

Université de Nantes

École doctorale
Sciences et Technologies
de l'Information et des Matériaux

Année 2003
N° B.U. ...

Thèse de doctorat de l'Université de Nantes

en spécialité informatique
présentée et soutenue publiquement par

Yann-Gaël Guéhéneuc

le 23 juin 2003
à l'École Nationale Supérieure des
Techniques Industrielles et des Mines de Nantes

**Un cadre pour la traçabilité
des motifs de conception**

devant la commission d'examen composée de

Présidente	:	Isabelle Borne	Professeur, Université de Bretagne Sud
Rapporteurs	:	Stéphane Ducasse	Professeur, Université de Berne
		Jean-François Perrot	Professeur, Université Pierre et Marie Curie
Examineurs	:	Brian Barry	Docteur, Bedarra Research Labs, Eclipse.org
		Mourad Oussalah	Professeur, Université de Nantes
Directeur de thèse	:	Pierre Cointe	Professeur, École des Mines de Nantes

Équipes d'accueil : Objets, Composants et Modèles ; Programmation Par Contraintes
Laboratoire d'accueil : département informatique de l'École des Mines de Nantes

La Chantrerie – 4, rue Alfred Kastler – 44 307 Nantes Cedex 3 – France

Partenaire industriel : IBM OTI Labs, anciennement Object Technology International, Inc.

2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

N° E.D. 0366-...

À Seung-hee

À mon frère et mes parents

Remerciements

...

*Si tu sais méditer, observer et connaître
Sans jamais devenir sceptique ou destructeur ;*

...

Tu seras un homme, mon fils.

Extrait de la traduction (1918) en vers par André Maurois
du poème “If...” (1910) de Rudyard Kipling.

Il m’est difficile de remercier ici toutes les personnes grâce à qui j’ai pu mener à bien ces trois années de recherche ; trois années de rencontre, d’échange qui ne peuvent se résumer à une simple liste de noms.

Pourtant, je veux remercier Seung-hee, qui m’a accompagné pendant toutes ces années, qui a su me préserver, me garder la tête sur les épaules et qui sera à mes côtés j’espère pour très longtemps encore.

Aussi, je voudrais remercier ceux auprès de qui j’ai découvert la recherche, ses rigueurs et ses joies : mon directeur de thèse, Pierre Cointe, qui m’a accueilli au sein du département et m’a permis de réaliser ma thèse dans les meilleures conditions, Christine Violeau, Hervé Albin-Amiot, Brian Barry, Rémi Douence, Andrés Farías, Narendra Jussien, Hervé Leblanc, Thierry Petit...

Bien sûr, je tiens à remercier ma famille et mes amis pour leur présence dans les moments de joie et leur soutien dans les moments de peine : Christiane, André et Gwénaél, mes parents bien-aimés ; Elisa, Fanny, Livia, Benjamin, Hicham, Pascal, Per-Erik...

Enfin, je remercie les membres de mon jury et en particulier mes rapporteurs, Stéphane Ducasse et Jean-François Perrot, pour leurs conseils éclairés ; et la société Object Technology International, Inc. : Paddy Armstrong, Nora Rojas, Brian Barry, Paul Buck, Kevin McGuire, Steve McDermid, Dave Thomas... pour leur aide et leur confiance.

Merci également à Jean-François Bédard pour sa lecture complète et ses nombreux commentaires.

Note aux lecteurs

NOUS présentons nos travaux sur l'identification et la traçabilité des motifs de conception. Ces travaux s'inscrivent dans le cadre de la qualité des programmes à objets et des règles de bonne programmation.

En particulier, nous respectons la règle contraignant le code source des langages de programmation comme C++ ou JAVA à être en anglais par cohérence avec leurs mots-clés, même si les algorithmes correspondants sont en pseudo-code et en français.

Aussi, nous utilisons les conventions typographiques suivantes pour mettre en valeur ou distinguer des éléments du texte et en préciser le sens :

- les noms des langages de programmation, des méthodes, des outils, des produits sont en petites majuscules : JAVA, PALM ;
- les noms des constituants d'un métamodèle ou d'un modèle sont en police non-proportionnelle avec empattement : **Figure**, **PolyLineFigure** ;
- les références bibliographiques sont données entre crochets et en français, quelle que soit la langue utilisée pour rédiger le document référencé : [Gamma *et al.*, 1994] ;
- les citations sont entre guillemets et en italique : “*Every model needs a meta model.*” [Thomas, 2002, page 18] ;
- les mots importants sont en italique dans la phrase où ils apparaissent ;
- les entrées du glossaire sont en italique et suivies d'une étoile la première fois qu'elles apparaissent dans le texte : *patron de conception*^{*}, *traçabilité*^{*} ;
- les noms des motifs interclasses et des motifs de conception sont en police sans empattement avec première lettre capitale : **Composite**, **Singleton** ;
- les noms des rôles des participants d'un motif de conception sont aussi en police sans empattement avec première lettre capitale : **Composant**, **Feuille**, **Collègue** ;
- les noms des niveaux d'abstraction sont en police sans empattement avec première lettre minuscule : **implémentation**, **conception** ou **analyse** ;
- les mots en langue étrangère sont en italique : *chunks*, *design pattern* ;
- une paire de crochets indique un intervalle sur les entiers : $[i, j] \subset \mathbb{N} \mid i \in \mathbb{N} < j \in \mathbb{N}$;
- une paire de parenthèses indique un couple, un 2-uplet : soit \mathcal{C} un ensemble de contraintes, $(c_1, c_2) \subset \mathcal{C}$ avec $c_1 \in \mathcal{C}, c_2 \in \mathcal{C}$;
- une paire d'accolades indique un ensemble d'éléments : soit \mathcal{V} un ensemble de variables et n un entier : $\{v_1, \dots, v_n\} \subset \mathcal{V}$ avec $\forall i \in [1, n], v_i \in \mathcal{V}$;
- les lettres \mathcal{C} et \mathcal{V} désignent toujours un ensemble de contraintes et un ensemble de variables, respectivement.

Sommaire

Introduction	5
I Problématique	9
1 Domaine d'étude	11
1.1 Contexte : la maintenance des programmes à objets	11
1.2 Motivation : la compréhension des programmes par les motifs	14
1.3 Problème : la traçabilité des motifs interclasses et de conception	17
1.4 Thèse : la définition de ces motifs, des analyses de programmes et la programmation par contraintes avec explications	22
1.5 Scénario : le programme JHOTDRAW et le motif de conception Composite .	22
1.6 Contribution : un cadre pour la traçabilité des motifs de conception	26
2 État de l'art sur la traçabilité des motifs	31
2.1 Classification utilisée	32
2.2 Motifs interclasses	37
2.3 Motifs de conception	60
Bilan	73
II Traçabilité des motifs	75
3 Motifs interclasses	77
3.1 Modélisation d'un programme	79
3.2 Définitions des motifs	84
3.3 Propriétés des motifs	92
3.4 Formalisations des motifs avec leurs propriétés	101
3.5 Algorithmes d'identification des motifs	116
3.6 Algorithmes de traçabilité des motifs	122
3.7 Discussion des modèles, des définitions et des algorithmes	126
3.8 Application à JHOTDRAW	136
Bilan	141

4	Motifs de conception	143
4.1	Modélisation d'un programme	145
4.2	Définitions des motifs	148
4.3	Programmation par contraintes et explications	155
4.4	Contraintes associées aux motifs	166
4.5	Algorithmes de résolution des problèmes à contraintes	169
4.6	Algorithmes de traçabilité des motifs	176
4.7	Discussion des modèles, des contraintes et des algorithmes	178
4.8	Application à JHOTDRAW	184
	Bilan	190
	Bilan de la traçabilité des motifs	191
III	Mise en œuvre de la traçabilité des motifs avec PTIDEJ	195
5	Métamodélisation	197
5.1	Présentation du métamodèle PDL	198
5.2	Extension du métamodèle avec PADL	201
5.3	Visualisation des modèles avec PTIDEJ UI	206
	Bilan	208
6	Motifs interclasses	211
6.1	Analyses statiques avec INTROSPECTOR	212
6.2	Analyses dynamiques avec CAFFEINE	216
6.3	Algorithmes de traçabilité des motifs	228
	Bilan	232
7	Programmation par contraintes avec explications	233
7.1	Solveur de contraintes PALM	234
7.2	Solveur de contraintes PTIDEJ SOLVER	238
7.3	Bibliothèque de contraintes PTIDEJ LIBRARY	242
7.4	Stratégies de recherche	247
	Bilan	252
8	Motifs de conception	253
8.1	Modélisation d'un motif avec PADL	254
8.2	Génération d'un problème avec PTIDEJ	256
8.3	Résolution du problème avec PTIDEJ SOLVER	263
8.4	Algorithmes de traçabilité des motifs	269
	Bilan	272
	Bilan de la mise en œuvre de la traçabilité des motifs avec PTIDEJ	273

IV Conclusion et perspectives	275
Conclusion	277
Perspectives	279
10.1 Utilisations du cadre	279
10.2 Extensions du cadre	281
V Annexes	283
Autres plans du mémoire	285
Détails sur JHotDraw	289
Détails sur PTIDEJ	291
Bibliographie	293
Glossaire	315
Listes	329
Algorithmes	329
Codes source	329
Définitions	331
Exemples	331
Figures	332
Propriétés	334
Tableaux	334
Index	337

Introduction

LE LIVRE “*Design Patterns – Elements of Reusable Object-Oriented Software*” [Gamma *et al.*, 1994] propose vingt-trois patrons de conception reconnus par la communauté du génie logiciel à objets et utilisés pour développer des programmes de qualité.

Un patron de conception identifie et nomme un problème récurrent de conception, propose une solution élégante à ce problème et décrit les compromis réalisés lorsque cette solution est appliquée.

La solution proposée par un patron est un motif de conception dont les variantes sont utilisées lors de la phase d’implantation des programmes pour résoudre les problèmes identifiés pendant la phase de conception.

Ainsi, les patrons permettent aux développeurs de réaliser des programmes de qualité avec des solutions éprouvées. Ils permettent aussi de caractériser la qualité de l’implantation et de la conception des programmes avec les motifs utilisés.

Les programmes entrent en maintenance après leur implantation et leur mise en exploitation. Ils sont alors modifiés pour corriger des bogues, pour ajouter de nouvelles fonctionnalités ou pour améliorer leur réutilisabilité.

La maintenance nécessite toujours une phase de rétroconception des programmes pour identifier les choix réalisés lors de leur conception et de leur implantation. La rétroconception peut être implicite ou explicite, par la production de modèles.

Les motifs de conception ont une place prépondérante à prendre lors de la rétroconception des programmes pour aider à identifier les choix de conception réalisés et ainsi faciliter la compréhension et la modification des programmes.

Cependant, les motifs utilisés pendant l’implantation des programmes n’apparaissent souvent plus explicitement dans le code source des programmes, ils y sont disséminés et leur traçabilité n’est pas garantie entre l’implantation et la rétroconception.

Ce mémoire propose et décrit des modèles et des algorithmes pour garantir la traçabilité des motifs de conception entre les phases d’implantation et de rétroconception des programmes par l’identification semi-automatique des micro-architectures similaires à ces motifs dans le code source des programmes.

Les motifs de conception sont habituellement modélisés par des diagrammes de classes, proches des diagrammes de classes de la notation UML, définissant la structure et le comportement global des participants du motif et des relations entre ces participants.

Aussi, nous devons modéliser les programmes, dans lesquels nous voulons identifier les motifs de conception, par des diagrammes de classes pour garantir la cohérence entre les modèles des motifs recherchés et ceux des programmes.

La modélisation des programmes par des diagrammes de classes nous conduit à préciser la sémantique des constituants de ces diagrammes car les langages à classes industriels, comme JAVA, ne proposent pas de constructions qui rendent compte de leur variété.

D'une manière générale, la notation UML propose pour constituants des diagrammes de classes : les classes, les interfaces et les relations d'appel de méthodes, d'héritage, d'instanciation, d'association, d'agrégation et de composition.

Les classes, les interfaces et les relations d'appel de méthodes, d'héritage et d'instanciation correspondent à des constructions existantes dans les langages de programmation, elles apparaissent explicitement dans le code source des programmes.

Au contraire, les relations d'association, d'agrégation et de composition n'ont pas de construction correspondante dans les langages de programmation, elles sont implicites dans le code source des programmes.

Nous définissons précisément les relations d'association, d'agrégation et de composition de UML et nous proposons des algorithmes d'analyses statiques et dynamiques pour les identifier dans des programmes en JAVA.

Nous proposons des définitions aussi consensuelles que possible des relations d'association, d'agrégation et de composition des points de vue de l'implantation et de la conception, après avoir dressé un état de l'art des définitions existantes.

En résumé, la relation d'association entre deux classes indique que les instances d'une classe peuvent envoyer des messages aux instances de l'autre classe. La relation d'agrégation entre deux classes caractérise une classe qui possède des références sur des instances de l'autre classe. La relation de composition est une relation d'agrégation avec des contraintes sur la durée de vie et l'appartenance des instances des deux classes.

Les définitions de ces relations interclasses nous conduisent à isoler quatre propriétés minimales qui les caractérisent et qui nous permettent de les définir opérationnellement : durée de vie, exclusivité, multiplicité et site d'invocation.

La durée de vie contraint l'ordre des destructions des instances de deux classes en relation de composition ; l'exclusivité contraint l'appartenance des instances de ces deux classes. La multiplicité précise les nombres d'instances de deux classes en relation d'association, d'agrégation ou de composition ; le site d'invocation caractérise les appels de méthodes entre les instances de ces deux classes.

Nous identifions les relations entre deux classes en calculant les valeurs de leurs propriétés statiques et dynamiques, durée de vie et site d'invocation, exclusivité et multiplicité, et en les comparant avec les valeurs définies pour les relations interclasses.

Nous proposons des algorithmes d'analyses syntaxiques pour calculer les valeurs des propriétés statiques et des algorithmes d'analyses des traces d'exécution pour calculer les valeurs des propriétés dynamiques.

Ces algorithmes nous permettent de construire automatiquement des diagrammes de classes représentatifs des programmes analysés et de garantir la traçabilité des relations interclasses entre ces modèles et le code source des programmes.

Alors, nous pouvons identifier dans ces modèles de programmes, représentés par des diagrammes de classes, les micro-architectures similaires aux modèles de motifs de conception, également représentés par des diagrammes de classes.

Les micro-architectures peuvent être des formes complètes d'un motif de conception si leurs constituants respectent le type et la cardinalité des participants définis dans le motif et les relations préconisées entre ses participants ; ou des formes approchées, le cas le plus courant, si leurs constituants ou leurs relations sont différents de ceux préconisés.

Ces formes complètes et approchées traduisent les compromis réalisés par les développeurs lors de l'implantation des programmes et de l'application des motifs ; leur intérêt pour la compréhension des programmes varie avec les objectifs de la maintenance : correction de bogues, redocumentation, restructuration.

L'identification des micro-architectures similaires à un motif de conception doit être interactive et doit expliquer les raisons pour lesquelles une micro-architecture est une variante d'un motif de conception, une forme complète ou approchée.

Nous définissons des algorithmes pour identifier interactivement les micro-architectures similaires à des motifs de conception, formes complètes et approchées, et pour expliquer les raisons de leur identification comme des variantes des motifs.

L'identification des micro-architectures similaires au modèle d'un motif de conception nécessite l'identification des constituants du modèle du programme, classes et interfaces, avec des relations similaires aux relations préconisées par le motif.

Les relations définies entre les participants du motif contraignent le choix des constituants du modèle du programme qui implantent une variante de ce motif pour résoudre le problème de conception correspondant.

Nous traduisons l'identification des micro-architectures similaires à un motif en un problème de satisfaction de contraintes dans lequel les participants du motif sont les variables du problème, les relations entre ces participants sont les contraintes et les constituants du modèle du programme forment le domaine des variables.

Une solution à ce problème de satisfaction de contraintes est un ensemble de constituants du modèle du programme dont les relations satisfont celles préconisées par le motif de conception, qui forment une micro-architecture similaire au motif.

La méthode de résolution du problème de satisfaction de contraintes doit être interactive et dynamique pour permettre aux mainteneurs de guider la recherche, pour expliquer les solutions obtenues et pour supporter l'identification des formes approchées.

Nous utilisons la programmation par contraintes avec explications comme méthode de résolution car elle permet la relaxation interactive et dynamique des contraintes et du problème pour obtenir et expliquer les formes complètes et approchées.

En particulier, l'explication associée à une forme approchée est l'ensemble des contraintes qui ont été relaxées ou retirées du problème pour l'identifier : l'ensemble des relations préconisées par le motif qui ne sont pas satisfaites.

Cet ensemble de relations insatisfaites représente les compromis réalisés lors de l'application du motif de conception, lors de son implantation par la micro-architecture identifiée dans le code source du programme.

Aussi, nous proposons des algorithmes pour construire un modèle du programme qui intègre les micro-architectures similaires à des motifs de conception et qui garantit leur traçabilité entre ce modèle du programme et son code source.

La suite d'outils PTIDEJ est une implantation en JAVA des modèles et des algorithmes décrits. Elle est intégrée à l'environnement de développement ECLIPSE. Elle permet la modélisation des programmes JAVA, l'identification et la traçabilité des relations interclasses et des motifs de conception entre les phases d'implantation et de rétroconception.

Elle inclut le métamodèle PADL, dérivé du métamodèle PDL issu d'une précédente thèse de doctorat [Albin-Amiot, 2003], pour modéliser les relations interclasses, les motifs de conception et les programmes JAVA.

Elle propose des outils d'analyses statiques et dynamiques, INTROSPECTOR et CAF-EINE, pour calculer les valeurs des propriétés de durée de vie, d'exclusivité, de multiplicité et de site d'invocation et pour identifier les relations interclasses.

Enfin, elle offre un solveur de contraintes, PTIDEJ SOLVER, dérivé de l'implantation de référence de la programmation par contraintes avec explications, PALM, pour identifier semi-automatiquement les micro-architectures similaires à des motifs de conception.

Nous mettons en œuvre la suite d'outils PTIDEJ pour identifier les micro-architectures similaires au motif de conception **Composite** dans le programme JHOTDRAW et pour garantir leur traçabilité entre les phases d'implantation et de rétroconception de ce programme.

Première partie

Problématique

Nous présentons le contexte de nos travaux et précisons le vocabulaire utilisé pour étudier et mettre en œuvre la traçabilité des motifs de conception entre les phases d'implantation et de rétroconception d'un programme, lors de sa maintenance.

Nous détaillons nos motivations en montrant que l'identification et la traçabilité des motifs de conception sont difficiles et que des outils sont nécessaires pour faciliter la compréhension de l'implantation et de la conception du programme.

La construction d'outils pour garantir la traçabilité des motifs de conception pose deux problèmes distincts :

- l'identification automatique et la traçabilité des relations interclasses ;
- l'identification semi-automatique et l'explication et la traçabilité des micro-architectures similaires à des motifs de conception.

Nous soutenons la thèse qu'il est possible de définir les motifs interclasses et de conception, de proposer des algorithmes d'analyses de programmes pour identifier automatiquement les motifs interclasses et garantir leur traçabilité et d'utiliser la programmation par contraintes avec explications pour identifier les micro-architectures similaires à des motifs, les expliquer et garantir leur traçabilité.

Les définitions et les algorithmes que nous proposons pour soutenir notre thèse sont illustrés sur l'exemple fil conducteur de la traçabilité du motif de conception **Composite** dans le programme JHOTDRAW.

Enfin, nous résumons nos contributions sur le problème de la traçabilité des motifs de conception et détaillons le contenu des différentes parties et des chapitres qui composent ce mémoire avant de présenter un état de l'art sur la traçabilité des motifs.

Chapitre 1

Domaine d'étude

1.1 Contexte : la maintenance des programmes à objets

LA PHASE de *maintenance*^{*} succède aux phases de conception et d'implantation des programmes, après leur mise en exploitation. Elle est prépondérante dans le coût financier, temporel et humain des programmes [Sharon, 1996 ; Takang et Grubb, 1996 ; Pressman, 2001]. Elle se décompose généralement en trois sous-phases :

1. La sous-phase de rétroconception consiste à analyser un programme pour en créer des *modèles*^{*} de plus hauts niveaux d'abstraction que le code source¹ ;
2. La sous-phase de compréhension consiste à analyser les modèles du programme pour comprendre son implantation et identifier les modifications à effectuer ;
3. La sous-phase de modification consiste à retourner au niveau de l'implantation et à réaliser les modifications désirées.

La rétroconception d'un programme fournit des modèles de son *architecture*^{*} qui facilitent la compréhension des problèmes de conception rencontrés pendant son développement, des solutions adoptées et des compromis réalisés [Perrochon et Mann, 1999].

Les modèles d'un programme s'expriment à différents *niveaux d'abstraction*^{*}, typiquement : le niveau **implémentation**, dans lequel le programme est écrit de façon concrète ; le niveau **conception**, dans lequel les choix de conception sont représentés ; le niveau **analyse**, dans lequel l'architecture générale du programme est explicitée².

Les niveaux d'abstraction se distinguent des *vues* [Object Management Group, Inc., 2001, page Glossary-16] car une vue présente, à un niveau d'abstraction donné, un sous-ensemble des informations contenues dans le modèle du programme ; un niveau d'abstrac-

¹Le code binaire peut aussi être utilisé dans la sous-phase de rétroconception, mais cette pratique pose des problèmes légaux [Samuelson, 1990 ; Samuelson, 2002].

²Le sens donné à l'expression *niveaux d'abstraction* est à distinguer des sens donnés aux niveaux d'abstraction M_0 , M_1 , M_2 et M_3 en métamodélisation [Lemesle, 2000].

tion abstrait ou précise un modèle du programme pour en obtenir une nouvelle représentation, un nouveau modèle.

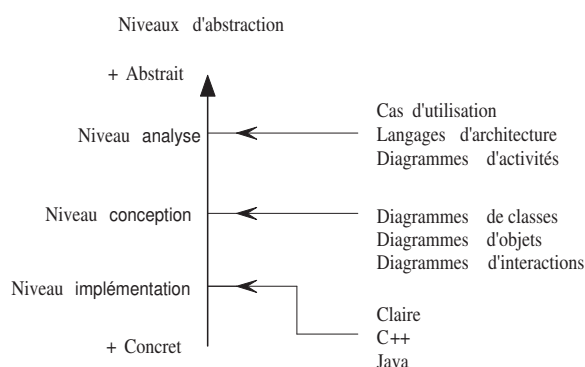
Les niveaux d'abstraction donnent une vision à la fois horizontale et verticale du programme par rapport à son *développement** : d'une part, chaque niveau d'abstraction représente le programme dans une vision horizontale des phases de son développement [Antoniol *et al.*, 2001] ; d'autre part, il propose un modèle plus abstrait et plus compréhensible du programme dans une hiérarchie verticale des niveaux d'abstraction [Sartipi *et al.*, 2000].

Le modèle d'un programme à un niveau d'abstraction donné est décrit par un *formalisme** approprié. Un formalisme définit les constituants nécessaires à la description du modèle du programme au niveau d'abstraction considéré.

Par exemple, sur la figure 1.1, un modèle du programme au niveau **implémentation** est décrit par du code source en JAVA. Un modèle du même programme au niveau **conception** est décrit par des diagrammes de conception, tels les diagrammes de classes de la notation³ UML. Un modèle du programme au niveau **analyse** est décrit par des cas d'utilisation, des diagrammes d'activités ou des spécifications dans un langage de description des architectures, tel WRIGHT [Allen et Garlan, 1997].

Les *constituants** du formalisme sont utilisés pour décrire les entités d'un programme au niveau d'abstraction considéré, avec la précision désirée. Ils sont textuels, par exemple le mot-clé `class` du langage de programmation JAVA annonce la déclaration d'une classe, ou graphiques, par exemple une boîte compartimentée dans la notation UML représente une classe, une interface ou un type paramétré.

FIG. 1.1 – Exemples de formalismes pour décrire les niveaux d'abstraction.



□

³Nous considérons, sans entrer dans le débat, que UML est une *notation* plutôt qu'un *langage* car un langage doit offrir une sémantique claire sinon formelle de tous ses constituants, ce qui n'est pas le cas de UML, comme expliqué dans la section 1.3 page 17 et montré dans la section 2.2 page 37.

Un formalisme définit aussi des *motifs*^{*}. Un motif regroupe et abstrait, sous une forme atomique, des constituants et d'autres motifs des niveaux d'abstraction inférieurs à celui du formalisme considéré. Un motif rassemble en un tout cohérent des constituants et d'autres motifs autrement disséminés dans le modèle du programme et qui participent à la résolution d'un même problème. Ainsi, les motifs contribuent à la séparation des préoccupations [Kiczales *et al.*, 1997 ; Hannemann et Kiczales, 2002].

Un formalisme peut aussi définir des *patrons*^{*}. Les patrons⁴ ont d'abord été proposés et utilisés en architecture par Christopher Alexander [1978], puis appliqués en génie logiciel [Coplien, 1991 ; Gamma *et al.*, 1994 ; Isazadeh *et al.*, 1995 ; Buschmann *et al.*, 1996 ; Fowler, 1996]. Un patron est un ensemble $\{nom, problème, solution, compromis\}$. Il associe à un problème récurrent d'implantation, de conception, d'analyse, une solution et les compromis réalisés lorsque la solution proposée est appliquée. Il décrit à la fois un constituant du formalisme et le processus nécessaire pour l'appliquer [Coplien, 1998].

Par exemple, le niveau d'abstraction *conception* rassemble des entités comme les classes, les méthodes, et aussi des patrons de conception [Gamma *et al.*, 1994], comme **Composite**, **Médiateur** et **Singleton**. Chacun de ces patrons propose une solution sous la forme d'un motif qui se retrouve dans le modèle du programme au niveau *conception*⁵.

Les patrons aident à la compréhension des modèles d'un programme et les *mainteneurs*^{*} profitent de la connaissance des motifs présents dans l'implantation du programme pour comprendre les problèmes résolus lors de sa conception et de son implantation et les solutions appliquées [Lieberman, 1987 ; Fischer *et al.*, 1992 ; Johnson et Erdem, 1995].

Par exemple, pour reprendre l'exemple donné dans [Rich et Waters, 1990]⁶, supposons que nous montrions un circuit électronique à un ingénieur électronicien et que nous lui en demandions le gain, un ingénieur expérimenté tentera d'abord d'identifier la forme du circuit et en utilisera les propriétés connues pour calculer le gain. De même, en génie logiciel, supposons que nous montrions à un ingénieur informaticien un programme de traitement de données et que nous lui demandions le temps de calcul maximal pour une entrée donnée, un ingénieur expérimenté identifiera d'abord les algorithmes utilisés et calculera le temps maximum d'exécution avec leurs propriétés. Les ingénieurs identifient une implantation de la solution à un problème récurrent d'électronique ou d'informatique, une micro-architecture similaire à un motif, et ils utilisent cette micro-architecture pour en déduire le problème posé et comprendre la solution appliquée.

⁴Les patrons sont appelés *patterns* en anglais [Office québécois de la langue française, 2003].

⁵Les mots *motif* et *patron* sont en général utilisés pour traduire la notion de *pattern* ; nous utilisons ces mots pour distinguer une solution générale, le motif, de la description du processus pour appliquer cette solution, le patron [Coplien, 1998].

⁶Dans leur livre, Charles Rich et Richard C. Waters [1990, page 11] remarquent : “*Suppose you show an electrical engineer a circuit and ask him to tell you its gain [...]. The first thing an experienced engineer will do is attempt to recognize the form of the circuit. [...] Similarly in software engineering, suppose you show an experienced software engineer a large data-processing system and inquire as to its maximum running time for given size inputs. Rather than resorting to the first principles of complexity analysis, the experienced engineer will first identify the algorithms being employed and then use their known properties to compute the running time.*”

Cependant, les mainteneurs disposent, en général, *uniquement* du modèle d'un programme au niveau **implémentation** car les documents de conception et d'implantation sont souvent absents ou obsolètes [Gall *et al.*, 1996 ; Richner et Ducasse, 1999].

Ils n'ont pas d'assistance pour identifier dans le modèle du programme au niveau **implémentation**, son code source, les *micro-architectures** similaires à des motifs et pour en déduire les patrons utilisés et les compromis réalisés.

Ils doivent analyser manuellement le modèle du programme au niveau **implémentation** pour y identifier les micro-architectures dont les *structures** sont similaires à des motifs, ces analyses sont fastidieuses et sources d'erreurs [Bansiya, 1998 ; Mancoridis *et al.*, 1998 ; Richner et Ducasse, 1999].

1.2 Motivation : la compréhension des programmes par les motifs

Nous cherchons à assister *semi-automatiquement* les mainteneurs lors de la rétroconception et de la compréhension d'un programme en favorisant l'utilisation des patrons de conception.

Nous décomposons les sous-phases de rétroconception et de compréhension d'un programme en trois phases, représentées sur la figure 1.2 page 15 :

1. Une phase d'*identification* d'une micro-architecture similaire aux motifs d'un ou plusieurs patrons connus. Un mainteneur analyse manuellement le modèle du programme au niveau **implémentation** et y identifie les constituants dont la structure et l'organisation sont similaires à la solution d'un patron connu. Cette phase est fastidieuse et source d'erreurs [Bansiya, 1998 ; Mancoridis *et al.*, 1998 ; Richner et Ducasse, 1999]. Elle dépend des patrons connus par le mainteneur et de la similarité entre la micro-architecture isolée et le motif proposé comme solution.
2. Une phase de *contextualisation* de la micro-architecture pour isoler un patron unique : avec des informations sémantiques [Albin-Amiot, 2003, page 109] extérieures au modèle du programme. Le mainteneur choisit, parmi les patrons candidats, le patron dont le motif est vraisemblablement implanté par la micro-architecture identifiée à la phase 1. La contextualisation dépend du domaine d'application du programme et de l'expérience du mainteneur.
3. Une phase de *compréhension* du programme à un plus haut niveau d'abstraction. Avec le patron reconnu à la phase 2, dont le motif est implanté par la micro-architecture identifiée à la phase 1, le mainteneur déduit le problème résolu et obtient une meilleure compréhension de la raison d'être de la micro-architecture, de sa structure, des relations entre ses constituants, à un plus haut niveau d'abstraction. Le mainteneur peut alors juger de la pertinence de la solution et de sa qualité par rapport au contexte d'application.

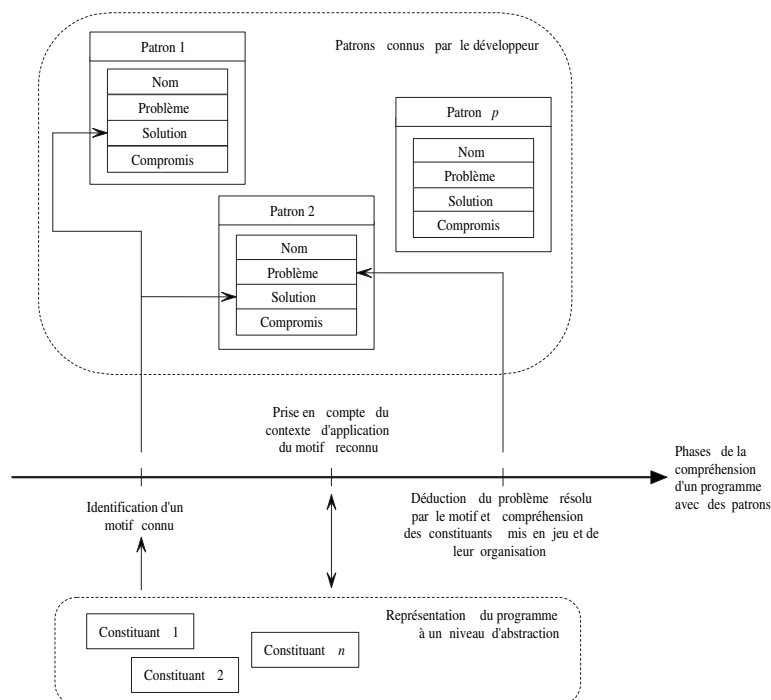
Pour reprendre l'exemple donné dans [Rich et Waters, 1990], cité page 13, lorsqu'un développeur identifie un algorithme et en déduit un temps de calcul, il identifie d'abord un motif proposé par un patron et implanté par un algorithme et il déduit de ce motif le problème résolu par le patron et les propriétés de l'implantation.

Les phases 2 et 3, de contextualisation et de compréhension, dépendent de l'expérience des mainteneurs et du contexte d'utilisation des patrons. Ces phases sont donc difficiles à *automatiser* car le modèle du programme ne contient pas toutes les informations nécessaires. Au contraire, la phase 1, d'identification d'un motif, semble une bonne candidate à l'automatisation : elle ne fait pas intervenir d'éléments extérieurs au modèle du programme, elle est fastidieuse et source d'erreurs.

En plus de l'identification des motifs de conception, les mainteneurs effectuent des *allers-retours*^{*} [Demeyer *et al.*, 1999b ; Niere *et al.*, 2001 ; Booch, 1993, page 517] entre les différents niveaux d'abstraction implémentation, conception et analyse.

Les allers-retours⁷ permettent aux mainteneurs d'obtenir tantôt un modèle concret d'un programme, au niveau **implémentation**, tantôt un modèle abstrait, aux niveaux **conception** et **analyse**, et de s'abstraire des détails d'implantation lors de la rétroconception et de la compréhension du programme.

FIG. 1.2 – Phases de la compréhension avec les patrons.



□

⁷Les allers-retours sont particulièrement intéressants lors du développement incrémental d'un programme à objets [Gabriel, 1996 ; Beck, 1999].

Ces allers-retours sont difficiles à supporter cognitivement car les mainteneurs utilisent leur mémoire à court terme [Lieberman, 1987] pour mémoriser la *continuité* entre les constituants qui composent les différents niveaux d'abstraction.

La continuité entre les constituants des différents niveaux d'abstraction doit être clairement définie. La *traçabilité** [Soukup, 1995 ; Ducasse, 1997b ; Bosch, 1998 ; Antoniol *et al.*, 2001 ; Ho *et al.*, 2002] des constituants et des motifs doit être garantie entre les niveaux d'abstraction pour faciliter les allers-retours.

La traçabilité est importante pour les motifs qui peuvent être explicites à un niveau d'abstraction (constituants uniques) et disséminés dans le modèle du programme à un niveau d'abstraction inférieur (représentés par plusieurs constituants).

Nous décomposons la traçabilité des motifs en quatre phases :

1. Une phase de modélisation du programme dans lequel identifier les motifs ;
2. Une phase de modélisation des motifs à identifier ;
3. Une phase d'identification des micro-architectures similaires au modèle d'un motif dans le modèle du programme. Une micro-architecture peut être :
 - une *forme complète** si les constituants de la micro-architecture et leurs relations sont en tout point similaires aux constituants du motif et à leurs relations ;
 - une *forme approchée** si les constituants et leurs relations ne sont pas en tout point similaires au modèle du motif, par exemple si deux constituants de la micro-architecture ne sont pas liés par la relation définie par le motif.
4. Une phase de modélisation des micro-architectures identifiées. Les modèles des micro-architectures doivent permettre l'aller-retour avec le modèle du programme dans lequel elles ont été identifiées.

Les mainteneurs ont besoin d'outils pour modéliser le programme et les motifs, pour identifier les micro-architectures dans le modèle du programme, formes complètes et formes approchées, et pour garantir la traçabilité des micro-architectures et de leurs constituants entre les niveaux d'abstraction.

Des outils existent pour aider les mainteneurs dans leurs allers-retours entre le niveau *implémentation* et les algorithmes implantés ; par exemple, KBEMACS [Rich et Waters, 1990], qui facilite l'implantation et l'identification d'algorithmes⁸ dans des programmes LISP, et BUNCH [Mancoridis *et al.*, 1998 ; Traverso et Mancoridis, 2002], qui décompose automatiquement des programmes en modules.

Cependant, aucun outil satisfaisant existe pour aider à l'aller-retour entre les constituants et les motifs des niveaux *implémentation* et *conception*. Pourtant, au niveau *conception*, les *patrons de conception** du livre "*Design Patterns – Elements of Reusable Object-Oriented Software*" par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [1994] définissent des problèmes de conception et leurs solutions au niveau *implémentation* et facilitent la compréhension des programmes, qu'ils soient supportés par les langages

⁸Dans leur livre, Charles Rich et Richard C. Waters [1990, page 29] utilisent le mot *plan* pour désigner une notion similaire à la notion de motif.

de programmation ou par des outils de développement et de rétroconception [Agerbo et Cornils, 1998 ; Chambers *et al.*, 2000].

Les définitions données aux patrons de conception laissent une large place à l'interprétation ; cette interprétation a fait leur succès mais limite une utilisation systématique des *motifs de conception** qu'ils proposent comme solutions [Florijn *et al.*, 1997].

Aussi, nous étudions l'identification et la traçabilité des motifs de conception entre les niveaux implémentation et conception pour aider les développeurs à comprendre l'architecture d'un programme lors de la rétroconception et de la compréhension du programme.

1.3 Problème : la traçabilité des motifs interclasses et de conception

LES LANGAGES de programmation utilisés dans la recherche et dans l'industrie, comme JAVA [Seemann et von Gudenberg, 1998 ; Tatsubori, 1999 ; Thimbleby, 1999], n'intègrent pas *directement* les motifs de conception.

Nous étudions l'identification et la traçabilité des motifs de conception *déjà* connus dans la littérature et répertoriés dans des catalogues, tel [Gamma *et al.*, 1994]. Nous ne cherchons pas à résoudre le problème de la découverte de motifs.

La découverte de motifs est l'identification dans l'architecture d'un programme de micro-architectures dont les structures ont suffisamment de points communs pour être abstraites en un motif.

Cette découverte est un problème proche du problème d'apprentissage en intelligence artificielle et elle demande des méthodologies spécifiques [Shull *et al.*, 1996 ; Tonella et Antoniol, 1999].

Aussi, nous ne nous intéressons pas à la composition de motifs [Taibi et Ngo, 2003] ni à son utilisation pour l'identification car ces problèmes sont difficiles et nécessitent d'abord la modélisation et l'identification de motifs indépendants.

1.3.1 Formalismes et niveaux d'abstraction

Les motifs de conception sont décrits par des *diagrammes** qui traitent des différents aspects d'un programme :

- les diagrammes de classes décrivent le modèle global [Jackson et Rinard, 2000] du programme avec ses classes et les relations entre ses classes ;
- les diagrammes d'interactions spécifient les interactions locales [Jackson et Rinard, 2000] entre les instances des classes du programme, les séquences d'appels de méthodes entre instances.

Nous choisissons de modéliser le programme dans lequel nous cherchons à identifier les motifs de conception avec le formalisme utilisé pour les décrire : avec un diagramme de classes.

Les *diagrammes de classes*^{*} sont un formalisme utilisé pendant tout le développement du programme [Tonella et Potrich, 2001]. Ils offrent une vision globale de l'architecture et du comportement du programme.

Ils décrivent les classes et les interfaces du programme et leurs attributs, tels les champs, les méthodes et les relations⁹ d'appel de méthodes, d'héritage¹⁰, d'instanciation, d'association, d'agrégation et de composition.

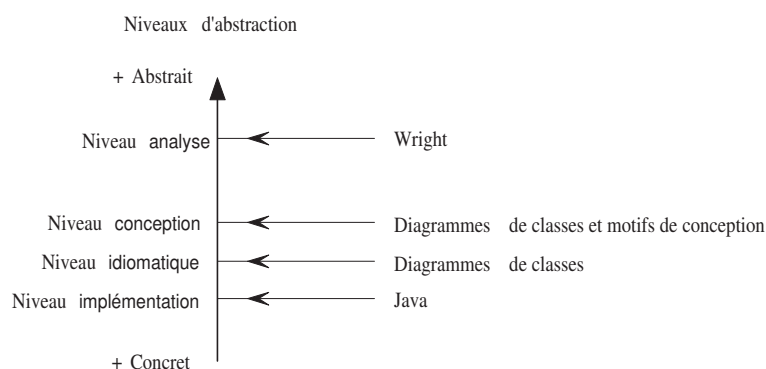
Les motifs de conception sont souvent décrits avec des diagrammes de classes, par exemple les motifs **Composite** et **Médiateur** extraits de [Gamma *et al.*, 1994] et représentés figures 1.7(c) page 25 et 1.4 page 20.

Les diagrammes de classes définissent un niveau d'abstraction intermédiaire entre les niveaux **implémentation** et **conception**. Ce niveau d'abstraction est distinct du niveau **implémentation** car il propose les relations d'association, d'agrégation et de composition in-existantes au niveau **implémentation**. Il est différent du niveau **conception** car il n'offre pas de motifs de conception, nous l'utilisons *pour* décrire les motifs de conception.

Nous nommons ce niveau d'abstraction *niveau idiomatique*^{*} en référence aux idiomes de programmation qui abstraient des constituants du niveau **implémentation** et qui sont des motifs intermédiaires entre les niveaux **implémentation** et **conception**.

Ainsi, le modèle du programme au niveau **idiomatique** est décrit avec un diagramme de classes dont les constituants sont les classes, les interfaces, les méthodes, les champs et les relations entre classes et interfaces, comme présenté sur la figure 1.3.

FIG. 1.3 – Formalismes pour décrire les niveaux implémentation, idiomatique, conception et analyse.



□

⁹Nous considérons uniquement des relations binaires.

¹⁰Nous utilisons le mot *héritage* pour désigner la relation de spécialisation entre classes et la relation d'implantation entre classes et interfaces.

Les classes, les interfaces, les champs et les méthodes sont bien connus et existent dans le programme au niveau **implémentation**. Aussi, les relations d'appel de méthodes, d'héritage et d'instanciation¹¹ sont fondamentales à tous les langages de programmation par objets [Berard, 1990 ; Wolczko, 1992 ; Baumgartner *et al.*, 1996 ; Shull *et al.*, 1996].

Au contraire, les relations d'association, d'agrégation et de composition n'existent pas au niveau **implémentation**¹². Elles forment un ensemble de motifs au niveau **idiomatique** car elles abstraient sous une forme atomique des constituants du niveau **implémentation**, tel les champs, les appels de méthodes, etc. Nous parlons indifféremment de motifs interclasses **Association**, **Agrégation** et **Composition** ou de relations d'association, d'agrégation et de composition.

Alors, l'identification des micro-architectures dont les structures sont similaires à un motif de conception et la traçabilité des micro-architectures identifiées entre les niveaux **implémentation** et **conception** nécessitent l'identification des motifs **Association**, **Agrégation** et **Composition** au niveau **implémentation** pour construire un modèle du programme au niveau **idiomatique** dans lequel nous identifions les motifs de conception pour construire un modèle du programme au niveau **conception**.

1.3.2 Identification des motifs interclasses

Les définitions des motifs interclasses laissent place à l'interprétation et ne précisent aucun choix d'implantation [Henderson-Sellers et Barbier, 1999]. Par exemple, les définitions des motifs interclasses dans UML [Object Management Group, Inc., 2003] se résument par :

- une association est un lien binaire entre exactement deux classes (et possiblement d'une classe vers elle-même) ;
- une agrégation est une relation conceptuelle et ne fait rien de plus que de distinguer un tout d'une partie ;
- une composition est une forme d'agrégation avec une forte appartenance des parties au tout et des durées de vie coïncidentes entre le tout et ses parties.

Ces définitions présentent certaines ambiguïtés au niveau **implémentation** [Lauder et Kent, 1998] : quel est ce *lien* dans le motif d'association ? Quelle est la différence *concrète* entre les motifs **Association** et **Agrégation** ? Quelles sont exactement les propriétés du motif de composition ? Ces questions se résument en “*comment définir et identifier les motifs interclasses dans le modèle d'un programme au niveau implémentation ?*”

¹¹Nous confondons les mécanismes d'appel de méthodes, d'héritage et d'instanciation et les relations de mêmes noms qu'ils supportent. Par exemple, la relation d'appel de méthodes lie l'émetteur d'un message au receveur.

¹²À la rédaction de ce mémoire, les langages de programmation par objets industriels, comme C++, JAVA, et SMALLTALK, n'explicitent pas les notions de relations d'association, d'agrégation et de composition. De futurs langages de programmation pourront intégrer ces notions et ainsi faciliter la modélisation des programmes au niveau **idiomatique**.

L'ambiguïté de ces définitions pénalise les outils de conception industriels et académiques, tels RATIONAL ROSE et le logiciel libre ARGOUML. Ces outils proposent et distinguent graphiquement les trois relations interclasses mais les algorithmes de rétroconception identifient des motifs erronés. (Nous développons ces problèmes dans la section 2.2 page 37.)

1.3.3 Identification des motifs de conception

Les définitions des patrons de conception dans [Gamma *et al.*, 1994] laissent une large place à l'interprétation. Cette interprétation limite une utilisation systématique des motifs qu'ils suggèrent comme solutions [Florijn *et al.*, 1997].

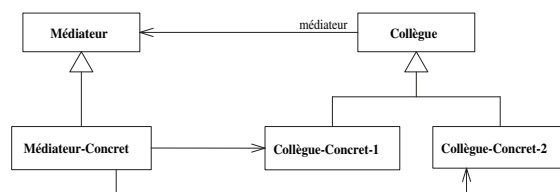
La figure 1.7(c) page 25 montre un modèle au niveau **idiomatique** illustrant le motif de conception **Composite** décrit par un diagramme de classes. Le modèle et la documentation associée au motif n'indiquent pas à quoi correspond la relation entre les classes **Branche** et **Ramification** et les choix induits : si c'est une relation d'agrégation, la présence de la classe **Feuille** est-elle nécessaire ? Si c'est une relation de composition, l'absence de la classe **Feuille** a-t-elle un sens ? Peut-il y avoir plusieurs classes **Feuille** ?

La figure 1.4 montre le diagramme de classes illustrant le motif de conception **Médiateur**. Les classes **Collègue-Concret-*x*** ne doivent pas avoir directement connaissance les unes des autres, cela veut-il dire que le motif de conception *n'est pas* respecté si deux classes ont connaissance l'une de l'autre pour des raisons d'optimisation ? Toutes ces questions se résument en "*comment modéliser et identifier les variantes des motifs de conception ?*"

L'existence de variantes limite le support offert par les outils qui manipulent les motifs de conception [Gall *et al.*, 1996 ; Sunyé, 1999]. Par exemple, l'outil d'IBM pour appliquer des motifs de conception [Budinsky *et al.*, 1996] n'offre qu'un choix limité et statique de valeurs pour la paramétrisation et l'utilisation de motifs. L'outil SOUL [Wuyts, 1998 ; Wuyts *et al.*, 1999] nécessite la définition de toutes les variantes possibles d'un motif de conception avant son identification.

Ainsi, l'identification des motifs de conception doit prendre en compte les variations syntaxiques dans l'implantation des motifs de conception et la non-contiguïté de leurs constituants. Elle doit également expliquer les micro-architectures identifiées [Soloway, 1986]. (Nous développons ces problèmes dans la section 2.3 page 60.)

FIG. 1.4 – Modèle du motif de conception **Médiateur** au niveau **idiomatique** décrit par un diagramme de classes.



□

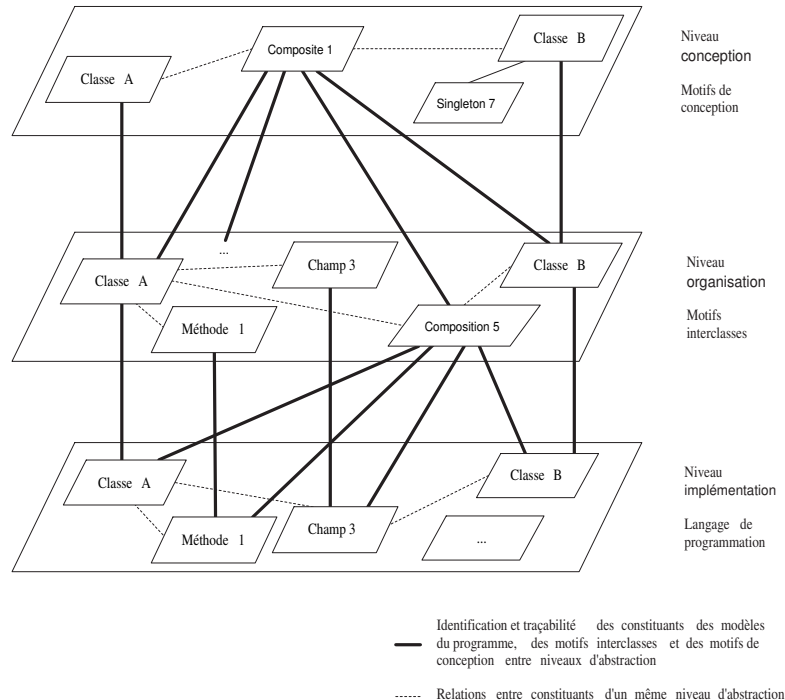
1.3.4 Résumé des problèmes

La figure 1.5 résume les phases de l'identification et de la traçabilité des motifs interclasses et de conception. D'abord, nous analysons le modèle d'un programme au niveau implémentation pour identifier les constituants du modèle qui forment une micro-architecture similaire à un motif interclasse. Par exemple, les constituants **Classe A**, **Méthode 1**, **Champ 3** et **Classe B** forment une micro-architecture similaire au motif interclasse **Composition 5**.

Ensuite, nous analysons le modèle du programme au niveau idiomatique pour identifier les micro-architectures similaires à un motif de conception. Par exemple, les constituants **Classe A**, **Classe B** liés par le motif interclasse **Composition 5** forment une micro-architecture similaire au motif de conception **Composite**.

Les modèles du programme aux niveaux idiomatique et conception lient leurs constituants avec les niveaux implémentation et idiomatique inférieurs, respectivement. Ainsi, la traçabilité est garantie entre les niveaux d'abstraction.

FIG. 1.5 – Identification et traçabilité des relations interclasses et des motifs de conception.



□

1.4 Thèse : la définition des motifs, des analyses de programmes et la programmation par contraintes

Nous soutenons la thèse qu'il est possible de donner des définitions précises aux motifs interclasses et aux motifs de conception pour créer des modèles d'un programme aux niveaux **idiomatique** et **conception** et d'offrir des algorithmes pour construire ces modèles et pour garantir la traçabilité des motifs de conception.

Les définitions des motifs interclasses et des motifs de conception et les modèles du programme nous permettent de résoudre les problèmes de leur traçabilité entre les niveaux **implémentation** et **conception**.

Nous décomposons les définitions des motifs interclasses avec quatre propriétés minimales (durée de vie, exclusivité, multiplicité et site d'invocation). Nous proposons des algorithmes d'analyses statiques et dynamiques des programmes pour identifier les motifs interclasses avec ces propriétés.

Nous montrons la similarité de l'identification des motifs de conception avec un problème de satisfaction de contraintes et nous proposons l'utilisation de la programmation par contraintes avec explications pour identifier les micro-architectures similaires à un motif et pour expliquer leur identification.

Ces modèles et ces algorithmes nous permettent d'implanter des outils qui aident les mainteneurs à effectuer les allers-retours entre les niveaux **implémentation** et **conception** et qui apportent ainsi une aide concrète à la compréhension des programmes à objets et à l'amélioration de leur qualité.

1.5 Scénario : le programme JHOTDRAW et le motif de conception Composite

Nous utilisons le programme JHOTDRAW et le patron de conception structurel **Composite** pour illustrer les problèmes que nous cherchons à résoudre, nos solutions et nos expérimentations.

Le programme JHOTDRAW et le patron de conception **Composite** forment un cas favorable d'étude [Albin-Amiot, 2003, page 189] car le patron de conception **Composite** est utilisé dans l'implantation de JHOTDRAW et son utilisation est documentée par les auteurs du programme.

Nous connaissons donc *a priori* les micro-architectures dont les structures sont similaires au motif de conception **Composite**; son utilisation facilite la compréhension des modèles, des algorithmes et des exemples présentés. Cependant, nos modèles et nos algorithmes restent valables pour d'autres patrons de conception.

JHOTDRAW [Gamma et Eggenschwiler, 1998] est un programme de dessin vectoriel dont l'architecture met en œuvre de nombreux patrons de conception. Ses auteurs originaux, Erich Gamma et Thomas Eggenschwiler, ont commencé son implantation pour

expérimenter l'utilisation des patrons et ont continué son développement devant les succès rencontrés. Les patrons utilisés sont décrits dans la documentation du programme. En particulier, la conception et l'implantation de JHOTDRAW utilise le patron de conception Composite.

Le patron de conception structurel Composite [Gamma *et al.*, 1994, pages 163–173] permet d'organiser des objets en une structure arborescente dans laquelle objets simples et compositions d'objets sont traités uniformément¹³. Il est simple, structurel et sert souvent d'exemple [Sunyé *et al.*, 2000] pour l'évaluation à la fois des systèmes de génération de code [Budinsky *et al.*, 1996] et des systèmes de détection [Brown, 1996 ; Wuyts, 1998]. Dans la conception et l'implantation de JHOTDRAW, il est appliqué à la hiérarchie de classes qui représentent les figures géométriques et les éléments composant ces figures, lignes, textes, rectangles, etc.

La figure 1.6 page 24 montre un extrait du diagramme de classes représentant le programme JHOTDRAW. Ce diagramme de classes est tiré de la documentation¹⁴ du programme et est augmenté par les classes **AbstractFigure**, **AttributeFigure**, **DecoratorFigure**, **PolyLineFigure** et **CompositeFigure** et par le motif de conception Composite.

Un éditeur (instance de **DrawingEditor**) est une fenêtre graphique (sous-classe de **Frame**) qui contient un ensemble de dessins (agrégation d'instances de **Drawing**) et des vues sur ces dessins (agrégation d'instances de **DrawingView**, sous-classes de **Panel**, créées avec chaque dessin). Un éditeur offre un ensemble d'outils pour manipuler les dessins (agrégation d'instances de **Tool**). Chaque dessin est composé de figures (agrégation d'instances de **Figure**). Une figure peut être un rectangle, une ligne, une ligne multiple (instance de **PolyLineFigure**) ou encore une figure avec des attributs (instance de **AttributeFigure**). Une figure peut aussi être une figure décorée (instance de **DecoratorFigure**) ou une figure composée d'autres figures (instances de **CompositeFigure**).

Ainsi, le diagramme de classes montre les classes principales du programme, les paquets auxquels elles appartiennent, les relations entre ces classes et les classes appartenant à une micro-architecture similaire au motif de conception Composite : la bulle Composite représente cette micro-architecture, les lignes en pointillés indiquent les classes du programme jouant un rôle dans ce motif, à rapprocher du modèle du motif montré sur la figure 1.7(c) page 25 :

- la classe **CompositeFigure** joue le rôle de Branche;
- l'interface **Figure** joue le rôle de Ramification;
- les classes **AttributeFigure**, **DecoratorFigure**, **PolyLineFigure** et leurs sous-classes jouant le rôle de Feuille.

¹³Objets simples et compositions d'objets répondent aux mêmes appels de méthodes.

¹⁴La documentation de JHOTDRAW est disponible à <http://jhotdraw.sourceforge.net/online-docs/documentation/>.

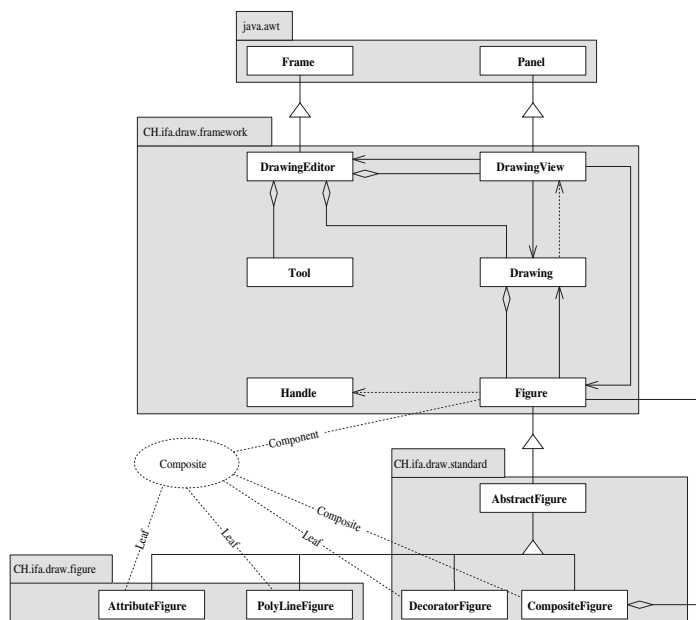
Les figures 1.7(a) à 1.7(d) page 25 illustrent nos motivations sur l'exemple de JHOT-DRAW avec le patron de conception Composite :

- la figure 1.7(a) représente le modèle du programme décrit au niveau implémentation avec le langage de programmation JAVA ;
- la figure 1.7(b) représente un sous-ensemble du modèle de ce programme au niveau idiomatique, décrit par un diagramme de classes, tel que proposé par ses auteurs, et augmenté de classes implantant l'interface **Figure** ;
- la figure 1.7(c) représente un modèle du motif de conception Composite, au niveau idiomatique, décrit par un diagramme de classes ;
- enfin, la figure 1.7(d) représente un modèle du programme et une micro-architecture similaire au motif de conception Composite, au niveau conception.

Nous voulons construire automatiquement le modèle du programme au niveau idiomatique figure 1.7(b) avec son modèle au niveau implémentation figure 1.7(a), les définitions des motifs interclasses au niveau implémentation et des algorithmes d'analyses statiques et dynamiques.

Puis, nous voulons construire semi-automatiquement le modèle du programme au niveau conception figure 1.7(d) avec son modèle au niveau idiomatique figure 1.7(b), le modèle du motif de conception Composite figure 1.7(c) au niveau idiomatique et la programmation par contraintes avec explications.

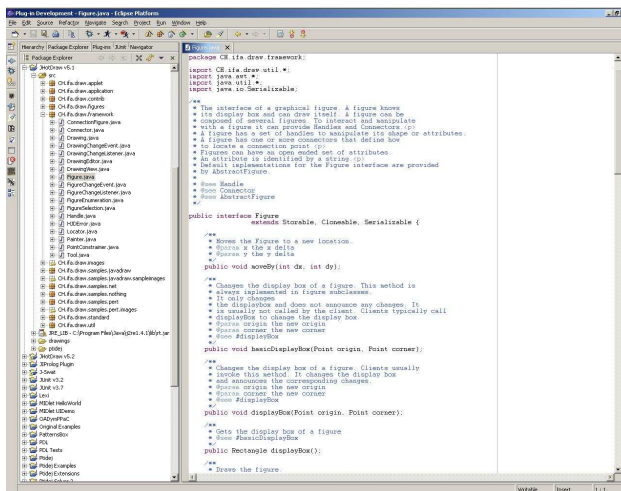
FIG. 1.6 – Modèle de JHOTDRAW au niveau conception, le diagramme de collaboration représente le motif de conception Composite.



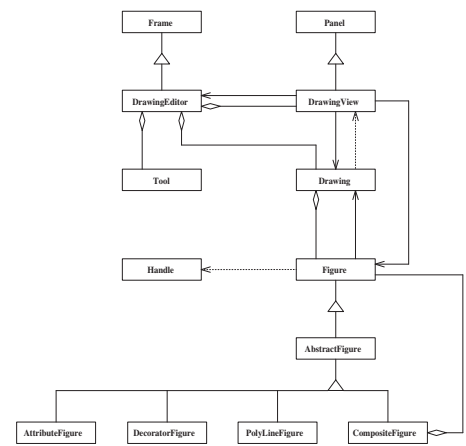
□

FIG. 1.7 – Modèles du programme JHOTDRAW : du niveau implémentation au niveau idiomatique; du niveau idiomatique au niveau conception.

(a) Modèle de JHOTDRAW au niveau implémentation :



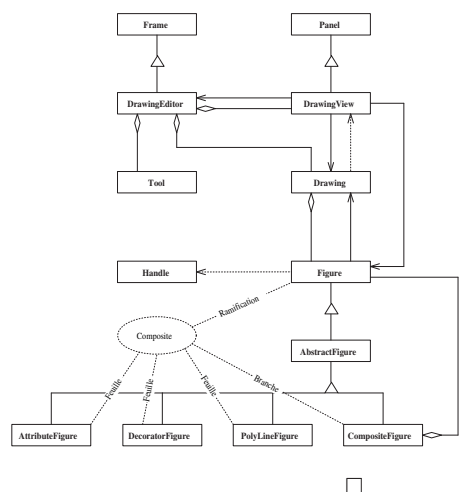
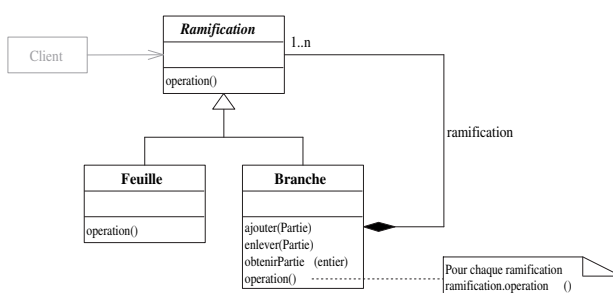
(b) Modèle de JHOTDRAW au niveau idiomatique :



(d) Modèle de JHOTDRAW au niveau conception.

La bulle **Composite** représente une micro-architecture similaire au motif de conception Composite, les lignes en pointillés lient la micro-architecture avec les classes y jouant un rôle :

(c) Modèle du motif de conception Composite au niveau idiomatique :



1.6 Contribution : un cadre pour la traçabilité des motifs de conception

LES modèles et les algorithmes proposés dans ce mémoire forment un cadre¹⁵ pour la traçabilité des motifs de conception. Ce cadre doit nous permettre d'étudier et de qualifier les bénéfices de la traçabilité des motifs de conception pour faciliter la compréhension des programmes.

Nous décomposons la réalisation de ce cadre en deux parties : une première partie sur les modèles et les algorithmes pour garantir la traçabilité des motifs interclasses et de conception ; une seconde partie d'implantation de ces modèles et de ces algorithmes. Parties et chapitres sont introduits par un rappel des problèmes à traiter et sont conclus par un bilan des solutions décrites.

Nous nous intéressons aux motifs de conception structuraux car ceux-ci sont mieux connus et plus simples à modéliser que les motifs comportementaux et générateurs. Cependant, nos travaux se généralisent aux motifs comportementaux et générateurs, comme nous le montrons dans les perspectives, page 279.

1.6.1 État de l'art sur la traçabilité des motifs

D'abord, nous présentons un état de l'art des techniques pour la traçabilité des motifs interclasses et de conception (chapitre 2 page 31). Cet état de l'art s'organise autour d'une classification des techniques existantes de rétroconception et de la qualité de leurs résultats. Il souligne les principales limitations des techniques actuelles et le besoin de modèles et d'algorithmes pour garantir la traçabilité des motifs de conception.

1.6.2 Motifs interclasses

Ensuite, nous proposons des modèles et des algorithmes pour garantir la traçabilité des motifs interclasses (partie II page 75, chapitre 3 page 77). Nous présentons nos travaux pour décrire les motifs interclasses par la métamodélisation. Nous proposons un métamodèle pour décrire le niveau **idiomatique**, nous intégrons dans ce métamodèle nos définitions des motifs interclasses aux niveaux **implémentation** et **idiomatique** (section 3.1 page 79).

Les définitions des motifs interclasses au niveau **implémentation** sont basées sur quatre propriétés des motifs : durée de vie, exclusivité, multiplicité et site d'invocation. Nous montrons que ces quatre propriétés forment un ensemble minimal de propriétés pour définir les motifs interclasses (sections 3.2 page 84, 3.3 page 92 et 3.4 page 101).

Nous décrivons des algorithmes d'analyses statiques et dynamiques pour calculer les valeurs des quatre propriétés et ainsi identifier les relations interclasses. Nous présentons aussi des algorithmes pour garantir la traçabilité des motifs interclasses identifiés (sections 3.5 page 116 et 3.6 page 122).

¹⁵Un cadre s'appelle *framework* en anglais [Office québécois de la langue française, 2003].

Nous utilisons trois programmes JAVA connus : JHOTDRAW [Gamma et Eggenschwiler, 1998], JUNIT [Gamma et Beck, 1998] et JAVA AWT [Sun Microsystems, Inc., 2002] pour valider nos résultats (section 3.7 page 126).

Nous concluons avec l'utilisation de nos modèles et de nos algorithmes pour construire un modèle du programme JHOTDRAW au niveau **idiomatique** avec son modèle au niveau **implémentation** (section 3.8 page 136).

1.6.3 Motifs de conception

Puis, nous proposons des modèles et des algorithmes pour garantir la traçabilité des motifs de conception (chapitre 4 page 143). Nous présentons nos travaux pour décrire les motifs de conception par la métamodélisation.

Nous proposons un métamodèle pour décrire le niveau **conception**, nous intégrons dans ce métamodèle les micro-architectures similaires à des motifs de conception (section 4.1 page 145).

Nous décrivons aussi un métamodèle pour modéliser les motifs de conception structuraux et nous montrons que le modèle d'un motif de conception se traduit en un système de contraintes et que l'identification des micro-architectures similaires à un motif de conception se traduit en un problème de satisfaction de contraintes (section 4.2 page 148).

Nous faisons des rappels sur la programmation par contraintes et nous montrons que la programmation par contraintes avec explications permet de justifier les micro-architectures identifiées et facilite l'interaction avec les mainteneurs dans la recherche des variantes d'un motif (section 4.3 page 155).

Nous proposons alors des contraintes pour la modélisation des motifs de conception comme des systèmes de contraintes. Nous détaillons deux stratégies de recherche pour l'identification des motifs et des algorithmes de traçabilité des motifs (sections 4.4 page 166, 4.5 page 169 et 4.6 page 176).

Nous vérifions la cohérence de nos modèles et de nos algorithmes et nous concluons avec leur utilisation pour construire un modèle du programme JHOTDRAW au niveau **conception** avec les micro-architectures similaires au motif de conception **Composite** (sections 4.7 page 178 et 4.8 page 184).

1.6.4 Mise en œuvre de la traçabilité des motifs

Nous étendons (partie III page 195) les travaux réalisés dans une précédente thèse de doctorat [Albin-Amiot, 2003] sur la formalisation des motifs de conception structuraux, sur un métamodèle, *Pattern Description Language* (PDL), pour décrire les motifs de conception.

Nous raffinons le métamodèle PDL et nous développons PADL, un métamodèle pour décrire les modèles d'un programme aux niveaux **implémentation**, **idiomatique** et **conception** et les motifs de conception (chapitre 5 page 197).

Nous développons des outils d'analyses statiques et dynamiques, INTROSPECTOR et CAFFEINE, pour identifier les motifs interclasses dans le modèle d'un programme au niveau **implémentation** et des algorithmes pour construire son modèle au niveau **idiomatique** et pour garantir la traçabilité des motifs interclasses (chapitre 6 page 211).

Nous présentons l'implantation de référence de la programmation par contraintes avec explications, PALM, et nous détaillons notre extension pour identifier les micro-architectures similaires à des motifs de conception, PTIDEJ SOLVER.

Nous décrivons aussi l'implantation des contraintes dédiées à l'identification des motifs de conception, PTIDEJ LIBRARY, et des différentes stratégies de recherche (chapitre 7 page 233).

Le métamodèle PADL est utilisé pour modéliser un motif de conception dont les micro-architectures similaires sont identifiées dans le modèle d'un programme au niveau **idiomatique** avec les solveurs de contraintes avec explications PTIDEJ SOLVER.

Nous présentons alors PTIDEJ, un outil intégré à l'environnement de développement intégré (*EDI**) ECLIPSE [Object Technology International, Inc. / IBM, 2001] pour garantir la traçabilité des motifs de conception entre les modèles d'un programme aux niveaux **implémentation** et **conception** (chapitre 8 page 253).

La figure 1.9 page 30 montre l'outil PTIDEJ intégré à l'EDI ECLIPSE. L'éditeur présente le modèle au niveau **conception** d'un sous-ensemble des classes de JHOTDRAW, représenté avec une notation proche de UML. Une micro-architecture similaire au motif de conception **Composite** est mise en valeur.

Ce modèle et les informations qu'il fournit sont à rapprocher du modèle de JHOTDRAW présenté sur la figure 1.6 page 24 et rappelé sur la figure 1.8 page 29. Ces deux modèles présentent essentiellement les mêmes informations, seule la représentation visuelle des informations est différente.

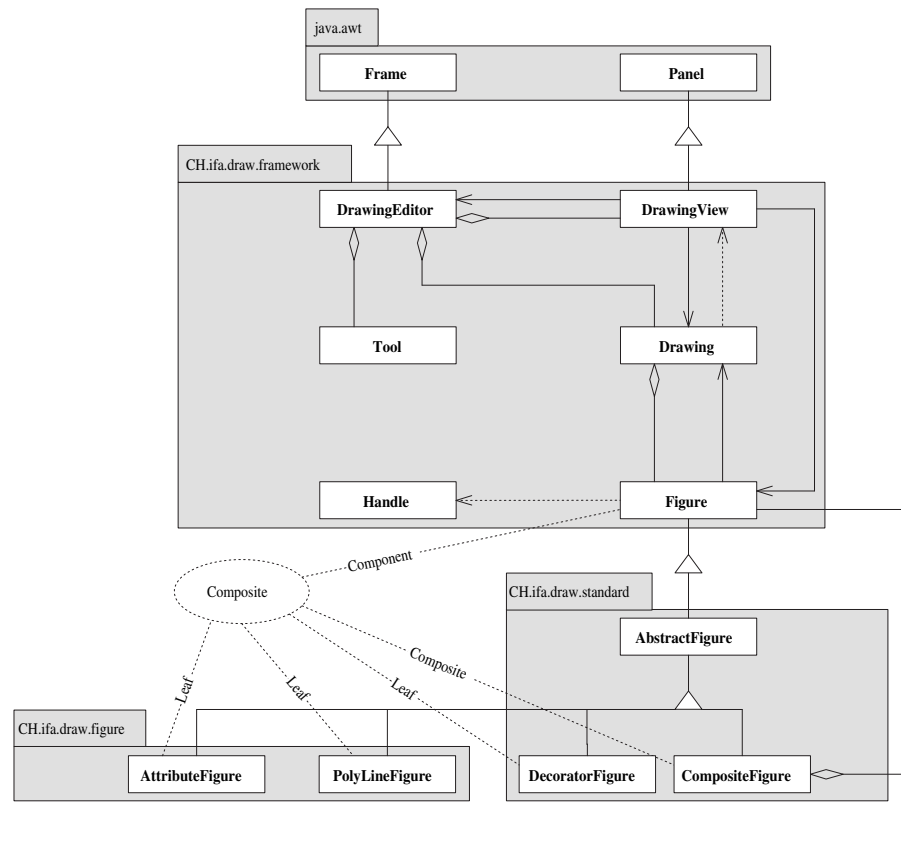
Le modèle présenté sur la figure 1.9 page 30 a été obtenu semi-automatiquement avec PTIDEJ, alors que le modèle présenté sur la figure 1.6 page 24 a été obtenu après une analyse manuelle du programme.

Ainsi, nous apportons un support à la traçabilité des motifs de conception entre les niveaux **implémentation** et **conception**. Ce support aide les mainteneurs à comprendre les programmes en maintenance et à améliorer la qualité de leur implantation.

Les modifications sont effectuées avec une meilleure compréhension des relations entre les constituants des programmes, elles sont plus pertinentes car produites avec des modèles plus fidèles de l'architecture des programmes.

Ces modèles sont obtenus semi-automatiquement pour libérer la mémoire à court terme des mainteneurs et pour rendre cohérente leur compréhension, quelles que soient leurs connaissances et leurs expériences.

FIG. 1.8 – Modèle de JHOTDRAW au niveau conception, le diagramme de collaboration représente le motif de conception Composite.



Chapitre 2

État de l’art sur la traçabilité des motifs

CE CHAPITRE présente un état de l’art des techniques pour l’identification et la traçabilité des motifs interclasses et de conception. Nous ne présentons pas un état de l’art de la modélisation et de l’identification des formes complètes des motifs de conception car un tel état de l’art a été récemment proposé dans un autre mémoire de thèse de doctorat [Albin-Amiot, 2003].

D’abord, nous présentons un cadre pour classer les techniques de rétroconception et qualifier leurs résultats. Ce cadre nous permet de comparer entre elles les techniques de rétroconception et les qualités de leurs résultats.

Ensuite, nous décomposons cet état de l’art en deux sections : la première section décrit les principales techniques existantes pour identifier les motifs interclasses et pour garantir leur traçabilité entre les niveaux `implémentation` et `idiomatique` ; la seconde section décrit les techniques pour l’identification des motifs de conception et pour leur traçabilité entre les niveaux `idiomatique` et `conception`.

2.1 Classification des techniques de rétroconception

DE NOMBREUX TRAVAUX ont présenté des techniques (modèles, algorithmes, outils) pour la rétroconception des programmes et des classifications existent pour les évaluer et les comparer [Biggerstaff *et al.*, 1993 ; Bellay et Gall, 1997 ; Gannod et Cheng, 1999].

Ces classifications nous permettent de comparer les techniques existantes avec nos travaux sur l'identification et la traçabilité des motifs interclasses et de conception, même si elles prennent en compte de façon très limitée le support cognitif fourni aux mainteneurs [Walenstein, 2002].

Nous utilisons la classification proposée par Gerald C. Gannod et Betty H. Cheng [1999] car elle permet de classer simplement les techniques existantes et de qualifier et comparer la qualité sémantique de leurs résultats.

La classification se décompose en une taxonomie des techniques de rétroconception et en une taxonomie des dimensions de la qualité sémantique des résultats de la rétroconception. Les techniques de rétroconception peuvent être basées sur des techniques *formelles* ou *informelles*, utiliser *l'identification de clichés*, des *analyses syntaxiques*, des *transformations* ou des *translations* et être implantées dans des outils *industriels* ou *académiques*, comme montré sur la figure 2.1 page 33 extraite de [Gannod et Cheng, 1999].

La qualité sémantique du résultat d'une technique de rétroconception (diagramme, métrique, etc.) mesure son habilité à transmettre des informations de haut niveau. Elle se décompose en quatre dimensions sémantiques qui sont :

- la *distance sémantique*, qui décrit le nombre de niveaux d'abstraction entre les informations fournies et celles produites ;
- la *précision sémantique*, qui décrit la précision des informations produites par rapport à celles fournies ;
- le *niveau de détails sémantiques*, qui décrit le degré de formalisation des informations produites ;
- la *continuité sémantique*¹, qui décrit le degré avec lequel les informations produites permettent de reconstruire un programme équivalent à celui qui est analysé.

2.1.1 Taxonomie des techniques de rétroconception

La différence principale entre les techniques formelles et informelles est l'utilisation par les premières de langages de spécifications formelles avec des syntaxes et des sémantiques bien définies.

Au contraire, les techniques informelles sont basées sur l'identification de clichés et sur des analyses syntaxiques. Elles permettent l'obtention d'abstractions structurelles et fonctionnelles du code source des programmes.

¹Les auteurs utilisent le terme *semantic traceability* pour nommer cette dimension. Nous évitons la traduction *traçabilité sémantique* qui serait ambiguë avec la notion de traçabilité des motifs.

