modélisation, en spécifiant et en développant MétaGen (Chapitre 3), un méta-outil capable de générer des environnements de modélisation à partir d'un méta-modèle représentant un langage d'un domaine spécifique.

Cette démarche se veut garante du respect scrupuleux des responsabilités des différents acteurs du processus de développement d'un système informatique. Pour réaliser cela, elle doit assurer une prise en compte équilibrée des points de vue des utilisateurs et des développeurs durant ce processus. Cet équilibre se traduit par la spécification de langages de modélisation permettant d'exprimer le point de vue des développeurs et par l'outillage de ces langages. Comme nous verrons plus tard (§ II.2), ces langages sont en réalité des moyens de description de la politique d'usage des cadres d'applications.

Cette dissociation claire entre le point de vue des utilisateurs et celui des développeurs permet la structuration des connaissances présentes dans le processus de développement (d'une part celles qui relèvent du domaine de l'application et d'autre part celles qui relèvent des principes du développement).

Au début de notre doctorat, nous nous sommes intéressés à la spécification et à l'outillage de langages de description de la politique d'usage de cadres d'applications. Plus précisément, nous avons spécifié et outillé des langages permettant la description de l'usage des cadres d'applications Accounts (§ II.2.3) et HotDraw (§ III.2.2). Ces expériences de mise en œuvre nous ont permis d'évaluer notre démarche de développement, et de proposer une évolution à cette démarche, par l'intégration de PatternGen.

Ce chapitre est divisé en 4 parties. La suite de la présente introduction décrit les motivations de notre démarche. La deuxième partie présente notre démarche de développement d'applications par changement de point de vue et de niveau de connaissances. La troisième partie illustre cette démarche par un exemple de mise en œuvre réalisé par l'auteur. La quatrième partie discute les avantages et problèmes affrontés durant la mise en œuvre de cet exemple et propose une manière de les résoudre par l'intégration de PatternGen à la démarche de développement.

1. Motivation

« Les informaticiens produisent sensiblement moins de systèmes informatiques que la société n'en a besoin ; de plus, une grande partie des systèmes développés ne fonctionne pas correctement. »

La constatation de cette « crise » du développement de logiciels [Brooks, 1982], datant de la fin des années soixante-dix, confirme les prévisions selon lesquelles le coût des logiciels serait prépondérant dans le développement d'un système informatique [Boehm, 1973]. Pour dénouer cette crise, plusieurs solutions ont été proposées, parmi lesquelles l'usage plus systématique de la modélisation est indubitablement la plus appliquée, même si elle ne couvre qu'un certain aspect d'un système, comme les données [Chen, 1976] ou les processus [DeMarco, 1982 ; Gane et Sarson, 1979]. Ces techniques ont été fortement développées au cours des années et, combinées avec des procédures d'application bien définies, ont donné naissance aux premières méthodes de développement.

Plus tard, ces méthodes ont été outillées (et soutenues) par les premiers ateliers de génie logiciel, qui promettaient alors de révolutionner le développement de systèmes [Yourdon, 1986]. Les ateliers, aussi bien que les autres outils de génie logiciel assisté par ordinateur, ont engendré des avancées importantes dans l'amélioration du processus de développement, mais la révolution attendue n'a pas eu lieu [Brooks, 1987]. En effet, le succès d'un atelier dépend intrinsèquement du succès de la méthode de développement supportée [Aaen *et al.*, 1992] et celui-ci dépend d'une relation complexe et variable entre les facteurs techniques et institutionnels. En plus, la fixité de la méthode de développement supportée par un atelier induit une des principales difficultés de leur utilisation. Par conséquent, le choix d'un outil impose le choix d'une méthode [Parkinson, 1990] et réciproquement.

Pour faire face à ce dernier problème, deux solutions sont possibles. La première consiste à développer des outils spécifiques à chaque organisation. Cette solution est coûteuse [Karrer et Scacchi, 1995] et n'offre pas un cadre acceptable pour l'évolution et la maintenance des environnements de développement, qui sont deux caractéristiques importantes pour ce type de logiciels [Boehm, 1990]. La seconde solution consiste à utiliser des « méta-outils » (CASE Shell [Bubenko, 1988] ou outils MetaCASE [Alderson, 1991 ; Hofstede *et al.*, 1996]), i.e. des outils capables de générer des environnements de développement.

Par ailleurs, l'utilisation d'une méthode améliore la maîtrise du processus de développement, en permettant, par exemple, de valider effectivement certaines étapes ou d'améliorer la communication entre les différents participants, grâce à une plus grande rigueur. Cependant, les méthodes restent limitées en ce qui concerne la compréhension d'un grand volume de connaissances spécifiques à un domaine. D'autre part, elles ne remplacent pas le manque de bons modèles d'un domaine [Barstow, 1985].

En effet, la communication entre les informaticiens et les utilisateurs reste l'un des problèmes essentiels du processus de développement [Boehm, 1981 ; Loucopoulos et Zacari, 1992]. Ces deux communautés jouent des rôles distincts durant le développement, et ont, évidemment, des intérêts différents. De plus, les métiers des utilisateurs n'étant pas toujours les mêmes, les informaticiens sont contraints de connaître le vocabulaire spécifique à chaque métier. Cette difficulté est critique, car il est reconnu que l'amélioration de la communication et de la compréhension entre utilisateurs et concepteurs réduit le travail superflu dans les étapes tardives du processus de développement [Curtis *et al.*, 1988].

Une manière de contourner ce problème est que l'une des communautés maîtrise le langage de l'autre. Par ailleurs, plusieurs références soutiennent que l'existence d'un concepteur exceptionnel [Brooks, 1987 ; Christiansen, 1987], est déterminante dans le succès du développement d'un système. Un concepteur exceptionnel est un informaticien qui, grâce à sa connaissance du domaine d'application et sa capacité d'abstraction et de communication, devient la pièce centrale du développement de systèmes.

L' « analyse du domaine » [Neighbors, 1984 ; Pietro-Diaz, 1990 ; Pietro-Diaz, 1993], est une des premières solutions alternatives à ce problème. Plutôt que d'analyser les actions spécifiques à un problème, l'analyse du domaine s'intéresse aux objets et actions spécifiques d'un domaine d'applications. Elle combine plusieurs disciplines informatiques, comme l'acquisition et la représentation de connaissances, la génération automatique de code et les méthodes de développement.

La démarche de développement de l'équipe MétaFor sera présentée par la suite. Cette présentation commence par un aperçu global de la démarche. L'expression du point de vue des utilisateurs et de celui des développeurs sera présentée ensuite. Finalement, les transformations entre les points de vue seront présentées.

2. Projets analogues de recherche

Le projet MétaFor rassemble l'application de plusieurs techniques, de plusieurs thèmes de recherche. Par exemple, le processus d'extraction de concepts du domaine de l'utilisateur et de l'environnement applicatif pour permettre l'expression des points de vue de l'utilisateur et du développeur est une application de l'analyse du domaine. La construction d'un outillage permettant l'expression de ces deux points de vue, celle du génie de méthodes et de la méta-modélisation. Finalement, la transformation de modèles est une application des techniques de transformation de schémas.

Il est hors des objectifs de ce mémoire de décrire de manière extensive tous les projets de recherche comparables à MétaFor en plusieurs aspects. Plusieurs travaux dans ce sens sont disponibles, comme les travaux de Pietro-Diaz [Pietro-Diaz, 1990 ; Pietro-Diaz, 1993] sur les perspectives dans les recherches de l'analyse du domaine, la comparaison de huit méthodes d'analyse de domaine de Arango [Arango, 1993], le rapport de Kogut et Nilson [Kogut et Nilson, 1994] (vol. 2) sur les outils d'analyse du domaine. Il nous paraît important de citer aussi les efforts de Brinkkemper, Lyytinen et Welke [Brinkkemper *et al.*, 1996] de réunir plusieurs travaux dans le génie de méthode, et ceux de Karrer et Scacchi [Karrer et Scacchi, 1993] dans le domaine des outils de méta-modélisation.

Nous présentons dans l'Annexe D de ce document, quelques projets dont les objectifs ou les techniques utilisées ressemblent à ceux utilisés dans le projet MétaFor et qui ont inspiré ou inspirent son évolution.

II. Développement d'applications par changement de point de vue et niveau de connaissances

Dans le cadre de l'étude de l'utilisation des connaissances dans l'ingénierie des applications informatiques, l'équipe MétaFor élabore des outils permettant (i) aux utilisateurs de définir les langages de modélisation à travers lesquels ils peuvent énoncer tout ou partie de leur système d'information et l'informatisation de tout ou partie de son fonctionnement, et (ii) aux informaticiens de générer économiquement d'une part l'outillage permettant la maîtrise des langages de modélisation décrits et, d'autre part, la traduction de ceux-ci en des représentations interprétables par des systèmes technologiques du marché.

La première difficulté de cette approche est que les utilisateurs ne travaillent pas tous dans le même domaine. Les concepts utilisés sont différents, ce qui implique un langage distinct pour chaque domaine. Plus précisément, il est quasiment impossible économiquement de générer un outillage de support pour chacun des langages concernés.

Pour répondre à ce problème, il a été défini un langage de méta-modélisation, PIR3, et un environnement qui le supporte, MétaGen [Lesueur *et al.*, 1998 ; Revault *et al.*, 1995]. Par langage de méta-modélisation, nous entendons un langage de modélisation permettant la description d'autres langages de modélisation. PIR3 est donc un langage de modélisation dont l'objectif est de modéliser la syntaxe de différents langages spécifiques aux différents domaines (pour la représentation de la sémantique, voir § 3). L'environnement MétaGen fournit un éditeur graphique basé sur PIR3 et qui génère des environnements de modélisation spécifiques à chaque langage de modélisation représenté dans cet éditeur. La génération d'un environnement de modélisation spécifique à chaque domaine passe alors par une étape d'analyse du domaine, qui consiste à reconnaître les concepts essentiels du domaine. Ces concepts essentiels donneront origine ensuite, lors d'une étape de méta-modélisation, à des constructeurs de modélisation du langage utilisateur. Ces constructeurs serviront à décrire le système informatique souhaité.

La deuxième difficulté concerne la génération automatique de code à partir des modèles issus de ce langage, les modèles « utilisateur ». En effet, ces modèles étant orientés domaine d'application, ils sont d'un très haut niveau d'abstraction et la génération de code doit remplacer le rôle de plusieurs participants du processus de développement.

Pour réaliser la génération de code, MétaFor propose l'utilisation de bases de règles de production, représentant les connaissances présentes dans le processus de développement. Les règles sont écrites dans MétaGen, qui est intégré avec le moteur d'inférence NéOpus [Pachet et Dojat, 1992].

Pour réduire la complexité d'une telle base de règles, Revault, Sahraoui et al. [Revault *et al.*, 1995] ont proposé d'inclure une étape intermédiaire durant la génération de code, le modèle « implémenteur » (Figure 60). Ce modèle est exprimé par un langage dit implémenteur, fortement lié au domaine de mise en œuvre de l'application. Cette étape permet, en plus de simplifier le codage des règles, de diviser deux importantes connaissances présentes dans la génération : celle des concepteurs exceptionnels, qui traduisent les concepts du domaine en des concepts compréhensibles par les informaticiens, et celle des informaticiens qui conçoivent et mettent en œuvre une application.

Le modèle implémenteur permet, en effet, d'inclure une étape de validation dans le processus de développement, ainsi que la division de la complexité des règles. En même temps, il apporte une augmentation potentielle de la réutilisation car deux modèles utilisateurs différents, écrits dans des langages différents, peuvent être traduits dans deux modèles implémenteurs utilisant un même langage. Ainsi, deux applications différentes sont générées à partir de ces modèles, par une même base de règles.

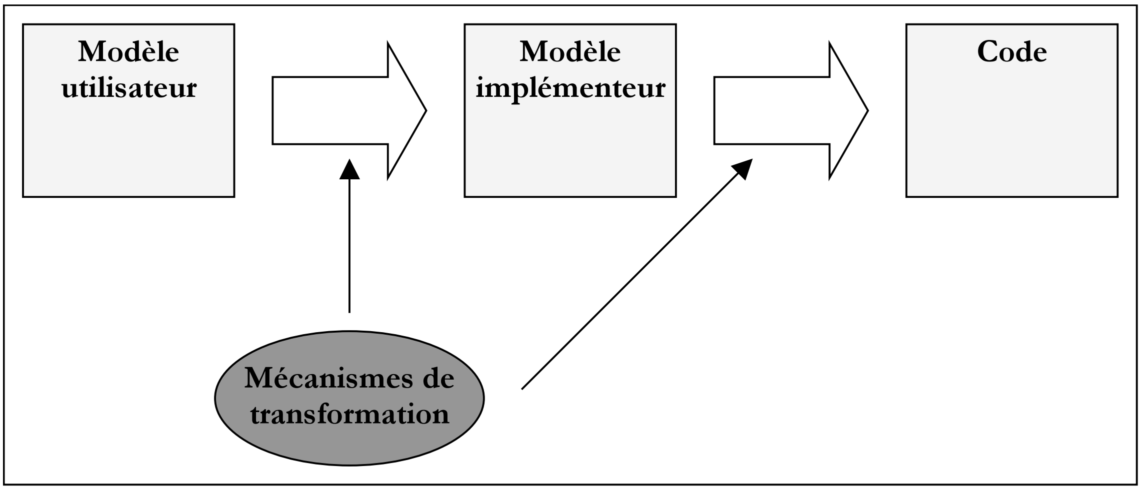

Figure 60 – Le développement en deux modèles

De façon similaire aux langages utilisateur, les langages implémenteur ont pour objectif de représenter les concepts de l'environnement informatique dans lequel l'application sera développée et, de cette manière, augmenter la réutilisation des ressources déjà existantes. Cette réutilisation n'est pas simplement utile, mais aussi fondamentale pour réduire la complexité des bases de règles de génération de code. Les modèles implémenteurs servent alors d'interface à des composants logiciels déjà existants, comme les cadres d'applications [Revault, 1996].

Actuellement, les travaux de recherche de l'équipe MétaFor suivent deux axes principaux. Le premier (suivi par Bruno Lesueur), MétaFor s'intéresse à l'adaptation des langages de modélisation à un domaine spécifique. Ceci comprend l'emploie de différentes notations (graphique, textuelle, etc.), la simplification et la synthèse de modèles [Lesueur, 1997].

Le deuxième axe de recherche (suivi par l'auteur) s'intéresse aux différents niveaux technologiques (architecture, cadres d'applications, patterns, paradigmes de langages) présents pendant la mise en œuvre d'une application. Notre objectif est de permettre une réutilisation plus ample des connaissances nécessaires à cette mise en œuvre, non restreinte à celle offerte par les cadres d'applications. Ces connaissances comprennent la maîtrise des facilités du langage d'implémentation et d'autres concepts moins tangibles, comme des solutions récurrentes de conception et d'implémentation (les patterns de conception) [Sunyé, 1999].

En résumé, la démarche proposée par l'équipe MétaFor préconise la mise en œuvre d'un environnement de développement en six étapes :

1. Extraire les concepts du domaine employés par l'utilisateur pour décrire son système, indépendants des choix technologies et informatiques ;

2. A partir de (1), écrire un méta-modèle qui fournit un environnement spécifique pour décrire les procédés de traitement liés aux besoins des utilisateurs (le méta-modèle utilisateur) ;

3. Reconnaître les informations nécessaires pour spécialiser un cadre d'applications existant pour générer une application.

4. A partir de (3), écrire un méta-modèle d'un langage qui permettra de décrire différentes adaptations du cadre d'applications (le méta-modèle implémenteur).

5. A partir de (2) et (4), définir les règles qui peuvent transformer un modèle utilisateur quelconque en un modèle implémenteur. Ceci est l'étape la plus difficile car elle contient l'expertise des concepteurs exceptionnels : la traduction des concepts du domaine en des concepts de l'environnement applicatif. La faisabilité de ces règles repose sur la distance sémantique entre les deux méta-modèles. Pour cette raison, il est possible, dans cette étape, de retourner aux étapes (2) et (4) pour réviser les méta-modèles, en ajoutant par exemple plus de détails.

6. A partir de (4), écrire les règles de transformation générant le code source approprié pour utiliser l'environnement applicatif existant, à partir de modèles implémenteurs.

1. Le point de vue des utilisateurs

L'utilisateur est intéressé à la finalité des opérations et à l'essence des entités mises en cause.

Pour permettre aux utilisateurs d'exprimer leur point de vue, le processus de développement prévu par le projet MétaFor passe par une phase de définition d'un langage de modélisation orienté utilisateur. L'objectif de l'utilisation d'un tel langage est de diminuer la distance entre la sémantique du domaine du problème et celle des modèles conceptuels décrivant une application.

C'est grâce à ce formalisme - nous employons formalisme ici dans le sens de [Krogstie et Solvberg, 1996] : un formalisme est un langage disposant d'un vocabulaire, d'une syntaxe et d'une sémantique précisément définis - que l'utilisateur pourra décrire (ou valider) l'application souhaitée. Pour permettre une description extensive et non ambiguë du point de vue de l'utilisateur, le formalisme doit fournir un vocabulaire riche et une syntaxe restreinte.

D'autre part, il est important que ce formalisme utilise un système de notation proche de celui utilisé par l'utilisateur. Nous entendons par ceci qu'un langage décrit par PIR3 peut assumer des formes de notation variées, comme une notation textuelle, graphique (voir aussi les travaux de Sommerville, Beer et Welland [Beer *et al.*, 1987], de Paulich [Paulich et Tichy, 1991] et de Bertolazzi [Bertolazzi *et al.*, 1995] sur les éditeurs génériques) ou matricielle [Kelly, 1994].

Un exemple récent de l'utilisation d'un formalisme proche à l'utilisateur est décrit dans [Guerrieri, 1994]. L'auteur a créé un langage de quatrième génération (L4G) spécialisé commerce, qui permet aux utilisateurs de concevoir des programmes qui sont traduits par la suite en code COBOL/SQL.

Nous présentons, ci-dessous, deux exemples de langages de modélisation orientés utilisateur, utilisant une notation graphique (réseaux sémantiques).

1.1 Langage de modélisation administratif

Le premier langage utilisateur que nous présentons ici, LMA, est employé pour modéliser des systèmes de gestion [Lesueur *et al.*, 1998]. La plupart de ces systèmes utilisent des procédés semblables pour organiser les données, en les triant dans des dossiers. Dans une compagnie d'assurances, par exemple, lorsqu'un client signe un contrat d'assurance, un dossier est ouvert à son nom. Ce dossier servira à garder le contrat et des documents complémentaires à celui-ci. La quantité de documents et leur nature dépendent intrinsèquement de la nature du contrat.

LMA est composé de trois constructeurs : 'Dossier', 'Chemise' et 'Pièce'. Il existe trois sortes de pièces : les pièces référentielles, faisant référence à une autre pièce, à une chemise ou à un dossier, les pièces informelles et les pièces à valeurs, qui contiennent plusieurs propriétés, saisies ou calculées. Les pièces sont organisées par des chemises et celles-ci, par des dossiers.

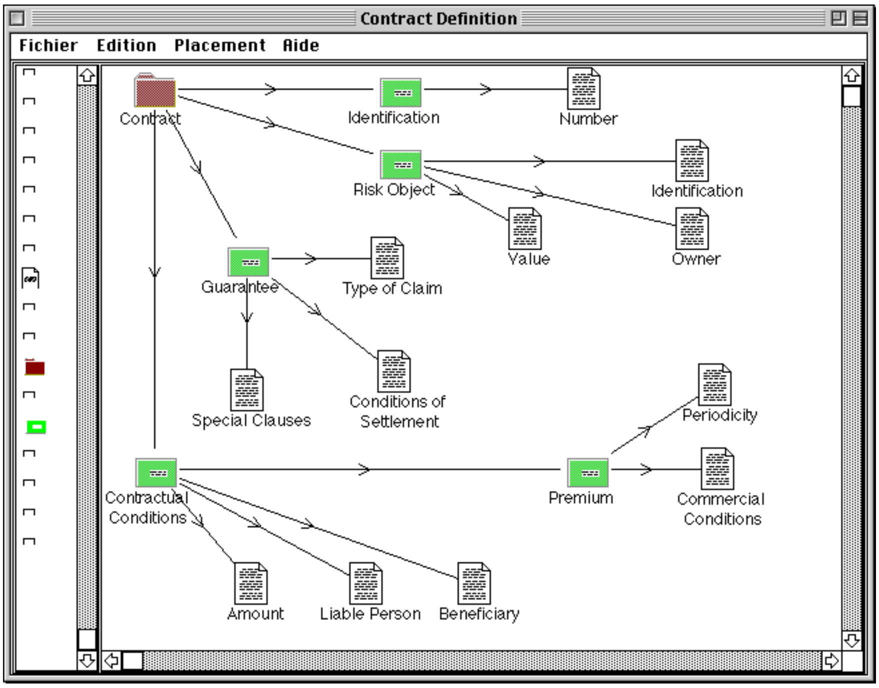

Figure 61 - Définition d'un contrat d'assurance

La Figure 61 est un exemple d'utilisation de LMA pour modéliser un contrat d'assurances (pour plus de détails, voir [Lesueur *et al.*, 1999]). Ici, un contrat est défini comme étant un dossier composé de quatre « chemises », qui représentent l'identification du contrat, l'objet de risque, la garantie et les conditions contractuelles. L'objet de risque est composé par trois pièces décrivant son propriétaire, sa valeur et son identification. La garantie est aussi composée par trois pièces, qui décrivent la nature du sinistre assuré, les conditions de règlement et les clauses particulières du contrat. Finalement, les conditions contractuelles sont composés par trois pièces qui établissent le tarif, le payeur et le bénéficiaire et par une autre chemise, la prime du contrat, qui possède à sont tour deux pièces décrivant la périodicité et la tarification.

1.2 Langage de définition de produits financiers

Ce langage est utilisé dans le projet FIBOF ESPRIT IV [Lesueur *et al.*, 1998]. L'objectif du projet FIBOF est de développer un cadre d'applications pour la communauté bancaire. Ce cadre fournit un modèle de définition de produits financiers dont la gestion peut être adaptée à une institution particulière.

Les cinq constructeurs qui composent le langage de définition de produits financiers sont : les produits eux-mêmes, les variables, les bundles, les contraintes de gestion et les équilibres. Un produit est une agrégation de variables, d'équilibres et de contraintes. Il existe trois types de variables : les variables calculées par une certaine expression, les variables de négociation qui doivent être définies dans la phase de négociation et les variables d'événement qui prennent une valeur booléenne et sont employées pour « attraper » des événements externes. Les contraintes sont exprimées sur ces variables. Un 'bundle' (un terme financier qui dans notre contexte signifie un comportement élémentaire) est exprimé en termes de contraintes de gestion et variables. Un équilibre décrit une action composite qui est exprimée en termes de bundle, de variables et de contraintes. Un équilibre représente une opération équilibrée de comptabilité (d'où son nom).

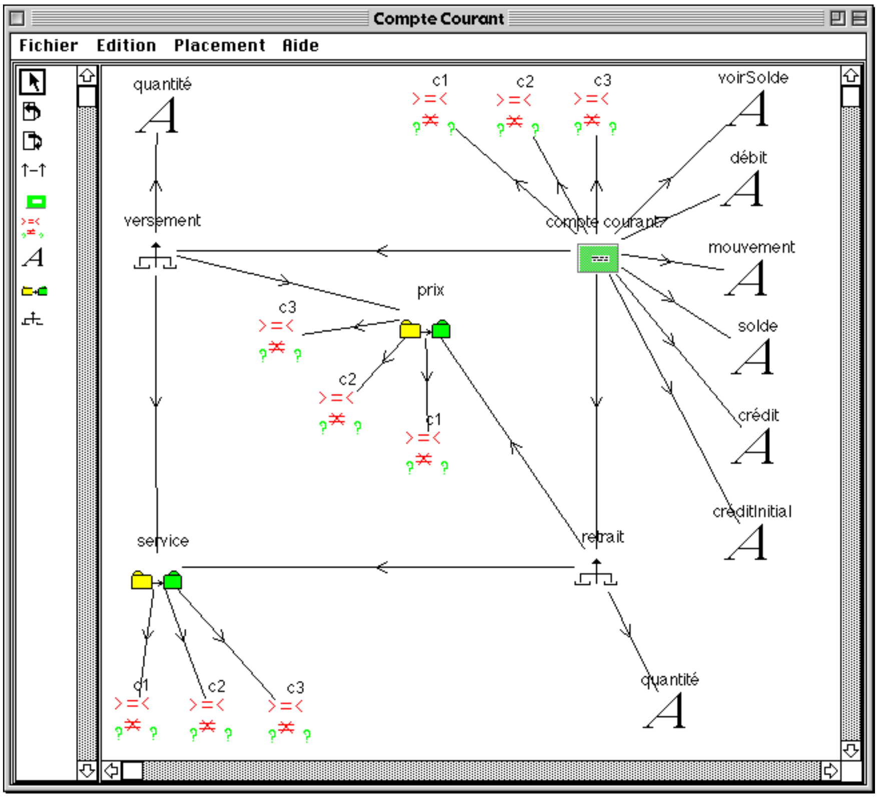

Figure 62 - Définition d'un compte courant

Un compte de vérification est un produit défini par trois équilibres : la retraite, le gisement et l'intérêt. Deux actions élémentaires, notamment prix et service, sont employées pour définir retraite et gisement. L'intérêt est défini par une action élémentaire : l'intérêt-équation. La description du produit inclut aussi deux variables de négociation (initial-balance et intérêt-taux), deux variables calculées (balance et mouvement) et trois variables d'événement (chèque-balance, débit et crédit). L'équilibre d'intérêt inclut une contrainte qui précise quand il doit être activé (e.g. chaque mois ou chaque année).

La Figure 62 est un exemple de définition d'un produit financier simple : un compte courant. Un compte courant est défini par deux équilibres : le retrait et le versement. Deux bundles, prix et service, sont employés pour définir les deux équilibres.

La description du produit inclut aussi une variable négociable (crédit initial), deux variables calculées (solde et mouvement) et trois variables d'événement (voir solde, débit et crédit). Plusieurs contraintes précisent le fonctionnement d'un compte courant, comme l'impossibilité de faire des retraits plus grands que le solde.

2. Le point de vue des développeurs

Pendant le processus de développement d'une application, la préoccupation principale des développeurs est de mettre en œuvre un ensemble de moyens informatiques de manière à automatiser les systèmes d'information décrits par les utilisateurs. Cette mise en œuvre mobilise plusieurs types de connaissances, comme celle des concepteurs exceptionnels qui traduisent les besoins d'un domaine en des termes informatiques ou celle des implémenteurs qui appliquent plusieurs techniques de programmation pour mettre en œuvre une application.

Pour mieux maîtriser ces connaissances, et par conséquent pour mettre en valeur la réutilisation, notre démarche de développement prévoit une étape de validation. Cette étape prend la forme d'un modèle conceptuel d'implémentation, représentant une réutilisation de l'environnement applicatif existant. Nous supposons, qu'une application est construite, au moins en partie, par la réutilisation de certains éléments déjà existants.

Une des difficultés de cette approche est de trouver ou de créer le langage conceptuel le plus adapté, ou plus précisément, de trouver le niveaux correct d'abstraction. D'une part, les langages de modélisation des méthodes à objets existants (comme UML ou OMT) ont un minimum de constructeurs ce qui implique la création de modèles trop granulaires [Schefström, 1993]. D'autre part, créer un langage très abstrait, riche en constructeurs - chacun décrivant par exemple comment réutiliser une des classes présentes dans l'environnement - n'assure pas la parfaite réutilisation de l'environnement existant. En effet, les classes sont trop élémentaires pour permettre de profiter entièrement des avantages de la réutilisation [Deutsch, 1989]. Ceci est dû à deux facteurs liés. En premier, la plupart des classes tendent à être de petite taille et en deuxième, les classes exécutent rarement un comportement par elles-mêmes ; elles contribuent souvent à une fonction donnée par l'interaction avec d'autres objets [Mili *et al.*, 1990].

La solution est donc de trouver, dans l'environnement existant, des concepts moins granulaires que les classes, comme des collections de classes réutilisables qui ensemble, exécutent certaines fonctions utiles [Deutsch, 1989 ; Johnson et Foote, 1988] - les cadres d'applications. Adopter une approche de programmation par cadre d'applications a plusieurs avantages. Les cadres sont conçus pour être réutilisés, ils peuvent engendrer différentes applications dans un même domaine et il existe plusieurs cadres disponibles actuellement.

Toutefois, la réutilisation des cadres d'applications pose plusieurs défis. Tout d'abord, il y a le problème de trouver le cadre pertinent à un besoin spécifique. Trouver le cadre adéquat à un besoin est seulement le premier pas d'un procédé long et compliqué. En supposant que le cadre adéquat ait été trouvé, il faut comprendre la manière par laquelle il résout le problème, et découvrir comment l'utiliser pour implémenter une solution. Ce dernier aspect requiert que les développeurs réduisent l'écart qui existe entre le besoin explicite, en termes de fonctionnalités à être réalisées ou un critère de conception à être optimisé, et la solution proposée par le cadre. Les langages de patterns s'adressent à ces besoins en essayant de fournir plusieurs *slots* qui différencient les aspects d'applicabilité et de « comment faire » [Johnson, 1992]. Cependant, un certain nombre d'auteurs sont d'accord pour dire que ces descriptions sont souvent insuffisantes et que les développeurs doivent souvent regarder dans le code source [Gabriel, 1994]. En plus, il y a peu d'assistance opérationnelle aux développeurs pour bâtir des solutions utilisant un cadre.

Une fois le langage conceptuel (décrivant la politique d'usage d'un cadre) créé, il reste à modéliser toute la connaissance, décrite précédemment, sur comment réutiliser un cadre d'applications. Pour cela, nous employons les bases de règles. Ainsi, un effet secondaire de notre démarche est de simplifier l'adaptation d'un cadre d'applications [Revault, 1995].

2.1 Les cadres d'applications

Les cadres d'applications à objets représentent la conception en des termes d'interfaces nécessaires et standards de communication entre les objets supportant ces interfaces. Parfois, ils contiennent aussi des bibliothèques de classes ou des générateurs de programmes qui simplifient la réalisation d'applications. Il existe deux sortes de cadres d'applications, ceux qui sont concernés par les applications spécifiques à un domaine et ceux concernés par l'implémentation d'un mécanisme particulier ou d'une tâche technique en relation avec l'ordinateur.

Nous nous sommes intéressés aux cadres d'applications qui sont suffisamment bien définis et dont le code nécessaire à la réalisation d'une solution est généré partiellement ou totalement. La façon dont ce code est généré (i.e. par des sous-classes ou des générateurs) est sans importance pour notre étude.

Du point de vue du développement d'un applicatif, un cadre d'applications a deux buts concurrents. Avant de décider de son utilisation, il faut être certain que la conception ou le domaine qu'il représente est suffisamment générique pour qu'il soit utile. En même temps, il doit être suffisamment précis pour être programmable et pour avoir une performance raisonnable. Ceci repose sur de nombreuses techniques de configuration qui permettent de réaliser un maximum de fonctionnalités génériques, tout en laissant le comportement – ou l'usage spécifique – à la discrétion des développeurs.

Les cadres d'application peuvent présenter plusieurs difficultés de mise en œuvre. En premier, il faut trouver l'adéquation entre les besoins du développeur et les descriptions du cadre d'applications. Cette difficulté repose sur le vocabulaire (ou la terminologie) employé pour la description du cadre d'applications. C'est un problème générique de la réutilisation [Mili et Mili, 1995] : les besoins des développeurs tendent à être exprimés dans le langage du domaine du problème (fonctionnalités), alors que les cadres d'applications tendent à être exprimés dans le langage du domaine de la solution(spécificité de l'implémentation). Une autre difficulté porte sur les niveaux d'abstractions. Elle est en relation avec le fait que les cadres d'applications sont des agrégations de composants réutilisables : une fonction définie au niveau global de l'applicatif par un développeur pour spécifier un besoin, pourra être décomposée en plusieurs classes collaboratrices au sein du cadre d'applications adapté ; les frontières fonctionnelles entre classes sont surtout motivées par des contraintes liées aux choix de conception [Mili *et al.*, 1996] au détriment de la sémantique du niveau du domaine.

Les langages à patterns fournissent des outils descriptifs qui essayent de couvrir les besoins de spécification des aspects d'opportunité ou d'applicabilité d'un cadre d'applications [Gamma *et al.*, 1995]. Cependant, ces descriptions sont fondamentalement ambiguës, ouverte à de multiples interprétations et très subjectivement liées aux expériences individuelles [Mili *et al.*, 1996]. Les notations formelles, comme les « contrats » [Helm *et al.*, 1990] ou les langages d'interconnexion de modules [Hall et Weedon, 1993] sont trop éloignées des langage liés au domaine des problèmes pour être d'usage pratique. Fréquemment, les développeurs sont forcés d'analyser le code source proposé pour comprendre le fonctionnement d'un cadre d'applications et décider s'il est adéquat à leurs besoins ou non.

En supposant qu'un cadre d'applications soit approprié à un problème donné, il reste le problème de décrire son usage. Décrire l'usage d'un cadre d'applications ou d'un quelconque composant réutilisable implique de résoudre deux problèmes. En premier, il y a la question de définir de quelle manière le cadre d'applications doit satisfaire le besoin énoncé ou la situation donnée. Ceci correspond à choisir une politique d'usage, et à instancier les structures fournies pour le problème donné. En deuxième, il y a le mécanisme d'implémentation de la solution. Sous cet aspect, en plus des informations traditionnelles qu'un développeur doit connaître, comme les types acceptables de paramètres ou les effets de bord, les cadres d'applications exigent un assemblage. Ceci peut englober trois actions : 1) sélectionner les participants, 2) préparer les participants pour qu'ils puissent interagir avec les autres composants du cadre d'applications (e. g. ajouter des méthodes de pont) et 3) créer des instances et des liaisons. La sélection est habituellement basée sur le rôle joué par le participant des cadres d'applications et la réalisation des bénéfices promis de réutilisation. La préparation est souvent nécessaire car le véritable participant qui joue le rôle correct peut ne pas avoir une interface appropriée.

Tous ces facteurs compliquent suffisamment la réutilisation d'un cadre d'applications pour que seuls des concepteurs expérimentés aient les compétences nécessaires pour mettre en œuvre ces conceptions réutilisables [Gamma *et al.*, 1995]. Certains ont même argumenté qu'en raison de la courbe d'apprentissage, seulement les concepteurs expérimentés étaient capables de tirer avantage des cadres (et des patterns) existants.

Notre étude à pour but de proposer un environnement de développement qui permettrait, par la réification des connaissances de ces développeurs, de les mettre à disposition des experts du domaine. Ainsi les cadres d'applications pourraient satisfaire réellement leurs objectifs de réduire les coûts de développement d'un ensemble d'applications.

2.2 Cadres d'applications (comment concevoir des langages permettant de décrire leur politique d'usage)

Actuellement, il nous est encore très difficile d'établir une procédure de conception d'un langage de modélisation à partir d'un cadre d'applications. Toutefois, la conception d'un tel langage pour plusieurs cadres d'applications, comme Accounts, HotDraw et DataModeler durant notre thèse nous permet de faire une supposition. Dans son essence, un cadre d'applications comprend une architecture et quelques fonctionnalités. L'architecture définie les rôles et les obligations des participants, alors que la fonctionnalité, comprise dans la bibliothèque de classes qui l'accompagnent, fournit une implémentation de ces rôles.

Notre argument est que le lien manquant entre les modèles du domaine et les schémas d'utilisation de cadres d'applications est l'architecture.

Alors, le langage de modélisation doit viser la vue architecturale de l'applications en identifiant les participants (les rôles et les interfaces) et leur configuration spécifique. De manière similaire, la transformation entre les modèles du domaine et les politiques d'usage des cadres d'applications doivent cibler l'identification des rôles et de leurs configurations, en choisissant un « acteur » particulier pour un rôle donné. Celui-ci doit être une activité orthogonale basée sur la spécialisation d'un rôle à partir des données contenues dans le modèle du domaine.

2.3 Le langage de modélisation CAT

Ce premier exemple de définition d'un langage de description de la politique d'usage d'un cadre d'applications est aussi le un des premiers essais réalisés au cours de notre doctorat. Il s'agit de CAT, un langage de description de la politique d'usage de Accounts (présenté dans l'Annexe C)

CAT est un langage de modélisation qui permet de décrire Accounts pour une solution donnée. Il est composé par trois constructeurs principaux, extraits de l'architecture de Accounts : compte, transaction et attribut. La Figure 63 est un exemple de l'utilisation de CAT pour modéliser un système comptable d'un magasin. Ce magasin a deux comptes principaux, Espèces et Bancaire.

Trois attributs composent le premier : le montant des entrées, le montant des sorties et le montant disponible, calculé à partir des deux premiers. Le compte bancaire sert à grouper les différents comptes bancaires utilisés par le magasin, comme les comptes chèques. Un compte chèque possède deux attributs, les montants des versements et des retraits, et sert comme template de tous les comptes chèque.

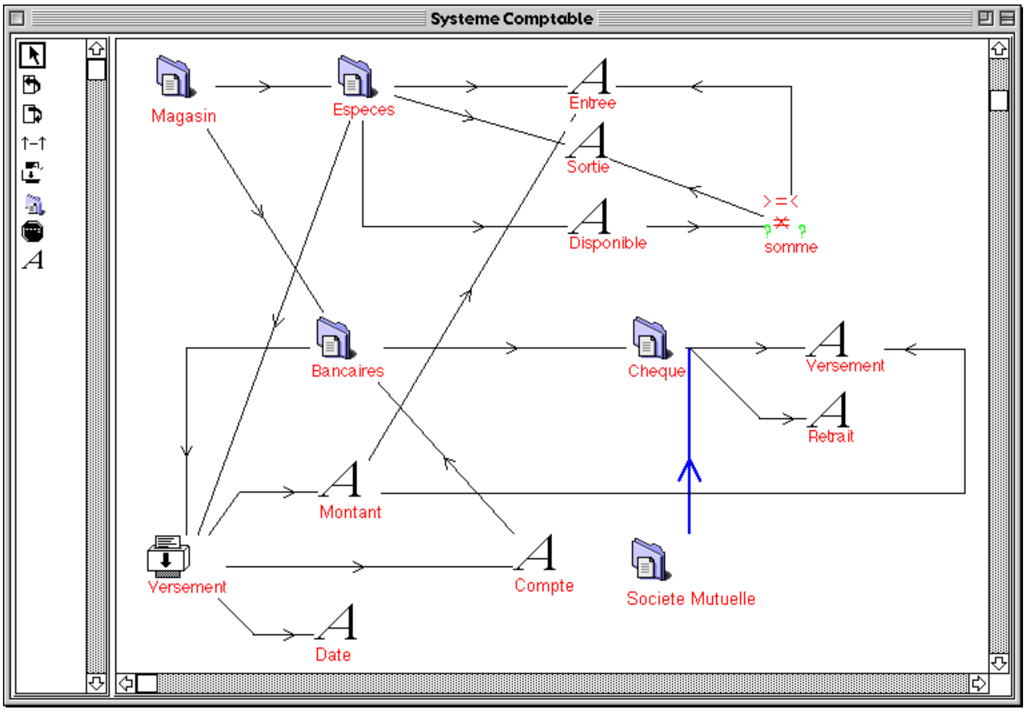

Figure 63 - Modèle CAT d'un système comptable

Une seule transaction a été décrite dans ce modèle, le versement, qui est utilisé pour actualiser le compte 'Espèces', aussi bien que les comptes chèques. Cette transaction possède trois attributs : la date (tout compte possède un attribut date, même s'il n'est pas défini), le montant du versement et le nom du compte. Ce dernier est utilisé seulement pour les comptes bancaires, c'est lui qui décide quel compte, parmi les différents comptes bancaires présents dans le système, sera actualisé. L'attribut 'montant' affecte directement l'attribut 'entrée' du compte espèces ou l'attribut versement des comptes chèque.

Pour une question de simplicité, le langage CAT ne permet pas la description d'un compte comme étant composite ou agrégé. Cependant, ces informations peuvent être déduites des modèles. Par exemple, les comptes espèces et chèque ne possèdent pas de compte, ils sont donc des comptes simples. Le compte bancaire groupe des comptes homogènes, il est donc un compte composite. Le dernier compte, magasin, groupe des comptes hétérogènes, il est un compte agrégé.

Ce modèle permettra l'adaptation du cadre Accounts pour réaliser une application de comptabilité. La manière par laquelle les modèles d'usage des cades d'application sont transformés en code source sera expliquée dans la section suivante.

3. Transformation

Le développement logiciel peut être vu comme une succession d'étapes, où chacune consiste en la transformation d'un modèle vers un autre. Ces modèles servent comme un point de validation d'une étape et sont en réalité une description de l'application selon un point de vue spécifique. Les transformations accomplissent deux choses : (i) elles changent le langage de la source vers la cible et (b) elles ajoutent des détails dégagés des connaissances concernées par cette étape.

Dans le cas particulier du projet MétaFor, au moins une transformation doit être faite : transformer une description fonctionnelle (le modèle du domaine) vers une description informatique (le modèle conceptuel). Le modèle source spécifique ce qui doit être fait, et peut être écrit en des termes de conditions à satisfaire quand une tâche donnée est exécutée. Le modèle cible spécifie comment c'est fait, et peut être écrit en termes de structures qui spécialisent les structures qui composent un cadre d'applications. D'une autre manière, l'un décrit les entrées et sorties, alors que l'autre décrit ce qu'il se passe entre les deux. La différence entre ces deux modèles peut être résumée par les quatre points suivants : (i) le langage utilisé, (ii) le vocabulaire employé, (iii) le niveau de détail et (iv) la nature de la description, fonctionnelle ou informatique.

L'idée d'identifier au moins deux langages de modélisation et de les séparer n'est pas nouvelle dans le contexte du développement assisté par ordinateur [voir Kent, 1981]. De certaine façon, les langages que nous utilisons pour écrire des modèles ont un double rôle : (i) représenter convenablement les concepts du domaine, avec une vue vers la communication et l'usage de l'utilisateur, et (ii) représenter ces mêmes concepts de façon maniable par l'ordinateur. Ce double rôle est souvent l'origine de la confusion dans la tête du concepteur. Pour Kent [Kent, 1981], le langage est plus qu'un moyen de liaison passif entre notre point de vue et le monde réel, il dirige ce point de vue et limite notre perception.

Pour résoudre ce problème, il est important de séparer le rôle d'un objet dans le monde réel de sa représentation dans l'application qui sera développée. Notre point de vue est cohérent avec celui de la communauté de l'acquisition de connaissances, bien que nos buts soient plus pragmatiques : nous allons construire un système qui ne correspond pas complètement au monde réel, du point de vue ontologique.

Pour rendre opérationnelle cette dualité, nous devons répondre à deux questions : (i) comment représenter chaque type de modèle ? et (ii) comment transformer l'un en l'autre ? On peut réponder à ces deux questions dans le contexte d'un environnement de méta-modélisation. Pour écrire des modèles en un langage donné, nous avons besoins d'un autre langage de modélisation qui connaît et comprend le premier. Si nous souhaitons un environnement de modélisation flexible, qui outille les notations employées par des différentes méthodes, les fonctionnalités de l'environnement doivent être configurées par ces notations. En plus, les étapes de développement et les transformations doivent être configurées, i.e., quelqu'un doit rentrer la description de la notation et des transformations de modèle, qui doit être exprimé dans un langage de méta-modélisation.

Comme les transformations utilisent des expertises de développement qui ne peuvent pas être formalisées par une procédure algorithmique, la meilleure manière de les exprimer est l'utilisation de règles de production. En plus, les décisions des développeurs sont souvent le résultat de besoins conflictuels ou compétitifs, qui sont supportés efficacement par des règles compétitives.

3.1 Transformation de modèles

La transformation d'un modèle source vers un modèle cible, décrit dans un langage plus proche de l'environnement applicatif est réalisé en deux étapes : (i) la mise en relation entre les constructeurs des deux langages et (ii) l'ajout de nouvelles informations. Ces informations sont particulières à la manière de modéliser (Weltanschauung) du concepteur et à ses connaissances sur l'ontologie du langage cible.

Toutes les transformations sont décrites de la même manière, i.e. par des règles de production. Plus précisément, par des règles NéOpus [Pachet, 1995]. Le couplage entre NéOpus et l'environnement Smalltalk est intéressant car une règle NéOpus peut contenir de méthodes Smalltalk dans sa partie conditions, aussi bien que dans sa partie actions. Ceci amène une très grande flexibilité à l'écriture des transformations, ce qui assure la faisabilité des transformations.

Le vocabulaire des règles de production est alors très vaste, comprenant toutes des méthodes de Smalltalk. Pour simplifier l'écriture de telles règles, l'équipe MétaFor a défini un ensemble d'opérations de base, permettant la manipulation de modèles. Il est important de signaler que ces opérations ne sont pas définies en fonction des constructeurs représentés dans le méta-modèle, mais en fonction d'une structure sous-jacente. Cette dernière est créée à partir des constructeurs d'un langage représenté dans le méta-modèle et sert de base aux modèles représentés par ce langage. La relation qu'il existe entre une structure sous-jacente d'un langage et les modèles décrits par ce langage est la même que la relation entre classes et instances. La structure sous-jacente et sa génération sont présentées à la section III(2.4) du Chapitre 3, page 75.

3.2 Génération d'applications

En principe, la démarche de développement est indépendante du langage cible, mais l'intégration des outils utilisés dans l'environnement VisualWorks [ParcPlace, 1998], et les facilités de cet environnement, (comme la compilation de classes à la volée pendant une exécution), ont motivé le choix fréquent de Smalltalk 80 comme langage cible. Les expériences menés au sein de l'équipe MétaFor ont employé des différents procédés de génération d'applications, comme décrit par la suite.

Un premier procédé de génération d'une application est l'utilisation de règles de production, applicables aux modèles implémenteur, pour générer le code source dans le langage cible. Le rôle des règles de génération est de représenter les connaissances des différents acteurs du développement logiciel. Dans le cas de l'adaptation d'un cadre d'applications, par exemple, elles représentent la connaissance d'un concepteur qui traduit les besoins décrits dans un modèle implémenteur en un modèle à objets et la connaissance d'un implémenteur qui génère le code correspondant à ce modèle, et qui l'intègre avec d'autres composants disponibles dans l'environnement informatique.

Un deuxième procédé consiste en coupler les modèles implémenteurs avec un moteur d'interprétation de graphes sémantiques, MPVC [Krief, 1990]. Cette approche est très flexible car les modifications réalisées dans un modèle sont immédiatement prises en compte par l'application. Toutefois, le langage cible reste limité à celui de l'environnement de modélisation et du moteur d'interprétation. Dans notre cas, à Smalltalk 80. Ce procédé a été utilisé pour développer une application de simulation de la fermentation du vin, développé en coopération avec le Cemagref [Agabra *et al.*, 1997].

Un dernier procédé utilisé est une approche mixte, le développement d'une partie du code source de l'application et l'interprétation d'un modèle. Ce procédé est très intéressant car il permet l'évolution dynamique d'une application à travers des changements effectués dans le modèle implémenteur. Lorsque cette application devient stable, le code peut être généré entièrement.

III. Exemple d'implémentation

Les concepteurs de logiciels utilisent fréquemment une forme de représentation graphique pour modéliser des aspects d'un système, e. g. diagrammes de flots de données, diagrammes d'état, etc. Cette représentation graphique prend souvent la forme d'un réseau sémantique, composé par deux figures élémentaires : les nœuds et les liens. Les nœuds sont représentés par un symbole (e. g. rectangle, cercle) et les liens par une ligne simple avec quelque chose de plus (flèche, petits cercles, etc.).

L'objectif de l'exemple que nous présentons par la suite est d'utiliser la démarche de développement décrite auparavant pour créer un environnement de génération d'éditeurs graphiques, dans lequel les concepteurs pourront décrire l'éditeur qu'ils souhaitent, en termes de concepts simples, comme les nœuds et arcs. A partir d'une simple combinaison de ces concepts, l'environnement générera l'éditeur souhaité par le concepteur.

Dans cette expérience, nous avons utilisé HotDraw, un cadre d'applications conçu pour assister l'implémentation d'éditeurs graphiques à deux dimensions. En résumé (voir Figure 64), l'expérience consiste en transformer la description d'un éditeur en des sous-classes de HotDraw, qui implémentent un éditeur graphique. Cette expériences est une amélioration d'une expérience similaire appelée Geds, qui est présentée dans l'Annexe E de ce document.

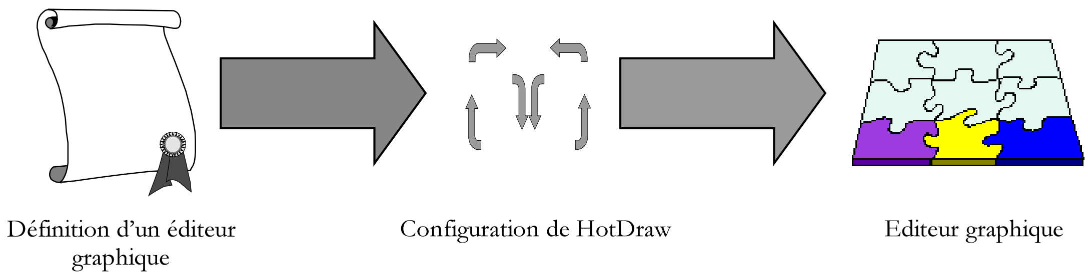

Figure 64 - Un environnement de génération d'éditeurs graphiques

Pour permettre aux concepteurs logiciels de représenter la description de leurs éditeurs, nous avons conçu un langage de modélisation appelé PAN. La transformation d'un modèle PAN en une spécialisation de HotDraw passe par une étape intermédiaire, un modèle représentant l'utilisation de ce cadre d'applications.

Ces modèles se construisent grâce à un deuxième langage de modélisation, appelé Thef, conçu pour utiliser HotDraw. Deux bases de règles complètent l'environnement, la première transforme les modèles décrits en OPRR en des modèles décrits par Thef, et la deuxième génère le code Smalltalk nécessaire pour spécialiser HotDraw et réaliser l'éditeur décrit préalablement.

Comme exemple d'utilisation de ces deux environnements de génération d'éditeurs graphiques, nous avons choisi de générer des éditeurs de diagrammes de flots de données DFD [DeMarco, 1982], dont la sémantique est décrite ci-dessous. Ensuite, nous décrirons notre expérience.