L'identification de clichés et les analyses syntaxiques sont considérées comme des techniques informelles car leurs résultats sont des modèles informels, elles ne permettent pas de vérifier rigoureusement la cohérence entre le code source et ces modèles informels.

L'identification de clichés utilisent des descriptions d'unités de calculs qui réalisent une fonction abstraite. Un cliché peut être local ou réparti, les instructions du code source satisfaisant ce cliché peuvent être contiguës ou disséminées dans le code source.

Les analyses syntaxiques utilisent les propriétés de la syntaxe des langages de programmation. Elles permettent de construire une abstraction de plus haut niveau que le code source, diagrammes de flots de données, représentations graphiques de l'architecture du programme.

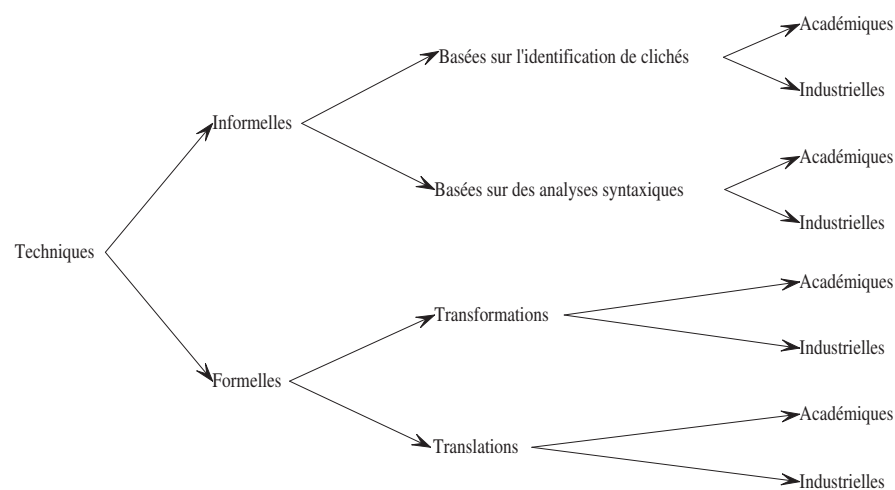
Les techniques formelles utilisent des méthodes analytiques de transformation ou de translation pour construire une spécification formelle du code source d'un programme. Elles permettent de dériver des abstractions fonctionnelles du code source.

La transformation et la translation sont considérées comme des techniques formelles car leurs résultats sont des modèles formels, décrits avec des langages de spécifications formelles, dont l'obtention peut être rigoureusement vérifiée.

Une transformation convertit le code source d'un programme en une spécification qui préserve la sémantique originale du programme. Elle convertit des ensembles d'instructions du code source en instructions du langage de spécifications formelles.

Une translation transforme une à une les instructions du code source d'un programme en leur équivalent dans un langage de spécifications formelles en préservant la sémantique du programme original.

FIG. 2.1 – Taxonomie des techniques de rétroconception extraite de [Gannod et Cheng, 1999].



□

Ces techniques, informelles ou formelles, basées sur l'identification de clichés ou des analyses syntaxiques, utilisant des transformations et des translations, sont issus du monde industriel ou de la recherche académique.

2.1.2 Qualité des résultats de la rétroconception

Les quatre dimensions sémantiques permettent aux mainteneurs d'évaluer les techniques de rétroconception et la qualité de leurs résultats, le niveau de cohérence entre les résultats obtenus par la rétroconception et le code source fourni.

La distance sémantique décrit le nombre de niveaux d'abstraction entre les informations fournies et celles produites par une technique de rétroconception. La distance sémantique est relative car aucune mesure raisonnable de l'abstraction d'un modèle existe à notre connaissance.

Par exemple, la distance sémantique entre le code source d'un programme décrit avec le langage de programmation JAVA et le code source du même programme en C++ est faible, voire nulle ; la distance sémantique entre le code source d'un programme et la description des notions existantes dans ce programme aux niveaux *idiomatique*, *conception* ou *analyse* est grande, comme montré sur la figure 2.2(a) page 35.

La précision sémantique décrit le degré de cohérence et de correction entre le résultat de la rétroconception et le code source utilisé pour l'obtenir. La précision sémantique peut être faible, moyenne ou grande.

Par exemple, les résultats obtenus par des analyses syntaxiques ont une grande précision sémantique ; les résultats obtenus par identification de clichés peuvent avoir une précision sémantique faible suivant les hypothèses faites sur les clichés et les techniques d'identification utilisées, comme montré sur la figure 2.2(b) page 35.

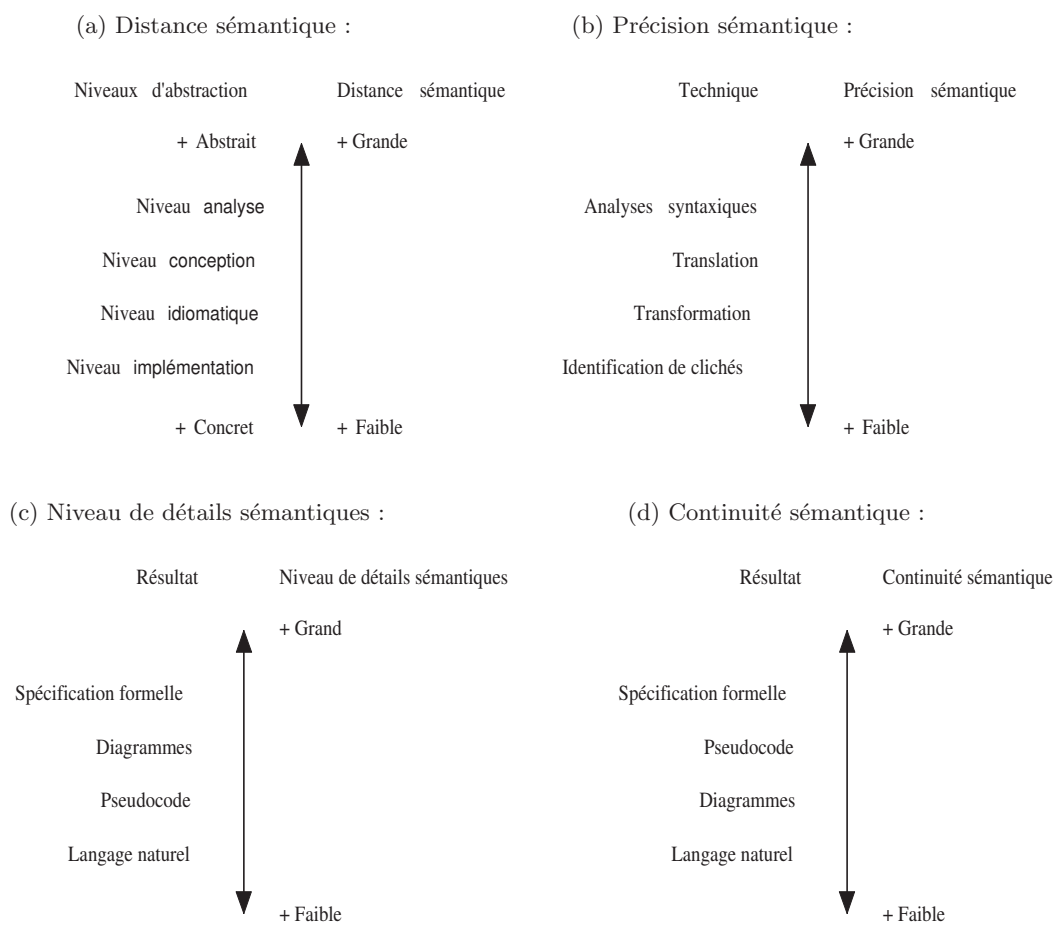
Le niveau de détails sémantiques décrit le degré de formalité du résultat de la rétroconception. Un résultat avec un grand niveau de détails sémantiques peut être utilisé pour réaliser des analyses automatiques, un résultat avec un faible niveau de détails sémantiques facilite la communication entre mainteneurs.

Par exemple, une spécification formelle possède un grand niveau de détails sémantiques ; le langage naturel contient un faible niveau de détails sémantiques par l'existence possible d'ambiguïtés, comme montré sur la figure 2.2(c) page 35.

La continuité sémantique représente le degré avec lequel le résultat de la rétroconception peut être utilisé par des développeurs pour reconstruire un programme équivalent au programme analysé. Elle décrit le degré de correspondance sémantique entre le programme analysé et le nouveau programme dérivé du résultat de la rétroconception.

Par exemple, une spécification formelle décrit les fonctions d'un programme et facilite ainsi la reconstruction d'un programme équivalent ; des diagrammes contiennent, en général, uniquement des informations syntaxiques qui offrent une aide limitée pour la reconstruction d'un programme équivalent, comme montré sur la figure 2.2(d) page 35.

FIG. 2.2 – Dimensions sémantiques de la qualité des résultats de la rétroconception.



□

2.1.3 Discussion

Cette classification en deux parties, une taxonomie des techniques de rétroconception et une taxonomie des dimensions de la qualité sémantique de leurs résultats, nous permet de classer et de qualifier les techniques d'identification et de traçabilité des motifs interclasses et de conception.

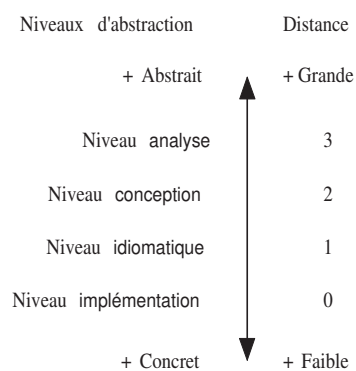
La qualification des techniques d'identification des motifs nous aide à montrer les problèmes des techniques existantes auxquelles nos travaux apportent des solutions. Elle nous permet aussi de comparer nos travaux aux techniques existantes.

En particulier, nous pouvons qualifier et comparer les techniques existantes et nos travaux par la distance sémantique de leurs résultats, par les niveaux d'abstraction auxquels leurs résultats appartiennent.

Alors, nous quantifions la distance sémantique des résultats par rapport à leur niveau d'abstraction : si le résultat est au niveau **implémentation**, la distance sémantique est nulle ; si le résultat est au niveau **idiomatique**, la distance sémantique est égale à un ; si le résultat est au niveau **conception**, la distance sémantique est égale à deux ; enfin, si le résultat est au niveau **analyse**, la distance sémantique est égale à trois, comme montré sur la figure 2.3.

Conclusion. Nous utilisons la classification des techniques de rétroconception et de leurs résultats proposée dans [Gannod et Cheng, 1999] pour classer et comparer les techniques de rétroconception des motifs interclasses et de conception. Nous présentons maintenant les techniques existantes de rétroconception des motifs interclasses.

FIG. 2.3 – Quantification de la distance sémantique.



□

2.2 Motifs interclasses

L'IDENTIFICATION des motifs interclasses consiste à analyser les modèles des programmes au niveau **implémentation** pour identifier les constituants dont la présence indique l'existence de ces motifs.

Nous présentons trois types de techniques pour identifier ces motifs et nous montrons qu'elles sont limitées par l'absence de définitions précises et consensuelles des relations d'association, d'agrégation et de composition, dont nous dressons un état de l'art.

Nous concluons par une discussion sur ces techniques d'identification des motifs interclasses et le besoin en algorithmes qui garantissent leur traçabilité entre les niveaux **implémentation** et **idiomatique**.

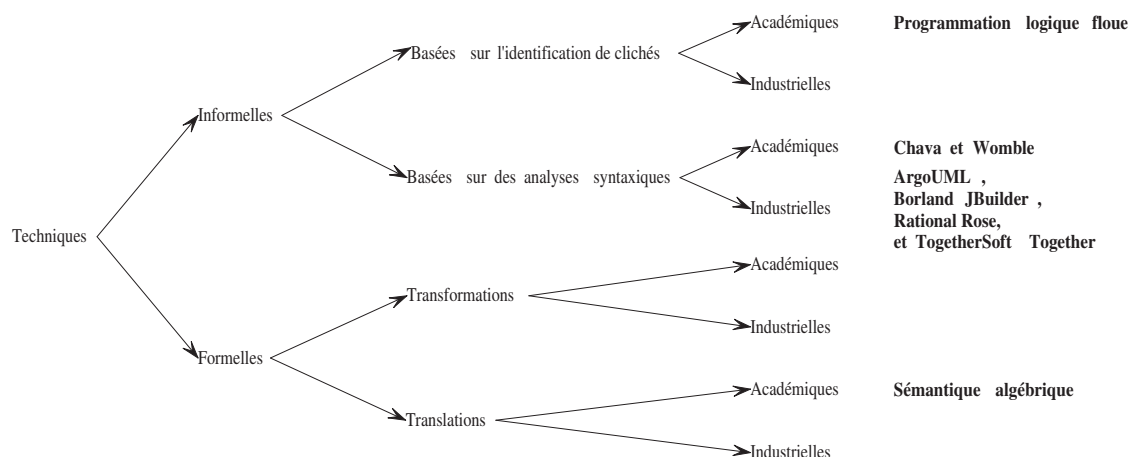
2.2.1 Techniques d'identification

Nous présentons, comme montré sur la figure 2.4 :

- une technique basée sur l'identification de clichés qui utilise la programmation logique floue ;
- des techniques basées sur des analyses syntaxiques académiques, avec CHAVA et WOMBLE, et industrielles, avec ARGOUML, BORLAND JBUILDER, RATIONAL ROSE et TOGETHERSOFT TOGETHER ;
- une technique par translation pour la spécification algébrique de UML.

Ces trois types de techniques sont représentatives des techniques existantes d'identification des motifs interclasses car nous avons été dans l'impossibilité de trouver une technique formelle basée sur des transformations.

FIG. 2.4 – Techniques de rétroconception des motifs interclasses.



Technique basée sur l'identification de clichés Plusieurs auteurs ont étudié l'utilisation de la logique floue et de réseaux génériques de raisonnement flou² pour identifier des micro-architectures similaires à des motifs interclasses [Jahnke *et al.*, 1997 ; Niere *et al.*, 2001 ; Niere, 2002].

Par exemple, Jörg Niere *et al.* [2001] utilisent des réseaux de raisonnement flou pour extraire les relations d'association entre classes. Ils identifient dans le code source de programmes des clichés d'accès en lecture et en écriture aux champs des classes pour en déduire des relations plus abstraites.

Ces relations plus abstraites sont, par exemple, les relations “lien de lecture vers un objet”, “lien de lecture vers un objet qualifié” ou “itération sur un ensemble d'objets”, dont les auteurs proposent une taxonomie.

Les clichés d'accès en lecture et en écriture aux champs varient suivant l'expérience et les préférences des mainteneurs. Ils sont associés avec un réseau de raisonnement flou pour les identifier avec une certaine marge de confiance et pour éviter la définition de clichés spécifiques à chaque variante des relations.

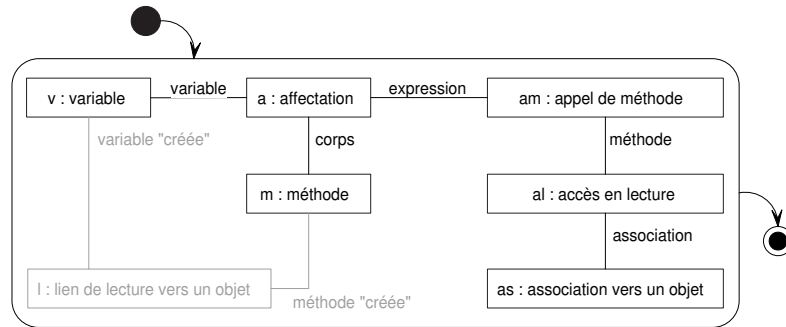
Par exemple, l'extrait suivant du code source de JHOTDRAW correspond au cliché “lien de lecture vers un objet” défini par le diagramme de collaboration figure 2.5 :

```
public abstract class AbstractFigure implements Figure {
    public void invalidate() {
        if (fListener != null) {
            Rectangle r = displayBox();
            ...
        }
    }
}
```

La variable **Rectangle** **r** (v) est affectée (a) par un appel à la méthode **displayBox()** (am). L'appel à la méthode **displayBox()** (am) est réalisé par un accès en lecture (al) à un objet (as). L'affectation est effectuée dans le corps de la méthode **invalidate()** (m). Un lien de lecture vers un objet de type **Rectangle** (v) est ainsi créé par le corps de la méthode **invalidate()** (m).

²Les réseaux génériques de raisonnement flou s'appellent des *generic fuzzy reasoning nets* en anglais.

FIG. 2.5 – Diagramme de collaboration représentant le cliché d’identification de liens de lecture vers un objet.



□

L'extrait suivant du code source de JHOTDRAW correspond au cliché "itération sur un ensemble d'objets" défini par le diagramme de collaboration figure 2.6 page 41 :

```
public class PolyLineFigure extends AbstractFigure {
    public void write(StorableOutput dw) {
        ...
        Enumeration k = fPoints.elements();
        while (k.hasMoreElements()) {
            Point p = (Point) k.nextElement();
            ...
        }
    }
}
```

La variable **Enumeration** *k* (*vi*) est affectée (*a2*) par un appel à la méthode **elements()** (*am2*). L'appel à la méthode **elements()** (*am2*) est réalisé par un accès en lecture (*al*) à l'objet **fPoints** (*as*). La variable **Point** *p* (*v*) est affectée (*a1*) par un appel à la méthode **nextElement()** (*am1*). L'appel à la méthode **nextElement()** (*am1*) est réalisé par un accès en lecture (*ai*) à l'itérateur **Enumeration** *k* (*vi*). L'affectation est effectuée dans le corps de la méthode **write()** (*m*). Une itération sur un ensemble d'objets de type **Point** (*v*) est ainsi créée par le corps de la méthode **write()** (*m*).

Ces deux extraits de code source ont des arbres de syntaxe abstraite différents mais les clichés qui les caractérisent utilisent les mêmes indicateurs : une variable (*v* ou *vi*), une affectation (*a* ou *a2*), un appel de méthode (*am* ou *am2*) et un accès en lecture (*al*). Les clichés peuvent être abstraits en un cliché plus général auquel est associé une valeur de confiance, comme présenté sur la figure 2.7 page 41.

L'identification de clichés est alors réalisée en quatre phases :

1. La décomposition des clichés en clichés flous.
2. La traduction des clichés flous en réseaux de raisonnement flou auxquels sont associés des valeurs de confiance.
3. Le regroupement de ces réseaux de raisonnement flou en plusieurs réseaux représentant les clichés à identifier.
4. La recherche de ces réseaux de raisonnement flou dans les arbres de syntaxe abstraite

des programmes, étendus par des annotations “variables”, “affectations”, “appels de méthodes”, “accès en lecture”, “association vers un objet”, etc.

Les résultats de la recherche sont de nouvelles annotations entre les nœuds des arbres de syntaxe abstraite : “liens de lecture vers un objet”, “itérations sur des ensembles d’objets”, etc. auxquelles sont associées des valeurs de confiance.

Ainsi, les auteurs apportent une solution au problème de l’identification de variantes de l’implantation des relations interclasses par l’utilisation d’un cliché général auquel est associé une valeur de confiance.

La qualité sémantique des résultats obtenus a une distance sémantique comprise entre zéro et un, une précision sémantique faible, un grand niveau de détails sémantiques et une continuité sémantique moyenne.

En particulier, la distance sémantique est comprise entre zéro et un car l'identification des relations d'association, d'agrégation et de composition utilise une valeur de confiance : le modèle du programme obtenu reflète le modèle "réel" du programme au niveau idiomatique dans la limite de cette valeur de confiance.

FIG. 2.6 – Diagramme de collaboration représentant le cliché d'identification d'itération sur des ensembles d'objets.

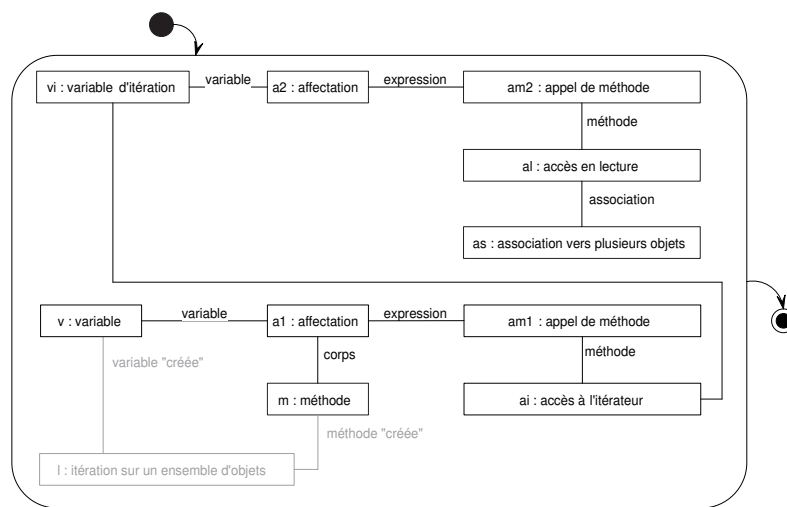
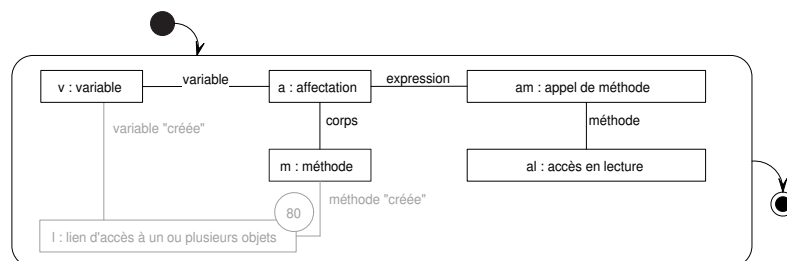


FIG. 2.7 – Diagramme de collaboration représentant le cliché flou d'identification de liens d'accès à un ou plusieurs objets.



Techniques académiques basées sur des analyses syntaxiques CHAVA³ [Korn *et al.*, 1999] est un outil de rétroconception et de suivi de l'évolution de programmes JAVA, dédié à l'analyse de pages Web et d'appliquettes JAVA.

Il construit avec le code source ou le *code octal** d'un programme un référentiel qui représente le programme et qui permet aux mainteneurs d'examiner sa structure et les interactions entre ses classes.

Le référentiel contient un ensemble d'entités. Une entité existe pour chaque construction du langage de programmation. Le référentiel est complet s'il existe une relation entre une entité A du programme et une entité B dans le référentiel lorsque la compilation de A dépend de B.

Une entité peut être une classe (une collection de champs et de méthodes), une interface (une collection de champs statiques et de méthodes abstraites), un paquetage (une collection de classes et d'interfaces), un fichier (un code source déclarant une ou plusieurs classes et interfaces), une méthode, un champ, une chaîne de caractères référencée par des champs ou des méthodes.

Une entité peut avoir des attributs qui enrichissent sa déclaration : *id* (un identifiant unique de l'entité dans le référentiel), nom, type, fichier, ligne de début, ligne de fin, *cksum* (un identifiant unique de l'entité entre référentiels), parent, portée, visibilité, paramètres, type de donnée (type d'un champ ou type de retour d'une méthode).

Le référentiel contient aussi des relations entre entités : sous-classe (relation de spécialisation entre classes) ; implantation (relation d'implantation entre une classe et une interface) ; appartenance (relation entre les champs, les méthodes et les entités les déclarant) ; lecture d'un champ (relation entre une champ et une méthode dans laquelle le champ est lu) ; écriture d'un champ (relations entre un champ et une méthode dans laquelle le champ est modifié) ; référence (relation entre deux méthodes, l'une appelant l'autre).

Ainsi, un référentiel contient un modèle d'un programme JAVA représentant fidèlement son code octal. Les relations entre entités sont uniquement celles qui existent *explicitement* dans le code octal du programme : sous-classe, implantation et appartenance (structure `ClassFile`) ; lecture et écriture d'un champ (codes octaux `getfield`, `putfield`, `getstatic` et `putstatic`) ; appel de méthode (codes octaux `invokevirtual`, `invokeinterface`, `invokespecial` et `invokestatic`) [Lindholm et Yellin, 1999, pages 83, 79 et 80, respectivement]. CHAVA ne prend pas en compte les relations d'association, d'agrégation et de composition.

La qualité sémantique du modèle d'un programme obtenu avec CHAVA a une distance sémantique nulle, une grande précision sémantique, un grand niveau de détails sémantiques et une grande continuité sémantique.

La distance sémantique est nulle car le modèle du programme est similaire au code source ou au code octal du programme, au niveau implémentation, et il ne distingue pas les relations d'association, d'agrégation et de composition existantes.

³Nous utilisons la version de CHAVA d'octobre 1999.

WOMBLE⁴ [Jackson et Waingold, 1999] est un outil de rétroconception pour l'extraction de modèles à objets du code octal de programmes JAVA. Un modèle à objets est un graphe dont les nœuds sont des classes et les arcs des relations interclasses.

Les relations interclasses considérées dans WOMBLE sont les relations d'héritage entre classes, d'implantation entre classes et interfaces et d'associations entre classes et interfaces. Une association abstrait la représentation interne d'une classe.

Par exemple, la classe `CompositeFigure` déclare un vecteur `fFigures` et des méthodes `add()` et `remove()` pour le manipuler, comme montré sur l'extrait de code source 2.1. Le modèle à objets obtenu avec WOMBLE présente une relation d'association entre la classe `CompositeFigure` et l'interface `Figure`, comme montré sur la figure 2.8.

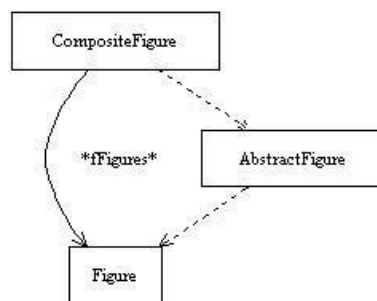
Code source 2.1 – Extrait de la classe `CompositeFigure`.

```
public abstract class CompositeFigure extends AbstractFigure ... {
    /**
     * The figures that this figure is composed of
     * @see #add
     * @see #remove
     */
    protected Vector fFigures;

    ...
}
```

□

FIG. 2.8 – Modèle à objets obtenu par la rétroconception de la classe `CompositeFigure` avec WOMBLE.



□

⁴Nous utilisons WOMBLE, version d'avril 2003, disponible à <http://sdg.lcs.mit.edu/womble/>.

Dans le modèle à objets obtenu par la rétroconception des classes `AbstractFigure`, `CompositeFigure` et `Figure`, les flèches en pointillés représentent, respectivement, les relations d'héritage et d'implantation entre les classes `CompositeFigure`, `AbstractFigure` et l'interface `Figure`; la flèche pleine représente la relation d'association de multiplicités zéro ou plus entre la classe `CompositeFigure` et l'interface `Figure`.

WOMBLE intègre des algorithmes d'analyses syntaxiques pour inférer les relations d'association entre classes et/ou interfaces, qu'elles soient implantées par des champs, des collections, des tableaux.

Ces algorithmes ignorent les types primitifs et les classes de la bibliothèque JAVA standard et utilisent la notion de *class container* pour inférer les relations d'association. Une classe est un container si elle ne référence explicitement que des types primitifs et si au moins une de ses méthodes a comme argument ou comme type de retour `java.lang.Object`.

Les algorithmes analysent les flots de données des méthodes qui utilisent une classe container pour trouver la classe cible de la relation d'association induite par cette classe container. Ils identifient la classe utilisée pour le transtypage d'instances retournées par des méthodes de la classe container et la classe des instances passées comme arguments à des méthodes de la classe container. Cette classe est la cible de l'association.

Les algorithmes annotent les associations avec des informations sur la multiplicité et la changeabilité de la classe cible. La multiplicité de la classe cible peut être zéro ou plus, zéro ou un ou exactement un.

La changeabilité de la classe cible indique si la classe origine de l'association modifie l'ensemble des instances de la classe cible qui lui sont associées. En général, des algorithmes d'analyses des alias sont nécessaires pour calculer multiplicité et changeabilité, WOMBLE utilise des heuristiques simples.

Ainsi, un modèle à objets construit par WOMBLE reflète partiellement le modèle d'un programme au niveau *idiomatique* car il abstrait le code octal du programme analysé par l'identification des relations d'association.

La qualité sémantique du modèle d'un programme obtenu avec WOMBLE a une distance sémantique comprise entre zéro et un, une grande précision sémantique, un niveau de détails sémantiques moyen et une continuité sémantique moyenne.

En particulier, la distance sémantique est comprise entre zéro et un car les algorithmes d'analyses syntaxiques n'identifient pas les relations d'agrégation et de composition : le modèle du programme contient les relations d'association mais pas les relations d'agrégation et de composition du niveau *idiomatique*.

Techniques industrielles basées sur des analyses syntaxiques La plupart des outils d'aide à la conception proposent la rétroconception des programmes à objets pour en construire des modèles au niveau **idiomatique**.

Ces outils incluent des algorithmes d'analyses syntaxiques qui identifient les classes, les interfaces et les relations d'appel de méthodes, d'héritage et d'instanciation dans les modèles des programmes au niveau **implémentation**, code source ou octal.

Par exemple, l'analyse avec l'outil ARGOUML⁵ des classes **AbstractFigure** et **CompositeFigure**, montrées sur le code source 2.2, crée le modèle du programme au niveau **idiomatique** présenté sur la figure 2.9 page 46 : la relation d'héritage a été identifiée dans le modèle du programme au niveau **implémentation** et modélisée au niveau **idiomatique**.

Cependant, les outils d'aide à la conception ne distinguent les relations d'association, d'agrégation et de composition que visuellement pour aider les développeurs à décrire l'architecture de leurs programmes et à communiquer leurs intentions pendant la phase de conception.

Code source 2.2 – Deux classes liées par une relation d'héritage.

```
public abstract class AbstractFigure implements Figure {
    /*
     * Serialization support.
     */
    private static final long serialVersionUID = -10857585979273442L;
    private int abstractFigureSerializedDataVersion = 1;
    ...
}

public abstract class CompositeFigure extends AbstractFigure ... {
    /**
     * The figures that this figure is composed of
     * see #add
     * see #remove
     */
    protected Vector fFigures;

    /*
     * Serialization support.
     */
    private static final long serialVersionUID = 7408153435700021866L;
    private int compositeFigureSerializedDataVersion = 1;
    ...
}
```

□

⁵Nous utilisons ARGOUML v0.8, plus d'informations à <http://argouml.tigris.com>.

Ces distinctions visuelles se basent sur les définitions des relations interclasses proposées *pour* les développeurs dans des notations, telle UML [Object Management Group, Inc., 2003], modifiées dans certains outils, comme dans ROSE [Rational Software Technical Support, 2000].

Les algorithmes d'analyses syntaxiques sont inaptes à différencier ces relations interclasses au niveau **implémentation** car elles n'ont pas de définitions suffisamment précises et les constituants qui les supportent sont disséminés dans le code source des programmes.

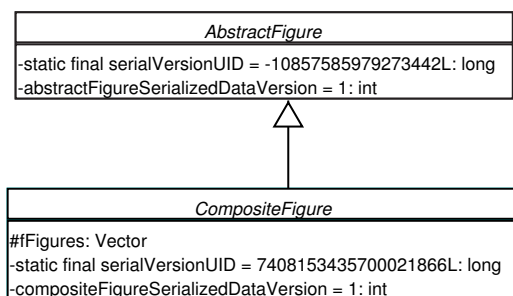
Par exemple, les diagrammes de classes UML figures 2.10(a), 2.10(b) et 2.10(c) page 47 décrivent deux classes A et B liées, respectivement, par une relation d'association, d'agrégation et de composition.

Le code source généré par RATIONAL ROSE⁶ pour chacune d'elles est identique, malgré les différences de sémantique attendues, comme montré sur l'extrait de code source 2.3 page 48.

Si nous remplaçons le tableau utilisé pour contenir les instances de la classe B par une collection, par exemple une instance de la classe `java.util.Vector`, comme dans l'extrait de code source 2.4 page 48, alors le diagramme de classes obtenu par rétroconception montre une relation d'association entre les classes A et `java.util.Vector` au lieu d'une relation d'agrégation entre les classes A et B, figure 2.10(d).

Cette relation d'association est inconsistante avec le diagramme de classe originel et montre l'inaptitude de cet outil à abstraire les choix d'implantation. Cette inaptitude n'est pas spécifique à RATIONAL ROSE, elle se retrouve dans BORLAND JBUILDER⁶, TOGETHERSOFT TOGETHER⁶ et ARGOUML.

FIG. 2.9 – Modèle au niveau **idiomatique** des classes présentées sur l'extrait de code source 2.2 par rétroconception avec ARGOUML. Les méthodes ne sont pas montrées.



□

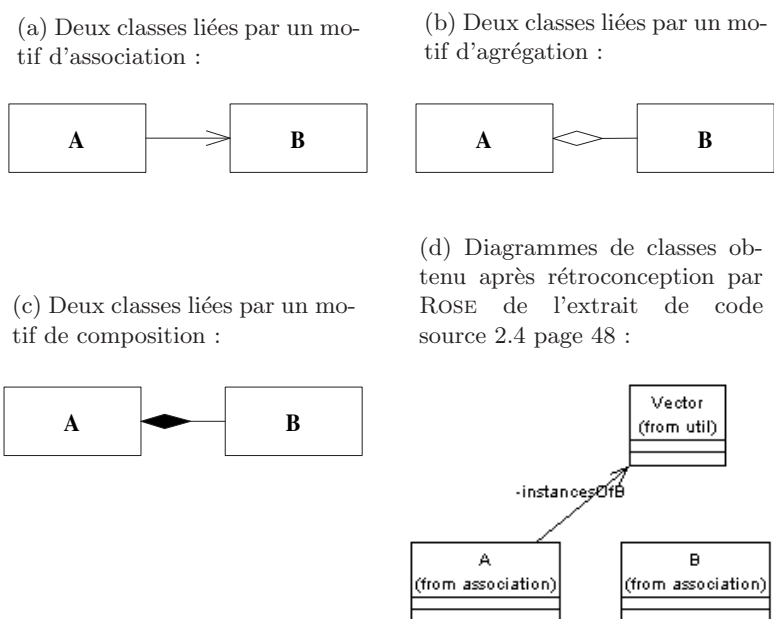
⁶Nous utilisons BORLAND JBUILDER v6.0.438.0, plus d'informations à <http://www.borland.com>; RATIONAL ROSE v2001.03.00, plus d'informations à <http://www.rational.com>; TOGETHERSOFT TOGETHER v5.5.1534v3, plus d'informations à <http://www.togethersoft.com>.

Cette inaptitude traduit un *flo* dans la définition des relations interclasses au niveau de l'implantation et de la conception, flo revendiqué dans la notation UML v1.5 [Object Management Group, Inc., 2003, page 2-66] dans laquelle certaines relations sont intentionnellement⁷ définies de manière imprécise !

En théorie, la qualité sémantique du modèle d'un programme obtenu avec les outils d'aide à la conception a une distance sémantique égale à un, une grande précision sémantique, un niveau de détails sémantiques moyen et une continuité sémantique moyenne.

En pratique, la distance sémantique est nulle car les relations d'association, d'agrégation et de composition n'ont pas de définitions précises dans les modèles de programmes au niveau implémentation, cette absence rend leur identification impossible.

FIG. 2.10 – Motifs interclasses.



□

⁷Il est précisé dans la spécification de la notation UML v1.5 [Object Management Group, Inc., 2003, page 2-66] que “UML gives a more precise meaning to two of these constructs (i.e., association and composite aggregate) and leaves the shareable aggregate more loosely defined in between.”

Code source 2.3 – Code source généré par ROSE pour les motifs inter-classes Association, Agrégation et Composition, avec les motifs interclasses représentés sur les figures 2.10(a), 2.10(b) et 2.10(c).

```
public class A {  
    private B instancesOfB[];  
  
    public A()  
    {  
    }  
}  
  
public class B {  
    public B() {  
    }  
}
```

□

Code source 2.4 – Code source généré par ROSE pour les relations d'association, d'agrégation et de composition, modifié pour utiliser une collection.

```
public class A {  
    private Vector instancesOfB;  
  
    public A()  
    {  
    }  
}  
  
public class B {  
    public B() {  
    }  
}
```

□

Technique par translation Pascal André *et al.* [2000] proposent des règles de translation des diagrammes de classes UML en spécifications algébriques pour vérifier et valider ces diagrammes formellement.

Les auteurs utilisent un ensemble de types de données abstraites pour représenter les notions existantes dans les diagrammes UML, telles les classes, les relations d'association, d'agrégation, de composition et d'héritage.

Par exemple, ils définissent un type de donnée abstraite `Assoc[A, B]` qui décrit une famille d'associations entre deux types quelconques `A` et `B`. Le type `Assoc[Figure, Drawing]` est une actualisation, une instantiation, du type `Assoc[A, B]`. Il définit le type de toutes les relations d'association entre les interfaces `Figure` et `Drawing`. En particulier, l'association de multiplicité zéro ou plus `fDrawing` entre les interfaces `Figure` et `Drawing` peut être définie comme :

```
Operator : fDrawing : -> Assoc[Figure Drawing]
Axiom :   name(fDrawing) = "fDrawing";
Axiom :   inf(0, rightCardinality(fDrawing, aFigure)) ^
           inf(rightCardinality(fDrawing, aFigure), maxint)
```

`Operator` déclare un nouveau constructeur pour l'association `fDrawing` de type `Assoc[Figure, Drawing]`. Les axiomes caractérisent l'association ainsi créée : nom (`name`) et multiplicité de la classe cible (`rightCardinality`).

Ensuite, les auteurs décrivent des règles de translation pour les relations d'association et d'agrégation. Les relations d'agrégation ne nécessitent pas de règles particulières car les auteurs considèrent les relations d'association et d'agrégation comme équivalentes.

Les relations de composition ont des règles spécifiques de translation pour représenter l'appartenance forte et les durées de vie coïncidentes d'une partie avec son tout. Ces règles autorisent plusieurs relations de composition sur une même classe jouant le rôle de partie avec la restriction que ses instances ne peuvent être partagées par plusieurs tous.

Ainsi, les auteurs précisent les définitions des relations d'association, d'agrégation et de composition au niveau *idiomatique* et donnent des indices sur ces relations au niveau *implémentation*. Cependant, ces indices sont insuffisants pour construire automatiquement la spécification algébrique d'un programme depuis son code source.

La qualité sémantique d'une spécification algébrique obtenue avec les règles définies a une distance sémantique comprise entre zéro et un, une grande précision sémantique, un grand niveau de détails sémantiques et une grande continuité sémantique.

Nous estimons que la distance sémantique est comprise entre zéro et un car les règles de translation ne distinguent pas les relations d'association et d'agrégation et les auteurs ne précisent pas la sémantique de la relation d'association au niveau *implémentation*, par exemple par rapport aux appels de méthodes.

2.2.2 Discussion des techniques

Nous avons présenté trois techniques pour identifier les motifs d'association, d'agrégation et de composition : la programmation logique floue, des analyses syntaxiques implantées par des outils académiques et industriels et les spécifications algébriques.

Le tableau 2.1 résume les dimensions de la qualité sémantique des résultats des techniques présentées. Aucune technique ne nous permet d'obtenir un modèle d'un programme au niveau **idiomatique** car aucune n'a une distance sémantique égale à un.

Ces techniques apportent des solutions limitées à l'identification des motifs interclasses par le manque de définitions consensuelles des relations d'association, d'agrégation et de composition au niveau **idiomatique** et par l'absence de règles précises d'identification de ces relations au niveau **implémentation**.

Nous présentons maintenant un état de l'art des définitions et des caractérisations des relations interclasses et de leurs règles d'identification aux niveaux **implémentation** et **idiomatique**.

Tableau 2.1 – Dimensions de la qualité sémantique des techniques présentées d'identification des motifs interclasses.

	Identification de clichés	Analyses syntaxiques		Translations
		Académiques	Industrielles	
Distance sémantique	Zéro-un	Nulle / Zéro-un	Nulle	Zéro-un
Précision sémantique	Faible	Grande	Grande	Grande
Niveau de détails sémantiques	Grand	Grand / Moyen	Moyen	Grand
Continuité sémantique	Moyenne	Grande / Moyenne	Moyenne	Grande

□

2.2.3 État de l'art des définitions

Le principal problème à l'identification des motifs interclasses est leur manque de définitions précises et utilisables pour les identifier et garantir leur traçabilité entre les niveaux implémentation et idiomatique.

Nous présentons des travaux proposant des définitions aux relations interclasses. Nous concluons que ces définitions sont inutilisables pour identifier les motifs interclasses au niveau implémentation, après l'avoir montré sur certains travaux représentatifs.

Définitions en langage naturel Dans la méthode industrielle OMT, qui fut très utilisée et dont s'inspire la notation UML, James Rumbaugh *et al.* [1991] considèrent deux sortes de relations interclasses, association et agrégation, avec la notion de *lien*. Un *lien* est une connexion physique ou conceptuelle entre deux instances.

Une association décrit un groupe de liens avec une structure et une sémantique communes. Une agrégation est une association transitive et non symétrique. Une association représente un ensemble de liens potentiels, comme une classe représente un ensemble d'instances potentielles.

Ces définitions sont simples mais reposent sur la notion de lien qui est mal définie au niveau implémentation et idiomatique. De plus, les auteurs ne mentionnent pas la relation de composition et ses propriétés.

L'OBJECT MANAGEMENT GROUP, INC. (OMG) propose régulièrement de nouvelles versions des spécifications de la notation UML, qui est aujourd'hui le standard *de facto* dans l'industrie.

En 1999, l'OMG donnait les définitions suivantes : une association est une relation entre exactement deux entités⁸ (classes ou interfaces) ; une agrégation est une association spécifiant une relation entre un tout et des parties ; une composition est une agrégation avec une appartenance forte et des durées de vie coïncidentes entre le tout et ses parties [Object Management Group, Inc., 1999].

Deux ans plus tard, l'OMG publiait de nouvelles définitions plus complexes aux relations d'association, d'agrégation et de composition [Object Management Group, Inc., 2001]. Ces nouvelles définitions utilisent plusieurs propriétés des relations : multiplicité ; navigabilité ; remplaçabilité ; partageabilité ; transitivité ; anti-symétrie ; durées de vie.

Cependant, comme noté par de nombreux auteurs [Kent *et al.*, 1999 ; Simons et Graham, 1999 ; Thomas, 2001], quelles que soient les définitions données, elles sont ambiguës. Les définitions incluses dans la version 1.5 de la notation UML, dernière en date, sont également ambiguës et mentionnent peu de détails d'implantation.

D'abord, une relation d'association est définie comme une relation sémantique entre deux entités [Object Management Group, Inc., 2003, page 2-19] sans plus de précision quant à la relation réellement établie entre les entités.

⁸L'OMG utilise le mot anglais *classifier* pour désigner une entité.

Ensuite, elle est définie comme une connexion, un lien, entre les instances des entités associées [Object Management Group, Inc., 2003, page 2-64]. Enfin, un lien est défini comme un couple de références sur des instances [Object Management Group, Inc., 2003, page 3-84] puis il est précisé qu'un lien est une instance d'une association ; or, une association a été définie comme un lien : ces définitions sont donc circulaires !

De plus, le métamodèle UML distingue les relations d'agrégation et de composition uniquement par un attribut multivalué **aggregation** $\in \{\text{none}, \text{aggregate}, \text{composite}\}$ [Object Management Group, Inc., 2003, page 2-22].

Les définitions données dans les spécifications de la notation UML aux relations d'association, d'agrégation et de composition sont ambiguës, difficiles à isoler et elles ne précisent pas les choix d'implantation des relations interclasses au niveau **implémentation**.

L'entreprise RATIONAL [2000], avec ROSE, propose un outil d'aide à la conception très utilisé car inspiré de la notation UML. Cet outil intègre les définitions des relations d'association, d'agrégation et de composition suivantes :

- une relation d'association est une relation entre exactement deux classes (avec la possibilité d'association réflexive d'une classe vers elle-même) ;
- une relation d'agrégation est une association qui distingue *conceptuellement* un "tout" d'une "partie". Une agrégation ne modifie pas la sémantique de l'association entre le tout et ses parties ; elle ne lie pas leurs durées de vie ;
- une relation de composition est une agrégation avec une appartenance forte des parties à leur tout et des durées de vie coïncidentes entre le tout et ses parties. Les parties avec des multiplicités non fixées peuvent être créées après le tout, mais elles sont détruites avec leur tout, sauf si elles en ont été retirées avant sa destruction.

Là encore, les définitions des relations interclasses donnent des indices quant à leur implantation, mais ils sont insuffisants pour en dériver des algorithmes d'identification des motifs interclasses.

D'autres travaux de définition des relations interclasses sont, par exemple, les travaux de David Skogan [1999] pour qui les développeurs doivent manipuler :

- une relation d'association quand ils veulent représenter une relation entre deux classes ;
- une relation d'agrégation quand les instances représentant les "parties" d'une instance "container" peuvent exister sans ce container.
- une relation de composition quand les parties ne peuvent exister sans leur container. Une composition est une agrégation forte : si un container est détruit, alors ses parties sont aussi détruites.

Ces définitions sont, d'une part, mal définies car la *relation* entre deux classes dans une relation d'association n'est pas caractérisée et, d'autre part, opérationnelles sur la destruction des parties avec le tout dans une relation de composition.

Ainsi, elles offrent des indices sur ce que doivent être les relations interclasses au niveau implémentation et idiomatique mais elles sont trop limitées pour permettre l'identification de ces relations.

Dans les travaux menés brièvement à l'Université de Vienne [Wien, 2000] et maintenant arrêtés, une relation d'association existe entre des instances si une instance est logiquement reliée à une ou plusieurs autres instances. Une relation d'agrégation est utilisée pour construire des instances composites.

Une relation de composition est une forme spéciale de relation d'agrégation. Une et une seule relation de composition référence une instance dépendante, mais plusieurs relations d'association peuvent la référencer.

Une instance est dépendante si son existence dépend de l'existence d'une autre instance. La relation de composition est nécessaire pour réaliser des destructions d'instances en cascade et la propagation des valeurs et des méthodes.

Pour Robert C. Martin [1998], une association représente la capacité d'une instance à envoyer un message à une autre instance. Une agrégation est une relation tout-partie similaire à une association mais elle interdit les relations d'agrégation cycliques. Une composition est une agrégation pour laquelle la durée de vie des parties est contrôlée par le tout. Le tout peut créer des parties ou les obtenir déjà créées, il peut passer ses parties à d'autres tous et il peut les détruire.

Définitions formelles Dans leur récent travail sur la spécification algébrique de UML, Pascal André *et al.* [2000] définissent une association entre classes comme un ensemble de liens. Un lien est un ensemble d'instances, une par classe dans la relation. Une association s'enrichit avec plusieurs annotations : nom, navigation, rôles, multiplicité, contraintes, propriétés, qualification et dérivation.

Les auteurs ne donnent pas de définition spécifique à l'agrégation, ils considèrent qu'une agrégation a la même sémantique qu'une association. Une composition est une forme d'agrégation avec une appartenance forte et des durées de vie coïncidentes entre le tout et les parties. Ils définissent l'appartenance forte et les durées de vie coïncidentes avec la notion de partageabilité des parties.

Ces définitions donnent des indices sur les implantations possibles des relations interclasses mais elles manquent de précision pour être utilisées pour identifier les relations dans le code source de programmes.

D'autres définitions formelles ont été proposées, par exemple, par James Noble et John Grundy [1995] pour qui une relation d'association indique qu'une instance utilise une autre instance par un quelconque moyen.

Une agrégation définit l'agrégation d'instances pour créer des instances plus "larges". Les auteurs ne donnent pas de définition à la relation de composition et ne proposent pas de définitions de ces relations au niveau **implémentation**.

Juan Bicaregui *et al.* [1997] proposent une formalisation de la méthode d'analyse par le calcul objet. Ils définissent une relation d'association comme une relation entre plusieurs instances de deux classes. Ils proposent deux contraintes sur cette relation.

La contrainte de cardinalité indique si une association est optionnelle, obligatoire ou une-pour-une. La contrainte de durée de vie est interprétée indépendamment de la multiplicité de la relation et indique qu'un agrégat d'instances existe uniquement avec un ensemble d'instances et que cet ensemble doit rester constant pendant toute sa vie.

Ruth Breu *et al.* [1997] décrivent l'utilisation de flots et de fonctions sur les flots pour formaliser UML. Ils définissent une association soit comme un ensemble de liens de données, soit comme un lien de communication.

Cependant, les auteurs laissent le choix de la sémantique appropriée aux développeurs, sans plus discuter les critères de ce choix. Ils distinguent les relations d'agrégation et de composition par leurs propriétés de partageabilité et de durée de vie ; mais ils ne définissent pas ces propriétés.

Robert B. France [1999] illustre l'utilisation de techniques formelles pour donner une sémantique précise aux diagrammes de classes UML "orientés besoins". Ces diagrammes de classes utilisent les notions de classe et d'association. Une association peut être une association générique, une agrégation, une composition ou une relation d'héritage.

Pour les associations génériques, il considère les propriétés de multiplicité et de changeabilité, les propriétés de navigabilité et de visibilité ne sont pas nécessaires dans les diagrammes de classes orientés besoins.

Une agrégation est une association avec une propriété additionnelle d'appartenance. Une composition est une agrégation avec une propriété additionnelle de durée de vie : si un tout est détruit, alors toutes les parties dont il est composé à *ce moment* sont aussi détruites.

Propriétés des relations interclasses Dans son travail précurseur dont nous sommes beaucoup inspiré, Franco Civello [1993] divise les associations tout-parties (ATP) en deux catégories : les ATP fonctionnelles et non fonctionnelles. Dans une ATP fonctionnelle, la partie est incluse dans le tout car elle contribue au fonctionnement du tout. Dans une ATP non fonctionnelle, la connexion entre la partie et le tout est lâche.

Les ATP non-fonctionnelles se divisent en deux catégories : les ATP groupe-membre, qui modélisent une association, et les ATP uplet-membre, qui modélisent une agrégation. Les groupes sont des ensembles d'instances avec des propriétés communes. Les uplets sont des relations entre entités qui existent indépendamment les uns des autres.

L'auteur propose une première classification des propriétés des ATP : inclusion spatiale et temporelle ; attribution ; visibilité ; encapsulation ; partage ; inséparabilité tout-partie ; inséparabilité partie-tout (durées de vie relatives d'un tout et de ses parties) ; non-changeabilité ; appartenance ; collaboration. L'auteur reste au niveau *idiomatique* et conclut que les langages de programmation manquent d'expressivité pour représenter la variété des ATP.

Dans leur étude exhaustive des relations d'agrégation et de composition de UML, Brian Henderson-Seller [1999 ; 2001] et Frank Barbier [2001] proposent un ensemble de caractéristiques pour la relation tout-partie.

Les auteurs montrent que les définitions des relations d'agrégation et de composition dans UML sont incomplètes, contradictoires et redondantes ; pour cela, ils définissent et utilisent des caractéristiques des relations : principales (tout-partie, propriété émergente, propriété résultante, irréflexivité au niveau des instances, anti-symétrie au niveau des instances et des classes et asymétrie au niveau des instances), secondaires (visibilité, chevauchement des durées de vie, transitivité, partageabilité, configurabilité, séparabilité, changeabilité) et dérivées (propagation d'une ou plusieurs opérations, appartenance, abstraction, dépendance existentielle, propagation des opérations de destruction).

Alors, les auteurs proposent des définitions révisées des relations interclasses avec quatre différentes options. La première option, la séparabilité, est compatible avec la littérature actuelle sur UML. Elle définit la relation d'agrégation comme une relation tout-partie irréflexive et asymétrique au niveau des instances et anti-symétrique aux niveaux des classes et des instances. Le tout et les parties sont séparables.

La relation d'agrégation induit la propagation d'au moins une opération du tout vers ses parties et l'appartenance des parties au tout. Il n'y a pas de lien existentiel entre le tout et les parties, pas de propagation des opérations de destruction. La relation de composition est une agrégation avec propagation des destructions entre le tout et ses parties.

Cependant, les auteurs ne discutent pas de l'implantation avec des langages de programmation par objets standards de ces relations et de ces caractéristiques pour permettre leur identification au niveau implémentation.

D'autres définitions des relations interclasses par leurs propriétés sont données, par exemple, par Sylvain Vauttier *et al.* [1999] qui étudient le comportement des objets composites. Les auteurs distinguent entre le comportement local et global des objets composites. Le comportement local correspond aux fonctionnalités particulières à l'objet composite. Le comportement global correspond aux fonctionnalités entre l'objet composite et ses composants et entre composants.

L'objectif de cette dichotomie entre comportements local et global est d'améliorer la réutilisabilité des composants, la séparation entre l'interface particulière à l'objet composite et son interface comme composite et le comportement de l'objet composite.

Dans leur travail, Monica Saksena *et al.* [1998] définissent l'agrégation avec trois caractéristiques principales et trois caractéristiques secondaires. Les caractéristiques principales sont : structure ; durée de vie ; et appartenance. Les caractéristiques secondaires sont : partage des parties ; homéomérisme⁹ ; et propagation des propriétés. Les auteurs proposent une étude des durées de vie entre parties et tous et ils mentionnent le besoin pour un tout d'avoir des méthodes qui appellent les méthodes des parties.

Enfin, Jean-Michel Bruel *et al.* [2001] proposent des améliorations aux notions d'agrégation et de composition sans ruptures entre les spécifications de UML v1.x et de UML v2.x. Ils basent leur travail sur les travaux précédents de Brian Henderson-Sellers. Ils décrivent UMLTRANZ, un outil de transformation automatique de diagrammes de classes en spécifications formelles Z.

Langages de programmation et extensions Stéphane Ducasse [1995] propose un modèle réflexif pour exprimer et gérer explicitement les dépendances entre objets. Le langage de programmation FLO est une extension du langage de programmation SMALLTALK dans laquelle les dépendances entre objets sont explicites. Cette extension existe aussi pour C++.

Dans son mémoire de thèse [Ducasse, 1997a, page 84], l'auteur décrit l'utilisation du mécanisme de dépendance pour définir la relation de composition. Une relation de composition est implantée par une variable d'instance du type de la classe des objets composants définie dans la classe des objets composites. Les objets composites garantissent leur cohérence interne en limitant l'accès aux objets composants. Ils proposent des méthodes pour accéder aux objets composants.

Ainsi, les objets composites ne respectent pas le principe suivant lequel les informations sur les objets composants qui ne sont pas modifiées par les objets composites doivent rester avec les objets composants [Blake et Cook, 1987].

Alors, l'auteur montre comment son modèle de dépendances explicites permet de regrouper les informations relatives à la relation de composition dans une dépendance manipulable et indépendante des objets composites.

Typiquement, cette dépendance décrit qu'un message reçu et non compris par un objet composite doit être renvoyé à un objet composant. Aussi, elle peut être composée avec d'autres dépendances pour maintenir la cohérence, par exemple une dépendance d'exclusion mutuelle.

Cette approche est intéressante car elle explicite la relation de composition entre classes. Cependant, elle nécessite l'utilisation du langage de programmation non standard FLO et l'auteur ne distingue par les relations d'association et d'agrégation.

D'autres langages de programmation ou extensions pour définir et expliciter les relations interclasses existent. Thorsten Hartmann *et al.* [1992] présentent un nouveau lan-

⁹La caractéristique d'homéomérisme indique que les parties et le tout sont similaires.

gage de programmation, TROLL. Ce langage inclut des événements et l'agrégation explicite d'objets. Des objets complexes décrivent l'agrégation d'objets en objets structurés.

Dans un objet complexe, les objets peuvent être modifiés uniquement par des événements locaux à l'objet complexe, mais leurs attributs sont visibles à l'extérieur. Il existe trois types d'objets complexes : agrégation statique, pour laquelle la composition est déterminée à la compilation ; agrégation dynamique, pour laquelle la composition peut changer à l'exécution ; et objet complexe disjoint, dans lequel les parties ne peuvent exister en-dehors de l'objet complexe.

L'inclusion d'objet est le fondement du langage de programmation TROLL : importation d'objets sûre et communication par création et partage d'événements. Le modèle d'objets complexes est intéressant car il repose sur une base mathématique, la description de processus objets, mais il nécessite un langage de programmation non standard.

Bent Bruun Kristensen [1994] introduit un mécanisme pour supporter des associations implicites et complexes. Des classes englobantes décrivent des associations implicites. Des classes d'association décrivent des associations complexes.

Par exemple, la relation entre un `Client` et un `Consortium` bancaire est une association complexe. Un `Client` se décompose en un `Compte` et une `Carte` de paiement, définissant ainsi une relation implicite entre le `Client` d'une part, et le `Compte` et la `Carte` de l'autre.

Cependant, l'approche présentée ne traite pas de la relation de composition et ne discute pas de l'implantation de ces associations complexes et de ces relations implicites dans un langage de programmation standard.

Définition au niveau implémentation Dianel Jackson et Allison Waingold [1999] développent un outil très abouti, WOMBLE, pour l'extraction de modèles objet du code octal JAVA. Ils proposent un mécanisme d'inférences pour distinguer les relations d'héritage et d'association entre classes.

Une association peut être annotée pour montrer la multiplicité (zéro-ou-plus, zéro-ou-un, exactement un) et la changeabilité (statique ou non) des instances des classes cibles des relations d'association.

Les auteurs décrivent des heuristiques pour inférer la multiplicité et la changeabilité et pour gérer les collections de JAVA. Leur travail est similaire au nôtre et nous a guidé dans nos réflexions sur l'identification des relations interclasses même s'il se limite à la relation d'association ; la relation d'agrégation étant mentionnée et la relation de composition non abordée.

Auparavant, d'autres travaux ont proposé des définitions des relations interclasses au niveau implémentation. Par exemple, James Rumbaugh [1987] propose une représentation explicite des relations interclasses.

L'auteur argumente que l'utilisation des relations comme construction d'un langage de programmation peut avoir un impact majeur sur la formulation et la résolution de la conception, mais uniquement si les relations sont des constructions d'un poids sémantique similaire aux classes et à la relation d'héritage¹⁰.

L'auteur définit alors une relation comme associant les instances de n classes ($n \in \mathbb{N}$), pour indiquer que ces instances sont associées d'une certaine façon. Cependant, il ne distingue pas entre relations d'association, d'agrégation et de composition.

Dans leur projet de laboratoire pour la programmation par objets, Bertrand Rousseau *et al.* [1995] proposent des définitions pour inclure les abstractions des méthodes de conception par objets, comme OMT, dans des langages de programmation comme C++.

Cependant, ils ne se préoccupent pas de l'identification des relations d'association, d'agrégation et de composition et ce projet est maintenant arrêté.

Pour Ajmal Chaumun [2000], une association existe quand une classe référence une variable du type d'une autre classe. Une agrégation existe quand la définition d'une classe met en jeu des instances d'une autre classe. Ces définitions sont proches de l'implantation et sont une source d'inspiration mais l'auteur ne discute pas de la relation de composition.

William Harrison *et al.* [2000] proposent une méthode pour appliquer des diagrammes de classes UML à JAVA. Ils introduisent la notion de *curseur* qui encapsule la complexité et l'implantation des associations. Cependant, ils ne donnent aucun détail d'implantation de ces curseurs et de ces associations.

Esperanza Marcos *et al.* [2001] proposent des règles de traduction des diagrammes de classes UML en des schémas de bases de données à objets. Les auteurs reprennent les définitions des agrégations et des compositions données dans [Rumbaugh *et al.*, 1999] : une agrégation est une forme spéciale d'association entre classes, qui représentent les notions de tous et de parties.

Avec une agrégation simple, plusieurs tous peuvent partager une même partie et les parties n'ont pas de contraintes de durée de vie par rapport aux tous. Une composition est une forme spéciale d'agrégation : une partie appartient à un et un seul tout. La partie vie et meurt avec le tout. Une partie peut être explicitement retirée d'un tout.

Les auteurs présentent alors une traduction des relations et leur implantation en SQL 1999. Cependant, ils ne considèrent pas les appels de méthodes et ils utilisent des constructions spécifiques à SQL 1999 pour gérer les durées de vie.

¹⁰Dans son article, James Rumbaugh déclare que “*use of relations as a semantic construct can have a major impact on the formulation and elucidation of a design, but only if they are considered as semantic constructs of similar weight to classes and generalization.*”

Enfin, des définitions des relations interclasses existent dans la communauté interface homme-machine. Par exemple, Holger Eichelberger *et al.* [2002] présentent un cadre pour la visualisation de diagrammes de classes UML décrits avec un langage de représentation des diagrammes, UMLSCRIPT. Cependant, ils se focalisent sur les techniques de visualisation et de mise en page des diagrammes de classes, pas sur les techniques d'identification des relations interclasses.

2.2.4 Discussion des définitions

Nous avons présenté des techniques pour identifier les relations d'association, d'agrégation et de composition. Ces techniques sont limitées par l'absence de définitions consensuelles et précises des relations interclasses et elles ne garantissent pas la traçabilité des relations identifiées.

Nous avons dressé un état de l'art des définitions des relations interclasses. Ces définitions varient d'un auteur à l'autre et elles ne sont pas utilisées, ou utilisables, pour assurer l'identification et garantir la traçabilité des relations interclasses.

Pourtant, certaines propriétés récurrentes des relations interclasses ressortent de ces définitions : par exemple, les propriétés d'exclusivité, de multiplicité, de durée de vie, la possibilité d'envoi de messages entre instances, etc.

Nous sommes convaincus qu'il est possible de définir précisément les relations d'association, d'agrégation et de composition avec leurs propriétés et de proposer des modèles et des algorithmes pour construire les modèles des programmes au niveau **idiomatique** dans lesquels nous voulons identifier les motifs de conception.

Conclusion. Nous avons étudié des techniques existantes d'identification et de traçabilité des motifs interclasses pour construire un modèle d'un programme au niveau **idiomatique** dans lequel nous voulons identifier les motifs de conception. Nous étudions maintenant l'identification et la traçabilité des motifs de conception.

2.3 Motifs de conception

NOUS présentons un état de l'art des techniques pour l'identification des motifs de conception. Ces techniques consistent à modéliser les motifs de conception puis à rechercher ces modèles dans des modèles de programmes.

Les modèles des motifs et des programmes sont, par exemple, des bases de faits PROLOG ou des modèles UML. Les techniques d'identification des modèles de motifs de conception sont, par exemple, la transformation de graphes, l'unification.

Nous montrons que ces techniques sont limitées par l'absence de modèles uniques des motifs de conception, d'interactions avec les mainteneurs et d'explications des micro-architectures similaires aux motifs identifiés.

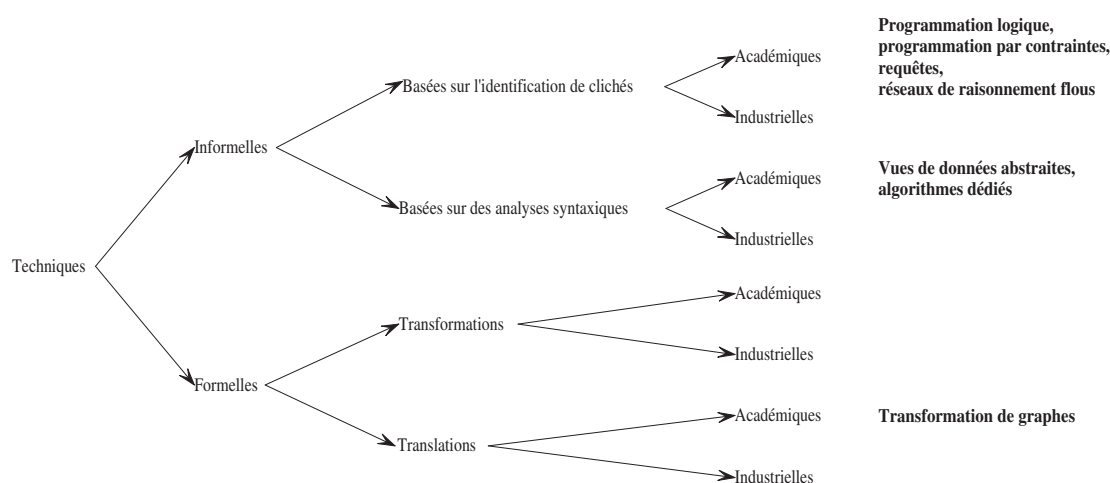
Nous concluons sur les caractéristiques spécifiques du problème d'identification des motifs de conception et de leur traçabilité dans des modèles de programmes entre les niveaux implémentation et conception.

2.3.1 Techniques d'identification

Nous présentons trois types de techniques, comme montré sur la figure 2.11 :

- des techniques basées sur l'identification de clichés qui utilisent la programmation logique, la programmation par contraintes, l'utilisation de requêtes et les réseaux de raisonnement flou ;
- des techniques basées sur des analyses syntaxiques, les vues de données abstraites et des algorithmes d'analyses dédiés ;
- une technique de translation par transformation de graphes.

FIG. 2.11 – Techniques de rétroconception des motifs de conception.



Ces trois types de techniques d'identification des motifs de conception sont représentatives des techniques existantes d'identification des motifs de conception car nous n'avons pu trouver une technique formelle basée sur des transformations.

Nous ne comparons pas ces techniques par la qualité sémantique de leurs résultats car elle ne varie que par sa précision sémantique : tous les résultats ont une distance sémantique égale à deux par définition, un niveau de détails sémantiques moyen et une continuité sémantique moyenne.

La précision sémantique des résultats varie de faible à grande : faible pour les techniques basées sur les clichés (programmation logique, programmation par contraintes, requêtes et réseaux de raisonnement flou) ; moyenne pour la technique par translation (transformation de graphes) ; grande pour la technique par analyses syntaxiques (vue de données abstraites et algorithmes dédiés).

Techniques basées sur l'identification de clichés L'utilisation de la programmation logique pour modéliser les programmes, les motifs de conception et réaliser leur identification a été étudiée par de nombreux auteurs [Bratko et Muggleton, 1995 ; Krämer et Prechelt, 1996 ; Sefika *et al.*, 1996 ; Prechelt et Krämer, 1998 ; Ciupke, 1999 ; Richner et Ducasse, 1999 ; Bergenti et Poggi, 2000]. La manipulation de code source avec la programmation logique a été récemment baptisée *métaprogrammation déclarative* [Brichau, 2000 ; Mens *et al.*, 2002b].

Par exemple, SOUL [Wuyts, 1998 ; Wuyts *et al.*, 1999 ; Mens *et al.*, 2002b ; Mens *et al.*, 2002a] est un système de programmation logique basé sur PROLOG et intégré à l'environnement de développement VISUALWORKS SMALLTALK.

La programmation logique permet l'unification de termes décrits en SMALLTALK et le raisonnement sur les nœuds de l'arbre de syntaxe abstraite maintenu par l'environnement. Les classes de l'environnement et leurs relations sont représentées par des faits obtenus directement par le système PROLOG, dont un exemple pour une implantation en SMALLTALK du programme JHOTDRAW est montré sur l'extrait du code source suivant :

```
interface(figure).
classe(abstractFigure).
classe(attributeFigure).
...
classe(compositeFigure).
hirarchie(figure, abstractFigure).
hirarchie(abstractFigure, attributeFigure).
...
hirarchie(compositeFigure, attributeFigure).
composition(compositeFigure, figure).
...
```

SOUL offre un ensemble de prédicats pour raisonner sur les faits extraits de l'environnement, par exemple le prédicat¹¹ `hiérarchie(SUPERCLASSE, CLASSE)` pour vérifier l'existence d'une relation d'héritage entre les classes `SUPERCLASSE` et `CLASSE` ou pour unifier les variables `SUPERCLASSE` et `CLASSE` avec deux classes de l'environnement appartenant à une même branche de l'arbre d'héritage.

Avec ces prédicats, il est possible de construire de nouveaux prédicats. Par exemple, le prédicat `superclasseCommune/3` identifie dans l'environnement `SMALLTALK` toutes les classes avec une superclasse commune :

```
superclasseCommune(CLASSE1, CLASSE2, SUPERCLASSE) :-  
    hiérarchie(SUPERCLASSE, CLASSE1),  
    hiérarchie(SUPERCLASSE, CLASSE2).
```

SOUL a été naturellement utilisé pour identifier des motifs de conception parmi les classes présentes dans l'environnement `SMALLTALK`. Par exemple, les prédicats `PROLOG` présentés sur le code source 2.5 décrivent l'identification du motif de conception `Composite` [Wuyts, 1998], présenté sur la figure 1.7(c) page 25.

Code source 2.5 – Prédicat d'identification du motif de conception `Composite` avec SOUL.

```
1 motifComposite(RAMIFICATION, BRANCHE, OPÉRATION) :-  
2     compositeStructure(RAMIFICATION, BRANCHE),  
3     compositeAgrégation(RAMIFICATION, BRANCHE, OPÉRATION).  
4  
5 compositeStructure(RAMIFICATION, BRANCHE) :-  
6     classe(RAMIFICATION),  
7     hiérarchie(RAMIFICATION, BRANCHE).  
8  
9 compositeAgrégation(RAMIFICATION, BRANCHE, OPÉRATION) :-  
10    méthodeCommune(RAMIFICATION, BRANCHE, _, BRANCHEMÉTHODE),  
11    méthodeSignature(OPÉRATION, BRANCHEMÉTHODE),  
12    unànDéclaration(BRANCHEMÉTHODE, _, ÉNUMÉRATION),  
13    contientEnvoiDeMessage(ÉNUMÉRATION, OPÉRATION).
```

□

¹¹Nous réécrivons avec la syntaxe "Edimbourg" et en français les prédicats `PROLOG` proposés avec la syntaxe marseillaise en anglais dans [Wuyts, 1998].

Le modèle du motif de conception **Composite** représenté par ces prédicats définit un motif comme une classe **BRANCHE** sous-classe d'une classe **RAMIFICATION**, une relation un-à-n entre la branche et la ramification supportée par une méthode nommée **OPÉRATION**.

D'abord, le prédicat décompose l'identification des participants au motif de conception **Composite** en deux phases, lignes 2 et 3 : l'identification de la structure du motif et l'identification de la relation d'agrégation. L'identification de la structure consiste à identifier toutes les classes appartenant à une même branche de l'arbre d'héritage, lignes 5–7.

L'identification de la relation d'agrégation consiste à identifier deux classes telles que la classe **BRANCHE** dans le motif de conception définisse une méthode **OPÉRATION** qui appelle une méthode de même nom sur les éléments d'une énumération et que la classe **RAMIFICATION** définisse aussi une méthode **OPÉRATION**.

L'utilisation de la programmation logique permet de déclarer explicitement les participants et les relations entre les participants d'un motif de conception pour identifier les classes qui satisfont ses relations.

Cependant, l'utilisation de la programmation logique oblige à déclarer de multiples prédicats pour prendre en compte les *variantes* d'un motif de conception. Par exemple, si nous désirons obtenir les micro-architectures similaires au modèle du motif de conception proposé *sans* la relation d'héritage entre les classes **RAMIFICATION** et **BRANCHE**, nous devons ajouter des prédicats **motifComposite2/3** et **compositeStructure2/2**, tels que :

```
motifComposite2(RAMIFICATION, BRANCHE, OPÉRATIONNAME)
  compositeStructure2(RAMIFICATION, BRANCHE),
  compositeAgrégation(RAMIFICATION, BRANCHE, OPÉRATIONNAME).

compositeStructure2(RAMIFICATION, BRANCHE) :–
  classe(RAMIFICATION).
```

Si nous désirons maintenant obtenir les micro-architectures similaires pour lesquelles **RAMIFICATION** et **BRANCHE** n'ont pas de méthode en commun, ou pour lesquelles la relation d'agrégation est de multiplicité zéro-à-n, nous devons ajouter autant de nouveaux prédicats. Cet ajout *manuel* de prédicats limite l'extensibilité du système et nécessite un aller-retour entre les développeurs du système et les mainteneurs qui l'utilisent.

Un travail similaire sur l'identification des motifs de conception par des prédicats logiques est présenté dans [Florijn *et al.*, 1997] : les prédicats logiques sont dérivés des modèles de motifs de conception exprimés par des modèles de fragments et ils opèrent sur une base de faits également représentée par le modèle de fragments.

Cependant, nous souhaiterions proposer un système qui offre directement aux mainteneurs l'identification des variantes d'un *unique* modèle de motif de conception.

La reconnaissance de plans [Rich et Waters, 1990 ; Quilici *et al.*, 1997], telle qu'elle est définie et utilisée en intelligence artificielle, peut aussi être utilisée pour identifier des motifs de conception.

Alex Quilici *et al.* [1997] définissent des algorithmes de résolution de problèmes de satisfaction de contraintes dédiées à la reconnaissance de plans dans des modèles de programmes impératifs, principalement en COBOL.

Un programme est modélisé par son arbre de syntaxe abstraite. Un plan est représenté par un ensemble de constituants (nœuds de l'arbre de syntaxe abstraite) et de contraintes entre ces constituants (flots de contrôle et de données, appels de fonctions, etc.).

Alors, l'identification d'un plan dans le modèle d'un programme est un problème de satisfaction de contraintes dans lequel les variables du problème sont les constituants du plan, les contraintes sont les contraintes entre les constituants du plan et le domaine des variables est l'arbre de syntaxe abstraite du programme à analyser.

Les auteurs proposent des algorithmes spécialisés pour identifier des plans par la résolution des problèmes de satisfaction de contraintes ainsi obtenus. En particulier, les auteurs montrent que leurs algorithmes passent à l'échelle.

Ces travaux sont intéressants pour l'identification des motifs de conception avec la programmation par contraintes, comme les auteurs le font remarquer [Quilici *et al.*, 1997, page 167]. Cependant, ils n'ont jamais été, à notre connaissance, appliqués pour identifier les micro-architectures *similaires* à des motifs de conception.

De plus, si ces travaux permettent d'identifier des plans dans des programmes impératifs, ils ne s'adaptent pas *directement* à la recherche de motifs de conception dans des programmes à objets et nécessitent la définition de nouveaux formalismes pour représenter des plans et des programmes à objets.

Enfin, ces travaux ne traitent pas le problème de la traçabilité des motifs de conception entre niveaux d'abstraction et le besoin d'expliquer les micro-architectures similaires à un motif identifiées.

La reconnaissance de plans par la résolution de problème de satisfaction de contraintes est une source d'inspiration importante pour nos travaux. Nous essayons de répondre à ses limitations avec la programmation par contraintes avec explications : identification des micro-architectures *similaires* et *explications*.

Des nombreux auteurs ont proposé d'utiliser des requêtes sur des modèles de programmes pour identifier les constituants dont les types et la structure sont similaires à des motifs [Kullbach et Winter, 1999].

Par exemple, SPOOL [Keller *et al.*, 1999 ; Schauer et Keller, 1999 ; Keller et Schauer, 2000] est un environnement de rétroconception qui se décompose en trois parties :

1. Un système de construction de modèles de programmes C++, JAVA ou SMALL-TALK. Un modèle est obtenu par analyses lexicale et syntaxique du programme au niveau **implémentation** et est représenté par un modèle intermédiaire UML/CDIF. Un modèle UML/CDIF est décrit avec la syntaxe du format d'échange CDIF et avec la sémantique définie par le métamodèle UML.
2. Un référentiel de composants de conception abstraits, de composants de conception implantés et de modèles de programmes. Un composant de conception abstrait est le modèle d'un motif de conception ; un composant de conception implanté est une micro-architecture identifiée comme similaire à un composant abstrait.
3. Un système de visualisation et de manipulation des modèles des programmes et des composants de conception abstraits et implantés.