1. Diagrammes de flot de données (DFD)

Les diagrammes de flot de données servent à représenter les aspects dynamiques des systèmes d'informatique. Ils mettent en évidence les flots de données, des informations générées ou envoyées à des dépôts de données ou à des acteurs externes, et traitées par des processus.

La Figure 65 est un exemple d'un diagramme représentant les flots de données présents durant le processus de commande et de livraison d'un produit. Un client est un acteur externe qui fait une commande et déclenche en conséquence un processus de contrôle de commande. Ce processus lit des informations stockées dans les dépôts de données client et produits, élabore ensuite une commande et la stocke dans un troisième dépôt de données, et déclenche finalement un deuxième processus. Celui-ci lit les informations du dépôt de commandes, s'occupe de la livraison du produit, élabore une commande, l'ajoute au dépôt de factures et déclenche le processus de contrôle de factures. Ce dernier processus lit les informations concernant la facture et la délivre au client.

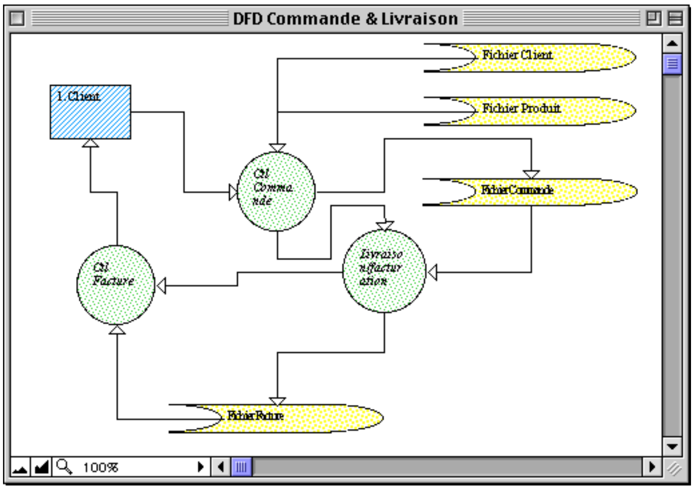

Figure 65 - Diagramme de Flots de Données

Les acteurs externes, les dépôts de données et les processus sont représentés respectivement par des carrés, des rectangles et des cercles, et identifiés par un numéro et un nom. Les flots de données sont représentés par des flèches, et symbolisent les informations échangées entre acteurs et processus, dépôts et processus et entre différents processus. Il n'existe pas de flot de données entre des acteurs et des dépôts.

2. Notre expérience

Le principe de l'environnement réalisé est le suivant : des éditeurs graphiques sont décrits dans un langage du domaine, et générés ensuite à partir de l'application de règles de production. Le langage de domaine, appelé PAN, est doté de seulement trois constructeurs. Le langage de description de l'usage de HotDraw a été baptisé Thef. Son objectif est permettre la représentation d'un nombre varié d'éditeurs graphiques.

La présentation de cette expérience est divisée quatre parties. En premier, nous présentons le langage qui servira aux utilisateurs de décrire l'éditeur graphique souhaité. L'objectif principal de ce langage est de limiter l'usage de HotDraw à une métaphore spécifique : la création d'éditeurs de réseaux sémantiques. Ensuite, nous proposons un langage conceptuel de modélisation de l'usage de HotDraw. La troisième partie consiste en la présentation de la base de règles de transformation de modèles. Cette base de règles est basée sur les constructeurs de deux langages de modélisation et nous permet de transformer des modèles PAN en des modèles Thef. Finalement, nous présentons une base de règles de transformation de modèles Thef en code Smalltalk.

L'entrée de notre environnement de génération est la description, en PAN, d'un éditeur de diagrammes de flots de données. Ce modèle est présenté dans la Figure 67. Par la suite, nous appliquons deux bases de règles à ce modèle. La première génère un deuxième modèle, représenté dans le langage Thef. Ce modèle est présenté par la Figure 70. La deuxième base de règles génère le code Smalltalk relatif à l'éditeur de DFD présenté par la Figure 66.

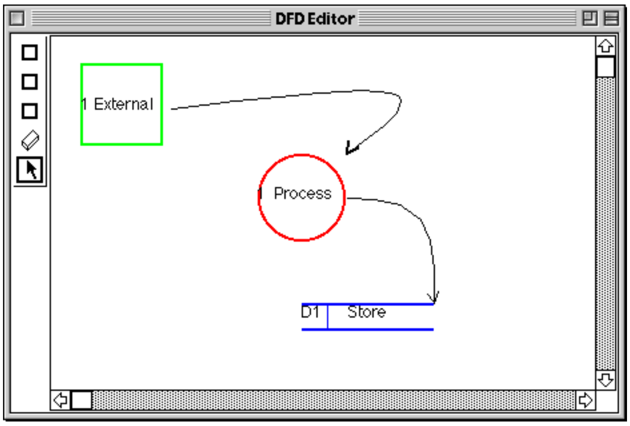

Figure 66 – L'éditeur DFD généré

Nous ne présenterons pas dans cette description la totalité des règles de production, mais simplement la logique d'exécution de celles-ci. Aussi, nous ne présenterons qu'une petite partie du code généré.

2.1 Le langage PAN

Le langage pan est formé par trois constructeurs : 'Nœud', 'Arc' et 'Propriété'. Les nœuds correspondent aux nœuds graphiques des réseaux sémantiques et sont liés par des arcs. Un nœud peur avoir des propriétés, comme un libellé, qui le différencie des autres nœuds. Les propriétés ont deux attributs : position et valeur initiale. La première définit la position de la propriété dans le nœud graphique. Elle peut assumer une des cinq valeurs suivantes : haut, bas, gauche, droite et centre. La deuxième propriété établie sa valeur initiale. Les arcs possèdent aussi deux propriétés : flèche et ligne. La première établit si l'arc est directionnel, et la deuxième s'il est représenté par une ligne droite ou par une ligne courbe.

La représentation graphique de PAN est la suivante : les nœuds sont représentés par des cubes, les arcs par des flèches et les propriétés par des serpents.

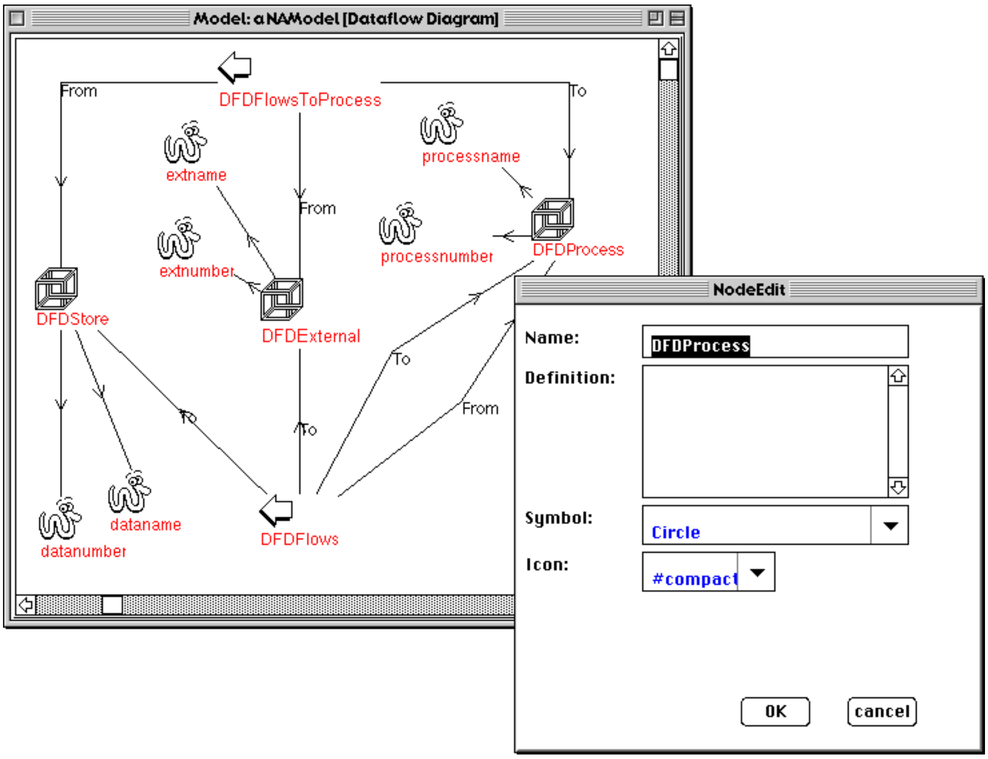

Figure 67 - Modèle PAN d'un éditeur de DFD

Un modèle PAN d'un éditeur de diagrammes de flots de données est montré par la Figure 67. Le modèle contient trois nœuds : 'DFDProcess', 'DFDExternal' et 'DFDStore', qui correspondent aux constructeurs processus, acteur externe et dépôt de données, respectivement. Les processus sont représentés par des cercles. Les rubriques ne sont pas visibles directement dans l'éditeur, mais par une boîte d'édition, présentée à droite. Ils sont identifiés par un numéro et un libellé. Un processus peut envoyer des données à d'autres processus, à des acteurs externes et à des dépôts de données. L'envoie correspond à l'arc 'DFDFlows', qui est représenté par une ligne courbe, orientée.

Les acteurs externes sont représentés par un carré et sont identifiés par un numéro et un libellé. Les dépôts de données sont représentés par deux lignes horizontales et sont, comme les deux autres nœuds, identifiés par un numéro et un libellé. Ces deux propriétés sont placées entre les deux lignes horizontales. Les acteurs et les dépôts peuvent seulement envoyer des données à des processus. Ils ne peuvent pas envoyer des données à d'autres acteurs ou à d'autres dépôts. Ceci est représenté par l'arc 'DFDFlowsToProcess'.

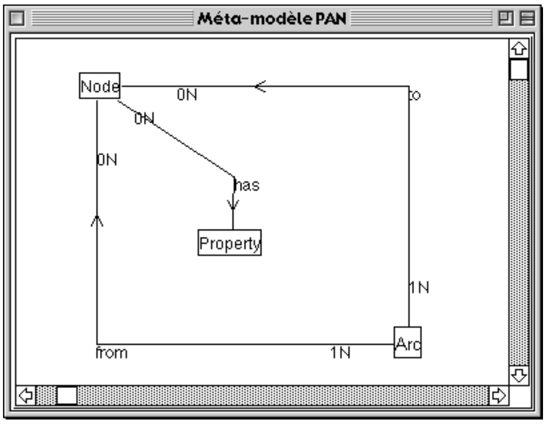

Figure 68 - Méta-modèle PIR3 du langage PAN

La Figure 68 représente le méta-modèle du langage PAN. C'est grâce à ce méta-modèle que MétaGen a généré l'éditeur et les boîtes présentés dans la Figure 67. Son interprétation est simple, il est composé par trois individus, 'Nœud', 'Propriété' et 'Arc'. Un nœud peut posséder entre zéro et plusieurs propriétés. Un arc part d'un ou plusieurs nœuds vers un autre ou plusieurs nœuds.

2.2 Le langage Thef

L'objectif du langage de modélisation Thef est de permettre la description de l'usage de HotDraw pour la réalisation d'éditeurs graphiques. Les constructeurs du langage Thef reflètent des concepts extraits de la conception de HotDraw (e.g. éditeur, figure, outil, poignée).

Les éditeurs sont les individus centraux d'un modèle Thef. Ils représentent la totalité d'un l'éditeur graphique. Plus précisément, la sous-classe de DrawingEditor qui sera générée. Un éditeur possède une palette d'outils, qui détermine quelles sont les actions qui peuvent être effectuées dans l'éditeur. Nous aurions pu représenter ce concept par la totalité d'un modèle, et considérer que tout composant d'un modèle appartient à un même éditeur. Le fait de le représenter par un individu augmente l'explicité des modèles et permet la génération de différents éditeurs au même temps, qui partagerait des figures ou des commandes, par exemple.

Un éditeur peut contenir deux sortes de figures, les figures élémentaires, i.e. rectangle, ellipse, texte, ligne, image, etc. et les figures composite, qui permettent la création de nouvelles figures à partir de la combinaison de plusieurs figures élémentaires. Les figures composite deviendront plus tard des sous-classes de 'CompositeFigure'. Toute figure doit posséder un outil de création, qui fera partie de la palette d'outils de l'éditeur.

Les outils possèdent trois propriétés : l'icône qui apparaîtra dans la palette d'outils, le pointeur utilisé lors que l'outil est actif, et la commande qui sera exécutée. Thef ne permet pas la définition de nouvelles commandes, seulement l'utilisation de celles déjà présentes dans HotDraw, comme la sélection ou l'effacement d'une figure.

Finalement, des poignées peuvent être attachées à une figure. Il existe deux sortes de poignées : celles de connexion, qui permettent qu'une figure soit liée à une autre, et les poignées de commande, qui permettent le déclenchement d'une commande.

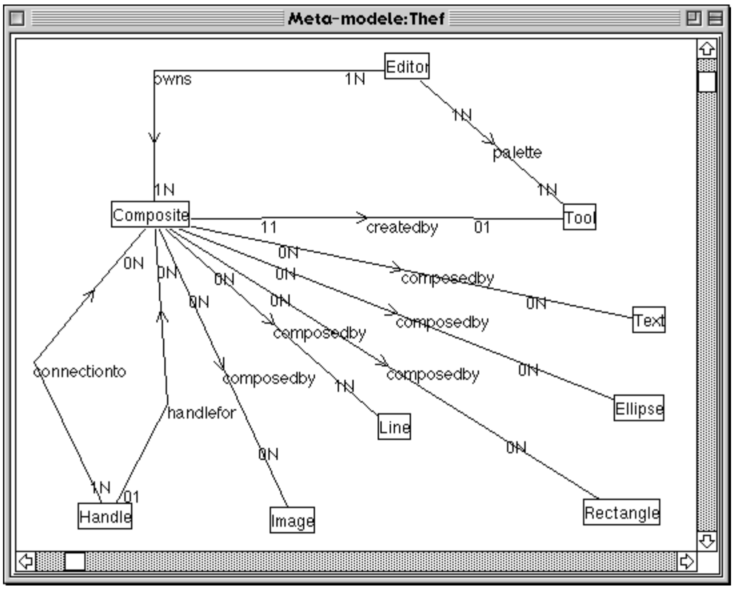

Figure 69 - Méta-modèle PIR3 du langage Thef

La Figure 69 présente le méta-modèle de Thef. Le méta-individu central de ce méta-modèle est 'Editor', qui correspond à un éditeur graphique HotDraw. Ce méta-individu possède une seule rubrique ('name'), son nom. Il est origine de deux méta-relations : 'owns' et 'palette'. La première établie qu'un éditeur doit avoir au moins une figure composite pour exister (cardinalité 1N), et qu'il peut en avoir plusieurs. La deuxième sert à former la palette d'outils de l'éditeur.

Le méta-individu 'Composite' permet l'expression de figures composites. Il doit avoir exactement une relation avec un individu de type 'Tool' (son outil de création), comme le définit la cardinalité de la méta-relation 'created by' (11). Les méta-relations 'composed by' admettent qu'une figure composite soit formée par des différentes figures élémentaires (i.e. les méta-individus 'Text', 'Ellipse', 'Rectangle', 'Ligne' et 'Image'). En même temps, elle permet le partage de figures élémentaires, par la cardinalité de sortie (0N). Le méta-individu 'Composite' possède deux rubriques : 'extent' (taille totale de la figure) et 'connection' (lieu de connexion accepté). L'apparence des figures élémentaires est déterminée par quelques-unes unes de leurs rubriques, comme point de départ et point d'arrivée pour les ellipses, rectangles et lignes ; image, pour les figures image ; ou texte, position et taille, pour les figures textuelles.

Finalement, le méta-individu 'Handle' (poignée) a deux méta-relations avec 'Composite'. La première ('handle for') l'attache à une figure, aidé par la rubrique 'position' (de la poignée dans la figure) et par la rubrique 'figure' (utilisé pour établir la connexion). La deuxième ('connection to') détermine les figures acceptées comme cible de la connexion.

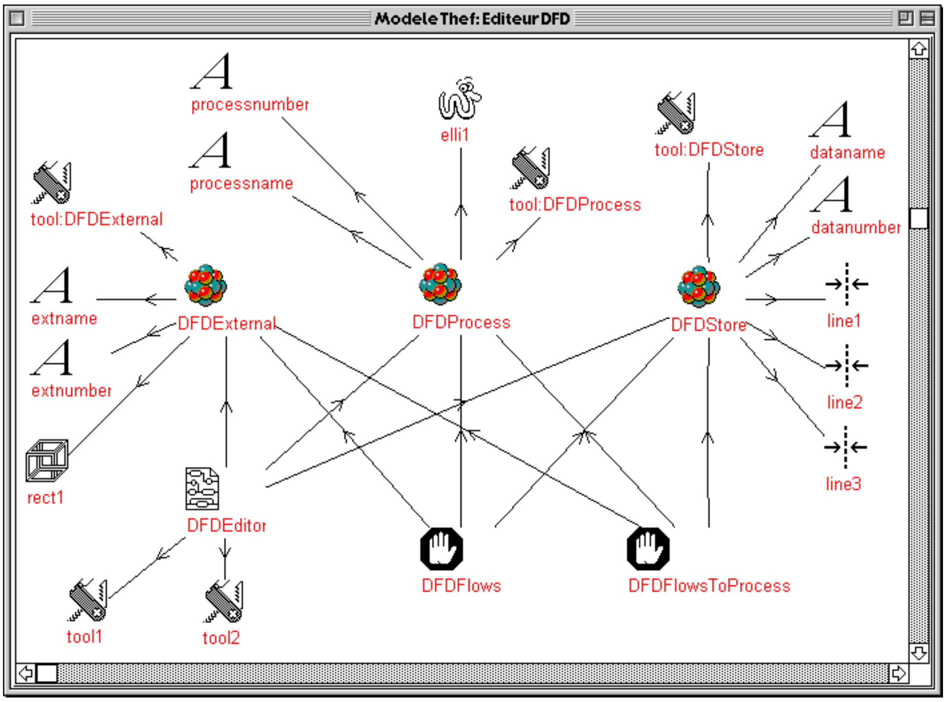

Figure 70 - Modèle Thef d'un éditeur de DFD

La Figure 70 introduit un modèle Thef d'un éditeur de flots de données. Ce modèle a été généré automatiquement à partir du modèle PAN présenté dans la Figure 67. Trois figures composites sont présentées au centre de l'image, représentées par une molécule. Chacune a son outil de création (les couteaux suisses) et plusieurs figures élémentaires. Par exemple, la figure DFDExternal est composée par trois figures élémentaires, un rectangle et deux textes. En bas de l'image, deux poignées (représentées par des mains) permettent des connexions entre ces figures.

2.3 Transformation de modèles PAN en des modèles THEF

Nous énumérons par la suite les principes de la transformation de modèles PAN en des modèles Thef. Ces principes sont divisés en deux parties. La première génère le modèle source et traduit les concepts exprimés en Pan en des concepts Thef. La deuxième calcule la taille des figures composite, et en fonction de celle-ci, la position exacte des propriétés utilisées comme libellées (la position étant établie par des valeurs relatives).

Transformation

1. Création d'un modèle Thef ;

2. Création d'un individu de type 'Editor' dans ce nouveau modèle. Le nom de l'éditeur est emprunté au modèle source. Ajout de deux individus de type 'Tool' à cet éditeur, nommés 'sélection' et 'effacement' ;

3. Chaque 'Nœud' représenté par un symbole devient un individu de type 'Composite' dans le modèle cible. Ajout, à cet individu, de toutes les figures élémentaires qui le composent ;

4. Les 'Nœuds' non représentés par un symbole deviennent aussi une figure composite. Ajout, à cette figure, d'un individu de type 'CachedFigure' ;

5. Chaque 'Arc' devient une poignée de la figure composite qui correspond au nœud source de l'arc. La destination de la poignée est la figure correspondant au nœud de destin de l'arc.

Calcul

1. Calcul, dans le modèle cible, de la taille de chaque figure composée, à partir de toutes les figures élémentaires qui la composent ;

2. Finalement, chaque propriété d'un arc devient un individu 'Text', qui fait partie de la figure composite correspondante à l'arc. La position du texte sera calculée en fonction de la taille de la figure qui le contient.

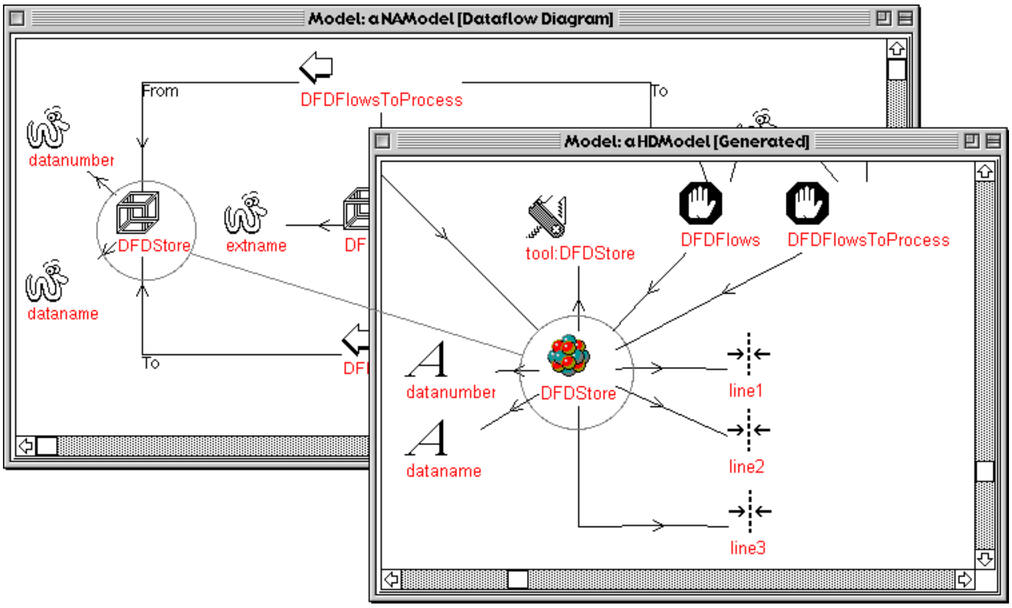

Figure 71 - De Pan à Thef

La Figure 71 est un exemple de l'application de cette base de règles. Le nœud 'DFDStrore' a été transformé en une figure composite, de même nom. Cette figure est composée par cinq figures élémentaires, possède un outil de création et une poignée de connexion ('DFDFlowsToProcess'), et est admise comme cible d'une autre poignée de connexion ('DFDFlows'). Les positions des figures élémentaires, en relation à la figure composite, ont été calculées pendant la transformation.

2.4 Génération de code Smalltalk

La transformation d'un modèle Thef en code Smalltalk se fait aussi par une base de règles. La base est divisée en deux parties, la première analyse le modèle et la deuxième génère le code. La séquence suivante décrit l'exécution des règles :

1. Vérification de l'intégrité du modèle ;

2. Calcul, si nécessaire, de la taille de chaque figure, en considérant tous les éléments qui la composent ;

3. Génération d'une sous-classe de 'DrawingEditor' à partir de chaque individu 'Editor' ;

4. Génération d'une sous-classe de 'CompositeFigure' à partir de chaque figure composite. En même temps, création d'une méthode de construction de la figure, en fonction des figures élémentaires ;

5. Génération d'une méthode correspondante à un outil de création pour chaque sous-classe de 'CompositeFigure' générée. Cette méthode sera appelée lors que la palette d'outils sera créée.

6. Génération d'une méthode qui établit la position d'attachement d'une figure, dans la sous-classe de 'CompositeFigure' correspondante à la figure de destin d'un 'Handle' ;

7. Génération d'une méthode qui ajoute une poignée de connexion pour chaque sous-classe de 'CompositeFigure' dont la figure correspondante est la source d'un individu 'Handle'. La figure utilisée pour la connexion est aussi établie par cette méthode ;

8. Génération d'une méthode de création de la palette d'outils dans chaque sous-classe de 'DrawingEditor' correspondante à un éditeur. La palette sera composée par tous les outils connectés à cet éditeur, plus les outils de création de toutes les figures.