Le modèle d'un programme contient des informations sur : les fichiers analysés (noms et chemins) ; les classes et les interfaces ; les relations de généralisation ; les attributs ; les méthodes ; les paramètres et les types de retour des méthodes ; les appels de méthodes ; les instanciations ; l'utilisation des variables ; les relations d'amitiés entre classes.

L'identification de composants de conception implantés est réalisée manuellement, automatiquement ou semi-automatiquement. Une requête est associée avec un composant de conception abstrait et est appliquée sur le modèle d'un programme. Les résultats de la requête sont alors transformés en des composants de conception implantés qui représentent les micro-architectures similaires au composant de conception abstrait.

La traçabilité des composants de conception implantés est garantie car l'environnement mémorise les modèles des programmes et les modèles des composants de conception abstraits et implantés dans trois bases de données distinctes, entre lesquelles il est possible de naviguer.

Cependant, la requête utilisée pour identifier les composants de conception implantés n'est pas générée automatiquement depuis le composant de conception abstrait. Cette requête permet uniquement l'identification des formes complètes du composant ; les formes approchées nécessitent la définition de nouveaux composants de conception abstraits.

Les réseaux de raisonnement flou ont été utilisés pour identifier des relations inter-classes, comme montré dans la section précédente page 37. Ils ont aussi été étudiés pour identifier des formes approchées de motifs de conception.

Jens H. Jahnke et Albert Zündorf [1997] présentent une utilisation des réseaux de raisonnement flou pour modéliser et identifier des mauvaises implantations de motifs de conception.

Par exemple, le motif de conception *Singleton* est décrit par un réseau de raisonnement flou qui représente les règles d'identification de mauvaises implantations de ce motif. Le réseau de raisonnement flou permet de calculer une valeur de confiance représentative de la présence d'une mauvaise implantation du motif. Le calcul est semi-automatique car il est dirigé par les mainteneurs.

Comme pour les relations interclasses, cette approche est intéressante car elle permet d'identifier les formes approchées d'un motif de conception et de quantifier la confiance que les mainteneurs peuvent avoir dans les formes identifiées.

Cependant, ce travail n'a pas été poursuivi, à notre connaissance. De plus, la représentation des motifs de conception comme des réseaux de raisonnement flou n'est pas très naturelle et elle nous semble se limiter aux motifs de conception proches des langages de programmation.

Enfin, l'interaction avec les mainteneurs pour identifier les constituants du programme est limitée et nécessite la connaissance des réseaux de raisonnement flou. Cette technique ne nous paraît pas adéquate pour garantir la traçabilité des motifs de conception.

Techniques basées sur des analyses syntaxiques Un travail original sur l'identification des motifs de conception utilise les vues et les objets de données abstraites. Paulo S. C. Alencar *et al.* [1995 ; 1996] proposent d'identifier des motifs de conception modélisés comme des constructeurs de vues de données abstraites.

Un modèle formel est défini pour décrire les composants d'un programme. Ce modèle décompose un composant en deux parties : une vue de données abstraites (VDA) qui représente un modèle du composant et des objets de données abstraites (ODA) qui représentent des instances de ce composant. Vues de données abstraites et objets de données abstraites permettent de distinguer l'interface d'un composant de son implantation.

Ce modèle formel est aussi utilisé pour modéliser les motifs de conception. Un motif de conception est décrit par un constructeur qui spécifie l'objectif du motif, ses paramètres, le processus d'instanciation du motif, les conséquences de son application et la spécification des participants par des vues et des objets de données abstraites dépendant du langage de programmation. Par exemple, le motif de conception **Composite** est décrit par le constructeur représenté sur la figure 2.2 page 68.

L'identification du motif de conception dans un programme consiste alors à regrouper les composants d'un programme obtenus par des analyses statiques et des analyses dynamiques en modules. L'appartenance d'un composant à un module est déterminée par les règles représentées par les vues et les objets de données abstraites [Kunz et Black, 1995]. Par exemple, les composants d'un programme sont regroupés suivant les déclarations définies par le constructeur du motif de conception **Composite**.

Ces travaux ont plusieurs limitations pour nos problèmes de l'identification et de la traçabilité des motifs de conception. Les auteurs mentionnent seulement brièvement des outils implantés pour reconnaître des motifs de conception modélisés par des vues et des objets de données abstraites, ceux-ci ne sont pas disponibles au téléchargement et nous n'avons pas pu les évaluer. Les algorithmes d'identification ne sont pas explicitement décrits et aucun exemple concret d'identification d'un motif de conception depuis le code source d'un programme jusqu'aux objets de données abstraites n'est présenté. Enfin, la traçabilité des motifs identifiés n'est pas discutée.

Tableau 2.2 – Constructeur du motif de conception Composite.

Opérateur : motif de conception Composite

Objectif : composer des objets en une structure d'arbre pour représenter des hiérarchies tous-parties

Paramètres : objets : RAMIFICATION

Sous-tâches :

1. Créer un structure d'arbre
 - (a) Instancier l'objet concret : RAMIFICATION \rightarrow BRANCHE
 - (b) Instancier l'objet concret : RAMIFICATION \rightarrow FEUILLES
 - (c) Composer les objets : FEUILLES, BRANCHE \rightarrow BRANCHE
 - (d) Si un sous-arbre est requis :
 - i. Créer récursivement un sous-arbre (étape 1)
 - ii. Composer les objets : subFEUILLES, subBRANCHE \rightarrow subBRANCHE

Conséquence : une structure d'arbre composée d'objets FEUILLES et BRANCHEs est créée dans laquelle les objets BRANCHEs sont les nœuds internes de l'arbre.

Texte de production :

VDA/ODA BRANCHE

Déclarations :

...

Attributs :

Ramification : ODA RAMIFICATION

VDA/ODA inclus :

Ensemble de ramifications

Héritage de la ramification

...

Fin de la VDA/ODA BRANCHE

VDA/ODA FEUILLE

Déclarations :

...

VDA/ODA inclus :

Héritage de la ramification

...

Fin de la VDA/ODA FEUILLE

Fin de l'opérateur

□

De nombreux auteurs proposent des algorithmes d'analyses syntaxiques dédiés à l'identification des motifs de conception, par exemple [Brown, 1996 ; Hedin, 1997 ; Albin-Amiot, 2003]. L'intérêt de ces approches réside dans l'efficacité des algorithmes proposés en temps, en rappel et en précision, même si nous cherchons à proposer une solution plus générale qui utilise des algorithmes standard, comme la programmation par contraintes.

Une autre technique consiste à utiliser les noms donnés aux classes, interfaces, champs et méthodes pour définir des algorithmes d'analyses syntaxiques et extraire la sémantique de programmes [Anquetil et Lethbridge, 1998]. Cette technique nécessite un traitement poussé du langage naturel utilisé pour nommer les constituants des programmes et s'appuie sur des conventions rarement respectées.

Michiaki Tatsubori et Shigeru Chiba [1998] utilisent la réflexion pour introduire explicitement les motifs de conception avec une extension du langage de programmation JAVA. L'identification et la traçabilité des motifs de conception est alors garantie par définition. Cependant, cette solution nécessite l'utilisation d'un langage de programmation non standard et ne supporte pas l'utilisation de formes approchées de motifs de conception.

Enfin, Pascal Rapicault et Mireille Fornarino [2000] proposent un protocole de modélisation des modèles abstraits de motif de conception spécifiquement créé pour la vérification des applications des motifs.

Un modèle abstrait d'un motif de conception représente un motif de conception hors de tout contexte, et un modèle concret représente alors le motif appliqué dans un contexte donné.

Le modèle abstrait d'un motif de conception inclut trois types de contraintes : des contraintes génératrices ou causales, qui induisent la création d'un constituant du motif de conception lorsqu'elles sont interprétées ; des contraintes de vérification pour exprimer les propriétés structurelles qui doivent être vérifiées par le modèle concret ; des contraintes de code pour préciser le code attendu dans les méthodes des modèles concrets.

Les contraintes sont traduites en OBJECT CONSTRAINT LANGUAGE (OCL) [Object Management Group, Inc., 1997] lors de la vérification. La vérification d'un modèle concret est réalisée semi-automatiquement car le lien entre les constituants du programme et d'un modèle concret est réalisé manuellement. Cette technique est donc limitée et ne satisfait pas notre besoin d'identifier automatiquement les constituants d'un programme participant à un motif de conception.

Technique par translation Les graphes et les transformations de graphes ont été utilisés pour modéliser et identifier les motifs de conception dans des programmes [Antoniol *et al.*, 1998 ; Seemann et von Gudenberg, 1998].

Par exemple, Jochen Seemann et Jürgen Wolf von Gudenberg [1998] proposent un processus de rétroconception de programmes JAVA par raffinement. D'abord, le code source du programme est analysé pour construire un graphe $S = (V, E, \phi, \eta)$ avec trois types de nœuds et huit types d'arcs :

- $V = CLASSES \times INTERFACES \times MÉTHODES$, l'ensemble des nœuds du graphe, respectivement les classes, les interfaces et les méthodes du programme ;
- $E \subseteq CLASSES \times CLASSES \cup INTERFACES \times INTERFACES \cup CLASSES \times INTERFACES \cup CLASSES \times MÉTHODES \cup MÉTHODES \times MÉTHODES \cup MÉTHODES \times CLASSES$, six types d'arcs représentant les différentes relations interclasses : héritage, implantation, référence, appartenance d'une méthode à une classe ou à une interface, appel d'une méthode par une autre, transtypage et instantiation ;
- $\phi : V \rightarrow TEXTE \times TYPE^n$, où $\phi(c), c \in CLASSES \cup INTERFACES$ dénote le nom d'une classe ou d'une interface et $\phi(m), m \in MÉTHODES$ dénote le nom, le type de retour et les types des paramètres d'une méthode à $n - 1$ paramètres.
- $\eta : E \rightarrow TEXTE \times TEXTE \times MULTIPLICITÉ$, une fonction de nommage des arcs en trois parties, les deux dernières sont optionnelles. La première partie nomme l'arc : héritage, implantation, référence, etc. Pour les arcs représentant un appel de méthode, la deuxième partie précise le site de l'appel : variable locale, champ, etc. La troisième partie dénote la multiplicité de la relation : un ou plusieurs.

Ensuite, le graphe V est transformé avec une grammaire de graphe pour abstraire les relations entre classes et interfaces, par exemple en introduisant une relation d'appel entre classes : $m_1 \text{ appelle } m_2 \wedge c_1 \text{ définit } m_1 \wedge c_2 \text{ définit } m_2 \Rightarrow c_1 \text{ appelle } c_2$, ou pour définir des relations d'association et d'agrégation :

$$c_1 \text{ est associée avec } c_2 \Leftrightarrow c_1 \text{ appelle } c_2 \wedge c_1 \text{ référence } c_2$$

$$c_1 \text{ agrège } c_2 \Leftrightarrow c_1 \text{ association } c_2 \wedge c_1 \text{ crée } c_2$$

Le graphe ainsi obtenu est similaire au modèle du programme que nous désirons au niveau *idiomatique*. Cependant, il contient moins d'information car la relation de composition n'est pas discutée.

Enfin, le graphe V est transformé avec une grammaire de graphe pour abstraire les motifs de conception. Par exemple, l'identification des micro-architectures *identiques* au motif de conception *Composite* est réalisée par une transformation qui ajoute un nœud *composite* : soit $SUB(C) = \{D \mid D \text{ sous-classe transitive de } C\}$, alors :

$$\exists D \in SUB(C) \mid D \text{ agrège } C \wedge D \text{ délègue à } C \Rightarrow \text{motif Composite}(C, \mathcal{D}, \mathcal{A})$$

$$\text{où } \mathcal{D} = \{D \mid D = SUB(C)\} \text{ et } \mathcal{A} = \{A \mid A = SUB(C) - SUB(D)\}$$

Cette technique d'identification des motifs interclasses et de conception est intéressante car les graphes sont bien connus et des algorithmes efficaces de manipulation et de transformation existent.

Cependant, cette technique ne satisfait pas nos besoins : d'une part, l'identification des formes approchées de micro-architectures similaires à un motif n'est pas assurée, elle nécessite la recherche de sous-graphes *similaires* à un graphe. Cette recherche est un problème difficile, voir par exemple [Eppstein, 1995]¹².

D'autre part, la traçabilité des motifs interclasses et des motifs de conception n'est pas garantie, les nœuds construits par transformation pour représenter les motifs interclasses et de conception identifiés ne mémorisent pas les nœuds et les arcs dont ils sont issus.

¹²Jacobo Torán [1996] déclare que “*The graph isomorphism problem is one of the few problems in the class NP that is not known to be complete nor polynomial time solvable.*”

2.3.2 Discussion

Nous avons présenté trois techniques pour l'identification des motifs de conception : des techniques basées sur l'identification de clichés (programmation logique, programmation par contraintes, requêtes et réseaux de raisonnement flou) ; des techniques basées sur des analyses syntaxiques (vues de données abstraites et algorithmes dédiés) ; une technique par translation (transformation de graphes).

Par définition, la qualité des résultats de ces techniques a une distance sémantique égale à deux, une précision sémantique qui varie de faible à grande, un niveau de détails sémantiques moyen et une continuité sémantique moyenne.

Ces techniques soulignent les caractéristiques des problèmes de l'identification et de la traçabilité des motifs de conception. D'une part, l'identification nécessite la définition de modèles uniques et de première classe des motifs. Ces modèles doivent contenir toutes les informations nécessaires à l'identification des formes complètes *et* approchées des motifs de conception qu'ils représentent.

D'autre part, l'identification des motifs de conception doit favoriser le dialogue avec les mainteneurs du programme :

- pour expliquer concrètement pourquoi une micro-architecture est similaire à un motif de conception (formes complètes et approchées) ;
- pour diriger dynamiquement la recherche de telles micro-architectures en proposant interactivement la prise en compte des caractéristiques du motif recherché et en évitant de déterminer *a priori* ses variantes ;
- pour garantir la traçabilité entre les micro-architectures identifiées et les constituants du programme dont la structure et l'organisation suggère l'application d'un motif de conception.

Conclusion. Nous avons étudié des techniques existantes d'identification et de traçabilité des motifs de conception. Ces techniques sont limitées par l'absence de modèles uniques et de première classe des motifs de conception et d'explications lors de l'identification des motifs. Nous dressons maintenant un bilan de l'état de l'art sur la traçabilité des motifs interclasses et de conception.

Bilan

DE NOMBREUX TRAVAUX ont cherché à identifier et à garantir la traçabilité des motifs interclasses et de conception. Aussi, nous avons présenté un cadre de classification et de comparaison des techniques de rétroconception [Gannod et Cheng, 1999] qui propose une taxonomie des techniques et une qualification de leurs résultats.

La taxonomie décompose les techniques de rétroconception en techniques formelles, par transformations ou translations, et informelles, basées sur l'identification de clichés ou sur des analyses syntaxiques.

La classification décompose la qualité des résultats de la rétroconception en quatre dimensions sémantiques, distance sémantique, précision sémantique, niveau de détails sémantiques et continuité sémantique.

Nous avons quantifié la distance sémantique avec le niveau d'abstraction auquel se situent les résultats des techniques de rétroconception : **implémentation**, **idiomatique**, **conception** et **analyse**.

Ensuite, nous avons décrit des techniques d'identification et de traçabilité des motifs interclasses. Ces techniques sont basées sur l'identification de clichés (programmation logique floue), utilisent des analyses syntaxiques¹³ (ARGOUML, CHAVA, JBUILDER, ROSE, TOGETHER et WOMBLE) et des translations (sémantique algébrique).

Les résultats de ces techniques ont une distance sémantique nulle car les définitions des relations interclasses sont ambiguës, diffèrent suivant les auteurs et ne précisent pas les choix d'implantation possibles.

Puis, nous avons décrit des techniques d'identification et de traçabilité des motifs de conception. Ces techniques sont basées sur l'identification de clichés (programmation logique floue, programmation par contraintes, utilisation de requêtes et réseaux de raisonnement flou), utilisent des analyses syntaxiques¹³ (vues de données abstraites et algorithmes dédiés) et des translations (transformation de graphes).

Les résultats de ces techniques se distinguent uniquement par leurs précisions sémantiques, qui varient de faible à grande, et ils montrent le besoin d'identifier les formes complètes et approchées des motifs, représentés avec des modèles uniques et de première classe, par l'interaction avec les mainteneurs et en expliquant les micro-architectures identifiées.

Maintenant, nous proposons des définitions consensuelles aux motifs interclasses aux niveaux **implémentation** et **idiomatique** et nous décrivons des algorithmes pour les identifier dans le modèle d'un programme au niveau **implémentation** et pour garantir leur traçabilité.

Puis, nous proposons des modèles des motifs de conception et l'utilisation de la programmation par contraintes avec explications pour identifier semi-automatiquement les micro-architectures similaires à ces motifs dans le modèle d'un programme au niveau **idiomatique** et pour justifier les formes complètes et approchées identifiées.

¹³Les techniques basées sur des analyses syntaxiques sont très utilisées pour l'identification des motifs interclasses et de conception en raison de la grande précision sémantique qu'elles offrent.

Deuxième partie

Traçabilité des motifs

NOUS proposons et décrivons des modèles et des algorithmes pour résoudre le problème de l'identification et de la traçabilité des motifs de conception entre les niveaux **implémentation** et **conception**.

Nous décomposons ce problème en deux sous-problèmes : un sous-problème de l'identification et de la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique** et un sous-problème de l'identification et de la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**.

D'abord, nous étudions l'identification et la traçabilité des motifs interclasses pour construire un modèle d'un programme au niveau **idiomatique** représentatif de son implantation. L'identification et la traçabilité des motifs interclasses sont limitées par l'absence de définitions consensuelles aux motifs et de choix d'implantation explicites.

Nous proposons des définitions aussi consensuelles que possible aux motifs interclasses aux niveaux **implémentation** et **idiomatique** pour les intégrer au métamodèle proposé. Nous raffinons ces définitions des motifs interclasses au niveau **implémentation** avec quatre propriétés minimales : durée de vie, exclusivité, multiplicité et site d'invocation.

Ces quatre propriétés nous permettent de définir des algorithmes d'analyses statiques et dynamiques pour identifier les motifs interclasses au niveau **implémentation** et pour construire un modèle représentatif du programme au niveau **idiomatique**.

Nous utilisons le langage de programmation JAVA pour décrire un programme au niveau **implémentation**, statiquement et dynamiquement, et nous proposons un métamodèle dédié à la description du programme au niveau **idiomatique**.

Ensuite, nous détaillons l'identification et la traçabilité des formes complètes et approchées des motifs de conception dans les modèles d'un programme au niveau **idiomatique** par l'interaction avec les mainteneurs.

Nous utilisons le modèle du programme obtenu au niveau **idiomatique** et nous présentons deux métamodèles pour décrire un programme au niveau **conception** et les motifs de conception au niveau **idiomatique**.

Nous montrons que l'identification des micro-architectures similaires aux motifs de conception se traduit en un problème de satisfaction de contraintes. Nous utilisons la programmation par contraintes avec explications pour résoudre ce problème.

Nous décrivons une bibliothèque de contraintes et des stratégies de recherche qui assurent l'identification et l'explication des formes complètes et approchées et nous montrons comment garantir leur traçabilité entre les niveaux *idiomatique* et *conception*.

Nos modèles et nos algorithmes sont décrits en pseudo-code. Nous utilisons le mécanisme de répartition multiple¹⁴ pour représenter et choisir les méthodes à exécuter. Le mécanisme de répartition multiple utilise tous les paramètres d'un appel de méthode pour choisir la méthode à exécuter. Il offre une plus grande sûreté de typage et une syntaxe plus concise.

Il est implanté dans des langages de programmation, tels COMMON LISP [Keene, 1989], CECIL [Chambers, 1992] et CLAIRE [Caseau et Laburthe, 1996], ou dans des extensions à d'autres langages de programmation, tel MULTIJAVA [Clifton *et al.*, 2000] pour JAVA.

Une implantation en JAVA des métamodèles et des algorithmes présentés est développée dans la partie III page 195. Nous ne faisons pas de preuves sur les algorithmes présentés, nous laissons ces preuves comme perspectives à nos travaux.

¹⁴Le mécanisme de répartition multiple est appelé *multi-dispatching* en anglais [Office québécois de la langue française, 2003].

Chapitre 3

Motifs interclasses

NOUS étudions les relations d'association, d'agrégation et de composition et nous proposons des modèles et des algorithmes pour construire le modèle d'un programme au niveau **idiomatique**.

Les relations interclasses sont utilisées au niveau **idiomatique** pour décrire l'architecture d'un programme avec des diagrammes de classes, par exemple le modèle du programme JHOTDRAW, figure 1.7(b) page 25, et pour représenter les motifs de conception, par exemple du motif de conception **Composite**, figure 1.7(c) page 25.

Les relations d'appel de méthodes, d'héritage et d'instanciation, pour des langages de programmation comme JAVA ou C++ sont bien définies et sont aisément identifiées dans le modèle d'un programme au niveau **implémentation**. Au contraire, les relations d'association, d'agrégation et de composition n'ont pas de définitions précises, elles sont décrites en termes vagues et sujettes à interprétation.

L'absence de définitions précises empêche la construction d'algorithmes pour identifier ces relations interclasses au niveau **implémentation** et pour garantir leur traçabilité entre les niveaux **implémentation** et **idiomatique**.

Les relations d'association, d'agrégation et de composition sont des motifs auxquels nous devons donner des définitions aux niveaux **implémentation** et **idiomatique**. Ces définitions doivent nous permettre de proposer des modèles et des algorithmes pour construire automatiquement le modèle d'un programme au niveau **idiomatique** et pour garantir la traçabilité des motifs interclasses.

Nos travaux sur la traçabilité des relations d'association, d'agrégation et de composition se décomposent comme suit :

- 3.1. Nous définissons les modèles utilisés pour décrire un programme aux niveaux **implémentation** et **idiomatique**. Nous présentons deux modèles liés au langage de programmation JAVA pour décrire le niveau **implémentation**. Nous proposons un métamodèle qui intègre les constituants nécessaires à la description de l'architecture d'un programme au niveau **idiomatique**.

- 3.2. Nous proposons des définitions des relations interclasses aux niveaux **implémentation** et **idiomatique**. Ces définitions sont basées sur l'état de l'art et sont aussi consensuelles que possible. Nous montrons que les relations d'association, d'agrégation et de composition sont des motifs au niveau **idiomatique**.
- 3.3. Nous isolons quatre propriétés communes aux définitions des motifs interclasses au niveau **implémentation**. Nous illustrons ces propriétés en termes des constituants des modèles d'un programme au niveau **implémentation**.
- 3.4. Nous formalisons les définitions des motifs interclasses au niveau **implémentation** avec les quatre propriétés précédemment proposées et nous montrons que ces quatre propriétés forment un sous-ensemble minimal des propriétés des motifs interclasses.
- 3.5. Nous décrivons un ensemble d'algorithmes pour calculer les valeurs des quatre propriétés et ainsi identifier les motifs interclasses au niveau **implémentation**.
- 3.6. Nous décrivons des algorithmes pour identifier et garantir la traçabilité des relations d'appel de méthodes, d'instanciation, d'héritage, d'association, d'agrégation et de composition entre les niveaux **implémentation** et **idiomatique**.
- 3.7. Nous vérifions la cohérence de nos algorithmes sur un ensemble de programmes bien connus et pour lesquels nous connaissons les motifs interclasses existants et nous discutons les modèles, les définitions et les algorithmes proposés.
- 3.8. Nous appliquons nos algorithmes pour garantir la traçabilité des motifs interclasses au programme JHOTDRAW.

Enfin, nous dressons un bilan de nos travaux sur la traçabilité des motifs interclasses.

3.1 Modélisation d'un programme

NOUS introduisons les formalismes utilisés pour décrire les niveaux **implémentation** et **idiomatique**. Au niveau **implémentation**, nous utilisons le langage de programmation JAVA pour décrire le modèle statique d'un programme et des traces d'exécution pour décrire ses modèles dynamiques.

Au niveau **idiomatique**, nous utilisons la technique de métamodélisation pour décrire un modèle d'un programme. Nous définissons un métamodèle qui inclut les constituants nécessaires à la représentation des relations interclasses.

C'est en terme des constituants de ces formalismes que nous définissons par la suite les motifs interclasses et les algorithmes garantissant leur traçabilité entre les niveaux **implémentation** et **idiomatique**.

3.1.1 Modèles du niveau **implémentation**

Au niveau **implémentation**, nous proposons l'utilisation de deux formalismes, l'un pour décrire le modèle statique d'un programme, l'autre pour décrire un modèle dynamique du programme.

Un modèle d'un programme peut représenter soit la séquence d'instructions qui est à exécuter par l'ordinateur, le *modèle statique** du programme ; soit la séquence d'opérations effectivement réalisées par l'ordinateur à l'exécution, un *modèle dynamique** du programme.

Il existe un unique modèle statique pour un programme car le programme est caractérisé par une et une seule séquence d'instructions, mais il existe plusieurs modèles dynamiques du programme car plusieurs chemins d'exécution existent potentiellement dans une séquence d'instructions.

Modèle statique Nous utilisons le langage de programmation JAVA comme formalisme pour décrire le modèle statique d'un programme au niveau **implémentation**. Le modèle d'un programme et les motifs interclasses sont représentés par du code source JAVA au niveau **implémentation**.

Nous choisissons le langage de programmation JAVA car il est très répandu et est utilisé pour implanter de nombreux programmes, aussi bien académiques qu'industriels [Seemann et von Gudenberg, 1998 ; Tatsubori, 1999 ; Thimbleby, 1999].

De plus, ce langage de programmation possède une syntaxe et une sémantique intermédiaires entre les langages de programmation C++ et SMALLTALK, nos algorithmes devraient ainsi pouvoir être adaptés à C++ et SMALLTALK.

Modèle dynamique Nous proposons un métamodèle construit sur le principe des traces d'exécution [Lieberman, 1987 ; Lieberman et Fry, 1995 ; Ducassé, 1999a ; Ducassé, 1999b]. Le métamodèle décrit l'exécution d'un programme comme une trace : une séquence d'*événements** d'exécution. Ces événements peuvent être de trois types :

- un événement d'affectation, *Affectation*, indique que l'instance d'une classe a été affectée dans un champ d'une instance d'une autre classe ;
- un événement de ramassage, *Ramassage*, indique qu'une instance d'une classe a été ramassée par le ramasse-miettes¹ ;
- un événement de terminaison, *Terminaison*, indique que l'exécution du programme vient de se terminer.

Nous considérons des traces *post-mortem* : nous supposons la séquence d'événements² modélisant l'exécution d'un programme complète.

Ce métamodèle propose les constituants nécessaires et suffisants pour garantir la traçabilité des motifs interclasses considéré dans ce mémoire.

3.1.2 Modèle du niveau **idiomatique**

Au niveau **idiomatique**, nous ne distinguons pas les modèles statiques et dynamiques d'un programme. Un modèle d'un programme au niveau **idiomatique** est un modèle global [Jackson et Rinard, 2000] qui représente l'architecture du programme et un sous-ensemble de son comportement. Nous ne limitons pas ainsi les informations présentes dans le modèle d'un programme à des informations uniquement statiques ou uniquement dynamiques [Sefika *et al.*, 1996 ; Richner et Ducasse, 1999].

Par exemple, au niveau **idiomatique**, le modèle d'un programme représente les classes et les relations d'héritage entre ces classes, son architecture, et les relations d'appel de méthodes et d'instanciation entre les instances de ses classes, un sous-ensemble de son comportement.

Nous proposons un métamodèle dédié à la modélisation globale d'un programme au niveau **idiomatique**. La technique de métamodélisation consiste en la définition d'un ensemble de constituants élémentaires pour construire un modèle du programme par instanciation et composition des instances.

La métamodélisation a déjà fait ses preuves pour décrire des modèles de programme et les motifs de conception [Pagel et Winter, 1996 ; Florijn *et al.*, 1997 ; Demeyer *et al.*, 1999a ; Sunyé, 1999 ; Eden, 2000 ; Rapicault et Fornarino, 2000 ; Sunyé *et al.*, 2000 ; Albin-Amiot, 2003] et nous pensons comme Dave Thomas que tout modèle a besoin d'un métamodèle³ [Thomas, 2002].

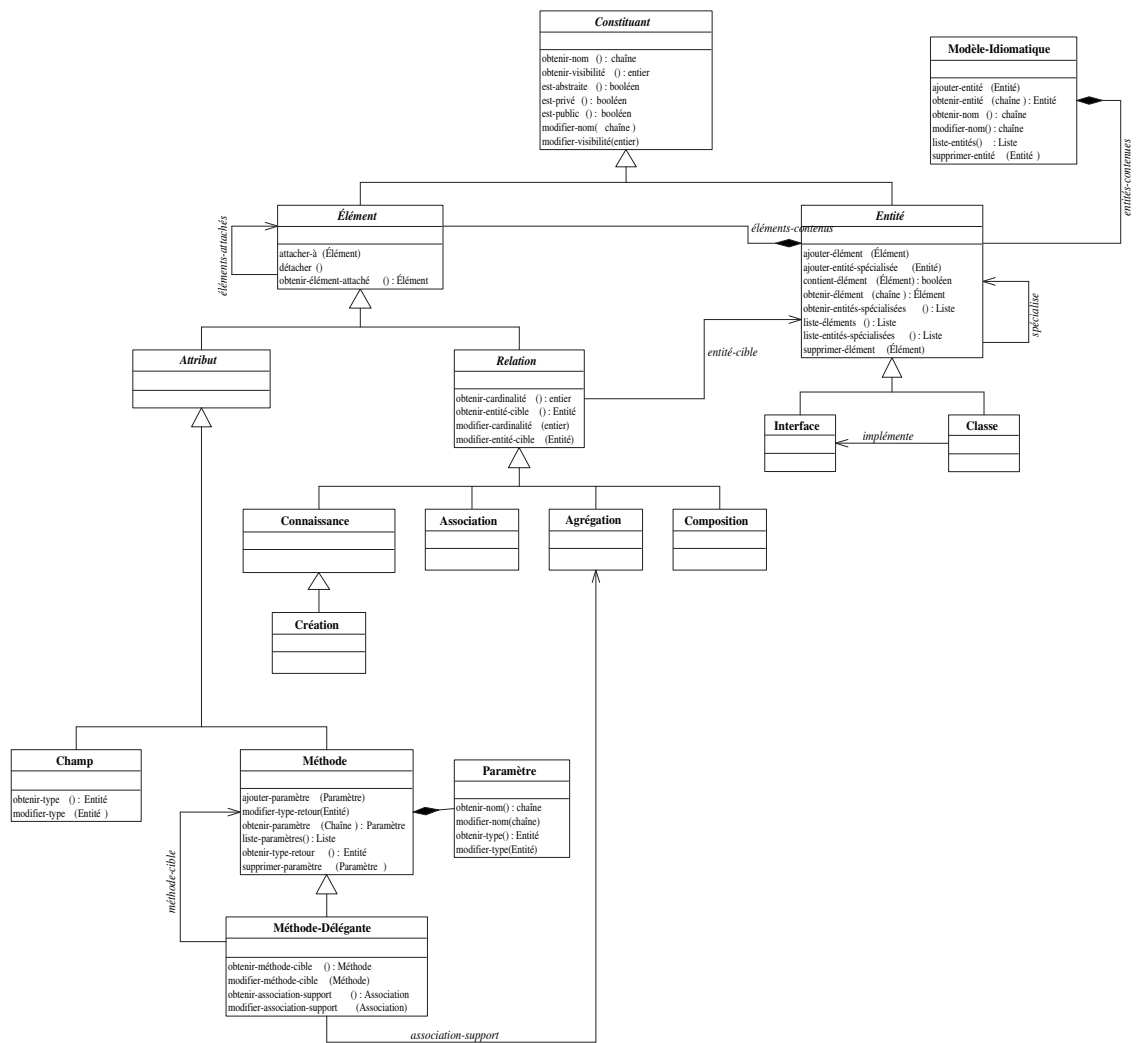
Notre principal objectif dans la définition d'un métamodèle est d'obtenir un langage de description des modèles de programmes au niveau **idiomatique** aussi concis que possible. Le métamodèle définit un ensemble de constituants (entités et éléments) et les règles régissant leurs interactions. Ces constituants décrivent l'architecture et un sous-ensemble du comportement d'un programme.

¹Cet événement correspond à l'appel au destructeur en C++.

²Nous ne considérons pas les événements d'invocations de messages, les invocations de messages sont décrites dans le modèle statique du programme.

³Dave Thomas [2002, page 18] déclare : “*Every model needs a meta model*”.

FIG. 3.1 – Sous-ensemble du métamodèle pour décrire l'architecture d'un programme au niveau idiomatique.



La figure 3.1 page 81 montre⁴ les principaux constituants du métamodèle et leurs relations. Par défaut, ce métamodèle définit les constituants suivants :

- **Entité**, pour représenter une entité du programme. Une entité peut être :
 - une **Classe**, pour représenter une classe du programme ;
 - une **Interface**, pour représenter une interface du programme ;
- **Élément**, pour représenter les attributs et les relations de ces entités. Un élément peut être :
 - une **Association**, **Agrégation** et **Composition** pour représenter les relations interclasses liant deux entités du programme.
 - une **Méthode**, pour représenter une méthode d'une classe ou d'une interface⁵ ;
 - un **Champ**, pour représenter un champ d'une classe ou d'une interface⁵ ;

Le modèle d'un programme prend la forme d'une instance de la classe **Modèle-Idiomatique**. Il est constitué d'une collection d'entités (instances du constituant **Entité**). Chaque entité est composée d'une collection d'éléments (instances du constituant **Élément**) représentant ses attributs. Ces entités et éléments décrivent l'architecture et un sous-ensemble du comportement du programme et ils sont ajoutés au métamodèle au fur et à mesure de la modélisation de nouvelles architectures ou de nouvelles relations interclasses, par spécialisation des classes **Entité** et **Élément** : l'héritage est utilisé pour produire une taxonomie des constituants de l'architecture d'un programme et des motifs interclasses.

Le métamodèle définit la sémantique des modèles qui en sont issus : le modèle d'un programme est composé d'une ou plusieurs classes et interfaces, instances respectivement de **Classe** et **Interface**, et sous-constituants de **Entité**. Une instance du constituant **Entité** peut contenir des méthodes (instances du constituant **Méthode**), des champs (instances du constituant **Champ**) et, plus généralement, n'importe quelle instance du constituant **Élément**, qu'il s'agisse d'une délégation (instance du constituant **Méthode-Délégante**) ou d'une relation (instance du constituant **Relation**).

Les opérations déclarées par les constituants définissent la sémantique du métamodèle :

- tous les constituants d'un modèle d'un programme au niveau **idiomatique** ont un nom (méthodes **obtenir-nom()** et **modifier-nom()**) et une visibilité⁶ (méthodes **obtenir-visibilité()**, **modifier-visibilité()**, **est-abstraite()**, **est-privé()** et **est-public()**) ;
- une entité est composée (relation de composition **éléments-contenus**) d'éléments qui représentent ses attributs (méthodes **ajouter-élément()**, **contient-élément (Élément)**, **obtenir-élément()**, **liste-éléments()** et **supprimer-élément (Élément)**) ;

⁴Les métamodèles sont représentés comme des modèles au niveau **idiomatique** et utilisent la notation graphique proche de UML proposée sur la figure 3.2 page 84.

⁵Nous modélisons aussi les méthodes et les champs en plus des relations interclasses car il est possible que des méthodes et des champs définissent des relations interclasses particulières.

⁶Nous considérons une notion de visibilité des constituants du métamodèle telle qu'elle existe en JAVA car nous utilisons le métamodèle pour modéliser des programmes décrits avec ce langage de programmation au niveau **implémentation**. Cependant, cette notion de visibilité est limitée, comme discuté dans [Ardourel et al., 2003].

- un élément peut être attaché à un autre élément (méthodes `attacher-à()`, `détacher()` et `obtenir-élément-attaché()`), pour lier les éléments entre eux, par exemple pour lier une méthode abstraite et la méthode concrète qui l'implante;
- un paramètre possède un nom (méthodes `obtenir-nom()` et `modifier-nom()`) et un type (méthodes `obtenir-type()` et `modifier-type()`);
- une méthode (ou un constructeur) possède un ensemble de paramètres (méthodes `ajouter-paramètre()`, `obtenir-paramètre()`, `liste-paramètres()` et `supprimer-paramètre()`) et un type de retour (méthodes `modifier-type-retour()` et `obtenir-type-retour()`) (le type de retour est `null` dans le cas d'un constructeur);
- une méthode délégante est une méthode (spécialisation du constituant `Méthode`) qui délègue son traitement à une autre méthode (association `méthode-cible` et méthodes `obtenir-méthode-cible()` et `modifier-méthode-cible()`) au travers d'une relation d'association entre leurs classes de déclaration respectives (association `association-support` et méthodes `obtenir-association-support()` et `modifier-association-support()`);
- un champ possède un type (méthodes `obtenir-type()` et `modifier-type()`);
- les relations ont une cible (méthodes `obtenir-entité-cible()` et `modifier-entité-cible()`) et une cardinalité (pour leur cible) (méthodes `obtenir-cardinalité()` et `modifier-cardinalité()`). Les relations interclasses spécialisent le constituant `Relation`.

Nous utilisons une notation graphique proche de UML pour représenter visuellement les modèles obtenus de ce métamodèle. Les modèles présentés sur la figure 3.2 page 84 montrent la correspondance entre les constituants du métamodèle et la notation graphique.

Conclusion. Un programme au niveau `implémentation` est représenté par un modèle statique en JAVA et des modèles dynamiques décrits par des traces de son exécution. Au niveau `idiomatique`, nous proposons un métamodèle pour décrire un modèle global du programme. Nous étudions maintenant les relations interclasses aux niveau `implémentation` et `idiomatique` et nous montrons qu'elles peuvent se réécrire sous la formes de motifs avec les formalismes proposés pour le niveau `implémentation`.

FIG. 3.2 – Notation pour représenter un modèle au niveau idiomatique.

(a) Une entité, ses méthodes et ses champs ; respectivement instances des constituants **Entité**, **Champ** et **Méthode** :

<Nom>
<nom du champ> : <type> = < valeur initiale >
<nom d'une méthode > (< liste des arguments >) : <type de retour >

(b) Relation d'association, instance du constituant **Association** :



(c) Relation d'agrégation, instance du constituant **Agrégation** :



(d) Relation de composition, instance du constituant **Composition** :



□

3.2 Définitions des motifs

Nous avons constaté dans l'état de l'art que les définitions des relations d'association, d'agrégation et de composition sont imprécises et ont des sémantiques différentes suivant les auteurs ; aussi, nous en proposons des définitions aussi consensuelles que possible, aux niveaux implémentation et idiomatique.

Nous donnons des exemples de nos définitions et nous montrons que les définitions des relations interclasses aux niveaux implémentation peuvent se réécrire sous la forme de motifs interclasses.

Le problème de la traçabilité des relations interclasses se traduit en un problème de traçabilité des motifs interclasses **Association**, **Agrégation** et **Composition** entre les niveaux implémentation et idiomatique.

3.2.1 Définitions des relations

Définition 3.1 – Relation d'association au niveau idiomatique.

Une relation d'*association*^{*} au niveau idiomatique est un lien de dépendance entre deux classes. Chaque classe peut avoir de multiples associations. □

La figure 2.10(a) page 47 représente une relation d'association entre deux classes A

et B, au niveau **idiomatique**, en utilisant le métamodèle et la notation définis section 3.1 page 79.

Définition 3.2 – Relation d’association au niveau **implémentation**.

Une relation binaire entre deux classes lie les instances des deux classes. Une relation binaire est orientée, irréflexive, anti-symétrique au niveau des instances et des classes, et asymétrique au niveau des instances [Henderson-Sellers et Barbier, 1999]. Une relation d’association entre deux classes A et B indique qu’une instance de A peut envoyer un message à une instance de B, avec la possibilité d’associations mutuelles entre les instances. □

Le code source 3.1 représente un exemple de relation d’association entre deux classes A et B, au niveau **implémentation**.

Code source 3.1 – Implantation d’une relation d’association.

La classe A est en relation d’association avec la classe B.

```
public class A {  
    public void operation1(B b) {  
        b.operation2();  
    }  
}  
public class B {  
    public void operation2() {  
    }  
}
```

La relation d’association entre A et B existe au travers de l’utilisation d’une instance de B comme paramètre de la méthode `operation1()` et de l’invocation de la méthode `operation2()` sur ce paramètre. □

Définition 3.3 – Relation d’agrégation au niveau **idiomatique**.

Une relation d’*agrégation*^{*} est une relation binaire entre deux classes, respectivement le tout et la partie. Conceptuellement, une partie n’a pas d’existence en dehors de son tout. □

La figure 2.10(b) page 47 représente une relation d'agrégation entre deux classes **A** et **B**, au niveau idiomatique.

Définition 3.4 – Relation d'agrégation au niveau implémentation.

Une relation d'agrégation existe quand la définition d'une classe contient des instances d'une autre classe. Elle distingue un tout (la classe contenant) d'une partie partie (la classe dont une ou plusieurs instances sont contenues). Le tout définit un champ du type de la partie (ou un tableau ou une collection). Les instances du tout envoient des messages aux instances de la partie référencées. Les sous-classes héritent de la relation d'agrégation car, dans les langages à classes, les sous-classes héritent du comportement et de la structure de leurs superclasses. Par exemple, si une relation d'agrégation existe entre deux classes **A** et **B**, alors une relation d'agrégation existe aussi entre les classes **A** et **SubB**, où **SubB** est une sous-classe de **B**. □

Le code source 3.2 représente une relation d'agrégation entre deux classes **A** et **B**, au niveau implémentation.

Code source 3.2 – Implantation d'une relation d'agrégation.

La classe **A** agrège une instance de la classe **B**.

```
public class A {  
    public B b;  
    public A(B b) {  
        this.b = b;  
    }  
    public void operation1() {  
        this.b.operation2();  
    }  
}  
public class B {  
    public void operation2() {  
    }  
}
```

La relation d'agrégation entre **A** et **B** existe au travers du champ **B b** et de l'utilisation de ce champ dans la méthode `operation1()`. □

Définition 3.5 – Relation de composition au niveau idiomatique.

Une relation de *composition*^{*} est une relation d'agrégation particulière pour laquelle les parties contenues dans le tout disparaissent quand le tout disparaît. Les parties peuvent changer durant la vie du tout, mais toutes les parties contenues par le tout au moment de sa disparition disparaissent simultanément. \square

La figure 2.10(c) page 47 représente une relation de composition entre deux classes A et B, au niveau idiomatique.

Définition 3.6 – Relation de composition au niveau implémentation.

Une relation de composition est une relation d'agrégation entre deux classes, avec une contrainte de coïncidence de durée de vie et une contrainte d'appartenance unique entre les instances du tout et les instances de la partie. Le tout peut créer les instances de la partie ou peut accepter des instances déjà créées. Un tout peut passer une des instances de sa partie à un autre tout, qui alors en assume la responsabilité. Quand un tout disparaît, toutes les instances de sa partie disparaissent également. Une instance d'un tout est propriétaire des instances de sa partie. Une instance de la partie ne peut appartenir à un autre tout, que ce soit au travers d'une relation d'agrégation ou de composition. L'instance de la partie appartient exclusivement à l'instance du tout. \square

Le code source 3.3 représente une relation de composition entre deux classes A et B, au niveau implémentation, en utilisant le langage de programmation JAVA, défini comme formalisme pour le niveau idiomatique.

Code source 3.3 – Implantation d’une relation de composition en JAVA.

La classe **A** est composée d’une instance de la classe **B**.

```
public class A {  
    private B b = new B();  
    public void operation1() {  
        this.b.operation2();  
    }  
}  
public class B {  
    public void operation2() {  
    }  
}
```

La relation de composition entre **A** et **B** existe au travers du champ **B b**, de l’utilisation de la méthode **operation1()** et du ramasse-miettes de la machine virtuelle **JAVA** : le ramasse-miettes ramasse l’instance de **B** avant l’instance de **A**. Le modificateur privé du champ **B b** et l’absence de méthode retournant une référence sur ce champ contraignent la durée de vie et l’exclusivité entre les instances de **A** et **B**. Il est impossible d’obtenir une référence sur le champ **B b**, référencé dans la classe **A**. □

Le code source 3.4 représente une relation de composition entre deux classes A et B, au niveau **implémentation**, pour le langage de programmation C++.

Code source 3.4 – Implantation d’une relation de composition en C++.

La classe A est composée d’une instance de la classe B.

```
class A {
private:
    B b* = new B();
public:
    void operation1() {
        b->operation2();
    }
    ~A() {
        delete b;
    }
}
class B {
public:
    void operation2() {
    }
    ~B() {
    }
}
```

La relation de composition entre A et B existe au travers du champ B **b** et du destructeur **~A()** : à la destruction d’une instance de la classe A, l’instance de la classe B est aussi détruite. Le modificateur privé pour le champ B **b** et l’absence de méthode retournant une référence sur ce champ contraignent l’exclusivité entre les instances de A et B. Il est impossible d’obtenir une référence sur le champ B **b**, référencé dans la classe A, l’encapsulation complète de l’instance de B dans A et les contraintes de durée de vie et d’exclusivité sont donc assurées.

□

3.2.2 Définitions des motifs

Nous avons défini les relations interclasses au niveau **implémentation** et **idiomatique**. Les définitions des relations d’association, d’agrégation et de composition au niveau **idiomatique** correspondent à des constituants uniques du métamodèle de ce niveau. Les définitions des relations interclasses au niveau **implémentation** correspondent à plusieurs constituants du formalisme de ce niveau.

Les relations interclasses au niveau **idiomatique** sont des *motifs*^{*} qui abstraient et rassemblent des constituants du formalisme du niveau **implémentation**. Nous parlons de motifs

interclasses et nous nommons les relations d'association, d'agrégation et de composition, respectivement, motif **Association**, **Agrégation** et **Composition**.

Le problème de l'identification des relations interclasses se traduit alors en un problème de traçabilité des motifs **Association**, **Agrégation** et **Composition** entre les niveaux **implémentation** et **idiomatique**.

Conclusion. Nous avons défini les relations interclasses informellement et nous en avons donné de simples exemples au niveau **implémentation**. Nous avons montré que les relations interclasses sont des motifs au niveau **implémentation**. Le problème de la traçabilité des relations interclasses se traduit en un problème de traçabilité de motifs. Nous isolons maintenant quatre propriétés communes aux motifs interclasses au niveau **implémentation** : durée de vie, exclusivité, multiplicité et site d'invocation.

3.3 Propriétés des motifs

Nous isolons un ensemble de quatre *propriétés** communes aux définitions informelles des motifs interclasses au niveau **implémentation**. Nous donnons des exemples de ces propriétés au niveau **implémentation**.

3.3.1 Définitions des propriétés

Les motifs interclasses possèdent, au niveau **implémentation**, les propriétés suivantes.

Propriété 3.1 – Durée de vie.

Cette propriété contraint les *durées de vie** des instances des classes dans une relation **A**–**B**. La durée de vie [Civello, 1993 ; Bonnano *et al.*, 1996] $DV(\text{Classe})$ d’une instance est le temps passé entre son instanciation $DV_i(\text{Classe}) \in \mathbb{N}$ et sa destruction $DV_d(\text{Classe}) \in \mathbb{N}$: $DV(\text{Classe}) = DV_d(\text{Classe}) - DV_i(\text{Classe}) \in \mathbb{N}$.

Nous définissons la durée de vie entre deux classes **A** et **B** par :

$$DV(\mathbf{A}, \mathbf{B}) = DV_d(\mathbf{A}) - DV_d(\mathbf{B}) \in \{-, +\}$$

Dans les langages de programmation avec ramasse-miettes, comme JAVA, $DV_d(\text{Classe})$ correspond au moment où le ramasse-miettes ramasse la première instance de **Classe**.

Dans les langages de programmation avec destructeurs explicites, comme C++, $DV_d(\text{Classe})$ correspond au moment où le destructeur est appelé. Nous nommons \parallel l’ensemble $\{+, -\}$ pour indiquer que les durées de vie des instances de deux classes ne sont pas liées, que la durée de vie entre les deux classes peut être indifféremment $+$ ou $-$. \square

Exemple 3.1 – Exemples $+$ et $-$ de la propriété de durée de vie.

Dans une relation entre une fenêtre et un bouton dans un système de fenêtrage, représentée par **Fenêtre**–**Bouton**, le moment de la destruction d’un bouton de la fenêtre ne peut dépasser le moment de la destruction de la fenêtre le contenant : $DV(\text{Fenêtre}, \text{Bouton}) = +$ et $DV(\text{Bouton}, \text{Fenêtre}) = -$. \square

Propriété 3.2 – Exclusivité.

L'*exclusivité*^{*} indique si une instance d'une classe **B** dans une relation **A–B** participe (*faux*) ou non (*vrai*) dans une autre relation à un moment donné.

$$EX(\mathbf{A}, \mathbf{B}) \in \{vrai, faux\}$$

Nous définissons *indifférent* comme l'ensemble $\{vrai, faux\}$ pour indiquer que la propriété d'exclusivité entre deux classes peut être indifféremment *vrai* ou *faux*. \square

Exemple 3.2 – Exemple *faux* de la propriété d'exclusivité.

Dans une relation entre des pays et des langues, représentée par **Pays–Langue**, une langue (instance de **Langue**) peut appartenir à plus d'un pays (instance de **Pays**) :

$$EX(\mathbf{Pays}, \mathbf{Langue}) = faux.$$

\square

Exemple 3.3 – Exemple *vrai* de la propriété d'exclusivité.

Dans une relation entre l'unité centrale d'un ordinateur et un clavier, représentée par **Ordinateur–Clavier**, un clavier (instance de **Clavier**) n'appartient qu'à une unité centrale (instance de **Ordinateur**) à un moment donné : $EX(\mathbf{Ordinateur}, \mathbf{Clavier}) = vrai$. La propriété d'exclusivité n'a de sens que pour un moment donné : elle n'interdit pas les changements. Ainsi, dans la relation **Ordinateur–Clavier**, un clavier n'appartient qu'à une et une seule unité centrale à un moment donné, et peut être utilisé pour une autre unité centrale à un autre moment. \square

Propriété 3.3 – Multiplicité.

La propriété de *multiplicité*^{*} dans une relation **A**–**B** indique le nombre d’instances de la classe **B** en relation avec une instance de la classe **A**. Pour plus de simplicité, nous utilisons l’intervalle du plus petit nombre et du plus grand nombre d’instances autorisées.

$$MU(\mathbf{A}, \mathbf{B}) \subset \mathbb{N} \cup \{+\infty\}$$

Nous nous intéressons uniquement à la multiplicité de la classe cible de la relation, Daniel Jackson et Allison Waingold [1999] discutent de la multiplicité pour la source et la cible. \square

Exemple 3.4 – Exemples de la propriété de multiplicité.

Dans une relation entre une licorne et sa corne, représentée par **Licorne**–**Corne**, une licorne ne possède qu’une et une seule corne : $MU(\mathbf{Licorne}, \mathbf{Corne}) = [1, 1]$.

Dans une relation entre une voiture et ses roues, représentée par **Voiture**–**Roue**, une voiture possède habituellement quatre roues et peut avoir une roue de secours : $MU(\mathbf{Voiture}, \mathbf{Roue}) = [4, 5]$.

Une image vectorielle peut être composée de zéro ou d’une infinité de lignes, la relation **Image**–**Ligne** a pour multiplicité : $MU(\mathbf{Image}, \mathbf{Ligne}) = [0, +\infty]$. \square

Propriété 3.4 – Site d’invocation.

La propriété de *site d’invocation*^{*} indique, dans une relation **A**–**B**, si les instances de la classe **A** envoient des messages aux instances de la classe **B** et précise les sites d’invocation des méthodes correspondant à ces envois de messages : un champ, un champ de type tableau, un champ de type collection, le paramètre d’une méthode ou une variable locale à une méthode. La valeur \emptyset indique que les instances de la classe **A** n’envoient pas de messages aux instances de la classe **B**.

$$SI(\mathbf{A}, \mathbf{B}) \subset \emptyset \cup \{\text{champ}, \text{champ tableau}, \text{champ collection}, \text{paramètre}, \text{variable locale}\}$$

Nous nommons *oui* l’ensemble $\{\text{champ}, \text{champ tableau}, \text{champ collection}, \text{paramètre}, \text{variable locale}\}$ et *indifférent* l’ensemble $\emptyset \cup \{\text{champ}, \text{champ tableau}, \text{champ collection}, \text{paramètre}, \text{variable locale}\}$. \square

Exemple 3.5 – Exemples de la propriété de site d’invocation.

Dans une relation entre une figure et un rectangle, représentée par **Figure**–**Rectangle**, la figure (instance de **Figure**) propage le message `dessine()` à ses rectangles (instances de la classe **Rectangle**, sans préciser les sites d’invocation des méthodes) : $SI(\mathbf{Figure}, \mathbf{Rectangle}) \subset \{\text{champ}, \text{champ tableau}, \text{champ collection}, \text{paramètre}, \text{variable locale}\}$ ou $SI(\mathbf{Figure}, \mathbf{Rectangle}) \subset \text{oui}$. Mais, un rectangle ne doit pas connaître la figure qui le contient et lui envoyer de message : $SI(\mathbf{Rectangle}, \mathbf{Figure}) = \emptyset$ [Blake et Cook, 1987]. \square

3.3.2 Discussion des définitions

Anti-symétrie de la propriété du durée de vie Par définition, la propriété de durée de vie est anti-symétrique : $DV(A, B) = + \Leftrightarrow DV(B, A) = -$. Nous rappelons les valeurs respectives de $DV(A, B)$ et de $DV(B, A)$ dans le reste de ce mémoire uniquement pour lever toute ambiguïté.

Relations entre propriétés Les quatre propriétés que nous proposons pour exprimer les motifs interclasses sont orthogonales. Cependant, les propriétés d'exclusivité et de multiplicité sont proches l'une de l'autre. Par exemple, dans une relation **Pays**–**Langue** :

- la propriété de multiplicité indique le nombre d'instances de la classe **Langue** que possède chaque instance de la classe **Pays** : $MU(\text{Pays}, \text{Langage}) = [1, +\infty]$. L'Italie a une langue principale, l'italien, et de nombreuses langues secondaires, comme le ladin, l'allemand, et le turc ;
- la propriété d'exclusivité indique si une instance de la classe **Langue** peut être partagée par plusieurs instances de la classe **Pays** ou d'autres classes : $EX(\text{Pays}, \text{Langage}) = faux$. Des habitants d'Allemagne et d'Italie parlent allemand.

Sous-typage Les propriétés entre deux classes sont héritées par leurs sous-classes car, dans les langages à classes, les sous-classes héritent du comportement et de la structure de leurs superclasses. Nous pouvons écrire, par exemple pour la relation d'exclusivité : si A' et B' sont des sous-classes de A et B respectivement, alors :

$$EX(A', B') = EX(A, B)$$

3.3.3 Exemples au niveau implémentation

Nous donnons maintenant des exemples⁷ d'implantation des quatre propriétés au niveau implémentation.

Exemple 3.6 – Durée de vie $DV(\text{Classe})$.

Le code source ci-dessous illustre la dépendance des durées de vie entre les instances de deux classes A et B :

```
1 public class Lifetime {
2     public static void main(String[] args) {
3         A a = new A(new B());
4         a.operation();
5     }
6 }
7 public class A {
8     private B b;
9     public A(B b) {
10         super();
11         this.b = b;
12     }
13     public void operation() {
14         b.operation();
15     }
16 }
17 public class B {
18     public void operation() {
19     }
20 }
```

L'instance de la classe B est exclusive à l'instance de la classe A. Lorsque le ramasse-miettes de la machine virtuelle ramasse l'instance de la classe A, il a d'abord ramassé l'instance de la classe B : $DV(A, B) = +$.

□

⁷Le code source de ces exemples est disponible à <http://www.yann-gael.gueheneuc.net/Work/Research/PhDThesis/>.

Exemple 3.7 – Exclusivité $EX(A, B)$.

Le code source ci-dessous illustre la propriété d'exclusivité :

```
1 public class Exclusivity {
2     public static void main(String[] args) {
3         Exclusivity e1 = new Exclusivity();
4         Exclusivity e2 = new Exclusivity();
5         A a = new A();
6         e1.operation1(a);
7         e2.operation1(a);
8         a.operation3(new B());
9         ...
10    }
11    public void operation1(A a) {
12        a.operation2();
13    }
14 }
15 public class A {
16     public void operation2() {
17     }
18     public void operation3(B b) {
19         b.operation();
20     }
21 }
22 public class B {
23     public void operation4() {
24     }
25 }
```

Dans cet exemple, l'instance de la classe **A** est partagée par plusieurs instances de la classe **Exclusivity**, lignes 6–7 : $EX(\text{Exclusivity}, A) = \text{faux}$, tandis que l'instance de la classe **B** est exclusive à la méthode **operation3()**, ligne 8 : $EX(A, B) = \text{vrai}$. \square

Exemple 3.8 – Multiplicité $MU(A, B)$.

Les valeurs de la propriété de multiplicité est illustrée par le code source suivant :

```
1 public class A {  
2     private B b;  
3     private C c = new C();  
4     private D[] d;  
5     private E[] e = new E[] { new E() };  
6 }
```

Dans cet exemple, une instance de la classe **A** possède différentes relations avec les classes **B**, **C**, **D**, et **E**. Les multiplicités des relations entre ces classes sont : $MU(A, B) = [0, 1]$, ligne 2, $MU(A, C) = [1, 1]$, ligne 3, $MU(A, D) = [0, +\infty]$, ligne 4, et $MU(A, E) = [1, 1]$, ligne 5. \square

Exemple 3.9 – Site d’invocation $SI(A, B)$.

Le code source suivant présente des exemples de valeurs de la propriété de site d’invocation :

```

1  public class MessageSend {
2      public static void main(String[] args) {
3          A a = new A();
4          a.operation1(new B());
5          a.operation3();
6      }
7  }
8  public class A {
9      private C c = new C();
10     public void operation1(B b) {
11         b.operation2();
12     }
13     public void operation3() {
14         c.operation4();
15     }
16 }
17 public class B {
18     public void operation2() {
19     }
20 }
21 public class C {
22     public void operation4() {
23     }
24 }
```

Dans ces exemples, une instance de la classe **A** peut envoyer un message à une instance de la classe **B**, par l’intermédiaire du paramètre **B b**, ligne 4 : $SI(A, B) = \{param\grave{e}tre\}$. Une instance de la classe **A** peut aussi envoyer un message à une instance de la classe **C** par l’intermédiaire du champ **C c**, ligne 7 : $SI(A, C) = \{champ\}$. Les instances des classes **B** et **C** ne peuvent s’envoyer mutuellement des messages ou envoyer des messages aux instances de la classe **A**, lignes 11–18 : $SI(B, A) = SI(C, A) = SI(B, C) = SI(C, B) = \emptyset$. \square

Conclusion. Nous avons isolé quatre propriétés communes aux définitions des motifs interclasses au niveau implémentation : exclusivité, durée de vie, multiplicité et site d’invocation. Nous avons montré comment ces quatre propriétés sont représentées au niveau implémentation avec des extraits de code source JAVA. Ces propriétés ne sont pas exhaustives [Civello, 1993 ; Henderson-Sellers et Barbier, 1999]. Nous définissons maintenant les motifs interclasses en utilisant ces quatre propriétés et nous montrons qu’elles sont minimales.

3.4 Formalisations des motifs avec leurs propriétés

Nous définissons les motifs interclasses **Association**, **Agrégation** et **Composition** au niveau **implémentation** comme des conjonctions, $AS(A, B)$, $AG(A, B)$ et $CO(A, B)$, des valeurs de leurs propriétés de durée de vie, d'exclusivité, de multiplicité et de site d'invocation.

Nous discutons plusieurs caractéristiques de ces définitions, dont la minimalité des quatre propriétés, et nous donnons des exemples au niveau **implémentation** des motifs interclasses avec leurs quatre propriétés.

3.4.1 Définitions des motifs

La définition informelle de la relation d'association entre deux classes **A** et **B** décrit qu'une instance de **A** peut envoyer un message à une instance de **B**, avec la possibilité d'associations mutuelles entre les instances.

Ainsi, la valeur de la propriété de site d'invocation est $SI(A, B) \subset \text{oui}$ et les autres propriétés peuvent prendre indifféremment une valeur de leur domaine de définition, comme montré par la définition 3.7.

Définition 3.7 – Motif Association $AS(A, B)$.

Le motif **Association** entre deux classes **A** et **B**, $AS(A, B)$, est :

$$\begin{aligned}
 AS(A, B) \triangleq & \\
 & (DV(A, B) \in \parallel) \wedge (DV(B, A) \in \parallel) \wedge \\
 & (EX(A, B) \in \text{indifférent}) \wedge (EX(B, A) \in \text{indifférent}) \wedge \\
 & (MU(A, B) = [0, +\infty]) \wedge (MU(B, A) = [0, +\infty]) \wedge \\
 & (SI(A, B) \subset \text{oui}) \wedge (SI(B, A) \subset \text{indifférent}) \\
 & \in \{ \text{vrai}, \text{faux} \}
 \end{aligned}$$

□

Une relation d'agrégation existe quand la définition d'une classe, le tout, contient des instances d'une autre classe, sa partie. Le tout définit un champ du type de la partie (ou un tableau ou une collection). Les instances du tout envoient des messages aux instances de la partie correspondantes.

Ainsi, les valeurs de la propriété de multiplicité sont $MU(A, B) = [1, +\infty]$ car un tout a au moins une partie, et $MU(B, A) = [0, +\infty]$ car une instance d'une partie peut exister en dehors d'un tout ou appartenir à plusieurs tous.

Les valeurs de la propriété de site d'invocation sont $SI(A, B) = \{champ, champ\ tableau, champ\ collection\}$ car les instances du tout envoient des messages aux instances de la partie par l'intermédiaire d'un champ, d'un tableau ou d'une collection, et $SI(B, A) = \emptyset$ car la partie ne doit pas connaître son tout.

De plus, les définitions des motifs Agrégation et Composition, sections 3.2.1 et 3.2.2, interdisent l'existence d'une relation de composition entre les classes B et A, $\neg CO(B, A)$, comme montré par la définition 3.8.

Définition 3.8 – Motif Agrégation $AG(A, B)$.

Le motif Agrégation entre deux classes A et B, $AG(A, B)$, est :

$$\begin{aligned}
 AG(A, B) \triangleq & \\
 & (DV(A, B) \in ||) \wedge (DV(B, A) \in ||) \wedge \\
 & (EX(A, B) \in indifférent) \wedge (EX(B, A) \in indifférent) \wedge \\
 & (MU(A, B) = [1, +\infty]) \wedge (MU(B, A) = [0, +\infty]) \wedge \\
 & (SI(A, B) = \{champ, champ\ tableau, champ\ collection\}) \wedge (SI(B, A) = \emptyset) \\
 & \wedge \neg CO(B, A) \\
 & \in \{vrai, faux\}
 \end{aligned}$$

□

Une relation de composition est une relation d'agrégation entre deux classes, avec une contrainte de coïncidence des durées de vie et une contrainte d'appartenance unique entre les instances du tout et les instances de la partie.

Quand l'instance d'un tout est détruite, les instances correspondantes de sa partie sont également détruites. Une instance d'un tout est propriétaire des instances de sa partie. Une instance de la partie ne peut appartenir à un autre tout, que ce soit au travers d'une relation d'agrégation ou de composition.

Ainsi, les valeurs de la propriété de durée de vie sont $DV(A, B) = +$ et $DV(B, A) = -$ car les instances de la classe A, le tout, sont détruites après les instances de la classe B, la partie : les instances de la partie appartiennent jusqu'à leur destruction à un tout.

Les valeurs de la propriété d'exclusivité sont $EX(A, B) = vrai$ car les instances de la partie appartiennent exclusivement à l'instance du tout, et $EX(B, A) \in indifférent$ car un tout peut avoir plusieurs parties.

Les valeurs de la propriété de multiplicité sont $MU(A, B) = [1, +\infty]$ car un tout a au moins une partie, et $MU(B, A) = [1, 1]$ car une instance d'une partie appartient à une et une seule instance du tout.

Les valeurs de la propriété de site d'invocation sont $SI(A, B) = \{champ, champ\ tableau, champ\ collection\}$ car les instances du tout envoient des messages aux instances de la partie par l'intermédiaire d'un champ, d'un tableau ou d'une collection, et $SI(B, A) = \emptyset$ car la partie ne doit pas connaître son tout.

De plus, les définitions des motifs Agrégation et Composition, sections 3.2.1 et 3.2.2, interdisent l'existence d'une relation d'agrégation ou de composition entre les classes B et A, $\neg AG(B, A) \wedge \neg CO(B, A)$, comme montré par la définition 3.9.

Définition 3.9 – Motif Composition $CO(A, B)$.

Le motif de composition entre deux classes A et B, $CO(A, B)$, est :

$$\begin{aligned}
 CO(A, B) \triangleq & \\
 & (DV(A, B) = +) \wedge (DV(B, A) = -) \wedge \\
 & (EX(A, B) = vrai) \wedge (EX(B, A) \in indifférent) \wedge \\
 & (MU(A, B) = [1, +\infty]) \wedge (MU(B, A) = [1, 1]) \wedge \\
 & (SI(A, B) = \{champ, champ\ tableau, champ\ collection\}) \wedge (SI(B, A) = \emptyset) \\
 & \wedge \neg AG(B, A) \wedge \neg CO(B, A) \\
 & \in \{vrai, faux\}
 \end{aligned}$$

□

3.4.2 Discussion des définitions

Relation mutuelle et pointeur en arrière La présence d'un motif interclasse entre deux classes A et B n'empêche pas l'existence d'autres relations entre les deux classes, sauf quand de telles relations sont spécifiquement interdites par les définitions des motifs, comme par exemple pour le motif **Composition**.

Les relations entre deux classes A et B peuvent induire un cycle. Par exemple, il y a un cycle quand un pointeur en arrière mémorise l'instance propriétaire d'une instance ou quand deux classes déclarent des champs du type de l'autre, comme sur l'extrait de code source suivant :

```
public class A {
    private B component ;
    ...
}
public class B {
    private A component ;
    ...
}
```

Dans un tel cas, deux motifs **Agrégation** symétriques existent : $AG(A, B)$ et $AG(B, A)$.

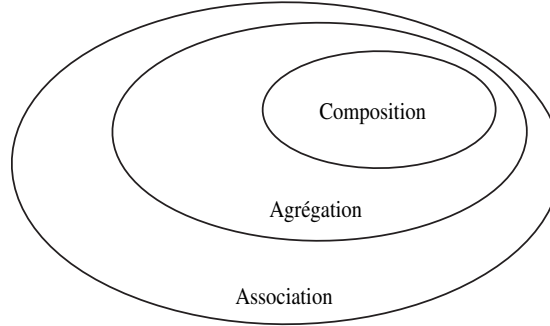
Relation d'association et optimisation Pour des raisons d'optimisation des appels de méthodes entre instances, une instance d'une classe A associée avec une instance d'une classe B peut déclarer une champ de type B pour mémoriser une instance de la classe B et optimiser l'accès à cette instance.

D'après nos définitions, la relation d'association entre A et B évolue alors d'un motif **Association** en un motif **Agrégation**. Cette évolution illustre l'identification et la traçabilité des motifs *existants* entre les niveaux *implémentation* et *idiomatique* pour aider ainsi les mainteneurs à comprendre l'architecture *réelle* des programmes.

Relation d'ordre entre les motifs Les définitions des motifs interclasses avec leurs propriétés montrent qu'il existe une relation d'inclusion entre les motifs, du plus contraignant au moins contraignant, comme représenté sur la figure 3.3 page 105.

Cette relation d'inclusion reflète l'intuition que les relations d'association, d'agrégation et de composition peuvent être ordonnées de la plus générale, la relation d'association, à la plus particulière, la relation de composition.

FIG. 3.3 – Relation d'inclusion entre les motifs interclasses.



□

Les valeurs des propriétés du motif **Agrégation** sont incluses dans les valeurs des propriétés du motif **Association**, donc plus contraignantes :

$$\begin{array}{lcl}
 \text{Motif Agrégation} & & \text{Motif Association} \\
 (MU(A, B) = [1, +\infty]) & \subset & (MU(A, B) = [0, +\infty]) \\
 (SI(A, B) = \{champ, & \subset & (SI(A, B) = oui) \\
 champ\ tableau, champ\ collection\}) & & \\
 (SI(B, A) = \emptyset) & \subset & (SI(B, A) = indifférent)
 \end{array}$$

Les valeurs des propriétés du motif **Composition** sont incluses dans les valeurs des propriétés du motif **Agrégation**. En particulier, la propriété d'exclusivité est plus contraignante pour la classe jouant le rôle de partie dans le motif **Composition** que dans le motif **Agrégation** : dans une relation de composition, la partie ne doit appartenir à aucune autre relation d'agrégation ou de composition :

$$\begin{array}{lcl}
 \text{Motif Composition} & & \text{Motif Agrégation} \\
 (DV(A, B) = +) & \in & (DV(A, B) \in ||) \\
 (DV(B, A) = -) & \in & (DV(A, B) \in ||) \\
 (EX(A, B) = vrai) & \in & (EX(A, B) \in indifférent) \\
 (MU(B, A) = [1, 1]) & \subset & (MU(B, A) = [0, +\infty])
 \end{array}$$

Propriétés minimales Les quatre propriétés que nous utilisons pour définir les motifs interclasses au niveau **implémentation** forment un sous-ensemble des propriétés des définitions présentées dans l'état de l'art, par exemple [Civello, 1993 ; Henderson-Sellers et Barbier, 1999].

Ce sous-ensemble des propriétés, $\mathcal{E} = \{exclusivité, durée\ de\ vie, multiplicité, site\ d'invocation\}$, est minimal. Nous établissons que l'ensemble \mathcal{E} est minimal en deux phases :

1. Nous montrons que si nous retirons une propriété de l'ensemble \mathcal{E} , nos définitions des motifs interclasses deviennent indiscernables.
2. Nous vérifions que les définitions des relations interclasses présentées dans l'état de l'art utilisent, au moins, toutes les propriétés de l'ensemble \mathcal{E} .

D'abord, nous retirons, l'une après l'autre, une propriété de l'ensemble \mathcal{E} .

- soit un motif **Composition** entre deux classes A et B : $CO(A, B)$. Si nous retirons la propriété d'exclusivité, $EX(A, B)$, alors nous ne pouvons plus distinguer ce motif **Composition** d'un motif **Agrégation**.

Par exemple, la valeur de la propriété de multiplicité pour le motif **Composition**, $[1, +\infty]$, satisfait la valeur requise par la définition du motif **Agrégation**, $[0, +\infty]$: $[1, +\infty] \subset [0, +\infty]$; la valeur de la propriété de durée de vie pour le motif **Composition**, $+$, satisfait la valeur pour le motif **Agrégation**, $\parallel : + \in \parallel$;

- soit un motif **Agrégation** entre deux classes A et B : $AG(A, B)$. Si nous retirons la propriété de site d'invocation, $SI(A, B)$, alors nous ne pouvons plus distinguer ce motif **Agrégation** d'un motif **Association**.

Par exemple, la valeur de la propriété de multiplicité pour le motif **Agrégation**, $[1, +\infty]$, satisfait la valeur requise par la définition du motif **Association**, $[0, +\infty]$: $[1, +\infty] \subset [0, +\infty]$

- soit un motif **Composition** entre deux classes A et B : $CO(A, B)$. Si nous retirons la propriété de durée de vie, $DV(A, B)$, alors nous ne pouvons plus distinguer ce motif **Composition** d'un motif **Agrégation**.

Par exemple, la valeur de la propriété de multiplicité pour le motif **Composition**, $[1, +\infty]$, satisfait la valeur requise par la définition du motif **Agrégation**, $[0, +\infty]$, et la valeur de la propriété d'exclusivité pour le motif **Composition**, *vrai*, satisfait la valeur pour le motif **Agrégation**, *indifférente* : $vrai \in indifférente$;

- soit un motif **Agrégation** entre deux classes A et B : $AG(A, B)$. Si nous retirons la propriété de multiplicité, $MU(A, B)$, alors nous ne pouvons plus distinguer ce motif **Agrégation** d'un motif **Association**.

Par exemple, la valeur de la propriété de site d'invocation pour le motif **Agrégation**, $\{champ, champ\ tableau, champ\ collection\}$, satisfait la valeur requise par la définition du motif **Association**, *oui* : $\{champ, champ\ tableau, champ\ collection\} \subset oui$.

Ensuite, si nous reprenons les définitions des relations interclasses présentées dans l'état de l'art, toutes les définitions proposées utilisent les propriétés de l'ensemble \mathcal{E} .

Par exemple, les définitions des relations d'agrégation et de composition de [Henderson-Sellers et Barbier, 1999, table 4, page 356] utilisent différentes caractéristiques, parmi lesquelles : *C1. Propagation d'une ou plusieurs opérations* et *C5. Propagation des opérations de destruction* sont associées aux propriétés de durée de vie et de site d'invocation ; *C2. Propriétaire* est associée à la propriété d'exclusivité ; *P1. Tout-partie* est associée à la propriété de multiplicité.

3.4.3 Exemples au niveau implémentation

Nous présentons maintenant des exemples et des contre-exemples d'implantation des motifs interclasses. Nous ne présentons pas d'exemples de programmes réels, nous cherchons à donner une idée des implantations possibles des motifs interclasses. Nous utilisons ces mêmes exemples pour illustrer les algorithmes d'identification dans la section suivante page 116.

Exemple 3.10 – Implantation du motif Association.

Cet exemple présente deux classes A et B liées par un motif Association. Le motif Association existe par la présence de la méthode `operation1()`.

```

1  public class Association {
2      public static void main(String[] args) {
3          A a = new A(); // EX(A,B) = faux
4          B b = new B(); // EX(B,A) = faux
5          a.operation1(b); // DV(A,B) ∈ ||
6          b.operation2(); // DV(B,A) ∈ ||
7      }
8  }
9  public class A {
10     // MU(A,B) = [0, +∞]
11     // MU(B,A) = [0, +∞]
12     public void operation1(B b) {
13         b.operation2(); // SI(A,B) = {paramètre} ⊂ oui
14     }
15 }
16 public class B {
17     public void operation2() { // SI(B,A) = ∅ ⊂ indifférent
18     }
19 }
```

La propriété de durée de vie $DV(A,B) \in ||$: aucune référence incluse ne lie les propriétés de durée de vie des instances des classes A et B.

Les propriétés d'exclusivité sont *faux*, lignes 3 et 4 : le nommage explicite des instances des classes A et B permet à d'autres instances de les référencer et de les utiliser.