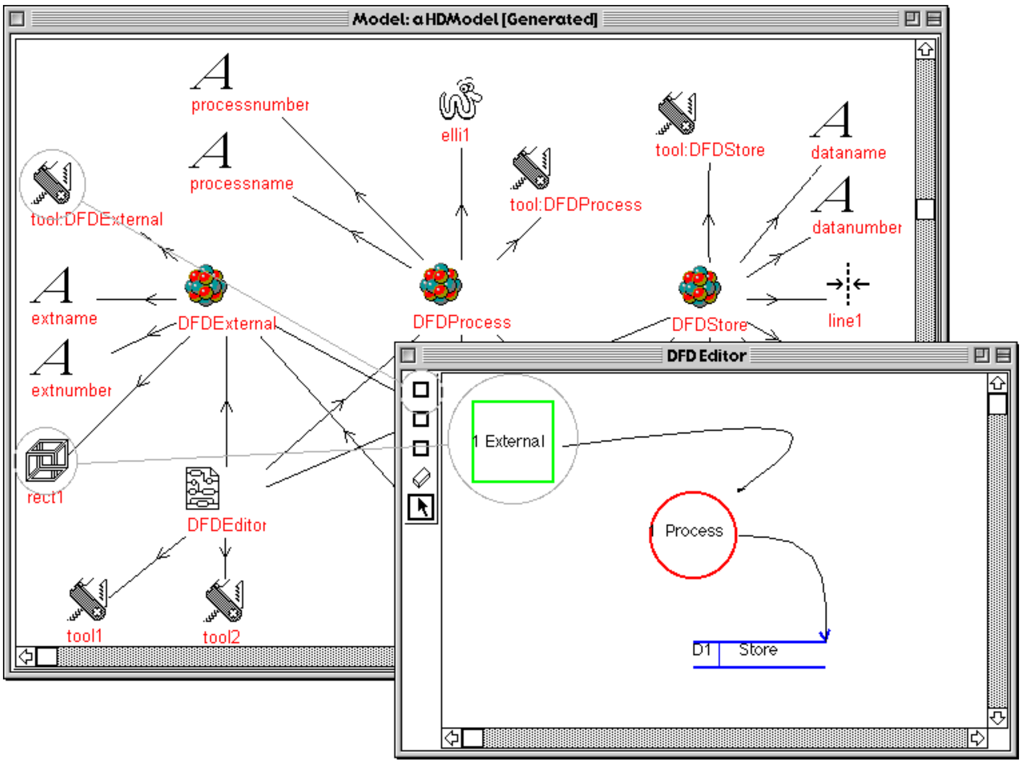

Figure 72 - De Thef à HotDraw

Le modèle Thef d'un éditeur et l'éditeur généré, après l'application de la base de règles décrite ci-dessus, sont présentés dans la Figure 72. L'éditeur généré dispose d'une palette de trois outils de création de figures, un outil de suppression et un outil de sélection, qui correspondent aux cinq outils présents dans le modèle Thef. Les trois outils de création donnent origine aux trois figures composite présentes dans partie dessin (à droite) de l'éditeur.

```
compositeInitialize
|HDComposite fig. Association ass|
      fig notNil. ass notNil. ass key = fig.

actions
|vars method corps var|
vars := OrderedCollection new.
corps := ‘’.
elements := ( fig allTargetsAt: #HDFigline), ( fig allTargetsAt:
      #HDFigtext), (fig allTargetsAt: #HDFigellipse), (fig
      allTargetsAt: #HDFigrectangle), (fig allTargetsAt:
      #HDFigimage).
elements do: [:each |  var := each name value asLowercase.
      corps := corps, ‘\’, var, ‘ := ‘, (each creationMessage).
      vars add: var].
method :=  ‘initializeAt: aPoint \ | myFigures ‘.
vars do: [:each | method := method, ‘ ‘ , each].
method := method, ‘ | \’, corps.
method := method,  ‘\myFigures := (OrderedCollection new) ‘.
vars do: [:each | method := method, ‘\add: ‘, each, ‘;’].
method := method, ‘\yourself.’.
method := method, ‘\self setFigures: myFigures visibleArea: (aPoint
      extent: ‘, fig extent value, ‘).’.
ass value compile: (method) withCRs classified: ‘initialize-release’
      notifying: nil.
```

Nous présentons ci-dessous un exemple d'une règle de transformation. Plus précisément, celle décrite dans (4), la création d'une méthode de construction des figures composite. Dans sa partie conditions, cette règle filtre une occurrence de figure composite et une association, utilisée pour retrouver la classe Smalltalk où la méthode sera introduite. Cette classe est générée par une règle précédente.

La partie actions de cette règle compose un texte, qui sera ensuite compilé et ajouté à une classe. Pour cela, elle fait appel à la méthode ‘creationMessage' (le message utilisé pour créer une figure élémentaire), écrite en Smalltalk. Une des méthodes créées par l'application de cette règle est présentée ci dessous :

```
initializeAt: aPoint
|myFigures line1 line2 line3 text1 text2 |
line1 := (LineFigure start: aPoint + ( 1@1) stop: aPoint + (100@1))
      setLineColor: #blue; width: 2.
line2 := (LineFigure start: aPoint + ( 1@20) stop: aPoint +
      (100@20)) setLineColor: #blue; width: 2.
line3 := (LineFigure start: aPoint + ( 20@0) stop: aPoint + (20@20))
      setLineColor: #blue; width: 1.
text1 := (FixedTextFigure string:’D1’ at: aPoint + (0@0)) recompose;
      maxLength: 10.
text2 := (FixedTextFigure string:’Store’ at: aPoint + (25@0))
      recompose; maxLength: 60.
myFigures := (OrderedCollection new) add: line1; add: line2; add:
      line3; add: text1; add: text2; yourself.
self setFigures: myFigures visibleArea: (aPoint extent: 100@20).
```

2.5 Perspectives

Le langage Thef, présenté dans cette section, est un résultat de nombreuses itérations faites par des expériences similaires à l'éditeur DFD présenté ici (voir Annexe E). Il reflet un compromis entre la simplicité et l'amplitude de la génération de code, la complexité de la base de règles de transformation de modèles et la simplicité du langage PAN.

Actuellement, Thef ne permet pas l'expression de toutes les capacités de HotDraw, mais il constitue un départ important pour cela. L'addition de nouveaux constructeurs, permettant la création de commandes et surtout la configuration du système de contraintes présentes dans ce cadre d'application seront des étapes importantes dans l'évolution de ce langage.

IV. Intégration de PatternGen à la démarche de l'équipe MétaFor

Nous présentons dans cette section une proposition d'évolution à la démarche de développement décrite dans ce chapitre. Cette proposition commence par une critique de la démarche actuelle, suivie par une proposition de modification de la démarche et par une adaptation des exemples présentés précédemment à cette proposition.

1. Critique de la démarche actuelle

Construire un langage d'adaptation d'un cadre d'applications (comme HotDraw ou Accounts), est une tâche difficile, mais motivante. Dans un sens, ce langage reflète la « vu du monde » d'un cadre d'applications (ou de son concepteur). Par exemple, le cadre MVC voit le « monde des applications avec des interfaces graphiques » comme un ensemble de triades, chacune composée par un modèle, une vue et un contrôleur.

HotDraw, qui a été construit autour du cadre MVC, introduit des concepts plus spécialisés, comme 'Figure', 'Editeur', 'Dessin', etc. Lors de l'ingénierie d'un langage de modélisation, l'association de chacun de ces concepts avec un constructeur du langage de modélisation est tentatrice. Cela signifierait que la « vue du monde » du monde du cadre est très proche à la vue du code d'implémentation. Cette vue ne serait pas correcte, car, si elle simplifie et optimise la génération de code, elle compliquerait la transformation entre les modèles utilisateurs et les modèles d'adaptation du cadre d'applications. Cette complexité se serait manifestée, soit dans les règles de transformation, soit dans les langages utilisateur (dans ce cas, PAN), qui devrait incorporer plus de détails de l'utilisateur. En plus, d'un point de vue conceptuel, le langage ne refléterait pas, dans le niveau correct d'abstraction, la vue du monde du cadre d'applications. Ceci pourrait demander des changements dans la base de règles de génération car l'adaptation du cadre changerait un peu.

Le langage Thef est le résultat de plusieurs expérimentations de génération d'éditeurs graphiques similaires à l'éditeur DFD montré ici. Il reflète des compromis entre la simplicité (et la porté) de la génération de code, la complexité de la base de règles de transformation, et la simplicité du langage utilisateur.

La mise en œuvre de langages d'adaptation de différents cadres d'applications, comme HotDraw, Accounts ou DataModeller nous a permis d'effectuer plusieurs observations sur la démarche de développement de l'équipe MétaFor, concernant l'expression du point de vue des développeurs.

Tout d'abord, une partie de la génération de code est fortement dépendante des facilités du langage cible, et non du cadre d'applications. Ces facilités peuvent être vues comme des patterns d'implémentation spécifiques au langage. Dans le cas de HotDraw, implémenté en Smalltalk 80, l'adapter signifie utiliser une partie non-triviale de la bibliothèque de classes de cet environnement. Cela inclut l'utilisation de menus, contrôleurs, modèles, vues, etc. Ce serait intéressant d'explorer la configuration de la génération de code, en employant un modèle explicite du langage cible ou des patterns de conception qu'il implémente. Cette caractéristique serait tout au plus appréciable dans l'émergence d'un environnement de développement permettant la génération de code en différents langages.

Une deuxième observation concerne l'intégration de plusieurs cadres d'applications. Il est très difficile d'imaginer qu'un seul cadre pourra répondre aux différents besoins d'une application. En effet, une application est mise en œuvre par l'intégration de plusieurs cadres d'applications. Cela signifie que l'implémentation doit combiner des constructeurs empruntés de différents langages d'adaptation.

Cette combinaison peut être réalisée de deux manières. La première consiste en créer un langage implémenteur doté d'une vaste gamme de constructeurs, extraits de plusieurs cadres. Cette approche est rejetée d'emblée car d'une part les modèles mélangerait des concepts exprimant les différents aspects d'une application et seraient incompréhensibles, et d'autre part, cette approche ne motive pas la réutilisation des connaissances d'intégration des cadres d'applications. La deuxième manière consiste en permettre des liens d'intégration entre des modèles d'adaptation de différents cadres d'applications. Il est de notre avis que ces liens d'intégration peuvent être établis par l'explicitation de différents patterns de conception.

Une dernière observation concerne l'approche de développement par deux systèmes de modélisation [Sahraoui, 1995] ou par double méta-modélisation [Revault, 1996]. La séparation des points de vue des utilisateurs et des développeurs est très importante car elle permet aux utilisateurs de décrire (et comprendre la modélisation de) leurs problèmes, indépendamment des contraintes informatiques existantes. Toutefois, l'utilisation d'un seul langage (ou méta-modèle) par point de vue, nous paraît une contrainte trop rigide.

Lorsque le développement d'un système devient plus complexe, dû soit par les caractéristiques propres au domaine d'application, soit par la variété des domaines technologiques impliqués dans le développement, cette complexité se reflète dans les langages de modélisation et dans les transformations. Dans ces cas, il est plus souhaitable de diviser cette complexité par l'utilisation de plus de deux langages de modélisation et par conséquent, de plus de deux bases de règles.

2. Notre proposition

Les difficultés observées ci-dessous peuvent être atténuées par inclusion d'une étape supplémentaire à la démarche, qui se traduit par un modèle représenté en MAC (le langage de l'éditeur de diagrammes de PatternGen). L'objectif de cette étape est double : permettre la réutilisation de solutions connues de conception et simplifier l'intégration de différents cadres d'applications.

En effet, les patterns peuvent générer une partie non négligeable du code nécessaire à l'adaptation d'un cadre d'applications. Plus précisément, certains patterns de création (comme **Monteur**, **Prototype** ou **Fabrication**) sont particulièrement utiles à l'adaptation d'un cadre car ils s'intéressent à la création d'un objet par la combinaison d'objets existants.

L'utilisation des patterns de conception pour intégrer les cadres d'applications n'est pas une idée nouvelle. Cette solution a été exploitée par Meijler et Engel durant le développement de FACE [Meijler et Engel, 1996]. Ce choix s'explique facilement : certains patterns (e.g. **Adaptateur**, **Pont**, **Proxy**) s'attaquent à l'intégration de classes existantes.

Nous présentons dans la section suivante, un exemple d'utilisation de patterns dans la génération du code d'adaptation d'un cadre d'applications (l'intégration de cadres ne sera pas abordée dans cette thèse). Le cadre utilisé sera le même des exemples précédents, HotDraw. Le pattern **Fabrique Abstraite** est utilisé pour implémenter la méthode de création des spécialisations de la classe 'CompositeFigure'. Ce même pattern peut être utilisé pour implémenter les comptes du cadre d'applications Accounts.

3. De Thef à MAC

Pour illustrer notre proposition, nous avons réécrit les règles de génération de code de Thef (§ III - 2.4) de manière à les faire générer un modèle MAC et à ajouter dans ce modèle une instance du pattern **Fabrique Abstraite** et les relations avec les classes participantes de cette instance. Pour ce faire, nous avons modifié le méta-modèle de MAC de manière à ajouter ce pattern.

L'objectif de ce pattern est de fournir une interface pour la création d'objets interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes. Sa représentation dans le méta-modèle de PatternGen est similaire à celles des autres patterns et ne sera pas présentée ici.

La Figure 73 présente le modèle MAC généré à partir de la description en Thef (voir Figure 70) d'un éditeur de DFD. Ce modèle présente une partie des classes de HotDraw et les classes générées de mise en œuvre de l'éditeur ('DFDEditor', 'DFDProcess', 'DFDExternal' et 'DFDStore'). Une instance du pattern **Fabrique Abstraite** est présente dans le modèle (les deux autres ont été masquées pour des questions de lisibilité), et indique que la classes 'DFDProcess' joue le rôle de fabrique.

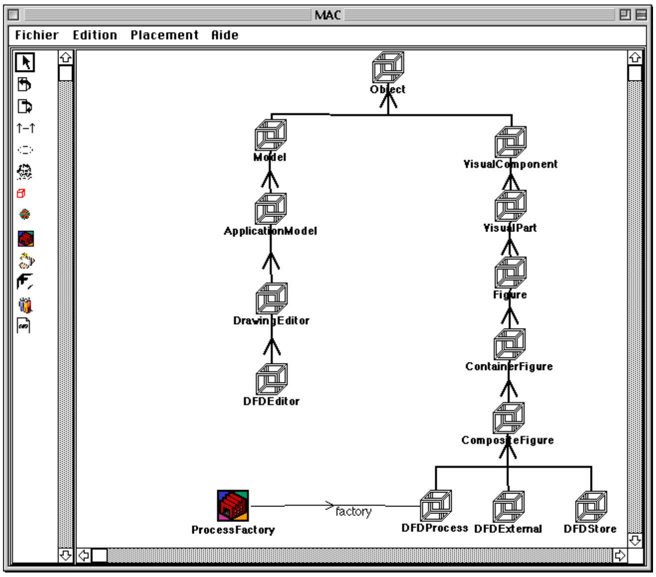

Figure 73 – Modèle MAC d'un éditeur de DFD

Cette instance spécifie aussi, par des relations (non visibles sur la figure), les méthodes de création des classes 'produit', comme 'string:at:' de la classe 'FixedTextFigure' et 'ellipse:' de la classe 'EllipseFigure'. Chaque relation possède une propriété 'arguments' contenant les arguments passés en paramètre. Cette instance de pattern a engendré la méthode présentée ci-dessous, qui permet la création d'une collection de figures. Cette méthode sera utilisée en complément à la méthode 'initializeAt:', dont le contenu a été simplifié.

```
createDFDProcess: aPoint
| collection |
collection := (Array new)
collection add: (FixedTextFigure string:'Process' at: aPoint +
      (13@24)).
collection add: EllipseFigure ellipse: (Rectangle origin: aPoint
      +(5@5) extent: (60@60)).
collection add: (FixedTextFigure string:'1' at: aPoint + (0@24)).
^collection.
```

Cet exemple simple est un premier aperçu des avantages de l'utilisation des patterns (et de PatternGen) dans notre démarche de développement. Plus précisément, les patterns peuvent simplifier l'écriture des règles de génération, qui n'ont pas besoin de remettre en œuvre des solutions déjà connues. Toutefois, la génération d'une partie du code d'adaptation du cadre d'applications, qui est spécifique à l'utilisation du cadre lui-même et qui n'utilise pas de pattern de conception, n'est pas simplifiée.

V. Conclusion

Dans ce chapitre, nous avons présenté une démarche de développement qui préconise une séparation entre les connaissances des utilisateurs et des développeurs pendant le processus de développement d'applications. Son objectif est d'offrir à la fois des moyens d'expression plus adaptables au métier des utilisateurs et à l'environnement applicatif existant. Pour cela, cette démarche combine des techniques de méta-modélisation et une approche de développement par des transformations de modèles.

Plus précisément, elle utilise un langage de méta-modélisation, appelé PIR3, pour permettre l'ingénierie de deux langages de modélisation. Le premier composé par des constructeurs fortement liés au métier de l'utilisateur. Le deuxième est en réalité un langage d'adaptation d'un cadre d'applications. Ce cadre est choisi préalablement et doit être adapté au développement d'applications dans ce métier.

PIR3 possède peu de constructeurs et se veut indépendant de la notation utilisé par le langage qu'il représente (et donc, méta-modélise) et du domaine que ce langage servira à modéliser. Nous l'avons employé pour créer des méta-modèles de différents langages de modélisation pour des différents domaines. PIR3 est aussi utilisé comme base pour l'écriture de transformation entre de modèles de différents langages. Contrairement aux approches formelles de transformation de modèles comme [Oei et Hemmen, 1994] ou [Verschaeve *et al.*, 1996], nous utilisons une approche de transformations par règles de production. Outre les transformations, ces règles nous permettent d'exprimer les différentes connaissances présentes pendant le processus de développement.

Nous avons illustré cette démarche par le développement d'un outil de génération d'éditeurs graphiques, à partir d'une description simple des graphes qui doivent être acceptés par un éditeur. Dans ce ca précis, des diagrammes de flots de données.

Conclusion

Dans cette thèse, nous avons présenté l'outil PatternGen, le méta-outil utilisé pour le développer, MétaGen et la démarche de développement de l'équipe MétaFor.

PatternGen est un outil de mise en œuvre de patterns de conception qui prend en compte les variantes d'implémentation. Pour cela, il intègre (i) un éditeur de diagrammes capable de représenter le modèle de conception d'une application et d'y spécifier des instances de patterns ; (ii) un méta-modèle permettant la représentation de patterns ; et (iii) une base de règles représentant les connaissances nécessaires à l'implémentation de ces instances.

Les fonctionnalités principales de PatternGen ont été présentées par deux exemples d'utilisation. Le premier exemple illustre l'approche générative de l'utilisation : le développement d'une application à partir de la spécification des instances de patterns présentes dans son modèle de conception et des compromis d'implémentation spécifiques à ces instances. Le deuxième exemple illustre, quant à lui, l'approche de reconstruction : la transformation d'un ensemble de classes d'une application existante en une instance d'un pattern.

La représentation de patterns dans PatternGen a été illustrée par quatre exemples. Cette représentation comprend l'inclusion du pattern dans le méta-modèle, le choix de la représentation des compromis d'implémentation et l'écriture d'un ensemble de règles de production. Ces règles, qui expriment la connaissance de mise en œuvre de patterns, génèrent le code spécifique à chacune des variantes d'implémentation. Des exemples de règles ont été présentés et illustrent l'adaptation de celles-ci à la nature conflictuelle et compétitive des compromis d'implémentation.

PatternGen a été développé grâce à MétaGen, un outil qui permet la création de l'outillage d'un langage de modélisation partir d'un méta-modèle de ce langage. Plus précisément, son éditeur de diagrammes a été généré automatiquement à partir du méta-modèle du langage supporté par l'éditeur. L'utilisation de MétaGen pour implémenter PatternGen nous offre une grande flexibilité qui nous permet d'inclure des nouveaux patterns et d'ajouter des nouveaux compromis aux patterns déjà existants sans corrompre les modèles déjà existants.

La version de MétaGen actuelle dispose d'un éditeur graphique amélioré, doté de plusieurs mécanismes de simplification de modèles, d'une structure générique de modèles mieux définie et de nouveaux dialogues d'édition. Cette version a été développée grâce à l'intégration de différents cadres d'applications, réalisée par l'application de plusieurs patterns de conception.

MétaGen sert également d'outil support à la démarche de développement de l'équipe MétaFor. L'objectif de celle-ci est de permettre d'une part, l'expression des besoins des utilisateurs dans un langage proche de leur domaine et d'autre part, la prise en compte du contexte de développement et plus précisément des cadres d'applications. Cette prise en compte se traduit par des langages permettant la description de la politique d'usage de ces cadres.

Cette démarche a été illustrée par un exemple, qui nous a permis de donner un aperçu de l'intégration de PatternGen à la démarche. Cette intégration est importante car d'une part, les patterns simplifient la génération du code nécessaire à l'adaptation du cadre et d'autre part, les cadres offrent ce qu'il manque aux patterns : le développement complet d'une application.

Après le développement de PatternGen, nous nous sommes interrogés sur la généralité de notre approche : est-il possible de l'étendre à d'autres patterns de conception ?

Au cours de la réalisation de PatternGen, nous avons étudié les patterns décrits dans le catalogue de Gamma et al. [Gamma *et al.*, 1995] et en avons choisi quelques-uns pour enrichir le langage de modélisation supporté par l'éditeur de diagrammes. Notre choix n'a pas été aléatoire : les patterns choisis sont reconnus comme étant d'usage fréquent (page XV) et sont un échantillon de chacune des trois catégories : création (**Fabrique Abstraite**), structurelle (**Composite**, **Proxy**) et comportementale (**Observateur**, **Stratégie**). Nous pouvons donc penser que l'inclusion dans l'éditeur de diagrammes des autres patterns de ces catégories ne posera pas de problème particulier.

Toutefois, il est difficile de dire si notre approche est applicable à de nouveaux patterns ou à des patterns catalogués par d'autres auteurs. Cela dépendra essentiellement de la nature des compromis d'implémentation car en principe, la représentation des classes participantes est toujours possible (en raison de la nature du méta-modèle), ainsi que les modifications structurelles (en raison des règles). La description des compromis étant informelle, la réponse dépend de chaque pattern.

Une deuxième question, plus générale, porte sur notre approche. Le jeu en vaut-il la chandelle ? Notre réponse est oui, pour deux cas précis :

Premièrement, les patterns qui ont un grand nombre de compromis (et par conséquent, de variantes) d'implémentation, comme **Observateur** ou **Composite**, s'adaptent parfaitement à notre démarche et leur support s'avère très intéressant. En effet, ces patterns permettent une aide effective aux implémenteurs, par la génération d'une partie significative de leur code de mise en œuvre.

Deuxièmement, les patterns qui, à l'instar de **Proxy** et de **Stratégie**, offrent moins de variantes, mais qui définissent clairement des modifications structurelles à partir d'une situation initiale, s'adaptent aussi à notre démarche. En effet, ces modifications engendrent le déplacement de méthodes d'une classe à une autre et la création de méthodes d'adaptation, ce qui surcharge le développement. L'automatisation de ces patterns peut alors aider l'implémenteur.

La représentation dans PatternGen d'un pattern qui ne propose pas de changement structurel et dont les compromis d'implémentation n'engendrent pas de variantes d'implémentation a donc peu d'intérêt. Sa représentation est cependant utilise pour deux raisons : la génération des méthodes qui le composent et une meilleur documentation des modèles de conception.

I. Perspectives

L'implémentation de PatternGen a été une expérience très intéressante. Ce prototype nous a permis de mettre en œuvre plusieurs idées et de les évaluer. Dans cette optique, l'utilisation d'un langage de modélisation disposant de seulement trois constructeurs a été un choix adéquat. Le prochain pas nous paraît naturel : employer les connaissances acquises durant le développement de notre prototype dans la réalisation d'un logiciel, conforme aux standards actuels (e.g. UML [OMG, 1997b]).

La version actuelle de PatternGen peut évoluer de différentes manières et adopter de nouvelles fonctionnalités, comme la reconnaissance de patterns, la génération de code dans d'autres langages et l'aide à la conception.

D'une part, PatternGen ne reconnaît pas les patterns présents dans le code d'une application. Les idées énoncées par Kyle Brown [Brown, 1995], lors du développement de KT, constituent un point de départ à la reconnaissante de patterns en Smalltalk. Il reste à découvrir si ces idées peuvent être mises en œuvre par notre outil.

D'autre part, notre approche ne permet pas de retrouver les patterns après la génération de code, au contraire des travaux de Soukup et de Meijers [Meijers, 1996 ; Soukup, 1995]. Une solution possible à ce problème est d'utiliser le mécanisme de marquage de méthodes, disponible dans Smalltalk. Ce mécanisme permet de retrouver les méthodes correspondantes à un marquage, par exemple, toutes les méthodes implémentant une collection d'observateurs.

PatternGen dispose actuellement de fonctions permettant l'importation de code Java et la génération du code d'une application à partir de sa représentation dans l'éditeur de diagrammes. L'implémentation en Java des patterns présents dans notre outil et plus précisément, la généralisation des règles de mise en œuvre existantes constituera une expérience importante.

La mise en œuvre d'un système d'aide à la conception est sans doute intéressante, mais elle nous mène à nous interroger sur la problématique même de la conception. Est-il possible de suggérer au concepteur l'utilisation d'un pattern de conception ? Il est difficile de répondre à cette question car premièrement, peu de patterns caractérisent précisément les situations où leur application est applicable et deuxièmement, la détection de ces situations n'est pas toujours possible.

Nous pouvons également nous interroger sur les conditions dans lesquelles nous pourrons déduire les variantes d'implémentation les plus adéquates à chaque situation. D'un côté, la structure de classes d'une application nous apporte des connaissances permettant de reconnaître certaines caractéristiques propres aux instances à générer. Par exemple, pour un ensemble de données, la nature du sujet d'un pattern **Proxy** détermine la manière d'accéder aux données. En effet, la représentation d'une table de base de données, d'une image graphique ou d'un objet lointain imposera la spécialisation de classes différentes. D'un autre côté, d'autres caractéristiques dépendront essentiellement des techniques de programmation et pourront difficilement être déduites.

L'application de certains patterns engendre une modification structurelle d'une application (e.g. création de classes abstraites, déplacement de méthodes, etc.). Ces modifications peuvent être exprimées par un ensemble d'opérations de base (voir le *Refactoring Browser* de Opdyke [Opdyke, 1992]), qui simplifieraient une partie significative des règles de mise en œuvre.