Les propriétés de multiplicité $MU(A,B)$ et $MU(B,A)$ sont égales à $[0, +\infty]$: rien n'interdit à une instance de la classe A de référencer plusieurs instances de la classe B ; réciproquement, rien n'interdit à plusieurs instances de la classe A de référencer la même instance de la classe B.

La propriété $SI(A,B)$ est *oui*, ligne 13 : les instances de la classe A envoient des messages aux instances de la classe B. La propriété $SI(B,A)$ est \emptyset car les instances de la classe B n'envoient pas de message aux instances de la classe A. \square

Exemple 3.11 – Autre implantation du motif **Association**.

Voici un autre exemple d'implantation du motif **Association** entre deux classes **A** et **B**.

```

1  public class Association {
2      public static void main(String[] args) {
3          A a = new A(); //  $EX(A, B) = faux$ 
4          B b = a.operation1(); //  $EX(B, A) = faux$ 
5          b.operation2(); //  $DV(A, B) = DV(B, A) \in ||$ 
6      }
7  }
8  public class A { //  $MU(A, B) = MU(B, A) = [1, 1] \subset [0, +\infty]$ 
9      public B operation1() {
10         B b = new B();
11         b.operation2(); //  $SI(A, B) = \{variable\ locale\} \subset oui$ 
12         return b;
13     }
14 }
15 public class B {
16     public void operation2() { //  $SI(B, A) = \emptyset \subset indifférent$ 
17     }
18 }
```

La propriété $DV(A, B) \in ||$: aucune référence incluse ne lie les instances des classes **A** et **B**. La création d'une instance de la classe **B** dans la méthode `operation1()` de la classe **A**, ligne 12, ne lie pas leurs durées de vie. Quand la méthode `operation1()` retourne, soit l'instance de **B** reste accessible au travers d'une autre référence, comme dans cet exemple, ligne 4, soit aucune référence sur cette instance existe et elle est prête à être ramassée par le ramasse-miettes, indépendamment de l'instance de la classe **A**.

Les propriétés d'exclusivité $EX(A, B)$ et $EX(B, A)$ sont *faux*, lignes 3 et 4. Le nommage explicite des instances des classes **A** et **B** et le type de retour de la méthode `operation1()`, ligne 11, permettent à d'autres instances de les référencer.

Les propriétés de multiplicité $MU(A, B)$ et $MU(B, A)$ sont $[0, +\infty]$: rien n'interdit à une instance de la classe **A** de référencer plusieurs instances de la classe **B**. Réciproquement, rien n'interdit à plusieurs instances de la classe **A** de référencer la même instance de la classe **B**.

La propriété $SI(A, B)$ est *oui*, la ligne 13 : une instance de la classe **A** peut envoyer un message à une instance de la classe **B**. La propriété $SI(B, A)$ est \emptyset car les instances de la classe **B** n'envoient pas de message aux instances de la classe **A**. □

Exemple 3.12 – Implantation du motif Agrégation.

Cet exemple présente deux classes **A** et **B** liées par un motif Agrégation. La classe **A** joue le rôle de tout et la classe **B** celui de partie. Le motif Agrégation existe grâce au champ **B b** et à la méthode **operation1()** de la classe **A**.

```

1  public class Aggregation {
2      public static void main(String[] args) {
3          B b = new B(); // EX(A,B) = faux
4          A a = new A(b); // EX(B,A) = faux
5          a.operation1(); // DV(A,B) ∈ ||
6          b.operation2(); // DV(B,A) ∈ ||
7      }
8  }
9  public class A {
10     private B b; // MU(A,B) = [1, 1]
11     public A(B b) {
12         this.b = b; // MU(B,A) = [1, 1] ∈ [0, +∞]
13     }
14     public void operation1() {
15         this.b.operation2(); // SI(A,B) = {champ}
16     }
17 }
18 public class B {
19     public void operation2() { // SI(B,A) = ∅
20     }
21 }
```

La propriété de durée de vie $DV(A,B) \in ||$: les instances de la classe **B** peuvent survivre aux instances de la classe **A**. Réciproquement, le champ **B b** de la classe **A** peut être mis à **null** (l'instance référencée peut alors être ramassée par le ramasse-miettes) et l'instance de la classe **A** survivre à l'instance de la classe **B**.

Les propriétés d'exclusivité $EX(A,B)$ et $EX(B,A)$ sont *faux*, lignes 3 et 4. Le nommage explicite des instances des classes **A** et **B** permet à d'autres instances de les référencer.

La propriété de multiplicité $MU(A,B)$ est $[1, 1]$ et la propriété $MU(B,A)$ est $[0, +\infty]$: une instance de la classe **A** référence une unique instance de la classe **B** à un instant donné, une instance de la classe **B** peut être référencée par aucune, une ou plusieurs instances de la classe **A**.

La propriété $SI(A,B)$ est $\{champ\}$, ligne 15 : une instance de la classe **A** peut envoyer un message à une instance de la classe **B** au travers du champ **B b** déclaré dans la classe **A**. La propriété $SI(B,A)$ est \emptyset car les instances de la classe **B** n'envoient pas de message aux instances de la classe **A**. □

Exemple 3.13 – Implantation du motif *Composition*.

Cet exemple présente deux classes *A* et *B* liées par le motif *Composition*. La classe *A* joue le rôle de tout et la classe *B* joue le rôle de partie. Le motif *Composition* existe grâce au champ *B b*, aux méthodes *attach()* et *operation1()* de la classe *A* et au ramasse-miettes de la machine virtuelle *JAVA* : la machine virtuelle ramasse l'instance de la classe *B* référencée par l'instance de la classe *A* avant de ramasser l'instance de la classe *A*. La privauté du champ *B b* participe à l'existence du motif *Composition* : elle assure l'exclusivité de l'instance de la classe *B* dans l'instance de la classe *A* et, ainsi, la dépendance des durées de vie.

```

1 public class Composition {
2     public static void main(String[] args) {
3         A a = new A(); //  $EX(B, A) = faux$ 
4         a.attach(new B()); //  $(EX(A, B) = vrai)$ 
5         a.operation1(); //  $(DV(A, B) = +) \wedge (DV(B, A) = -)$ 
6     }
7 }
8 public class A {
9     private B b; //  $MU(A, B) = [1, 1]$ 
10    public void attach(B b) { //  $MU(B, A) = [1, 1] \subset [0, +\infty]$ 
11        this.b = b;
12    }
13    public void operation1() {
14        this.b.operation2(); //  $SI(A, B) = \{champ\}$ 
15    }
16 }
17 public class B {
18    public void operation2() { //  $SI(B, A) = \emptyset$ 
19    }
20 }
```

La propriété de durée de vie des classes *A* et *B* est telle que : $DV(A, B) = +$. Le ramasse-miettes de la machine virtuelle *JAVA* ramasse l'instance de la classe *B* avant de ramasser l'instance de la classe *A* qui la référence. La propriété d'exclusivité $EX(A, B)$ est *vrai* car aucun autre objet que l'instance de la classe *A* créée ligne 3 ne peut référencer l'instance de la classe *B* créée ligne 4. La propriété $EX(B, A)$ est *faux*, ligne 3 : le nommage explicite de l'instance de la classe *A* permet à d'autres instances de la référencer.

La propriété de multiplicité $MU(A, B)$ est $[0, 1]$ et $MU(B, A)$ est $[1, 1]$: une instance de la classe *A* ne référence qu'au plus une instance de la classe *B* à un instant donné. Une instance de la classe *B* ne peut être référencée que par une instance de la classe *A*.

La propriété $SI(A, B)$ est $\{champ\}$, ligne 14 : les instances de la classe *A* envoient des messages aux instances de la classe *B*. La propriété $SI(B, A)$ est \emptyset car la classe *B* ne référence pas la classe *A*. \square

Exemple 3.14 – Autre implantation du motif *Composition*.

Cet exemple montre deux classes **A** et **B** liées par un motif *Composition*. L'instance de la classe **B** appartient complètement à l'instance de la classe **A** : l'instance de la classe **A** crée et possède l'unique référence sur l'instance de la classe **B**.

```

1  public class Composition {
2      public static void main(String[] args) {
3          A a1 = new A(); //  $EX(B, A) = faux$ 
4          a1.operation1(); //  $(DV(A, B) = +) \wedge (DV(B, A) = -)$ 
5      }
6  }
7  public class A {
8      private B b; //  $MU(A, B) = [1, 1]$ 
9      public A() { //  $MU(B, A) = [1, 1] \subset [0, +\infty]$ 
10         this.b = new B(); //  $EX(A, B) = vrai$ 
11     }
12     public void operation1() {
13         this.b.operation2(); //  $SI(A, B) = \{champ\}$ 
14     }
15 }
16 public class B {
17     public operation2() { //  $SI(B, A) = \emptyset$ 
18     }
19 }
```

La propriété de durée de vie des classes **A** et **B** est $DV(A, B) = +$. Le ramasse-miettes ramasse d'abord l'instance de la classe **B** référencée par une instance de la classe **A** avant de ramasser cette dernière.

La propriété d'exclusivité $EX(A, B)$ est *vrai* car aucun autre objet que l'instance de la classe **A** créée ligne 3 ne peut référencer l'instance de la classe **B** créée ligne 10. La propriété d'exclusivité $EX(B, A)$ est *faux*, ligne 3 : le nommage explicite de l'instance de la classe **A** permet à d'autres instances de la référencer.

La propriété de multiplicité $MU(A, B)$ est $[1, 1]$ et $MU(B, A)$ est $[1, 1]$: une instance de la classe **A** référence une unique instance de la classe **B**. Une instance de la classe **B** référencée par une instance de la classe **A** ne peut être référencée par aucun autre objet.

La propriété $SI(A, B)$ est $\{champ\}$, ligne 13 : une instance de la classe **A** peut envoyer un message à une instance de la classe **B** par l'intermédiaire du champ **B b**, déclaré par la classe **A**, ligne 8. La propriété $SI(B, A)$ est \emptyset car les instances de la classe **B** ne peuvent envoyer de message aux instances de la classe **A**. \square

Exemple 3.15 – Implantation de deux motifs **Composition** croisés.

Les deux classes **A** et **B** sont liées par un motif **Composition** au travers de deux implantations croisées. Le motif **Composition** existe par l'existence du champ **B b**, des méthodes **attach()** et **operation1()** et du ramasse-miettes.

```

1  public class Composition {
2      public static void main(String[] args) {
3          A a1 = new A(); // EX(A,B) = faux
4          A a2 = new A();
5          a1.attach(new B()); // EX(B,A) = vrai
6          a1.attach(null);
7          a2.attach(new B()); // EX(B,A) = vrai
8          a2.operation1();
9          // (DV(A,B) = +) ∧ (DV(B,A) = -)
10     }
11 }
12 public class A {
13     private B b; // MU(A,B) = [1, 1]
14     public void attach(B b) { // MU(B,A) = [1, 1] ⊂ [0, +∞]
15         this.b = b;
16     }
17     public void operation1() {
18         this.b.operation2(); // SI(A,B) = {champ}
19     }
20 }
21 public class B {
22     public operation2() { // SI(B,A) = ∅
23     }
24 }
```

La propriété de durée de vie des classes **A** et **B** est telle que : $DV(A,B) = +$. Le ramasse-miettes ramasse les instances de la classe **B** avant de ramasser les instances de la classe **A** qui la référence.

La propriété d'exclusivité $EX(A,B)$ est *vrai* car aucun objet ne peut référencer les instances de la classe **B**, créées lignes 5 et 7, sauf les instances **a1** et **a2** de la classe **A**, lignes 3 et 4. La propriété d'exclusivité $EX(B,A)$ est *faux*, lignes 3 et 4 : le nommage explicite des instances de la classe **A** permet à d'autres instances de les référencer.

La propriété de multiplicité $MU(A,B)$ est $[1, 1]$ et $MU(B,A)$ est $[1, 1]$: une instance de la classe **A** référence une unique instance de la classe **B**, une instance de la classe **B** est référencée par une unique instance de la classe **A**.

La propriété $SI(A,B)$ est $\{champ\}$, la ligne 18 : les instances de la classe **A** envoient des messages aux instances de la classe **B** par l'intermédiaire du champ **B b**, ligne 13. La propriété $SI(B,A)$ est \emptyset car la classe **B** ne référence pas la classe **A**. □

Exemple 3.16 – Contre-exemple au motif **Composition**.

Voici un contre-exemple de l'implantation du motif **Composition**. Les classes **A** et **B** ne satisfont pas les propriétés du motif **Composition** car l'instance de la classe **B**, créée ligne 4, peut être référencée par un autre objet que l'instance de la classe **A**, créée ligne 3. Ainsi, les propriétés d'exclusivité et de durée de vie n'ont pas les valeurs attendues : $\neg(EX(A, B) = vrai) \vee \neg(DV(A, B) = +)$.

```

1  public class Composition {
2      public static void main(String[] args) {
3          A a = new A(); //  $EX(B, A) = faux$ 
4          B b = new B(); //  $(EX(A, B) = faux) \Leftrightarrow \neg(EX(A, B) = vrai)$ 
5          a.attach(b);
6          a.operation1();
7          //  $(DV(A, B) = -) \Leftrightarrow \neg(DV(A, B) = +)$ 
8          //  $(DV(B, A) = +) \Leftrightarrow \neg(DV(B, A) = -)$ 
9      }
10 }
11 public class A {
12     private B b; //  $MU(A, B) = [1, 1]$ 
13     public attach(B b) { //  $MU(B, A) = [1, 1] \subset [0, +\infty]$ 
14         this.b = b;
15     }
16     public void operation1() {
17         this.b.operation2(); //  $SI(A, B) = \{champ\}$ 
18     }
19 }
20 public class B {
21     public operation2() { //  $SI(B, A) = \emptyset$ 
22     }
23 }
```

□

Exemple 3.17 – Contre-exemple à deux motifs **Composition** croisés.

Voici un contre-exemple de l'implantation de deux motifs **Composition** croisés. Les classes **A** et **B** ne satisfont pas les propriétés du motif **Composition** car l'instance de la classe **B**, créée ligne 7, référencée par l'instance **a2** de la classe **A**, peut être référencée par d'autres objets au travers de la référence **B b**, ligne 9. Ainsi, les propriétés d'exclusivité et de durée de vie n'ont pas les valeurs attendues : $\neg(EX(A, B) = vrai) \vee \neg(DV(A, B) = +)$.

```

1  public class Composition {
2      public static void main(String[] args) {
3          A a1 = new A(); // EX(B, A) = faux
4          A a2 = new A();
5          a1.attach(new B()); // EX(A, B) = vrai
6          a1.attach(null);
7          a2.attach(new B());
8          a2.operation1();
9          B b = a2.getB(); // EX(A, B) = faux
10         a2 = null;
11         a1 = null;
12         b.operation2();
13         // (DV(A, B) = -)  $\Leftrightarrow$   $\neg(DV(A, B) = +)$ 
14         // (DV(B, A) = +)  $\Leftrightarrow$   $\neg(DV(B, A) = -)$ 
15     }
16 }
17 public class A {
18     private B b; // MU(A, B) = [1, 1]
19     public attach(B b) { // MU(B, A) = [1, 1]  $\subset$   $[0, +\infty]$ 
20         this.b = b;
21     }
22     public B getB() {
23         return this.b; // EX(A, B) = faux
24     }
25     public void operation1() {
26         this.b.operation2(); // SI(A, B) = {champ}
27     }
28 }
29 public class B {
30     public operation2() { // SI(B, A) =  $\emptyset$ 
31     }
32 }

```

□

Exemple 3.18 – Autre contre-exemple à deux motifs **Composition** croisés.

Voici un autre contre-exemple de l'implantation de deux motifs **Composition** croisés. Les relations entre les classes **A** et **B** ne satisfont les propriétés du motif **Composition** car la propriété d'exclusivité $EX(A, B)$ n'est pas égale à *vrai* même si $DV(A, B) = +$.

```

1  public class Composition {
2      public static void main(String[] args) {
3          A a1 = new A(); //  $EX(B, A) = faux$ 
4          A a2 = new A();
5          a1.attach(new B());
6          a2.attach(a1.getB()); //  $EX(A, B) = faux$ 
7          a1.operation1();
8          a2.operation1();
9          //  $DV(A, B) = +$ 
10         //  $DV(B, A) = -$ 
11     }
12 }
13 public class A {
14     private B b; //  $MU(A, B) = [1, 1]$ 
15     public attach(B b) { //  $MU(B, A) = [1, 1] \subset [0, +\infty]$ 
16         this.b = b;
17     }
18     public B getB() {
19         return this.b;
20     }
21     public void operation1() {
22         this.b.operation2(); //  $SI(A, B) = \{champ\}$ 
23     }
24 }
25 public class B {
26     public operation2() { //  $SI(B, A) = \emptyset$ 
27     }
28 }
```

□

Conclusion. Nous avons défini les motifs interclasses au niveau **implémentation** comme des conjonctions des valeurs de leurs quatre propriétés minimales et nous en avons donné des exemples et des contre-exemples sous la forme de modèle statique. Ainsi, nous avons fait le lien entre les définitions des relations interclasses au niveau **idiomatique** et au niveau **implémentation**. Nous proposons maintenant des algorithmes pour identifier les motifs interclasses au niveau **implémentation**.

3.5 Algorithmes d'identification des motifs

LES DÉFINITIONS des motifs interclasses se décomposent en quatre propriétés minimales : durée de vie, exclusivité, multiplicité et site d'invocation. L'identification des motifs interclasses nécessite le calcul des valeurs de ces quatre propriétés et la comparaison des valeurs obtenues avec les valeurs définies pour chaque motif interclasse.

3.5.1 Principes

L'identification du motif **Association** nécessite la valeur de la propriété $SI(A, B)$ car les valeurs des autres propriétés sont indifférentes. L'identification du motif **Agrégation** nécessite les valeurs des propriétés $MU(A, B)$ et $SI(A, B)$. L'identification du motif **Composition** nécessite les valeurs des propriétés $MU(A, B)$ et $SI(A, B)$, comme pour le motif **Agrégation**, et des valeurs des propriétés $EX(A, B)$ et $DV(A, B)$.

Nous obtenons les valeurs des propriétés de multiplicité et de site d'invocation par des analyses statiques sur les modèles statiques des programmes. Nous calculons les valeurs des propriétés de durée de vie et d'exclusivité par des analyses dynamiques sur les modèles dynamiques des programmes.

Ainsi, les analyses statiques et dynamiques que nous proposons sont propres au langage de programmation JAVA. Les analyses statiques sont basées sur les idiomes de programmation de JAVA, les analyses dynamiques sur la machine virtuelle JAVA.

Les analyses dépendent de JAVA car nous cherchons à identifier les constituants supportant les quatre propriétés dans les modèles d'un programme au niveau **implémentation**, modèles dépendant du langage de programmation avec lequel le programme est écrit, pour les abstraire dans un modèle du programme au niveau **idiomatique**, modèle dans lequel les idiomes utilisés sont explicites.

Cependant, la dépendance des algorithmes d'identification sur le langage de programmation JAVA ne remet pas en cause l'indépendance de nos définitions des motifs interclasses au niveau **implémentation** et **idiomatique**.

D'une part, cette dépendance utilise quatre propriétés des motifs interclasses qui sont indépendantes des langages de programmation et minimales ; d'autre part, elle souligne l'intérêt de nos travaux pour construire le modèle d'un programme au niveau **idiomatique** et donc s'abstraire du langage de programmation utilisé au niveau **implémentation**.

3.5.2 Analyses statiques

L'algorithme 3.1 calcule les valeurs de la propriété $MU(A, B)$ pour deux classes **A** et **B** données. Il parcourt l'ensemble des champs de type **B** de la classe **A** obtenu par analyses syntaxiques et détermine leur multiplicité.

Une difficulté pour le calcul de la multiplicité survient lorsque le champ est une collection non typée (telles les collections du langage de programmation JAVA : **Map**, **List** et **Set**).

Dans ce cas, nous faisons l'hypothèse raisonnable [Jackson et Waingold, 1999] que la collection est homogène : tous les éléments ont une racine commune dans l'arbre d'héritage autre que `java.lang.Object`. Alors, nous pouvons déterminer le type de la collection avec les idiomes de programmation propres au langage de programmation JAVA pour les collections.

Par exemple, la présence d'une paire d'accesseurs sur une collection avec les signatures `void add(<type>)`–`void remove(<...>)` nous permet de déduire le type de la collection à partir du type du paramètre de la méthode `void add()`.

Algorithme 3.1 – Calcul de la valeur de la propriété $MU(A, B)$.

```

1   $MU(A, B) \subset \{(\text{champ}, B, \text{multiplicité})\},$ 
2      multiplicité  $\subset \mathbb{N} \cup \{+\infty\}$ 
3  Début
4      résultats  $\leftarrow \{\}$ 
5      champs  $\leftarrow \text{ensemble-champs}(A)$ 
6      TantQue  $\exists$  champs  $\in$  champs alors
7          Si champ instance de type(B) alors
8              Choisir genre(champ)
9              Cas Tableau
10                 résultats  $\leftarrow$  résultats  $\cup \{(\text{champ}, B, [0, +\infty])\}$ 
11             Cas Collection
12                 // Voir l'algorithme 3.2 page 118.
13                 résultats  $\leftarrow$  résultats  $\cup$  type-collection(A, champ, B)
14             Défaut
15                 résultats  $\leftarrow$  résultats  $\cup \{(\text{champ}, B, [1, 1])\}$ 
16             FinChoisir
17         FinSi
18     FinTantQue
19     Retourne résultats
20 Fin
```

□

L'algorithme 3.2 calcule le type d'une collection homogène avec ses accesseurs. D'autres techniques peuvent être implantées pour inférer le type d'une collection [Jackson et Waingold, 1999 ; Tonella et Potrich, 2001].

Algorithme 3.2 – Calcul basé sur les accesseurs de la valeur de la propriété $MU(A, B)$ pour une collection homogène.

```

1 type-collection(A, champ-collection, B)
2    $\subset \{\emptyset, (\text{champ}, B, \text{multiplicité})\}$ ,  $\text{multiplicité} \subset \mathbb{N} \cup \{+\infty\}$ 
3 Début
4   résultats  $\leftarrow \{\}$ 
5   méthodes-ajout  $\leftarrow \{\}$ 
6   méthodes-retrait  $\leftarrow \{\}$ 
7   méthodes  $\leftarrow \text{ensemble-méthodes}(A)$ 
8   TantQue  $\exists$  méthode  $\in$  méthodes alors
9     Si méthode-ajout(méthode) alors
10       méthodes-ajout  $\leftarrow$  méthodes-ajout  $\cup \{ \text{méthode} \}$ 
11     FinSi
12     Si méthode-retrait(méthode) alors
13       méthodes-retrait  $\leftarrow$  méthodes-retrait  $\cup \{ \text{méthode} \}$ 
14     FinSi
15   FinTantQue
16
17   TantQue  $\exists$  méthode-ajout  $\in$  méthodes-ajout alors
18     méthode-retrait-correspondante
19        $\leftarrow$  méthode-retrait-correspondante(méthode-ajout)
20     Si  $\exists$  méthode-retrait-correspondante  $\in$  méthodes-retrait
21     et accesseur(méthode, champ-collection)
22     et accesseur(méthode-retrait-correspondante, champ-collection)
23     et paramètre(méthode) instance de type(B)
24     alors
25       résultats  $\leftarrow$  résultats  $\cup (\text{champ}, B, [0, +\infty])$ 
26     FinSi
27   FinTantQue
28   Retourne résultats
29 Fin
```

□

L'algorithme 3.3 calcule les valeurs de la propriété $SI(A, B)$ pour deux classes **A** et **B** données. Il parcourt l'ensemble des méthodes de la classe **A** obtenu par des analyses syntaxiques, et il analyse le corps de chacune des méthodes à la recherche d'envois de messages à un receveur de type **B**.

Algorithme 3.3 – Calcul de la valeur de la propriété $SI(A, B)$.

```

1   $SI(A, B) \subset \emptyset \cup \{champ, champ\ tableau, champ\ collection,$ 
2       $param\grave{e}tre, variable\ locale\}$ 
3  Début
4      méthodes  $\leftarrow$  ensemble-méthodes(A)
5      résultat  $\leftarrow \emptyset$ 
6      TantQue  $\exists$  méthode  $\in$  méthodes alors
7          instructions  $\leftarrow$  ensemble-instructions(méthode)
8          TantQue  $\exists$  instruction  $\in$  instructions alors
9              Si type(instruction) = Envoi-Messsage alors
10                 receveur  $\leftarrow$  receveur(instruction)
11                 Si receveur instance de type(B) alors
12                     Choisir genre(receveur)
13                     Cas Champ
14                         résultat  $\leftarrow$  résultat  $\cup \{champ\}$ 
15                     Cas Champ-Tableau
16                         résultat  $\leftarrow$  résultat  $\cup \{champ\ tableau\}$ 
17                     Cas Champ-Collection
18                         résultat  $\leftarrow$  résultat  $\cup \{champ\ collection\}$ 
19                     Cas Paramètre
20                         résultat  $\leftarrow$  résultat  $\cup \{param\grave{e}tre\}$ 
21                     Cas Variable-Locale
22                         résultat  $\leftarrow$  résultat  $\cup \{variable\ locale\}$ 
23                     FinChoisir
24                 FinSi
25             FinSi
26         FinTantQue
27     FinTantQue
28     Retourne résultat
29 Fin
```

□

3.5.3 Analyses dynamiques

Nous identifions les valeurs des propriétés $EX(A, B)$ et $DV(A, B)$ pour le motif **Composition** avec des analyses dynamiques réalisées par des algorithmes d'analyse de traces d'exécution sur les modèles dynamiques des programmes au niveau implémentation.

L'algorithme 3.4 calcule la valeur de la propriété $EX(A, B)$. Il analyse la séquence d'événements d'exécution obtenue de la trace d'exécution. Pour chaque événement d'affectation d'une instance *affectée* à une instance *receveur* produit pendant l'exécution du programme, il vérifie qu'une instance de la classe **B** affectée à une instance de la classe **A**, receveur, n'est jamais affectée à une instance d'une autre classe, à un autre receveur. Il poursuit l'analyse tant que la séquence contient des événements. Nous supposons qu'il existe une méthode **obtenir-événements()** qui fournit la séquence d'événements d'exécution du programme analysé.

D'abord, nous initialisons une liste vide **couples-affectations**, ligne 3. Cette liste reçoit les couples (**receveur**(événement), **affecté**(événement)) correspondants à l'affectation d'une instance de la classe **B** à une instance de la classe **A**, lignes 6–7 et 11–12. Tant qu'il existe des événements d'exécution et que ceux-ci sont du type *Affectation* et concernent des instances des classes **A** et **B**, lignes 4–7, nous vérifions que l'instance affectée n'a pas déjà été affectée à une autre instance, ligne 8 : si c'est le cas, la propriété d'exclusivité est *faux*, sinon nous mémorisons ce couple dans la liste **couples-affectations**, lignes 11–12, et nous poursuivons l'analyse.

Algorithme 3.4 – Calcul de la valeur de la propriété $EX(A, B)$.

```

1   $EX(A, B) \in \{vrai, faux\}$ 
2  Début
3    couples-affectations  $\leftarrow \{\}$ 
4    événements-exécution  $\leftarrow$  obtenir-événements()
5    TantQue  $\exists$  événement  $\in$  événements-exécution alors
6      Si type(événement) = Affectation
7      et affecté(événement) instance de B alors
8        Si  $\exists (c, c_2) \in$  couples-affectations  $| c_2 =$  affecté(événement) alors
9          Retourne faux
10       Sinon
11         couples-affectations  $\leftarrow$  couples-affectations
12          $\cup \{(receveur(événement), affecté(événement))\}$ 
13       FinSi
14     FinSi
15   FinTantQue
16   Retourne vrai
17 Fin
```

□

L'algorithme 3.5 calcule la valeur de la propriété $DV(A, B)$. Cet algorithme est similaire à l'algorithme 3.4 page 120 de calcul de la valeur de la propriété $EX(A, B)$. D'abord, nous initialisons une liste vide **couples-affectations**, ligne 3.

Tant qu'il existe des événements d'exécution, si ces événements sont du type *Affectation* et concernent des instances des classes **A** et **B**, nous mémorisons ces affectations dans la liste **couples-affectations**, lignes 10–11; si ces événements sont du type *Ramassage* alors nous vérifions que l'instance affectée (jouant le rôle de partie) est bien ramassée avant l'instance d'affectation (jouant le rôle de tout), si ce n'est pas le cas, ligne 13, alors la propriété de durée de vie n'est pas vérifiée et l'algorithme retourne *faux*.

Conclusion. Nous avons proposé des algorithmes d'analyses statiques et dynamiques pour calculer les valeurs des propriétés des motifs interclasses. Les algorithmes d'analyses statiques utilisent des analyses syntaxiques pour identifier la multiplicité des champs et les site d'invocation des appels de méthodes. Les algorithmes d'analyses dynamiques utilisent les traces d'exécution d'un programme pour calculer les durées de vie et l'exclusivité des instances créées par le programme. Nous utilisons maintenant ces algorithmes pour identifier et garantir la traçabilité des motifs interclasses.

Algorithme 3.5 – Calcul de la valeur de la propriété $DV(A, B)$.

```

1   $DV(A, B) \in \{-, +\}$ 
2  Début
3    couples-affectations  $\leftarrow \{\}$ 
4    événements-exécution  $\leftarrow$  obtenir-événements()
5    TantQue  $\exists$  événement  $\in$  événements-exécution alors
6      Choisir type(événement)
7      Cas Affectation
8        Si receveur(événement) instance de type(A)
9        et affecté(événement) instance de type(B) alors
10         couples-affectations  $\leftarrow$  couples-affectations
11          $\cup \{ (\text{receveur}(\text{événement}), \text{affecté}(\text{événement})) \}$ 
12      FinSi
13      Cas Ramassage
14        Si  $\exists (c_1, c_2) \in$  couples-affectations  $| c_1 = \text{receveur}(\text{événement})$  alors
15          Retourne  $-$ 
16        FinSi
17      FinChoisir
18    FinTantQue
19    Retourne  $+$ 
20 Fin
```

□

3.6 Algorithmes de traçabilité des motifs

Nous décomposons l'identification et la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique** en deux phases⁸. D'abord, nous construisons un modèle du programme au niveau **idiomatique** en analysant son modèle statique au niveau **implémentation**.

Ce modèle représente l'architecture du programme au niveau **idiomatique** sans prendre en compte les résultats des analyses dynamiques pour des raisons de simplicité, de disponibilité et de performance.

Puis, nous raffinons le modèle du programme au niveau **idiomatique** en appliquant les algorithmes d'analyses dynamiques sur un sous-ensemble de ses modèles dynamiques au niveau **implémentation**.

3.6.1 Analyses statiques et modèle partiel du programme

Nous analysons le modèle du programme au niveau **implémentation** pour en construire un modèle au niveau **idiomatique** et nous utilisons les algorithmes d'analyses statiques pour identifier les motifs **Association** et **Agrégation**. Nous utilisons de simples analyses syntaxiques pour identifier les relations d'appel de méthodes, d'instanciation et d'héritage.

L'algorithme 3.6 page 124 construit le modèle du programme au niveau **idiomatique** en utilisant les constituants du métamodèle défini section 3.1 page 79. D'abord, l'algorithme crée un nouveau modèle du programme au niveau **idiomatique**, ligne 4. Puis, pour chaque classe du modèle du programme au niveau **implémentation**, l'algorithme crée l'entité correspondante au niveau **idiomatique**, lignes 5–8. Cette nouvelle entité est modifiée pour refléter la classe dont elle est le modèle, lignes 9–12, puis elle est ajoutée au modèle en construction, ligne 13.

Ensuite, pour chaque constituant du modèle au niveau **implémentation**, nous créons, si nécessaire, les relations d'association et d'agrégation qui lient les entités correspondantes deux à deux. Pour cela, nous utilisons les prédicats $AS(A, B)$ et $AG(A, B)$, lignes 18, 25, 29 et 36, définis dans la section 3.2 page 84. Typiquement, nous créons une nouvelle relation, ligne 19, que nous modifions pour refléter les classes qu'elle lient entre elles, lignes 20–22, avant de l'ajouter à l'entité nouvellement créée, ligne 23.

Nous créons aussi les relations d'appel de méthodes, d'instanciation et d'héritage avec de simples analyses syntaxiques du modèle statique du programme au niveau **implémentation**, lignes 40–42.

⁸Un découpage similaire est utilisé dans [Heuzeroth *et al.*, 2002] pour identifier des motifs d'interactions par des analyses statiques raffinées par des analyses dynamiques.

Nous ne vérifions pas la condition $\neg CO(A, B)$ du prédicat $AG(A, B)$ dans cette phase de la construction du modèle du programme au niveau **idiomatique**. La condition $\neg CO(A, B)$ est vérifiée après les analyses dynamiques, dans la deuxième phase de construction du modèle du programme.

L'algorithme retourne une instance du modèle du programme du niveau **idiomatique**, ligne 45. Ce modèle reflète partiellement l'architecture du programme car seuls les algorithmes d'analyses statiques ont été utilisés pour le construire. Il doit être raffiné pour prendre en compte les analyses dynamiques capables de distinguer les motifs **Agrégation** et **Composition**.

3.6.2 Analyses dynamiques et modèle du programme

L'utilisation des algorithmes d'analyses dynamiques pour raffiner le modèle du programme est décrite par l'algorithme 3.7 page 125. D'abord, nous instrumentons le modèle du programme au niveau **implémentation** pour que celui-ci génère effectivement les événements définis dans la section 3.5 page 116 : événements d'*Affectation* et de *Ramassage*, puis nous exécutons le programme ainsi modifié, lignes 4–5, pour obtenir un ensemble d'événements d'exécution utilisé pour calculer les valeurs des propriétés de durée de vie et d'exclusivité.

Ensuite, pour chaque classe du programme analysé, ligne 6, nous obtenons l'entité correspondante au niveau **idiomatique** et vérifions si cette entité est liée par un motif **Agrégation** avec une autre classe du programme, lignes 7–10 ; si c'est le cas, nous utilisons le prédicat $CO(A, B)$ pour calculer l'existence d'un motif **Composition** au lieu du motif **Agrégation**. Le prédicat $CO(A, B)$ utilise les algorithmes $EX(A, B)$ et $DV(A, B)$ qui utilisent la séquence d'événements d'exécution produite par l'exécution du programme pour calculer les valeurs des propriétés de durée de vie et d'exclusivité, ligne 13. Nous vérifions aussi que deux entités ne peuvent être mutuellement liées par des motifs **Agrégation** et **Composition**. Dans le cas où un motif **Composition** est identifié, nous créons le nouveau motif, nous le modifions pour qu'il reflète les entités qu'il lie entre elles et nous l'ajoutons à l'entité correspondante, lignes 14–17. Nous retirons de cette dernière le motif **Agrégation** qui vient d'être raffiné en un motif **Composition**, ligne 18.

Conclusion. Nous avons présenté des algorithmes pour construire le modèle d'un programme au niveau **idiomatique** avec les modèles statiques et dynamiques du programme au niveau **implémentation** et les algorithmes d'identification des motifs **Association**, **Agrégation** et **Composition**. Ces algorithmes garantissent la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique** car les constituants représentant les motifs au niveau **idiomatique** sont liés aux constituants du niveau **implémentation** qu'ils abstraient. Nous vérifions maintenant la cohérence des définitions des motifs interclasses, de leurs modèles et des algorithmes d'identification.

Algorithme 3.6 – Construction du modèle statique du programme au niveau idiomatique.

```

1  construire-modèle-idiomatique-statique(modèle-implémentation)
2    ∈ Méta-Modèle-Idiomatique
3  Début
4    modèle-idiomatique ← nouveau(Modèle-Idiomatique)
5    classes1 ← ensemble-classes(modèle-implémentation)
6    TantQue ∃ classe1 ∈ classes1 alors
7      // Création de la nouvelle entité :
8      nouvelle-entité ← nouvelle(Entité)
9      modifier-nom(nouvelle-entité, obtenir-nom(classe1))
10     // Modification des autres attributs de la nouvelle entité
11     // en fonction des attributs de la classe :
12     ...
13     ajouter-entité(modèle-idiomatique, nouvelle-entité)
14     // Ajout à la nouvelle entité (et aux autres entités
15     // du modèle concernées) des éléments nécessaires :
16     classes2 ← ensemble-classes(modèle-implémentation)
17     TantQue ∃ classe2 ∈ classes2 alors
18       Si AS(classe1, classe2) = vrai alors
19         association ← nouvelle(Association)
20         modifier-cardinalité(association, MU(classe1, classe2))
21         entité-cible ← obtenir-entité(modèle-idiomatique, obtenir-nom(classe2))
22         modifier-entité-cible(association, entité-cible)
23         ajouter-élément(nouvelle-entité, association)
24       FinSi
25       Si AS(classe2, classe1) = vrai alors
26         // Mêmes instructions que dans la condition précédente,
27         // avec les paramètres classe1 et classe2 inversés.
28       FinSi
29       Si AG(classe1, classe2) = vrai alors
30         agrégation ← nouvelle(Agrégation)
31         modifier-cardinalité(agrégation, MU(classe1, classe2))
32         entité-cible ← obtenir-entité(modèle-idiomatique, obtenir-nom(classe2))
33         modifier-entité-cible(agrégation, entité-cible)
34         ajouter-élément(nouvelle-entité, agrégation)
35       FinSi
36       Si AG(classe2, classe1) = vrai alors
37         // Mêmes instructions que dans la condition précédente,
38         // avec les paramètres classe1 et classe2 inversés.
39       FinSi
40       // Identification des relations d'envoi de messages,
41       // d'instanciation et d'héritage par analyses syntaxiques.
42       ...
43     FinTantQue
44   FinTantQue
45   Retourne modèle-idiomatique
46 Fin

```

□

Algorithme 3.7 – Construction du modèle complet du programme au niveau idiomatique.

```

1  construire-modèle-idiomatique-complet(modèle-implémentation, modèle-idiomatique)
2    ∈ Méta-Modèle-Idiomatique
3  Début
4    instrumenter(modèle-implémentation)
5    exécuter(modèle-implémentation)
6    classes ← ensemble-classes(modèle-implémentation)
7    TantQue ∃ classe1 ∈ classes alors
8      entité1 ← obtenir-entité(modèle-idiomatique, obtenir-nom(classe1))
9      éléments ← obtenir-éléments(entité1)
10     Si ∃ agrégation ∈ éléments | type(élément) instance de Agrégation alors
11       entité2 ← obtenir-entité-cible(agrégation)
12       classe2 ← obtenir-entité(modèle-implémentation, obtenir-nom(entité2))
13       Si  $CO(classe1, classe2) = vrai$  et  $\neg AG(classe2, classe1)$  alors
14         composition ← nouvelle(Composition)
15         modifier-cardinalité(composition,  $MU(classe1, classe2)$ )
16         modifier-entité-cible(composition, entité2)
17         ajouter-élément(entité1, composition)
18         retirer-élément(entité1, agrégation)
19       FinSi
20     FinTantQue
21   FinTantQue
22   Retourne modèle-idiomatique
23 Fin

```

□

3.7 Discussion des modèles, des définitions et des algorithmes

Nous vérifions la cohérence, la validité de nos modèles, de nos définitions des motifs interclasses et de nos algorithmes d'identification en trois phases. D'abord, nous discutons nos modèles par rapport à d'autres techniques de modélisation et à d'autres modèles.

Puis, nous montrons que la validation de nos définitions est difficile car nous n'avons pu trouver de programmes dans lesquels les développeurs aient explicitement distingué les motifs interclasses *Association*, *Agrégation* et *Composition*.

Ensuite, nous appliquons nos algorithmes sur trois programmes bien connus : JHOT-DRAW v5.1, JUNIT v3.7, et AWT v1.2.2 et comparons leurs résultats avec nos définitions par des analyses manuelles lorsque nécessaire. Nous réalisons ces validations avec une implantation des modèles et des algorithmes détaillée dans la partie III page 195.

3.7.1 Modèles

Les modèles pour décrire un programme aux niveaux *implémentation* et *idiomatique* sont à comparer avec d'autres techniques de description. Le métamodèle est à comparer avec d'autres métamodèles existants.

Modèles au niveau implémentation Les modèles statiques et dynamiques pour représenter un programme au niveau *idiomatique* nous sont imposés par notre choix du langage de programmation JAVA. Le code source est la représentation naturelle, la plus concrète, d'un programme.

Les traces d'exécutions ont été utilisées pour d'autres langages, C [Ducassé, 1999a] et PROLOG [Ducassé, 1999b], pour décrire des modèles dynamiques de programmes. Elles sont complémentaires du modèle statique et nécessaires à la compréhension des programmes [Soloway, 1986].

Autres modèles pour le niveau idiomatique Roel Wuyts [1998] décrit les modèles de programmes comme des ensembles de faits PROLOG sur lesquels les auteurs raisonnent alors avec des prédicats dédiés. Le code source des programmes est analysé lexicalement et syntaxiquement pour produire des bases de faits, comme présentée section 2.3 page 60.

Les motifs interclasses sont définis comme des prédicats dont les variables sont unifiées avec une base de faits pour identifier les classes liées par ces motifs. Les résultats de l'unification sont de nouveaux faits qui sont ajoutés à la base. Ils sont utilisés par d'autres prédicats pour identifier des motifs de conception.

Jens H. Jahnke *et al.* [1997] modélisent et analysent des programmes avec des réseaux génériques de raisonnement flou. Un programme est représenté par un graphe annoté obtenu automatiquement par des analyses syntaxiques.

Les motifs interclasses sont représentés par des diagrammes de collaboration traduits en réseaux de raisonnement flou avec lesquels il est possible d'identifier ces motifs et d'ajouter de nouveaux nœuds au graphe du programme les représentant.

Richard Lemesle [2000] utilise le formalisme des SNETS, basé sur les réseaux sémantiques, pour modéliser des programmes. L'auteur définit précisément avec les SNETS un univers sémantique représentant un sous-ensemble du métamodèle UML. L'utilisation des SNETS permet d'explicitier, de formaliser et d'étudier précisément la relation d'instanciation entre modèle, métamodèle et métamétamodèle.

Cependant, ces modèles et ce métamodèle pour UML ne sont pas toujours très adaptés à la modélisation des programmes aux niveaux **idiomatique** et **conception** ou à la modélisation des motifs de conception. De plus, aucun algorithme n'est proposé pour construire le modèle d'un programme au niveau **idiomatique** depuis son code source.

Jochen Seemann et Jürgen Wolf von Gudenberg [1998] proposent un processus de rétroconception de programmes JAVA par raffinement, comme présenté section 2.2 page 37.

Le code source du programme est analysé pour construire un graphe $S = (V, E, \phi, \eta)$. Le graphe V est transformé avec une grammaire de graphe pour abstraire des relations d'association et d'agrégation.

Le graphe obtenu contient alors des informations similaires aux informations représentées par le modèle d'un programme issu de notre métamodèle mais limitées pour les motifs interclasses par manque de distinction entre association, agrégation et composition. De plus, la traçabilité entre les niveaux **implémentation** et **idiomatique** n'est pas garantie.

Ces différentes techniques sont similaires à notre utilisation de la métamodélisation : elles définissent toutes des constituants dont les instances forment les modèles des programmes.

Par exemple, des faits représentant un programme respectent des conventions pour être ensuite utilisés comme base par des prédicats logiques, ces conventions forment un "métamodèle" pour ces modèles de programmes.

La différence principale entre ces techniques et nos métamodèles provient de leurs utilisations : les faits PROLOG sont intéressants pour raisonner par unification ; les réseaux génériques de raisonnement flou sont utiles pour prouver des formules sur les modèles des programmes ; les réseaux sémantiques sont simples et expressifs.

Ainsi, nous pensons qu'il n'y a pas de *meilleure* technique, seulement des techniques adaptées à l'utilisation qui en est faite. La métamodélisation est une technique adéquate pour identifier les motifs interclasses et pour garantir leur traçabilité entre les niveaux **implémentation** et **idiomatique**.

Autres métamodèles pour le niveau idiomatique La notation UML [Booch *et al.*, 1999] propose un métamodèle pour décrire des modèles de programmes dans chaque phase de leur cycle de développement avec neuf types de *diagrammes*^{*} représentant chacun un aspect d'un programme et de son développement.

Les travaux réalisés dans le cadre du projet FAMOOS⁹ ont abouti au développement du métamodèle FAMIX [Demeyer *et al.*, 1999b] utilisé pour assurer l'interopérabilité et l'aller-retour entre outils de conception et de rétroconception.

Nous n'avons pas utilisé l'un ou l'autre de ces métamodèles pour garder nos travaux les plus simples possibles. D'une part, l'utilisation d'un métamodèle plus général aurait nécessité de l'adapter à nos besoins spécifiques.

D'autre part, nous sommes convaincus que le pouvoir d'expression de nos métamodèles et des métamodèles de la littérature sont similaires et qu'il est possible de convertir les modèles des uns en des modèles des autres.

Conclusion Ainsi, les modèles présentés section 3.1 page 79 sont adéquats pour modéliser l'architecture d'un programme et pour garantir la traçabilité des motifs interclasses entre les niveaux implémentation et idiomatique.

3.7.2 Définitions

La validation de nos définitions et de nos résultats par rapport à des définitions et des outils indépendants est une tâche difficile. Nous aimerions appliquer nos algorithmes sur des programmes bien documentés et comparer les résultats obtenus avec les documentations de ces programmes et d'autres outils d'analyses.

Cependant, les développeurs utilisent principalement des langages de modélisation comme UML lors de la conception et ils ne peuvent donc pas distinguer clairement les motifs Association, Agrégation et Composition.

Ainsi, nous avons été incapables de trouver un programme disponible librement et gratuitement, comme JHOTDRAW ou JUNIT, dans lequel les développeurs aient explicitement distingué les trois motifs.

Par exemple, les auteurs de JHOTDRAW utilisent la notation de [Gamma *et al.*, 1994]. Cette notation offre la notion d'association et une relation d'agrégation générique mais ne fait pas mention de la relation de composition.

Nous ne pouvons donc pas valider nos définitions avec d'autres définitions ou d'autres outils. Nos définitions seront validées à terme par la pertinence des modèles des programmes au niveau idiomatique.

⁹FAMOOS est l'acronyme anglais de *Framework-based Approach for Mastering Object-Oriented*, projet européen ESPRIT 21 975.

3.7.3 Algorithmes

Nos algorithmes de calcul des valeurs de propriétés des motifs interclasses sont pour une part statiques et pour une autre part dynamiques.

Analyses statiques Nous avons testé les implantations des algorithmes d’analyses statiques sur trois programmes (dont une bibliothèque) :

- AWT v1.2.2 ;
- JHotDraw v5.1 ;
- JUnit v3.7.

Nous avons choisi ces programmes car ils sont connus et des informations architecturales les concernant sont disponibles : diagrammes de classes, études [Gamma et Eggenschwiler, 1998 ; Gamma et Beck, 1998 ; Seemann et von Gudenberg, 1998 ; Sun Microsystems, Inc., 2002 ; Kaiser, 2001 ; Gamma et Beck, 2002].

Le motif **Association** est le moins contraignant des trois motifs et son identification fournit un nombre important de résultats. Il y a $2784 + 1505 + 636 = 4925$ motifs **Association** identifiés dans les 583 classes des trois programmes.

Le motif **Agrégation** est le motif le plus intéressant par rapport à l’analyse statique. Le tableau 3.1 montre les résultats de l’identification du motif **Agrégation** pour les différents programmes et la bibliothèque. Ce tableau détaille les nombres : de *classes* par programme ; de motifs **Agrégation** *identifiés* ; de motifs **Agrégation** trouvés par *analyse manuelle* ; de motifs identifiés par *erreur* ; et, de motifs *manqués* par nos algorithmes.

Tableau 3.1 – Résultats de l’identification du motif **Agrégation**.

Programmes	Classes	Identifiés	Analyse manuelle	Erreurs	Manqués
AWT v1.2.2	367	17	20	1	3
JHotDraw v5.1	171	6	8	0	2
JUnit v3.7	45	1	4	0	3
Total	583	24	32	1	8
Précision : $0.75 \left(\frac{\text{Identifiés}}{\text{Analyse manuelle}} \right)$		Rappel : $0.96 \left(\frac{\text{Identifiés}}{\text{Identifiés} + \text{Erreurs}} \right)$			

□

Ces résultats n'incluent pas les motifs **Agrégation** dont la multiplicité est 1 car l'identification de ces motifs ne présente pas de difficulté et augmenterait artificiellement la précision de nos résultats. Les motifs **Agrégation** de multiplicité 1 sont identifiables en analysant les champs simples (ni tableaux, ni collections) des classes.

De plus, ces résultats impliquent uniquement les motifs **Agrégation** avec des collections homogènes, sans types primitifs, sans types d'emballage¹⁰ ni de chaînes de caractères¹¹ car l'identification du motif **Agrégation** pour ces types n'est pas intéressante. De telles collections sont habituellement implantées en utilisant la classe `java.util.Vector`.

Nous réalisons une analyse manuelle pour vérifier la précision de nos algorithmes. Les tableaux 3.2, 3.3 et 3.4 pages 131 et 132 résument les motifs **Agrégation** trouvés dans les trois programmes et soulignent les motifs identifiés par nos algorithmes. Pour plus de clarté, nous incluons les noms des paquetages uniquement pour lever une ambiguïté. Nous ne mentionnons pas les motifs faisant intervenir des auditeurs¹² car les mécanismes de notification sont une préoccupation orthogonale aux motifs.

Nous n'obtenons pas une précision de 100% car les mainteneurs des trois programmes n'ont pas respecté tous les idiomes de programmation propres au langage de programmation JAVA requis par nos algorithmes d'analyses ; par exemple, la classe `PolyLineFigure` offre une méthode créant directement des instances de la classe `Point`, `addPoint()`, et une méthode supprimant le point dont l'indice est passé en paramètre, `removePointAt()` ; ou la classe `Figure` qui utilise une convention de nommage particulière avec les méthodes `addToContainer()` et `removeFromContainer()`.

Nos algorithmes d'analyses statiques ne sont pas indépendants du langage de programmation utilisé pour décrire les modèles du programme au niveau implémentation, ils sont sensibles au respect des idiomes de programmation. Les algorithmes ne peuvent calculer les valeurs de la propriété de multiplicité pour les collections si les développeurs n'ont pas respecté de bonnes pratiques de programmation. Nous envisageons de modifier les algorithmes pour prendre en compte des formes approchées d'implantation.

¹⁰Les types d'emballage sont appelés *wrappers* en anglais [Office québécois de la langue française, 2003].

¹¹Les chaînes de caractères sont instances de la classe `java.lang.String`.

¹²Les auditeurs sont appelés *listeners* en anglais [Office québécois de la langue française, 2003].

Tableau 3.2 – Identification du motif Agrégation dans AWT.

Classe	En relation d'agrégation avec	Classe(s)
CardLayout	1	Component
Component	1	PopupMenu
Container	1	Component
Window	1	WeakReference
EventQueue	1	Queue
GridBagLayout	1	Component
MenuBar	1	Menu
Menu	1	MenuItem
StringSelection	1	DataFlavor
DragGestureRecognizer	1	InputEvent
TextJustifier	1	GlyphJustificationInfo
TextLine	1	TextLineComponent
TextMeasurer	1	TextLineComponent
Area	1	Curve
ColorConvertOp	2	ICC_Profile, ColorSpace
IndexColorModel	1	IndexColorModel
FilteredImageSource	1	ImageConsumer
MemoryImageSource	1	ImageConsumer
RenderableImageProducer	1	ImageConsumer
Book	1	Book\$BookPage
Total	21	

□

Tableau 3.3 – Identification du motif Agrégation dans JHOTDRAW.

Classe	En relation d'agrégation avec	Classe(s)
NodeFigure	1	Connector
CompositeFigure	1	Figure
StandardDrawingView	4	Painter (×2), Figure (×2)
CommandMenu	1	Command
CommandChoice	1	Command
Iconkit	1	Image
StorableInput	1	Storable
StorableOutput	1	Storable
StorageFormatManager	1	StorageFormat
CustomToolBar	2	Component (×2)
PertFigure	2	PertFigure (×2)
PolyLineFigure	1	Point
Total	17	

□

Tableau 3.4 – Identification du motif Agrégation dans JUNIT.

Classe	En relation d'agrégation avec	Classe(s)
TestResult	2	TestFailure (×2)
TestSuite	1	Test
junit.awtui.TestRunner	2	Test, Throwable
junit.swingui.TestRunner	1	TestRunView
TestTreeModel	3	Test (×3)
Total	9	

□

Analyses dynamiques Les tableaux 3.5, 3.6 et 3.7 page 134 présentent les résultats de l'identification du motif **Composition** pour JUNIT, en appliquant nos algorithmes d'analyses dynamiques sur le programme `junit.samples.money.MoneyTest` et son interface utilisateur textuelle, AWT et SWING. Nous n'avons pas réalisé d'analyses dynamiques sur JHOTDRAW car ce programme nécessite des interactions avec l'utilisateur et sur AWT car cette bibliothèque n'est pas un programme exécutable.

Lorsque nous exécutons la classe `junit.samples.money.MoneyTest` avec l'interface textuelle, les motifs **Agrégation** identifiés avec les algorithmes d'analyses statiques sont remplacés par des **Composition** : les propriétés de durée de vie et d'exclusivité sont vérifiées.

Lorsque nous exécutons la classe `junit.samples.money.MoneyTest` avec l'interface AWT, les analyses dynamiques révèlent aussi les motifs **Agrégation** comme étant des **Composition**. Cependant, ces résultats sont sujets à caution car ils correspondent à un sous-ensemble de tous les chemins d'exécution. Dans le cas d'une erreur dans un des tests, les motifs **Agrégation** restent les mêmes. En effet, en cas d'échec ou d'erreur, la propriété d'exclusivité n'est plus valide entre les classes `TestSuite` et `junit.awt.TestRunner` et la classe `Test`. La classe `TestResult` donne l'instance défectueuse de la classe `Test` à l'instance de la classe `junit.awt.TestRunner` qui la mémorise dans une collection.

Lorsque nous exécutons la classe `junit.samples.money.MoneyTest` avec l'interface SWING, l'analyse dynamique montre des résultats similaires à ceux obtenus avec l'interface AWT :

- un motif **Composition** existe entre les classes `TestResult` et `TestFailure` et entre les classes `junit.swingui.TestRunner` et `TestRunView`;
- un motif **Agrégation** existe entre les classes `TestSuite`, `TestTreeModel` et `Test`, car les instances de la classe `Test` sont partagées entre les deux premières classes en cas d'échec ou d'erreur.

Ces résultats montrent l'intérêt des analyses dynamiques et aussi leurs limitations :

- la détection de motifs contraignants (par rapport à leurs propriétés) découvre des dépendances subtiles entre les classes ;
- les résultats des calculs des valeurs des propriétés dépendent des chemins d'exécution suivis et les motifs interclasses peuvent changer d'une manière inattendue par les mainteneurs.

Des analyses de couvertures de code [Tikir et Hollingsworth, 2002] doivent être mises en place pour augmenter la précision de l'identification des motifs.

Conclusion. Nous avons validé nos modèles des programmes au niveau implémentation et idiomatique par rapport à d'autres techniques de modélisation. Puis, nous avons validé nos définitions et nos algorithmes sur trois programmes connus. Ces validations ont montré la pertinence de nos définitions et de la mise en œuvre de nos algorithmes pour l'identification des motifs interclasses. Nous illustrons maintenant l'application de nos travaux à JHOTDRAW.

Tableau 3.5 – Identification du motif **Composition** dans **JUNIT** avec l'interface textuelle.

Classe	En relation de composition avec	Classe(s)
TestResult	2	TestFailure (×2)
TestSuite	1	Test
<code>junit.awtui.TestRunner</code>	/	Test , Throwable
<code>junit.swingui.TestRunner</code>	/	TestRunView
TestTreeModel	/	Test (×3)
Total	3	

□

Tableau 3.6 – Identification du motif **Composition** dans **JUNIT** avec l'interface **AWT**.

Classe	En relation de composition avec	Classe(s)
TestResult	2	TestFailure (×2)
TestSuite	0	Test
<code>junit.awtui.TestRunner</code>	1 (Throwable)	Test , Throwable
<code>junit.swingui.TestRunner</code>	/	TestRunView
TestTreeModel	/	Test (×3)
Total	3	

□

Tableau 3.7 – Identification du motif **Composition** dans JUNIT avec l'interface SWING.

Classe	En relation de composition avec	Classe(s)
<code>TestResult</code>	2	<code>TestFailure</code> (×2)
<code>TestSuite</code>	0	<code>Test</code>
<code>junit.awtui.TestRunner</code>	/	<code>Test</code> , <code>Throwable</code>
<code>junit.swingui.TestRunner</code>	1	<code>TestRunView</code>
<code>TestTreeModel</code>	0	<code>Test</code> (×3)
Total	3	

□

3.8 Application à JHOTDRAW

Nous mettons en œuvre les modèles, les définitions et les algorithmes proposés pour l'identification et la traçabilité des motifs interclasses. Le détail des implantations est présenté dans la partie III page 195.

La figure 3.4 page 137 montre un extrait du modèle statique au niveau implémentation de JHOTDRAW, son code source en JAVA, dans l'EDI ECLIPSE. Nous appliquons nos algorithmes à un sous-ensemble de ce modèle pour construire le modèle du programme au niveau idiomatique représenté sur la figure 3.5 page 138.

Ce modèle décrit JHOTDRAW, ses classes et ses interfaces, et leurs relations au niveau idiomatique. Il présente essentiellement les mêmes informations que le modèle du programme décrit par le diagramme de classes fourni avec sa documentation, représenté sur la figure 3.6 page 139, comme le montre la figure 3.7 page 140.

Certaines relations sont différentes car les auteurs de JHOTDRAW ont créé un diagramme de classes avec uniquement les principales classes et interfaces du programme, sur lesquelles ils ont reporté certaines des relations existantes entre leurs sous-classes, le modèle obtenu automatiquement montre les relations *réelles* entre les entités du programme.

Par exemple, la relation d'instanciation entre l'interface **Figure** et l'interface **Handle** dans le diagramme de classes existe uniquement entre les classes **StandardDrawingView** (qui implante **Figure**) et **NullHandle** (qui implante **Handle**) dans le modèle obtenu automatiquement car une interface ne peut contenir d'instructions d'instanciation.

Nous avons validé manuellement toutes les relations entre entités obtenues automatiquement, par exemple le motif **Association** entre les interfaces **DrawingView** et **DrawingEditor**, le motif **Agrégation** entre la classe **CompositeFigure** et l'interface **Figure** et nous trouvons que les informations fournies automatiquement sont correctes par rapport à nos définitions et qu'elles précisent le diagramme de classes de JHOTDRAW.

De plus, la traçabilité est garantie entre les modèles de JHOTDRAW aux niveaux implémentation et idiomatique : par exemple, les constituants **Figure** des modèles aux niveaux implémentation et idiomatique se mettent en valeur simultanément lorsque les mainteneurs sélectionnent l'un de ceux-ci avec l'interface graphique.

Conclusion. Nous avons illustré nos modèles, nos définitions et nos algorithmes pour garantir la traçabilité des motifs interclasses entre les niveaux implémentation et idiomatique pour le programme JHOTDRAW. Nous concluons maintenant sur la traçabilité des motifs interclasses avant d'étudier la traçabilité des motifs de conception entre les niveaux idiomatique et conception.

FIG. 3.4 – Modèle de JHOTDRAW au niveau implémentation.

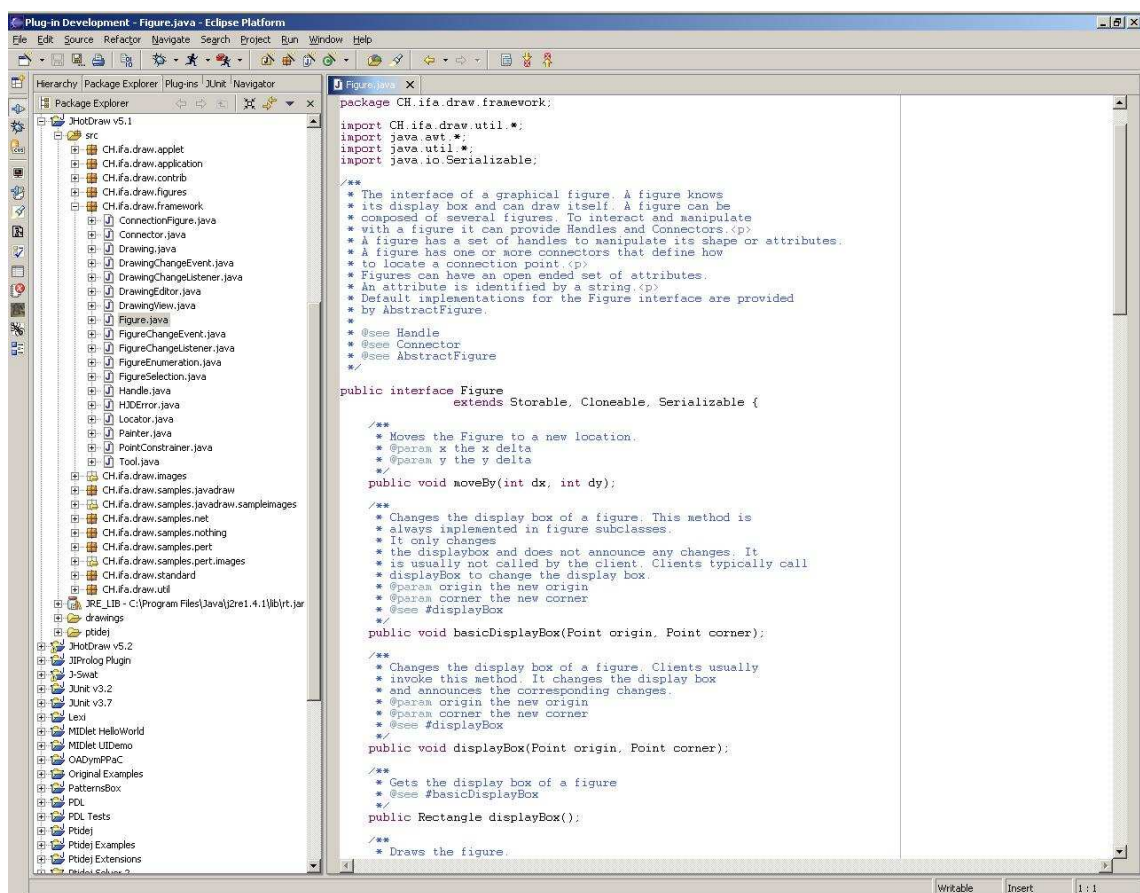




FIG. 3.6 – Modèle de JHOTDRAW au niveau idiomatique fourni avec la documentation du programme.

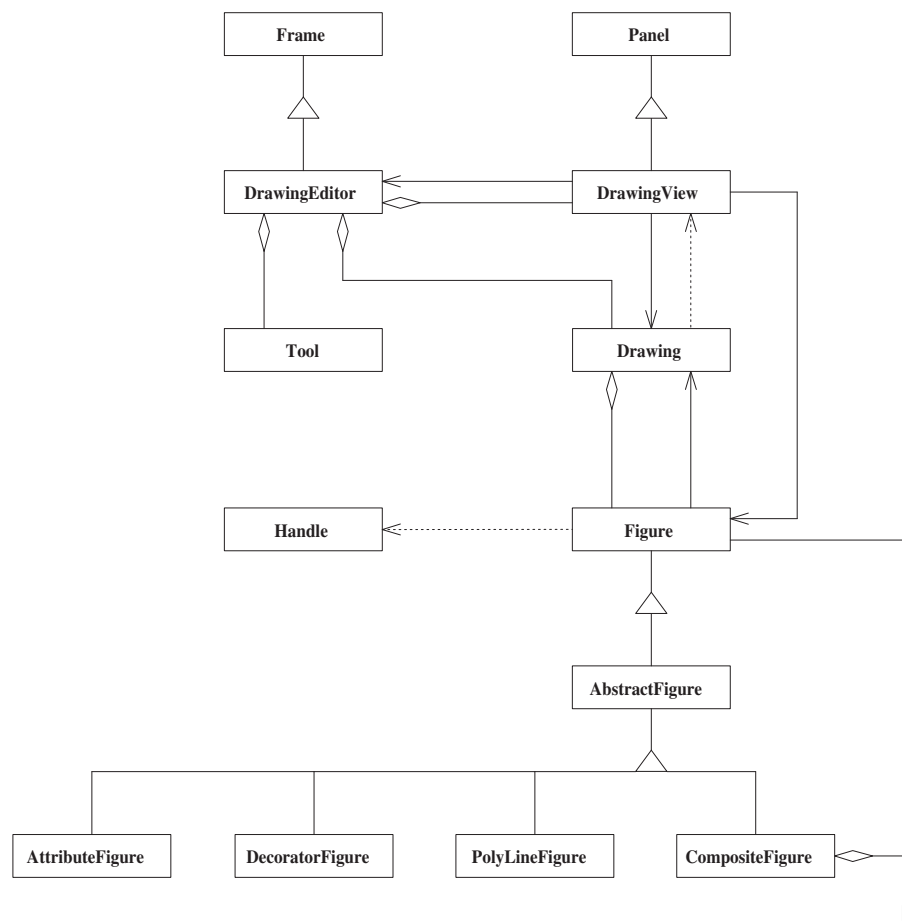
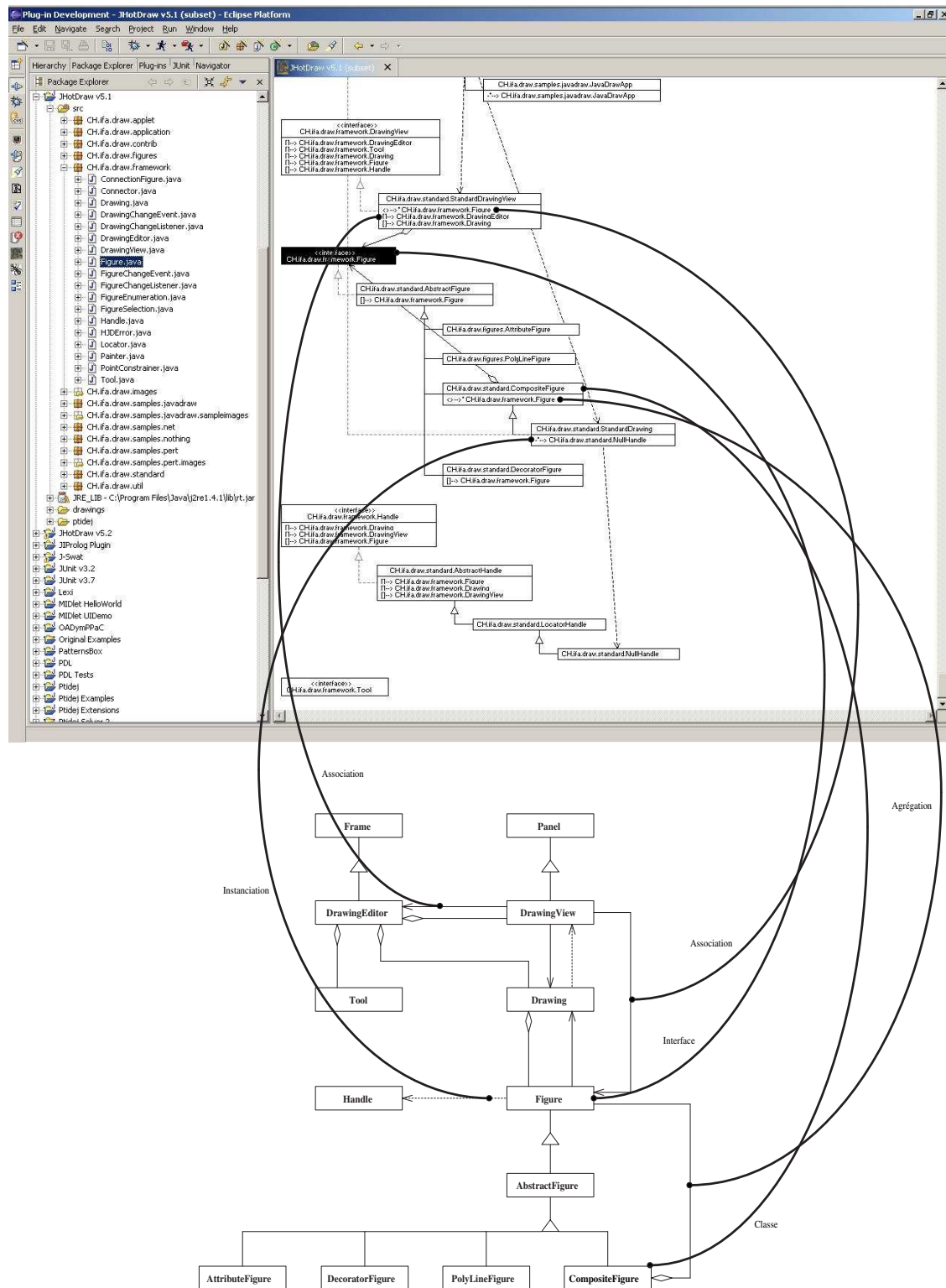


FIG. 3.7 – Correspondances entre les modèles de JHOTDRAW au niveau idiomatique fourni avec la documentation du programme et obtenu automatiquement.



Bilan

NOUS avons étudié la traçabilité des motifs interclasses **Association**, **Agrégation** et **Composition**. D'abord, nous avons proposé des modèles des programmes aux niveaux **implémentation** et **idiomatique**.

Au niveau **implémentation**, un programme est décrit par un modèle statique représentant son code source et par des modèles dynamiques représentant des séquences d'événements d'exécution du programme.

Au niveau **idiomatique**, un programme est décrit avec un métamodèle dédié pour représenter son modèle global. Ce métamodèle inclut les motifs interclasses **Association**, **Agrégation** et **Composition**.

Nous avons proposé des définitions consensuelles aux motifs interclasses aux niveaux **implémentation** et **idiomatique**. Nous avons isolé quatre propriétés minimales des définitions des motifs interclasses au niveau **implémentation** : durée de vie, exclusivité, multiplicité et site d'invocation. Nous avons défini précisément les motifs interclasses au niveau **implémentation** avec ces quatre propriétés et donné des exemples des motifs en JAVA.

Ces définitions nous ont permis de proposer un ensemble d'algorithmes pour garantir la traçabilité des motifs interclasses et des relations d'appel de méthodes, d'héritage et d'instanciation entre les niveaux **implémentation** et **idiomatique**.

Nous avons développé des algorithmes d'analyses statiques et dynamiques et des algorithmes de construction du modèle d'un programme au niveau **idiomatique**. Les algorithmes d'analyses statiques calculent les valeurs des propriétés de multiplicité et de site d'invocation dans le modèle statique du programme au niveau **implémentation**. Les algorithmes d'analyses dynamiques calculent les valeurs des propriétés de durée de vie et d'exclusivité dans les modèles dynamiques du programme au niveau **implémentation**.

Les algorithmes pour obtenir un modèle du programme au niveau **conception** construisent d'abord un modèle statique partiel du programme avec les algorithmes d'analyses statiques, puis raffinent ce modèle avec les résultats des analyses dynamiques.

Nous avons vérifié la cohérence de nos modèles, de nos définitions et de nos algorithmes sur des programmes connus, JHOTDRAW, JUNIT, et AWT, puis nous les avons appliqué à JHOTDRAW : le modèle obtenu est similaire au modèle fourni avec la documentation de ce programme.

Conclusion. Avec ces travaux sur la traçabilité des motifs interclasses, nous pouvons réaliser les quatre phases de la traçabilité des motifs interclasses décrites dans la section 1.2 page 14 et construire automatiquement le modèle d'un programme au niveau **idiomatique** avec son modèle au niveau **implémentation**. Nous étudions maintenant la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**, l'identification dans le modèle d'un programme au niveau **idiomatique** des micro-architectures similaires à des motifs de conception pour construire un modèle du programme au niveau **conception**.

Chapitre 4

Motifs de conception

NOUS étudions la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**. Les patrons de conception, tels que définis dans [Gamma *et al.*, 1994], proposent des solutions, sous la forme de motifs, à des problèmes récurrents d'organisation des classes dans l'architecture des programmes à objets.

Les motifs de conception sont décrits informellement et illustrés par des diagrammes de classes, par exemple le diagramme de classe du motif de conception **Composite** figure 1.7(c) page 25, qui ne représentent qu'une variante possible du motif et qui sont soumis à interprétation.

Les micro-architectures similaires aux motifs de conception sont disséminées dans l'architecture du programme au niveau **idiomatique** et sont donc difficiles à identifier. Les constituants de ces micro-architectures peuvent participer à plusieurs motifs et avoir des relations autres que celles préconisées par les motifs.

Pourtant, la connaissance des micro-architectures similaires à des motifs de conception facilitent les phases de rétroconception et de compréhension des programmes lors de la maintenance.

Nous décrivons des modèles et des algorithmes pour identifier, dans le modèle d'un programme au niveau **idiomatique**, les micro-architectures similaires à des motifs de conception. Les algorithmes d'identification prennent en compte les formes complètes et approchées des motifs de conception et facilitent l'interaction avec les mainteneurs.

Nos travaux sur la traçabilité des motifs de conception se décomposent comme suit :

- 4.1. Nous décrivons le modèle du programme dans lequel nous cherchons à identifier des motifs de conception, au niveau **idiomatique** avec le métamodèle présenté au chapitre précédent, section 3.1 page 79. Nous définissons un nouveau métamodèle pour décrire le modèle d'un programme au niveau **conception**. Ce métamodèle intègre les motifs de conception.
- 4.2. Nous discutons la modélisation des motifs de conception et nous montrons avec l'exemple du motif de conception **Composite** que l'identification des micro-architec-

tures similaires à un motif de conception se traduit en un problème de satisfaction de contraintes.

- 4.3. Nous faisons quelques rappels sur la programmation par contraintes et nous présentons la notion d'explication et son utilisation pour le débogage de programmes à contraintes et la résolution de problèmes sur-contraints.
- 4.4. Nous présentons une bibliothèque de contraintes associées à l'identification des motifs de conception et détaillons les différentes contraintes et leur sémantique.
- 4.5. Nous décrivons la résolution d'un problème de satisfaction de contraintes pour identifier les micro-architectures similaires à un motif de conception dans un modèle d'un programme.
- 4.6. Nous décrivons des algorithmes pour construire le modèle d'un programme au niveau **conception** et pour garantir la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**.
- 4.7. Nous vérifions la cohérence de nos travaux pour l'identification des motifs de conception avec la programmation par contraintes avec explications et nous discutons les modèles, les contraintes et les algorithmes proposés.
- 4.8. Nous appliquons la programmation par contraintes avec explications pour l'identification de micro-architectures similaires au motif de conception **Composite** dans un modèle de JHOTDRAW au niveau **idiomatique** et leur traçabilité entre les niveaux **idiomatique** et **conception**.

Enfin, nous dressons un bilan de nos travaux sur la traçabilité des motifs de conception.

4.1 Modélisation d'un programme

NOUS définissons maintenant les formalismes utilisés pour décrire les modèles d'un programme aux niveaux **idiomatique** et **conception** et pour garantir la traçabilité des motifs de conception.