Finalement, l'évolution de MétaGen permettra une amélioration des fonctionnalités actuelles de PatternGen. Plus spécifiquement, l'amélioration de l'éditeur graphique et la mise en œuvre d'un véritable éditeur de règles, pourraient certainement améliorer PatternGen.

Annexe A

Une approche constructive de recherche

Nunamaker, Chen et Purdin [Nunamaker *et al.*, 1991], voient le développement des systèmes comme une méthode de recherche et démontrent comment celui-ci peut s'adapter à un cadre traditionnel de recherche (problème, hypothèse, analyse et argument).

Ces auteurs insistent sur l'importance d'une approche méthodologique qui intègre quatre stratégies : (i) la construction d'une théorie ; (ii) le développement de systèmes ; (iii) l'observation et ; (iv) l'expérimentation. Ces quatre stratégies sont liées : la théorie suggère des démarches pour la résolution, supporte et est supportée par le développement d'un outil. Elle est l'initiatrice et le résultat de l'expérimentation et de l'observation [Nunamaker, 1992].

La construction d'une *théorie* débute par la sélection d'idées directrices, puis inclue la création des concepts. Ceux-ci permettent la construction de nouvelles architectures conceptuelles et de nouvelles méthodes (ou langages de modélisation). La théorie doit suggérer des hypothèses de recherche, les guider et les viabiliser.

L'*expérimentation* inclut d'autres méthodes de recherche comme la recherche d'action ou les expériences de laboratoire, de champ et de simulation. Elle doit essayer de valider la théorie ou de trouver des issues d'acceptation et de transfert de technologies. Ses résultats doivent être utilisés pour affiner la théorie et améliorer le système.

L'*observation* est fondée sur les méthodes de recherche discrète, comme l'étude de cas, l'étude de champ et les enquêtes. Elle est fréquemment utilisée lorsque l'on connaît peu le champ de recherche, de façon à fournir des hypothèses à tester ou à envisager de nouvelles recherches.

De son côté, le *développement de systèmes* peut être décomposé en cinq étapes : (i) conception ; (ii) construction d'une architecture ; (iii) prototypage ; (iv) développement d'un produit et (v) transfert technologique. Ces étapes correspondent grosso modo aux cinq phases suggérées par la plupart des méthodes de développement de systèmes, où le prototypage et le développement d'un produit sont confondus dans sa mise en œuvre et sont précédés par l'analyse et la conception.

Cette correspondance imparfaite semble affecter la discussion sur les phases proposées, où la clarté du discours et son application évidente à travers les quatre stratégies présentées sont diluées. Le contenu originel des phases reste cependant clair :

1. **Construire une architecture conceptuelle et choisir un problème de recherche.** L'architecture conceptuelle sert comme base à la recherche, en ce qui en concerne la direction et la description. L'étude du champ nécessaire à cette architecture fournit une meilleure

compréhension de la situation et est utilisée pour générer le problème de recherche. Le problème engendre, alors, la construction d'une théorie dans l'architecture conceptuelle.

2. **Développer une architecture de systèmes.** L'architecture pilote le processus de construction du système et démontre comment les divers composants s'associent et interagissent. Les contraintes de l'environnement, l'objectif de développement et la fonctionnalité du système résultant doivent être établis avec des besoins vérifiables. De nouvelles fonctionnalités et de meilleures interfaces utilisateurs seront plus fréquemment abordées, en dépit de la performance.

3. **Analyser et concevoir le système.** Ceci englobe : (i) la compréhension du domaine ; (ii) la création, l'évaluation et la synthèse d'alternatives ; et (iii) la collation de connaissances scientifiques et de techniques relevantes. La conception basée sur la théorie et le modèle (de nature abstraite) doit être autant qu'il se peut indépendante de l'environnement opérationnalisation spécifique choisi. Les structures de données dénotant les objets du discours doivent être choisies et spécifiées.

4. **Construire le système.** Un prototype peut en démontrer la faisabilité et prouver la théorie réglant ses fonctionnalités. Pour les essais requis dans le monde réel, le prototype (s'il est réussi) doit être développé comme un produit et utilisable dans les organisations cibles. Le processus de mise en œuvre réintroduit des informations utiles aux phases précédentes.

5. **Expérimenter, observer et évaluer le système.** Une fois qu'un système fonctionne, il peut être testé par rapport aux besoins et son efficacité par des études empiriques. Le résultat des essais engendre le développement du système, des concepts et des hypothèses.

Encore selon Nunamaker et al. [Nunamaker *et al.*, 1991], les projets de recherche faisant appel au développement de systèmes doivent être conformes à cinq critères : (i) l'objectif est d'étudier un phénomène important dans le champ des systèmes d'information par la construction d'un système ; (ii) les résultats constituent une contribution importante au domaine ; (iii) le système est testable par rapport aux objectifs et besoins établis ; (iv) le nouveau système doit fournir une meilleure solution aux problèmes de l'informatisation des systèmes d'information que les systèmes existants ; (v) l'expérience et les connaissances de conception acquises lors de la construction du système peuvent être généralisées pour leur utilisation future.

Annexe B

Outils de modélisation du processus

B.1 PROTEGE II

Le principe de PROTÉGÉ est de construire des modèles de tâches. Ces modèles de tâches sont en fait caractérisés par une description des concepts du domaine et de ceux de la méthode de résolution de problèmes, et par des liens de mise en correspondance (*mappings*) entre ces deux types de concepts. Ces modèles sont destinés à piloter la génération des éditeurs d'acquisition à partir de composants génériques d'interface[1]. Les éditeurs d'acquisition générés sont ensuite utilisés pour modéliser l'expertise dans les termes habituellement employés dans la communauté concernée. Les modèles conceptuels ainsi obtenus sont enfin transformés (quasi-) automatiquement en bases de connaissances.

La construction des modèles de tâches introduit les concepts du domaine comme éléments de base du processus de modélisation. C'est en partie en cela que ce type d'outils constitue une nouvelle alternative aux méthodes de construction des SBC du type « première génération ».

La démarche constitutive de l'approche PROTÉGÉ permet de mieux gérer la répartition des responsabilités entre les différents acteurs intervenant dans le processus de construction des SBC. Dans une première phase, les cogniticiens et les spécialistes du domaine interagissent au niveau méta du système pour créer les modèles de tâche. Par la suite, les experts utilisent les éditeurs d'acquisition pour modéliser leurs expertises. Ils instancient en fait les modèles de tâches adaptés au domaine de leur application qui permettront finalement de générer les bases de connaissances sur lesquelles travaillent les utilisateurs finaux.

B.2 ALF

Les objectifs du projet européen ALF ESPRIT (Accueil de Logiciel Futur) est de générer des environnements de développement centrés processus, qui offrent de l'assistance aux développeurs. Ce projet à développé un outil appelé ALF, construit autour d'un langage de modélisation de processus, MASP. Une description d'un processus en MASP comprend plusieurs modèles, qui décrivent : (i) les objets présents durant le processus, (ii) les états possibles d'un processus, (iii) les activités possibles et leur pré et post-conditions, (iv) l'ordonnancement d'exécutions des activités, (v) les réactions automatiques, sous la forme de règles de productions et (vi) les contraintes réifiées pendant l'exécution d'un processus.

[1] Ces composants sont par exemple des éditeurs élémentaires de listes, de tableaux, de graphes, etc...

Les modèles de processus sont exécutés par un interpréteur, mis en œuvre par un générateur de systèmes experts. L'interpréteur interagit avec une interface graphique et avec le système de gestion d'objets de PCTE. Ce dernier est aussi employé pour envoyer des messages aux outils de développement (en réalité, des objets de PCTE).

B.3 ALMA

L'objectif du projet ALMA [Lamsweerde *et al.*, 1988] de l'Université de Namur est de créer des environnements de développement composés par un référentiel de données et ses outils de mise à jour, d'interrogation et de gestion de versions des objets et par des outils d'édition basés sur une syntaxe spécifique, permettant d'éditer les données gardées dans le référentiel.

Le modèle relationnel du référentiel de données spécifique à un environnement de développement est défini à partir d'un modèle du cycle de vie, i.e. un modèle Entité/Association décrivant le cycle de vie supporté par l'environnement. La syntaxe sur laquelle est basé un outil d'édition est décrite en BNF. Cette syntaxe est traduite automatiquement en des entrées de LEX et YACC. En suite un « code mixer » génère le code correspondant à l'outil.

ALMA a été développée en langage C, sous Ultrix et utilise le SGBD relationnel Ingres pour implémenter les référentiels de données.

B.4 Articulator

Le projet DSF [Mi et Scacchi, 1991] de l'Université de Southtern California a développé Articulator, un environnement qui utilise des techniques de représentation connaissances pour modeler le processus de développement.

Dans Articulator, un processus de développement est modélisé par des agents, qui exécutent des tâches et en employant des outils ou des systèmes qui utilise ou produit des ressources. Comme les agents, les outils et les tâches sont aussi des ressources, ils peuvent aussi être produits ou utilisés par d'autres agents ou d'autres tâches.

Son modèle du processus combine l'héritage des modèles objets avec des mécanismes d'inférence en chaînage arrière. Les agents communiquent l'état des tâches et la disponibilité des ressources les un aux autres par des requêtes et des messages. Ces requêtes peuvent occasionner une récupération directe ou déductive d'informations de l'exécution du processus, déclencher l'exécution symbolique d'un nouveau processus ou exécuter à nouveaux des processus. Des processus concurrents peuvent être exécutés et les agents peuvent interagir à travers ces processus.

Articulator garde des informations sur les processus exécutés, lors des transitions. Cette information est utilisée pour tracer l'historique d'exécution d'activités prévues, et pour faciliter la planification (rescheduling) incrémentale du processus et des nouvelles prévisions lorsque des changements dynamiques ou imprévus du processus arrivent.

Articulator a été développé en KnowledgeCraft, un environnement de génie de connaissances.

B.5 SMART

SMART [Garg *et al.*, 1994] est environnement de modelage, d'analyse et d'exécution du processus de développement. Il combine le langage de modélisation de processus de Articulator avec le moteur d'exécution de processus SynerVision et l'environnement de programmation en équipe Matisse de Hewlett-Packard.

Le modèle de processus crée par Articulator est gardé dans le référentiel de Matisse et peut ainsi être exécuté par le moteur SynerVision. Ceci envoie des informations sur les tâches qui doivent être exécutées à BMS, un service de gestion de messages, qui les retransmet à des outils de développement. De manière analogue, Articulator reçoit des messages concernant l'état d'exécution des tâches, ce qui lui permet d'améliorer son modèle du processus.

B.6 Arcadia

Le projet Arcadia [Taylor, 1988] est un ensemble de projets de développement et de recherche, cordonnés et conduits par un consortium. L'objectif commun des projets est de fournir des outils personnalisables, permettant de composer un environnements de développement logiciel. En raison de cette caractéristique, Arcadia aurait pu être classé comme un canevas d'environnement. Cependant, comme son architecture est centrée sur la capacité d'exécuter des modèles du processus, il est aussi un outil de modélisation du processus.

L'adaptabilité de l'environnement est donc le point central de l'architecture d'Arcadia, qui dispose de différents outils de définition d'un modèle du processus de développement. Grâce à une interface commune, le modèle du processus interagit avec différents systèmes, comme Endeavors [Bolcer et Taylor, 1996] ou ProcessWall [Heimbigner, 1992]. Le premier est un environnement d'exécution de processus, permettant la définition et l'exécution d'une application de type workflow. Le deuxième un serveur persistant d'états de processus, proposant des opérations de définition et de manipulation de la structure de ces états.

Arcadia dispose de deux langages de modélisation du processus : APPL/A [Sutton *et al.*, 1988 ; Sutton *et al.*, 1990] et TeamWare [Young, 1994] qui s'adressent à différents aspects du processus. APPL/A est une extension de Ada qui inclue la notion de persistance, grâce aux relations un système de gestion d'objets (Triton). Teamware est un environnement (et un langage) de modélisation de processus, dont la caractéristique principale est celle d'être à la porté d'utilisateurs non techniciens.

Plusieurs systèmes de gestion d'interfaces sont disponibles dans Arcadia, comme Chiron-2 [Keller *et al.*, 1990 ; Taylor *et al.*, 1996] et GLAD [Snider, 1991]. Le premier permet la définition d'une interface utilisateur comme une hiérarchie de composants, liés entre eux par des connecteurs (i.e. routeurs de messages) et qui respectent certaines « règles de style ». Ces définitions sont utilisées pour personnaliser des cadres d'applications de mise en œuvre d'interfaces, disponibles en Java et en C++. GLAD est un paquetage Ada qui propose plusieurs facilités de construction de graphes orientés, comme l'équilibrage d'un graphe ou la réduction de croisements entre arcs.

Arcadia dispose aussi de différents systèmes de gestion d'objets, comme PLEIADES [Tarr et Clarke, 1993] Triton [Heimbigner, 1990] et PGraphite [Wolf *et al.*, 1988], qui peuvent être combinés grâce à une structure sous-jacente commune. Les outils du projet ont été développés en Ada, C++, Lisp et Java.

Annexe C

Exemples de cadres d'applications

C.1 HotDraw

HotDraw [Johnson, 1992] est un cadre d'applications implémenté en Smalltalk 80, et conçu pour assister la réalisation d'éditeurs graphiques à deux dimensions. L'éditeur standard fourni par HotDraw, présenté dans la Figure 74, possède deux parties visibles : une fenêtre de dessin (à droite) et une palette d'outils (à gauche) contenant une succession d'outils de création et de manipulation de figures.

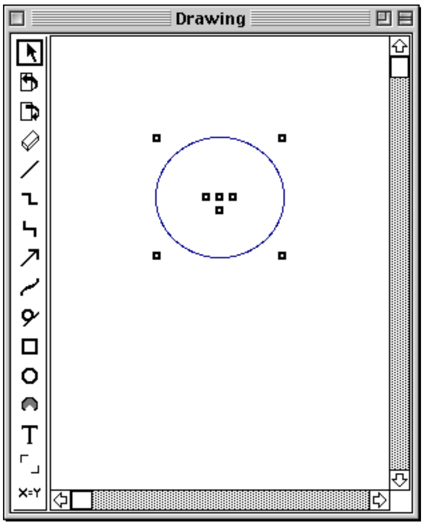

Figure 74 - Editeur standard de HotDraw

HotDraw a été implémenté au-dessus d'un autre cadre d'applications, MVC [Krasner et Pope, 1988] (Figure 75). Toutefois, dans HotDraw, la triade MVC a été étendue par un quatrième élément, 'Drawing' (dessin), qui contient les figures éditées et partage le rôle de modèle avec une autre classe, 'DrawingEditor'. 'Drawing' a une sous-classe très importante, 'ConstraintDrawing' qui contient la description d'un système de contraintes qui permet d'entretenir des relations entre les figures grâce à des contraintes entre variables de coordonnés graphiques. Les deux autres composants du MVC sont les classes 'DrawingView' et 'DrawingController'.

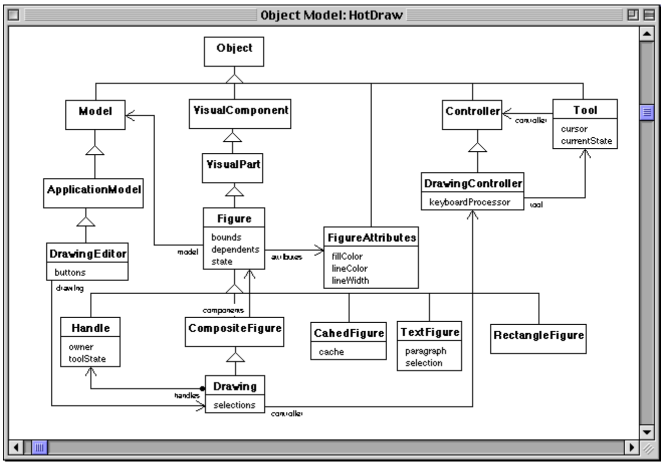

Figure 75 - Diagramme OMT de HotDraw

La conception de HotDraw a fait appel à un ensemble de patterns de conception [Beck et Johnson, 1994], ce qui simplifie sa compréhension. Le pattern Composite, par exemple, est employé pour mettre en œuvre la relation parent/enfant existante entre les classes 'CompositeFigure' et 'Figure'.

Pour compléter cette description succincte de HotDraw, il faut citer deux autres classes importantes, 'Figure' et 'Command'. 'Figure' est la classe abstraite de toutes les figures (e. g. ligne, rectangle, ellipse). Elle est complétée par la définition de 'ContainerFigure', qui permet la création des figures composées par un ensemble de figures et par les poignées (handles) qui sont des petits carrés, attachés aux figures, permettant soit l'exécution de commandes spéciales, soit l'établissement de contraintes entre figures. 'Command' est la classe abstraite de toutes les commandes (e. g. connexion, contrainte, sélection multiple). Les commandes sont appelées par des poignées ou par des outils, placés sur la palette à outils. Quand un outil est sélectionné, il crée un lecteur ('Reader') et une commande. Le lecteur lit l'état courant, et communique avec la commande qui réalise les modifications requises dans le dessin

C.1.1 Editeur PERT

Le premier exemple, fourni avec Hotdraw, est un éditeur de graphe PERT (voir Figure 76). Il montre l'utilisation du système de contraintes de HotDraw. Un graphe PERT est composé par un ensemble d'événements, chacun représentant une tâche devant être exécutée à une date déterminée et prenant un temps d'exécution précis. Les contraintes à satisfaire, sont que les dates de début des événements doivent être ultérieures à celles des tâches qui les précédent dans le graphe, augmentées de leur durée. Ainsi, quand deux événements sont connectés, une contrainte est créée, pour empêcher une inconsistance dans la construction du graphe.

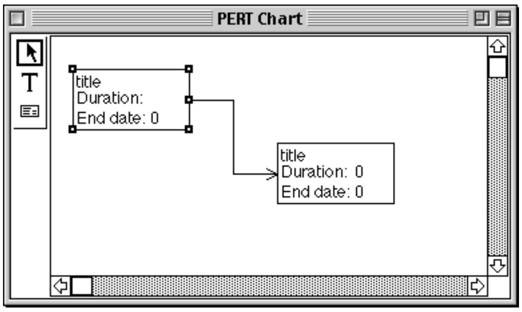

Figure 76 - Editeur PERT

Pour réaliser cet exemple, deux classes ont été créées, 'PERTChart' et 'PERTEvent', sous-classes de 'DrawingEditor' et de 'CompositeFigure', respectivement.

Les trois outils présents dans la palette sont définis par la méthode 'defaultTools' de la classe 'PERTChart', qui surcharge la méthode originelle de l'éditeur. De cette manière, l'utilisation de l'éditeur est limitée à la sélection de figures, à la création d'une zone de texte et à la création d'événements. Seulement le troisième outil a dû être implémenté (les deux autres existaient déjà), il est créé par la méthode 'creationTool', de la classe 'PERTEvent'.

Les figures qui représentent les événements sont composées par cinq champs textuels et un rectangle. Cette composition est faite lors de la création d'un événement, par la méthode 'initialize', qui a été surchargée par 'PERTEvent'.

Finalement, une poignée permettant d'effectuer des connexions a été ajoutée à la figure représentant les événements, grâce à la surcharge de la méthode 'handles'. Cette poignée est composée par une commande de connexion et la méthode qui sera exécutée lorsque la connexion est réalisée. La contrainte entre deux événements est établie par cette méthode.

C.1.2 Editeur de réseaux

L'éditeur de réseaux est un exemple de création d'un éditeur permettant l'animation de dessins de graphes. Cette animation est déterminée par l'utilisateur en spécifiant les distances à respecter entre les nœuds.

L'éditeur est divisé en deux parties (Figure 77). La partie supérieure est une surface de dessin contenant un réseau de rectangles interconnectés par des lignes. Les rectangles représentent les nœuds du réseau, et les lignes, la distance entre deux nœuds. La partie inférieure de l'éditeur est divisée en trois secteurs : deux listes et une vue d'édition de texte. Chacune des deux listes contient la désignation de tous les nœuds du réseau et dispose d'un menu contextuel qui permet l'ajout ou la suppression d'un nœud. Quand un nœud est ajouté à une de ces listes, il apparaît automatiquement, en même temps, sur la surface de dessin et dans la deuxième liste. De plus, il est connecté par de nouvelles lignes aux autres nœuds. Ensuite, l'utilisateur peut associer une distance à chacune de ces lignes, en sélectionnant, dans la vue d'édition de texte, un nœud source suivi d'une destination.

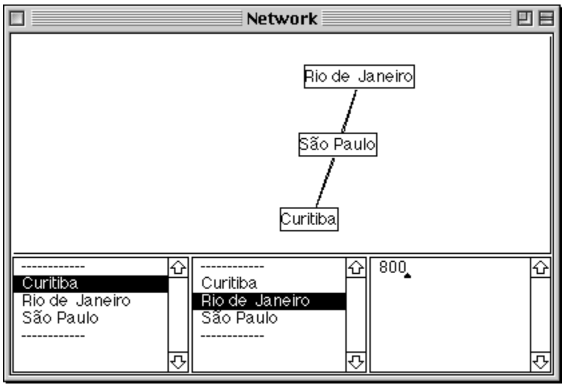

Figure 77 - Editeur de réseaux

La dynamique de l'animation de cet exemple est réalisée par la surcharge de la méthode 'step' de la classe 'Drawing', qui est appelée périodiquement par l'éditeur. L'algorithme utilisé dans cette méthode se divise en deux parties. En premier, il calcule la vitesse de chaque nœud à partir de la force que chacun a sur les autres. Pour cela, cette classe garde une liste de tous les nœuds. Ensuite, elle envoie un message à chacun des nœuds pour les forcer à bouger. Quand les vitesses deviennent trop petites, l'algorithme s'arrête.

La classe 'DrawingEditor' est aussi spécialisée, pour deux raisons : en premier, pour travailler avec la nouvelle classe 'Drawing', et en deuxième, pour surcharger la méthode 'initialize', qui crée les deux listes et la vue d'édition de texte. La troisième classe créée dans cet exemple s'appelle 'NetwokNode', une spécialisation de 'CachedFigure', et surcharge la méthode 'translateBy' pour effectuer l'animation. Chaque nœud a un nom et une force, calculée quand la méthode 'step' est exécutée. Cette classe surcharge aussi la méthode 'fillCache', utilisée pour afficher un rectangle de taille variable contenant à l'intérieur le nom du nœud.

C.2 Accounts

Accounts [Keefer, 1994] est un cadre d'applications pour informatiser un système de comptabilité. Il a été construit autour de deux sortes d'objets : les comptes et les transactions. Un compte garde la trace des transactions qui lui sont envoyées et est capable de calculer la valeur d'un attribut depuis une certaine date. Une transaction est un enregistrement d'un événement significatif, concernant des données d'une certaine catégorie. Une facture, par exemple, est une transaction qui contient des informations d'en-tête (nom du vendeur, date, etc.) et aussi une liste de produits contenant le nom, la quantité et le prix des articles vendus.

Trois formes de compte sont prévues dans Accounts : « simple », « composite » et « agrégé ». Les comptes simples représentent les comptes élémentaires, qui servent de base à un système de comptabilité. Les comptes simples stockent les transactions. Les comptes composites et agrégés groupent tous le deux plusieurs comptes. Les comptes composites groupent des comptes relativement homogènes, i.e., ayant un ensemble commun d'attributs. Les comptes agrégés rassemblent des composants hétérogènes. Quand une transaction est envoyée à un compte composite, ce compte l'envoie à un de ses comptes. Les comptes agrégés la transforment en d'autres transactions, qui sont envoyés aux composants concernés.

Les comptes simples possèdent des attributs internes qui donnent accès aux valeurs des transactions qui leur ont été envoyées. Tous les comptes peuvent posséder des attributs constants. D'autres sortes d'attributs, les attributs calculés, utilisent des expressions arithmétiques formées par d'autres attributs pour produire des nouvelles valeurs.

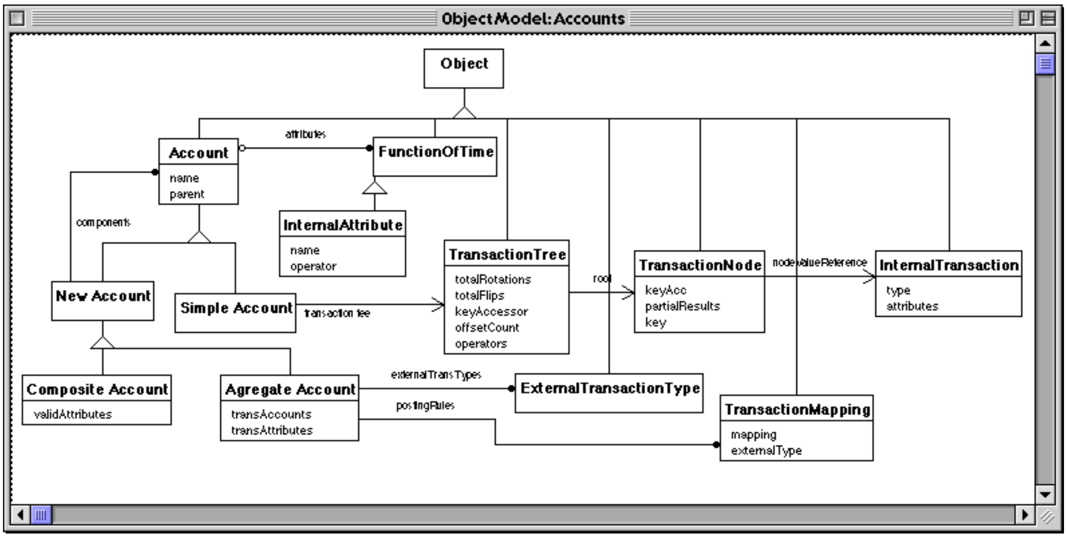

Figure 78 - Diagramme OMT de Accounts

Pour simplifier la génération d'un système de comptabilité, Accounts dispose d'un générateur d'applications, i.e. un éditeur graphique permettant de définir les comptes et capable de générer le code Smalltalk relatif à cette définition.

La Figure 78 est un diagramme OMT simplifié de la conception de Accounts. Les comptes composites et agrégés se composent de plusieurs comptes simples qui sont les seuls à garder une trace des transactions qui leur sont envoyées. Les comptes agrégés gardent une description (mapping) de la conversion de transactions externes vers un ensemble de transactions internes. Les comptes peuvent posséder plusieurs sortes d'attributs.

Dans la version actuelle de Accounts, les nouveaux comptes ne sont pas des spécialisations des trois comptes de base, mais des instances de ces classes. Pour permettre la spécialisation entre comptes, un mécanisme de « templates » entre instances a été mis en œuvre. Cette solution particulière a été adoptée pour séparer le concept de spécialisation entre comptes et celui intrinsèque à Smalltalk.

Deux solutions sont envisageables pour créer un système comptable sans passer par le générateur d'applications. La première solution consiste à créer des instances des comptes, attributs, mappings, etc. et de les associer de manière à composer le compte souhaité. La deuxième solution, et celle que nous préférons, fait appel au pattern de conception **Fabrique Abstraite** (décrit dans [Gamma *et al.*, 1995]). Elle consiste à créer une sous-classe du compte souhaité et à créer une méthode d'initialisation qui se charge de construire le compte.

Annexe D

Projets de recherche analogues à MétaFor

D.1 Méthodes à objets

Les premières méthodes de développement adoptaient des différents langages de modélisation pour matérialiser la conclusion des étapes du processus de développement. Plus précisément, l'expression des besoins des acteurs de ce processus était synthétisée en deux modèles. Le premier était le résultat de l'analyse des besoins et représentait un système d'information en termes d'entrées et de sorties. Le deuxième représentait comment ce même système serait implémenté, en termes de modules.

L'avènement des méthodes à objets, comme OMT [Rumbaugh *et al.*, 1991], OOA/OOD [Coad et Yourdon, 1990 ; Coad et Yourdon, 1991], Booch [Booch, 1991], Shlaer-Mellor [Shlaer et Mellor, 1992], Classe-Relation [Desfray, 1992] ou OOSE (Objectory) [Jacobson *et al.*, 1992] est aussi celui de l'unification des langages de conception et d'analyse. En effet, le paradigme des objets sert aussi bien à représenter le domaine du problème que celui de l'implémentation. Toutes ces méthodes s'orientent actuellement vers l'adoption d'un seul langage de modélisation (dit universel), UML. Si, en principe, l'objectif de UML est de seulement unifier le langage de modélisation, il nous paraît improbable que le processus de développement soutenu par chaque méthode reste indemne après l'adoption d'un langage plus expressif.

Les scénarii (ou use-cases), introduits par la méthode OOSE et présents dans l'UML, décrivent l'interaction entre un utilisateur et son application. Ils sont un pas important vers une expression des besoins des utilisateurs plus compréhensible par ces derniers.

D'une certaine manière, la séparation entre l'analyse et la conception est similaire à celle entre les points de vue de l'utilisateur et du développeur, préconisée par le projet MétaFor. Cependant, les langages de modélisation ont des objectifs bien distincts. D'une part, le langage utilisateur sert à décrire des concepts d'un métier, en employant des constructeurs extraits de ce métier et non des constructeurs informatiques. Il est donc beaucoup moins générique et moins compréhensible par les informaticiens qu'un modèle d'analyse. D'autre part, le langage implémenteur utilise plusieurs constructeurs et limite les relations entre eux. Il est moins expressif que les langages de conception car il ne permet que la description d'applications pour un seul environnement applicatif.

D.2 FACE

L'objectif du projet FACE (Framework Adaptive Composition Environment) [Meijler, 1997] est de simplifier la configuration d'un cadre d'applications lors du développement d'une application. Pour cela, il préconise une approche de développement visuel par composants, enrichie par des schémas d'objets. Ces schémas décrivent les structures possibles d'objets et leur coopération.

Face propose un atelier de développement à deux niveaux : la description des composants génériques (extraits d'un cadre d'applications) et l'utilisation des composants pour générer une application. Ces deux niveaux utilisent un même langage de programmation, enrichi par une interface de composition. En principe, la démarche est indépendante du langage de programmation utilisé, mais des exemples ont été réalisés en C++ et Self.

Il existe deux familles de composants (class-components) : les composants conceptuels et ceux d'implémentation. Les premiers ressemblent aux classes des langages à objets, dans le sens où ils peuvent être spécialisés. Les composants d'implémentation sont utilisés pour personnaliser ou instancier une partie d'un cadre d'applications déjà existante, et ne peuvent pas être spécialisés. Les composants se lient par des relations explicites, qui sont aussi divisées en familles. Certaines relations ont été inspirées par des patterns de conception, comme par exemple la relation de dépendance, inspirée du pattern Observateur ou Dépendance.

Les relations et les composants sont définies dans d'un méta-schéma, par de composants de méta-classe (le principal constructeur du langage de méta-modélisation employé). Un langage sous-jacent (le langage de programmation) permet l'association d'une sémantique à ce méta-schéma.

La démarche préconisée par FACE ressemble à celle de MétaFor, en ce qui concerne l'utilisation d'un langage de modélisation, dont les constructeurs ont été extraits d'un cadre d'applications, pour expliciter la réutilisation pendant le processus de développement. Comme le langage de méta-modélisation de FACE est spécifique aux composants, il nous paraît raisonnable de penser qu'il est plus adéquat que PIR3 pour construire des langages de configuration de cadres d'applications.

Toutefois, une différence importante des deux projets est liée à la génération de code. FACE défend une interprétation des composants en temps d'exécution de l'application, au détriment de la génération de code. Bien que cette dernière ne soit pas complètement abandonnée, elle est réservée à la génération de la version finale d'une application.

D.3 Programmers' Apprentice

L'objectif du projet Programmers' Apprentice [Rich et Waters, 1990] est d'offrir aux « ingénieurs logiciels » une assistance intelligente, qui remplace celle des autres acteurs qui partagent la tâche de développement, comme les programmeurs, documentaristes, etc. Il rejette l'idée de remplacement de l'ingénieur de systèmes, soutenue par les ateliers de génie logiciel.

Le concept clé de ce projet est que « les programmeurs rarement pensent en termes d'éléments de base (i.e. instructions de bas niveau), mais dans la plupart du temps, en termes de combinaisons d'éléments fréquemment utilisées ». Cette combinaison est appelée cliché. Un cliché est une unité de connaissance bien définie sur un constructeur de programmation, qui peut être abstrait, représenté et réutilisé.

L'outil central de ce projet est KBEmacs, qui propose deux modes d'édition de programmes : texte et plan. Le premier présente le code source du programme, et le deuxième, un résumé de ce même programme sous la forme d'une combinaison de clichés. La traduction entre le premier mode et le deuxième est assurée par un outil, appelé Analyzer. Le processus inverse est réalisé par un autre outil, appelé coder. Le mode plan permet une interaction avec une bibliothèque de clichés.

Pour être intégré à cette bibliothèque, un cliché doit tout d'abord être identifié, à travers une analyse qui montre qu'un tel constructeur de programmation est récurrent dans plusieurs implémentation. Le cliché est ensuite abstrait et raffiné, de manière à représenter effectivement toutes les implémentation étudiées. Finalement, le cliché est décrit dans un le langage Plan Calculus, ce qui simplifie sa réutilisation par la suite.

L'activité de constitution de la bibliothèque de clichés peut être vue comme une application de l'analyse du domaine. Bien que concernée surtout par les efforts d'implémentation, cette activité est similaire à celle de la construction des méta-modèles dans MétaFor. Dans ce sens, le paradigme des clichés nous paraît indiqué comme une solution à la complexité de la génération de code dans notre processus de développement. Il reste à savoir comment ces clichés, fortement basés sur les langages procéduraux, peuvent s'adapter aux langages à objets.

D.4 PhiNix

PhiNix [Barstow, 1985] est un système de génération transformationnelle de programmes d'interaction avec des périphériques physiques, à travers de collections ordonnées (streams) de données et de contrôles. Plus spécifiquement, dans le domaine des pilotes de contrôle d'outils de « oil well logging tools ». Le langage de programmation cible est StreamMachine.

Un des objectifs du projet PhiNix des laboratoires de recherche de Schlumberger-Doll est de démontrer que la programmation automatique est plus effective s'il y a une séparation entre les connaissances du domaine et les connaissances de programmation. Pour cela, PhiNix dispose d'un langage composé d'un large éventail de constructeurs, extraits des domaines de l'applications et du langage cible, appelé PhiLang.

PhiNix capture des connaissances spécialisées dans le domaine pétrolifère, et les organise sous la forme de contraintes pour former les spécifications informelles. Celles-ci sont ensuite transformées, par un processus de raffinement, qui additionne des détails spécifiques à ce domaine, en des expressions qui décrivent des valeurs d'entrée et sortie. Le processus de développement est achevé par l'application de règles spécifiques, traitant des problèmes de mise en œuvre, qui génèrent le code source de l'application.

Les transformations dans PhiNix sont pilotées manuellement ou par des scripts. Une des évolutions prévues de PhiNix prévoit l'inclusion d'un langage stratégique les contrôler. Une autre prévision est l'enrichissement de PhiLang par des constructeurs permettant la description de contraintes de temps réels relatives aux systèmes embarqués. Ce projet démontre un besoin de connaître plus sur le savoir-faire de l'organisation et la structuration des connaissances du domaine, dans le but des les réutiliser.

D.5 L'approche Meta Model Transformation (MMT)

MMT [Oei et Hemmen, 1994] est une approche transformationnelle de méta-modèles pour harmoniser les langages de modélisation. L'objectif de cette approche est de fournir des moyens puissants pour résoudre les inconvénients causés par l'actuelle jungle de méthodes [Avison et Fitzgerald, 1988].

Bien que les objectifs de MMT soient différents de ceux du projet MétaFor, les deux projets se ressemblent en deux aspects : l'utilisation d'un langage de méta-modélisation, et la mise en relation entre différents langages de modélisation à travers de leurs constructeurs. Un aspect intéressante de cette approche est la définition formelle des transformations, réalisée par la combinaison de trois mises en relation. Celles-ci correspondent chacune à une paire, une opération et son inverse, et sont indépendantes du langage de méta-modélisation :

1. Spécialisation et généralisation : un langage peut devenir plus spécifique par l'introduction de constructeurs plus spécialisés. En autres termes, un méta-modèle y est une spécialisation d'un méta-modèle x si les constructeurs de x sont un sous-ensemble de ceux de y, et si tous les autres constructeurs de y sont définis en termes de constructeurs déjà existants de x.

2. Restriction et libéralisation : un langage peut devenir plus restreint par l'introduction de contraintes. Ainsi, un méta-modèle y est une restriction d'un méta-modèle x si l'ensemble des populations possibles de y est un sous-ensemble de l'ensemble des populations possibles de x.

3. Dégénération et génération : la dégénération est la seule opération qui diminue l'expressivité (la capacité de décrire des différents domaines) d'un langage. Un méta-modèle y est appelé une dégénération d'un méta-modèle x, s'il existe un sous-ensemble des constructeurs de x qui ne peut pas être peuplé en y.

Ces opérations sont exprimées grâce au langage formel LISA-D et s'appliquent sur des méta-modèles décrits par le langage de méta-modélisation ML, basé sur le langage NIAM. MMT sert de base formelle pour la comparaison, la sélection, l'intégration et l'évolution des langages de modélisation.

D.6 Kate

Le projet Kate [Fickas et Nagarajan, 1988] est un exemple intéressant d'utilisation de techniques d'acquisition de connaissances dans le processus de développement logiciel. Ce projet s'attaque aux limitations des techniques actuelles spécifications de logiciels. Dans ces techniques, la plupart de la connaissance dans la construction de spécifications se concentrent sur une personne, l'expert.

Le projet propose d'automatiser les connaissances de cet expert, de manière à acquérir des spécifications du problème qui sera informatisé. L'énumération des spécifications est vue comme un processus interactif de résolution de problèmes, basé sur la critique. Un assistant, Kate, pose des questions et génère des critiques à partir d'une liste initiale de spécifications. Les sessions interactives entre le client et l'assistante sont, en effet, un processus de raffinement qui crée des spécifications valides, non-ambiguës et consistantes.

Comme « les clients on une vague notion de ce qu'ils veulent et seulement une vague idée de ce qu'il est possible », un « bon analyste a l'expertise de faire ressortir les points clés d'un problème ». L'expertise de Kate se fonde sur un ensemble de protocoles collectés sur plusieurs sessions d'analyse. Kate utilise ces protocoles pour obtenir des spécifications. Pour être effectif, toutefois, Kate fait aussi appel à d'autres connaissances, comme celle du domaine d'application, celles d'implémentation et celles de l'environnement de production.

Une contribution plus importante du projet est le concept d'acquisition de connaissances par critique. Une base initiale de connaissances du domaine peut être étendue par des questions hypothétiques, sous la forme de critiques. En faisant ces critiques indépendantes au domaine, le processus peut être appliqué à n'importe quel domaine.

D.7 DRACO

Le projet DRACO [Neighbors, 1984 ; Neighbors, 1992] propose une approche de développements d'applications par l'organisation de composants logiciels réutilisables. Son objectif est de d'introduire, dans sa base de connaissances, l'expériences de deux nouveaux acteurs du développement logiciel : l'« analyste du domaine » et le « concepteur du domaine ».

Le premier examine les besoins de plusieurs applications d'un domaine spécifique et défini une collection d'objets et d'opérations dans ce domaine. Le deuxième détermine différentes implémentations pour ces objets et ces opérations dans les termes d'un domaine déjà connu par DRACO.

Grâce à ces définitions, un analyste de systèmes peut développer la spécification d'un nouveau système. Ces spécifications seront raffinées en code exécutable par DRACO. Le raffinage est piloté par un concepteur de systèmes, qui peut choisir entre les différentes implémentations qui ont été spécifiées au préalable par le concepteur du domaine.

L'outil DRACO fourni plusieurs outils, comme des mécanismes permettant la création de « parsers » de langages, un générateur de code et un système de transformation. Le rôle de ce dernier est de traduire des constructeurs de modélisation d'un langage du domaine en un objet déjà connu par DRACO.

Annexe E

Geds

Pour permettre aux concepteurs logiciels de représenter la description de leurs éditeurs, nous avons choisi un langage de modélisation déjà existant, OPRR, employé par l'outil de méta-modélisation MétaEdit [Smolander *et al.*, 1991]. La transformation d'un modèle OPRR en une spécialisation de HotDraw passe par une étape intermédiaire, un modèle représentant l'utilisation de ce cadre d'applications.

Ces modèles se construisent grâce à un deuxième langage de modélisation, appelé Edito, conçu spécialement pour utiliser HotDraw. Deux bases de règles complètent l'environnement, la première transforme les modèles décrits en OPRR en des modèles décrits par Edito, et la deuxième génère le code Smalltalk nécessaire pour sous-classer HotDraw et réaliser l'éditeur décrit préalablement. Cette expérience a été réalisée en collaboration avec Bruno Lesueur, qui a conçu et implémenté le langage Edito et la base de règles de génération de code.

L'environnement consiste en un éditeur OPRR permettant la description d'éditeurs graphiques et d'un ensemble de règles de production. Ces règles sont applicables à tout modèle OPRR et sont capables de générer le code d'implémentation Smalltalk de l'éditeur décrit.

Le processus de génération de code est réalisé en deux étapes. La première transforme les modèles OPRR en des modèles Edito, qui décrivent comment l'usage souhaité de HotDraw. En principe, les modèles Edito ne sont pas sujets à des modifications, ils servent seulement comme un point de vérification et validation. La deuxième étape transforme les modèles Edito en code, plus précisément, en des sous-classes de HotDraw.

Nous présenterons par la suite les divers composants de l'environnement. En premier, le langage OPRR et un modèle d'un éditeur DFD en OPRR. En deuxième, le langage Edito et le modèle Edito correspondant au modèle OPRR. Finalement, nous présenterons les principes de la transformation en deux étapes.

E.1 Le langage OPRR

OPRR est le langage de méta-modélisation utilisé par le méta-outil MetaEdit [Smolander *et al.*, 1991]. Comme PIR3, OPRR est une variante du langage E/A. Toutefois, il existe une différence de base entre ces deux langages : PIR3 s'intéresse à la modélisation des constructeurs des langages de modélisation, alors que OPRR s'intéresse à modéliser l'outillage de ces langages. Ainsi, OPRR permet la spécification de détails spécifiques, comme les symboles de représentation d'un éditeur graphique, ce qui n'est pas possible avec PIR3. OPRR est composé par quatre constructeurs :

Un Objet est une « chose » qui a une existence propre. Il est identifié par un nom et possède un symbole de représentation. Il est représenté par un carré.

Une Relation est une association entre deux ou plusieurs objets. Elle peut être orientée et doit définir le type de ligne. Elle est représentée par un losange.

Un Rôle est un lien entre un objet et une relation. Chaque objet possède un rôle dans une relation. Ce rôle détermine la nature de la participation d'un objet dans une relation. Il est représenté par un cercle.

Une Propriété est une caractéristique descriptive ou qualificative des autres trois concepts. Les propriétés sont représentées par des ellipses.

La Figure 79 présente le méta-modèle PIR3 de OPRR. Il contient quatre méta-individus, qui représentent les quatre constructeurs de OPPR ('Object', 'Property', 'Relationship' et 'Role'). Objets, relations et rôles peuvent avoir de propriétés. Ceci est représenté par trois méta-relations 'has'. Les cardinalités de ces méta-relations assurent que ces méta-individus peuvent avoir entre aucune à plusieurs propriétés (0N) et que les propriétés peuvent être partagées (0N). Les relations possèdent exactement deux rôles (22), qui ne peuvent pas être partagés avec d'autres relations (11) et les lient à un ou plusieurs objets (par rôle). Les deux rôles d'une relation définiront quels sont ses objets d'entrée et de sortie.

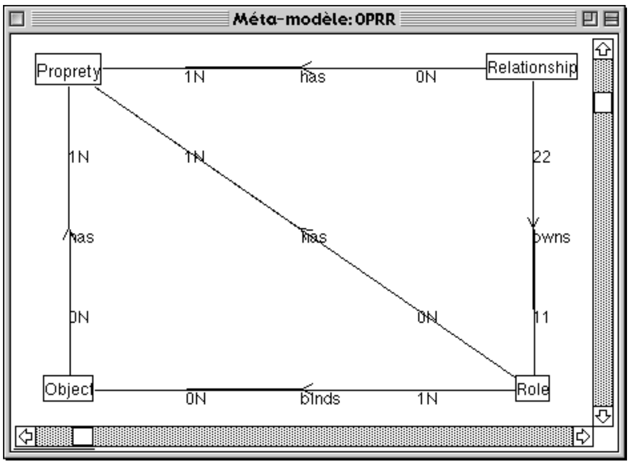

Figure 79 – Méta-modèle PIR3 de OPRR

Dans la Figure 80, un éditeur de diagrammes de flots de données est représenté en OPRR. Les trois principaux constructeurs (Processus, Dépôt de données et Acteur externe) sont directement représentés comme des Objets : 'Process', 'Store' et 'External'. Les flots de données sont représentés par deux Relations : 'Flows' et 'FFP'. La première représente les flots de données qui partent de 'Processus', 'Dépôt' ou 'Acteur' vers 'Processus'. Le deuxième représente les flots qui partent de 'Processus' vers 'Dépôt' et 'Acteur'. Cette division est nécessaire pour interdire les flots de données entre dépôts et acteurs et inversement. Les deux relations partagent une même propriété, 'nom', qui est aussi leur identifiant. Les trois objets partagent aussi une propriété, 'code'.

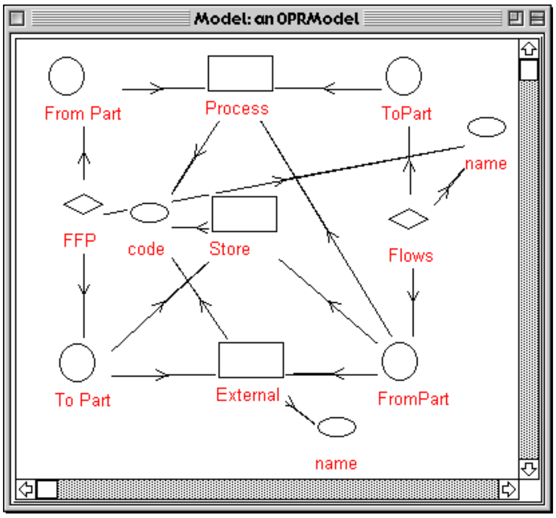

Figure 80 - Modèle OPRR de DFD

E.2 Le langage Edito

L’objectif du langage Edito est de permettre la description d’un environnement de modélisation, composé par un éditeur de réseaux sémantiques, un modèle sous-jacent et des boîtes de dialogue. Le rôle du modèle sous-jacent est de stocker les données représentées dans l’éditeur de manière à séparer les données d’un domaine de ça représentation graphique. En analogie avec le cadre d’applications MVC, l’éditeur joue le rôle de vue, tandis que le modèle sous-jacent joue le rôle de modèle. Edito se compose de treize constructeurs, qui se divisent en quatre parties.

La première partie s’occupe de la configuration de HotDraw. Elle est composée par trois constructeurs : ‘Editeur’, ‘Arc Graphique’ et ‘Nœud Graphique’. ‘Editeur’ est certainement le constructeur principal du langage. Il permet de représenter l’éditeur graphique qui sera généré, (i.e. la sous-classe de ‘DrawingEditor’). Les arcs et nœuds graphiques servent à représenter les arcs et nœuds d’un réseau sémantique et donneront origine à des sous-classes de ‘ArrowFigure’ et ‘CachedFigure’, respectivement. Ces trois constructeurs possèdent une propriété, ‘name’, qui servira comme leur identifiant. Les nœuds graphiques possèdent une deuxième propriété, ‘icon’, qui décrit l’image qui lui sera associée pendant l’édition. Les arcs graphiques possèdent aussi une deuxième propriété, appelée ‘type’, qui établie le type de ligne utilisée (courbe ou droite).

La deuxième partie permet d’ajouter des menus contextuels à l’éditeur, aux arcs et aux nœuds graphiques. Un menu est composé par plusieurs labels, et chaque label désigne une méthode Smalltalk qui sera déclenchée lors que ce label sera choisi.

La troisième partie sert à représenter le modèle sous-jacent à l’éditeur. Quatre constructeurs la composent : ‘Modèle’, ‘Nœud’, ‘Arc’ et ‘Rubrique’. Lors de l’implémentation, les trois premiers deviendront des spécialisations de la classe ‘Model’. Les arcs et les nœuds peuvent posséder des propriétés, les rubriques, qui seront transformées en des variables d’instance des classes générées.

La quatrième partie permet la représentation des boîtes de dialogues d’édition des nœuds et arcs du modèle sous-jacent. Trois méta-individus la composent : ‘Dialogue’, ‘Saisie’ et ‘Bouton’. Chaque champ de saisie correspond à une rubrique, i.e. une variable d’instance des classes générées. Les champs de saisie représentent, au même temps, un libellé et un champ d’édition.

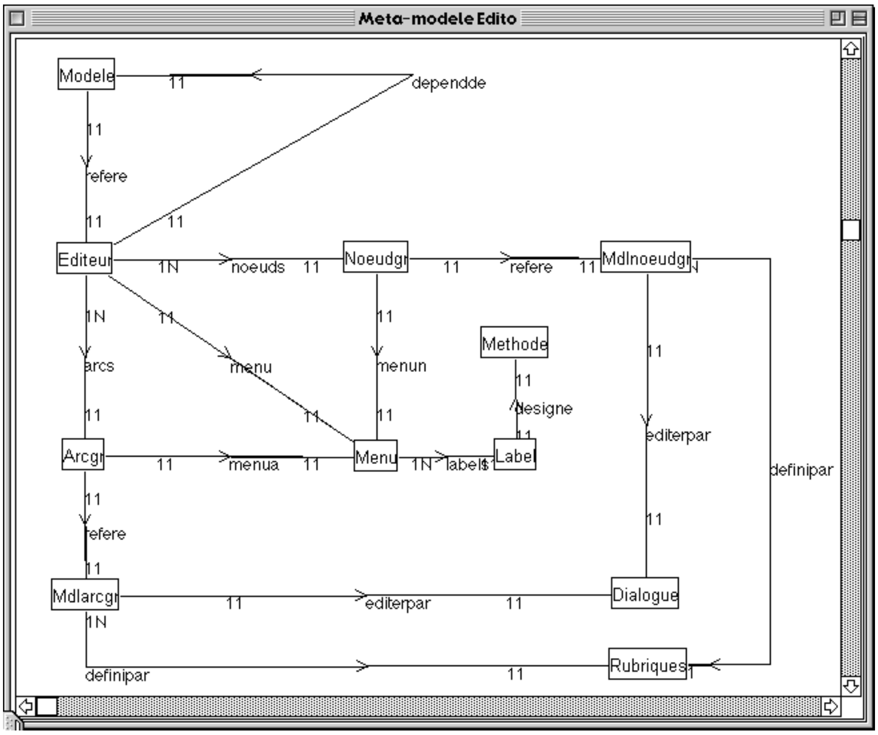

Figure 81 - Méta-modèle PIR3 de Edito

Le plan principal du méta-modèle PIR3 du langage Edito est présenté par la Figure 81 (un deuxième plan, contenant les constructeurs des dialogues d'édition, est accessible à partir du menu contextuel du méta-individu 'Dialogue'). Le point central de ce méta-modèle est le méta-individu 'Editeur', qui possède des méta-relations avec 'Modèle', 'Menu', 'Nœud' et 'Arc Graphique'. Les cardinalités de ces méta-relations assurent qu'une occurrence d'éditeur doit, précisément, se référer à un modèle et avoir un menu (11), et qu'elle peut avoir plusieurs nœuds et arcs (1N). Au même temps, elles assurent que les occurrences de ces quatre méta-individus auront des relations avec une seule occurrence d'éditeur (11).