Au niveau **idiomatique**, nous utilisons le métamodèle présenté dans la section 3.1 page 79. Au niveau **conception**, nous proposons un nouveau métamodèle pour distinguer les modèles d'un programme aux niveaux **idiomatique** et **conception**.

Les deux métamodèles sont similaires, leurs principales différences sont l'existence d'un constituant **Champ** au niveau **idiomatique** et l'existence d'un constituant **Micro-Architecture** au niveau **conception**.

Le modèle d'un programme au niveau **conception** prend la forme d'une instance de la classe **Modèle-Conception**. Il est constitué d'une collection d'entités (instances du constituant **Entité**). Chaque entité est composée d'une collection d'éléments (instances du constituant **Élément**) traduisant ses attributs. Il est aussi constitué d'une collection de micro-architectures (instances du constituant **Micro-Architecture**) décrivant les motifs de conception identifiés au niveau **idiomatique**. Chaque micro-architecture agrège une collection d'entités du modèle du programme.

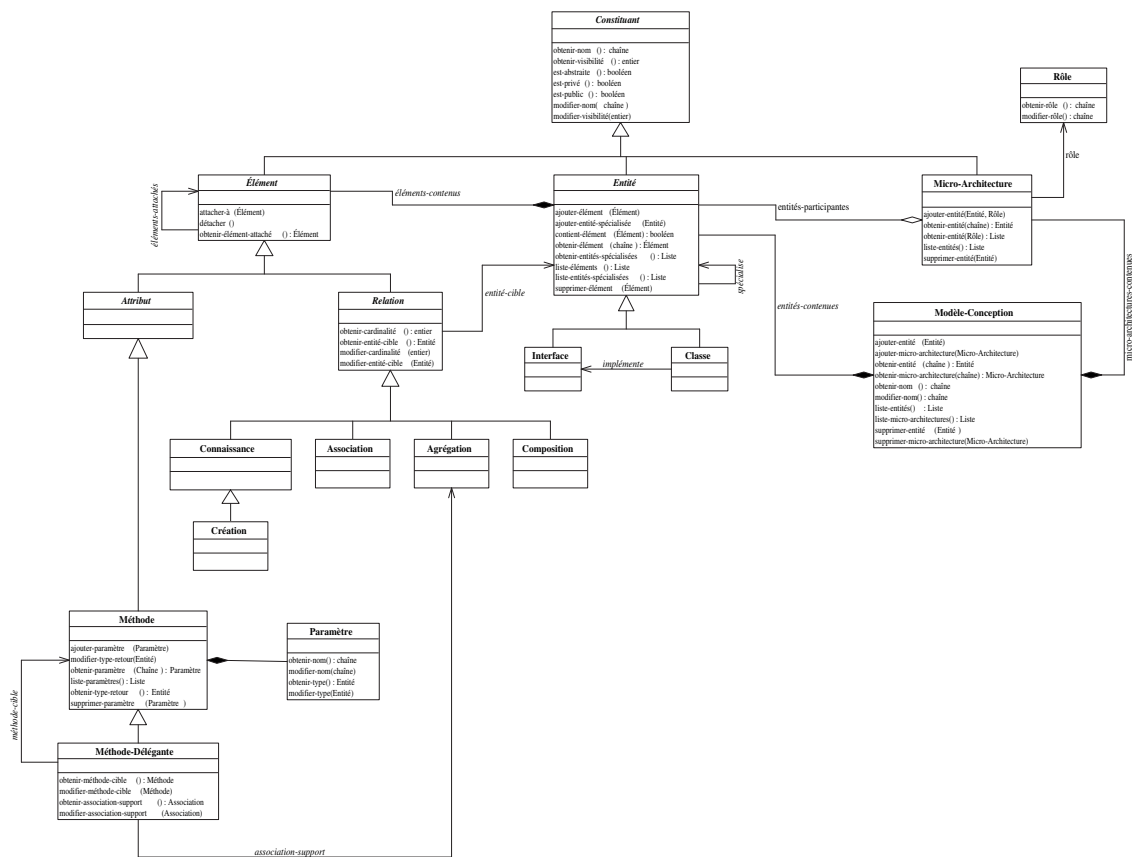
Ces entités, éléments et micro-architectures décrivent l'architecture et un sous-ensemble du comportement du programme, au niveau **conception**, et sont ajoutés au métamodèle au fur et à mesure de la modélisation de nouvelles architectures ou de nouveaux motifs interclasses, par spécialisation des classes **Entité** et **Élément**. Tout comme pour le précédent métamodèle, l'héritage est utilisé pour produire une taxonomie des constituants de l'architecture d'un programme.

La figure 4.1 page 146 montre les principaux constituants du métamodèle et leurs relations. Le métamodèle définit des entités, des éléments et des micro-architectures. Il définit la sémantique d'un modèle d'un programme niveau **conception** et nous utilisons une notation proche de la notation UML pour représenter visuellement les modèles qui en sont issus.

Les deux métamodèles pour décrire les modèles d'un programme aux niveaux **idiomatique** et **conception** proposent des constituants en commun, tels **Constituant**, **Entité** et **Élément**. Ils diffèrent pour les constituants suivants :

- nous choisissons de ne pas proposer un constituant **Champ** dans le métamodèle pour décrire le niveau **conception**. Nous pensons que les champs représentent une information de plus bas niveau dont les mainteneurs ne doivent pas se soucier au niveau **conception** ;
- le constituant **Micro-Architecture** n'existe pas dans le métamodèle pour décrire le niveau **idiomatique**. Ce constituant décrit une micro-architecture identifiée dans le modèle du programme au niveau **idiomatique** et similaire à un motif de conception. Une micro-architecture est une instance du constituant **Micro-Architecture**. La méthode `ajouter-entité()` ajoute une entité jouant le rôle donné à la micro-architecture. La méthode `obtenir-entité()` retourne une entité participant à une

FIG. 4.1 – Sous-ensemble du métamodèle pour décrire le modèle d'un programme au niveau conception.



micro-architecture avec son nom ou avec son rôle. Il est possible que plusieurs entités jouent un même rôle dans une micro-architecture ; par exemple, le rôle `Feuille` dans le motif de conception `Composite`. La méthode `liste-entités()` retourne la liste des entités participantes à une micro-architecture. La méthode `supprimer-entité()` supprime une entité participante à une micro-architecture.

- les constituants `Modèle-Conception` et `Modèle-Idiomatique` sont similaires. Le constituant `Modèle-Conception` propose en plus des méthodes pour gérer les micro-architectures identifiées au niveau `idiomatique` dans un modèle au niveau `conception` : `ajouter-micro-architecture()`, `obtenir-micro-architecture()`, `liste-micro-architectures()` et `supprimer-micro-architecture()`.

Conclusion. Nous modélisons les niveaux `idiomatique` et `conception` avec deux métamodèles similaires mais conceptuellement différents. Nous présentons maintenant la définition et la description des motifs de conception que nous cherchons à identifier au niveau `idiomatique` et dont nous voulons garantir la traçabilité entre les niveaux `idiomatique` et `conception`.

4.2 Définitions des motifs

LES PATRONS de conception proposent des solutions à des problèmes récurrents de conception des architectures des programmes à objets. Un patron de conception nomme et décrit un problème, sa solution, et les compromis associés à cette solution.

La solution d'un patron de conception est un motif de conception dont les variantes apparaissent comme des micro-architectures dans l'architecture du programme au niveau **idiotique** ; nous voulons identifier dans ce modèle d'un programme les micro-architectures similaires à un motif de conception.

D'abord, nous détaillons le contenu d'un motif de conception et les informations que nous voulons modéliser. Ensuite, nous présentons un métamodèle pour décrire les motifs de conception. Enfin, nous montrons que le problème de l'identification des micro-architectures similaires à un motif de conception se traduit en un problème de satisfaction de contraintes.

4.2.1 Patrons et motifs de conception

Un patron de conception de [Gamma *et al.*, 1994] suit un schéma général qui capture le problème résolu par le patron, la solution proposée et les décisions, les alternatives et les compromis associés à cette solution. Ce schéma est utilisé pour tous les patrons du catalogue pour faciliter leur compréhension, leur apprentissage et leur utilisation ; le tableau 4.1 page 153 résume le schéma¹ utilisé.

Avec ce schéma, un patron de conception est principalement décrit sous la forme de textes : les parties intention, autres désignations, motivation et applicabilité, qui expliquent le *problème* résolu par le patron, contiennent des informations difficilement formalisables. De même, les parties conséquences, usages connus et patrons connexes, qui précisent les *compromis* liés à la solution proposée par ce patron sont aussi décrites informellement. Seules les parties structure, participants, collaboration et implantation, qui présentent la *solution* du patron, utilisent des diagrammes de classes, d'interactions et des exemples de code source formalisables.

La solution d'un patron de conception, les parties structure, participants, collaboration et implantation, proposent un motif de conception dont les variantes sont appliquées dans l'architecture d'un programme pour résoudre le problème décrits par le patron. Nous voulons donc formaliser ce motif pour le rendre manipulable par un ordinateur pour identifier automatiquement dans le modèle d'un programme les micro-architectures similaires.

Les possibilités de modéliser les patrons de conception et leurs solutions et l'intérêt présenté par des modèles de solutions ont été présentés, par exemple, dans [Ducasse *et al.*, 1995 ; Pagel et Winter, 1996 ; Florijn *et al.*, 1997 ; Eden, 2000 ; Sunyé *et al.*, 2000].

¹Le schéma utilisé pour décrire les patrons de conception est lui-même un motif récurrent à la description de chacun des vingt-trois patrons.

4.2.2 Description des motifs

Le motif de conception proposé comme solution à un patron de conception est décrit par des diagrammes de classes et des diagrammes d'interactions avec une notation proche de OMT.

Nous cherchons à identifier ce motif de conception dans le modèle de l'architecture du programme au niveau **idiomatique**, nous modélisons donc aussi le motif de conception à ce même niveau.

Nous présentons sur la figure 4.2 page 150 un métamodèle pour décrire la structure et un sous-ensemble du comportement d'un motif de conception au niveau **idiomatique**. Ce métamodèle est similaire au métamodèle présenté section 3.1 page 79 car, par définition, il s'agit aussi d'un formalisme pour le niveau **idiomatique**.

Ce métamodèle est inspiré pour l'essentiel des travaux présentés dans un précédent mémoire de thèse de doctorat [Albin-Amiot, 2003], dans lequel l'auteur propose un métamodèle pour décrire les motifs de conception pour les synthétiser et les détecter.

Le métamodèle propose deux catégories de constituants : les participants et leurs éléments, respectivement les classes **Participant** et **Élément**. Participants et éléments ont un nom et des modificateurs. Les participants ont aussi un rôle attribué par le motif auxquels ils appartiennent.

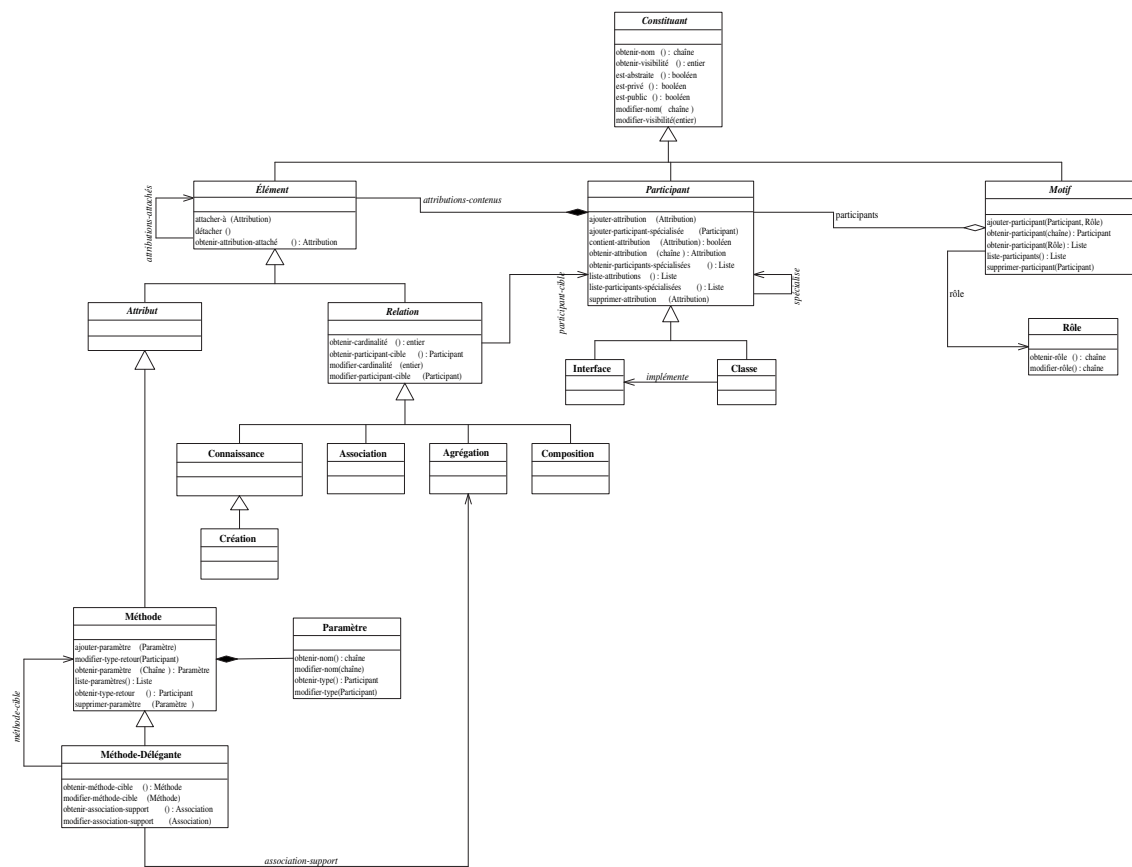
Avec les constituants de ce métamodèle, nous pouvons décrire les motifs de conception comme des entités de première classe manipulables par l'ordinateur. Un motif de conception est représenté par un constituant sous-classant le constituant **Motif** du métamodèle. Il est constitué de participants (instances du constituant **Participant**), chaque participant possède différents éléments (instances de **Élément**).

Le modèle d'un motif de conception est l'unique instance du constituant qui lui est dédié et qui spécialise le constituant **Motif**. Les différentes instances du motif, les *modèles concrets*^{*}, sont obtenues par clonage du *modèle abstrait*^{*}, unique et prototypique, d'un motif de conception [Albin-Amiot, 2003, page 85].

Nous décrivons, en utilisant les constituants du métamodèle, le motif de conception **Composite**, présenté sur la figure 1.7(c) page 25 par le modèle présenté sur la figure 4.4(a) page 152 avec la notation graphique résumée sur la figure 4.4(b) page 152.

La définition du motif **Composite** se décompose en quatre phases représentées sur la figure 4.3 page 152. La première phase consiste à définir le constituant **Composite**. D'abord, le métamodèle est raffiné (point 1, figure 4.3), pour ajouter tous les constituants structurels et comportementaux nécessaires à la modélisation du motif. Ces constituants sont par la suite disponibles pour la modélisation d'autres motifs. Dans le cas du motif **Composite**, le métamodèle présenté figure 4.1 est suffisant. Si la modélisation du motif nécessitait l'introduction d'une nouvelle propriété de classe, par exemple la notion de classe immuable, une nouvelle entité, **Classe-Immuable**, spécialisant l'entité **Classe**, serait rajoutée.

FIG. 4.2 – Sous-ensemble du métamodèle pour décrire un motif de conception.



Ensuite, la description du constituant **Composite** spécialisant le constituant générique **Motif** (point 2, figure 4.3) se fait en suivant la sémantique définie par le métamodèle : en énumérant ses différents constituants.

Ainsi, la déclaration du modèle d'un motif de conception est similaire à la déclaration d'une classe dans un langage à classes. Le constituant **Composite** joue deux rôles :

- il décrit les modèles qui pourront en être issus par instanciation ;
- il fixe les règles d'interactions avec les instances du motif **Composite** résultant de l'instanciation du constituant **Composite**.

Le constituant **Composite** est construit avec le diagramme de classes proposé dans [Gamma *et al.*, 1994], et repris figure 1.7(c) page 25, et de toutes les informations informelles traduisant le *leitmotiv* du motif [Eden, 2000] et qui décrivent le motif : une synthèse des rubriques structure, participants et collaboration.

Nous n'utilisons pas la notation UML pour représenter visuellement un modèle d'un motif de conception car les langages de conception, comme OMT et UML, sont trop ambiguës et pauvres pour représenter les motifs de conception [Eden *et al.*, 1998].

Cependant, notre notation visuelle est strictement limitée à la visualisation des modèles de motifs décrits avec notre métamodèle. Nous ne cherchons pas à remplacer une notation graphique comme celle qui accompagne le langage de description des motifs de conception LEPUS [Eden, 2000].

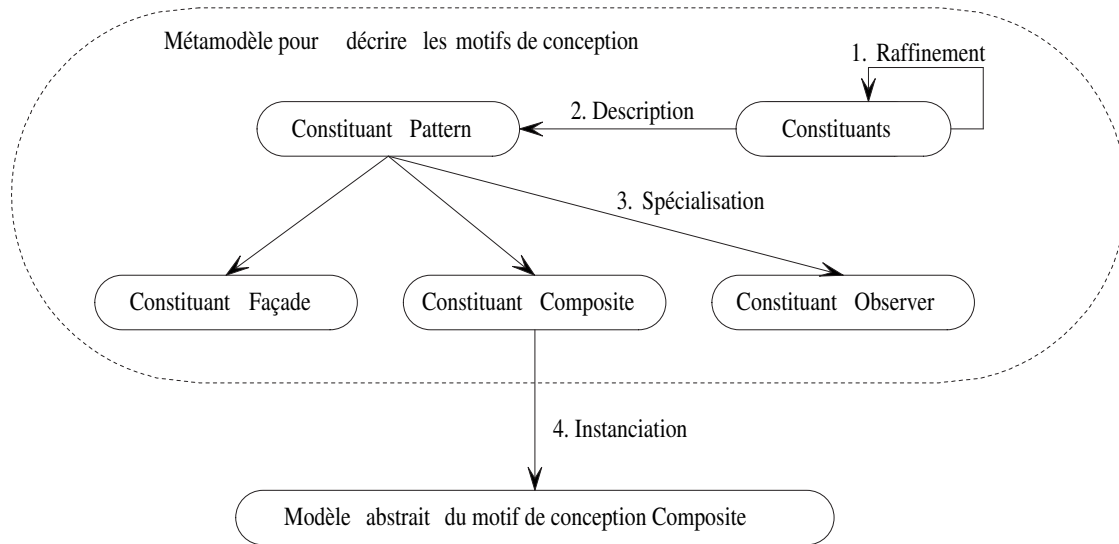
4.2.3 Motifs et problèmes de satisfaction de contraintes

Le modèle d'un motif de conception décrit les participants à la solution proposée par le patron de conception et leurs relations. Un mainteneur qui cherche à identifier dans un programme à objets un motif de conception doit identifier les constituants du modèle du programme au niveau **idiomatique** qui satisfont la modélisation des participants du motif et dont les relations satisfont les relations modélisées entre les participants du motif.

Problème de satisfaction de contraintes Le problème d'identification d'un motif de conception dans un programme à objets se traduit donc en un problème de satisfaction de contraintes pour lequel :

- les variables représentent les participants du modèle du motif de conception ;
- le domaine des variables représente les constituants du modèle du programme dans lequel identifier les micro-architectures similaires au modèle du motif de conception ;
- les contraintes représentent les relations entre les participants du modèle du motif de conception ;
- la sémantique des contraintes représente les relations effectives entre les constituants du modèle du programme.

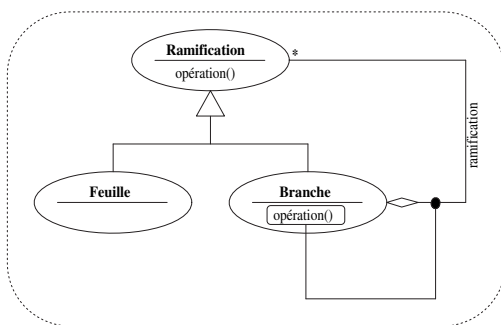
FIG. 4.3 – Processus de description des modèles abstraits des motifs de conception.



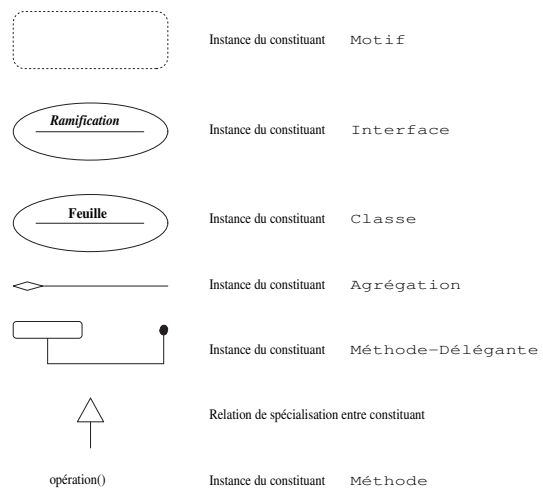
□

FIG. 4.4 – Modèle du motif de conception Composite et notation.

(a) Description du motif de conception Composite avec le métamodèle :



(b) Notation pour représenter les instances des constituants du métamodèle :



□

Tableau 4.1 – Schéma général des patrons de conception.

Nom et catégorie du patron

Identifie le patron et la catégorie à laquelle il appartient : créateur, structurel et comportemental.

Intention

Donne une courte description (deux-trois lignes) de la raison d'être du patron.

Aussi connu comme

Liste les autres noms connus du patron.

Motivation

Propose un scénario court illustrant le contexte d'utilisation du patron.

Applicabilité

Fournit une liste non exhaustive de cas où l'utilisation du patron est justifiée.

Structure

Propose une représentation graphique, proche de OMT, de l'architecture de la solution du patron.

Participants

Décrit les classes et les instances intervenant dans l'architecture de la solution du patron.

Collaborations

Décrit les interactions entre les participants (classes et instances).

Conséquences

Indique les conséquences de l'utilisation du patron.

Implantation

Propose des conseils et des techniques d'implantation.

Exemple de code source

Donne un exemple d'implantation en C++ et/ou en SMALLTALK.

Usages connus

Liste des utilisations du patron dans des programmes existants.

Patrons connexes

Indique les patrons qui sont en étroite relation avec le patron décrit.

□

Exemple du motif de conception Composite et de JHOTDRAW L'identification des micro-architectures similaires au modèle du motif de conception **Composite**, figure 4.4(a) page 152, dans le modèle du programme JHOTDRAW au niveau **idiomatique** se traduit en un problème de satisfaction de contraintes dans lequel :

- les variables sont au nombre de trois : **ramification**, **feuille** et **branche**, représentant les rôles **Ramification**, **Feuille** et **Branche** ;
- le domaine des variables est représenté par l'ensemble {**AttributeFigure**, **DecoratorFigure**, **PolyLineFigure**, **CompositeFigure**, ...} ;
- les contraintes sont au nombre de trois :
 - une contrainte d'héritage entre les variables **feuille** et **ramification** ;
 - une contrainte d'héritage entre les variables **branche** et **ramification** ;
 - une contrainte de composition entre les variables **branche** et **ramification** ;
- la sémantique des contraintes est donnée par les relations réelles entre les constituants du modèle du programme JHOTDRAW, par exemple :
 - une relation d'héritage entre le constituant **AttributeFigure** et le constituant **AbstractFigure** ;
 - une relation d'héritage entre le constituant **CompositeFigure** et le constituant **AbstractFigure** ;
 - une relation d'agrégation entre les constituants **CompositeFigure** et **Figure**.

Conclusion. Nous avons montré qu'un patron de conception offre suffisamment d'informations pour modéliser sa solution, un motif de conception, en une entité de première classe. Cette entité de première classe est décrite avec un métamodèle similaire au métamodèle pour décrire le niveau **idiomatique**. Nous avons aussi montré que l'identification de cette entité se traduit en un problème de satisfaction de contraintes. Nous utilisons ce problème pour identifier dans un programme les micro-architectures similaires au motif de conception décrit. Nous faisons maintenant des rappels sur la programmation par contraintes et la notion d'explications.

4.3 Programmation par contraintes et explications

Nous avons besoin d'explications, d'interactions et de dynamique lors de l'identification des micro-architectures similaires à un motif de conception. Nous pensons que la programmation par contraintes avec explications est une technique adéquate.

La programmation par contraintes avec explications permet de justifier les solutions, ou l'absence de solution, à un problème de satisfaction de contraintes [Jussien, 2001b] en indiquant les contraintes satisfaites ou qui ne peuvent être satisfaites.

Elle a été aussi utilisée pour développer de nouvelles stratégies de recherche dynamique, par exemple le retour arrière chronologique intelligent [Guéret *et al.*, 2000]. Nous faisons quelques rappels sur la programmation par contraintes avant de présenter la programmation par contraintes avec explications.

4.3.1 Programmation par contraintes

La *programmation par contraintes*^{*} est un paradigme de programmation au carrefour de diverses disciplines, telles l'intelligence artificielle, la recherche opérationnelle, la réécriture, le calcul formel, l'analyse numérique. Elle a montré son intérêt dans de nombreux domaines d'application, comme la résolution de problèmes combinatoires, l'ordonnancement, l'aide à la décision. Le plus étudié des formalismes appartenant au paradigme de la programmation par contraintes est le formalisme des problèmes de satisfaction de contraintes.

Le formalisme des *problèmes de satisfaction de contraintes*^{*2} permet de modéliser, d'étudier et de résoudre de nombreux problèmes combinatoires [Loudni, 2002]. Après avoir rappelé ce qu'est un problème combinatoire, nous donnons les principales définitions issues du formalisme des problèmes de satisfaction de contraintes. Puis, nous décrivons brièvement trois algorithmes (ou classes d'algorithmes) de résolution de ces problèmes : l'algorithme de retour en arrière, la classe des algorithmes rétrospectifs et la classe des algorithmes prospectifs. Enfin, nous présentons certaines limitations propres aux algorithmes de résolution présentés, limitations auxquelles la programmation par contraintes avec explications apporte un solution.

²Un problème de satisfaction de contraintes est appelé *constraint satisfaction problem* en anglais [Office québécois de la langue française, 2003] et a pour acronyme CSP.

Définition 4.1 – Problème combinatoire.

Un problème combinatoire $\mathcal{P}_c(\mathcal{S}, \mathcal{C})$ est défini par :

- un ensemble \mathcal{S} , fini ou infini, de solutions potentielles ;
- un ensemble de propriétés, ou de contraintes, \mathcal{C} à satisfaire.

La résolution d'un problème combinatoire consiste à trouver une solution admissible $s \in \mathcal{S}$ qui vérifie toutes les propriétés, ou qui satisfait toutes les contraintes $c_i \in \mathcal{C}$. \square

Le formalisme des problèmes de satisfaction de contraintes utilise la notion de système de contraintes :

Définition 4.2 – Problème de satisfaction de contraintes.

Un problème de satisfaction de contraintes à domaines finis est un triplet $\mathcal{R} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, où :

- $\mathcal{V} = \{v_1, \dots, v_n\}$ est un ensemble de n variables ;
- $\mathcal{D} = \{D_1, \dots, D_n\}$ est un ensemble de n domaines finis. Chaque domaine D_i représente l'ensemble des valeurs possibles pour la variable v_i ;
- $\mathcal{C} = \{C_1, \dots, C_e\}$ est un ensemble de e contraintes. Chaque contrainte $C_i = (V_i, R_i)$ d'arité q_i est définie par le sous-ensemble ordonné $V_i = \{v_{i_1}, \dots, v_{i_{q_i}}\}$ des q_i variables sur lesquelles la contrainte porte et par le sous-ensemble $R_i = D_{i_1} \times \dots \times D_{i_{q_i}}$ spécifiant les valeurs des variables $v_{i_1}, \dots, v_{i_{q_i}}$ compatibles entre elles. (La spécification des valeurs des variables par un sous-ensemble R_i représente une définition formelle de ces valeurs qui ne préjuge en rien de l'implantation des contraintes en intention ou en extension.)

Un système de contraintes est caractérisé par le triplet (n, d, e) où :

- $n = |\mathcal{V}|$ est le nombre de variables ;
- $d = \max_{i \in [1, n]}(|D_i|)$ est la taille du plus grand domaine ;
- $e = |\mathcal{C}|$ est le nombre de contraintes.

Nous limitons les domaines des variables à des ensembles finis de valeurs sur \mathbb{N} . Le couple $(\mathcal{V}, \mathcal{C})$ est appelé un système de contraintes. \square

Une fois un système de contraintes posé, le formalisme des problèmes de satisfaction de contraintes permet d'en trouver les solutions par instanciation³ des variables du système.

³Nous confondons les processus d'énumération et d'instanciation des variables car le processus d'énumération n'est pas important dans notre utilisation de la programmation par contraintes.

Définition 4.3 – Instanciation.

Un instanciation consiste à donner une valeur à une variable ou à un ensemble de variables :

- l'instanciation d'une variable v_i est l'affectation d'une valeur à cette variable par réduction de son domaine D_i à un singleton ;
- une instanciation complète \mathcal{A}_c est l'affectation d'une valeur à toutes les variables de l'ensemble \mathcal{V} par réduction de leur domaine respectif à un singleton.

Nous parlons d'instanciation vide lorsqu'aucune variable du système n'a été encore affectée. □

L'instanciation des variables permet de trouver une solution au système de contraintes.

Définition 4.4 – Solution.

Une solution à un problème de satisfaction de contraintes défini par le système de contraintes $\mathcal{R} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ est une instanciation complète \mathcal{A}_c qui satisfait toutes les contraintes de \mathcal{C} . Un problème ayant au moins une solution est dit cohérent ou satisfiable. Selon la théorie de la complexité, la satisfiabilité d'un problème définit en général un problème de décision NP-complet. \square

L'obtention d'une solution est réalisée par la résolution du système de contraintes.

Définition 4.5 – Résolution.

La résolution d'un problème défini par le système de contraintes $\mathcal{R} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ consiste à définir et à exécuter un algorithme pour obtenir une instanciation complète \mathcal{A}_c et cohérente des variables du système de contraintes. L'algorithme est intégré à un solveur de contraintes. Un solveur gère les problèmes de satisfaction de contraintes, les algorithmes de résolutions, les solutions. \square

La notion de cohérence est essentielle pour la résolution d'un problème de satisfaction de contraintes.

Définition 4.6 – Cohérence.

La notion de cohérence d'un sous-ensemble $\{v_i, \dots, v_k\}$ des variables d'un système de contraintes est indispensable pour réduire l'espace des solutions potentielles. Un sous-ensemble des variables d'un système de contraintes est cohérent par rapport à une propriété donnée si cette propriété est vérifiée pour toutes instanciations des variables. La réduction de l'espace des solutions est réalisé par *filtrage** des valeurs des domaines des variables $\{v_i, \dots, v_k\}$ qui ne satisfont pas la propriété de cohérence portant sur $\{v_i, \dots, v_k\}$. \square

Différentes propriétés de cohérence existent qui définissent des degrés de cohérence, par exemple la cohérence locale, la k -cohérence [Freuder, 1978], la k -cohérence forte [Freuder,

1978] et la cohérence d'arc [Montanari, 1974 ; Tsang, 1993]. La cohérence d'arc (aussi appelée consistance d'arc) correspond à une 2-cohérence forte et est sans doute la notion de cohérence la plus utilisée [Loudni, 2002], comme par exemple dans [Albin-Amiot *et al.*, 2002], où elle est appliquée à l'identification des formes complètes de motif de conception.

Les notions d'instanciation et de cohérence permettent d'exprimer l'absence de solution à un problème.

Définition 4.7 – Problème sur-contraint et contradiction.

Un système de contraintes est dit sur-contraint lorsqu'une instanciation complète, \mathcal{A}_c , et cohérente des variables du système ne peut être obtenue. La réduction de l'espace des solutions échoue alors car les contraintes, dont aucune valeur des domaines des variables ne satisfait les algorithmes de filtrage, induisent une contradiction. \square

Lorsqu'une contradiction survient lors de la résolution d'un problème, deux techniques sont possibles pour lever la contradiction.

Définition 4.8 – Relaxation du problème et de contraintes.

Un système de contraintes sur-contraint peut être modifié pour obtenir des solutions [Petit, 2002]. Il est possible :

- de relaxer le problème en retirant du système de contraintes les contraintes qui amènent la résolution à une contradiction ;
- de relaxer les contraintes qui amènent la résolution à une contradiction en les remplaçant par des contraintes sémantiquement moins fortes entre les mêmes variables.

\square

La résolution d'un problème de satisfaction de contraintes par filtrage des valeurs des domaines des variables incohérentes peut se faire avec différents algorithmes, reposant tous sur le principe de l'algorithme de retour en arrière⁴ proposé dans [Golomb et Baumert, 1965]. Ces algorithmes garantissent un parcours complet de l'espace des solutions, de l'ensemble des combinaisons des valeurs des variables potentiellement solutions.

Algorithme de retour en arrière chronologique Cet algorithme commence avec une instanciation vide des variables du système de contraintes. Il cherche alors à instancier chaque variable du système. Si l'instanciation d'une variable v_k est cohérente, alors

⁴Le principe de l'algorithme de retour en arrière est souvent désigné par le mot anglais *backtrack* [Office québécois de la langue française, 2003].

l'algorithme passe à la variable suivante. Si l'instanciation est incohérente, il revient sur l'instanciation de la variable v_k et essaie une autre valeur de son domaine D_k . Si toutes les valeurs du domaine D_k ont été essayées, l'algorithme effectue un *retour en arrière* sur la variable v_{k-1} et essaie une autre valeur de son domaine D_{k-1} . L'algorithme instancie les variables du système et leurs domaines jusqu'à une instanciation complète ou jusqu'à avoir parcouru tout l'espace de recherche. Dans le pire des cas, la complexité de l'algorithme de retour en arrière chronologique est en $O(e \times d^n)$. Une implantation de cet algorithme est décrite dans [Tsang, 1993, page 36]

Algorithmes rétrospectifs Les algorithmes rétrospectifs mettent à profit l'exploration de l'espace de recherche pour identifier les causes d'échec des instanciations et éviter les parcours infructueux dans l'espace de recherche. Ils analysent les causes des échecs soit pour choisir un meilleur point de retour dans le temps que l'algorithme de retour en arrière chronologique, soit pour identifier les instanciations partielles qui ne participent à aucune solution et ainsi éviter de les instancier. L'algorithme MAC-DBT [Jussien *et al.*, 2000] est un algorithme rétrospectif.

Algorithmes prospectifs Les algorithmes prospectifs réduisent la taille de l'espace de recherche en filtrant les valeurs des domaines des variables non instanciées qui ne participent à aucune solution. Les algorithmes de cohérence d'arc sont des algorithmes prospectifs [Montanari, 1974 ; Tsang, 1993].

4.3.2 Programmation par contraintes avec explications

La *programmation par contraintes avec explications** [Jussien, 2001b] est une extension de la programmation par contraintes dans laquelle le solveur fournit une explication de son comportement à chaque phase de la résolution du problème.

Nous considérons maintenant les problèmes de satisfaction de contraintes comme représentés par un couple $(\mathcal{V}, \mathcal{C})$. \mathcal{V} est un ensemble de variables et \mathcal{C} un ensemble de contraintes portant sur ces variables. Les domaines des variables sont représentés par des contraintes unaires. Ce formalisme unifie en un tout cohérent les contraintes et les domaines de variables.

Nous considérons l'instanciation comme une série d'ajouts et de retraits de contraintes spécifiques : les contraintes de décision. Nous ne nous limitons pas aux affectations de valeur à une variable mais acceptons tout type de contraintes de décision (comme, par exemple, des contraintes de séquençement entre tâches dans le domaine de l'ordonnancement, des contraintes de séparation dans les problèmes numériques).

Une *explication de contradiction**⁵ [Schiex et Verfaillie, 1994], est un sous-ensemble incohérent du système courant de contraintes : un problème contenant uniquement ce sous-ensemble de contraintes ne possède pas de solution.

⁵Une explication de contradiction est appelée *nogood* en anglais [Jussien, 1997].

Une explication de contradiction se décompose en deux parties : un sous-ensemble des contraintes initiales du problème, $\mathcal{C}' \subset \mathcal{C}$, et un sous-ensemble des contraintes de décision introduites jusqu'à la présente contradiction par la recherche d'une solution, $\{cd_1, \dots, cd_k\}$.

L'équation 4.1 représente une explication de contradiction spécifiant le sous-ensemble de contraintes \mathcal{C}' incohérent avec le sous-ensemble des contraintes de décision, des instantiations, $\{cd_1, \dots, cd_k\}$.

$$\neg (\mathcal{C}' \wedge cd_1 \wedge \dots \wedge cd_k) \quad (4.1)$$

Par réécriture, nous obtenons une vision opérationnelle des explications de contradiction de l'équation 4.1, présentée par l'équation 4.2 : le sous-ensemble de contraintes \mathcal{C}' et le sous-ensemble des contraintes de décision $\bigwedge_{i \in [1,k] \setminus j} cd_i$ interdisent l'ajout de la contrainte décision cd_j .

$$\mathcal{C}' \wedge \left(\bigwedge_{i \in [1,k] \setminus j} cd_i \right) \rightarrow \neg cd_j \quad (4.2)$$

Dans le cas où cd_j est de la forme $v_k = a$ (contrainte unaire représentant l'instanciation de la variable k à la valeur a du domaine), la partie gauche de l'implication est appelée une *explication de retrait*^{*} car elle justifie le retrait de la valeur a du domaine de la variable v_k . Elle est notée $\text{expl}(v_k \neq a)$.

Nous considérons les opérations de filtrage dans un problème de satisfaction de contraintes avec explications comme la production de nouvelles contraintes du type $v_k \neq a$, auxquelles nous associons une explication. L'explication la plus simple consiste simplement en l'ensemble des contraintes courantes : l'ensemble des contraintes initiales du problème plus l'ensemble de toutes les contraintes de décision ajoutées jusqu'à la présente contradiction.

Nous produisons de nouvelles explications d'explications existantes. En effet, si $(cd_1 \vee \dots \vee cd_j)$ est un ensemble de choix possibles pour une décision donnée (valeurs possibles, séquençements possibles), alors s'il existe des explications $\mathcal{C}'_1 \rightarrow \neg cd_1, \dots, \mathcal{C}'_j \rightarrow \neg cd_j$, on peut en déduire que $\neg(\mathcal{C}'_1 \wedge \dots \wedge \mathcal{C}'_j)$. En particulier, nous pouvons déduire l'explication de contradiction montrée sur l'équation 4.3 du domaine vide d'une variable v_k . Lorsqu'une explication de contradiction ne contient aucune contrainte de décision, alors le problème n'admet pas de solution, il est sur-contraint.

$$\neg \left(\bigwedge_{a \in D_v} \text{expl}(v \neq a) \right) \quad (4.3)$$

Il existe généralement plusieurs explications différentes pour un retrait d'une valeur du domaine d'une variable. Nous pourrions les conserver toutes mais ceci conduirait à une complexité spatiale exponentielle [Piechowiak et Rodriguez, 2000]. Une autre approche consiste à supprimer une partie des explications de contradiction : celles qui ne sont plus

pertinentes, celles dont toutes les contraintes sont encore valides dans l'état courant du système [Bayardo Jr. et Miranker, 1996]. De cette façon, la complexité spatiale de l'enregistrement reste polynômiale [Jussien, 1997]. Nous avons choisi de ne conserver qu'une seule explication de retrait à la fois, d'appliquer l'approche de [Bayardo Jr. et Miranker, 1996] qui a été prouvée correcte dans [Jussien, 1997].

L'algorithme 4.1 présente un algorithme de résolution des problèmes de satisfaction de contraintes avec explications générique. La résolution commence avec toutes les variables non instanciées, ligne 3. Tant qu'il reste une variable non instanciée, l'algorithme choisit (dans un ordre déterminé à l'avance ou dynamiquement) une variable à instancier, ligne 7, et la valeur à laquelle l'instancier parmi toutes les valeurs de son domaine, ligne 9. Alors, l'algorithme essaie, lignes 10 et 15–20, d'instancier la variable déterminée avec la valeur choisie en ajoutant au système de contraintes une contrainte de décision, ligne 13, et en propageant cet ajout dans le système de contraintes, ligne 14. Si l'instanciation échoue, lignes 15–20, alors l'algorithme de gestion des contradictions, algorithme 4.2 page 164, est appelé ; si le problème est prouvé sur-contraint, l'algorithme arrête la résolution, lignes 22–24 ; si toutes les variables sont instanciées, le système de contraintes est cohérent, une solution a été trouvée, ligne 25.

Les contradictions sont gérées par un algorithme de retour en arrière (chronologique ou rétrospectif) dans le cadre de la programmation par contraintes avec explications, par exemple l'algorithme 4.2 page 164.

Algorithme 4.1 – Résolution générique d'un problème de satisfaction de contraintes avec explications.

```

1 résoudre-générique(problème) ∈ {vrai, faux}
2 Début
3   variable-non-instanciées ← variables(problème)
4   Essai
5     TantQue variable-non-instanciées ≠ ∅ alors
6       // Choix de la prochaine variable à instancier :
7       prochaine-variable ← prochaine-variable(problème)
8       // Choix de la valeur avec laquelle instancier la variable :
9       valeur-variable ← valeur-variable(problème, prochaine-variable)
10      Essai
11        variable-non-instanciées ← variable-non-instanciées
12          \ { prochaine-variable }
13        problème ← problème ∪ { (prochaine-variable = valeur-variable) }
14        propage-modification(problème)
15      Attrape Contradiction-Instanciation
16        // Le domaine d'une variable est vide après propagation des
17        // modifications du problème. La gestion de la contradiction
18        // fait typiquement appel à un algorithme de retour en arrière :
19        gère-contradiction(problème)
20      FinEssai
21    FinTantQue
22    Attrape Contradiction-Problème
23    Retourne faux
24  FinEssai
25  Retourne vrai
26 Fin
```

□

L'algorithme commence par déterminer l'explication de contradiction associée avec le problème, ligne 3. S'il n'existe aucune contrainte de décision dans l'explication de contradiction, ligne 5, alors le problème est sur-contraint et n'admet pas de solution, ligne 8; sinon, l'algorithme choisit une contrainte de décision (chronologiquement ou rétrospectivement), ligne 10, puis essaie, lignes 11 et 17–20, de retirer cette contrainte de décision du système de contraintes, ligne 13, et d'ajouter la contrainte opposée, ligne 15 (pour entrer dans une autre branche de choix pour la valeur de la variable associée), puis de propager ces modifications dans le système de contraintes, ligne 16. Si une contradiction survient, l'algorithme continue récursivement jusqu'à un retrait réussi ou à une explication sans contraintes de décision, ligne 8.

Un algorithme de gestion des contradictions s'appelle aussi un algorithme de *réparation* car il répare la solution du problème de satisfaction de contraintes en construction lorsque l'instanciation d'une variable induit une contradiction.

Conclusion. Un problème de satisfaction de contraintes se compose d'un ensemble de contraintes liant des variables et d'un domaine pour ces variables. Nous pouvons résoudre un tel problème, expliquer les choix faits par le solveur et favoriser l'interaction avec les mainteneurs avec la programmation par contraintes avec explications. Nous présentons maintenant la bibliothèque de contraintes associées à la représentation de l'identification des motifs de conception par un problème de satisfaction de contraintes.

 Algorithme 4.2 – Gestion des contradictions.

```

1  gère-contradiction(problème) ∈ ∅
2  Début
3    explication ← raison-contradiction(problème)
4    contraintes-décision ← énumération-contraintes-décision(explication)
5    Si contraintes-décision = ∅ alors
6      // Si l'explication de contradiction ne contient aucune contrainte de
7      // décision, le problème est sur-contraint et n'a donc pas de solution :
8      lance-exception(Contradiction-Problème)
9    Sinon
10     contrainte-décision ← choisir-contrainte-décision(contraintes-décision)
11     Essai
12       variable-non-instanciées ← variable-non-instanciées
13       ∪ variable(contrainte-décision)
14       problème ← problème \ { contrainte-décision }
15       explication ← explication \ { contrainte-décision }
16       problème ← problème ∪ { contrainte-opposée(contrainte-décision) }
17       propage-modification(problème)
18     Attrape Contradiction-Instanciation
19       // Gestion récursive des contradictions :
20       gère-contradiction(problème)
21     FinEssai
22   FinSi
23 Fin

```

□

4.4 Contraintes associées aux motifs

LA DÉTECTION des micro-architectures similaires à un motif de conception par la programmation par contraintes avec explications nécessite la résolution d'un problème de satisfaction de contraintes caractérisé par les contraintes entre les participants du motif de conception.

Nous détaillons maintenant la bibliothèque de contraintes associées à la description du motif de conception par un problème de satisfaction, construite en étudiant et en développant les relations interclasses définies dans [Gamma *et al.*, 1994].

La bibliothèque de contraintes offre les contraintes binaires, entre deux variables v_1 et v_2 , suivantes :

- *égalité* : assure que les valeurs des domaines des deux variables sont égales : $v_1 = v_2$;
- *inégalité* : assure que les valeurs des domaines des deux variables sont différentes : $v_1 \neq v_2$;
- *héritage strict* : établit une contrainte d'héritage strict entre les deux variables telle que définie dans l'exemple 4.1. Nous dérivons une contrainte d'héritage de cette contrainte d'héritage strict telle que les deux variables peuvent représenter les mêmes entités. Nous définissons aussi deux contraintes sur la fermeture transitive de ces contraintes : l'appartenance des entités des domaines des variables à une même branche de l'arbre d'héritage, strictement ou non.

Exemple 4.1 – Contrainte d'héritage strict.

Une relation d'héritage lie deux entités dans une relation de type parent-enfant, ou super-entité-sous-entité. Dans le cas de l'héritage simple (à contraster avec l'héritage multiple), la relation d'héritage crée un ordre partiel $<$ sur l'ensemble des entités⁶.

Pour tout couple de variables distinctes v_1 et v_2 , si les valeurs du domaine de v_1 représentent un ensemble d'entités qui héritent des entités appartenant au domaine de v_2 alors la contrainte $v_1 < v_2$ est vérifiée. La sémantique opérationnelle de cette contrainte est (D_{v_i} représente le domaine de la variable v_i) :

$$\forall c_1 \in D_1, \exists c_2 \in D_2, c_1 < c_2$$

$$\forall c_2 \in D_2, \exists c_1 \in D_1, c_1 < c_2$$

□

⁶L'ordre partiel exprime les positions relatives des deux entités dans l'arbre d'héritage, pas la relation de typage entre les deux entités ; nous utilisons la notation *sous-entité* $<$ *super-entité*, pas la notation *sous-type* $>$ *super-type*.

- *héritage* : établit une contrainte d'héritage entre les valeurs des domaines des deux variables. Les domaines des deux variables peuvent contenir les mêmes valeurs, donc représenter les mêmes entités, contrairement à la contrainte précédente : $v_1 \leq v_2$.
- *héritage transitif strict* : établit une contrainte d'héritage entre les valeurs des domaines des deux variables, les valeurs des domaines représentent des entités différentes appartenant à la même branche de l'arbre d'héritage : $v_1 \leqslant v_2$.
- *héritage transitif* : établit une contrainte d'héritage entre les valeurs des domaines des deux variables, les domaines des deux variables contiennent des valeurs qui représentent des entités appartenant à la même branche de l'arbre d'héritage : $v_1 \leqslant v_2$.
- *connaissance* : établit une contrainte de connaissance entre les valeurs des domaines des deux variables. Une entité A connaît une entité B si des méthodes définies dans l'entité B sont invoquées par des méthodes de l'entité A. Cette relation est binaire, orientée et non-transitive, telle que présentée dans l'exemple 4.2. Nous notons cette contrainte : $v_1 \xrightarrow{\kappa} v_2$.

Exemple 4.2 – Contrainte de connaissance.

Il existe une relation de connaissance entre une entité A et une entité B lorsque l'entité A a connaissance de l'existence de l'entité B. Cette relation se matérialise dans le code par l'invocation depuis l'entité A (ou depuis une instance de cette entité) de méthodes définies par l'entité B (méthodes de entité ou d'instance). Cette relation est binaire (entre deux entités ou deux instances) et orientée (que l'entité A ait connaissance de l'entité B n'implique pas que l'entité B ait connaissance de l'entité A). □

- *ignorance* : assure la contrainte opposée à la contrainte précédente, les valeurs du domaine de la variable v_1 représentent des entités qui ne doivent pas avoir connaissance des entités appartenant au domaine de la variable v_2 : $v_1 \not\xrightarrow{\kappa} v_2$. Cette contrainte est nécessaire pour contraindre *explicitement* l'ignorance entre deux variables.
- *composition* : assure qu'un motif **Composition** lie les valeurs des domaines des deux variables : $v_1 \blacktriangleright v_2$.
- *agrégation* : assure qu'un motif **Agrégation** lie les valeurs des domaines des deux variables : $v_1 \triangleright v_2$.
- *association* : assure qu'un motif **Association** lie les valeurs des domaines des deux variables : $v_1 \rightarrow v_2$.
- *création* : établit qu'une entité représentée dans le domaine de la variable v_1 instancie (au moins une fois) une entité appartenant au domaine de la variable v_2 : $v_1 \xrightarrow{*} v_2$.

La bibliothèque de contraintes propose aussi des contraintes ternaires, entre trois variables v_1 , v_2 et *distance*, pour contraindre la distance entre deux entités pour une relation transitive entre entités ; par exemple :

- *distance dans l'arbre d'héritage* : calcule la distance entre deux entités dans l'arbre d'héritage. Si les deux entités n'appartiennent pas à la même branche de l'arbre d'héritage, la distance entre ces entités est infinie ; si les deux entités sont égales, la distance est nulle ; sinon, la distance est égale au nombre d'entités les séparant sur la branche de l'arbre d'héritage plus un. Nous écrivons : $d_{\leq}(v_1, v_2, distance)$, la variable numérique *distance* représente la distance entre les entités des domaines des variables v_1 et v_2 .
- *distance par rapport à l'association* : calcule la distance entre deux entités par rapport à la relation d'association. Nous considérons que la relation d'association induit un graphe \mathcal{G} entre les entités du programme. S'il existe un chemin minimal de \mathcal{G} entre deux entités, la distance entre ces entités est le nombre d'entités rencontrées sur ce chemin moins un ; si les deux entités sont égales, la distance est nulle ; s'il n'est pas possible de trouver un chemin entre ces entités, la distance est infinie. Nous notons cette contrainte : $d_{\perp}(v_1, v_2, distance)$, la variable numérique *distance* représente la distance entre les entités des domaines des variables v_1 et v_2 .

Nous associons un poids à chacune des contraintes caractérisant un problème de satisfaction de contraintes avec explications pour les ordonner et les présenter aux mainteneurs dans un ordre consistant et cohérent. Le poids associé avec une contrainte est un entier $p \in [1, 100] \cup \mathbb{N}$. Il peut indiquer l'importance relative de la contrainte à laquelle il est associé dans le problème ou simplement l'ordre des contraintes.

Les nombreuses contraintes offertes par la bibliothèque nous permettent de modéliser un grand nombre de motifs de conception, quelles que soient leurs caractéristiques. Elles ne sont pas utilisées pour modéliser les variantes d'un motif de conception. Les algorithmes de résolution utilisent le modèle d'un motif de conception et les autres contraintes de la bibliothèque pour identifier les micro-architectures similaires à ce motif, identiques ou approchées, et pour guider les mainteneurs.

Pour reprendre l'exemple du modèle du motif de conception **Composite**, présenté dans la section 3.2 page 84, les contraintes entre les trois variables, **ramification**, **feuille** et **branche**, définies par le système de contraintes associé au motif de conception **Composite** sont :

- une contrainte d'héritage entre les variables **feuille** et **ramification** :
feuille < **ramification**
- une contrainte d'héritage entre les variables **branche** et **ramification** :
branche < **ramification**
- une contrainte de composition entre les variables **branche** et **ramification** :
branche ► **ramification**

Conclusion. Nous avons détaillé les contraintes associées à la représentation d'un motif de conception par un système de contraintes. Nous montrons maintenant comment nous résolvons le problème de satisfaction de contraintes associé avec la programmation par contraintes avec explications pour identifier les micro-architectures similaires à un motif de conception dans le modèle d'un programme au niveau **idiomatique**.

4.5 Algorithmes de résolution des problèmes de satisfaction de contraintes

NOUS proposons deux stratégies de recherche des solutions d'un problème de satisfaction de contraintes avec explications novatrices pour identifier les micro-architectures similaires à un motif de conception.

Nous offrons aux mainteneurs les possibilités de relaxer les contraintes et le problème pour guider interactivement l'identification ou pour obtenir combinatoirement toutes les micro-architectures possibles.

La relaxation des contraintes est intéressante car des constituants ou des motifs peuvent être liés par une relation d'ordre, telle la relation d'inclusion, présentée dans la section 3.4 page 101, qui existe entre les motifs **Association**, **Agrégation** et **Composition**.

La relaxation du problème est intéressante car des constituants ou des motifs peuvent ne pas avoir d'équivalents dans le modèle du programme ; le problème de satisfaction de contraintes est alors sur-contraint.

Dans un premier temps, nous décrivons le principe d'une recherche basée sur les explications. Puis, nous développons deux stratégies de recherche, en décrivant les algorithmes de recherche et de réparation qui soutiennent ces stratégies.

4.5.1 Principes

La résolution est basée sur l'interaction avec les mainteneurs. D'abord, le solveur cherche les *solutions complètes**, les formes complètes des micro-architectures similaires à un motif de conception. Puis, il fournit une explication à l'absence d'autres solutions.

Le mainteneur peut alors sélectionner une ou plusieurs contraintes de l'explication dont la vérification n'est pas fondamentale de son point de vue. Cette sélection est alors retirée dynamiquement du système de contraintes et des *solutions approchées** sont recherchées, les formes approchées des micro-architectures.

Ce processus continue jusqu'à ce que le mainteneur décide que trop de contraintes ont été relaxées (s'éloignant alors trop du motif originel). Le système permet aussi la réintégration de contraintes si cela est requis par le mainteneur.

Pour aider l'utilisateur dans sa tâche, nous avons attribué *a priori* un poids à chacune des contraintes du système, pour classer les affaiblissements intéressants. Les solutions approchées fournies sont annotées par leur écart par rapport à la solution complète. L'écart est exprimé par une valeur entière calculée avec les poids p_1, \dots, p_a des contraintes affaiblies ou retirées du problème, par exemple :

$$\text{écart} = \prod_{p \in \{p_1, \dots, p_a\}} p/100$$

Le système peut aussi être automatisé pour passer en revue toutes les possibilités : les contraintes sont alors relaxées combinatoirement, dans l'ordre défini par les poids des contraintes, pour trouver les solutions approchées et les classer. L'ensemble maximal des solutions approchées possibles est le même que la recherche soit dirigée par l'utilisateur ou soit automatisée. Cet ensemble maximal dépend uniquement de la modélisation choisie pour le motif de conception.

La différence entre une résolution guidée par l'utilisateur et une résolution automatique est que l'utilisateur peut choisir de relaxer les contraintes dans un ordre différent de celui proposé dans la modélisation. L'utilisateur peut ainsi faire apparaître plus rapidement les solutions approchées qui lui semblent intéressantes et donc contourner la difficulté d'attribuer *a priori* une importance relative aux contraintes reflétant correctement ses objectifs.

Cette interaction entre l'utilisateur et le système de contraintes est très importante pour notre application d'identification des motifs de conception. Il est en effet nécessaire que l'utilisateur ait un contrôle total sur la recherche. La programmation par contraintes avec explications est une technique particulièrement adaptée pour un haut niveau d'interactions avec l'utilisateur et pour identifier les variantes d'un motif en relaxant les contraintes liées au motif.

4.5.2 Algorithmes de résolution générique et de choix des contraintes interactifs

La résolution d'un problème de satisfaction de contraintes avec explications avec les algorithmes 4.1 page 163, et 4.2 page 164, est basée sur un algorithme de retour en arrière chronologique.

Nous étendons ces algorithmes pour prendre en compte les directives du mainteneur, qui peut vouloir retirer ou ajouter à nouveau une contrainte dans le système de contraintes. Nous proposons un nouvel algorithme de réparation pour retirer et pour ajouter des contraintes au système en cas de contradiction.

Nous utilisons cet algorithme plus général en combinaison avec un algorithme de choix des contraintes à retirer ou à ajouter guidé par le mainteneur. Ces deux algorithmes permettent au mainteneur de guider interactivement la recherche de solutions approchées.

L'algorithme 4.3 page 171 décrit l'algorithme générique de réparation pour le retrait et l'ajout de contraintes du problème de satisfaction de contraintes. D'abord, l'algorithme demande au mainteneur de choisir les contraintes à retirer du problème et les contraintes à ajouter au problème, ligne 3. Ensuite, pour chaque contrainte à retirer du problème, lignes 6 et 7, l'algorithme met à jour les variables non instanciées, lignes 8–9, retire la contrainte du problème, ligne 10, et ajoute la contrainte opposée au problème, ligne 11, pour entrer dans une autre branche de choix. Puis, pour chaque contrainte à ajouter au problème, lignes 15 et 16, l'algorithme ajoute la contrainte désirée au problème, ligne 17. Enfin, l'algorithme propage les modifications du problème, ligne 20. La propagation des contraintes peut induire des contradictions, lignes 4 et 22–25.

L'algorithme 4.5 page 173 décrit un algorithme pour choisir la ou les contraintes à retirer ou à ajouter au système. L'algorithme commence par obtenir l'explication de contradiction, ligne 5. Puis, il affiche toutes les contraintes qui sont présentes dans l'explication et leurs successeurs affaiblis et demande au mainteneur lesquelles de ces contraintes relaxer ou retirer du système de contraintes, lignes 7–22 ; l'algorithme fait la même chose pour les contraintes qui avaient été retirées précédemment et que le mainteneur peut vouloir réintégrer, lignes 24–37. Finalement, l'algorithme retourne le couple des ensembles de contraintes à retirer et de contraintes à ajouter à nouveau au système de contraintes, ligne 38.

Ce premier algorithme permet au mainteneur de relaxer les contraintes et le problème en retirant les contraintes jugées trop contraignantes et en les remplaçant par des contraintes affaiblies, calculées avec l'algorithme 4.7 page 175.

L'algorithme de construction d'une contrainte affaiblie ne décrit qu'un sous-ensemble des relaxations de contraintes possibles. Il crée une nouvelle contrainte portant sur les mêmes variables que la contrainte à relaxer mais représentant un affaiblissement sémantique de cette contrainte.

Algorithme 4.3 – Gestion générique des contradictions.

```

1  gère-contradiction(problème) ∈ ∅
2  Début
3      contraintes ← choisir-contraintes(problème)
4      Essai
5          // Relaxation du problème (contraintes à retirer) :
6          Si contraintes = (c1, c2) | c1 ≠ ∅ alors
7              Pour chaque contrainte ∈ c1 faire
8                  variable-non-instanciées ← variable-non-instanciées
9                  ∪ variable(contrainte)
10                 problème ← problème \ { contrainte }
11                 problème ← problème ∪ { contrainte-opposée(contrainte) }
12             FinPour
13         FinSi
14         // Modification du problème (contraintes à ajouter) :
15         Si contraintes = (c1, c2) | c2 ≠ ∅ alors
16             Pour chaque contrainte ∈ c2 faire
17                 problème ← problème ∪ { contrainte }
18             FinPour
19         FinSi
20         propage-modification(problème)
21     Attrape Contradiction-Instanciation
22     // Gestion récursive des contradictions :
23     gère-contradiction(problème)
24     FinEssai
25 Fin
```

□

Par exemple, les définitions des motifs interclasses **Association**, **Agrégation** et **Composition** induisent une relation d'inclusion, comme présenté sur la figure 3.3 page 105, section 3.4 page 101. Le motif **Composition** représente ainsi une relation plus forte que le motif **Agrégation**, qui lui-même représente une relation plus forte que le motif **Association**. Avec cet ordre, il est possible de remplacer la contrainte de composition qui amène à une contradiction par une contrainte d'agrégation, la contrainte d'agrégation par une contrainte d'association.

L'interaction avec le mainteneur permet d'obtenir toutes les solutions intéressantes au problème de contraintes et donc d'obtenir toutes les micro-architectures similaires à un motif de conception dans le modèle d'un programme au niveau **idiomatique**.

4.5.3 Algorithmes de résolution et de relaxation automatique

Cependant, l'interaction avec le mainteneur oblige celui-ci à participer physiquement à l'identification des micro-architectures. Cette présence peut être trop contraignante lors de l'identification de micro-architectures similaires à plusieurs motifs de conception dans des programmes de grandes tailles. De plus, le mainteneur peut avoir une connaissance limitée des motifs de conception et guider la recherche difficilement.

Nous proposons de nouveaux algorithmes de résolution et de choix des contraintes à ajouter ou à retirer pour rechercher systématiquement toutes les combinaisons possibles de contraintes pour lesquelles une solution existe par relaxations successives des contraintes et du problème.

L'algorithme 4.4 présente un algorithme de résolution automatique qui cherche les solutions aux systèmes de contraintes formés par toutes les combinaisons des contraintes du système initial.

Algorithme 4.4 – Recherche automatique de toutes les solutions.

```

1 résoudre-automatique(problème) ∈ {vrai, faux}
2 Début
3   contraintes ← contraintes(problème)
4   e ← | contraintes |
5   solution-existe ← faux
6   Pour p ∈ [1, e] faire
7     combinaisons-contraintes ←  $C_e^p$ 
8     Pour c ∈ combinaisons-contraintes faire
9       problème-dynamique ← (V, c)
10      solution-existe ← solution-existe ∨ résoudre-générique(problème-dynamique)
11    FinPour
12  FinPour
13  Retourne solution-existe
14 Fin
```

□

Algorithme 4.5 – Choix des contraintes à ajouter et à retirer.

```

1  choisir-contrainte-décision(problème) = (E, F) | E ⊂ C, F ⊂ C
2  Début
3      ensemble-contraintes-retirer ← {}
4      ensemble-contraintes-ajouter ← {}
5      explication ← raison-contradiction(problème)
6
7      contraintes ← énumération-contraintes(explication)
8      affiche("Il n'y a plus de solutions à cause des contraintes :")
9      TantQue existe-suivante(contraintes) alors
10         contrainte ← suivante(contraintes)
11         affiche(nom(contrainte))
12         affiche(poids(contrainte))
13         affiche("à remplacer par")
14         affiche(suivante(contrainte))
15     FinTantQue
16     affiche("Quelle(s) contrainte(s) voulez-vous relaxer/retirer?")
17     choix ← lire-clavier()
18     TantQue choix ≠ ∅ alors
19         ensemble-contraintes-retirer ← ensemble-contraintes-retirer
20             ∪ { get(contraintes, choix) }
21         choix ← lire-clavier()
22     FinTantQue
23
24     contraintes ← énumération-contraintes-déjà-retirer(explication)
25     affiche("Les contraintes suivantes avaient amené à une contradiction :")
26     TantQue existe-suivante(contraintes) alors
27         contrainte ← suivante(contraintes)
28         affiche(nom(contrainte))
29         affiche(poids(contrainte))
30     FinTantQue
31     affiche("Quelle(s) contrainte(s) voulez-vous ajouter à nouveau?")
32     choix ← lire-clavier()
33     TantQue choix ≠ ∅ alors
34         ensemble-contraintes-ajouter ← ensemble-contraintes-ajouter
35             ∪ { get(contraintes, choix) }
36         choix ← lire-clavier()
37     FinTantQue
38     Retourne (ensemble-contraintes-retirer, ensemble-contraintes-ajouter)
39 Fin

```

□

L'algorithme commence par obtenir l'ensemble des contraintes du système et sa cardinalité, lignes 3 et 4. Puis, il énumère les ensembles de combinaisons possibles, ligne 6, par calcul combinatoire, ligne 7. Pour chaque combinaison possible, ligne 8, il crée dynamiquement un problème de satisfaction de contraintes tel que l'ensemble des variables est égal à l'ensemble des variables du problème initial et que l'ensemble des contraintes soit un sous-ensemble des contraintes du système originel, ligne 9. Il résout alors ce problème, ligne 10, et se termine après avoir essayé toutes les combinaisons de contraintes, ligne 13.

L'algorithme 4.6 page 175 présente un mécanisme relaxation des contraintes. Il commence par obtenir l'explication de contradiction du problème, ligne 5 ; puis, la meilleure contrainte à retirer, lignes 7 et 8. S'il existe une contrainte approchée de la contrainte à retirer, ligne 10, la contrainte est retirée et une contrainte approchée la remplace, lignes 11 et 12.

La meilleure contrainte est celle dont le poids est le plus faible parmi toutes les contraintes liées à l'explication de contradiction. Le retrait de cette meilleure contrainte a l'impact le plus faible sur la sémantique du problème de satisfaction de contraintes, par rapport aux autres contraintes liées à l'explication de contradiction.

Ces relaxations successives permettent au solveur de trouver des solutions même si les contraintes caractérisant le motif ne sont pas satisfiables avec les domaines des variables obtenus du modèle du programme.

Conclusion. Nous avons implanté deux stratégies de recherche pour l'identification des micro-architectures similaires à un motif de conception avec la programmation par contraintes avec explications. Les deux stratégies utilisent la relaxation des contraintes et des problèmes interactivement ou combinatoirement et le chaînage successifs des contraintes. Nous utilisons maintenant ces stratégies pour garantir la traçabilité des motifs.

 Algorithme 4.6 – Algorithme de relaxation des contraintes.

```

1  choisir-contrainte-décision(problème) = (E, F) | E ⊂ C, F ⊂ C
2  Début
3      ensemble-contraintes-retirer ← {}
4      ensemble-contraintes-ajouter ← {}
5      explication ← raison-contradiction(problème)
6
7      contraintes ← énumération-contraintes(explication)
8      contrainte ← meilleure(contraintes)
9
10     Si suivante(contrainte) ≠ ∅ alors
11         ensemble-contraintes-retirer ← ensemble-contraintes-retirer
12             ∪ { contrainte }
13         ensemble-contraintes-ajouter ← ensemble-contraintes-ajouter
14             ∪ { suivante(contrainte) }
15     FinSi
16
17     Retourne (ensemble-contraintes-retirer, ensemble-contraintes-ajouter)
18 Fin

```

□

 Algorithme 4.7 – Algorithme de création d'une contrainte affaiblie.

```

1  suivante(contrainte) ⊂ C | C ensemble des contraintes de la bibliothèque
2  Début
3      contrainte-affaiblie ← ∅
4      Choisir genre(contrainte)
5      Cas Composition
6          contrainte-affaiblie ← crée-agrégation(contrainte)
7      Cas Agrégation
8          contrainte-affaiblie ← crée-association(contrainte)
9      Cas Association
10         contrainte-affaibli ← crée-connaissance(contrainte)
11     ...
12     FinChoisir
13
14     Retourne contrainte-affaibli
15 Fin

```

□

4.6 Algorithmes de traçabilité des motifs

Nous présentons la mise en œuvre des modèles des motifs de conception et des algorithmes pour les identifier et en garantir la traçabilité entre les niveaux **idiomatique** et **conception**.

L'algorithme 4.8 page 177 construit le modèle d'un programme au niveau **conception**. D'abord, nous créons un nouveau modèle du programme, ligne 4. Ensuite, pour chaque entité du modèle **idiomatique**, nous créons une entité équivalente dans le modèle **conception**, lignes 5–15. Puis, nous construisons un problème de satisfaction de contraintes du modèle du motif à identifier et du modèle du programme dans lequel l'identifier, ligne 16.

La résolution réalisée ligne 17 peut utiliser indifféremment l'algorithme de résolution générique, section 4.1 page 163, ou de résolution automatique, section 4.4 page 172. La gestion des contradictions peut se faire indifféremment manuellement, section 4.3 page 171, ou par relaxations automatiques, section 4.7 page 175.

Pour chaque solution, ligne 18–20, nous construisons la micro-architecture similaire au motif de conception identifié, lignes 21–27, que nous ajoutons au modèle du programme au niveau **conception**, ligne 28.

Nous distinguons une micro-architecture similaire à un motif de conception du modèle concret de ce motif. Conceptuellement, le modèle concret du motif de conception *est* une variante du motif de conception adaptée au contexte, la micro-architecture similaire au motif de conception représente *possiblement* une variante du motif de conception, c'est au mainteneur de décider si la micro-architecture identifiée est réellement une variante du motif de conception.

Conclusion. Nous avons présenté un algorithme pour créer le modèle d'un programme au niveau **conception** avec son modèle au niveau **idiomatique** et les algorithmes d'identification des motifs de conception basés sur la programmation par contraintes avec explications. Cet algorithme garantit la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception** car les constituants représentant les motifs au niveau **conception** sont liés aux constituants du niveau **idiomatique** qu'ils abstraient. Nous vérifions maintenant la cohérence des modèles des motifs de conception et des algorithmes d'identification.

Algorithme 4.8 – Construction du modèle du programme au niveau conception.

```

1  construire-modèle-conception(modèle-idiomatique, modèle-motif-conception)
2    ∈ Méta-modèle-Conception
3  Début
4    modèle-conception ← nouveau(Modèle-Conception)
5    entités ← énumération-entités(modèle-idiomatique)
6    TantQue suivante-existe(entités) alors
7      entité ← suivante(entités)
8      nouvelle-entité ← nouvelle(Entité)
9      modifier-nom(nouvelle-entité, obtenir-nom(entité))
10     // Modification des autres attributs de la nouvelle entité
11     // en fonction des attributs de l'entité correspondante dans
12     // le modèle de l'architecture au niveau idiomatique.
13     ...
14     ajouter-entité(modèle-conception, nouvelle-entité)
15  FinTantQue
16  problème ← construire-problème(modèle-motif-conception, modèle-idiomatique)
17  Si résout(problème) alors
18    solutions ← énumération-solutions(problème)
19    TantQue suivante-existe(solutions) alors
20      solution ← suivante(solutions)
21      micro-architecture ← nouveau(Micro-Architecture)
22      entités-participantes ← énumération-entités(solutions)
23      TantQue suivante-existe(entités-participantes) alors
24        entité ← suivante(entités-participantes)
25        rôle ← obtenir-rôle(solution, entité)
26        ajouter-entité(micro-architecture, entité, rôle)
27      FinTantQue
28      ajouter-motif(modèle-conception, micro-architecture)
29    FinTantQue
30  FinSi
31  Retourne modèle-conception
32 Fin

```

□

4.7 Discussion des modèles, des contraintes et des algorithmes

Nous vérifions la cohérence de nos modèles des motifs de conception et de nos algorithmes d'identification en deux phases. D'abord, nous discutons nos métamodèles par rapport à d'autres techniques de modélisation et à d'autres métamodèles.

Ensuite, nous comparons nos algorithmes à d'autres techniques d'identification des motifs de conception. Contrairement à la validation de la traçabilité des motifs interclasses, section 3.7 page 126, la validation des algorithmes est difficile et limitée.

4.7.1 Modèles

Les métamodèles pour décrire les modèles de programmes au niveau **conception** et les motifs de conception aux niveaux **idiomatique** et **conception** sont à comparer avec d'autres techniques de description et d'autres métamodèles.

Exemples d'autres techniques de description Dirk Heuzeroth *et al.* [2002] modélisent les programmes comme des arbres de syntaxe abstraite dans lesquels ils recherchent des motifs d'interactions pour abstraire les programmes comme des composants communiquant par des ports d'entrées-sorties.

Les motifs d'interactions recherchés sont décrits par des algorithmes dédiés qui analysent les arbres de syntaxe abstraite. Ces motifs d'interactions sont similaires à des motifs de conception comportementaux.

Le processus de rétroconception proposé par Jochen Seemann et Jürgen Wolf von Gudenberg [1998], comme présenté section 2.3 page 60, peut aussi être utilisé pour identifier des motifs de conception.

Le code source d'un programme est analysé pour construire un graphe $S = (V, E, \phi, \eta)$. Le graphe V est transformé avec une grammaire de graphe pour abstraire des relations d'association et d'agrégation puis des motifs de conception.

Les motifs de conception sont inclus dans le graphe comme de nouveaux nœuds. Le graphe contient alors des informations similaires aux informations représentées par le modèle du programme issu de notre métamodèle mais plus limitées car seules les formes complètes des motifs de conception sont identifiées et la distinction entre les niveaux **idiomatique** et **conception** n'est pas assurée.

Kamran Sartipi *et al.* [Sartipi *et al.*, 2000] présentent une technique de modularisation de programmes non objets dirigée par les mainteneurs. La modularisation d'un programme consiste à définir une requête représentant l'architecture supposée du programme et à appliquer cette requête sur le programme pour y identifier les constituants la satisfaisant.

L'identification des constituants du programme satisfaisant une requête est réalisée par un algorithme de séparation et d'évaluation progressive⁷ modifié pour calculer une valeur de confiance avec chaque solution. Cette valeur de confiance représente la similitude entre les constituants identifiés et l'architecture supposée.

Nicole Lévy et Francis Losavio [1998] modélisent le comportement des motifs de conception avec le langage de spécifications formelles pour systèmes distribués LOTOS. Les auteurs prouvent l'existence d'une relation de spécialisation entre deux motifs de conception, Médiateur et Courtier : le motif de conception Courtier spécialise le motif de conception Médiateur. L'utilisation d'un langage de spécifications formelles comme LOTOS est intéressante pour modéliser le comportement des motifs.

Anthony Lauder et Stuart Kent [1998] utilisent les diagrammes de contraintes et définissent trois niveaux pour visuellement et précisément définir des modèles de motifs de conception. Le modèle d'un motif de conception est décomposé en un modèle de rôles, un modèle de types et un modèle de classes. Un rôle est une abstraction de types, un type est une abstraction de classes.

Des diagrammes de contraintes permettent de décrire précisément et visuellement les interactions entre les constituants de chaque niveau et entre les niveaux. Ils permettent de capturer l'*essence* d'un motif de conception en retirant du modèle du motif les informations superflues à chaque niveau.

Le métamodèle utilisé pour décrire chaque niveau est similaire au métamodèle UML augmenté avec des diagrammes de contraintes. Ces travaux semblent une alternative intéressante à notre métamodèle pour décrire les motifs de conception même si elle est plus complexe.

La représentation de motifs de conception par des graphes de notions de conception⁸ [Baniassad *et al.*, 2002] permet d'identifier les notions-clés définissant les motifs de conception.

Cependant, cette représentation par graphes de notions de conception est réalisée manuellement et son utilisation pour l'identification des motifs de conception, par exemple par la génération de systèmes de contraintes, n'a pas encore été étudiée. Elle est néanmoins séduisante car elle facilite la compréhension des motifs de conception et de leur implémentation.

Jan Bosch [1998] introduit au niveau du langage un modèle à objets en couches pour modéliser les motifs de conception. Un objet dans ce modèle est constitué de variables, de méthodes, d'états, de catégories et de couches.

⁷Un algorithme de séparation et d'évaluation progressive s'appelle *branch and bound* en anglais [Office québécois de la langue française, 2003].

⁸Les graphes de notions de conception s'appellent *design rationale graphs* en anglais.