Le même principe d'unicité qui lie un éditeur à son modèle (11 et 11) est présent dans les méta-relations qui lient les arcs et nœuds graphiques à leurs modèles et à leur menu. Aussi, dans méta-relation entre 'Arc' et 'Nœud' avec leur dialogue d'édition. 'Arc' et 'Nœud' peuvent posséder une ou plusieurs rubriques (1N), mais une rubrique ne peut pas être partagée (11).

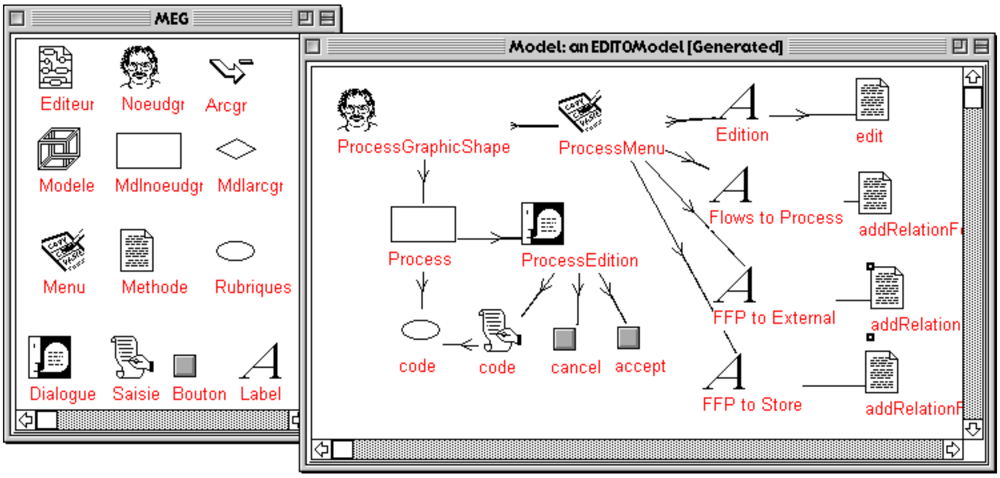

Figure 82 - Modèle partiel Edito d'un éditeur de DFD

Un modèle exprimé par le langage Edito est présenté dans la Figure 82. La représentation graphique d'Edito est exhibée à gauche de la figure. Ce modèle a été généré à partir du modèle OPPR d'un éditeur de DFD, présenté ultérieurement. Plus spécifiquement, les concepts représentés ici ont été générés à partir de l'objet (OPRR) processus. La transformation de modèles OPRR en des modèles Edito sera décrite dans la section suivante.

L'individu central de ce modèle est processus, une occurrence du méta-individu nœud. Un processus est représenté par un nœud graphique ('ProcessGraphicShape'), possède une rubrique (code) et est édité par une boîte de dialogue ('ProcessEditon'). La boîte de dialogues est composée par deux boutons ('Cancel' et 'Accept') et un champ de saisie ('code') qui modifiera la seule rubrique de processus. Un menu contextuel ('ProcessMenu') est attribué au nœud graphique. Il contient quatre labels, déclenchant quatre méthodes Smalltalk : la première ('edit') ouvre la boîte de dialogues et les autres permet de créer des flots de données vers d'autres processus, acteurs externes et dépôts de données.

E.3 De OPPR à Edito

L'objectif de cette première base de règles est de transformer tout modèle représenté en OPRR en des modèles Edito. Elle se divise en deux parties. La première crée le modèle cible et traduit les objets et les relations du modèle OPPR en des nœuds et des arcs du modèle Edito. La deuxième partie ajoute des nouvelles informations au modèle Edito cible, comme des boites de dialogue et des menus, à partir des informations déjà présentes dans ce modèle. Nous énumérons par la suite les principes de cette transformation :

Traduction

1. Création d'un modèle Edito (le modèle cible) ;
2. Création d'une occurrence d'éditeur graphique, de son modèle, de son menu et des relations entre eux. Leurs noms seront déduits à partir du nom du modèle source ;
3. Chaque occurrence d'objet dans le modèle source donne origine à un doublé, un nœud graphique et sont modèle, à une relation entre ces deux individus et à une relation entre l'éditeur (créé dans 2) et le nœud graphique ;

4. Chaque occurrence de relation donne origine à un arc (modèle sous-jacent), de même nom ;

5. Les propriétés d'un objet ou d'une relation, deviennent des rubriques du nœud ou de la relation, généré(e) à partir du premier ;

6. Les relations orientées donnent origine à des arcs graphiques, dont les nœuds graphiques d'origine et de destin seront déterminés par les individus liés aux deux rôles de la relation. Comme plusieurs objets peuvent être liés à un même rôle, plusieurs arcs graphiques peuvent être générés (le nombre total est déterminé par la multiplication du nombre d'objets en entrée par le nombre d'objets en sortie). Toutefois, les arcs graphiques seront toujours binaires ;

7. Les relations non-orientées donnent aussi origine à des arcs graphiques, mais en plus grand nombre. En effet, un arc graphique sera généré à partir de chaque pair d'objets liés à des rôles différents d'une même relation ;

8. Ajout d'un label au menu de l'éditeur, pour chaque occurrence d'objet dans le modèle source. Le nom du label sera « Créer … » et le nom de l'objet. Une méthode de création sera associée au label ;

Expansion

9. Création d'un dialogue d'édition pour chaque arc et chaque nœud générés dans le modèle cible. Chaque dialogue aura deux boutons, 'Accepter' et 'Annuler' ;

10. Ajout d'un menu pour chaque occurrence de nœud et arc et d'un label « Editer » à ce menu. Le label fait référence à une méthode d'édition. Chaque menu sera ensuite lié aux différents nœuds ou arcs graphiques qui leur font référence (un menu pourra être partagé par plusieurs arcs graphiques) ;

11. Pour chaque nœud graphique, origine d'un arc graphique, ajout, à son menu, d'un label « Créer une relation vers … » et le nom du nœud de destin. Une méthode sera associée à ce label ;

12. Ajout de champs de saisie aux boîtes de dialogue, à partir des différentes rubriques générées.

E.4 De Edito à Smalltalk et HotDraw

Cette deuxième base de règles a trois responsabilités : configurer HotDraw de manière à générer l'éditeur souhaité, créer le modèle sous-jacent correspondant à cet éditeur et créer les boîtes de dialogue d'édition des composants du modèle sous-jacent. Les principes de la génération sont les suivants :

1. Création d'une sous-classe de 'DrawingEditor' à partir de chaque occurrence d'éditeur ;

2. Création d'une sous-classe de Model à partir de chaque occurrence d'arc ou de nœud. Les variables d'instance de cette classe sont les rubriques du nœud ou de l'arc ;

3. Ajout d'une méthode « menu » aux classes générées dans (1 et 2). Cette méthode rend un menu, dont les options sont déduites à partir des labels liés aux menus, dans le modèle source ;

4. Création des sous-classes de 'ArrowFigure' et 'ImageFigure' à partir des occurrences d'arc et nœud graphique ;

5. Création des méthodes d'accès aux variables d'instance des classes créées dans (2) ;

6. Création d'une méthode 'edit' pour chaque classe créée dans (2). Cette méthode déclenche l'ouverture des boîtes d'édition ;

7. Chaque boîte de dialogue devient une sous-classe de 'SimpleDialog' à laquelle une méthode de classe, 'windowSpec' est ajoutée. Cette méthode s'occupe de la construction de la boîte de dialogue ;

E.5 L'éditeur DFD généré

Le résultat de l'application des bases de règles au modèle OPRR d'un éditeur de DFD (Figure 80), est une collection de classes Smalltalk, qui adaptent HotDraw pour créer des éditeurs de diagrammes de flots de données. Plus précisément, chacun des trois objets OPRR a été réifié par une sous-classe de Model. Chaque classe est représentée par une sous-classe de 'DrawingFigure' et édité par une sous-classe de 'SimpleDialog'. Les deux relations OPPR ont aussi été réifiées par des sous-classes de 'Model' et ont donné origine à un dialogue d'édition. Cependant, elles seront représentées par plusieurs sous-classes de 'ArrowFigure', qui reflètent les liens possibles, établis par des rôles OPRR. Finalement, le modèle OPPR a donné origine à la classe 'DataFlowEditor', sous-classe de 'DrawingEditor'.

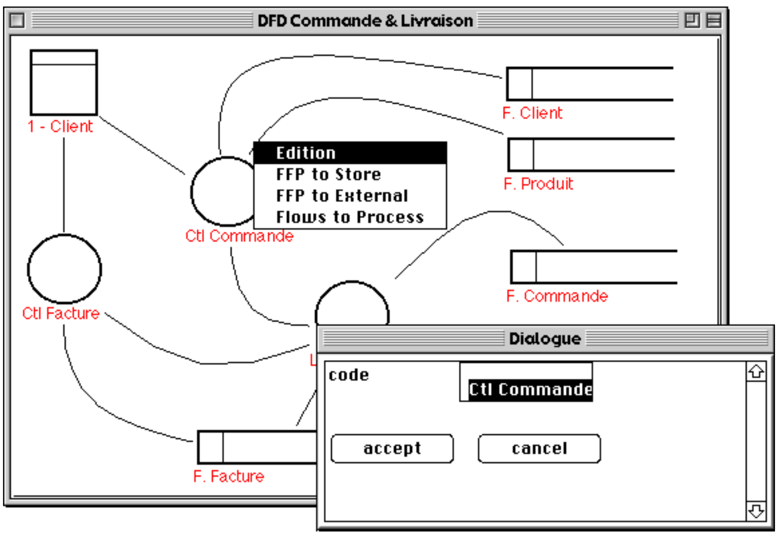

Figure 83 - L'éditeur DFD généré

La Figure 83 est un exemple d'utilisation de l'éditeur DFD généré. Dans cette figure, il est possible d'observer le menu contextuel et la boîte d'édition d'un processus. Grâce aux menus, le concepteur peut créer des acteurs, des processus et des dépôts de données et ensuite, les lier par des flots de données. Ils permettent aussi l'ouverture des boîtes d'édition de ces éléments. Au même temps qu'un diagramme est dessiné, un modèle sous-jacent est instancié.

E.6 Critique

L'environnement décrit dans cette section (l'éditeur OPRR couplé avec des bases de règles) s'est avéré capable de générer des éditeurs graphiques pour des divers langages de modélisation de type réseau sémantique. Il sert aussi comme base d'études de OPRR en tant que langage de méta-modélisation et de base de comparaison avec PIR3.

Cette expérience nous a permis de mettre en œuvre un langage de modélisation de schémas d'usage d'un cadre d'applications, Edito, selon les objectifs établis par Nicolas Revault [Revault, 1996]. Le langage Edito et les bases de règles de génération simplifient effectivement, la configuration de HotDraw, par la réutilisation des connaissances de mise en œuvre d'éditeurs graphiques.

Cependant, Edito a présenté plusieurs faiblesses. D'une part, il ne permet pas la création de vrais schémas d'utilisation de HotDraw car les éditeurs de réseaux sémantiques ne constituent pas le seul usage de ce cadre d'applications. D'autre part, les schémas d'utilisation doivent être, en principe, indépendants du langage d'implémentation du cadre d'applications [Revault, 1996] (page 78). Ceci n'est pas le cas d'Edito, car plusieurs constructeurs de ce langage (comme méthode ou bouton) sont susceptibles de contenir des morceaux de code Smalltalk.

Bibliographie

[Aaen *et al.*, 1992] Aaen, I., A. Siltanen, C. Sorensen et V.-P. Tahvanainen, *A tale of two countries: CASE experiences and expectations*, in *The impact of Computer Supported Technologies on Information Systems Development*. 1992, North-Holland: Amsterdam.

[Aarsten *et al.*, 1995] Aarsten, A., G. Elia et G. Menga, *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing*, in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, Editors. 1995, Addison-Wesley. pp. 576.

[Agabra *et al.*, 1997] Agabra, J., I. Alvarez et P. Brezillon, Contextual knowledge based systems: A study and design in oenology, in *First International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-97)*, Rio de Janeiro, 1997.

[Alderson, 1991] Alderson, A., Meta-CASE Technology, in *European Symposium in Software Development Environments and CASE Technology Proceedings*, Königswinter, Germany, 1991.

[Alexander, 1979] Alexander, C., *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Alexander *et al.*, 1975] Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson et S. Angel, *The Oregon Experiment*. Oxford University Press, New York, 1975.

[Alexander *et al.*, 1977] Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King et S. Angel, *A Pattern Language*. Oxford University Press, New York, 1977.

[Apple, 1989] Apple, *Macintosh Programmers Workshop Pascal 3.0 Reference*, Cupertino, California, 1989.

[Apple, 1997] Apple, *The Application Kit Reference*. MacOS X - Objectif C Framework Reference. Apple Computer, Cupertino, CA, 1997.

[Arango, 1993] Arango, G., *Domain Analysis Methods*, in *Software Reusability*, W. Shaefer, R. Pietro-Diaz, and M. Matsumoto, Editors. 1993, Ellis Horwood.

[Atzeni et Torlone, 1993] Atzeni, P. et R. Torlone, *A Metamodel Approach for the Management of Multiple Models and the Translation of Schemes*. 1993: pp. 349-362.

[Avison et Fitzgerald, 1988] Avison, D.E. et G. Fitzgerald, *Information Systems Development: Methodologies, Techniques and Tools*. Blavkwell Scientific Publications, Oxford, UK, 1988.

[Barstow, 1985] Barstow, D.R., *Domain-specific automatic programming* in *IEEE Transactions on Sofware Engineering*. 1985. pp. 1321-1336.

[Beck, 1996] Beck, K., *Smalltalk Best Practice Patterns*. Vol. Volume 1 : Coding, 1996.

[Beck et Johnson, 1994] Beck, K. et R.E. Johnson, Patterns Generate Architectures, in *Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, 1994.

[Beer *et al.*, 1987] Beer, S., RayWelland et I. Sommerville, Software Design in an IPSE, in *1st European Software Engineering Conference, ESEC'87*, Strasbourg, 1987.

[Benali *et al.*, 1989] Benali, K., N. Boudjlida, F. Charoy, J.-C. Derniame, C. Godart, P. Griffiths, V. Gruhn, P. Jamart, D. Oldfield et F. Oquendo, Presentation of the ALF project, in *International Conference on System Development Environments and Factories*, Berlin, 1989.

[Bertolazzi *et al.*, 1995] Bertolazzi, P., G.D. Battista et G. Liotta, *Parametric Graph Drawing* in *IEEE Transactions on Software Engineering*. 1995. pp. 662-673.

[Bézivin, 1995] Bézivin, J., *Technologie objet et ingénierie des besoins : une réconciliation nécessaire*. L'Objet : Logiciel, Bases de Données, Réseaux, 1995. **1**(1).

[Blain et Beugniet, 1996] Blain, G. et F. Beugniet, Technologie objet et méta-modélisation, in *RPO'96*, , 1996.

[Boehm, 1973] Boehm, B.W., *Software and its impact: a quantitative assessment* in *Datamation*. 1973.

[Boehm, 1981] Boehm, B.W., *Software Engineering Economics*. Advances in computing science and technology series, ed. E.C. NJ. Printice Hall, NJ, 1981.

[Boehm, 1990] Boehm, B.W., Software Engineering environments in the United States (Plenary Talk), in *Fourth ACM SIGSOFT Symposium on Software Development Environments*, , 1990.

[Bohem, 1986] Bohem, B., *A Spiral Model of Software Development and Enhancement* in *ACM Software Engineering Notes*. 1986.

[Bolcer et Taylor, 1996] Bolcer, G.A. et R.N. Taylor, Endeavors: A Process System Integration Infrastructure, in *Interpational Conference on Software Process*, Brighton, UK, 1996.

[Boloix *et al.*, 1992] Boloix, G., P.G. Sorenson et J.P. Tremblay, *On transformations using a metasystem approach to software development*. Software Engineering, 1992. **7**: pp. 425-437.

[Bommel *et al.*, 1991] Bommel, P.v., A.H.M.t. Hofsted et T.P.v.d. Weide, *Semantics and verification of object-role models*. Information Systems, 1991. **5**(16): pp. 471-495.

[Bonne et Razavi, 1993] Bonne, J.M. et R. Razavi, ACKPRO-GLA (conception générale et documentation technique), Rapport de projet DESS GLA, Université de Paris 6, Paris 1993

[Booch, 1991] Booch, G., *Object Oriented Design with Applications*. The Benjamin Cummings Publishing Company Inc., 1991.

[Borland, 1994] Borland, *A Technical Comparison of Borland ObjectWindows 2.0 and Microsoft MFC 2.5*, Scotts Valley, CA, 1994.

[Borne et Revault, 1999] Borne, I. et N. Revault, *Comparaison d'outils de mise en oeuvre de design patterns*. L'Objet, 1999. **5**(1).

[Bosch, 1995] Bosch, J., Delegating Compiler Objects - An Object-Oriented Approach to Crafting Compilers, in *Compiler Construction '96*, , 1995.

[Bosch, 1996] Bosch, J., Langage Support for Design Patterns, in *TOOLS Europe'96*, , 1996.

[Brinkkemper, 1990] Brinkkemper, S., *Formalisation of Information Systems Modelling*, , University of Nijmegen, Amsterdam, 1990.

[Brinkkemper *et al.*, 1996] Brinkkemper, S., K. Lyytinen et R.J. Welke, *Method Engineering - Principles of method construction and tool support*. Working Conference on Method Engineering, ed. S. Brinkkemper, K. Lyytinen, and R.J. Welke. Chapman & Hall, Atlanta, USA, 1996.

[Brooks, 1982] Brooks, F.P., *The Mitical Man-Month: Essays on Software Engineering*. Addison-Wesley, Reading, Mass, 1982.

[Brooks, 1987] Brooks, F.P., *No silver Bullet* in *IEEE Computer*. 1987. pp. 10-19.

[Brown, 1995] Brown, K., *Design reverse-engineering and automated design patterns detection in Smalltalk*. Technical Journal, 1995.

[Bubenko, 1988] Bubenko, J.A., A Method Engineering Approach to Information Systems Development, in *IFIP WG8.1 Working Conference on Information Systems Development Process*, , 1988.

[Budinsky *et al.*, 1996] Budinsky, F.J., M.A. Finnie, J.M. Vlissides et P.S. Yu, *Automatic code generation from design patterns*. IBM Systems Journal, 1996. **35**(2).

[Buschmann *et al.*, 1996] Buschmann, F., R. Meunier, H. Rohnert, P. Somerlad et M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd., 1996.

[CADWARE, 1988] CADWARE, SYLVA Foundry User Manual, Release 6.0, CADWARE Soft., 1988

[Cagan, 1990] Cagan, M.R., *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools*. Hewlett-Packard Journal, 1990. **41**(3): pp. 36-47.

[Campbell, 1988] Campbell, I., *Portabl Common Tool Environment*. Computer Standards and Interfaces, ed. N. Holland. Vol. 8, 1988.

[Chaplais *et al.*, 1993] Chaplais, C., B. Gaudinat et C. Tora, L'Atelier de Méthodes: Un Modele Objet Support à l'Expression de Méthodes, in *Sixth International Conference on Software Engineering ans its Applications Proceeding*, Paris, 1993.

[Chen, 1976] Chen, P.P.-S., *The Entity-Relationship Model - Toward a Unified View of Data*. ACM Transactions on Database Systems, 1976. **1**(1): pp. 9-36.

[Christiansen, 1987] Christiansen, D., *On good designers*. IEEE Spectrum, 1987. **24**(5): pp. 25.

[Coad, 1992] Coad, P., *Object-oriented patterns*. Communications of the ACM, 1992. **35**(9): pp. 152-159.

[Coad *et al.*, 1995] Coad, P., D. North et M. May, *Object Models, Strategies, Patterns & Applications*, ed. E. Cliffs. Yourdon Press, 1995.

[Coad et Yourdon, 1990] Coad, P. et E. Yourdon, *Object-Oriented Analysis*. Computing Series, ed. Y. Press. Prentice Hall, 1990.

[Coad et Yourdon, 1991] Coad, P. et E. Yourdon, *Object-Oriented Desing*. Computing Series, ed. Y. Press. Prentice Hall, 1991.

[Coplien, 1992] Coplien, J.O., *Advanced C++ Programming Style and Idioms*. Addison-Wesley, Reading, MA, 1992.

[Coplien, 1995] Coplien, J.O., *A Generative Development-Process Pattern Language*, in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, Editors. 1995, Addison Wesley: Reading, MA. pp. 183-237.

[Coplien, 1996] Coplien, J.O., *Software Patterns*. SIGS Management Briefings. SIGS Books & Multimedia, 1996.

[Coplien et Schmidt, 1995] Coplien, J.O. et D.C. Schmidt, eds. *Pattern Languages of Program Design*. . Addison-Wesley, Reading, MA, 1995.

[Cunnigham, 1995] Cunnigham, W., *The CHECKS Pattern Language of Information Integrity*, in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schmidt, Editors. 1995, Addison Wesley: Reading, MA.

[Curtis *et al.*, 1988] Curtis, B., H. Krasner et N. Iscoe, *A Field Study of the Software Design Process for Large Systems*. Communications of the ACM, 1988. **31**(1): pp. 1268-1287.

[DeMarco, 1982] DeMarco, T., *Controlling Software Projects*. Yourdon Press, 1982.

[Desfray, 1992] Desfray, P., *Ingénierie du logiciel avec Classe-Relation*. Masson, Paris, 1992.

[Deutsch, 1989] Deutsch, L.P., *Design Reuse and Frameworks in the Smalltalk 80 Porgramming System*, in *Software Reusability*, T.J. Biggerstaff and A.J. Perlis, Editors. 1989, ACM Press.

[Edelson, 1992] Edelson, D.R., Smart pointers: They're smart, but they're not pointers, in *USENIX C++*, Portland, OR, 1992.

[Eden *et al.*, 1997] Eden, A.H., A. Yehudai et J. Gil, Patterns of the Agenda, in *LSDF97: Workshop in conjunction with ECOOP'97*, , 1997.

[EIA, 1990] EIA, CDI Organization and Procedure Manual, cdif-doc-n2-v1 edition, Electronics Industry Association, 1990

[Emmerich *et al.*, 1991] Emmerich, W., G. Junkermann et W. Schäfer, MERLIN: knowledege based process modelling, in *First European Workshop on Software Process Modelling*, Milan, 1991.

[Fickas et Nagarajan, 1988] Fickas, S. et P. Nagarajan, *Critiquing Software Specifications* in *IEEE Software*. 1988. pp. 37-47.

[Florijn *et al.*, 1997] Florijn, G., M. Meijers et P.v. Winsen, *Tool support for object-oriented patterns*. 1997.

[Fowler, 1997] Fowler, M., *Analysis Patterns - Reusable Object Models*. 5th ed. Object Technology, ed. Booch, Jacobson, and Rumbaugh. Addison-Wesley, 1997.

[Fugetta, 1993] Fugetta, A., *A classification of CASE technology* in *IEEE Computer*. 1993. pp. 25-38.

[Gabriel, 1994] Gabriel, R.P., *The Failure of Pattern Languages*. Journal of Object-Oriented Programming, 1994: pp. 84-88.

[Gamma *et al.*, 1993] Gamma, E., R. Helm, R. Johnson et J. Vlissides, Design Patterns: Abstraction an Reuse of Object-Oriented Design, in *ECOOP'93*, , 1993.

[Gamma *et al.*, 1995] Gamma, E., R. Helm, R. Johnson et J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.

[Gane et Sarson, 1979] Gane, C. et T. Sarson, *Structured systems analysis - tool and techniques*. Prentice-Hall, 1979.

[Garg *et al.*, 1994] Garg, P.K., P.M.T. Pham, W. Scacchi et G. Yhunquest, The SMART approach for software process engineering, in *Sixtennth Conference on Software Engineering*, Sorrento, Italy, 1994.

[Goldberg et Robson, 1983] Goldberg, A. et D. Robson, *Smalltalk-80: The Language and Its Implementation*, ed. Addison-Wesley, 1983.

[Goldfine et Konig, 1988] Goldfine, A. et P. Konig, A technical overview of the information resource dictionary system, U. S. Department of Commerce, National Bureau of Standarsds, Institute for Computer Science and Technology, Gaithersburg, MD 20899 1988

[Guerrieri, 1994] Guerrieri, E., *Case study: Digital's application generator* in *IEEE Software*. 1994. pp. 95-6.