Une couche intercepte les envois de messages de et vers l'objet qu'elle entoure. Elle représente une notion abstraite comme une relation entre objets ou un motif de conception. L'utilisation de ce modèle à objets en couches nécessite l'utilisation d'un langage de programmation non standard pour appliquer et identifier les motifs de conception.

Amnon Eden *et al.* [1997] modélisent les motifs de conception par un ensemble de transformation de l'arbre de syntaxe abstraite d'un programme pour y appliquer le motif modélisé. Les transformations sont décrites en SMALLTALK et agissent sur l'arbre de syntaxe abstraite de programmes EIFFEL. Ces travaux sont spécifiques à l'application des motifs de conception et ne permettent de les identifier et d'en garantir la traçabilité.

Ces différentes techniques sont similaires à notre utilisation de la métamodélisation, comme observé dans la section 3.7 page 126 : elles définissent toutes des constituants dont les instances forment les modèles des programmes et des motifs.

Aussi, nous pensons qu'il n'y a pas là de *meilleure* technique, seulement des techniques adaptées à l'utilisation qui en est faite. La métamodélisation est une technique adéquate pour permettre l'identification des motifs de conception et pour en garantir la traçabilité entre les niveaux *idiomatique* et *conception*.

Exemples d'autres métamodèles Hervé Albin-Amiot [2003, page 24] propose un état de l'art sur la *métareprésentation* [Borne et Revault, 1999] des motifs de conception, par la logique d'ordre supérieur [Eden, 2000], par la logique temporelle [Mikkonen, 1998] ou encore par la métamodélisation [Rapicault et Fornarino, 2000 ; Sunyé *et al.*, 2000].

Cet état de l'art montre que chaque métamodèle de la littérature pour décrire les motifs de conception a été défini avec un objectif précis : représentation de la solution, de l'intention du motif, validation, classification, etc.

Nous n'avons pas utilisé l'un ou l'autre de ces métamodèles pour garder notre solution à la traçabilité des motifs de conception la plus simple possible. D'une part, l'utilisation d'un métamodèle peu adapté à nos objectifs aurait compromis ceux-ci. D'autre part, la métamodélisation est pour nous un outil, comme la programmation par contraintes avec explications, choisi uniquement pour son adéquation avec nos objectifs.

Discussion Les métamodèles présentés sections 4.1 et 4.2 pages 145 et 148 sont adéquats pour modéliser l'architecture globale d'un programme au niveau *conception*, les motifs de conception aux niveaux *idiomatique* et *conception* et pour garantir la traçabilité des motifs interclasses entre les niveaux *implémentation* et *idiomatique*.

Ils proposent une alternative à des techniques et des métamodèles plus généraux ou ne satisfaisant pas nos besoins. Cependant, ils ont des limites, en particulier ils ne nous permettent pas de modéliser les motifs de conception comportementaux et créateurs. La modélisation de ces motifs nécessite de nouveaux métamodèles ou l'extension des métamodèles proposés.

4.7.2 Algorithmes

La validation des algorithmes doit être réalisée suivant trois critères :

1. L'identification et la traçabilité sur des programmes simples dans lesquels un unique motif de conception connu est utilisé.
2. L'adéquation des résultats de l'identification des motifs de conception avec les motifs de conception effectivement présents dans un ensemble de programmes développés indépendamment de nos recherches.
3. La comparaison des résultats avec d'autres outils d'identification des motifs de conception.

Programmes simples Nous avons développé un ensemble de programmes simples comme jeu de tests à nos algorithmes. Cet ensemble de programmes contient dix-huit programmes de deux à quinze entités.

Un programme de cet ensemble utilise un unique motif de conception, par exemple le motif de conception **Composite**, dont l'utilisation peut être une forme complète ou une forme approchée.

Avec ces programmes, la connaissance des motifs de conception utilisés et des formes présentes, nous avons développé une suite de cent soixante-dix-huit tests unitaires pour tester l'identification des formes complètes et approchées des motifs de conception.

Cette suite de tests unitaires nous a permis d'évaluer l'adéquation des résultats de nos algorithmes avec les formes complètes et approchées existant réellement dans les programmes. De plus, elle nous garantit la non-régression de nos algorithmes lors de leur maintenance.

Programmes indépendants Nous avons eu des difficultés à trouver des programmes développés indépendamment de nos recherches dans lesquels les motifs de conception utilisés sont documentés.

Un des rares exemples de programmes dont les motifs de conception sont documentés est le programme JHOTDRAW. C'est une des raisons pour lesquelles nous avons choisi ce programme pour scénario fil conducteur.

Les motifs de conception de [Gamma *et al.*, 1994] suivants sont documentés dans le code source du programme JHOTDRAW :

- Adaptateur ;
- Chaîne de responsabilités ;
- Commande ;
- Composite ;
- Décorateur ;
- État ;
- Itérateur ;
- Médiateur ;

- Méthode gabarit⁹ ;
- Méthode usine ;
- Observateur ;
- Poids-plume ;
- Pont ;
- Prototype ;
- Singleton ;
- Usine abstraite.

Nous avons modélisé les motifs de conception Composite, Décorateur, Itérateur, Méthode usine et Observateur. Nous avons alors cherché à identifier ces motifs de conception dans le programme JHOTDRAW. Le tableau 4.2 présente les résultats de l'identification de ces motifs de conception.

D'une part, ce tableau montre que peu de motifs de conception sont utilisés dans le programme JHOTDRAW même si l'objectif de ce programme était de mettre en œuvre des motifs. Ainsi, la présence de motifs de conception dans des programmes quelconques pourrait sembler utopique.

Pourtant, nous pensons comme Harald C. Gall *et al.* [1996] que les développeurs utilisent toujours des motifs, même implicitement¹⁰ ; les mainteneurs profiteraient donc de modèles et d'algorithmes pour les identifier.

Tableau 4.2 – Résultats de l'identification de motifs de conception dans le programme JHOTDRAW.

Motifs de conception	Existants	Identifiés	Erreurs	Manqués	Approchés	Sec.
Composite	1	1	0	0	3	65,5
Décorateur	1	1	0	0	7	29,9
Itérateur	3	0	3	100	0	231,1
Méthode usine	2	2	0	37	1	103,7
Observateur	2	2	2	2	0	61,4
Total	9	6	5	139	11	98,32

□

⁹Le motif de conception Méthode gabarit s'appelle *Template method* en anglais [Office québécois de la langue française, 2003].

¹⁰Dans leur article [1996], Harald C. Gall *et al.* remarquent que *"It is fair to assume, that original developers—even if not explicitly—had some kind of patterns or stereotypical solutions in mind when developing the product in the first place."*

D'autre part, ce tableau souligne les limites de nos métamodèles et de nos algorithmes : la représentation des motifs de conception non structuraux est très limitée, comme les motifs **Itérateur** et **Observateur** et nos algorithmes fournissent un grand nombre de formes approchées, soit réelles soit erronées avec un temps de calcul relativement long.

Nous devons donc étendre nos métamodèles ou développer de nouveaux métamodèles pour modéliser les motifs de conception non structuraux et améliorer nos algorithmes et nos stratégies de recherche pour diminuer le nombre de formes approchées erronées et pour augmenter leur efficacité.

Comparaisons avec d'autres outils Nous avons cherché à comparer nos algorithmes avec d'autres outils d'identification des motifs de conception. Cependant, nous n'avons pas trouvé d'autres outils académiques ou industriels qui cherchent à identifier les formes complètes et les formes approchées de motifs de conception, comme présenté dans la section 2.2 page 37.

Aussi, l'évaluation de nos résultats doit être manuelle. Nous aimerions réaliser une évaluation de nos algorithmes indépendamment de nos recherches pour établir le bien-fondé, ou non, de l'identification des motifs de conception pour faciliter la compréhension et la maintenance des programmes.

Aussi, cette évaluation permettrait de quantifier les degrés de dégradations tolérées pour chaque motif de conception et d'offrir des stratégies de recherche automatique plus pertinentes.

Conclusion. Nous avons cherché à valider nos modèles et nos algorithmes pour l'identification et la traçabilité des motifs de conception. Nos métamodèles sont adéquats pour modéliser les programmes au niveau **conception** et les motifs de conception aux niveaux **idiomatique** et **conception** car ils nous permettent d'identifier les motifs et de garantir leur traçabilité avec des modèles uniques et de première classe des motifs. Nos algorithmes sont difficiles à évaluer formellement car aucun autre outil réalisant la traçabilité des formes complètes et approchées existe à notre connaissance. Cependant, nos algorithmes semblent adéquats car ils répondent à nos besoins. Nous illustrons maintenant l'application de nos modèles et de nos algorithmes pour garantir la traçabilité du motif de conception **Composite** dans les modèles du programme JHOTDRAW entre les niveaux **idiomatique** et **conception**.

4.8 Application à JHOTDRAW

Nous mettons en œuvre les modèles et les algorithmes proposés pour l'identification et la traçabilité des motifs de conception. Nous utilisons le modèle du programme JHOTDRAW au niveau **idiomatique** et les algorithmes présentés pour construire semi-automatiquement un modèle du programme au niveau **conception** par l'identification des micro-architectures similaires au motif de conception **Composite**.

Le modèle obtenu ressemble essentiellement au modèle du programme décrit par le diagramme de classes fourni avec sa documentation augmenté du motif de conception **Composite**. Le détail des implantations est présenté dans la partie III page 195.

La figure 4.5 page 185 montre le modèle du programme JHOTDRAW au niveau **idiomatique** obtenu automatiquement dans le chapitre précédent. Les figures 4.6(a) et 4.6(b) page 186 montrent le diagramme de classes du motif de conception **Composite**, tel que proposé dans [Gamma *et al.*, 1994], et son modèle au niveau **idiomatique**.

Nous appliquons nos algorithmes à ces modèles pour obtenir un problème de satisfaction de contraintes représentant la recherche des micro-architectures similaires au motif de conception **Composite**.

Nous résolvons ce problème avec l'algorithme de résolution interactive, comme montré sur la figure 4.7 page 186 et nous construisons un modèle du programme au niveau **conception** incluant les micro-architectures identifiées.

Ce modèle représente JHOTDRAW, ses classes et ses interfaces, les relations et les motifs interclasses, et les micro-architectures similaires au motif de conception **Composite** au niveau **conception**, comme montré sur la figure 4.8 page 187.

Il présente essentiellement les mêmes informations que le modèle du programme décrit par le diagramme de classes fourni avec sa documentation et augmenté du motif de conception **Composite**, figure 4.9 page 188, comme montré sur la figure 4.10 page 189.

En particulier, la micro-architecture similaire au motif de conception **Composite** mise en valeur est identique au motif de conception représenté sur le diagramme de classes de JHOTDRAW fourni par les auteurs du programme.

Nous avons aussi validé manuellement les autres micro-architectures identifiées et nous trouvons qu'elles sont correctes par rapport à nos définitions et qu'elles précisent le diagramme de classes de JHOTDRAW.

La traçabilité est garantie entre les modèles de JHOTDRAW aux niveaux **implémentation** et **idiomatique** : par exemple, les constituants du modèle du programme qui participe à la micro-architecture similaire au motif de conception **Composite** sont mis en valeur lorsque cette micro-architecture est sélectionnée.

Conclusion. Nous avons apporté une solution au problème de la traçabilité des motifs de conception entre les niveaux implémentation et conception tel que décrit dans la section 1.3 page 17 et représenté sur la figure 1.7 page 25 : nous avons garanti la traçabilité des motifs de conception entre implantation et conception. Nous concluons maintenant sur la traçabilité des motifs de conception et présentons un bilan général de la traçabilité entre les niveaux implémentation, idiomatique, et conception.

FIG. 4.5 – Modèle de JHOTDRAW au niveau idiomatique obtenu automatiquement.

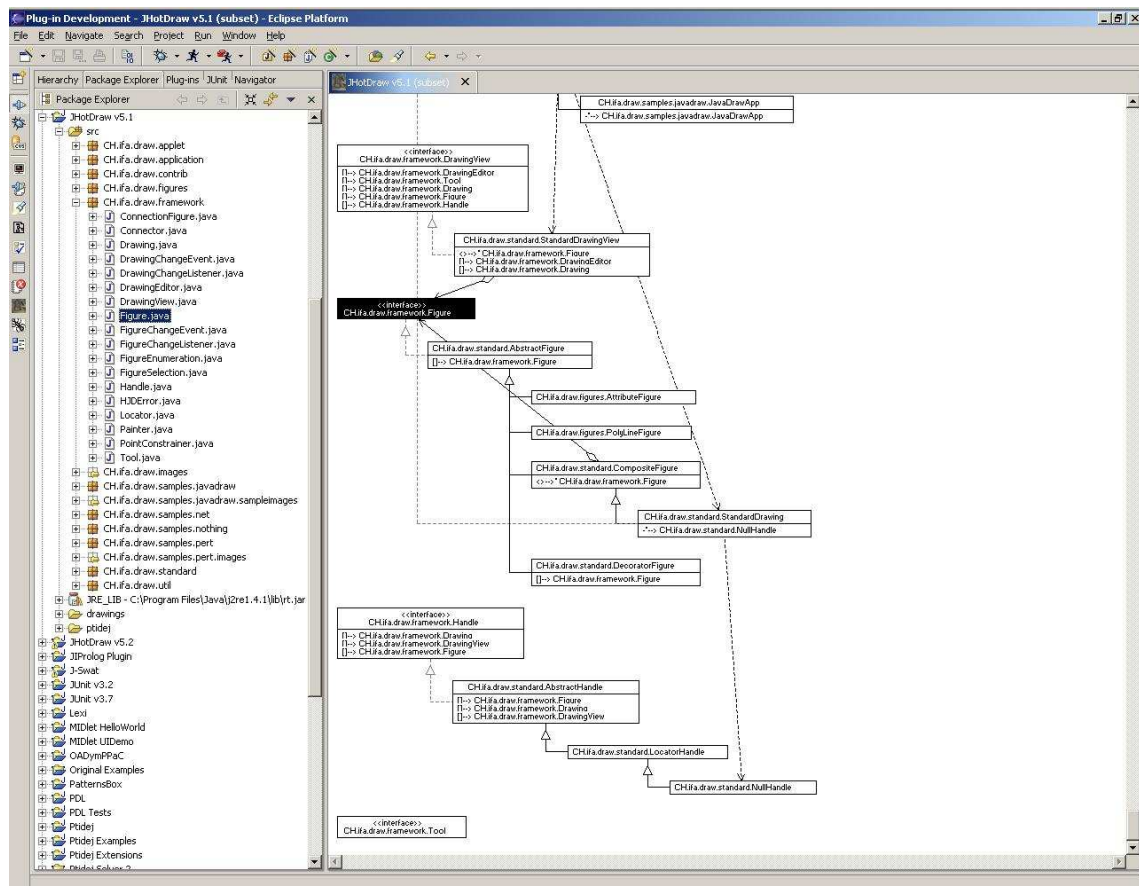
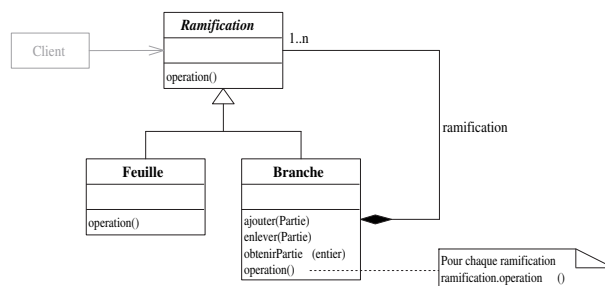
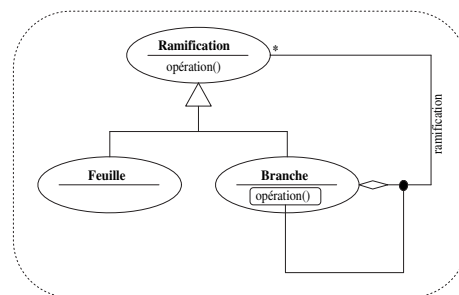


FIG. 4.6 – Motif de conception Composite.

(a) Motif de conception Composite :



(b) Modèle du motif de conception Composite au niveau idiomatique :



□

FIG. 4.7 – Interactions avec le solveur de contraintes pour identifier les micro-architectures similaires au modèle du motif de conception Composite.

```

Commandes - compiledrun
Choco comes with ABSOLUTELY NO WARRANTY; for details read licence.txt
This is free software, and you are welcome to redistribute it
under certain conditions; read licence.txt for details.
Iceberg version 0.8, Copyright (C) 2000-02 Bouygues e-lab
** PaLM : Constraint Programming with Explanations
** PaLM v1.324 (Dec 1, 2002), Copyright (c) 2000-2002 N. Jussien
++ Ptidej: Pattern Trace Identification Detection and Enhancement for Java
++ Constraint Programming for Design Patterns and Design Defects Detection
++ Ptidej Solver v0.324 (May 8, 2003), Copyright (c) 2001-2003 Y.-G. Guéhéneuc

Loading domain file
Clearing result file
Calling solver
There is no more solution because of the constraint:
1. "[#]->" "Composite <-- Component"
   (weight 90)
   To be replaced with: "[ ]-->"
Which one do you want to relax? (0 -> none) 1

Solution 1:
  Leaf = CH.ifa.draw.figures.AttributeFigure
  Composite = CH.ifa.draw.standard.AbstractFigure
  Component = CH.ifa.draw.framework.Figure

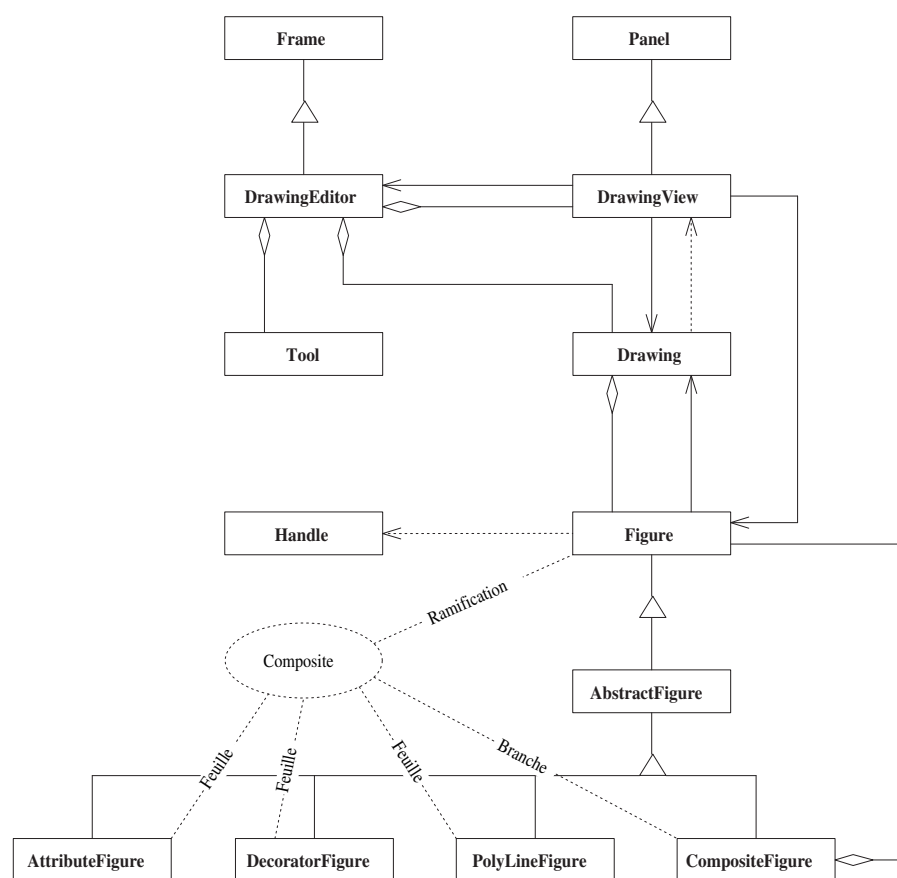
Do you want another solution? (y/n)
  
```

□

FIG. 4.8 – Modèle de JHOTDRAW au niveau **conception** obtenu automatiquement avec une micro-architecture similaire au motif de conception **Composite** mise en valeur.



FIG. 4.9 – Modèle de JHOTDRAW au niveau conception fourni avec la documentation du programme.



□



Bilan

Nous avons étudié la traçabilité des motifs de conception. D'abord, nous avons proposé des modèles des programmes aux niveaux **idiomatique** et **conception** et des motifs de conception au niveau **idiomatique**. Programmes et motifs sont décrits par des métamodèles similaires mais conceptuellement différents.

Ensuite, nous avons décrit un processus de modélisation des motifs de conception en quatre phases pour construire les modèles abstraits des motifs, qui représentent les *leitmotive* des motifs, la synthèse des rubriques structure, participants et collaborations.

Puis, nous avons montré que le modèle abstrait d'un motif de conception peut se traduire en un système de contraintes et que la recherche des micro-architectures similaires au motif de conception se traduit en un problème de satisfaction de contraintes.

Nous avons choisi la programmation par contraintes avec explications pour résoudre le problème de satisfaction de contraintes traduisant l'identification des motifs car elle permet d'expliquer les micro-architectures identifiées et l'interaction avec les mainteneurs.

Nous avons développé une bibliothèque de contraintes qui représentent les relations interclasses, comme l'appel de méthodes, l'héritage, l'instanciation, l'association, l'agrégation et la composition et établi un ordre partiel entre ces relations.

Nous avons présenté deux stratégies de recherche pour l'identification des micro-architectures : une stratégie de recherche interactive où l'identification est guidée par les mainteneurs, une stratégie de recherche automatique où les contraintes et le problème sont relaxés combinatoirement.

Nous avons validé nos modèles et nos algorithmes sur des programmes de tests et sur JHOTDRAW, puis nous les avons illustrés avec le programme JHOTDRAW et le motif de conception **Composite** : le modèle au niveau **conception** obtenu est similaire au modèle fourni avec la documentation de ce programme.

Conclusion. Avec ces travaux sur la traçabilité des motifs de conception, nous pouvons réaliser les quatre phases de la traçabilité des motifs interclasses décrites dans la section 1.2 page 14 et construire semi-automatiquement le modèle d'un programme au niveau **conception** de son modèle au niveau **idiomatique** et des modèles de motifs de conception. Nous dressons maintenant un bilan général de nos travaux sur la traçabilité des motifs entre les niveaux **implémentation**, **idiomatique** et **conception**.

Bilan de la traçabilité des motifs

NOUS avons étudié le problème de l'identification et de la traçabilité des motifs de conception dans des programmes JAVA entre les trois niveaux d'abstraction **implémentation**, **idiomatique** et **conception**.

Nous avons décomposé ce problème en deux sous-problèmes traitant de l'identification et de la traçabilité, respectivement, des motifs interclasses, entre les niveaux **implémentation** et **idiomatique**, et des motifs de conception, entre les niveaux **idiomatique** et **conception**.

Les motifs interclasses représentent les relations d'association, d'agrégation, et de composition. Ils existent au niveau **idiomatique** mais sont disséminés dans l'architecture du programme au niveau **implémentation**.

Nous avons proposé trois formalismes pour décrire :

- des programmes JAVA au niveau **implémentation** avec des modèles statiques et dynamiques, respectivement les séquences d'instructions du code source et des traces d'exécutions des programmes ;
- les modèles globaux des programmes comme des instances particulières d'un méta-modèle dans lesquelles les relations interclasses sont explicites.

Nous avons défini les motifs interclasses aux niveaux **implémentation** et **idiomatique** et raffiné leurs définitions au niveau **implémentation** avec quatre propriétés minimales : durée de vie, exclusivité, multiplicité et site d'invocation.

Nous avons illustré ces propriétés au niveau **implémentation** avec les formalismes proposés et décrit des algorithmes d'analyses statiques et dynamiques pour identifier les motifs interclasses avec les modèles statiques et dynamiques.

Aussi, nous avons décrit des algorithmes pour construire les modèles des programmes au niveau **idiomatique** en garantissant la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique**.

Nous avons vérifié la cohérence de ces modèles, de ces définitions et de ces algorithmes, puis montré leur utilisation et illustré leur mise en œuvre complète avec le programme JHOTDRAW.

L'identification et la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception** doivent prendre en compte les formes approchées des micro-architectures similaires à ces motifs et permettre l'interaction avec les mainteneurs.

Nous avons réutilisé le métamodèle défini précédemment pour décrire les modèles des programmes au niveau **idiomatique** et proposé un métamodèle dédié à la description des motifs de conception comme des entités de première classe uniques.

Nous avons présenté un nouveau métamodèle pour décrire les modèles des programmes au niveau **conception**, avec lequel les micro-architectures similaires à des motifs de conception sont explicites.

Nous avons montré que l'identification des micro-architectures similaires à un motif de conception, formes complètes et approchées, se traduit en un problème de satisfaction de contraintes dans lequel les variables représentent les participants définis par le motif, les contraintes représentent les relations entre ces participants et le modèle d'un programme représente le domaine des variables.

Nous avons montré qu'un tel problème se déduit des modèles du motif et du programme au niveau **idiomatique** et nous avons utilisé la programmation par contraintes avec explications pour le résoudre.

La programmation par contraintes avec explications nous permet d'expliquer les micro-architectures identifiées et de diriger la recherche de ces micro-architectures en sélectionnant les caractéristiques du motif recherché par relaxation des contraintes et du problème.

Aussi, nous avons décrit un algorithme pour construire les modèles des programmes au niveau **conception** en garantissant la traçabilité des micro-architectures identifiées entre les niveaux **idiomatique** et **conception**.

Nous avons vérifié la cohérence de nos modèles et de nos algorithmes avec le programme JHOTDRAW et illustré leur utilisation en construisant un modèle de ce programme et des micro-architectures similaires au motif de conception **Composite** au niveau **conception**.

Selon le cadre de classification des techniques de rétroconception présenté section 2.1 page 32, nos modèles et nos algorithmes traitant des motifs interclasses sont des techniques informelles basées sur des analyses syntaxiques. Les modèles des programmes JAVA obtenus au niveau **idiomatique** ont :

- une distance sémantique de un, par définition ;
- une précision sémantique élevée car les techniques d'analyses syntaxiques garantissent la précision des modèles obtenus ;
- un niveau de détails sémantiques moyen car ils sont représentés par un métamodèle offrant plus de détails que le code source mais ne supportant pas l'utilisation de preuves formelles ;
- une traçabilité sémantique élevée car nos modèles et nos définitions des motifs interclasses fournissent toutes les informations nécessaires à la création un programme équivalent au programme analysé.

Ainsi, la qualité sémantique des résultats de nos modèles, de nos définitions et de nos algorithmes est supérieure aux qualités sémantiques des techniques existantes de rétroconception des motifs interclasses présentées dans l'état de l'art.

Nos modèles et nos algorithmes pour garantir la traçabilité des motifs de conception sont des techniques informelles basées sur l'identification de clichés. Les modèles des programmes obtenus au niveau **conception** ont :

- une distance sémantique de deux, par définition ;
- une précision sémantique faible car l'identification des motifs dépend des modèles des motifs et de leur similarité avec les micro-architectures réellement implantées. Cependant, l'identification est guidée par les mainteneurs, elle est donc précise par rapport à leurs besoins ;
- un niveau de détails sémantiques moyen car ils sont représentés par un métamodèle offrant plus de détails que le code source mais ne permettant pas la réalisation de preuves formelles ;
- une traçabilité sémantique élevée car nos modèles fournissent les informations nécessaires pour créer un programme équivalent au programme analysé, dont les choix de conception.

Ainsi, la qualité sémantique des résultats de nos modèles et de nos algorithmes est supérieure aux qualités sémantiques des techniques existantes de rétroconception des motifs de conception.

Conclusion. Nos modèles et nos algorithmes combinent les bénéfices des techniques informelles de rétroconception : analyses syntaxiques et identification de clichés. Ils permettent d'abstraire le code source de programmes et facilitent la compréhension des programmes par la distance sémantique et la précision sémantique de leur résultat. Ils sont une solution aux problèmes de l'identification et de la traçabilité des motifs interclasses et de conception entre les niveaux **implémentation**, **idiomatique** et **conception**. Nous détaillons maintenant l'implantation de ces modèles et de ces algorithmes.

Troisième partie

Mise en œuvre de la traçabilité des motifs avec PTIDEJ

Nous détaillons une implantation des métamodèles et des algorithmes pour garantir la traçabilité des motifs interclasses et de conception entre les niveaux implémentation, idiomatique, et conception. Cette implantation définit un ensemble d'outils réunis dans la suite PTIDEJ¹¹. PTIDEJ est aussi l'outil principal pour l'identification et la traçabilité des motifs interclasses et de conception dans des programmes à objets.

D'abord, nous présentons l'implantation des métamodèles pour décrire les niveaux idiomatique et conception et les motifs de conception. Ces métamodèles sont similaires car ils ont un large ensemble de constituants en commun. Aussi, nous proposons un unique métamodèle, PADL, pour décrire les niveaux idiomatique et conception et les motifs de conception. Le métamodèle PADL est une extension du métamodèle PDL pour décrire les motifs de conception. Le métamodèle PDL propose un sous-ensemble des constituants nécessaires à la description des niveaux idiomatique et conception. Nous décrivons aussi brièvement l'implantation d'une bibliothèque graphique, PTIDEJ UI, pour représenter visuellement les modèles issus du métamodèle PADL.

Puis, nous présentons l'implantation des algorithmes d'identification des motifs interclasses. Les algorithmes d'analyses statiques sont définis dans l'outil INTROSPECTOR. Les algorithmes d'analyses dynamiques sont basés sur l'outil générique d'analyses des traces d'exécution de programmes, CAFFEINE. Nous détaillons les implantations des algorithmes de traçabilité et donnons un exemple d'utilisation avec JHOTDRAW. Ces implantations permettent de réaliser la première phase de l'identification des motifs de conception : l'obtention automatique du modèle d'un programme au niveau idiomatique.

Ensuite, nous présentons l'implantation de référence de la programmation par contraintes avec explications, PALM. Nous détaillons notre extension de PALM dédiée à l'identification des micro-architectures similaires à des motifs de conception dans des modèles de programmes au niveau idiomatique, PTIDEJ SOLVER. Nous proposons une bibliothèque de contraintes associées aux relations entre les constituants d'un modèle au niveau idiomatique, PTIDEJ LIBRARY. Nous décrivons la mise en œuvre des différentes stratégies de recherche.

¹¹PTIDEJ est l'acronyme anglais de *Pattern Trace Identification, Detection, and Enhancement in JAVA* : identification (de donner une identité, modéliser), détection et amélioration de motifs en JAVA.

Alors, nous présentons la modélisation d'un motif de conception avec PADL et les algorithmes de génération d'un problème de satisfaction de contraintes avec ce modèle d'un motif et le modèle d'un programme au niveau **idiomatique** dans l'outil PTIDEJ. Nous décrivons la mise en œuvre de PTIDEJ SOLVER pour obtenir les micro-architectures similaires au motif de conception et des algorithmes de traçabilité pour construire un modèle du programme au niveau **conception**. Nous donnons un exemple d'application sur JHOT-DRAW. Ces implantations permettent de réaliser la seconde phase de l'identification des motifs de conception : l'obtention automatique d'un modèle du programme au niveau **conception**.

Enfin, nous dressons un bilan de nos implantations avant de conclure sur les travaux présentés.

Chapitre 5

Métamodélisation

NOUS présentons l'implantation des trois métamodèles pour décrire les modèles d'un programme aux niveaux **idiomatique** et **conception** et les motifs de conception au niveau **idiomatique**.

Ces métamodèles sont implantés avec un unique métamodèle, PADL, extension du métamodèle PDL développé lors d'une précédente thèse de doctorat pour décrire les motifs de conception.

Aussi, nous décrivons brièvement une bibliothèque graphique, PTIDEJ UI, qui nous permet de visualiser les modèles issus du métamodèle PADL dans l'EDI ECLIPSE pour en faciliter la compréhension et l'utilisation.

5.1 Présentation du métamodèle PDL

NOUS présentons le métamodèle PDL¹ pour décrire les motifs de conception tel que proposé dans [Albin-Amiot, 2003]. PDL définit un ensemble minimal de constituants² pour décrire les motifs de conception.

Ce métamodèle a été défini dans l'intention d'utiliser les modèles de motifs de conception qui en sont issus pour synthétiser le code et détecter les formes complètes des motifs. Il représente un bon compromis entre la modélisation des motifs et la synthèse de code et la détection de ses constituants dans du code source JAVA.

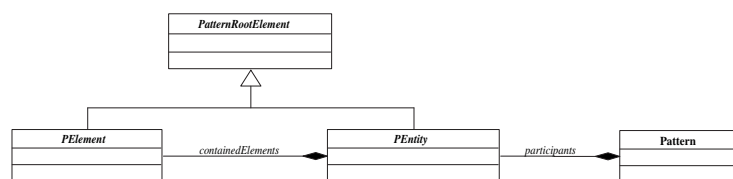
PDL est construit sur le principe que tout motif de conception est constitué de *participants* ayant des *éléments*, comme défini dans [Gamma *et al.*, 1994]³, [Albin-Amiot, 2003, page 76]. Un motif de conception est donc, en général, constitué d'entités (les participants) composées d'éléments (leurs attributs).

Le noyau de PDL reflète ce principe en définissant trois constituants **Pattern**, **PElement** et **PEntity**, correspondant respectivement au modèle d'un motif, aux participants du modèle et à leurs attributs, comme montré sur la figure 5.1.

Les constituants **Pattern**, **PElement** et **PEntity** définissent les services communs à tous les modèles, à tous les participants et à tous leurs attributs. Les services communs sont, par exemple, le choix d'un rôle, d'un nom, l'attribution d'un commentaire, les mécanismes de notification, les mécanismes de clonage et la transformation en d'autres modèles, par l'utilisation du patron de conception *Visiteur*, [Albin-Amiot, 2003, page 79].

Le noyau de PDL est insuffisant pour décrire un modèle de motif de conception, [Albin-Amiot, 2003, page 90]. Il doit être raffiné pour ajouter les constituants nécessaires à la modélisation des motifs de conception. Il est étendu pour ajouter des constituants correspondant aux notions :

FIG. 5.1 – Noyau du métamodèle PDL.



□

¹PDL est l'acronyme anglais de *Pattern Description Language* : langage de description des motifs.

²Dans son mémoire de thèse de doctorat, Hervé Albin-Amiot [2003] désigne un constituant du métamodèle PDL par le mot *méta-entité*. Un constituant du méta-modèle est une méta-entité car il est utilisé pour décrire des entités des modèles obtenus avec PDL, comme en SMALLTALK une métaclasse est utilisée pour décrire des classes. Cependant, nous conservons le mot *constituant* pour unifier notre vocabulaire.

³Dans leur livre, Erich Gamma *et al.* [1994] parlent des participants et de leurs *attributs* ; nous voulons distinguer les *attributs* et les *relations* des participants, aussi nous utilisons le mot *éléments* pour parler à la fois des attributs et des relations.

- de classe, constituant `PClass` ;
- d'interface, constituant `PInterface` ;
- de méthode, constituant `PMethod` ;
- de champ, constituant `PField` ;
- d'association, constituant `PAssociation` ;
- et de méthode délégante, constituant `PDelegatingMethod`.

Le raffinement du métamodèle utilise naturellement le mécanisme d'héritage pour spécialiser les constituants du noyau : `Pattern`, `PElement` et `PEntity`. La figure 5.2 page 200 montre les constituants du métamodèle PDL et leurs relations après raffinement.

PDL inclut des mécanismes de notification⁴ pour assurer la cohérence interne des modèles de motifs de conception, [Albin-Amiot, 2003, page 77]. Les mécanismes de notification sont utilisés, par exemple, pour interdire à un élément d'un modèle de prendre un nom déjà attribué à un autre élément⁵.

Aussi, PDL inclut un mécanisme de visite comme suggéré par le patron de conception `Visiteur`. L'interface `Walker` permet la définition de nouvelles opérations sur les instances des constituants du métamodèle sans modifier les constituants. Par exemple, le mécanisme de visite est utilisé pour calculer des métriques sur les modèles de programmes.

Enfin, PDL offre un ensemble de classes satellites utiles au fonctionnement et à l'utilisation du métamodèle, de ses constituants et des modèles de motifs de conception. En particulier, le métamodèle définit une classe `TypeRepository` qui offre un point d'accès unique aux constituants du métamodèle et une classe `PatternRepository` qui offre un point d'accès unique aux modèles de motifs de conception définis.

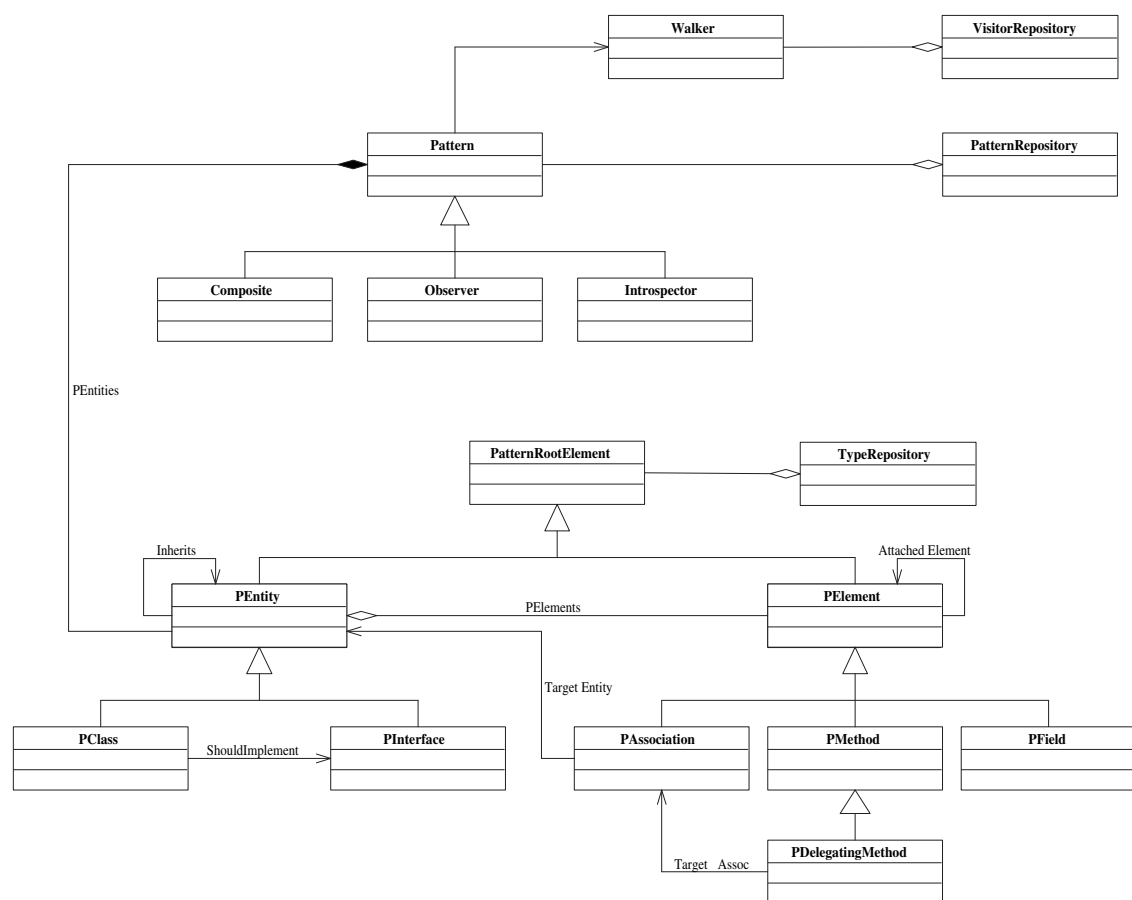
Le métamodèle PDL a été implanté en JAVA, il repose sur un ensemble de classes et d'interfaces JAVA représentant ses constituants. Le raffinement s'effectue par ajout de classes JAVA, spécialisant les classes `Pattern`, `PEntity` et `PElement`.

Conclusion. Le métamodèle PDL permet de décrire les motifs de conception et le motif interclasse `Association`. En l'état, il ne permet ni de décrire les niveaux `idiomatique` et `conception`, ni les motifs interclasses `Agrégation` et `Composition` ; mais, il est suffisamment flexible pour que nous le restructurions pour décrire les niveaux `idiomatique` et `conception` et pour que nous le raffinions pour modéliser tous les motifs interclasses.

⁴Les mécanismes de notification sont appelés *listeners* en anglais.

⁵L'interdiction est réalisée par le mécanisme appelé *veto-listeners* en anglais.

FIG. 5.2 – Sous-ensemble simplifié du métamodèle PDL tel que présenté dans [Albin-Amiot, 2003, page 100].



□

5.2 Extension du métamodèle avec PADL

Nous restructurons et raffinons le métamodèle PDL en un nouveau métamodèle, PADL⁶, pour décrire les programmes aux niveaux **idiomatique** et **conception** et les motifs de conception comme des entités uniques et de première classe.

D’abord, nous restructurons le métamodèle PDL pour mieux refléter nos métamodèles. Ensuite, nous ajoutons les constituants nécessaires à la modélisation des niveaux **idiomatique** et **conception**, en particulier les constituants correspondant aux motifs interclasses **Association**, **Agrégation** et **Composition** et les constituants correspondant aux niveaux **idiomatique** et **conception**. Enfin, nous discutons l’usage de ce métamodèle pour modéliser un programme aux niveaux **idiomatique** et **conception** et les motifs de conception.

5.2.1 Restructuration

Nous utilisons le métamodèle PDL et nous le restructurons pour refléter les constituants des métamodèles proposés dans les sections 3.1 page 79, 4.1 page 145, et 4.2 page 148.

Le métamodèle PADL obtenu après restructuration est présenté sur la figure 5.3 page 209. La restructuration consiste principalement à renommer les constituants de PDL et leurs opérations pour refléter nos trois métamodèles.

Nous ne faisons pas apparaître les constituants satellites, tels les classes **PatternRepository**, **Visitor**, **TypeRepository**, car ils surchargeraient inutilement la représentation du métamodèle.

Les tableaux 5.1 et 5.2 page 202 résument les opérations de renommage réalisées sur les constituants et les relations entre les constituants des métamodèles PDL et PADL.

Tableau 5.1 – Correspondance des relations entre les métamodèles PDL et PADL.

PDL	PADL
PEntities	containedEntities
Inherits	superEntities
PElements	containedElements
Attached Element	attachedElement
ShouldImplement	implementedInterfaces
Target Entity	targetEntity
TargetAssoc	supportAssociation

□

⁶PADL est l’acronyme anglais de *Pattern and Abstract-level Description Language* : langage de description des motifs et de niveaux d’abstraction.

5.2.2 Extension

Nous étendons le métamodèle PADL et modifions les relations entre ses constituants pour obtenir un métamodèle similaire à l'union des métamodèles pour décrire les niveaux **idiomatique**, **conception** et les motifs de conception. La figure 5.4 page 210 présente le métamodèle PADL étendu.

D'abord, nous modifions les constituants **Pattern** et **Method** pour introduire deux constituants **Role** et **Parameter** décrivant la notion de rôle dans les modèles des motifs de conception et la notion de paramètre de méthodes dans les modèles **idiomatique** et **conception**. Ces constituants nous permettent de caractériser plus finement les modèles issus du métamodèle PADL.

Ensuite, nous étendons la hiérarchie des relations interclasses pour refléter les motifs **Association**, **Agrégation** et **Composition**. Nous modifions aussi le constituant support du constituant décrivant une méthode délégante : avec nos définitions, une méthode délégante est supportée par une relation d'agrégation.

Puis, nous ajoutons un constituant **MicroArchitecture** pour décrire au niveau **conception** les micro-architectures similaires à des motifs de conception identifiées au niveau **idiomatique**. Une micro-architecture agrège des entités (relation **participants**) et les rôles correspondants dans le motif de conception représenté (relation **role**).

Alors, nous étendons la hiérarchie des constituants pour ajouter les constituants nécessaires pour décrire les niveaux **idiomatique** et **conception** : les constituants **IdiomLevelModel** et **DesignLevelModel**. Ces constituants sont composés d'entités (relation de composition **containedEntities**) et le constituant **DesignLevelModel** est composé de plus de micro-architectures (relation de composition **containedMicroArchitectures**).

Enfin, nous ajoutons un constituant **Ghost** pour décrire les constituants du modèle d'un programme référencés par ces constituants mais inconnus lors de l'analyse (et qui ne sont pas déjà présents dans le modèle).

Tableau 5.2 – Correspondance des constituants entre les métamodèles PDL et PADL.

PDL	PADL
PatternRootElement	Constituent
PEntity	Entity
PElement	Element
PClass	Class
PInterface	Interface
PAssociation	Association
PMethod	Method
PField	Field
PDelegatingMethod	DelegatingMethod

□

Par exemple, le constituant **Ghost** nous permet de décrire une entité cliente implicite [Gamma *et al.*, 1994, page 365] dans le modèle d'un motif de conception : nous pouvons représenter la classe `java.awt.Container` dans le modèle d'un programme sans avoir à représenter aussi toute la hiérarchie des composants graphiques de AWT dont elle dépend, aux niveaux **idiomatique** ou **conception**.

5.2.3 Utilisation

La description manuelle d'un sous-ensemble du modèle du programme JHOTDRAW au niveau **idiomatique** montré sur la figure 1.7(b) page 25 s'exprime alors par l'extrait du code source 5.1. Nous décrivons déclarativement le modèle du programme avec les constituants du métamodèle, par exemple ligne 2 le constituant **Interface** pour créer le constituant représentant l'interface **Figure**, et en utilisant la sémantique définie par leurs méthodes, par exemple ligne 7 la méthode `addImplementedEntity()` pour définir une relation d'implantation entre une interface et une classe.

Code source 5.1 – Description d'un sous-ensemble du modèle du programme JHOTDRAW au niveau **idiomatique**.

```

1 // Déclaration de l'interface Figure :
2 final Interface figure = new Interface("Figure");
3
4 // Déclaration de la classe abstraite AbstractFigure,
5 // qui implémente l'interface Figure :
6 final Class abstractFigure = new Class("AbstractFigure");
7 abstractFigure.addImplementedEntity(figure);
8 abstractFigure.setAbstract(true);
9
10 // Déclaration de la classe abstraite CompositeFigure,
11 // qui spécialise la classe abstraite AbstractFigure,
12 // et est en relation d'agrégation avec l'interface Figure :
13 final Class compositeFigure = new Class("CompositeFigure");
14 compositeFigure.addInheritedEntity(abstractFigure);
15 final Aggregation figureAggregation = new Aggregation("figures", figure, 2);
16 compositeFigure.addElement(figureAggregation);
17 ...
18
19 // Déclaration du modèle de JHOTDRAW au niveau implémentation :
20 final IdiomLevelModel jhotdrawModel =
21     new IdiomLevelModel("JHotDraw idiom-level model");
22 jhotdrawModel.addEntity(figure);
23 jhotdrawModel.addEntity(abstractFigure);
24 jhotdrawModel.addEntity(compositeFigure);
25 ...

```

□

Conclusion. Le métamodèle PADL reflète les métamodèles des niveaux **idiomatique** et **conception** et des motifs de conception proposés dans les sections 3.1 page 79, 4.1 page 145 et 4.2 page 148. Ce métamodèle est une extension du métamodèle PDL et nous permet de représenter les modèles de programmes aux niveaux **idiomatique** et **conception** et les motifs de conception au niveau **idiomatique** comme des entités uniques et de première classe. Nous présentons maintenant une bibliothèque graphique dédiée à la visualisation des modèles décrits avec ce métamodèle.

5.3 Visualisation des modèles avec PTIDEJ UI

LES MODÈLES des programmes aux niveaux **idiomatique** et **conception** sont décrits de manière déclarative en utilisant les constituants du métamodèle PADL. Ils peuvent avoir une taille importante et être difficiles à comprendre et à manipuler.

Nous proposons une bibliothèque graphique, PTIDEJ UI⁷, pour représenter visuellement les modèles d'un programme aux niveaux **idiomatique** et **conception** et les modèles des motifs de conception.

La bibliothèque graphique associe une classe graphique à chaque constituant du métamodèle et propose une classe **Representation** pour représenter visuellement un modèle décrit avec le métamodèle PADL.

La bibliothèque graphique est basée sur un ensemble de primitives graphiques. Elle implante le motif de conception proposé comme solution au patron de conception **Usine abstraite** pour offrir différentes implantations des primitives graphiques pour différents systèmes graphiques.

Nous avons implanté les primitives graphiques pour les systèmes graphiques AWT fournis par défaut avec le langage de programmation JAVA et SWT est utilisé pour implémenter l'interface graphique de l'EDI ECLIPSE.

Ainsi, nous pouvons représenter un modèle décrit avec le métamodèle PADL uniformément dans un programme JAVA indépendant (interface AWT ou SWING) ou dans l'EDI ECLIPSE.

Aussi, la bibliothèque graphique implante les patrons de conception **Observateur** et **Stratégie**. Les mainteneurs peuvent interagir avec le modèle graphique par un mécanisme de notification ; les mises en page des modèles graphiques sont implantées dans des classes répondant à l'interface **ModelLayout**.

L'extrait de code source 5.2 page 207 montre l'utilisation de la bibliothèque graphique pour visualiser le modèle de JHOTDRAW au niveau **idiomatique** construit sur l'extrait de code source 5.1 page 204.

D'abord, nous construisons le modèle du programme JHOTDRAW au niveau **idiomatique**, lignes 1–3. Ensuite, nous créons un modèle graphique avec les primitives graphiques pour AWT, ligne 7 et avec une mise en page basée sur l'héritage, ligne 9. Enfin, nous incluons ce modèle graphique dans un canevas spécifique pour AWT, lignes 12–17, qui peut être visualisé dans un **Frame**, par exemple.

La figure 5.5 page 210 montre un sous-ensemble du résultat de la visualisation du modèle graphique de JHOTDRAW au niveau **idiomatique**.

⁷PTIDEJ UI est l'acronyme de PTIDEJ *User Interface* : interface utilisateur pour PTIDEJ.

Conclusion. Nous avons présenté une bibliothèque graphique, PTIDEJ UI, dédiée à la visualisation des modèles décrits avec le métamodèle PADL. Cette bibliothèque est implantée suivant les patrons de conception **Observateur**, **Usine abstraite** et **Stratégie** et fonctionne avec les bibliothèques graphiques AWT et SWT. Nous dressons maintenant un bilan de l'implantation des métamodèles des niveaux **idiomatique** et **conception** et des motifs de conception.

Code source 5.2 – Visualisation d'un sous-ensemble du modèle du programme JHOTDRAW au niveau **idiomatique**.

```
1 final IdiomLevelModel jhotdrawModel
2     = new IdiomLevelModel("JHotDraw idiom-level model");
3 ...
4
5 final GraphicModel graphicModel =
6     new GraphicModel(
7         jtu.ui.primitive.awt.PrimitiveFactory.getPrimitiveFactory(),
8         jhotdrawModel,
9         new InheritanceClusterLayout());
10 graphicModel.build();
11
12 final AWTCanvas awtCanvas =
13     new AWTCanvas(
14         new Canvas(
15             jtu.ui.primitive.awt.PrimitiveFactory.getPrimitiveFactory(),
16             graphicModel));
17 ...
```

□

Bilan

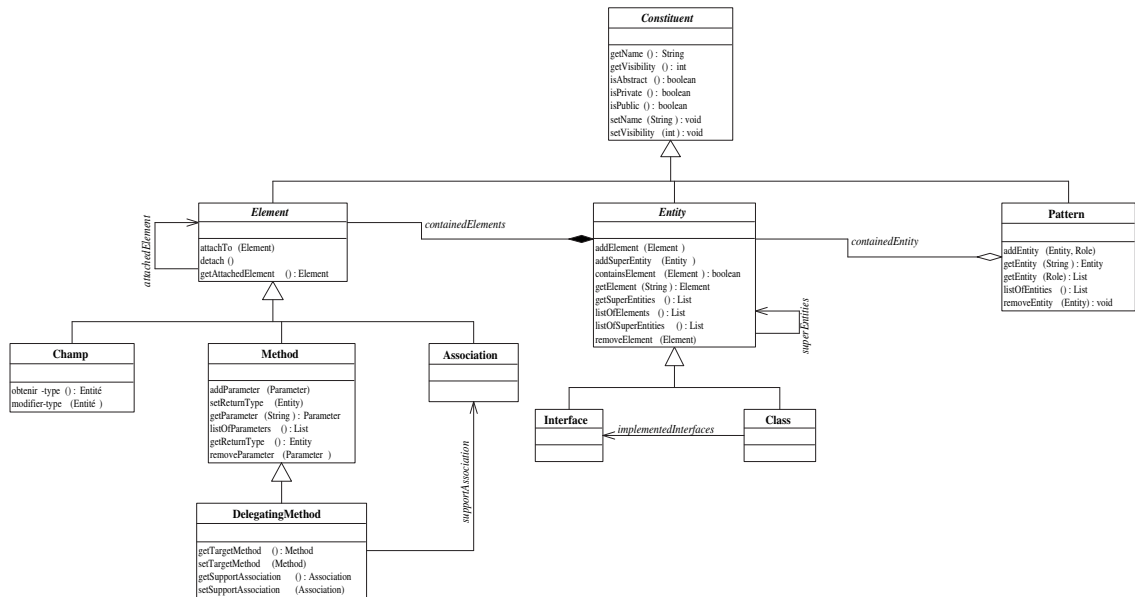
NOUS avons présenté PADL, un métamodèle pour décrire un programme aux niveaux **idiomatique** et **conception** et les motifs de conception au niveau **idiomatique** comme des entités uniques et de première classe.

Le métamodèle PADL est implanté en JAVA et propose des classes, des interfaces et des méthodes qui synthétisent les métamodèles présentés dans les sections 3.1 page 79, 4.1 page 145 et 4.2 page 148.

Les modèles décrits avec PADL peuvent être représentés graphiquement avec la bibliothèque graphique PTIDEJ UI implantée pour les bibliothèques graphiques AWT et SWT de l'EDI ECLIPSE.

Conclusion. Nous pouvons modéliser avec PADL un programme aux niveaux **idiomatique** et **conception** et un motif de conception au niveau **idiomatique**. Nous présentons maintenant l'implantation des algorithmes pour construire automatiquement le modèle du programme au niveau **idiomatique**.

FIG. 5.3 – Sous-ensemble simplifié du métamodèle PADL après restructuration du métamodèle PDL.



□

FIG. 5.4 – Sous-ensemble simplifié du métamodèle PADL après extension.

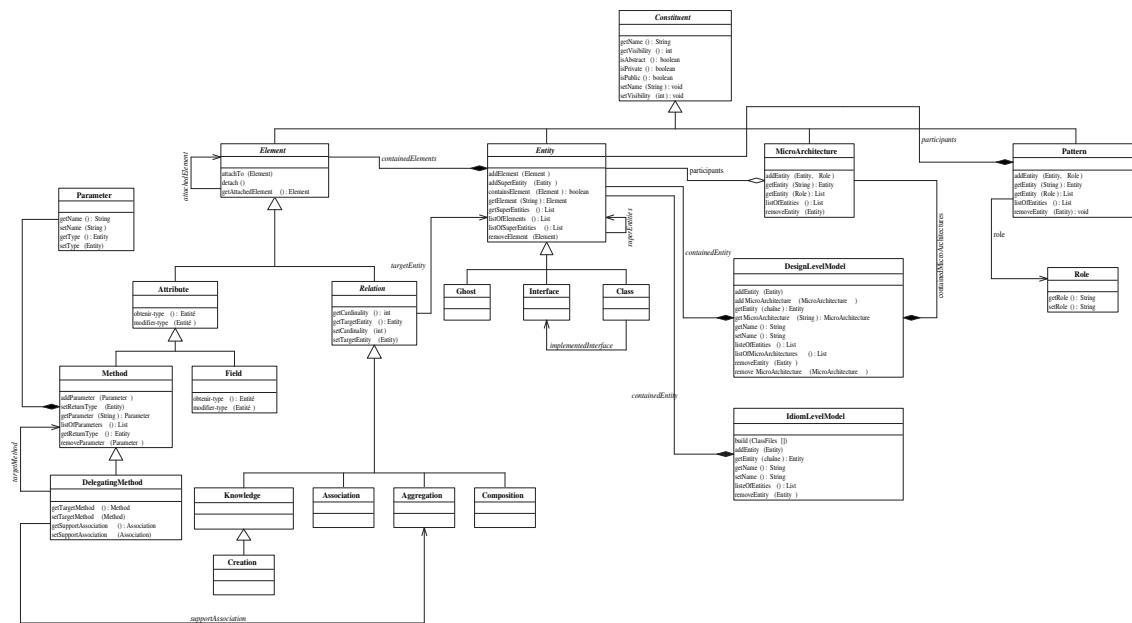
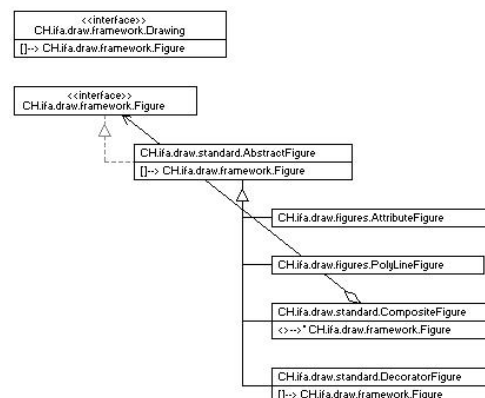


FIG. 5.5 – Modèle graphique d'un sous-ensemble du modèle du programme JHOTDRAW au niveau idiomatique.



Chapitre 6

Motifs interclasses

NOUS décrivons l’implantation des algorithmes d’analyses statiques et dynamiques pour identifier les motifs interclasses et les algorithmes de traçabilité, présentés dans les sections 3.5 page 116 et 3.6 page 122.

D’abord, nous présentons notre implantation des analyses statiques et leurs intégration avec le métamodèle PADL. Nous illustrons l’utilisation de cette implantation avec le programme JHOTDRAW.

Ensuite, nous détaillons notre implantation des analyses dynamiques. Celles-ci sont basées sur un outil générique d’analyses dynamiques des traces d’exécution de programmes JAVA avec des règles PROLOG. Nous justifions cette implantation et nous illustrons son utilisation avec le programme JHOTDRAW.

Enfin, nous décrivons nos implantations des algorithmes de traçabilité et leur intégration avec le métamodèle PADL. Nous concluons avec une évaluation de nos analyses et un bilan de nos implantations.

6.1 Analyses statiques avec INTROSPECTOR

NOUS présentons l'implantation de nos algorithmes d'analyses statiques, appelée INTROSPECTOR, pour calculer les valeurs des propriétés de multiplicité et de site d'invocation entre deux classes.

Ces algorithmes constituent la première phase de l'identification des motifs interclasses au niveau **implémentation**. Ils analysent rapidement le modèle statique d'un programme pour identifier les motifs **Association** et **Agrégation**.

Nous illustrons ces algorithmes sur le programme JHOTDRAW pour construire son modèle partiel au niveau **idiomatique**, qui sera raffiné par des analyses dynamiques.

6.1.1 Analyses statiques et métamodèle PADL

Les analyses statiques sont intégrées à chaque constituant du métamodèle et mises en œuvre dans la classe `IdiomLevelModel` dont une implantation est fournie par l'outil INTROSPECTOR.

Chaque constituant définit par le métamodèle sait reconnaître dans le modèle statique du programme au niveau **implémentation**, un code source quelconque, les constituants qui lui correspondent, par l'intermédiaire de la méthode d'identification :

```
public static List recognize(  
    final List listOfSubmittedConstituents,  
    final IdiomLevelModel idiomLevelModel)
```

La méthode `recognize()` de chaque constituant du métamodèle est exécutée dans l'ordre défini par les méthodes `recognizeRequestOrder()` avec comme paramètres la liste des entités du programme au niveau **implémentation** et le modèle en construction de l'architecture du programme au niveau **idiomatique**.

Une méthode `recognize()` instancie le constituant qui la définit lorsqu'elle reconnaît une entité qui lui correspond. Elle ajoute la nouvelle instance du constituant au modèle du programme au niveau **idiomatique**.

Ainsi, chaque constituant du métamodèle définit une méthode `recognize()` et `recognizeRequestOrder()` pour construire, de proche en proche, un modèle du programme au niveau **idiomatique**.

Par exemple, la méthode `recognize()` définie par le constituant `Class` est présentée sur l'extrait de code source simplifié 6.1 page 213. Pour chaque constituant donné en paramètre, ligne 7, si ce constituant est une classe, lignes 8–9, alors une nouvelle instance du constituant `Class` est créée, ligne 10, et ajoutée au modèle du programme au niveau *idiomatique*, ligne 11 ; puis les liens entre ce nouveau constituant et les autres constituants du modèle sont mis à jour, lignes 13–24. Si le constituant n'est pas une classe, alors il est ajouté à la liste des constituants non reconnus par la méthode `recognize()` retournée par la méthode, lignes 28 et 32.

Code source 6.1 – Méthode `recognize()` simplifiée du constituant `Classe`.

```

1  public static List recognize(
2      final List listOfSubmittedConstituents,
3      final IdiomLevelModel idiomLevelModel) {
4
5      final List notRecognizedConstituents = new ArrayList();
6      final Iterator enum = listOfSubmittedConstituents.iterator();
7      while (enum.hasNext()) {
8          final Constituent constituent = (Constituent) enum.next();
9          if (Misc.isClass(constituent)) {
10             final Class currentClass = new Class(constituent);
11             idiomLevelModel.addActor(currentClass);
12
13             // Mise à jour des liens avec les constituants
14             // déjà existants du modèle de l'architecture au
15             // niveau idiomatique :
16             final InterfaceList interfaceList =
17                 currentClassFile.getInterfaces();
18             for (int x = 0; x < interfaceList.length(); x++) {
19                 final String superInterfaceName =
20                     interfaceList.get(x);
21                 final Entity matchedActor =
22                     (Entity) idiomLevelModel.getActor(
23                         superInterfaceName);
24                 currentClass.addImplementedEntity(matchedActor);
25             }
26         }
27         else {
28             notRecognizedConstituents.add(currentClassFile);
29         }
30     }
31
32     return notRecognizedConstituents;
33 }
```

□

En particulier, les constituants **Association** et **Agrégation**, qui représentent les motifs **Association** et **Agrégation**, définissent des méthodes `recognize()` qui calculent les valeurs des propriétés $MU(A,B)$ et $SI(A,B)$ pour un ensemble de constituants du modèle d'un programme au niveau **implémentation** et qui utilisent les résultats de ces calculs pour instancier et ajouter au modèle les constituants **Association** et **Agrégation** requis pour modéliser le programme au niveau **idiomatique**.

6.1.2 Mise en œuvre des analyses statiques

Les analyses statiques sont mises en œuvre au travers de la méthode de construction `build()` du constituant `IdiomLevelModel` pour la création de modèles de programmes au niveau **idiomatique**.

La méthode `build()` prend en paramètre un ensemble d'unités de compilation JAVA en code octal fourni par l'outil d'analyse de code octal CFPARSE¹ [Greenwood, 2000]. Ces unités de compilation représentent un modèle du programme au niveau **implémentation**.

Nous utilisons le code octal JAVA car il est sémantiquement équivalent à du code source JAVA. De plus, il est plus simple à analyser et il est toujours disponible, contrairement au code source JAVA qui nécessite une analyse lexicale, syntaxique et la résolution des noms et qui peut ne pas être fourni avec, par exemple, des bibliothèques réutilisables. Nous supposons que l'analyse du code octal n'est pas utilisée illégalement [Samuelson, 1990 ; Samuelson, 2002].

L'extrait de code 6.2 page 214 présente une version simplifiée de la méthode `build()`. D'abord, la méthode prépare l'analyse en déclarant et en initialisant les listes des unités de compilation à analyser, lignes 2–6, les listes des constituants, entités et éléments, lignes 9–10, et les règles de priorités entre entités et entre éléments, lignes 11–12. Ensuite, l'algorithme soumet pour analyse les unités de compilation à chaque entité du métamodèle, lignes 14–22, puis les éléments des unités de compilation à chaque élément du métamodèle, lignes 24–48, en invoquant les méthodes `recognize()` respectives de ces constituants.

Conclusion. Nous avons détaillé l'implantation de nos algorithmes d'analyses statiques d'un programme pour calculer les valeurs des propriétés de multiplicité et de site d'invo-cation. Nous analysons le code octal JAVA pour identifier la présence de différents consti-tuants, tels les champs, les appels de méthodes, dont nous déduisons l'existence de motifs **Association** et **Agrégation** avec lesquels nous construisons un modèle partiel du programme. Nous présentons maintenant l'implantation des analyses dynamiques et leur utilisation pour raffiner ce modèle partiel et construire un modèle complet du programme au niveau **idiomatique**.

¹CFPARSE est un outil de manipulation et d'analyse du code octal distribué librement par INTERNATIONAL BUSINESS MACHINES, INC., disponible à <http://www.alphaworks.ibm.com/tech/cfparse>.

Code source 6.2 – Méthode `build()` simplifiée du constituant `IdiomLevelModel`.

```

1  public void build(final ClassFile[] classFiles) {
2      // La liste des unités de compilation à analyser :
3      final List listOfClasses = new ArrayList(classFiles.length);
4      for (int i = 0; i < classFiles.length; i++) {
5          listOfClasses.add(classFiles[i]);
6      }
7      // Les listes des entités et des éléments définis par le
8      // métamodèle et leurs priorités les unes par rapport aux autres :
9      final ClassFile[] listOfEntities = constituentRepository.getEntities();
10     final ClassFile[] listOfElements = constituentRepository.getElements();
11     final int[] sortedEntities = this.sortByPriority(listOfEntities);
12     final int[] sortedElements = this.sortByPriority(listOfElements);
13
14     // Analyse de l'architecture du programme au niveau
15     // implémentation par les entités :
16     for (int x = 0; x < listOfEntities.length
17         && listOfClasses.size() > 0; x++) {
18         Misc.getDeclaredMethod(
19             constituentRepository.getEntities()[sortedEntities[x]],
20             "recognize")
21             .invoke(null, new Object[] { listOfClasses, this });
22     }
23
24     // Analyse de l'architecture du programme au niveau
25     // implémentation par les éléments, entité par entité :
26     final List temporaryListOfEntities = this.listOfActors();
27     for (int i = 0; i < temporaryListOfEntities.size(); i++) {
28         final Entity entity = (Entity) temporaryListOfEntities.get(i);
29         final String currentClassName = entity.getName();
30
31         // Construction de la liste des éléments
32         // appartenant à l'entité analysée :
33         for (int x = 0; x < entity.getMethods().length();
34             listOfSubmittedElements.add(entity.getMethods());
35         for (int x = 0; x < entity.getFields().length();
36             listOfSubmittedElements.add(entity.getFields());
37
38         // Analyse de l'architecture du programme au niveau
39         // implémentation par les éléments :
40         for (int x = 0; x < listOfElements.length
41             && listOfSubmittedElements.size() > 0; x++) {
42             Misc.getDeclaredMethod(
43                 constituentRepository.getElements()[sortedElements[x]],
44                 "recognize")
45                 .invoke(null,
46                     new Object[] { listOfSubmittedElements, this });
47         }
48     }
49 }

```

□

6.2 Analyses dynamiques avec CAFFEINE

Nous raffinons le modèle partiel d'un programme avec des analyses dynamiques. Les analyses dynamiques calculent les valeurs des propriétés de durée de vie et d'exclusivité.

Ces algorithmes constituent la seconde phase de l'identification des motifs interclasses et nous permettent de préciser les motifs trouvés par les analyses statiques. Les analyses dynamiques utilisent un outil générique d'analyses de la trace d'exécution d'un programme JAVA avec des règles d'analyses en PROLOG dédiées aux propriétés de durée de vie et d'exclusivité.

Nous présentons d'abord notre outil d'analyse de la trace d'exécution d'un programme puis les règles dédiées à l'analyse des propriétés de durée de vie et d'exclusivité. Nous illustrons leur utilisation pour identifier les motifs *Composition*.

6.2.1 Analyse dynamique sur la trace d'exécution

Nous utilisons l'outil d'analyses des traces des programmes à l'exécution, CAFFEINE, pour réaliser les analyses dynamiques nécessaires aux calculs des valeurs de propriétés de durée de vie et d'exclusivité.

CAFFEINE est un outil 100% JAVA d'analyses de l'exécution de programmes JAVA pour aider à la compréhension de programmes, en particulier la relation entre le code source et son comportement à l'exécution.

Elliot Soloway [1986] caractérise l'activité de compréhension d'un programme comme composée d'épisodes de recherche durant lesquels les mainteneurs posent une question sur le comportement dynamique du programme, conjecturent une réponse et recherchent dans le code source du programme la confirmation de leur conjecture.

La confirmation de la conjecture est difficile et sujette à erreur. Pour aider les mainteneurs à vérifier leur conjecture, CAFFEINE génère une trace de l'exécution du programme et analyse cette trace, soit en ligne, soit hors ligne, avec une requête écrite en PROLOG.

Nous avons choisi PROLOG comme langage de requête pour ses mécanismes d'unification et de filtrage² et pour son expressivité au travers du mécanisme de retour en arrière chronologique. PROLOG a déjà montré son intérêt pour réaliser des analyses de la trace d'un programme à l'exécution [Ducassé, 1999a ; Ducassé, 1999b].

Modèle de la trace CAFFEINE définit le modèle de la trace d'un programme comme un historique des événements d'exécution du programme. Nous considérons des traces *post mortem* et des traces dynamiques. Dans les deux cas, nous ne faisons pas l'hypothèse d'une trace complète de l'exécution du programme. Nous pouvons requérir du programme le prochain événement d'exécution mais pas les précédents.

²Le filtrage s'appelle *pattern matching* en anglais [Office québécois de la langue française, 2003].

Un moteur PROLOG s'exécute comme une co-routine de la machine virtuelle JAVA dans laquelle le programme à analyser est exécuté. Le moteur PROLOG contrôle l'exécution du programme avec le prédicat `nextEvent/3` :

```
nextEvent(
    [<Liste de filtres>],
    [<Liste des événements d'exécution requis pour l'analyse>],
    E)
```

Ce prédicat supervise les classes (et leurs instances) spécifiées par les filtres ; il laisse s'exécuter le programme analysé jusqu'à l'émission d'un événement requis par l'analyse, il suspend alors le programme analysé, prend le contrôle du flot d'exécution et la variable `E` est unifiée avec l'événement émis par le programme analysé.

Modèle de l'exécution L'exécution d'un programme JAVA est modélisée comme une séquence d'événements d'exécution. Nous utilisons trois types d'événements, décrits sous la forme de termes PROLOG associés à un numéro d'identification unique, comme proposés par les algorithmes d'analyses dynamiques, section 3.5 page 116 :

- un événement d'affectation est généré par la machine virtuelle pendant l'exécution du programme à chaque fois qu'un champ d'une instance de la classe `A`, numérotée 1000, reçoit en affectation une instance de la classe `B`, numérotée 1001. Cet événement correspond à l'événement *Affectation* :

```
assignation(<event number>, A, 1000, B, 1001).
```

- un événement de ramassage est généré quand le ramasse-miettes ramasse une instance. Nous émettons l'hypothèse qu'un tel événement est obtenu aussitôt que l'instance est prête à être ramassée. Cet événement correspond à l'événement *Ramassage* :

```
finalization(<event number>, A, 1000).
```

- un événement de terminaison du programme est généré quand le programme se termine. Cet événement correspond à l'événement *Terminaison* :

```
programEnd(<event number>).
```

Ces événements sont abstraits avec les événements générés par CAFFEINE. CAFFEINE propose un ensemble plus large d'événements³ d'exécution d'un programme JAVA, le tableau 6.1 page 218 en présente une liste complète.

³L'implantation de CAFFEINE et des mécanismes de génération des événements est détaillée dans [Guéhéneuc *et al.*, 2002b].

Tableau 6.1 – Liste des événements possibles dans la trace d'exécution d'un programme.

Événement d'exécution JAVA	Définition et paramètres
<pre>fieldAccess(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>)</pre>	Le champ <Field name> de l'instance identifiée par <Unique ID of the instance possessing this field> va être lu.
<pre>fieldModification(<Field name>, <Event unique ID>, <Unique ID of the instance possessing this field>, <Unique ID of the new object assigned to this field>, <Fully qualified name of the class of the new object>)</pre>	Le champ <Field name> de l'instance identifiée par <Unique ID of the instance possessing this field> va être modifié; le numéro de l'instance affecté à ce champ est <Unique ID of the new object assigned to this field>. Le nom complètement qualifié de l'instance est donnée.
<pre>programEnd(<Event unique ID>)</pre>	Le programme se termine.
<pre>classLoad(<Class name>, <Event unique ID>)</pre>	Le programme nécessite la classe <Class name> qui est chargée par la machine virtuelle.
<pre>constructorEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the newly created instance>)</pre>	Une nouvelle instance de la classe <Declaring class name> est instanciée. Cette instance est identifiée par <Unique ID of the newly created instance> pour tout le reste de l'exécution du programme.
<pre>finalizeEntry(<Declaring class name>, <Event unique ID>, <Unique ID of the being-finalized instance>)</pre>	L'instance de la classe <Declaring class name> identifiée par <Unique ID of the being-finalized instance> est finalisée.
<pre>methodEntry(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the declaring class>)</pre>	La méthode <Method name> de la classe <Fully-qualified name of the declaring class> est appelée sur l'instance identifiée par <Unique ID of the receiver instance>.

Définitions identiques pour les événements duaux :
 classUnload/2; constructorExit/3; finalizerExit/3.

La définition change pour l'événement methodExit/6 :

<pre>methodExit(<Method name>, <Event unique ID>, <Unique ID of the caller instance>, <Unique ID of the receiver instance>, <Fully-qualified name of the declaring class>, <The value returned by this method>)</pre>	La méthode <Method name> de la classe <Fully-qualified name of the declaring class> a complété son exécution sur l'instance identifiée par <Unique ID of the receiver instance>. La valeur de retour est fournie dans <The value returned by this method>.
--	--

□

6.2.2 Analyses des propriétés

Pour des raisons d'efficacité temporelle, nous ne nous limitons pas à calculer les valeurs des propriétés pour deux classes données, nous calculons les valeurs des propriétés pour un ensemble de couples de classes.

Calcul de la valeur de $DV(A,B)$ Nous décrivons l'exécution du programme donné comme exemple d'implantation du motif *Composition*, exemple 3.13 page 109, par la trace d'exécution suivante :

```
TRACE1 = [
    assignation(1, A, 1000, B, 1001),
    finalization(2, B, 1001),
    finalization(3, A, 1000),
    programEnd(4)
]
```

Cette trace vérifie la propriété de durée de vie requise par la définition du motif *Composition* : $DV(A,B) = +$. Un champ de l'instance de la classe *A* numérotée 1000 reçoit en affectation l'instance de la classe *B* numérotée 1001, puis l'instance de *B* numérotée 1001 est ramassée par le ramasse-miettes avant l'instance de *A* numérotée 1000 qui la référence, enfin le programme termine.

Nous définissons et vérifions la propriété de durée de vie requis par la définition du motif *Composition* avec le prédicat, `checkLTProperty/3`⁴. Ce prédicat construit une liste de termes qui abstrait les événements d'exécution du programme et qui dépend de l'ordre des événements d'affectation, de ramassage, entre deux classes *A* (le tout) et *B* (la partie), et de terminaison du programme. Ces termes sont :

- un terme quand les instances de deux classes vérifient la propriété de durée de vie, $DV(A,B) = +$:

`lifetimeProperty(A, AID, B, BID, true)`

- un terme quand les instances de deux classes ne vérifient pas la propriété de durée de vie, $DV(A,B) = -$:

`lifetimeProperty(A, AID, B, BID, false)`

Nous définissons le prédicat `checkLTProperty/3` pour calculer la valeur de la propriété de durée de vie comme montré sur le code source 6.3 page 220. D'abord, à la réception d'un événement d'affectation `assignation/5` émis par le programme analysé, ligne 2, le prédicat mémorise cette affectation en ajoutant à la liste un terme `pendingAssignment/5`, ligne 5.

Ensuite, à la réception d'un événement de ramassage `finalization/3`, ligne 7, le prédicat vérifie tous les événements d'affectation reçus pour les remplacer, si nécessaire, par des termes `lifetimeProperty/5` adéquats, ligne 10. La vérification des événements d'affectation est réalisée par le prédicat `doesLTPropertyHold/3`, ligne 10, montré sur le code source 6.4 page 222.

Enfin, à la réception d'un événement de terminaison `programEnd/1`, ligne 12, le prédicat remplace tous les événements d'affectation restants par des termes `lifetimeProperty/5` adéquats, ligne 16. La conversion est réalisée par le prédicat `convertPendingAssignment/2`, ligne 16, montré sur le code source 6.5 page 222.

L'évaluation du prédicat sur la trace `TRACE1` retourne la liste suivante :

```
[lifetimeProperty(A, 1000, B, 1001, true)]
```

Le prédicat `doesLTPropertyHold/3` vérifie les événements d'affectation à la réception d'un événement de ramassage, comme montré sur le code source 6.4 page 222. D'abord, si l'événement de ramassage concerne une classe jouant le rôle de partie, ligne 3, le prédicat mémorise cette information dans l'événement d'affectation, ligne 5.

Ensuite, si l'événement de ramassage concerne une classe jouant le rôle de tout, ligne 9, et si un événement de ramassage a déjà été émis et mémorisé pour l'événement d'affectation correspondant, ligne 9, le prédicat remplace l'événement d'affectation par un terme `lifetimeProperty/5` positif, ligne 10. Si l'événement de ramassage concerne une classe jouant le rôle de tout, ligne 12, dont la partie n'a pas été déjà ramassée, ligne 13, le prédicat remplace l'événement d'affectation par un terme `lifetimeProperty/5` négatif, ligne 14. Enfin, le prédicat poursuit la vérification au reste de la liste des événements d'affectation, lignes 15–19.

Le prédicat `convertPendingAssignment/2` vérifie les événements d'affectation restants à la réception d'un événement de terminaison, comme montré sur le code source 6.5 page 222.

⁴La propriété de durée de vie (DV) s'appelle *lifetime* en anglais (LT).

Code source 6.3 – Prédicat de vérification de la propriété de durée de vie $DV(A,B)$.

```

1  checkLTProperty(
2      assignment(-, A, AID, B, BID),
3      LIST,
4      NLIST) :-
5      append(LIST, [pendingAssignment(A, AID, B, BID, [])], NLIST).
6  checkLTProperty(
7      finalization(EID, A, AID),
8      LIST,
9      NLIST) :-
10     doesLTPropertyHold(finalization(EID, A, AID), LIST, NLIST).
11 checkLTProperty(
12     programEnd(-),
13     LIST,
14     NLIST) :-
15     convertPendingAssignations(LIST, NLIST).
16 checkLTProperty(-, LIST, LIST).

```

□

D'abord, le prédicat remplace tous les événements d'affectation pour lesquels un événement de ramassage a été émis, ligne 3, par un terme `lifetimeProperty/5` positif, ligne 4. Puis, il remplace tous les événements d'affectation pour lesquels aucun événement de ramassage n'a été émis, ligne 7, par un terme `lifetimeProperty/5` positif, ligne 8. Enfin, il poursuit avec le reste de la liste des événements d'affectation, lignes 10–13.

Code source 6.4 – Prédicat de vérification des affectations à la réception d'un événement de ramassage.

```

1  doesLTPPropertyHold(finalization(EID, A, AID), [], []).
2  doesLTPPropertyHold(
3    finalization(–, B, BID),
4    [pendingAssignment(A, AID, B, BID, []) | REST],
5    [pendingAssignment(A, AID, B, BID, [finalization(B, BID))] | NLIST]) :–
6      doesLTPPropertyHold(Y, LIST, NLIST).
7  doesLTPPropertyHold(
8    finalization(–, A, AID),
9    [pendingAssignment(A, AID, B, BID, [finalization(B, BID))] | REST],
10   [lifetimeProperty(A, AID, B, BID, true) | REST]).
11 doesLTPPropertyHold(
12   finalization(–, A, AID),
13   [pendingAssignment(A, AID, B, BID, []) | REST],
14   [lifetimeProperty(A, AID, B, BID, false) | REST]).
15 doesLTPPropertyHold(
16   finalization(EID, A, AID),
17   [CAR | CDR],
18   [CAR | NCDR]) :–
19     doesLTPPropertyHold(finalization(EID, A, AID), CDR, NCDR).

```

□

Code source 6.5 – Prédicat de conversion des affectations à la réception d'un événement de ramassage ou de terminaison du programme.

```

1  convertPendingAssignations([], []).
2  convertPendingAssignations(
3      [pendingAssignment(A, AID, B, BID, [-]) | REST],
4      [lifetimeProperty(A, AID, B, BID, true) | NLIST]) :-
5      convertPendingAssignations(REST, NLIST).
6  convertPendingAssignations(
7      [pendingAssignment(A, AID, B, BID, []) | REST],
8      [lifetimeProperty(A, AID, B, BID, true) | NLIST]) :-
9      convertPendingAssignations(REST, NLIST).
10 convertPendingAssignations(
11     [A | REST],
12     [A | NLIST]) :-
13     convertPendingAssignations(REST, NLIST).

```

□

Calcul de la valeur de $EX(A, B)$ Nous définissons et vérifions la propriété d'exclusivité requise par la définition du motif **Composition** avec le prédicat **checkEXProperty/3**. Ce prédicat construit une liste de termes qui abstrait les événements d'exécution du programme et qui dépend de l'ordre des événements d'affectation :

- un terme quand une instance d'une classe **B** est affectée à une et une seule instance d'une classe **A**, $EX(A, B) = vrai$:

exclusivityProperty(A, AID, B, BID, true)

- un terme quand une instance d'une classe **B** est affectée à plus d'une instance d'une classe **A**, $EX(A, B) = false$:

exclusivityProperty(A, AID, B, BID, false)

Nous définissons le prédicat **checkEXProperty/3** pour calculer la valeur de la propriété d'exclusivité entre deux classes **A** (le tout) et **B** (la partie) comme montré sur le code source 6.6 page 224. Ce prédicat maintient une liste des termes **exclusivityProperty/5** pour vérifier la propriété d'exclusivité avec le prédicat **updateEXProperties/7**, ligne 5, montré sur le code source 6.7 page 225.

Code source 6.6 – Prédicat de vérification de la propriété d'exclusivité $EX(A, B)$.

```

1 checkEXProperty(
2   assignment(_, A, AID, B, BID),
3   LIST,
4   NLIST) :-
5   updateEXProperties(A, AID, B, BID, LIST, NLIST, true).
6 checkEXProperty(_, LIST, LIST).
```

□

Le prédicat `updateEXProperties/5` vérifie l'exclusivité entre les classes A et B à la réception d'un événement d'affectation, comme montré sur le code source 6.7 page 225. D'abord, à la réception d'un événement d'affectation, lignes 2 et 7, le prédicat vérifie si la propriété d'exclusivité est vérifiée pour deux classes A et B, lignes 1–10 ; si la propriété d'exclusivité n'est pas vérifiée, lignes 12 et 13, le prédicat remplace le terme positif par un terme négatif, ligne 14 ; si la propriété d'exclusivité est vérifiée, lignes 18 et 19, le prédicat garde un terme positif, ligne 20. Enfin, le prédicat poursuit la vérification sur le reste de la liste, lignes 23–28.

Code source 6.7 – Prédicat de mise à jour des termes `exclusivityProperty/5`.

```

1  updateEXProperties(
2      A, AID, B, BID,
3      [],
4      [exclusivityProperty(A, AID, B, BID, true)],
5      true).
6  updateEXProperties(
7      A, AID, B, BID,
8      [],
9      [exclusivityProperty(A, AID, B, BID, false)],
10     false).
11 updateEXProperties(
12     A, AID, B, BID,
13     [exclusivityProperty(A0, AID0, B, BID, true) | REST],
14     [exclusivityProperty(A0, AID0, B, BID, false) | NREST],
15     EX) :-
16     updateEXProperties(A, AID, B, BID, REST, NREST, false).
17 updateEXProperties(
18     A, AID, B, BID,
19     [exclusivityProperty(A0, AID0, B0, BID0, true) | REST],
20     [exclusivityProperty(A0, AID0, B0, BID0, true) | NREST],
21     EX) :-
22     updateEXProperties(A, AID, B, BID, REST, NREST, EX).
23 updateEXProperties(
24     A, AID, B, BID,
25     [CAR | CDR],
26     [CAR | NCDR],
27     EX) :-
28     updateEXProperties(A, AID, B, BID, CDR, NCDR, false).

```

□

6.2.3 Mise en œuvre des analyses dynamiques

Le code source 6.8 page 226 présente une version simplifiée des prédicats PROLOG utilisés pour vérifier les valeurs des propriétés dynamiques et, ainsi, l'existence d'un motif Composition entre deux classes.

L'analyse se décompose en deux phases. Dans la première phase, le prédicat `checkProperties/4`, construit les listes des termes composés `exclusivityProperty/5` et `lifetimeProperty/5` avec les prédicats précédemment définis.

Code source 6.8 – Prédicats simplifiés pour vérifier l'existence d'un motif Composition.

```

1  main(LIST) :-
2      checkProperties([], LEXP, [], LLTP),
3      checkComposition(LEXP, LLTP, LIST).
4
5  checkProperties(LEXP, NNLEXP, LLTP, NNLLTP) :-
6      nextEvent(
7          [generateConstructorEntryEvent,
8            generateFieldModificationEvent,
9            generateFinalizerExitEvent,
10           generateProgramEndEvent],
11          E),
12      interpretEvent(E, IE),
13      checkEXProperty(IE, LEXP, NLEXP),
14      checkLTPProperty(IE, LLTP, NLLTP),
15      !,
16      (
17          (IE = programEnd,
18            NNLEXP = NLEXP,
19            NNLLTP = NLLTP)
20          ;
21          checkProperties(NLEXP, NNLEXP, NLLTP, NNLLTP)
22      ).
23 checkProperties(LEXP, LEXP, LLTP, LLTP).
24
25 checkComposition([], [], []).
26 checkComposition(
27     [exclusivityProperty(A, AID, B, BID, OKAY) | EXPREST],
28     [lifetimeProperty(A, AID, B, BID, OKAY) | LTPREST],
29     [composition(A, AID, B, BID, OKAY) | NLIST]) :-
30     checkComposition(EXPREST, LTPREST, NLIST).
31 checkComposition(
32     [exclusivityProperty(A, AID, B, BID, _) | EXPREST],
33     [lifetimeProperty(A, AID, B, BID, _) | LTPREST],
34     [composition(A, AID, B, BID, false) | NLIST]) :-
35     checkComposition(EXPREST, LTPREST, NLIST).

```

□

Dans la seconde phase, le prédicat compare les résultats des calculs des propriétés pour vérifier si les valeurs requises pour le motif **Composition** sont présentes. Si les propriétés de durée de vie et d'exclusivité ont toutes deux les mêmes valeurs (**true** ou **false**), un terme composé **composition/5** est construit avec cette valeur, ligne 29 ; si les propriétés n'ont pas les mêmes valeurs, alors les propriétés dynamiques du motif de **Composition** ne sont pas vérifiées, lignes 32–34.

Le résultat est une liste de termes **composition/5** qui indiquent, pour deux classes, si les propriétés dynamiques du motif **Composition** sont vérifiées. Nous utilisons ces termes **composition/5** pour raffiner le modèle du programme obtenu par les analyses statiques au niveau **idiomatique**, comme décrit dans la section 3.6 page 122.

Conclusion. Nous avons présenté CAFFEINE, un outil d'analyses des traces d'exécution de programmes JAVA. Cet outil analyse un programme avec des règles écrites en PROLOG. Nous avons implanté un ensemble de règles pour calculer les valeurs des propriétés de durée de vie et d'exclusivité et en déduire la présence du motif **Composition**. Nous décrivons maintenant l'implantation des algorithmes pour mettre en œuvre les analyses statiques et dynamiques et construire un modèle du programme au niveau **idiomatique**.

6.3 Algorithmes de traçabilité des motifs

LA CONSTRUCTION du modèle d'un programme au niveau **idiomatique** se décompose en deux phases : la construction du modèle partiel du programme avec les analyses statiques et le raffinement de ce modèle avec les analyses dynamiques.

L'obtention du modèle du programme JHOTDRAW au niveau **idiomatique** avec les méthodes de détection intégrées au métamodèle PADL et leurs mises en œuvre dans le constituant `IdiomLevelModel` se réalise comme suit :

```
final String path = "<path of JHotDraw classfiles>";
final IdiomLevelModel idiomLevelModel = new IdiomLevelModel("JHotDraw");
idiomLevelModel.build(
    TypeLoader.loadSubtypesFromDir(null, path, ".class"));
```

La méthode `loadSubtypesFromDir()` construit une liste d'unités de compilation d'un programme avec le chemin d'accès au code octal du programme et l'extension des fichiers à analyser.

La méthode `build()` est utilisée pour construire un modèle du programme au niveau **idiomatique**. L'instance `idiomLevelModel` représente le modèle du programme JHOTDRAW au niveau **idiomatique** qui peut maintenant être manipulé en utilisant les méthodes définies par le métamodèle.

L'obtention du modèle du programme JHOTDRAW au niveau **idiomatique** avec les analyses dynamiques se réalise alors en deux phases :

1. Analyse du programme JHOTDRAW pour calculer les valeurs des propriétés de durée de vie et d'exclusivité pour un sous-ensemble des classes du programme liées par des motifs **Agrégation**.
2. Modification du modèle du programme JHOTDRAW obtenu par analyses statiques pour refléter les résultats des analyses dynamiques.

Analyse du programme JHOTDRAW L'analyse du programme JHOTDRAW est réalisée par l'exécution d'un lanceur `CAFFEINE`, comme montré sur le code source 6.9 page 229. Le lanceur exécute le programme JHOTDRAW avec la classe `JavaDrawApp`, ligne 7. Il analyse l'existence du motif **Composition**, ligne 5, entre les classes `DrawApplication` et `StandardDrawingView`, supporté par le champ `fView`, lignes 12–15. L'analyse est réalisée uniquement sur les classes `DrawApplication` et `StandardDrawingView` avec la génération des événements requis par l'analyse, lignes 8–11, comme montré sur les figures 6.1, 6.2 et 6.3 pages 230 et 231.

Modification du modèle du programme JHOTDRAW Avec les résultats des analyses dynamiques, il est possible de modifier le modèle du programme JHOTDRAW au niveau **idiomatique** pour rendre compte de l'existence de compositions : si les termes composés associés à deux classes sont *vrai*, alors les valeurs des propriétés requises par le motif **Composition** sont vérifiées. Le modèle est modifié pour refléter ce motif.

Les résultats de l'analyse dynamique sont pris en compte dans le modèle en remplaçant les instances du constituant **Aggregation** par des instances de **Composition** dans l'entité source du motif, comme montré sur le code source 6.10 page 231.

Pour chaque relation dans la classe source, lignes 1-3, si cette relation représente un motif **Agrégation**, ligne 4, et que ce motif lie la classe cible, lignes 5-6, alors l'instance du constituant **Agrégation** est supprimée de l'entité source et remplacée par une instance du constituant **Composition**, lignes 7-9.

Conclusion. Nous avons montré l'utilisation de nos algorithmes d'analyses statiques et dynamiques pour identifier les motifs interclasses et pour garantir leur traçabilité entre les niveaux **implémentation** et **idiomatique** sur l'exemple du programme JHOTDRAW. Nous concluons maintenant sur la traçabilité des motifs interclasses avant de passer à la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**.

Code source 6.9 – Lanceur CAFFEINE pour identifier le motif Composite entre les classes **DrawApplication** et **StandardDrawingView** de JHOTDRAW.

```

1 public class CaffeineLauncher {
2     public static void main(final String[] args) {
3         Caffeine
4         .run(
5             "Rules/Composition.pl",
6             "<Classpath>",
7             "CH.ifa.draw.samples.javadraw.JavaDrawApp",
8             new String[] {
9                 "CH.ifa.draw.application.DrawApplication",
10                "CH.ifa.draw.application.StandardDrawingView" },
11            Caffeine.AnalysisControlledEvents,
12            new String[][] { new String[] {
13                "CH.ifa.draw.application.DrawApplication",
14                "CH.ifa.draw.application.StandardDrawingView",
15                "fView" }
16            });
17     }
18 }
```

□

FIG. 6.1 – Démarrage de CAFFEINE pour analyser le programme JHOT-DRAW.

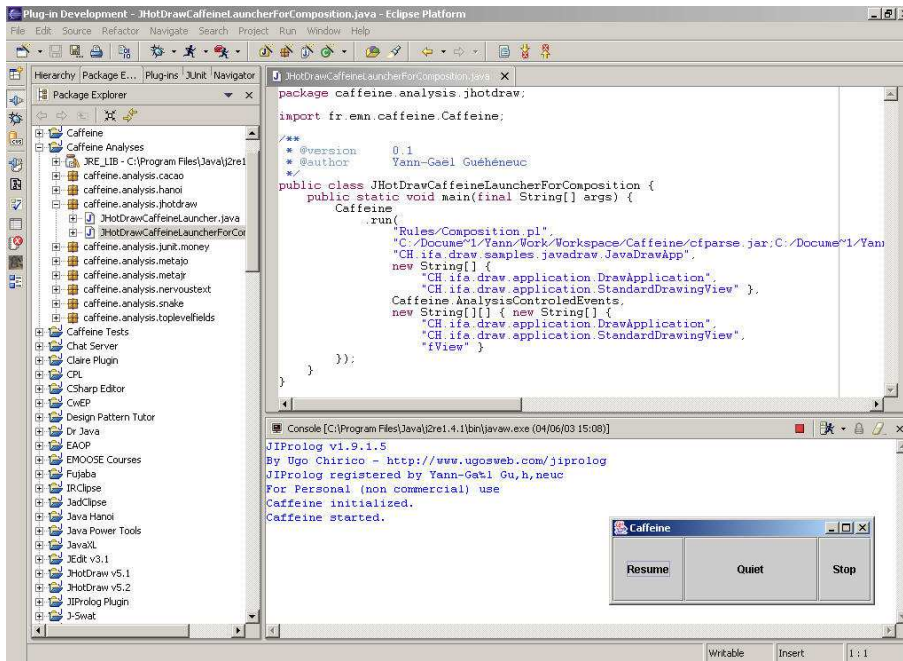


FIG. 6.2 – Utilisation du programme JHOTDRAW, CAFFEINE analyse les événements générés en arrière-plan.

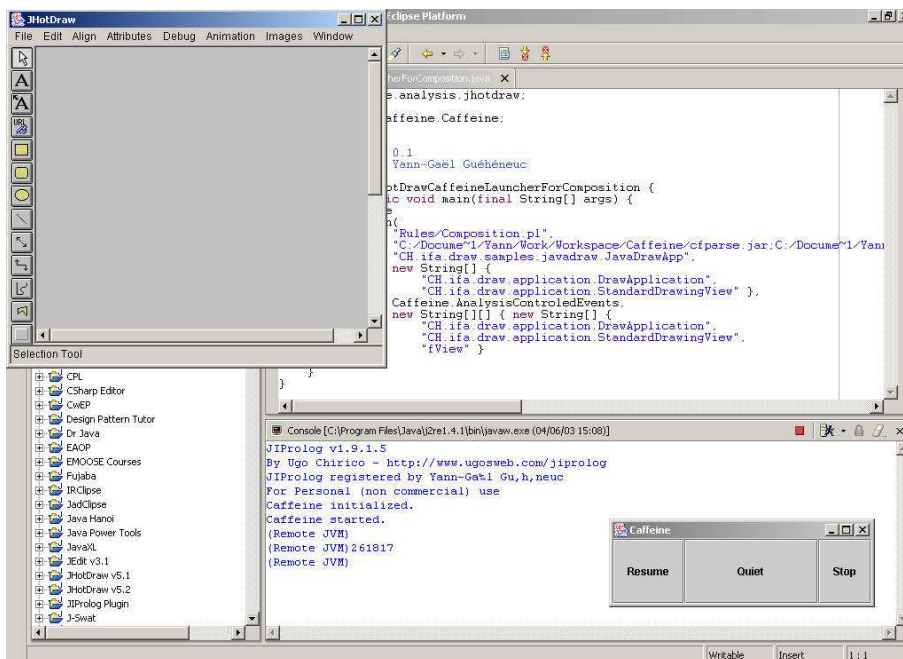
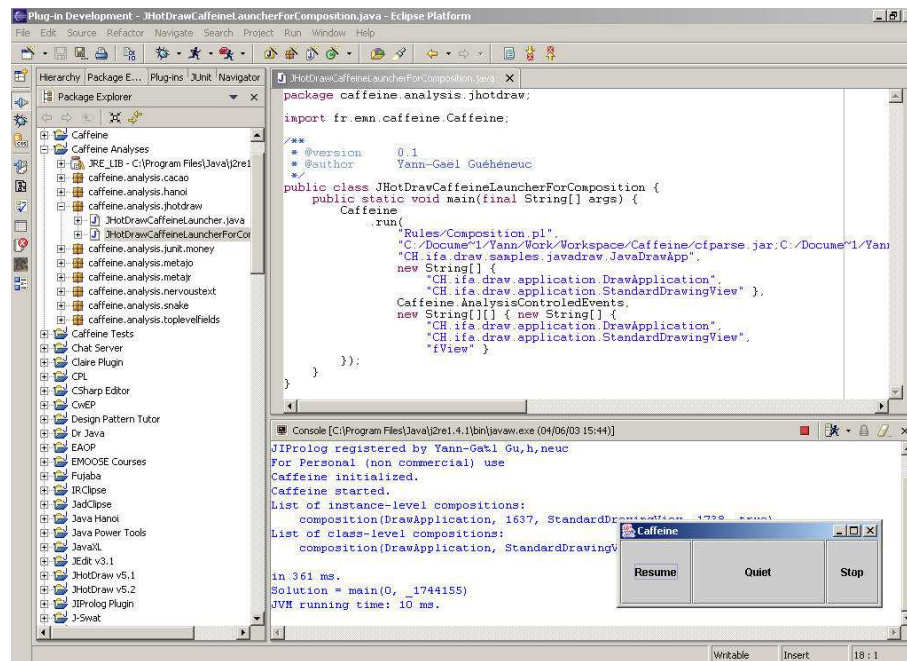


FIG. 6.3 – Résultats de l'analyse avec CAFFEINE du programme JHOT-DRAW.



Code source 6.10 – Extrait du code source simplifié pour remplacer un motif Agrégation par un motif Composition.

```

1 final List listOfElements = originEntity.listOfElements();
2 for (int i = 0; i < listOfElements.size(); i++) {
3     final Element element = (Element) listOfElements.get(i);
4     if (element instanceof Aggregation) {
5         final Aggregation aggregation = (Aggregation) element;
6         if (aggregation.getTargetActor().equals(targetEntity)) {
7             originEntity.removeActor(aggregation);
8             originEntity.addActor(
9                 new Composition(aggregation));
10        }
11    }
12 }

```

Bilan

Nous avons implanté les algorithmes d'analyses statiques et dynamiques pour construire un modèle d'un programme au niveau **idiomatique** et pour garantir la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique**.

Nous avons utilisé des algorithmes d'analyses statiques dédiés au code octal JAVA pour calculer les valeurs des propriétés de multiplicité et de site d'invocation. Ces algorithmes sont implantés par l'outil INTROSPECTOR, intégré au métamodèle PADL.

Nous avons utilisé un outil générique d'analyses des traces d'exécution de programmes JAVA avec des règles PROLOG, CAFFEINE, pour calculer les valeurs des propriétés de durée de vie et d'exclusivité.

Nous avons montré l'utilisation de INTROSPECTOR et CAFFEINE pour analyser le programme JHOTDRAW et pour construire son modèle au niveau **idiomatique** en deux phases : la construction de son modèle partiel par les analyses statiques et le raffinement de son modèle par les analyses dynamiques.

Conclusion. Ces outils nous permettent de garantir la traçabilité des motifs interclasses et de réaliser la première phase de l'identification des motifs de conception : la construction automatique du modèle d'un programme au niveau **idiomatique**, comme présenté sur les figures 1.7(a) et 1.7(b) page 24. Nous décrivons maintenant les implantations des algorithmes d'identification des micro-architectures similaires aux motifs de conception avec la programmation par contraintes avec explications.

Chapitre 7

Programmation par contraintes avec explications

NOUS présentons l'implantation des stratégies de recherche avec la programmation par contraintes avec explications utilisées pour identifier les micro-architectures similaires aux motifs de conception.

D'abord, nous introduisons l'implantation de référence de la programmation par contraintes avec explications, PALM. Puis, nous présentons PTIDEJ SOLVER, notre solveur de contraintes avec explications dédiés à la traçabilité des motifs de conception.

PTIDEJ SOLVER étend le solveur PALM et nous détaillons la modélisation d'une variable, des contraintes, des problèmes et des domaines des variables.

Enfin, nous résumons l'implantation de la bibliothèque de contraintes, PTIDEJ LIBRARY, et des stratégies de recherche dans PTIDEJ SOLVER.

7.1 Solveur de contraintes PALM

LE SOLVEUR PALM est l'implantation de référence de la programmation par contraintes avec explications. Il est implanté avec le langage de programmation CLAIRE et utilise la plate-forme de contraintes CHOCO.

7.1.1 CLAIRE et CHOCO

Le langage de programmation CLAIRE¹ [Caseau et Laburthe, 1996] est un langage de programmation hybride qui fournit un système de type riche (intervalles et types du second ordre), des classes et des types paramétriques, des règles de propagation basées sur les événements et un mécanisme de versions dynamique.

Il utilise des mécanismes de programmation sur les ensembles, un système de programmation par objets (avec ramasse-miettes, héritage simple et métaclasse), des fonctions polymorphiques et paramétriques et une approche entités-relations (avec relations explicites, inverses et inconnues).

Il permet l'expression d'algorithmes complexes simplement et leur utilisation en conjonction avec des programmes C++ ou JAVA : CLAIRE offre les mécanismes pour traduire automatiquement les programmes CLAIRE en programmes C++ ou JAVA.

La plate-forme de contraintes CHOCO [Laburthe, 2000 ; Laburthe et Le Projet OCRE, 2000] est une plate-forme implantée en CLAIRE dans le cadre du projet OCRE². Elle offre les mécanismes fondamentaux de la programmation par contraintes : gestion des domaines, propagation des contraintes, algorithmes de recherche globale et locale. Elle facilite l'implantation et l'évaluation de solveurs de contraintes.

CHOCO est aussi le nom d'un solveur de contraintes implanté avec la plate-forme CHOCO. Le solveur de contraintes CHOCO montre qu'il est possible d'utiliser effectivement et efficacement la plate-forme pour planter un solveur de contraintes, des contraintes et des algorithmes de résolution.

Le code source 7.1 page 236 présente un exemple de programme CHOCO pour la résolution d'un simple problème de satisfaction de contraintes. Ce code source illustre les principaux choix de conception et d'implantation réalisés dans la plate-forme et le solveur de contraintes CHOCO.

¹CLAIRE est l'acronyme anglais de *Combining Logical Assertions, Inheritance, Rules and Entities* : combiner les assertions logiques, l'héritage, les règles et les entités.

²OCRE est l'acronyme de Outil Contraintes pour la Recherche et l'Enseignement

Un problème de satisfaction de contraintes, les variables et les contraintes du problème sont représentés par des instances de leurs classes respectives, par exemple : **Problem**, ligne 2, pour le problème ; **IntVar**, lignes 3–5, pour les variables. Les contraintes posées, lignes 7–9, héritent toutes de la classe **AbstractConstraint**.

L’extension de la plate-forme est simplifiée par l’utilisation de l’héritage pour définir de nouveaux problèmes, variables, contraintes et mécanismes de résolution. La propagation des contraintes dispose de quatre événements sur les variables (modification des bornes inférieure et supérieure, instanciation et retrait d’une valeur du domaine) et de différents modes de réveil des contraintes :

- la méthode **awake()** est appelée au réveil des contraintes lors de la première propagation ;
- la méthode **awakeOnInf()** est appelée au réveil des contraintes lors de la modification de la borne inférieure d’une variable ;
- la méthode **awakeOnSup()** est appelée au réveil des contraintes lors de la modification de la borne supérieure d’une variable ;
- la méthode **awakeOnInst()** est appelée au réveil des contraintes lors de l’instanciation d’une variable ;
- enfin, la méthode **awakeOnRem()** est appelée au réveil des contraintes lors du retrait d’une valeur du domaine d’une variable ;

7.1.2 PALM

Le solveur de contraintes avec explications PALM³ est une extension du solveur de contraintes CHOCO développée principalement par Narendra Jussien [Jussien et Barichard, 2000]. PALM propose, par héritage, un ensemble de classes pour décrire :

- un problème de satisfaction de contraintes avec explications : **PalmProblem** (sous-classe de **Problem**) ;
- des variables numériques entières : **PalmIntVar** (sous-classe de **IntVar**) ;
- des contraintes avec explications, par exemple les contraintes d’égalité et d’inégalité : **PalmEqualxyc** et **PalmNotEqualxyc** (sous-classes de **AbstractConstraint**) ;
- des contraintes unaires, binaires et n -aires, avec $n \in \mathbb{N}, n > 2$, **PalmUnIntConstraint**, **PalmBinIntConstraint** et **PalmLargeIntConstraint**, respectivement (sous-classes de **AbstractConstraint**).

Ainsi, le code source 7.2 page 236 illustre l’adaptation à PALM du problème de satisfaction de contraintes écrit pour CHOCO, code source 7.1 page 236 : seule la déclaration du problème et le type des variables changent, les déclarations des variables et des contraintes sont identiques par l’utilisation de l’héritage et du polymorphisme.

Le solveur de contraintes avec explications PALM propose de nouvelles classes et méthodes pour contrôler la résolution ; parmi celles-ci :

³PALM est l’acronyme anglais de *Propagation and Learning with Move* : propagation et apprentissage avec déplacements.

- la classe `PalmRepair` dont une instance accompagne la résolution d'un problème de satisfaction de contraintes avec explications pour implanter la gestion des contradiction ;
- la méthode `selectDecisionToUndo()` de la classe `PalmRepair` pour choisir la contrainte à relaxer ou à retirer du problème en cas de contradiction ;
- les méthodes `awakeOnRestoreInf()`, `awakeOnRestoreSup()` et `awakeOnRestoreVal()` pour gérer le réveil des contraintes lors du rétablissement de la valeur d'une borne inférieure, supérieure et d'une valeur du domaine d'une variable.

Conclusion. PALM est un solveur de contraintes avec explications développé en CLAIRE comme une extension à la plate-forme de contraintes CHOCO. PALM offre les abstractions nécessaires à la description et à la résolution de problèmes de satisfaction de contraintes : problèmes, variables et domaines.

Code source 7.1 – Programme de résolution d'un problème de satisfaction de contraintes en CHOCO.

```

1  let
2    p:Problem := makeProblem("Simple problem"),
3    a:IntVar   := makeIntVar("A", 1, 100),
4    b:IntVar   := makeIntVar("B", 2, 7),
5    c:IntVar   := makeIntVar("C", -5, 12)
6  in (
7    post(p, a >= 3 * b - c),
8    post(p, 2 * a - 6 * b == 5 * c),
9    post(p, (a <= 10) implies (b + c > 14)),
10   solve(p, true)
11 )

```

□

Code source 7.2 – Programme de résolution d'un problème de satisfaction de contraintes avec explications en PALM.

```
1  let
2    p:PalmProblem := makePalmProblem("Simple problem with explanations"),
3    a:PalmIntVar   := makeIntVar("A", 1, 100),
4    b:PalmIntVar   := makeIntVar("B", 2, 7),
5    c:PalmIntVar   := makeIntVar("C", -5, 12)
6  in (
7    post(p, a >= 3 * b - c),
8    post(p, 2 * a - 6 * b == 5 * c),
9    post(p, (a <= 10) implies (b + c > 14)),
10   solve(p, true)
11 )
```

□

7.2 Solveur de contraintes PTIDEJ SOLVER

Nous proposons le solveur PTIDEJ SOLVER, une extension au solveur de contraintes PALM pour prendre en compte les spécificités de notre problème d'identification des micro-architectures similaires à un motif de conception par la programmation par contraintes avec explications. Nous implantons PTIDEJ SOLVER avec le langage de programmation CLAIRE, utilisé pour développer le solveur de contraintes PALM.

Le solveur PALM instancie toutes les variables car il ne gère pas les contraintes sur les ensembles, sur des variables représentant des ensembles. Aussi, nous ne réduisons pas à des singletons les domaines des variables que nous voulons instancier par des sous-ensembles de leurs domaines, la cohérence est assurée par les mécanismes de propagation.

Dans PTIDEJ SOLVER, nous spécialisons les classes `PalmIntVar` pour offrir des variables énumérées et non énumérées, les classes `PalmEqualxyc`, `PalmNotEqualxyc`, `PalmBinIntConstraint` et `PalmLargeIntConstraint` pour nommer les contraintes et leur associer des commentaires utiles pour guider les mainteneurs dans leur recherche de micro-architectures similaires à un motif de conception et la classe `PalmProblem` pour prendre en compte les particularités de notre implantation. Nous ne spécialisons pas la contrainte `PalmUnIntConstraint` car nos contraintes sont toutes, au moins, binaires.

7.2.1 Problèmes

Nous définissons une classe `PtidejProblem` pour décrire un problème de satisfaction de contraintes avec explications dans PTIDEJ SOLVER. La classe `PtidejProblem` spécialise la classe `PalmProblem` :

```
PtidejProblem <: PalmProblem()
```

Le constructeur⁴ de la classe `PtidejProblem`, `makePtidejProblem()`, instancie un nouveau problème et initialise le problème en lui donnant un nom, la taille maximum des domaines des variables et le niveau maximum de relaxation des contraintes et du problème.

Dans notre cas, la taille maximum des domaines des variables correspond toujours au nombre de classes dans le modèle du programme au niveau *idiomatique* et le niveau maximum de relaxation des contraintes est modifié dynamiquement par les algorithmes de résolution pour gérer la relaxation dynamique des contraintes et du problème :

```
[makePtidejProblem(
  s:string,
  sizeOfTheDomain:integer,
  maxRelaxationLevel:integer) : PtidejProblem
-> let pb := PtidejProblem(name = copy(s))
  in (
    // Initialisation des champs de la classe PtidejProblem.
    ...
    pb
  )
]
```

7.2.2 Variables

Nous définissons `PtidejVar` comme une sous-classe de la classe `PalmIntVar`, telle que⁵ :

```
PtidejVar <: PalmIntVar(toBeEnumerated: boolean = true)
```

La valeur de la variable d'instance `toBeEnumerated` distingue les variables énumérées des variables non énumérées. Par défaut, nous considérons que toute variable est énumérée, mais il est possible de changer la valeur du champ `toBeEnumerated` par `false` pour obtenir une variable non énumérée.

Le constructeur de la classe `PtidejVar`, `makePtidejVar()`, code source 7.3 page 239, associe la nouvelle variable avec un problème, ligne 2 ; nomme la variable, ligne 3 ; définit son domaine par ses bornes inférieures et supérieures, lignes 4 et 5 ; et indique si la variable est énumérée ou non, ligne 6.

Code source 7.3 – Constructeur de la classe `PtidejVar`.

```

1  [makePtidejVar(
2      p:palm/PalmProblem,
3      s:string,
4      i:integer,
5      j:integer,
6      enumerate:boolean) : PtidejVar
7      -> assert(i <= j),
8          let v := PtidejVar(name = copy(s), originalInf = i,
9                               originalSup = j, toBeEnumerated = enumerate)
10         in (
11             // Initialisation des champs de la classe PtidejVar.
12             v
13         )
14 ]
```

□

⁴La méthode `makePtidejProblem()` n'est pas un constructeur au sens des langages de programmation C++ ou JAVA mais une méthode CLAIRE qui retourne une nouvelle instance de la classe `PtidejProblem`.

⁵Dans le langage de programmation CLAIRE, la relation d'héritage entre deux classes est notée `<:`.

7.2.3 Contraintes

Nous offrons deux classes `PtidejBinConstraint` et `PtidejLargeConstraint`, sous-classes de `PalmBinIntConstraint` et `PalmLargeIntConstraint`, respectivement. La classe `PtidejBinConstraint` est telle que :

```
PtidejBinConstraint <: PalmBinIntConstraint(
  private/name:string,
  private/command:string,
  private/thisConstraint:any = unknown,
  private/nextConstraint:any = unknown)
```

et la classe `PtidejLargeConstraint` telle que :

```
PtidejLargeConstraint <: PalmLargeIntConstraint(
  private/name:string,
  private/command:string,
  private/thisConstraint:any = unknown,
  private/nextConstraint:any = unknown)
```

Dans ces deux classes, la variable d'instance `name:string` est une chaîne de caractères pour nommer la contrainte pour future référence pendant la recherche. La variable d'instance `command:string` est une chaîne de caractères pour guider les mainteneurs dans leurs choix des contraintes à relaxer. Les variables d'instances `thisConstraint:any` et `nextConstraint:any` mémorisent l'enchaînement des contraintes à relaxer.

Les classes `PtidejBinConstraint` et `PtidejLargeConstraint` sont abstraites et sont spécialisées pour définir les contraintes propres au problème de l'identification de micro-architectures similaires à un motif de conception, présentées section 4.4 page 166. Une implantation de ces contraintes est présentée dans la section suivante page 242.

7.2.4 Domaine des variables

Le solveur de contraintes `PTIDEJ SOLVER` travaille avec des contraintes numériques sur des variables entières. La classe `PtidejVar` décrit une variable numérique. Les domaines des variables sont donc des ensembles de valeurs entières.

À l'initialisation du problème de satisfaction de contraintes, toutes les variables ont le même domaine $\mathcal{D} = \{1, \dots, d\}$, avec $d \in \mathbb{N}$ le nombre d'entités dans le modèle du programme au niveau *idiomatique*.

Les domaines des variables sont réduits par filtrage des valeurs qui ne satisfont pas les contraintes portant sur $\{v_1, \dots, v_n\}$, avec $n \in \mathbb{N}$ le nombre de variables dans le problème, pendant la résolution du problème pour identifier les micro-architectures similaires à un motif de conception.

La sémantique des contraintes est donnée par le modèle du programme représenté dans le solveur. Nous définissons une liste globale `listOfEntities`, qui associe le modèle d'une entité à une valeur du domaine d'une variable. La liste est définie comme :

```
listOfEntities:list<Entity> := list<Entity>()
```


La classe `Entity` décrit les entités du modèle du programme dans le solveur. La définition simplifiée de la classe `Entity` est montrée par le code source 7.4 page 241. La classe `Entity` définit une entité comme une instance avec des attributs, qui représentent la sémantique des contraintes :

- `name:string` : une chaîne de caractères qui représente le nom de la classe ;
- `superEntities:list<Entity>` : la liste des superclasses de cette entité ;
- `aggregatedEntities:list<Entity>`, la liste des entités liées par un motif Agrégation avec cette entité ;
- `associatedEntities:list<Entity>`, la liste des entités liées par un motif Association avec cette entité ;
- `composedEntities:list<Entity>`, la liste des entités liées par un motif Composition avec cette entité ;
- `createdEntities:list<Entity>`, la liste des entités qui sont potentiellement instanciées par cette entité ;
- `knownEntities:list<Entity>`, la liste des entités connues de cette entité ;
- `unknownEntities:list<Entity>`, la liste des entités inconnues de cette entité.

Conclusion. Nous avons présenté PTIDEJ SOLVER, notre extension du solveur de contraintes de référence, PALM. Nous avons détaillé les classes offertes par notre solveur de contraintes pour décrire les variables (classe `PtidejVar`), les contraintes binaires (classe `PtidejBinConstraint`) et n -aires (classe `PtidejLargeConstraint`) et un problème (classe `PtidejProblem`). Nous présentons maintenant l'implantation de la bibliothèque de contraintes.

Code source 7.4 – Définition simplifiée de la classe `Entity`.

```

1  Entity <: ephemeral_object(
2      name:                string,
3      superEntities:       list<Entity>,
4      aggregatedEntities:  list<Entity>,
5      associatedEntities:  list<Entity>,
6      componentsType:     list<Entity>,
7      composedEntities:   list<Entity>,
8      createdEntities:    list<Entity>,
9      knownEntities:      list<Entity>,
10     unknownEntities:     list<Entity>
11 )

```

□

7.3 Bibliothèque de contraintes PTIDEJ LIBRARY

LA BIBLIOTHÈQUE de contraintes avec explications présentée dans la section 4.4 page 166 est implantée comme une extension au solveur de contraintes avec explications de référence, PALM.

La bibliothèque de contraintes offre les classes suivantes :

- **EqualConstraint** : cette classe représente la contrainte d'égalité, telle que :

```
EqualConstraint <: PalmEqualxyc(
  name:string,
  command:string,
  thisConstraint:any = unknown,
  nextConstraint:any = unknown
)
```

qui est instanciée par le constructeur :

```
[makeEqualConstraint(
  na:string,
  co:string,
  v1:PtidejVar,
  v2:PtidejVar) : EqualConstraint
-> let equalConstraint := makePalmBinIntConstraint(EqualConstraint, v1, v2, 0)
  in (
    equalConstraint.name := na,
    equalConstraint.command := co,
    equalConstraint.thisConstraint := makeEqualConstraint @ string,
    equalConstraint.nextConstraint := nil,
    equalConstraint as EqualConstraint
  )
]
```

dans lequel :

- **na:string** est le nom donné à la contrainte. Ce nom est utilisé pour guider les mainteneurs dans leur recherche de solutions ;
- **co:string** contient les transformations associées avec cette contrainte. Ces transformations sont les changements à opérer sur les classes concernées pour qu'elles satisfassent la contrainte. Ces transformations ne sont pas discutées plus en détails et sont utilisées pour commenter la contrainte ;
- **v1:PtidejVar** et **v2:PtidejVar** sont les variables, énumérées ou non énumérées, dont les valeurs doivent être égales ;
- la contrainte se connaît elle-même, **thisConstraint** := **makeEqualConstraint** @ string ;
- la contrainte n'est pas chaînée avec une contrainte plus faible sémantiquement, **equalConstraint.nextConstraint** := **nil**.
- **NotEqualConstraint** : cette classe représente la contrainte d'inégalité, de façon similaire à la classe **EqualConstraint** pour la contrainte d'égalité ;

- **PropertyTypeConstraint** : cette classe représente une contrainte générique utilisée pour définir par spécialisation les contraintes d'association, d'agrégation, de composition, de connaissance, d'ignorance et de création, telle que :

PropertyTypeConstraint <: **PtidejBinConstraint**(field:property)

Elle est instanciée par le constructeur :

```
[makePropertyTypeConstraint(
  ct:class,
  na:string,
  co:string,
  v1:PtidejVar,
  v2:PtidejVar,
  fd:property) : PtidejBinConstraint
-> let
  propertyTypeConstraint := makePalmBinIntConstraint(
                                ct,
                                v1,
                                v2,
                                0)
  in (
    propertyTypeConstraint.name := na,
    propertyTypeConstraint.command := co,
    propertyTypeConstraint.field := fd,
    propertyTypeConstraint as PropertyTypeConstraint
  )
]
```

dans lequel :

- **ct:class** est la classe de la contrainte que nous cherchons à définir en spécialisant la classe **PropertyTypeConstraint** ;
- **na:string** est le nom donné à la contrainte. Ce nom est utilisé pour guider les mainteneurs dans leur recherche de solutions ;
- **co:string** contient les transformations associées qui sont utilisées pour commenter la contrainte ;
- **v1:PtidejVar** et **v2:PtidejVar** sont les variables, énumérées ou non énumérées, dont les valeurs sont contraintes les unes par rapport aux autres ;
- **fd:property** est le champ de la classe **Entity**, utilisée pour modéliser les valeurs du domaine des variables, sur le contenu duquel la contrainte porte. Le champ **fd:property** peut correspondre au champ **superEntities**, **aggregatedEntities**, **associatedEntities**, **composedEntities**, **createdEntities**, **knownEntities** ou **unknownEntities**.

et avec laquelle il est possible de définir, par exemple, la contrainte d'association :

AssociationConstraint <: **PropertyTypeConstraint**()

qui est instanciée par le constructeur :

```

[makeAssociationConstraint(
  na:string,
  co:string,
  v1:PtidejVar,
  v2:PtidejVar) : AssociationConstraint
-> let
  associationConstraint := makePropertyTypeConstraint(
    AssociationConstraint,
    na, co, v1, v2,
    associatedEntities)

  in (
    associationConstraint.thisConstraint := makeAssociationConstraint @ string,
    associationConstraint.nextConstraint := makeKnowledgeConstraint @ string,
    associationConstraint as AssociationConstraint
  )
]

```

dans lequel :

- **na:string** est le nom donné à la contrainte. Ce nom est utilisé pour guider les mainteneurs dans leur recherche de solutions ;
- **co:string** contient les transformations associées qui sont utilisées pour commenter la contrainte ;
- **v1:PtidejVar** et **v2:PtidejVar** sont les variables, énumérées ou non énumérées, dont les valeurs sont liées par une relation d'association ;
- la contrainte porte sur le champ **associatedEntities** de la classe **Entity** utilisée pour modéliser les valeurs du domaine des variables ;
- la contrainte se connaît elle-même, **associationConstraint.thisConstraint := makeAssociationConstraint @ string** ;
- la contrainte est chaînée avec la contrainte de connaissance sémantiquement moins forte, **associationConstraint.nextConstraint := makeKnowledgeConstraint @ string**.

La classe **PropertyTypeConstraint** implante les méthodes abstraites de réveil des contraintes (**awake()**, **awakeOnRem()**, etc.) requises par le solveur de contraintes PALM :

- le code source 7.5 page 245 montre l'implantation de la méthode **awake()**. La méthode **awake()** assure la cohérence du sous-ensemble $\{v_1, v_2\}$ des variables du solveur de contraintes en filtrant les valeurs de leurs domaines. D'abord, l'algorithme parcourt le domaine de la variable v_1 et retire de ce domaine toutes les valeurs qui correspondent à des entités qui ne référencent pas au moins une entité dont la valeur appartient au domaine de la variable v_2 , lignes 3–24. Ensuite, l'algorithme parcourt le domaine de la variable v_2 et retire de ce domaine toutes les valeurs qui correspondent à des entités qui ne sont pas référencées par au moins une entité dont la valeur correspondante appartient au domaine de la variable v_1 , lignes 26–47.
- les méthodes **awakeOnRem()**, **awakeOnRestoreVal()** et **doAwake()**, par simplification, appellent la méthode **awake()**. Par exemple **awakeOnRem()** :

```

[awakeOnRem(c:PropertyTypeConstraint, idx:integer, v:integer) : void
-> awake(c)
]

```

Code source 7.5 – Implantation de la méthode `awake()`.

```

1  [awake(c:PropertyTypeConstraint) : void
2      -> // Pour chaque entité numérotée i dans le domaine de la variable  $v_1$  :
3      for i in domainSequence(c.v1.bucket) (
4          // Nous supposons que l'entité numérotée i ne référence pas une entité dont le
5          // numéro appartient au domaine de  $v_2$ , la valeur i doit être retirée du domaine de  $v_1$  :
6          let toBeRemoved := true in (
7              // Pour chaque entité numérotée j, référencée par l'entité numérotée i :
8              for j in (1 .. length(get(c.field, listOfEntities[i]))) (
9                  // S'il existe une entité numérotée k dans le domaine de  $v_2$  qui
10                 // correspond à une entité référencée à l'entité numérotée j:
11                 if (exists(k in domainSequence(c.v2.bucket) |
12                     get(c.field, listOfEntities[j])[k].name = listOfEntities[k].name)) (
13                     // Alors, la valeur i doit être laissée dans le domaine de  $v_1$  :
14                     toBeRemoved := false
15                 )
16             ),
17             if (toBeRemoved) (
18                 let expl := Explanation() in (
19                     self_explain(c, expl), self_explain(c.v2, DOM, expl),
20                     removeVal(c.v1, i, c.idx1, expl)
21                 )
22             )
23         )
24     ),
25     // Pour chaque entité numérotée i dans le domaine de la variable  $v_2$  :
26     for i in domainSequence(c.v2.bucket) (
27         // Nous supposons qu'il n'y a pas d'entité dans le domaine de  $v_1$  qui
28         // correspond à une entité liée à l'entité numérotée i :
29         let toBeRemoved := true in (
30             // Pour chaque entité numérotée j dans le domaine de  $v_1$  :
31             for j in domainSequence(c.v1.bucket) (
32                 // S'il existe une entité numérotée k, dans la liste des entités que
33                 // l'entité numérotée j référence, qui correspond à l'entité numérotée i :
34                 if (exists(k in (1 .. length(get(c.field, listOfEntities[j]))) |
35                     get(c.field, listOfEntities[j])[k].name = listOfEntities[i].name)) (
36                     // Alors, la valeur i doit être laissée dans le domaine de  $v_2$  :
37                     toBeRemoved := false
38                 )
39             ),
40             if (toBeRemoved) (
41                 let expl := Explanation() in (
42                     self_explain(c, expl), self_explain(c.v1, DOM, expl),
43                     removeVal(c.v2, i, c.idx2, expl)
44                 )
45             )
46         )
47     )
48 ]

```

□

Conclusion. Nous avons présenté l'implantation des contraintes représentant les relations interclasses. Nous avons défini une hiérarchie de contraintes avec trois racines principales pour décrire les contraintes d'égalité, d'inégalité et des contraintes binaires, respectivement les classes `EqualConstraint`, `NotEqualConstraint` et `PropertyTypeConstraint`. Ces classes étendent les classes proposées par le solveur de contraintes avec explications PALM. Nous présentons maintenant l'implantation des stratégies de recherche pour l'identification des micro-architectures similaires aux motifs de conception.

7.4 Stratégies de recherche

LE SOLVEUR de contraintes PTIDEJ SOLVER résout les problèmes de satisfaction de contraintes représentant l'identification des micro-architectures similaires à un motif de conception dans le modèle d'un programme au niveau **idiomatique**.

Aussi, il doit être capable de relaxer les contraintes qui ne peuvent être satisfaites, de les remplacer par des contraintes affaiblies, et de calculer ces relaxations par interactions avec le mainteneur ou automatiquement.

Nous présentons l'implantation novatrice en CLAIRE des deux stratégies de recherche définies dans la section 4.5 page 169. Le premier algorithme est basé sur l'interaction avec l'utilisateur. Le second algorithme relaxe automatiquement les contraintes et le problème.

7.4.1 Solveur interactif

La première stratégie de recherche interagit avec le mainteneur. Le mainteneur guide la résolution en indiquant au solveur de contraintes les contraintes à relaxer, à retirer ou à ajouter de nouveau.

Nous implantons cette stratégie dans la classe `PtidejInteractiveRepair`, sous-classe de la classe `PalmRepair` présentée dans la section 7.1 page 234, dans laquelle nous redéfinissons la méthode `selectDecisionToUndo()`.

Le code source 7.6 page 247 présente l'implantation simplifiée de la méthode `selectDecisionToUndo()` pour la résolution interactive utilisée par le solveur de contraintes `interactiveSolve()`.

D'abord, nous initialisons les listes de contraintes à relaxer et à ajouter de nouveau au problème, lignes 4–5. Puis, nous traitons la relaxation des contraintes et du problème en demandant au mainteneur les contraintes à relaxer, lignes 9–22. Nous construisons la liste des contraintes amenant à une contradiction présentes dans l'explication de contradiction et triées par leur poids, ligne 10. Nous présentons au mainteneur ces contraintes et leurs successeuses affaiblies si elles existent, lignes 12–17. Nous attendons le choix du mainteneur et ajoutons la contrainte choisie à la liste des contraintes à relaxer, lignes 18–21.

Ensuite, nous traitons le rajout de contraintes précédemment relaxées ou retirées du problème, lignes 25–44. La liste des contraintes précédemment relaxées est mémorisée par l'instance de la classe `PtidejInteractiveRepair`. Les contraintes sont triées par rapport à leurs poids, lignes 26–27. Tant que le mainteneur rajoute des contraintes, ligne 29, nous lui en présentons la liste, lignes 28–35. Nous attendons le choix du mainteneur, nous mettons la contrainte choisie dans la liste des contraintes à rajouter au problème et nous la supprimons de la liste des contraintes relaxées, ligne 36–41.

Enfin, si aucune contrainte n'a été relaxée ou rajoutée, nous propageons la contradiction courante ; sinon, nous retournons le couple des listes de contraintes à relaxer et à ajouter, ligne 47.

Code source 7.6 – Implantation simplifiée de la méthode `selectDecisionToUndo()` pour le solveur interactif.

```

1 [selectDecisionToUndo(repairer:PtidejInteractiveRepair, e:Explanation) :
2   tuple(list<AbstractConstraint>, list<AbstractConstraint>)
3   -> let
4     re:tuple(list<AbstractConstraint>, list<AbstractConstraint>) :=
5       tuple(list<AbstractConstraint>(), list<AbstractConstraint>())
6   in (
7     // D'abord, nous traitons la relaxation du problème
8     // par rétraction de contraintes :
9     let cts := nil in (
10      for c in e ( cts := add c ), cts := sort(isBetterConstraint @ AbstractConstraint, cts),
11      printf("\nThere is no more solution because of the constraints:\n"),
12      for i in (1 .. length(cts)) (
13        printf("  ~S. ~S\n    (weight ~S)\n", i, cts[i], weight(cts[i]))
14        if (hasWeakestConstraint(cts[i])) (
15          printf("    To be replaced with: ~S\n", getNextWeakestConstraint(cts[i]))
16        )
17      ),
18      printf("Which one do you want to relax? (0 -> none) "),
19      let selecC := read(stdin) as integer in (
20        if (selecC != 0) ( add(re[1], cts[selecC]) )
21      )
22    ),
23    // Ensuite, nous traitons l'ajout de contraintes
24    // précédemment retirées au problème :
25    if (length(repairer.userRelaxedConstraints) > 0) (
26      repairer.userRelaxedConstraints := sort(isBetterConstraint @ AbstractConstraint,
27        repairer.userRelaxedConstraints),
28      let selecC := 1 in (
29        while (selecC != 0 & length(repairer.userRelaxedConstraints) > 0) (
30          printf("\nThe following constraints led to contradictions:\n"),
31          for i in (1 .. length(repairer.userRelaxedConstraints)) (
32            printf("  ~S. ~S\n    (weight ~S)\n", i,
33              repairer.userRelaxedConstraints[i],
34              weight(repairer.userRelaxedConstraints[i]))
35          ),
36          printf("Which one do you want to put back? (0 -> none) "),
37          selecC := read(stdin) as integer,
38          if (selecC != 0) (
39            add(re[2], repairer.userRelaxedConstraints[selecC]),
40            delete(repairer.userRelaxedConstraints, repairer.userRelaxedConstraints[selecC])
41          )
42        )
43      )
44    ),
45    // La recherche s'arrête si aucune contrainte
46    // n'a été retirée ou ajoutée au problème courant :
47    if (length(re[1]) = 0 & length(re[2]) = 0) ( PalmContradiction() ), re
48 ]

```

□

7.4.2 Solveur automatique

Le solveur automatique relaxe combinatoirement les contraintes et le problème pour trouver toutes les solutions possibles. La relaxation du problème est réalisée par la méthode `combinatorialAutomaticSolve()` et la relaxation des contraintes par la méthode `selectDecisionToUndo()`.

L'extrait de code source 7.7 page 249 présente l'implantation simplifiée de la méthode `combinatorialAutomaticSolve()`. D'abord, nous initialisons le nombre de contraintes pouvant ou non être retirées du problème, lignes 3–5. Une contrainte peut être retirée du problème si son poids est inférieur à une limite donnée par le mainteneur, lignes 7–15.

Ensuite, nous utilisons une méthode de calcul des combinaisons C_n^p étendue pour générer les listes de combinaisons de contraintes pour le nombre de contraintes pouvant être retirées, ligne 18.

Pour chaque combinaison de contraintes pouvant être retirées du problème, ligne 21, nous créons et initialisons dynamiquement un problème de satisfaction de contraintes, ligne 22–26.

Ce problème a le même ensemble de variables que le problème général, lignes 27–34, le même ensemble de contraintes qui ne peuvent être retirées, lignes 35–37, et un sous-ensemble des contraintes pouvant être retirées en fonction de la combinaison courante, lignes 38–42. Enfin, nous résolvons le problème créé dynamiquement, ligne 43.

L'implantation simplifiée de la méthode `selectDecisionToUndo()` pour relaxer les contraintes est présentée sur le code source 7.8 page 251. D'abord, nous initialisons la meilleure contrainte qui mène à la contradiction et le couple des listes des contraintes à retirer du problème et à ajouter de nouveau au problème, ligne 4–7.

Si la meilleure contrainte amenant à une contradiction peut être affaiblie car elle connaît la contrainte sémantiquement moins forte qui lui succède, lignes 10–12, alors nous retirons cette contrainte du problème, ligne 15, et nous construisons une nouvelle contrainte, lignes 19–32.

La nouvelle contrainte représente un affaiblissement sémantique de la contrainte retirée du problème, ligne 20. Elle porte sur les mêmes variables que la contrainte retirée, ligne 27, et nous l'ajoutons au problème, ligne 31.

Conclusion. Nous avons présenté l'implantation des algorithmes de résolution du problème de l'identification de motifs de conception basée sur l'implantation du solveur PALM. Le premier algorithme base sa résolution sur l'interaction avec l'utilisateur ; le second calcule automatiquement toutes les solutions par relaxation automatique des contraintes et du problème. Nous présentons maintenant un bilan des implantations du solveur, de la bibliothèque de contraintes et des stratégies de recherche avant de détailler l'utilisation de ces algorithmes pour identifier et garantir la traçabilité des motifs de conception.

Code source 7.7 – Implantation simplifiée de la méthode `combinatorialAutomaticSolve()` pour le solveur automatique.

```

1  [combinatorialAutomaticSolve(currentProblem:PtidejProblem) : void
2    → let
3      numberOfRemovableConstraints:integer := 0,
4      removableConstraints:list<AbstractConstraint>
5        := list<AbstractConstraint>(),
6      nonRemovableConstraints:list<AbstractConstraint>
7        := list<AbstractConstraint>(),
8    in (
9      for constraint in currentProblem.constraints (
10        if (weight(constraint) < currentProblem.maxRelaxLvl) (
11          numberOfRemovableConstraints := numberOfRemovableConstraints + 1,
12          add(removableConstraints, constraint)
13        )
14        else (
15          add(nonRemovableConstraints, constraint)
16        )
17      ),
18      for p in (0 .. numberOfRemovableConstraints) (
19        let
20          setsOfRemovableConstraints:list := C(numberOfRemovableConstraints, p),
21          dynamicProblem:PtidejProblem := nil,
22        in (
23          for setOfRemovableConstraints in setsOfRemovableConstraints (
24            dynamicProblem := makePtidejProblem(
25              currentProblem.name /+ " (subset)",
26              currentProblem.propagationEngine.maxSize - 1,
27              currentProblem.maxRelaxLvl),
28            initCombinatorialAutomaticSolver(dynamicProblem),
29            for var in currentProblem.vars (
30              makePtidejVar(dynamicProblem,
31                var.name, var.originalInf,
32                var.originalSup, var.toBeEnumerated)
33            ),
34            for constraint in nonRemovableConstraints (
35              post(dynamicProblem, constraint)
36            ),
37            for i in (1 .. numberOfRemovableConstraints) (
38              if (not(i % setOfRemovableConstraints)) (
39                post(dynamicProblem, removableConstraints[i])
40              )
41            ),
42            solve(dynamicProblem, true)
43          )
44        )
45      )
46    )
47 ]

```

□

Code source 7.8 – Implantation simplifiée de la méthode `selectDecisionToUndo()` pour le solveur automatique.

```

1  [selectDecisionToUndo(repairer:PtidejCombinatorialAutomaticRepair, e:Explanation) :
2      tuple(list<AbstractConstraint>, list<AbstractConstraint>)
3      -> let
4          ct:AbstractConstraint :=
5              min(standard_better_constraint @ AbstractConstraint, set!(e)),
6          re:tuple(list<AbstractConstraint>, list<AbstractConstraint>) :=
7              tuple(list<AbstractConstraint>(), list<AbstractConstraint>())
8          in (
9              // Si la contrainte amenant à une contradiction peut être affaiblie :
10             if (get(nextConstraint, ct) != unknown
11                 & get(nextConstraint, ct) != nil
12                 & ct.nextConstraint.isa = method) (
13
14                 // Nous retirons cette contrainte du problème :
15                 add(re[1], ct),
16
17                 // Nous la remplaçons par une contrainte affaiblie
18                 // qui porte sur les mêmes variables :
19                 let
20                     met:method := ct.nextConstraint as method,
21                     args:list<any> := list<any>(),
22                     nnm:string := ct.name as string,
23                     nct:AbstractConstraint := nil
24                 in (
25                     add(args, nnm),
26                     add(args, ct.command),
27                     for i in (1 .. getNbVars(ct)) ( add(args, getVar(ct, i)) ),
28                     nct := apply(met, args) as AbstractConstraint,
29
30                     // Nous ajoutons la contrainte affaiblie au problème :
31                     add(re[2], nct)
32                 )
33             ),
34             // La recherche s'arrête si aucune contrainte
35             // n'a été retirée ou ajoutée au problème courant :
36             if (length(re[1]) = 0 & length(re[2]) = 0) ( PalmContradiction() ),
37             re
38         )
39 ]

```

□

Bilan

NOUS avons présenté PALM, l'implantation de référence de la programmation par contraintes avec explications. PALM est implanté en CLAIRE et étend la plateforme de contraintes CHOCO dans laquelle problèmes, variables, contraintes et domaines des variables sont modélisés par des classes CLAIRE.

Nous avons étendu PALM pour implanter PTIDEJ SOLVER, un solveur de contraintes dédié à l'identification des micro-architectures similaires à des motifs de conception. PTIDEJ SOLVER implante les stratégies de recherche présentées dans la section 4.5 page 169.

PTIDEJ SOLVER utilise une bibliothèque de contraintes dédiés, PTIDEJ LIBRARY, qui propose une implantation des contraintes présentées dans la section 4.4 page 166. Cette bibliothèque définit trois classes desquelles sont dérivées toutes les contraintes.

Conclusion. PTIDEJ SOLVER et PTIDEJ LIBRARY offrent les implantations nécessaires à l'identification des motifs de conception. Ces implantations nous permettent maintenant d'identifier les micro-architectures similaires à un motif de conception par la recherche des solutions complètes et approchées au problème de satisfaction de contraintes correspondant.

Chapitre 8

Motifs de conception

NOUS montrons l'utilisation de PADL pour décrire un motif de conception et de PTIDEJ pour obtenir un système de contraintes de ce modèle et un domaine pour ce système du modèle du programme au niveau idiomatique.

Ensuite, nous présentons l'utilisation de PTIDEJ SOLVER et PTIDEJ LIBRARY pour résoudre le problème de satisfaction de contraintes traduisant l'identification des micro-architectures similaires au modèle du motif de conception.

Puis, nous décrivons l'implantation de l'algorithme pour construire un modèle du programme au niveau **conception** qui intègre les micro-architectures similaires au motif de conception identifiées et son utilisation.

Enfin, nous illustrons nos implantations sur le programme JHOTDRAW et le motif de conception **Composite**.

8.1 Modélisation d'un motif avec PADL

LA MODÉLISATION d'un motif de conception au niveau **idiomatique** consiste à décrire ses participants et leurs relations avec les constituants du métamodèle PADL. Cette modélisation fournit un modèle abstrait du motif.

Nous suivons le processus présenté sur la figure 4.3 page 152 pour déclarer et instancier un modèle abstrait du motif de conception **Composite**.

En terme de structure, le constituant **Composite**, qui représente un modèle abstrait du motif de conception **Composite**, est déclaré comme une sous-classe du constituant **Pattern**. Son constructeur instancie les différents constituants du modèle et les inclut au modèle abstrait, comme montré par l'extrait de code source 8.1 page 254.

En terme de comportement, le constituant **Composite** décrit les services assurant l'adaptation de ses instances à un contexte donné; par exemple, pour le constituant **Composite**, la définition des méthodes `addLeaf()`, `removeLeaf()` et `addComposite()`.

Un modèle abstrait du motif **Composite** est obtenu par instanciation du constituant correspondant, par instanciation de la classe JAVA **Composite**. Chaque modèle abstrait est une entité de première classe qui contient toute la logique du motif.

Conclusion. Nous avons montré comment construire un modèle abstrait d'un motif de conception en utilisant les constituants du métamodèle PADL. Nous présentons maintenant l'implantation des algorithmes de génération d'un problème de satisfaction de contraintes avec le modèle abstrait d'un motif de conception et le modèle d'un programme au niveau **idiomatique**.

Code source 8.1 – Déclaration simplifiée du constituant **Composite**.

```

public class Composite extends Pattern {
    // Déclaration du leitmotiv du motif Composite,
    // faite dans le constructeur de la classe Composite,
    // sous-classe de la classe Pattern :
    public Composite() {
        // Déclaration de l'interface Component :
        final Interface iComponent = new Interface("Component");
        // Déclaration de la méthode operation(),
        // définie dans l'interface Component :
        final Method mOperation = new Method("operation");
        iComponent.addElement(mOperation);
        // Ajout de l'interface Component au modèle :
        this.addActor(iComponent);

        // Motif de composition ayant pour cible Component et de cardinalité > 1 :
        final Composition aComposition =
            new Composition("children", iComponent, 2);

        // Déclaration de la classe Composite :
        final Class cComposite = new Class("Composite");
        // La classe Composite a pour interface Component :
        cComposite.addImplementedEntity(iComponent);
        // La classe Composite et l'interface Component sont liées
        // par le motif de composition children :
        cComposite.addActor(aComposition);
        // La méthode operation() définie dans la classe Composite implémente
        // la méthode operation() de l'interface Component et est liée via le
        // motif de composition aComposition à cette interface :
        final DelegatingMethod mDelegation =
            new DelegatingMethod("operation", aComposition);
        mDelegation.attachTo(mOperation);
        cComposite.addActor(mDelegation);
        // Ajout de la classe Composite au modèle :
        this.addActor(cComposite);

        // Ajout d'une classe Leaf :
        this.addLeaf("Leaf");
    }
    // Déclaration des services propres au modèle.
    // Exemple de la méthode addLeaf() :
    public void addLeaf(final String leafName) {
        // Déclaration de la classe Leaf :
        final Class aClass = new Class(leafName);
        // La classe Leaf a pour interface Component :
        aClass.addImplementedEntity((Interface) this.getActor("Component"));
        // L'interface publique de la classe Leaf est générée automatiquement
        // (création d'une méthode operation()) :
        aClass.assumeAllInterfaces();
        // Ajout de la classe Leaf au modèle :
        this.addActor(aClass);
    }
    ...
}

```

8.2 Génération d'un problème avec **PTIDEJ**

UN PROBLÈME de satisfaction de contraintes se décompose en un système de contraintes, avec des variables et des contraintes entre ces variables, et en un domaine pour les variables.

Nous générons automatiquement un système de contraintes du modèle abstrait du motif de conception recherché avec les contraintes de la bibliothèque de contraintes et le domaine des variables du modèle du programme au niveau *idiomatique*.

Le modèle abstrait du motif de conception et le modèle du programme sont décrits avec le métamodèle PADL. Nous implantons un mécanisme de *Visiteur* dans le métamodèle pour parcourir ses constituants.

Nous présentons ce mécanisme et les visiteurs utilisés pour générer le problème de satisfaction de contraintes avec explications correspondant à l'identification des micro-architectures similaires à un motif de conception dans le modèle d'un programme.

Nous illustrons les visiteurs sur le modèle du programme **JHOTDRAW** au niveau *idiomatique* et sur le modèle abstrait du motif de conception **Composite**.

8.2.1 Domaine des variables

Nous générons automatiquement du modèle d'un programme le code nécessaire pour créer les instances de la classe **Entity** et la liste globale **listOfEntities**. Nous spécialisons l'interface **Walker** intégrée au métamodèle PADL pour implanter un visiteur qui parcourt tous les constituants d'un modèle d'un programme exprimé avec PADL.

Le code source 8.2 page 257 présente un extrait de l'implantation du visiteur. Typiquement, le visiteur **DomainVisitor** déclare un ensemble de listes sur les constituants du modèle à traiter, par exemple des listes des relations d'associations, d'agrégation, de composition, etc., lignes 2–7.

Puis, les méthodes **close()** du visiteur sont utilisées pour générer les informations nécessaires. En particulier, la méthode **close(AbstractModel)** est appelée quand tout le modèle a été parcouru et est utilisée pour générer la liste globale des entités du modèle, lignes 10–18.

Les autres méthodes du visiteur sont utilisées pour calculer les informations nécessaires à la génération, par exemple la méthode **void visit(Association)**, ligne 21, est appelée quand un élément représentant une relation d'association est visité. Cette méthode ajoute à la liste courante des relations d'association, lignes 23–25, le nom de l'entité cible de la relation, ligne 22.

8.2.2 Application à JHOTDRAW

Nous utilisons le visiteur `DomainVisitor` pour générer automatiquement le domaine des variables associées au programme JHOTDRAW comme une suite d'instructions d'instanciation de la classe `Entity` et pour générer automatiquement la liste globale des entités existantes dans le modèle.

Le code source 8.3 page 258 présente un extrait du domaine des variables généré pour le modèle du programme JHOTDRAW par le visiteur `DomainVisitor`. D'abord, les instances représentant les entités du modèle du programme sont créées¹, lignes 1–13. Ensuite, les attributs de chaque entité sont initialisés pour rendre compte des relations de l'entité considérée avec les autres entités du modèle, lignes 15–32. Enfin, la liste globale des entités représentant le domaine des variables est construite et initialisée, lignes 34–38.

Code source 8.2 – Extrait de l'implantation du visiteur pour générer le domaine des variables d'un modèle du programme au niveau *idiomatique*.

```

1  public class DomainGenerator implements Walker {
2      private StringBuffer buffer = new StringBuffer(0);
3      private List listOfAggregations = new ArrayList(0);
4      private List listOfAssociations = new ArrayList(0);
5      private List listOfCompositions = new ArrayList(0);
6      private List listOfCreations = new ArrayList(0);
7      private List listOfKnowledges = new ArrayList(0);
8      ...
9      private void close(final AbstractModel abstractModel) {
10         this.buffer.append("listOfEntities:list<Entity> := list<Entity>(");
11         final Iterator iterator = abstractModel.listOfActors().iterator();
12         while (iterator.hasNext()) {
13             this.buffer.append(((Entity) iterator.next()).getName());
14             if (iterator.hasNext()) {
15                 this.buffer.append(", ");
16             }
17         }
18         this.buffer.append(')');
19     }
20     ...
21     public void visit(final Association association) {
22         final String entityName = association.getTargetActor().getName();
23         if (!this.listOfAssociations.contains(entityName)) {
24             this.listOfAssociations.add(entityName);
25         }
26     }
27     ...
28 }

```

□

¹Les noms des instances sont construits avec leurs codes de hachage.

Code source 8.3 – Extrait du domaine des variables généré pour le modèle du programme JHOTDRAW au niveau idiomatique.

```

1 pe988072791:Entity := Entity(name = "CH.ifa.draw.framework.Drawing")
2 pe-2037400668:Entity := Entity(name = "CH.ifa.draw.framework.DrawingEditor")
3 pe1517942428:Entity := Entity(name = "CH.ifa.draw.framework.DrawingView")
4 pe-473192309:Entity := Entity(name = "CH.ifa.draw.framework.Figure")
5 pe1597020576:Entity := Entity(name = "CH.ifa.draw.standard.AbstractFigure")
6 ...
7 pe-1597069526:Entity := Entity(name = "CH.ifa.draw.figures.AttributeFigure")
8 pe656417050:Entity := Entity(name = "CH.ifa.draw.figures.PolyLineFigure")
9 pe290968465:Entity := Entity(name = "CH.ifa.draw.standard.CompositeFigure")
10 pe-663125435:Entity := Entity(name = "CH.ifa.draw.standard.DecoratorFigure")
11 pe-423130161:Entity := Entity(name = "CH.ifa.draw.framework.Handle")
12 pe-232471297:Entity := Entity(name = "CH.ifa.draw.framework.Tool")
13 pe1063877011:Entity := Entity(name = "java.lang.Object")
14
15 (pe988072791.superEntities := list<Entity>(pe-299881549, pe-1343835665, pe1832181019),
16 pe988072791.aggregatedEntities := list<Entity>(pe-543894340, pe-473192309, pe-432734277, ...),
17 pe988072791.associatedEntities := list<Entity>(pe-543894340, pe-473192309, pe-192747385),
18 pe988072791.composedEntities := list<Entity>(),
19 pe988072791.knownEntities := list<Entity>(),
20 pe988072791.createdEntities := list<Entity>(),
21 pe988072791.unknownEntities := list<Entity>(pe1597020576, pe290968465, pe-1343835665, ...))
22
23 ...
24
25 (pe1063877011.superEntities := list<Entity>(),
26 pe1063877011.aggregatedEntities := list<Entity>(),
27 pe1063877011.associatedEntities := list<Entity>(),
28 pe1063877011.componentsType := list<Entity>(),
29 pe1063877011.composedEntities := list<Entity>(),
30 pe1063877011.knownEntities := list<Entity>(),
31 pe1063877011.createdEntities := list<Entity>(),
32 pe1063877011.unknownEntities := list<Entity>(pe1597020576, pe290968465, pe-1343835665, ...))
33
34 listOfEntities: list<Entity> := list<Entity>(
35     pe988072791, pe-2037400668, pe1517942428, pe-473192309, pe1597020576,
36     ...,
37     pe-1597069526, pe656417050, pe290968465, pe-663125435, pe-423130161,
38     pe-232471297, pe1063877011)

```

□

8.2.3 Système de contraintes

Nous générons automatiquement du modèle du motif de conception le problème de satisfaction de contraintes représentant l'identification des micro-architectures similaires à ce motif. Nous spécialisons l'interface **Walker** intégrée au métamodèle PADL pour implanter un visiteur qui parcourt tous les constituants d'un modèle d'un motif de conception décrit avec PADL.