[Hainaut *et al.*, 1992] Hainaut, J.-L., M. Cadelli, B. Decuyper et O. Marchand, TRAMIS: a Transformation-Based Database CASE Tool, in *Fifth International Conference on Software Engineering ans its Applications*, Toulouse, 1992.

[Hall et Weedon, 1993] Hall, P. et R. Weedon, *Object-Oriented Module Interconnection Languages*, in *Advances in Foftware Reuse, Slected Papers from the 2nd International Workshop on Software Reusability*. 1993, IEEE Computer Society Press: Lucca, Italy.

[Harmsem *et al.*, 1994] Harmsem, F., S. Brinkkemper et H. Oei, *Situational Method Engineering for Information System Project Approaches*, in *Methods and Associated Tools for the Information System*, A.A. Verrijn-Stuart and T.W. Olle, Editors. 1994, North Holland.

[Harmsen et Brinkkemper, 1995] Harmsen, F. et S. Brinkkemper, Design and implementation of a method base management system for a situational CASE environment, in *2nd Asian-Pacific Software Engineering Conference*, Los Alamitos, CA, 1995.

[Harmsen et Saeki, 1996] Harmsen, F. et M. Saeki, Comparison of four Method Engineering languages, in *Working Conference on Method Engineering*, Atlanta, USA, 1996.

[Heimbigner, 1990] Heimbigner, D., Triton reference manual, Technical Report, University of Colorado, Department of Computer Science, Boulder, CO 1990

[Heimbigner, 1992] Heimbigner, D., The Process Wall: A Process State Server Approach to Process Programming, in *5th ACM SIGSOFT Symposium on Software Development Environments*, , 1992.

[Helm *et al.*, 1990] Helm, E., I. Holland et D. Gangopadhyay, Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, in *OOPSLA'90*, Ottawa, Canada, 1990.

[Heroux et Desevedavy, 1992] Heroux, S. et P. Desevedavy, Validation Assitée par Ordinateur - Interprétation d'un MCT Merise, Rapport de Projet Annuel, Université de Paris VI - DESS GLA, Paris 1992

[Hofstede, 1993] Hofstede, A.H.M.t., *Information modelling in data intensice domains*, , University of Nijmegen, Amsterdam, the Netherlands, 1993.

[Hofstede *et al.*, 1996] Hofstede, A.H.M.t., H. Osterle et T.F. Verhoef, *Meta-CASE: Is the game worth the candle?* Information Systems Journal, 1996(6): pp. 41-68.

[IBM, 1990] IBM, Repository Manager/MVS. General Information Version 1 Release 2, Technical report, 1990

[Jackson, 1983] Jackson, M.A., *System Development.* Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Jacobson *et al.*, 1992] Jacobson, I., M. Christerson, P. Jonsson et G. Overgaard, *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[Jaczynski, 1998] Jaczynski, M., *Modèle et plate-forme à objets pour l'indexation des cas par situations comportementales : application à l'assistance à la navigation sur le web*, Thèse de Doctorat, Université de Nice-Sophia Antipolis, , 1998.

[Johnson, 1992] Johnson, R.E., Documenting Frameworks using Patterns, in *OOPSLA'92*, Vancouver BC, 1992.

[Johnson et Foote, 1988] Johnson, R.E. et B. Foote, *Designing Reusable Classes.* Journal of Object-Oriented Programming, 1988.

[Johnson *et al.*, 1992] Johnson, R.E., C. McConnell et J.M. Lake, The RTL system: A framework for code optimization., in *International Workshop on Code Generation*, Dagstuhl, Germany, 1992.

[Jordan et Davis, 1991] Jordan, A.K. et A.M. Davis, Requirements Engineering Metamodel: An Integrayed View of Requirements, in *Fifteenth COMPSAC*, , 1991.

[Karrer et Scacchi, 1993] Karrer, A.S. et W. Scacchi, *Meta-Environments for Software Production.* Intl. Journal of Software Engineering and Knowledge Engineering, 1993. **3**(1): pp. 139-162.

[Karrer et Scacchi, 1995] Karrer, A.S. et W. Scacchi, *Meta-Environments for Software Production*, in *Software Engineering and Knowledge Engineering: Trends for the Next Decade*, W.D.H.U. Pittsburgh), Editor. 1995, World Scientific Press,.

[Katayama, 1989] Katayama, T., A hierarchical and functional software process description and its enaction, in *Eleventh International Conference on Sotware Engineering*, , 1989.

[Keefer, 1994] Keefer, P.D., *An Object Oriented Framework for Accounting Systems*, Master of Science Thesis, University of Ilinois, Urbana, 1994.

[Keller *et al.*, 1990] Keller, K., M. Cameron, R.N. Taylor et D.B. Troup, Chiron-1: a user interface developent system tailored to software environments, Technical Report, University of California - Department of Information and Computer Science, Irvine 1990

[Kelly, 1994] Kelly, S., *A Matrix Editor for MetaCASE Environment.* Information and Software Technology, 1994. **36**(6): pp. 361-371.

[Kelly, 1995] Kelly, S., *Towards Better MetaCASE and CAMES Environments*, Licenciate Thesis, University of Jyräskylä, Jyvaskyla, Finland, 1995.

[Kelly, 1998] Kelly, S., *Private talk* . 1998.

[Kelly *et al.*, 1996] Kelly, S., K. Lyytinen et M. Rossi, MetaEdit+ A Fully configurable multi-user and multi-tool CASE and CAME environment, in *CAiSE'96*, Heraklion, Crete, Greece, 1996.

[Kelly et Smolander, 1994] Kelly, S. et K. Smolander, *Evolution and issues in meta-CASE.* Information and Software Technology, 1994.

[Kent, 1981] Kent, W., *Data and Reality: Basic assumptions in data processing reconsidered.* 3rd ed. North-Holland, 1981.

[Kim et Benner, 1995] Kim, J.J. et K.M. Benner, A Design Patterns Experience: Lessons Learned and Tool Support, in *Position Paper, Workshop on Patterns, ECOOP'95*, Aarhus, Denmark, 1995.

[Kogut et Nilson, 1994] Kogut, P. et R. Nilson, Domain Enginnering Methods and Tools Handbook, Software for Adaptable, Reliable Systems (STARS), 1994

[Krämer et Prechelt, 1996] Krämer, C. et L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in *Working Conference on Reverse Engineering (WCRE'96)*, , 1996.

[Krasner et Pope, 1988] Krasner, G.E. et S.T. Pope, *A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80.* Journal of Object Oriented Programming, 1988. **1**(3): pp. 26-49.

[Krief, 1990] Krief, P., *M.Pv.C - Un Système Interactif de Construction d'Environnements de Prototypage de Multiples Outils d'Interprétation de Modèles de Représentation*, Thèse de Doctorat, Université de Paris VIII, Paris, 1990.

[Krief, 1991] Krief, P., ACKPRO (2.5) Atelier de conception et de kits pour le prototypage, Rapport interne, ACKIA, Paris 1991

[Krogstie et Solvberg, 1996] Krogstie, J. et A. Solvberg, A classification of methodological framewoks for computerized information systems support in organizations, in *Working Conference on Method Engineering*, Atlanta, USA, 1996.

[Kulmar et Welke, 1992] Kulmar, K. et R.J. Welke, *Methodology Engineering: A proposal for Situation Specific Methodology Construction*, in *Challenges and Strategies for Research in Systems Development*, W.W. Kottermann and J.A. Senn, Editors. 1992: Washington.

[Lajoie et Keller, 1994] Lajoie, R. et R.K. Keller, Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert, in *ACFAS*, Montreal, Canada, 1994.

[Lamsweerde *et al.*, 1988] Lamsweerde, A.V., B. Delcourt, E. Delor, M.-C. Schayes et R. Champagne, *Generic Lifecycle support in the ALMA Environment.* IEEE Transactions on Software Engineering, 1988. **14**(6): pp. 720-741.

[Lesueur, 1997] Lesueur, B., MétaGen, Rapport de pré-southenance, Université de Paris VI, Paris 1997

[Lesueur *et al.*, 1998] Lesueur, B., N. Revault, G. Sunyé et M. Ziane, Using the MétaGen Modeling and Development Environment in the FIBOF-Esprit Project, in *Automating the Object-Oriented Software Development Workshop (ECOOP'98)*, Bruxelles, 1998.

[Lesueur *et al.*, 1999] Lesueur, B., G. Sunyé, Z. Guessoum, G. Blain et J.-F. Perrot, La métaphore du dossier, in *Inforsid'99*, , 1999.

[Linton *et al.*, 1989] Linton, M.A., J.M. Vlissides et P.R. Calder, *Composing user interfaces with Interviews* in *Computer.* 1989. pp. 8-22.

[Linval *et al.*, 1992] Linval, I.C.d., Y. Djorno et G. Levy, An Object-Oriented Representation in Attribute Grammars for the Generation of Incremental CASE Design Tools, in *Fifth International Conference on Software Engineering ans its Applications*, Toulouse, 1992.

[Loucopoulos et Theodoulidis, 1992] Loucopoulos, P. et B. Theodoulidis, *CASE-Methods and Support Tools*, in *Conceptual modeling, databases and CASE: an integrated view of information system development*, P. Loucopoulos and R. Zicari, Editors. 1992: USA. pp. 373-388.

[Loucopoulos et Zacari, 1992] Loucopoulos, P. et R. Zacari, *Conceptual modeling, databases and CASE: an integrated view of information systems development.* Wiley, New York, 1992.

[Martiin *et al.*, 1993] Martiin, P., M. Rossi, V.-P. Tahvanainen et K. Lyytinen, *A Comparative review of CASE shells: A preliminary framework and research outcomes.* Information & Management, 1993. **25**: pp. 11-31.

[Martin *et al.*, 1998] Martin, R., D. Riehle et F. Buschmann, eds. *Pattern Languages of Program Design 3.* . Addison-Wesley, Reading, MA, 1998.

[Meijers, 1996] Meijers, M., *Tool Support for Object-Oriented Design Patterns*, , Utrecht University, Utrecht, 1996.

[Meijler, 1997] Meijler, T.D., Overview of FACE, a Framework Adaptive Composition Environment, Universita de Bern, Bern, Switzerland 1997

[Meijler et Engel, 1996] Meijler, T.D. et R. Engel, Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment, in *Preliminary EuroPLoP Conference Proceedings*, , 1996.

[Mi et Scacchi, 1990] Mi, P. et W. Scacchi, *A knowledge-based environment for modelin and simulating software engineering processes.* IEEE Transactions on on Knowledge and Data Engineering, 1990. **2**(3): pp. 283-294.

[Mi et Scacchi, 1991] Mi, P. et W. Scacchi, Process integration for CASE environments, in *First International Conference on Sofware Processes*, , 1991.

[Mili et Mili, 1995] Mili, H. et A. Mili, *Reusing Software: Issues and Research Directions.* IEEE Transactions on Software Engineering, 1995. **21**(6): pp. 428-562.

[Mili *et al.*, 1996] Mili, H., H. Sahraoui et I. Benyahia, *Representing and Querying Object Frameworks.* 1996.

[Mili *et al.*, 1990] Mili, H., J. Sibert et Y. Intrator, *An Object-Oriented Model Based on Relations.* Journal of Systems and Software, 1990. **12**: pp. 139-155.

[Morejon *et al.*, 1991] Morejon, J., R. Oudrhiri, M.d. Gaudemont et P. Negros, *GraphOR: A Meta Design Tool*, in *Entity-Relationship Approach: The Core of Conceptual Modelling*, H. Kangassalo, Editor. 1991, Elsevier Science Publishers B. V. (North Holland): ERI. pp. 43-56.

[Mylopoulos *et al.*, 1990] Mylopoulos, J., A. Borgida, M. Jarke et M. Kourbatis, *Telos: representing knowledge about information system.* ACM Transactions on Information Systems, 1990. **8**(4): pp. 325-362.

[Neighbors, 1984] Neighbors, J.M., *The DRACO approach to constructing software from reusable components.* IEEE Transactions on Software Engineering, 1984. **10**(5): pp. 564-573.

[Neighbors, 1992] Neighbors, J.M., The Commercial Application of Domain Analysis, in *Fifth Workshop on Instituionalizing Software Reuse*, , 1992.

[Next, 1994] Next, *NEXTSTEP General Reference: Release 3.* Vol. 1,2. Addison-Wesley, Reading, MA, 1994.

[Nijssen et Halpin, 1989] Nijssen, G.M. et T.A. Halpin, *Conceptual Schema and Relational Database Design: A Fact Oriented Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Nunamaker, 1992] Nunamaker, J.F., *Build and Learn, Evaluate and Learn.* Informatica - The Journal of Management Information Systems Development, 1992. **1**(1): pp. 1-6.

[Nunamaker *et al.*, 1991] Nunamaker, J.F., M. Chen et T.D.M. Purdin, *Systems Development in Information Systemes Research.* Journal of Management Information Systems, 1991. **7**(3): pp. 89-106.

[Oei et Hemmen, 1994] Oei, J.L.H. et L.J.G.T.v. Hemmen, Harmonisation of Information System Modelling and Specification Techniques, in *Working Conference on Methods and Associated tools for the Information System Life Cycle (IFIP WG8.1)*, Maastricht, The Nethrlands, 1994.

[OMG, 1997a] OMG, Meta Object Facility (MOF) Specification, Object Management Group, 1997a

[OMG, 1997b] OMG, UML Proposal to the Object Management Group, Object Management Group, 1997b

[Opdyke, 1992] Opdyke, W.F., *Refactoring Object-Oriented Frameworks*, , University of Illinois, Urbana, 1992.

[Oquendo *et al.*, 1992] Oquendo, F., J.-D. Zucker et P. Griffiths, A Meta-CASE Environment for Software Process-centred CASE Environments, in *CAiSE '92*, Manchester, UK, 1992.

[Pachet, 1995] Pachet, F., *On the embeddability of production systems in object-oriented langages*. Journal of Object-Oriented Programming, 1995. **8**(4): pp. 19-24.

[Pachet et Dojat, 1992] Pachet, F. et M. Dojat, Representation of a medical expertise using the Smalltalk environment, in *TOOLS*, Dortmund, Germany, 1992.

[Pachet et Perrot, 1994] Pachet, F. et J.-F. Perrot, Rule Firing with Metarules, in *Software Engineering and Knowledge Engineering (SEKE'94)*, Jurmala, Lettonie, 1994.

[Pagel et Winter, 1996] Pagel, B.-U. et M. Winter, Towards Pattern-Based Tools, in *EuroPLoP'96*, , 1996.

[ParcPlace, 1990] ParcPlace, ed. *ObjectWorks\Smalltalk Release 4 Users Guide*. Mountain View, CA, 1990.

[ParcPlace, 1998] ParcPlace, ed. *VisualWorks 2.5 Users Guide*. Mountain View, CA, 1998.

[Parkinson, 1990] Parkinson, J., *Making CASE Work*, in *CASE on Trial*, K. Spurr and P. Laysell, Editors. 1990, John Wiley & Sons: Chichester.

[Paulich et Tichy, 1991] Paulich, F.N. et W. Tichy, *EDGE: an extensible graphe editor*. Software Practice and Experience, 1991. **20**(S1).

[Perrot, 1998] Perrot, J.-F., A Multi-Agent CASE Tool Environment, in *ISCIS'98*, Belek-Antalya, Tutkey, 1998.

[Pietro-Diaz, 1990] Pietro-Diaz, R., *Domain Analysis: an Introduction* in *ACM SIGSOFT Software Engineering notes*. 1990. pp. 47-65.

[Pietro-Diaz, 1993] Pietro-Diaz, R., Some Experiences in Domain Analysis, in *Sixth Workshop on Instituionalizing Software Reuse*, , 1993.

[Pree, 1995] Pree, W., *Design Patterns for Object Oriented Software Engineering*. Addison-Wesley, 1995.

[Premerlani, 1995] Premerlani, W., Metamodeling, in *TOOLS Tutorials*, Versailles, 1995.

[Revault, 1995] Revault, N., Describing and (Re-)Using Technological Application Frameworks in a Metamodeling-based Development Tool - an application of the MétaGen system, in *Workshop on Framework Centered Develoment (OOPSLA'95)*, Austin, USA, 1995.

[Revault, 1996] Revault, N., *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MétaGen et les frameworks)*, Thèse de Doctorat, Université P. et M. Curie (Paris 6), Paris, 1996.

[Revault *et al.*, 1995] Revault, N., H. Sahraoui, G. Blain et J.-F. Perrot, A Metamodeling Technique: The MétaGen System, in *TOOLS Europe'95*, Versailles, France, 1995.

[Rich et Waters, 1990] Rich, C. et R.c. Waters, *The Programmers' Apprentice*. Frontier series, ed. A. press. Addison-Wrsley, New York, 1990.

[Royce, 1970] Royce, W.W., Managing the Development of Large Software Systems : Concept and Techniques, in *WESCON*, , 1970.

[Rumbaugh *et al.*, 1991] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy et W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Saeki et Wenyin, 1994] Saeki, M. et K. Wenyin, Specifying Software Specification & Design Methods, in *Sixth International Conference in Advanced Information Systems Engineering (CAiSE '94)*, Ultrecht, The Netherlands, 1994.

[Sahraoui, 1995] Sahraoui, H., *Application de la méta-modelisation à la génération des outils de conception et de mise en oeuvre des bases de données*, , Université Pierre et Marie Curie, Paris, 1995.

[Sahraoui *et al.*, 1995] Sahraoui, H., R. Missaoui, J.-M. Gagnon et G. Sunyé, Using a Metamodeling Approach for Building an Object-Oriented Database Modeling and Design Tool, in *OOPSLA*, , 1995.

[Sahraoui et Revault, 1994] Sahraoui, H. et N. Revault, Modélisation Conceptuelle des Bases de Données: Techniques de Méta-Modélisation, in *CARI'94*, , 1994.

[Schefström, 1993] Schefström, D., *Systems Development Environments: Contemporany Aspects*, in *Tool Integration - Environments and Frameworks*, W.P. Computing, Editor. 1993.

[Shlaer et Mellor, 1992] Shlaer, S. et S.J. Mellor, *Object Lifecycles: Modeling the world in states*. Computing Series. Yourdon Press, 1992.

[Smolander, 1991] Smolander, K., *OPRR: A Model for Modelling Development Methods*, in *Next Generation of CASE Tools*, K. Lyytinen and V.-P. Tahvanainen, Editors. 1991, IOS Press: Amsterdam.

[Smolander *et al.*, 1991] Smolander, K., K. Lyytinen, V.-P. Tahvanainen et P. Martiin, MetaEdit - A Flexible Graphical Environment for Methodology Modelling, in *CAiSE '91*, Trondheim, Norway, 1991.

[Smolárová *et al.*, 1998] Smolárová, M., P. Návrat et M. Bieliková, Abstracting and Generalising with Design Patterns, in *ISCIS'98*, Belek-Antalya, Turkey, 1998.

[Snider, 1991] Snider, C., GLAD user's manual, Arcadia Technical Report, University of California, Irvine 1991

[Soberman, 1992] Soberman, M., *Génie Logiciel en Informatique de Gestion*, ed. Eyrolles, Paris, 1992.

[Sommerville, 1995] Sommerville, I., *Software Engineering*. 5th ed. Addison-Wesley, 1995.

[Sorenson *et al.*, 1988] Sorenson, P.G., J.-P. Tremblay et A.J. McAllister, *The Metaview System for Many Specification Environments* in *IEEE Software*. 1988. pp. 30-38.

[Soukup, 1995] Soukup, J., *Implementing Patterns*, in *Pattern Languages of Program Design*, J.O. Coplien and D.C. Schimidt, Editors. 1995.

[Stobart *et al.*, 1993] Stobart, S.C., A.J.v. Reeken, G.C. Low, J.J.M. Trienekens, J.O. Jenkins, J.B. Thompson et D.R. Jeffery, An Empirical Evaluation of the Use of CASE Tools, in *Sixth International Workshop on Computer-Aided Software Engineering (CAiSE'93)*, , 1993.

[Sunyé, 1999] Sunyé, G., Génération de code à l'aide de patrons de conception, in *Langages et Modèles à Objets - LMO'99*, Villeneuve s/ mer, 1999.

[Sunyé *et al.*, 1997] Sunyé, G., H. Sahraoui, B. Lesueur et G. Blain, *Chroniques d'implémentation d'un méta-outil en Smalltalk*. L'Objet, 1997. **3**(4): pp. 411-427.

[Sutton *et al.*, 1988] Sutton, S.M., D. Heimbigner et L. Osterweil, Programmable relations for managing change during software development, Technical Report, University of Colorado, Boulder, CO 1988

[Sutton *et al.*, 1990] Sutton, S.M., D. Heimbigner et L.J. Osterweill, Language constructs for managing change in process-centered environments, in *SIGSOFT '90 Proceedings of the fourth ACM SIGSOFT Symposium on Software Development Environments*, , 1990.

[Tarr et Clarke, 1993] Tarr, P. et L.A. Clarke, Pleiades: An Object Management System for Software Engineering Environments, in *ACM SIGSOFT'93 Symposium on the Fondations of Software Engineering*, Los Angeles, California, 1993.

[Taylor, 1988] Taylor, R., Fundations for the Arcadia environment architecture, in *Third ACM Symposium on Software Development Environments*, , 1988.

[Taylor *et al.*, 1996] Taylor, R., N. Medvidovic, K.M. Anderson, E.J.W. Jr., J.E. Robbins, K.A. Nies, P. Oreizy et D.L. Dubrow, *A Component and Message-Based Architectural Style for GUI Software*. IEEE Transactions on Software Engineering, 1996(June 1996).

[Teichrorew *et al.*, 1980] Teichrorew, D., P. Macasovic, E.A. Hershey et Y. Yamamoto, *Application of the entity-relationship approach to information processing systems modeling*, in *Entity-Relationship Approach to Systems Analysis and Design*, P.P. Chen, Editor. 1980, North-Holland. pp. 15-38.

[Teitelman et Masinter, 1981] Teitelman, W. et L. Masinter, *The Interlisp programming environment*. IEEE Computer, 1981. **14**(4): pp. 25-33.

[Thomas, 1989] Thomas, I., *PCTE Interfaces Supporting Tools in Software-Engineering Environments* in *IEEE Software*. 1989. pp. 15-23.

[Tolvanen *et al.*, 1996] Tolvanen, J.P., M. Rossi et H. Liu, Method Engineering: Current research directions and implications for future research, in *Method Engineering '96 - Principles of method construction and support, Working Conference on Method Engineering (IFIP 8.1/8.2)*, , 1996.

[UMIST, 1995] UMIST, PHEDIAS: A meta-CASE Tool, Technical Report, UMIST, Manchester 1995

[Ungar et Smith, 1987] Ungar, D. et R.B. Smith, Self: The power of simplicity, in *OOPSLA'87*, Orlando, 1987.

[Verhoef et Hofstede, 1995] Verhoef, T.F. et A.H.M. Hofstede, Feasibility of flexible information modelling support, in *Advanced Information Systems Engineering*, , 1995.

[Verschaeve *et al.*, 1996] Verschaeve, K., B. Wydaeghe, V. Jonckers et L. Cuypers, Translating OMT* to SDL, coupling object-oriented analysis and design with formal description techniques, in *Working Conference on Method Engineering*, Atlanta, USA, 1996.

[Vlissides *et al.*, 1996] Vlissides, J.M., J.O. Coplien et N.L. Kerth, eds. *Pattern Languages of Program Design 2.* . Addison-Wesley, Reading, MA, 1996.

[Vlissides et Linton, 1988] Vlissides, J.M. et M.A. Linton, Applying object oriented design to structures graphics, in *USENIX C++*, Denver, CO, 1988.

[Vlissides et Linton, 1990] Vlissides, J.M. et M.A. Linton, *Unidraw: A framework for building domain-specific graphical editors.* ACM Transactions on Information Systems, 1990. **9**(33): pp. 104-124.

[Wang et Loucopoulos, 1995] Wang, X. et P. Loucopoulos, The Development of Phedias: a CASE Shell, in *International Workshop on Computer-Aided Software Engineering (CASE '95)*, Toronto, Ontario, Canada, 1995.

[Weinand *et al.*, 1988] Weinand, A., E. Gamma et R. Marty, ET++ An Object-Oriented Application Framework in C++, in *OOPSLA'88*, , 1988.

[Weinand *et al.*, 1989] Weinand, A., E. Gamma et R. Marty, *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework.* Structured Programming, 1989. **10**(2): pp. 63-87.

[Wijers, 1991] Wijers, G., *Modelling support in information systems development*, Ph.D. Thesis, University of Nijmegen, Amsterdam, 1991.

[Wolf *et al.*, 1988] Wolf, A., J. Wileden, C. Fisher et P. tarr, Pgraphite: an experiment in persistent typed object management, in *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, , 1988.

[Young, 1994] Young, P.S.C., *Customizable Process Specification and Enactment for Technical and non-Technical Users*, PhD, University of California, Irvine, 1994.

[Yourdon, 1986] Yourdon, E., *Whatever Happened to Structured Analysis?* in *Datamation*. 1986. pp. 133-138.

[Yourdon, 1989] Yourdon, E., *Modern Structured Analysis*, ed. E. Cliffs. Prentice Hall, New Jersey, 1989.