Le code source 8.4 page 259 présente un extrait de l'implantation du visiteur. Typiquement, le visiteur **ConstraintVisitor** déclare un ensemble de listes sur les constituants du modèle à traiter, par exemple le modèle du motif courant, l'entité courante, le nombre d'entités, etc., lignes 2–8.

Puis, les méthodes **open()** et **close()** du visiteur sont utilisées pour générer les informations nécessaires. En particulier, la méthode **void open(Pattern)** est au début du parcours du modèle et est utilisée pour déclarer le problème de satisfaction de contraintes, lignes 10–33.

La méthode **void close(Pattern)** est appelée quand tout le modèle a été parcouru et est utilisée pour générer les déclarations des contraintes, lignes 36–38, et pour fermer la déclaration du problème, lignes 39–41. Les méthodes **visit()** sont utilisées pour créer les contraintes des relations entre les participants du motif modélisé, lignes 44–46.

Code source 8.4 – Extrait de l’implantation du visiteur pour générer le système de contraintes du modèle d’un motif.

```

1  public abstract class PtidejSolverConstraintGenerator implements Walker {
2      private StringBuffer buffer = new StringBuffer();
3      private Pattern currentPattern;
4      private Map knowledgeLinks = new HashMap();
5      private Entity enclosingEntity;
6      private Entity[] pEntities;
7      private int numberOfEntities;
8      private List aggregationTargets = new ArrayList();
9      ...
10     private final void open(final Pattern p) {
11         this.currentModel = p;
12         this.numberOfEntities = p.listOfEntities().size();
13         this.pEntities = new Entity[this.numberOfEntities];
14         p.listOfEntities().toArray(this.pEntities);
15         this.buffer.append(' ');
16         this.buffer.append("customProblemFor");
17         this.buffer.append(p.getName());
18         this.buffer.append("Model() : PtidejProblem");
19         this.buffer.append("\n    -> verbose() := 0,");
20         this.buffer.append("\n        let pb := makePtidejProblem(\"");
21         this.buffer.append(p.getName());
22         this.buffer.append(" Model Problem\", length(listOfEntities), 90)");
23         for (int i = 0; i < numberOfEntities; i++) {
24             this.buffer.append(",\n        ");
25             this.buffer.append(Misc.minimizeFirstLetter(pEntities[i]));
26             this.buffer.append("Var");
27             this.buffer.append(" := makePtidejVar(pb, \"");
28             this.buffer.append(pEntities[i]);
29             this.buffer.append("\", 1, length(listOfEntities))");
30         }
31         this.buffer.append(" in (\n\n");
32         this.buffer.append("setVarsToShow(pb.globalSearchSolver, pb.vars),\n\n");
33     }
34     ...
35     private final void close(final Pattern p) {
36         this.inheritances();
37         this.ignorances();
38         this.inequalities();
39         this.buffer.append("        pb\n");
40         this.buffer.append("    )\n");
41         this.buffer.append("]\n");
42     }
43     ...
44     public void visit(final Association p) {
45         this.createConstraint(this.enclosingEntity, "Association", p.getWeight());
46     }
47     ...
48 }

```

□

8.2.4 Application au motif de conception **Composite**

Nous utilisons le visiteur `ConstraintVisitor` pour générer automatiquement le système de contraintes associés au modèle du motif de conception **Composite**.

Le code source 8.5 page 262 présente un extrait du système de contraintes généré pour le modèle du motif de conception **Composite** par le visiteur `ConstraintVisitor`. D'abord, le problème de satisfaction est déclaré, ligne 2. Ensuite, les variables représentant les participants du motif de conception sont ajoutées au problème, lignes 3–5. Puis, les relations entre les participants sont représentées sous la forme de contraintes, par exemple la relation de composition, lignes 7–13, la relation d'héritage, lignes 14–20, la relation d'ignorance, lignes 22–28, et la relation de différence entre les participants, lignes 30–36.

Conclusion. Nous avons appliqué le patron de conception *Visiteur* au métamodèle PADL et implanté deux visiteurs pour générer automatiquement un problème de satisfaction de contraintes du modèle abstrait d'un motif de conception et du modèle d'un programme au niveau *idiomatique*. Nous présentons maintenant la résolution du problème de satisfaction de contraintes pour l'identification des micro-architectures similaires à un motif de conception.

Code source 8.5 – Extrait du problème de satisfaction de contraintes généré pour le modèle du motif de conception **Composite**.

```

1  [problemForCompositeMotif() : PtidejProblem
2    -> let pb := makePtidejProblem("Composite motif problem", length(listOfEntities), 90),
3      componentVar := makePtidejVar(pb, "Component", 1, length(listOfEntities)),
4      compositeVar := makePtidejVar(pb, "Composite", 1, length(listOfEntities)),
5      leafVar := makePtidejVar(pb, "Leaf", 1, length(listOfEntities)) in (
6      ...
7      post(pb,
8        makeCompositionConstraint(
9          "Composite <#>-> Component",
10         "", // Commentaire pour les développeurs.
11         compositeVar,
12         componentVar),
13        100),
14      post(pb,
15        makeInheritancePathConstraint(
16          "Composite -|>- ... -|>- Component",
17          "", // Commentaire pour les développeurs.
18          compositeVar,
19          componentVar),
20        50),
21      ...
22      post(pb,
23        makeIgnoranceConstraint(
24          "Composite -/--> Leaf",
25          "", // Commentaire pour les développeurs.
26          compositeVar,
27          leafVar),
28        75),
29      ...
30      post(pb,
31        makeNotEqualConstraint(
32          "Component <> Composite",
33          "", // Commentaire pour les développeurs.
34          componentVar,
35          compositeVar),
36        100),
37      ...
38      pb
39    )
40 ]

```

□

8.3 Résolution du problème avec PTIDEJ SOLVER

NOUS mettons en œuvre PTIDEJ SOLVER et les implantations des stratégies de recherche des solutions aux problèmes de satisfaction de contraintes pour l'identification des micro-architectures similaires à des motifs de conception.

D'abord, nous présentons l'utilisation de la recherche guidée par les mainteneurs. Puis, nous détaillons l'utilisation de la recherche automatique combinatoire. Aussi, nous décrivons la forme des solutions générées par le solveur de contraintes PTIDEJ SOLVER.

8.3.1 Recherche interactive

La résolution interactive du problème de satisfaction de contraintes pour identifier les micro-architectures similaires au motif de conception **Composite** se réalise comme suit :

```
[solvePtidejProblem() : void
-> load("<path>/Domain.cl"),
    searchMicroArchitectures(
        interactiveSolve @ PtidejProblem,
        problemForCompositeMotif()),
    exit(-1)
]

(solvePtidejProblem())
```

D'abord, nous chargeons dans le système de contraintes PTIDEJ SOLVER le domaine des variables qui représente le modèle du programme JHOTDRAW au niveau **idiomatique**. Ensuite, nous cherchons les solutions au problème de satisfaction de contraintes représentant le modèle du motif de conception avec le solveur interactif.

Le solveur commence par chercher toutes les solutions complètes et demande au mainteneur de guider la recherche lorsque le système est en contradiction, lorsque le solveur ne peut trouver aucune (autre) solution complète, comme montré sur la figure 8.1 page 264.

Une seule contrainte amène à une contradiction : la contrainte représentant le motif **Composition** entre les entités jouant le rôle de **Ramification** et de **Branche** ; le mainteneur indique au solveur de retirer cette contrainte du système de contraintes et la recherche reprend, comme montré sur la figure 8.2 page 266, jusqu'à la contradiction suivante, comme montré sur la figure 8.3 page 266.

8.3.2 Recherche combinatoire automatique

La résolution automatique combinatoire du problème de satisfaction de contraintes pour identifier les micro-architectures similaires au motif de conception **Composite** se réalise comme suit :

```
[solvePtidejProblem() : void
-> load("<path>/Domain.cl"),
    searchMicroArchitectures(
        combinatorialAutomaticSolve @ PtidejProblem,
        problemForCompositeMotif()),
    exit(-1)
]
(solvePtidejProblem())
```

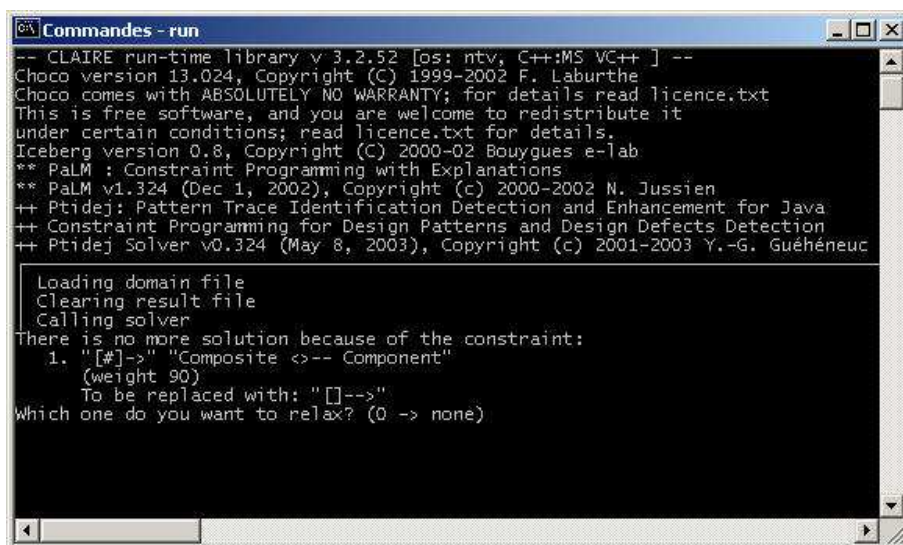
D'abord, nous chargeons dans le système de contraintes PTIDEJ SOLVER le domaine des variables qui représente le modèle du programme JHOTDRAW au niveau *idiomatique*. Ensuite, nous cherchons les solutions au problème de satisfaction de contraintes représentant le modèle du motif de conception avec le solveur automatique combinatoire.

Le solveur commence par chercher toutes les solutions complètes puis relaxe les contraintes et le problème combinatoirement : d'abord, une contrainte après l'autre, comme montré sur la figure 8.4 page 267 ; puis, en combinaisons, comme montré sur la figure 8.5 page 268.

Les solutions obtenues par le solveur automatique combinatoire sont sauvegardées dans un fichier qui résume les contraintes relaxées et retirées du problème de satisfaction de contraintes, les rôles et les entités du modèle du programme qui appartiennent à une micro-architecture similaire au modèle du motif de conception *Composite*, comme montré sur le tableau 8.1 page 265. Une solution est de la forme :

```
<numéro de la solution>.<écart>.<rôle dans le modèle du motif> =
  <entité dans le modèle du programme>
```

FIG. 8.1 – Système de contraintes en contradiction, le mainteneur guide la recherche.



Conclusion. Nous avons présenté l'utilisation des deux stratégies de recherche implantées dans PTIDEJ SOLVER pour identifier les micro-architectures similaires au motif de conception Composite dans le modèle du programme JHOTDRAW au niveau idiomatique et la forme des solutions obtenues. Nous détaillons maintenant l'obtention d'un modèle du programme JHOTDRAW au niveau conception avec les micro-architectures fournies par le solveur.

Tableau 8.1 – Solutions obtenues avec le solveur automatique combinatoire.

```

# Solution without constraint :
# -|>- : CompositeRoot -|>- Component
# Solution without constraint :
# [#]-> : CompositeRoot <>--> Component
# Solution with constraint :
# []--> : CompositeRoot <>--> Component

1.30.Component = CH.ifa.draw.framework.FigureChangeListener
1.30.Composite = CH.ifa.draw.standard.AbstractFigure
1.30.Leaf = CH.ifa.draw.framework.Drawing
1.30.CompositeRoot = CH.ifa.draw.standard.AbstractFigure

...

# Solution without constraint :
# -|>- : CompositeRoot -|>- Component
# Solution without constraint :
# [#]-> : CompositeRoot <>--> Component
# Solution with constraint :
# []--> : CompositeRoot <>--> Component
# Solution without constraint :
# []--> : CompositeRoot <>--> Component
# Solution with constraint :
# ----> : CompositeRoot <>--> Component

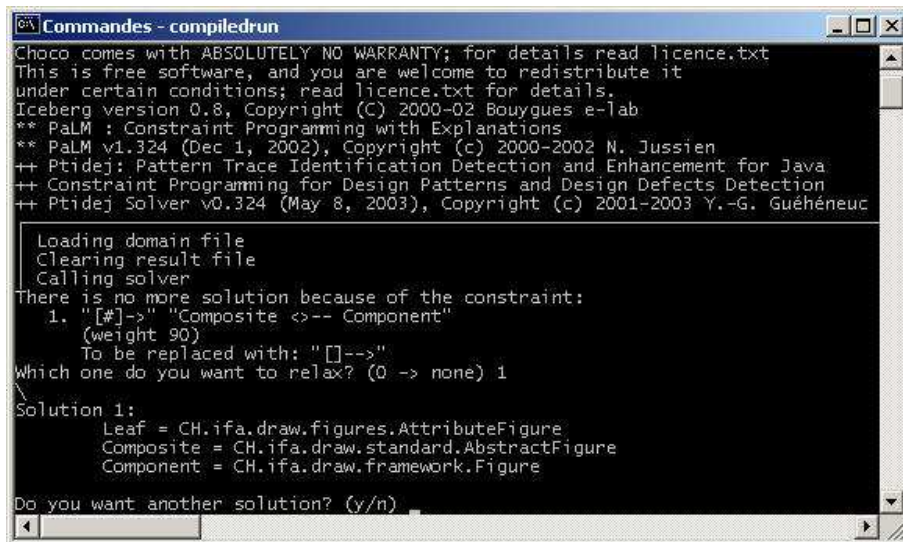
25.15.Component = CH.ifa.draw.standard.AbstractFigure
25.15.Composite = CH.ifa.draw.figures.AttributeFigure
25.15.Leaf-1 = CH.ifa.draw.figures.PolyLineFigure
25.15.Leaf-2 = CH.ifa.draw.standard.DecoratorFigure
25.15.Leaf-3 = CH.ifa.draw.standard.CompositeFigure
25.15.CompositeRoot = CH.ifa.draw.standard.AbstractFigure

...

```

□

FIG. 8.2 – Reprise de la recherche après rétraction de la contrainte de composition du système de contraintes.



```

Commandes - compiledrun
Choco comes with ABSOLUTELY NO WARRANTY; for details read licence.txt
This is free software, and you are welcome to redistribute it
under certain conditions; read licence.txt for details.
Iceberg version 0.8, Copyright (C) 2000-02 Bouygues e-lab
** PaLM : Constraint Programming with Explanations
** PaLM v1.324 (Dec 1, 2002), Copyright (c) 2000-2002 N. Jussien
++ Ptidej: Pattern Trace Identification Detection and Enhancement for Java
++ Constraint Programming for Design Patterns and Design Defects Detection
++ Ptidej Solver v0.324 (May 8, 2003), Copyright (c) 2001-2003 Y.-G. Guéhéneuc

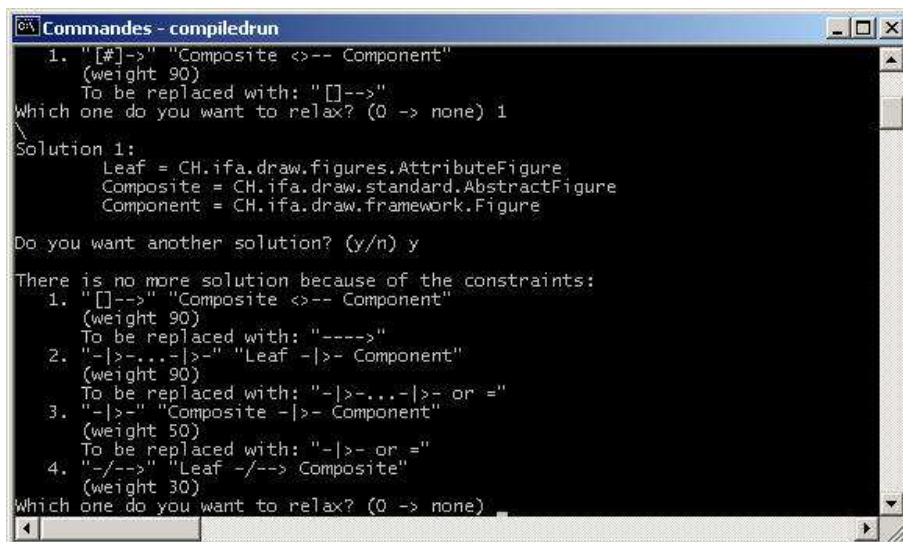
Loading domain file
Clearing result file
Calling solver
There is no more solution because of the constraint:
1. "[#]-->" "Composite <-- Component"
   (weight 90)
   To be replaced with: "[ ]-->"
Which one do you want to relax? (0 -> none) 1

Solution 1:
  Leaf = CH.ifa.draw.figures.AttributeFigure
  Composite = CH.ifa.draw.standard.AbstractFigure
  Component = CH.ifa.draw.framework.Figure

Do you want another solution? (y/n)

```

FIG. 8.3 – Système de contraintes en contradiction, le mainteneur doit retirer une autre contrainte.



```

Commandes - compiledrun
1. "[#]-->" "Composite <-- Component"
   (weight 90)
   To be replaced with: "[ ]-->"
Which one do you want to relax? (0 -> none) 1

Solution 1:
  Leaf = CH.ifa.draw.figures.AttributeFigure
  Composite = CH.ifa.draw.standard.AbstractFigure
  Component = CH.ifa.draw.framework.Figure

Do you want another solution? (y/n) y

There is no more solution because of the constraints:
1. "[ ]-->" "Composite <-- Component"
   (weight 90)
   To be replaced with: "---->"
2. "-|>...-|>-" "Leaf -|>- Component"
   (weight 90)
   To be replaced with: "-|>...-|>- or ="
3. "-|>-" "Composite -|>- Component"
   (weight 50)
   To be replaced with: "-|>- or ="
4. "-/--->" "Leaf -/---> Composite"
   (weight 30)
Which one do you want to relax? (0 -> none)

```

FIG. 8.4 – Recherche des solutions complètes avec le solveur automatique.



```

C:\>Commandes - compiledrun
increasing memory size by 2^6 and 2^6
-- CLAIRe run-time library v 3.2.52 [os: ntv, C++:MS VC++ ] --
Choco version 13.024, Copyright (C) 1999-2002 F. Laburthe
Choco comes with ABSOLUTELY NO WARRANTY; for details read licence.txt
This is free software, and you are welcome to redistribute it
under certain conditions; read licence.txt for details.
Iceberg version 0.8, Copyright (C) 2000-02 Bouygues e-lab
** PaLM : Constraint Programming with Explanations
** PaLM v1.324 (Dec 1, 2002), Copyright (c) 2000-2002 N. Jussien
++ Ptidej: Pattern Trace Identification Detection and Enhancement for Java
++ Constraint Programming for Design Patterns and Design Defects Detection
++ Ptidej Solver v0.324 (May 8, 2003), Copyright (c) 2001-2003 Y.-G. Guéhéneuc

Loading domain file
Clearing result file
Calling solver
Computing solutions (combinatorial automatic)

Soft constraints: 1/4, combination: 1/1
Solutions with all constraints
Trying weaker constraint:
  Solution without constraint:
    "-|>-" : "Composite -|>- Component"
  Solution with constraint:
    "-|>- or =" : "Composite -|>- Component"
Trying weaker constraint:
  Solution without constraint:
    "[#]>-" : "Composite <>-- Component"
  Solution with constraint:
    "[ ]>-" : "Composite <>-- Component"
Trying weaker constraint:
  Solution without constraint:
    "[ ]>-" : "Composite <>-- Component"
  Solution with constraint:
    "---->" : "Composite <>-- Component"
Trying weaker constraint:
  Solution without constraint:
    "---->" : "Composite <>-- Component"
  Solution with constraint:
    "-k-->" : "Composite <>-- Component"
Computed in 42942 ms.

Soft constraints: 2/4, combination: 1/3
Solution without constraint:
  "-/-->" : "Leaf -/--> Composite"
  
```

FIG. 8.5 – Recherche des solutions approchées avec le solveur automatique.

```

C:\Documents and Settings\Yann\Work\Ptidej Solver 3\Ptidej>

Solution without constraint:
"[]"--> : "Composite <-- Component"
Solution with constraint:
"---->" : "Composite <-- Component"
Trying weaker constraint:
Solution without constraint:
"---->" : "Composite <-- Component"
Solution with constraint:
"-k-->" : "Composite <-- Component"
Computed in 1613 ms.

Soft constraints: 3/4, combination: 3/3
Solution without constraints
"-|>-" : "Composite -|>- Component"
"-/-->" : "Component -/--> Leaf"
Trying weaker constraint:
Solution without constraint:
"[#]->" : "Composite <-- Component"
Solution with constraint:
"[]"--> : "Composite <-- Component"
Trying weaker constraint:
Solution without constraint:
"[]"--> : "Composite <-- Component"
Solution with constraint:
"---->" : "Composite <-- Component"
Trying weaker constraint:
Solution without constraint:
"---->" : "Composite <-- Component"
Solution with constraint:
"-k-->" : "Composite <-- Component"
Computed in 37724 ms.

Soft constraints: 4/4, combination: 1/1
Solution without constraints
"-|>-" : "Composite -|>- Component"
"-/-->" : "Component -/--> Leaf"
"-/-->" : "Leaf -/--> Composite"
Trying weaker constraint:
Solution without constraint:
"[#]->" : "Composite <-- Component"
Solution with constraint:
"[]"--> : "Composite <-- Component"
Trying weaker constraint:
Solution without constraint:
"[]"--> : "Composite <-- Component"
Solution with constraint:
"---->" : "Composite <-- Component"
Trying weaker constraint:
Solution without constraint:
"---->" : "Composite <-- Component"
Solution with constraint:
"-k-->" : "Composite <-- Component"
Computed in 170 ms.

C:\Documents and Settings\Yann\Work\Ptidej Solver 3\Ptidej>

```

8.4 Algorithmes de traçabilité des motifs

NOUS présentons l'algorithme qui instancie, au niveau **conception**, les micro-architectures identifiées par le solveur de contraintes et qui garantit la traçabilité entre les niveaux **idiomatique** et **conception**.

D'abord, nous créons un modèle du programme au niveau **conception** avec le modèle du programme au niveau **idiomatique**, l'instanciation du constituant `DesignLevelModel` est réalisée comme suit :

```
final String path = "<path of JHotDraw classfiles>";
final IdiomLevelModel idiomLevelModel = new IdiomLevelModel("JHotDraw");
idiomLevelModel.build(
    TypeLoader.loadSubtypesFromDir(null, path, ".class"));

final DesignLevelModel designLevelModel = new DesignLevelModel(idiomLevelModel);
```

Puis, nous construisons et ajoutons au modèle du programme au niveau **conception** les micro-architectures identifiées comme similaires à des motifs de conception par l'analyse du fichier contenant les solutions au problème de satisfaction de contraintes. Les solutions au problème de satisfaction de contraintes sont réifiées comme des instances de la classe `Solution` :

```
Solution[] solutions = <Solutions of the constraint satisfaction problem>;
for (int i = 0; i < solutions.length; i++) {
    final MicroArchitecture microArchitecture = this.createMicroArchitecture(solutions[i]);
    designLevelModel.addMicroArchitecture(microArchitecture);
}
```

La méthode de création et d'instanciation des micro-architectures est présentée sur la figure 8.6 page 269. Le principe de cette méthode est de créer un clone des entités du modèle du programme au niveau **idiomatique** qui jouent un rôle dans la micro-architecture et d'attacher ces clones à la micro-architecture pour assurer la traçabilité.

D'abord, nous créons une instance de la classe `MicroArchitecture` pour représenter la micro-architecture similaire à un motif de conception, lignes 5–9, avec le nom du motif identifié, le numéro de cette solution particulière et l'écart de cette micro-architecture par rapport au modèle du motif de conception.

Ensuite, pour chaque rôle identifié dans la solution, nous commençons la session de clonage des entités jouant un rôle dans la micro-architecture, lignes 11–19. Puis, nous réalisons le clonage et ajoutons les entités clonées à la micro-architecture, lignes 20–32. Enfin, nous terminons la session de clonage des entités, lignes 33–43. L'instance de la classe `DesignLevelModel` ainsi créée et augmentée des micro-architectures similaires à des motifs de conception représente le modèle du programme au niveau **conception**. Ce modèle nous garantit la traçabilité entre les niveaux **idiomatique** et **conception**.

Code source 8.6 – Algorithme simplifié d’instanciation des micro-architectures.

```

1  public MicroArchitecture createMicroArchitecture(
2      final Solution solution,
3      final IdiomLevelModel idiomLevelModel) {
4
5      final MicroArchitecture microArchitecture =
6          new MicroArchitecture(
7              solution.getMotifName(),
8              solution.getSolutionNumber(),
9              solution.getSolutionPercentage());
10
11     Iterator iterator = solution.getRoles().iterator();
12     while (iterator.hasNext()) {
13         final Role role =
14             (Role) iterator.next();
15         final String roleName = role.getName();
16         final String roleValue = role.getValue();
17
18         idiomLevelModel.getActor(roleValue).startCloneSession();
19     }
20     iterator = solution.getRoles().iterator();
21     while (iterator.hasNext()) {
22         final Role role =
23             (Role) iterator.next();
24         final String roleName = role.getName();
25         final String roleValue = role.getValue();
26
27         final Entity entity =
28             (Entity) idiomLevelModel.getActor(roleValue);
29
30         entity.performCloneSession();
31         microArchitecture.addActor((Entity) entity.getClone());
32     }
33     iterator = solution.getRoles().iterator();
34     while (iterator.hasNext()) {
35         final Role role =
36             (Role) iterator.next();
37         final String roleName = role.getName();
38         final String roleValue = role.getValue();
39
40         final Entity entity =
41             (Entity) idiomLevelModel.getActor(roleValue);
42         entity.endCloneSession();
43     }
44     ...
45
46     return microArchitecture;
47 }

```

□

Nous obtenons ainsi le modèle du programme JHOTDRAW au niveau **conception** et les micro-architectures similaires au motif de conception **Composite** présenté sur la figure 1.9 page 30 en quatre phases avec PTIDEJ :

1. Une phase de construction du modèle partiel du programme au niveau **idiomatique** par le chargement des unités de compilation adéquates dans PTIDEJ, avec les algorithmes présentés dans la section 6.1 page 212 ;
2. Une phase de construction du modèle complet du programme au niveau **idiomatique** après analyses dynamiques et intégration des résultats au modèle partiel, avec les algorithmes présentés dans la section 6.2 page 216 ;
3. Une phase d'identification des micro-architectures similaires à un motif de conception par la génération d'un problème de satisfaction de contraintes et sa résolution avec PTIDEJ SOLVER, comme présenté dans les sections 8.2 page 256 et 8.3 page 263 ;
4. Une phase de construction du modèle du programme au niveau **conception** par le chargement et l'intégration des résultats de l'identification au modèle du programme, avec les algorithmes présentés dans cette section.

Conclusion. Nous avons présenté l'implantation de l'algorithme pour construire le modèle d'un programme au niveau **conception** et pour garantir la traçabilité des micro-architectures similaires à un motif de conception entre les niveaux **idiomatique** et **conception**. Nous dressons maintenant un bilan de nos implantations.

Bilan

Nous avons présenté l'utilisation du métamodèle PADL et des stratégies de recherche proposées par le solveur de contraintes PTIDEJ SOLVER pour identifier dans le modèle d'un programme au niveau **idiomatique** les micro-architectures similaires au modèle d'un motif de conception et pour construire un modèle du programme au niveau **conception**.

D'abord, nous avons montré comment décrire et instancier un modèle abstrait d'un motif de conception avec les constituants proposés par le métamodèle PADL.

Ensuite, nous avons implanté deux visiteurs pour obtenir automatiquement un système de contraintes du modèle abstrait d'un motif de conception et le domaine des variables du modèle d'un programme au niveau **idiomatique**.

Puis, nous avons détaillé l'utilisation de PTIDEJ SOLVER pour résoudre le problème de satisfaction de contraintes et identifier les micro-architectures similaires au modèle d'un motif.

Enfin, nous avons décrit l'algorithme pour construire un modèle du programme au niveau **conception** dans lequel les micro-architectures similaires à un motif de conception sont représentées.

Conclusion. Ces implantations nous permettent de garantir la traçabilité des motifs de conception et de réaliser la seconde phase de l'identification des motifs de conception : l'obtention du modèle d'un programme au niveau **conception**, comme présenté sur les figures 1.7(c) et 1.7(d) page 24. Nous dressons maintenant un bilan général de nos implantations avant de conclure sur la traçabilité des motifs de conception.

Bilan de la mise en œuvre de la traçabilité des motifs avec PTIDEJ

Nous avons présenté nos implantations des modèles et des algorithmes pour garantir la traçabilité des motifs interclasses et de conception entre les niveaux implémentation, idiomatique et conception.

Les métamodèles pour décrire les niveaux idiomatique et conception et les motifs de conception sont implantés par un unique métamodèle, PADL. Ce métamodèle est une extension du métamodèle PDL présenté dans [Albin-Amiot, 2003] pour décrire les motifs de conception.

Il nous permet de décrire le modèle d'un programme au niveau idiomatique avec les motifs interclasses et au niveau conception avec les micro-architectures similaires à des motifs de conception.

Nous avons brièvement décrit une bibliothèque graphique pour visualiser et interagir avec les modèles issus du métamodèle PADL. Nous avons donné des exemples de leur utilisation pour modéliser et visualiser un sous-ensemble du programme JHOTDRAW.

Ensuite, nous avons décrit les outils d'analyses statiques et d'analyses dynamiques pour identifier les motifs interclasses au niveau implémentation avec leurs propriétés de durée de vie, d'exclusivité, de multiplicité et de site d'invocation.

INTROSPECTOR est un outil d'analyses statiques intégré au métamodèle PADL. Il nous permet de calculer les valeurs des propriétés de multiplicité et de site d'invocation et de construire un modèle statique d'un programme au niveau implémentation.

Chaque constituant du métamodèle PADL définit une méthode statique `recognize()` pour identifier les constituants qui lui correspondent dans le modèle d'un programme au niveau implémentation et pour s'instancier dans le modèle du programme au niveau idiomatique en construction.

CAFFEINE est un outil générique d'analyses des traces d'exécution des programmes JAVA avec des règles PROLOG. Il nous permet de calculer les valeurs des propriétés de durée de vie et d'exclusivité avec les règles PROLOG dédiées, `checkLTProperty/3` et

`checkEXProperty/3` et de raffiner le modèle statique du programme en son modèle au niveau **idiomatique**.

Nous avons aussi présenté la mise en œuvre de ces outils pour construire un modèle du programme JHOTDRAW au niveau **idiomatique**.

Puis, nous avons présenté l'implantation de référence de la programmation par contraintes avec explications, PALM. Nous avons montré que PALM étend la plate-forme de contraintes CHOCO.

Nous étendons PALM pour construire le solveur PTIDEJ SOLVER dédié à l'identification des micro-architectures similaires à un motif de conception dans le modèle d'un programme au niveau **idiomatique**.

PTIDEJ SOLVER propose des classes pour représenter les problèmes de satisfaction de contraintes, les variables, le domaine des variables et les contraintes, respectivement, `PtidejProblem`, `PtidejVariable`, `PtidejBinConstraint`, `PtidejLargeConstraint`, et `Entity`.

Nous avons détaillé l'implantation de la bibliothèque de contraintes et des classes `EqualConstraint`, `NotEqualConstraint` et `PropertyTypeConstraint` pour décrire un motif de conception comme un système de contraintes et les stratégies de recherche interactives et automatiques par relaxation des contraintes et des problèmes, `interactiveSolve()` et `combinatorialAutomaticSolve()`.

Enfin, nous avons décrit le processus de modélisation d'un motif de conception avec le métamodèle PADL pour construire un modèle abstrait du motif et nous l'avons illustré avec le motif de conception `Composite`.

Nous avons appliqué le patron de conception `Visiteur` au métamodèle PADL et implémenté deux visiteurs pour générer automatiquement un domaine d'un modèle d'un programme et un système de contraintes d'un modèle d'un motif de conception.

Nous avons montré l'utilisation de PTIDEJ SOLVER pour identifier les micro-architectures similaires au motif de conception `Composite` dans le modèle du programme JHOTDRAW au niveau **idiomatique** avec les stratégies de recherche interactive et combinatoire et les algorithmes de construction de son modèle au niveau **conception**.

Conclusion. Nous avons présenté des modèles, des algorithmes, et leurs implantations pour garantir la traçabilité des motifs interclasses et de conception entre les niveaux **implémentation**, **idiomatique** et **conception**. Nous concluons maintenant sur les travaux présentés.

Quatrième partie

Conclusion et perspectives

Conclusion

Nous avons proposé et décrit des modèles et des algorithmes pour garantir la traçabilité des motifs de conception entre les phases d'implantation et de rétroconception des programmes par l'identification semi-automatique des micro-architectures similaires à ces motifs dans le code source des programmes.

Ces modèles et ces algorithmes forment un cadre pour la traçabilité des motifs de conception. Nous avons implanté ce cadre JAVA en une suite d'outils intégrée à l'environnement de développement ECLIPSE, PTIDEJ.

Ce cadre nous a conduit à :

1. Décomposer le problème de la traçabilité des motifs de conception en deux sous-problèmes distincts : d'une part, l'identification automatique et la traçabilité des motifs interclasses entre les niveaux **implémentation** et **idiomatique** et, d'autre part, l'identification, l'explication et la traçabilité des motifs de conception entre les niveaux **idiomatique** et **conception**.
2. Proposer un nouveau niveau d'abstraction pour décrire les programmes. Nous avons introduit le niveau d'abstraction **idiomatique** entre les niveaux **implémentation** et **conception** dans lesquels sont représentés le code source des programmes et les motifs de conception utilisés lors de leur implantation. Au niveau **idiomatique**, les programmes sont décrits par des diagrammes de classes.
3. Préciser les définitions des motifs interclasses **Association**, **Agrégation** et **Composition** pour construire automatiquement les modèles des programmes au niveau **idiomatique** avec ces définitions et avec leurs modèles au niveau **implémentation**. Nous avons caractérisé les motifs interclasses avec quatre propriétés minimales : durée de vie, exclusivité, multiplicité et site d'invocation ; nous les avons identifiés avec des analyses statiques et dynamiques [Guéhéneuc *et al.*, 2002a ; Guéhéneuc *et al.*, 2002b ; Guéhéneuc *et al.*, 2002c ; Guéhéneuc et Albin-Amiot, 2003].
4. Étudier les motifs de conception pour identifier les micro-architectures qui leur sont similaires dans les modèles des programmes au niveau **idiomatique** et pour construire des modèles de ces programmes au niveau **conception**. Nous avons montré que l'identification des motifs de conception doit être interactive, dynamique et explicative. Nous avons proposé de modéliser les motifs de conception comme des systèmes de

contraintes et nous avons utilisé la programmation par contraintes avec explications pour identifier semi-automatiquement leurs formes complètes et approchées [Guéhéneuc et Jussien, 2001a ; Guéhéneuc et Jussien, 2001b].

5. Développer une suite d'outils, PTIDEJ, qui implante les modèles et les algorithmes décrits [Albin-Amiot *et al.*, 2001 ; Guéhéneuc, 2002]. La suite PTIDEJ inclut le métamodèle PADL, dérivé du métamodèle PDL [Albin-Amiot *et al.*, 2002 ; Albin-Amiot et Guéhéneuc, 2001a ; Albin-Amiot et Guéhéneuc, 2001c] ; des outils d'analyses statiques et dynamiques, INTROSPECTOR et CAFFEINE [Guéhéneuc *et al.*, 2002c] ; et un solveur de contraintes, PTIDEJ SOLVER [Guéhéneuc et Jussien, 2001a ; Guéhéneuc et Jussien, 2001b], dérivé du solveur de contraintes avec explications de référence PALM.
6. Détailler l'utilisation de la suite d'outils PTIDEJ pour garantir la traçabilité du motif de conception **Composite** dans le programme JHOTDRAW entre les niveaux **implémentation** et **conception**.

Ce cadre nous permet maintenant d'étudier, de qualifier et de quantifier les bénéfices de l'utilisation des patrons de conception et de la traçabilité des motifs de conception pour la compréhension des programmes à objets lors de la maintenance.

Perspectives

Nous envisageons d'utiliser ce cadre pour étudier la traçabilité des motifs de conception, la compréhension des programmes à objets et de l'étendre à d'autres types de motifs de conception, à d'autres langages de programmation.

10.1 Utilisations du cadre

Nous avons validé nos modèles et les algorithmes d'analyses statiques, d'analyses dynamiques et de résolution des problèmes de satisfaction de contraintes avec explications par la traçabilité du motif de conception **Composite** dans le programme JHOTDRAW.

Nous considérons cette validation insuffisante pour prouver l'intérêt de nos travaux et prétendre qu'ils sont intéressants pour des projets de maintenance réels et qu'ils supportent le passage à l'échelle.

Nous aimerions évaluer nos travaux, d'une part, théoriquement pour quantifier le support cognitif réellement fourni [Walenstein, 2002] et prouver les algorithmes présentés, d'autre part, expérimentalement en les comparant à d'autres techniques de rétroconception avec plusieurs groupes de mainteneurs : analyses manuelles, outils académiques et industriels, tels MOOSE, RIGI, RATIONAL ROSE, etc.

Cette évaluation doit nous permettre de qualifier l'intérêt pratique de la traçabilité des motifs de conception pour faciliter la phase de rétroconception des programmes et pour contribuer à leur compréhension.

Aussi, elle doit nous permettre de quantifier les degrés de dégradations tolérés pour chaque motif de conception et d'offrir des stratégies de recherche interactives et automatiques plus pertinentes.

Par ailleurs, nous sommes convaincus que la traçabilité des motifs de mauvaise conception et des défauts de conception permettrait d'aider les mainteneurs à isoler rapidement les défauts dans l'architecture des programmes et à les corriger rapidement.

Les défauts de conception sont d'une part des motifs représentatifs de mauvais choix architecturaux [Pal et Minsky, 1996 ; Brown *et al.*, 1998] ou des formes approchées de motifs de conception dont les structures pourraient être améliorées.

Nous avons commencé à étudier l'identification et la traçabilité des motifs de mauvaise conception et des défauts de conception, qui connotent des micro-architectures avec de mauvaises caractéristiques de qualité.

Nous avons proposé une première taxonomie des défauts de conception et nous avons expérimenté leur traçabilité avec les modèles et les techniques proposés pour les motifs de conception [Guéhéneuc et Albin-Amiot, 2001].

Les premiers résultats sont encourageants mais la modélisation des défauts de conception et l'interprétation des micro-architectures identifiées doivent être développées et connectées à des mécanismes de restructuration² [Johnson et Opdyke, 1993 ; Opdyke, 1997].

De plus, ce cadre nous permet maintenant d'étudier l'impact de la traçabilité des motifs de conception sur les caractéristiques de qualité des programmes, comme la réutilisabilité, la maintenabilité, etc.

Nous pouvons enrichir ce cadre avec des modèles de qualité pour quantifier les bénéfices de la traçabilité et critiquer les travaux sur la pertinence des motifs de conception, comme [Reiðing, 2001 ; Wendorff, 2001].

Ensuite, nous voulons explorer les liens entre nos travaux, qui corrélerent les langages de programmation et les langages de conception, et des travaux inverses, telle la conception de programmes dirigée par les modèles³ [Bézivin et Ploquin, 2001 ; Miller et Mukerji, 2001].

Nous avons cherché à construire des modèles de l'architecture de programmes depuis leur code source ; la conception de programmes par les modèles cherche à générer le code source des programmes depuis les modèles de leur architecture.

Ces deux approches semblent complémentaires et nous aimerions étudier la faisabilité de les connecter l'une à l'autre pour assurer l'aller-retour entre implantation et conception, quels que soient les langages de programmation et de conception utilisés.

Enfin, nous aimerions améliorer l'implantation de notre suite d'outils pour la rendre disponible lors de l'enseignement de la conception et de la programmation par objets avec les patrons de conception.

En particulier, nous pensons que les définitions données aux motifs interclasses **Association**, **Agrégation**, et **Composition** faciliteraient l'explication et l'utilisation de ces notions dans les cours de conception par objets.

²Les mécanismes de restructuration s'appellent *refactorings* en anglais.

³La conception dirigée par les modèles s'appelle *model driven architecture* (MDA) en anglais.

10.2 Extensions du cadre

Nous aimerions accroître la généralité de nos modèles et de nos algorithmes en les appliquant à d'autres types de motifs de conception, tels les motifs de conception comportementaux et créateurs⁴.

La généralisation de nos travaux aux motifs de conception comportementaux et créateurs nécessitent d'approfondir la modélisation du comportement des programmes, avec des diagrammes d'interactions ou avec des protocoles comportementaux.

Elle nécessite aussi des algorithmes d'analyses des flots et des données pour construire des modèles représentatifs du comportement des programmes dans lesquels nous voulons identifier ces motifs comportementaux et générateurs.

Nous avons commencé à étudier l'utilisation des protocoles comportementaux, machines à états finis dans lesquelles les transitions représentent des appels de méthodes, pour modéliser programmes et motifs.

Nous avons développé des algorithmes d'analyses des flots et des données pour extraire un protocole comportemental du code source d'un programme et pour le comparer avec un protocole donné [Farías *et al.*, 2002 ; Farías et Guéhéneuc, 2003].

Nous aimerions étendre ces algorithmes à la construction automatique de modèles comportementaux de programmes, à l'identification et à la traçabilité des motifs de conception comportementaux et créateurs.

Par ailleurs, ce cadre pour la traçabilité des motifs de conception nous permet d'étudier, de qualifier et de quantifier la composition des motifs de conception pour améliorer l'identification des motifs.

Certains motifs sont souvent appliqués simultanément, par exemple les motifs **Usine abstraite** et **Singleton**. Cette application conjointe devrait nous permettre d'améliorer l'identification des motifs et la compréhension des problèmes de conception résolus.

De plus, nos algorithmes n'utilisent pas la présence des méthodes pour utiliser les appels polymorphiques et la surcharge des méthodes lors de l'identification des motifs. D'autres travaux [Wuyts, 1998] ont montré l'intérêt de leur utilisation.

Nous aimerions développer nos modèles, nos algorithmes et les contraintes pour permettre un raisonnement précis sur la surcharge des méthodes et les appels polymorphiques, comme [Arévalo et Mens, 2002].

Aussi, nous aimerions développer ce cadre pour d'autres langages de programmation, en particulier C++ et SMALLTALK, qui présentent des caractéristiques distinctes de JAVA, comme l'héritage multiple et le typage dynamique.

⁴Les motifs de conception comportementaux et créateurs sont appelés *behavioral* et *creational*, respectivement, en anglais.

Ce cadre devrait alors nous permettre de mesurer l'impact des langages de programmation sur l'utilisation des motifs de conception et d'isoler des caractéristiques communes et distinctives des langages de programmation pour les motifs.

Ensuite, nous pensons que l'amélioration de l'interface homme-machine de nos outils est primordiale pour faciliter efficacement la compréhension des programmes, en présentant les informations de manière simple et constante.

Nos outils doivent inclure des algorithmes de mise en page automatique dédiés aux diagrammes de classes proches de la notation UML [Seemann, 1997 ; Schauer et Keller, 1999 ; Eichelberger et von Gudenberg, 2002] et aux micro-architectures similaires à des motifs de conception.

Nous aimerions étudier des techniques novatrices de présentation de l'information, tels les algorithmes "ressorts", les matrices d'adjacence [Ghoniem et Fekete, 2002 ; Ghoniem et Fekete, 2003] ou la visualisation des métriques et leur impact sur la compréhension des mainteneurs.

Nous voudrions implanter un mécanisme de "pliage" pour reporter sur les interfaces les relations existantes entre leurs classes d'implantation pour réduire la complexité des modèles au niveau *conception* et pour présenter des modèles plus concis.

Enfin, nous pensons important d'étudier d'autres techniques de reconnaissance de motifs, en particulier les techniques utilisées dans les domaines de la vision et du son et de confronter les techniques pour la compréhension des programmes avec, par exemple, les lois de la Gestalt [Simons et Graham, 1999].

Cinquième partie

Annexes

Autres plans du mémoire

CE MÉMOIRE peut être lu de plusieurs façons. Si l'intérêt se porte sur le mémoire dans son ensemble, le plan linéaire présenté section 1.6 page 26 en résume les différentes parties et leurs articulations. Si l'intérêt se porte sur les réalisations pratiques, l'approche logicielle donne un aperçu des outils implantés, de leurs objectifs et de leurs interactions. Enfin, nous référençons les publications liées aux travaux présentés.

A.1 Approche logicielle

La figure A.6 page 286 présente l'articulation des logiciels implantés pour expérimenter les travaux présentés.

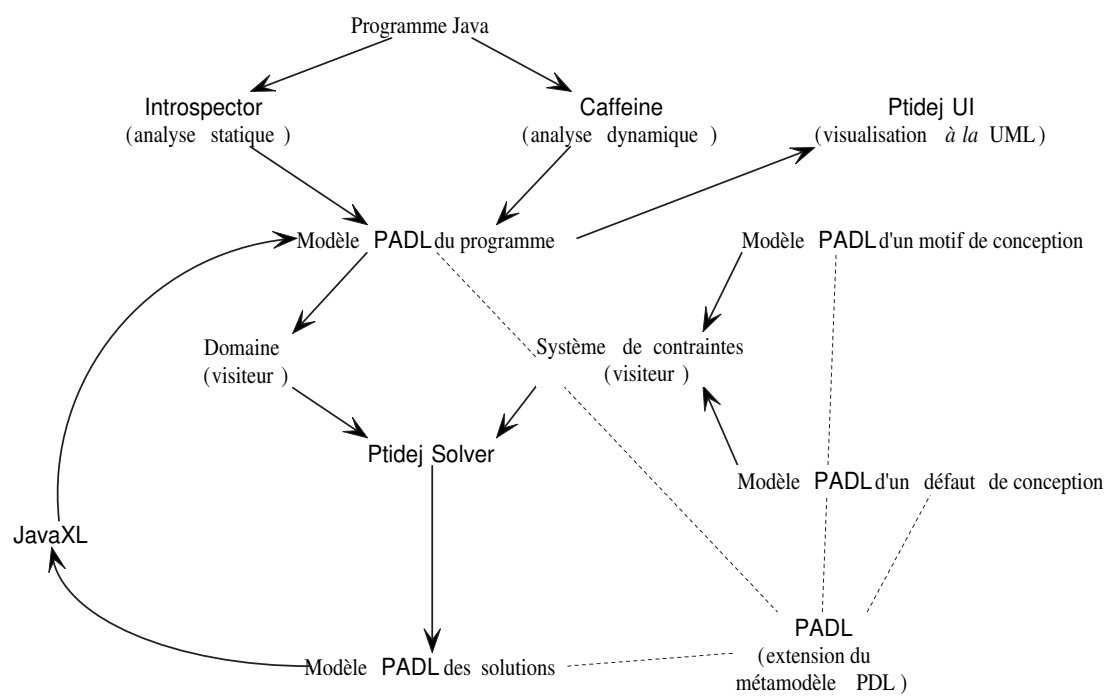
Nous obtenons un modèle du programme basé sur le métamodèle PADL, chapitre 5, par analyses statiques et dynamiques du code source d'un programme JAVA. Les analyses statiques sont réalisées par l'intermédiaire d'algorithmes spécialisés implantés dans le métamodèle PADL et mises en œuvre par l'intermédiaire de l'outil INTROSPECTOR, section 6.1. Les analyses dynamiques sont basées sur une bibliothèque spécialisée d'analyses mises en œuvre par l'intermédiaire de l'outil d'analyse dynamique CAFFEINE, section 6.2.

Parallèlement, les mainteneurs modélisent le motif de conception dont ils veulent identifier des micro-architecture similaires dans le modèle du programme avec les constituants du métamodèle PADL, section 8.1.

Nous générons automatiquement le système de contraintes associé avec le modèle de la solution et le domaine associé avec le modèle du programme, section 8.2, avec ces modèles et des algorithmes de visite spécialisés. Le système de contraintes est résolu interactivement ou automatiquement par relaxations successives des contraintes et du problème par le solveur de contraintes spécialisés PTIDEJ SOLVER, section 8.3. La résolution du système de contraintes nous permet d'obtenir les modèles des formes approchées de la solution du motif de conception, basés sur le métamodèle PADL, section 8.4.

Les modèles du programme, du motif de conception, et des micro-architectures identifiées sont visualisées par l'intermédiaire de PTIDEJ UI, section 5.3.

FIG. A.6 – Approche logicielle.



□

A.2 Références

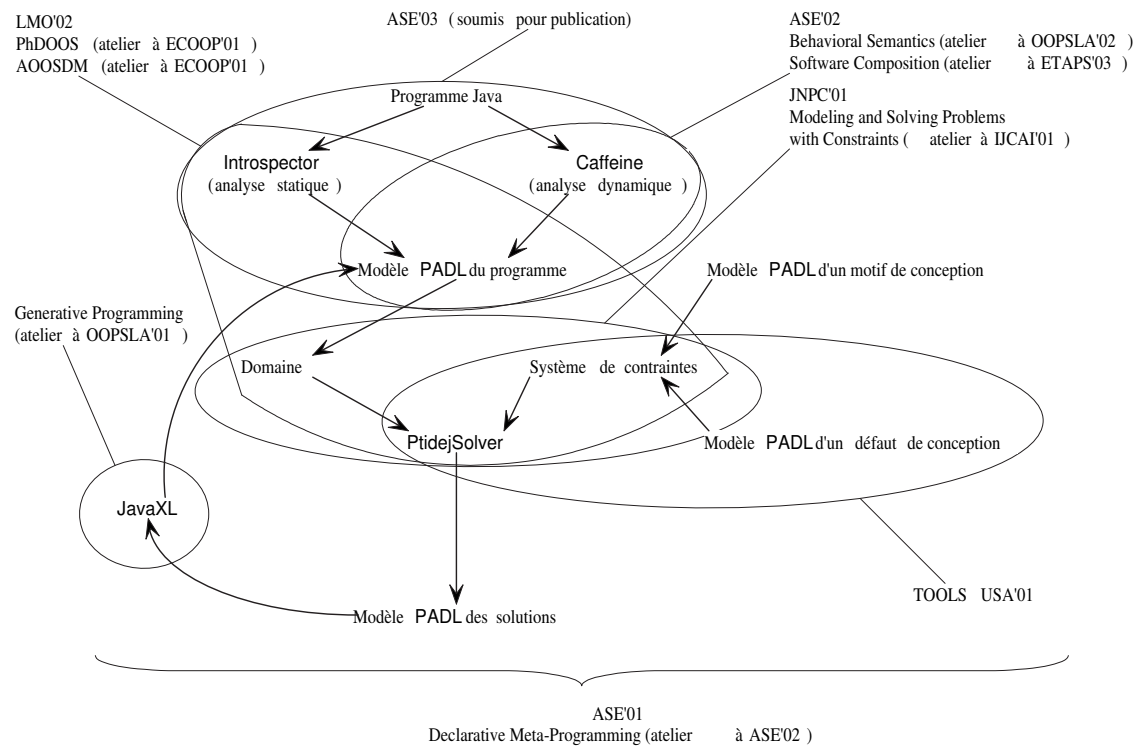
Les travaux présentés ont fait l'objet de différentes publications, articles ou rapports techniques, comme montré sur la figure A.7 page 288 :

- le métamodèle PDL et son utilisation pour décrire des modèles de programmes et de motifs de conception au niveau idiomatique [Albin-Amiot et Guéhéneuc, 2001a ; Albin-Amiot et Guéhéneuc, 2001c ; Albin-Amiot *et al.*, 2002] ;
- un synthèse de l'état de l'art sur les définitions des motifs interclasses **Association**, **Agrégation** et **Composition** et un proposition d'algorithmes d'identification [Guéhéneuc *et al.*, 2002a] ;
- la définition et l'identification des motifs interclasses pour combler le fossé entre langages de programmation et langages de modélisation [Guéhéneuc et Albin-Amiot, 2003] ;
- l'utilisation de la programmation par contraintes avec explications pour identifier les micro-architectures similaires à un motif de conception [Guéhéneuc et Jussien, 2001a ; Guéhéneuc et Jussien, 2001b] ;
- un outil générique d'analyses de la trace d'un programme à l'exécution et son application pour calculer les valeurs des propriétés des motifs interclasses [Guéhéneuc *et al.*, 2002b ; Guéhéneuc *et al.*, 2002c] ;
- un synthèse de la modélisation, de l'application et de l'identification des motifs de conception [Albin-Amiot *et al.*, 2001 ; Guéhéneuc, 2002].

D'autres aspects de nos recherches qui ne sont pas abordés ont été publiés :

- l'application de nos travaux à l'identification et à la correction automatique des défauts de conception dans les architectures de programmes à objets [Guéhéneuc et Albin-Amiot, 2001] ;
- l'application de motifs de conception dans les architectures de programmes à objets par génération ou transformation de code source [Albin-Amiot et Guéhéneuc, 2001b] ;
- la modélisation du comportement d'un programme avec des protocoles pour, à terme, modéliser les motifs de conception comportementaux [Farías *et al.*, 2002 ; Farías et Guéhéneuc, 2003].

FIG. A.7 – Références.



□

Détails sur JHotDRAW

JHotDRAW est programme de dessin vectoriel développé originellement par Erich Gamma et Thomas Eggenschwiler. C'est maintenant un logiciel libre, partagé sur SOURCEFORGE.NET. Nous utilisons JHotDRAW pour plusieurs raisons :

- sa conception et son implantation utilisent de nombreux patrons de conception ;
- sa documentation fournit une liste des patrons de conception utilisés ;
- sa taille est raisonnable : 155 classes réparties dans 11 paquets pour un total de 16 015 lignes de code commenté ;
- il est utilisé et étendu dans de nombreux programmes [Kaiser, 2001] ;
- il a été conçu et implanté indépendamment de nos recherches, il constitue donc un cas réel d'utilisation ;
- il est disponible librement, accessible depuis <http://members.pingnet.ch/gamma/JHD-5.1.zip> ou <http://jhotdraw.sourceforge.net/>.

Détails sur PTIDEJ

La suite d'outils PTIDEJ a été implantée en environ 700 heures :

- nous avons participé avec Hervé Albin-Amiot au développement du métamodèle PDL que nous avons ensuite personnellement étendu pour obtenir le métamodèle PADL, en particulier par l'ajunction d'un mécanisme de visite des constituants ;
- nous avons seul implanté la bibliothèque graphique PTIDEJ UI pour visualiser simplement les modèles décrits avec le métamodèle PADL ;
- nous avons participé avec Hervé Albin-Amiot au développement de INTROSPECTOR, qui est étroitement intégré aux métamodèles PDL et PADL ;
- nous avons seul mis en œuvre l'outil d'analyse dynamique CAFFEINE. Rémi Douence et Jacques Noyé nous ont apportés leur aide pour l'écriture des prédicats PROLOG pour calculer les valeurs des propriétés de durée de vie et d'exclusivité ;
- nous avons étendu, avec les précieux conseils de Narendra Jussien, le système de contraintes avec explications PALM pour obtenir l'outil PTIDEJ SOLVER et nous avons développé seul la bibliothèque de contraintes PTIDEJ LIBRARY ;
- nous avons seul implanté l'outil PTIDEJ, façade à la suite d'outils PTIDEJ, et les algorithmes de génération des problèmes de satisfactions de contraintes, de visualisation des micro-architectures et de traçabilité.

Cette suite d'outils représente :

- en JAVA : 20 projets, 132 paquetages, 669 classes, 72 interfaces, 79 330 lignes ;
- en CLAIRE : 41 fichiers, 45 classes, 235 méthodes, 7 835 lignes ;
- en PROLOG : 6 fichiers, 76 prédicats, 524 lignes.

Bibliographie

Nous avons choisi d'utiliser les langues des documents référencés pour les entrées de la bibliographie ; ainsi, une entrée référençant un document en français est en français, une entrée référençant un document en anglais est en anglais. Cependant, la référence à l'ouvrage est toujours donnée en français quelque soit la langue du document référencé : [Allen et Garlan, 1997].

- [Agerbo et Cornils, 1998] cité page 16
Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In Craig Chambers, editor, *proceedings of the 13th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 134–143. ACM Press, October 1998.
- [Albin-Amiot *et al.*, 2001] cité pages 278, 287
Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, November 2001.
- [Albin-Amiot *et al.*, 2002] cité pages 158, 278, 287
Hervé Albin-Amiot, Pierre Cointe et Yann-Gaël Guéhéneuc. Un méta-modèle pour coupler application et détection des design patterns. Michel Dao et Marianne Huchard, éditeurs, *actes du 8^e colloque Langages et Modèles à Objets*, volume 8, numéro 1-2/2002 de *RSTI – L'objet*, pages 41–58. Hermès Science Publications, janvier 2002.
- [Albin-Amiot et Guéhéneuc, 2001a] cité pages 278, 287
Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Design patterns: A round-trip. In Gilles Ardourel, Michael Haupt, Jose Luis Herrero Agustin, Rainer Ruggaber, and Charles Suscheck, editors, *proceedings of the 11th ECOOP workshop for Ph.D. Students in Object-Oriented Systems*, June 2001.
- [Albin-Amiot et Guéhéneuc, 2001b] cité page 287
Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Design patterns application: Pure-generative approach vs. conservative-generative approach. In Krzysztof Czarnecki,

- editor, *proceedings of the 1st OOPSLA workshop on Generative Programming*. ACM Press, October 2001.
- [Albin-Amiot et Guéhéneuc, 2001c] cité pages 278, 287
Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, Pim Van Den Broek, Motoshi Saeki, Pavel Hruby, and Gerson Sunyé, editors, *proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.
- [Albin-Amiot, 2003] cité pages 8, 14, 22, 27, 31, 69, 80, 149, 180, 198, 199, 273
Hervé Albin-Amiot. *Idiomes et Patterns JAVA : Application à la Synthèse de Code et à la Détection*. Thèse de doctorat, université de Nantes, février 2003.
- [Alencar *et al.*, 1995] cité page 67
Paulo S. C. Alencar, Donald D. Cowan, Daniel Morales-Germán, Kurt J. Lichtner, Carlos José Pereira de Lucena, and Luis C.M. Nova. A formal approach to design pattern definition and application. Technical report CS-95-29, Computer Systems Group, University of Waterloo, June 1995.
- [Alencar *et al.*, 1996] cité page 67
Paulo S. C. Alencar, Donald D. Cowan, Thomas Kunz, and Carlos José Pereira de Lucena. A formal architectural design patterns-based approach to software understanding. In Hausi Müller and ?, editors, *proceedings of the 4th International Workshop on Program Comprehension*, pages 154–163. IEEE Computer Society Press, March 1996.
- [Alexander *et al.*, 1978] cité page 13
Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1st edition, August 1978. ISBN: 0-19-501919-9.
- [Allen et Garlan, 1997] cité pages 12, 293
Robert Allen and David Garlan. A formal basis for architectural connection. In Axel Van Lamsweerde, editor, *Transactions On Software Engineering and Methodology*, 6(3):213–249. ACM Press, July 1997.
- [André *et al.*, 2000] cité pages 49, 53
Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. An algebraic view of UML class diagrams. In Christophe Dony and Houari Sahraoui, editors, *proceedings of the 6th colloquium on Languages and Models with Objects*, pages 261–276. Hermès Science Publications, January 2000.
- [Anquetil et Lethbridge, 1998] cité page 69
Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names: A new file clustering criterion. In Kokichi Futatsugi and Richard Kemmerer, editors, *proceedings of the 20th International Conference on Software Engineering*, pages 84–93. IEEE Computer Society Press, May 1998.
- [Antoniol *et al.*, 1998] cité page 70
Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in

- object-oriented software. In Scott Tilley and Giuseppe Visaggio, editors, *proceedings of the 6th International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.
- [Antoniol *et al.*, 2001] cité pages 12, 16
Giuliano Antoniol, Bruno Caprile, Alessandra Potrich, and Paolo Tonella. Design-code traceability recovery: Selecting the basic linkage properties. In Egidio Astesiano, Jan A. Bergstra, Dennis Smith, and Steve Woods, editors, *Science of Computer Programming, special issue on program comprehension*, 40(2–3):213–234. Elsevier Science, July 2001.
- [Ardourel *et al.*, 2003] cité page 82
Gilles Ardourel, Pierre Crescenzo et Philippe Lahire. LAMP : vers un LAngage de définition de Mécanismes de Protection. Jean-Pierre Briot et Jacques Malenfant, éditeurs, *actes du 9^e colloque Langages et Modèles à Objets*, pages 151–163. Hermès Science Publications, février 2003.
- [Arévalo et Mens, 2002] cité page 281
Gabriela Arévalo and Tom Mens. Analysing object-oriented application frameworks using concept analysis. In Jean-Michel Bruel and Zohra Bellahsene, editors, *proceedings of the 1st OOIS workshop on MANaging SPEcialization/Generalization Hierarchy*, pages 53–63. Springer-Verlag, September 2002.
- [Baniassad *et al.*, 2002] cité page 179
Elisa L. A. Baniassad, Gail Murphy, and Christa Schwanninger. Understanding design patterns with design rationale graphs. Technical report T2-2002-01, Department of Computer Science, University of British Columbia, January 2002.
- [Bansiya, 1998] cité page 14
Jagdish Bansiya. Automating design-pattern identification. *Dr. Dobb's Journal*, June 1998.
- [Barbier et Henderson-Sellers, 2001] cité page 55
Franck Barbier and Brian Henderson-Sellers. The whole-part relationship in object modeling: A definition in cOIoR. In Martin Shepperd and Michael Dyer, editors, *journal of Information and Software Technology*, 43(1):19–39. Elsevier Science, January 2001.
- [Baumgartner *et al.*, 1996] cité page 19
Gerald Baumgartner, Konstantin Läufer, and Vincent F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical report CSD-TR-96-020, Department of Computer Science, University of Purdue, February 1996.
- [Bayardo Jr. et Miranker, 1996] cité page 161
Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In Dan Weld and Bill Clancey, editors, *proceedings of the 13th national conference on artificial intelligence*, pages 298–304. AAAI Press / The MIT Press, August 1996.
- [Beck, 1999] cité page 15
Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition, October 1999. ISBN: 0-201-61641-6.

- [Bellay et Gall, 1997] cité page 32
Berndt Bellay and Harald Gall. A comparison of four reverse engineering tools. In Ira Baxter and Alex Quilici, editors, *proceedings of the 4th Working Conference on Reverse Engineering*, pages 2–11. IEEE Computer Society Press, October 1997.
- [Berard, 1990] cité page 19
Edward V. Berard. Object-oriented programming languages. Technical report, The Object Agency, L.L.C., January 1990.
- [Bergenti et Poggi, 2000] cité page 61
Federico Bergenti and Agostino Poggi. IDEA: A design assistant based on automatic design pattern detection. In Dan Cooke and Joseph Urban, editors, *proceedings of the 12th international conference on Software Engineering and Knowledge Engineering*, pages 336–343. Springer-Verlag, July 2000.
- [Bézivin et Ploquin, 2001] cité page 280
Jean Bézivin and Nicolas Ploquin. Tooling the MDA framework: A new software maintenance and evolution scheme proposal. In Richard Wiener, editor, *Journal of Object-Oriented Programming*, 14(12). SIGS Publications, December 2001.
- [Bicarregui *et al.*, 1997] cité page 54
Juan C. Bicarregui, Kevin C. Lano, and Tom S. E. Maibum. Objects, associations and subsystems: A hierarchical approach to encapsulation. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of 11th European Conference on Object-Oriented Programming*, pages 324–343. Springer-Verlag, June 1997.
- [Biggerstaff *et al.*, 1993] cité page 32
Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. The concept assignment problem in program understanding. In Victor R. Basili, Richard A. DeMillo, and Takuya Katayama, editors, *proceedings of the 15th International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press / ACM Press, May 1993.
- [Biggerstaff *et al.*, 1994] cité page 32
Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. In James D. Palmer and N. Ann Fields, editors, *Communications of the ACM*, 37(5):72–82, May 1994.
- [Blake et Cook, 1987] cité pages 56, 95
Edwin Blake and Steve Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *proceedings of the 1st European Conference on Object-Oriented Programming*, pages 41–50. Springer-Verlag, June 1987.
- [Bonnano *et al.*, 1996] cité page 92
Nathalie Bonnano, Youssef Lahlou et Noureddine Mouaddib. Une approche dynamique pour l'identification de liens inter-objets. Mokrane Bouzeghoub et Arnold Rochfeld, éditeurs, *Ingénierie des systèmes d'information*, 4(4) : 439–461. Hermès Science Publications, Septembre 1996.

- [Booch *et al.*, 1999] cité page 128
Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1st edition, October 1999. ISBN: 0-201-57168-4.
- [Booch, 1993] cité page 15
Grady Booch. *Object-Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 2nd edition, September 1993. ISBN: 0-8053-5340-2.
- [Borne et Revault, 1999] cité page 180
Isabelle Borne et Nicolas Revault. Comparaison d'outils de mise en oeuvre de design patterns. Dominique Rieu et Jean-Pierre Giraudin, éditeurs, *L'objet, numéro thématique "patrons orientés objet"*, 5(2) : 243–266. Hermès Science Publications, juillet 1999.
- [Bosch, 1998] cité pages 16, 179
Jan Bosch. Design patterns as language constructs. In Richard Wiener, editor, *Journal of Object-Oriented Programming*, 11(2):18–32. SIGS Publications, February 1998.
- [Bratko et Muggleton, 1995] cité page 61
Ivan Bratko and Stephen Muggleton. Applications of inductive logic programming. In Toshinori Munakata, editor, *Communications of the ACM*, 38(11):65–70. ACM Press, November 1995.
- [Breu *et al.*, 1997] cité page 54
Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of the 11th European Conference for Object-Oriented Programming*, pages 344–366. Springer-Verlag, June 1997.
- [Brichau, 2000] cité page 61
Johan Brichau. Declarative meta programming for a language extensibility mechanism. Technical report Vub-Prog-TR-00-09, Programming Technology Lab, Vrije Universiteit Brussel, March 2000.
- [Brown *et al.*, 1998] cité page 279
William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1st edition, March 1998. ISBN: 0-471-19713-0.
- [Brown, 1996] cité pages 23, 69
Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical report TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign, July 1996.
- [Bruel *et al.*, 2001] cité page 56
Jean-Michel Bruel, Brian Henderson-Sellers, Franck Barbier, Annig Le Parc, and Robert B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In Shushma Patel, Yingxu Wang, and Ronald H. Johnston, editors, *proceedings of the 7th international conference on Object-Oriented Information Systems*, pages 5–14. Springer-Verlag, August 2001.

- [Budinsky *et al.*, 1996] cité pages 20, 23
Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. In Gene F. Hoffnagle, editor, *IBM Systems Journal*, 35(2):151–171. IBM Publications Center, February 1996.
- [Buschmann *et al.*, 1996] cité page 13
Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1st edition, August 1996. ISBN: 0-47-195869-7.
- [Caseau et Laburthe, 1996] cité pages 76, 234
Yves Caseau and François Laburthe. Claire: Combining objects and rules for problem solving. In Yike Guo, Jose Meseguer, Tetsuo Ida, and Joxan Jaffar, editors, *proceedings of the JICSLP workshop on Multi-Paradigm Logic Programming*, pages 105–114. Technischen Universität Berlin, September 1996. Technical report 96-28.
- [Chambers *et al.*, 2000] cité page 16
Craig Chambers, Bill Harrison, and John Vlissides. A debate on language and tool support for design patterns. In Tom Reps, editor, *proceeding of the 27th conference on Principles of Programming Languages*, pages 277–289. ACM Press, January 2000.
- [Chambers, 1992] cité page 76
Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *proceedings of the 6th European Conference for Object-Oriented Programming*, pages 33–56. Springer-Verlag, July 1992.
- [Chaumon *et al.*, 2000] cité page 58
M. Ajmal Chaumon, Hind Kabaili, Rudolf K. Keller, François Lustman, and Guy Saint-Denis. Design properties and object-oriented software changeability. In Jürgen Ebert and Chris Verhoef, editors, *proceedings of the 4th Conference on Software Maintenance and Reengineering*, pages 45–54. IEEE Computer Society Press, February 2000.
- [Chiba, 1998]
Shigeru Chiba. Javassist – A reflection-based programming wizard for JAVA. In Jean-Charles Fabre and Shigeru Chiba, editors, *proceedings of the OOPSLA workshop on Reflective Programming in C++ and JAVA*. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4.
- [Chirico, 2002]
Ugo Chirico. JIProlog, April 2002.
- [Ciupke, 1999] cité page 61
Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In Donald Firesmith, editor, *proceeding of 30th conference on Technology of Object-Oriented Languages and Systems*, pages 18–32. IEEE Computer Society Press, August 1999.
- [Civello, 1993] cité pages 54, 92, 100, 105
Franco Civello. Roles for composite objects in object-oriented analysis and design. In Andreas Paepcke, editor, *proceedings of the 8th conference on Object-Oriented Program-*

- ming, Systems, Languages, and Applications*, pages 376–393. ACM Press, September 1993.
- [Clifton *et al.*, 2000] cité page 76
Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In Doug Lea, editor, *proceedings of the 15th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145. ACM Press, October 2000.
- [Coplien, 1991] cité page 13
James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1st edition, August 1991. ISBN: 0-201-54855-0.
- [Coplien, 1998] cité page 13
James O. Coplien. Software design patterns: Common questions and answers. In Linda Rising, editor, *The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320. Cambridge University Press, January 1998.
- [Deimel et Naveda, 1990]
Lionel E. Deimel and J. Fernando Naveda. Reading computer programs: Instructor’s guide and exercises. Technical report CMU/SEI-90-EM-3, Software Engineering Institute, Carnegie Mellon University, August 1990.
- [Demeyer *et al.*, 1999a] cité page 80
Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why FAMIX and not UML? Technical report, Software Composition Group, University of Bern, 1999.
- [Demeyer *et al.*, 1999b] cité pages 15, 128
Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. In Bernhard Rumpe, editor, *proceedings of the 2nd UML conference*, pages 630–644. Springer-Verlag, October 1999.
- [Ducasse *et al.*, 1995] cité pages 56, 148
Stéphane Ducasse, Mireille Blay-Fornarino, and Anne-Marie Pinna-Dery. A reflective model for first class dependencies. In Frank Manola, editor, *proceedings of 10th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–280. ACM Press, October 1995.
- [Ducasse, 1997a] cité page 56
Stéphane Ducasse. *Intégration Réflexive de Dépendances Dans un Modèle À Classes*. Thèse de doctorat, université de Nice à Sophia Antipolis, janvier 1997.
- [Ducasse, 1997b] cité page 16
Stéphane Ducasse. Réification de schémas de conception : une expérience. Roland Ducournau, éditeur, *actes du 3^e colloque Langages et Modèles à Objets*, pages 95–100. Hermès Science Publications, octobre 1997.
- [Ducassé, 1999a] cité pages 79, 126, 216
Mireille Ducassé. Coca: A debugger for C based on fine grained control flow and data events. In David Garlan and Jeff Kramer, editors, *proceedings of the 21st International Conference on Software Engineering*, pages 504–513. ACM Press, May 1999.

- [Ducassé, 1999b] cité pages 79, 126, 216
Mireille Ducassé. OPIUM: An extendable trace analyser for Prolog. In Annalisa Bossi and Yves Deville, editors, *Journal of Logic Programming, special issue on synthesis, transformation and analysis of logic programs*, 39(1–3):177–223. Elsevier Science, April 1999.
- [Eden *et al.*, 1997] cité page 180
Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Precise specification and automatic application of design patterns. In Michael Lowry and Yves Ledru, editors, *proceedings of the 12th conference on Automated Software Engineering*, pages 143–152. IEEE Computer Society Press, November 1997.
- [Eden *et al.*, 1998] cité page 151
Amnon H. Eden, Yoram Hirshfeld, and Amiram Yehudai. LePUS – A declarative pattern specification language. Technical report 326/98, Department of Computer Science, University of Tel Aviv, June 1998.
- [Eden, 2000] cité pages 80, 148, 151, 180
Amnon H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. Ph.D. thesis, Department of Computer Science, University of Tel Aviv, 2000.
- [Eichelberger et von Gudenberg, 2002] cité pages 59, 282
Holger Eichelberger and Jürgen Wolff von Gudenberg. On the visualization of Java programs. In Stephan Diehl, editor, *proceedings of the 1st international seminar on Software Visualization*, pages 295–306. Springer-Verlag, May 2002.
- [Eppstein, 1995] cité page 71
David Eppstein. Subgraph isomorphism in planar graphs and related problems. In Kenneth Clarkson, editor, *proceedings of the 6th annual Symposium On Discrete Algorithms*, pages 632–640. ACM Press, January 1995.
- [Fariás *et al.*, 2002] cité pages 281, 287
Andrés Fariás, Yann-Gaël Guéhéneuc, and Mario Südholt. Integrating behavioral protocols in Enterprise Java Beans. In Kenneth Baclawski and Haim Kilov, editors, *proceedings of the 11th OOPSLA workshop on Behavioral Semantics: Serving the Customer*, pages 80–89. College of Computer Science, Northeastern University, October 2002.
- [Fariás et Guéhéneuc, 2003] cité pages 281, 287
Andrés Fariás and Yann-Gaël Guéhéneuc. On the coherence of component protocols. In Uwe Assmann, Elke Pulvermueller, Isabelle Borne, Noury Bouraqadi, and Pierre Cointe, editors, *Electronic Notes in Theoretical Computer Science*, 82(5). Elsevier Science, April 2003.
- [Fischer *et al.*, 1992] cité page 13
Gerhard Fischer, Andreas Girgensohn, Kumiyo Nakakoji, and David Redmiles. Supporting software designers with integrated domain-oriented design environments. In Alex Borgida and Matthias Jarke, editors, *Transactions on Software Engineering*, 18(6):511–522. IEEE Computer Society Press, June 1992.

- [Florijn *et al.*, 1997] cité pages 17, 20, 63, 80, 148
Gert Florijn, Marco Meijers, and Pieter Van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of 11th European Conference for Object-Oriented Programming*, pages 472–496. Springer-Verlag, June 1997.
- [Fowler, 1996] cité page 13
Martin Fowler. *Analysis Patterns : Reusable Object Models*. Addison-Wesley – Object Technology Series, 1st edition, October 1996. ISBN: 0-201-89542-0.
- [France, 1999] cité page 54
Robert B. France. A problem-oriented analysis of basic UML static requirements modeling concepts. In Linda Northrop, editor, *proceedings of the 14th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–69. ACM Press, November 1999.
- [Freuder, 1978] cité page 158
Eugene Freuder. Synthesizing constraint expressions. In Robert L. Ashenhurst, editor, *Communications of the ACM*, 21(11):958–966. ACM Press, November 1978.
- [Gabriel, 1996] cité page 15
Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1st edition, April 1996. ISBN: 0-19-5100269-X.
- [Gall *et al.*, 1996] cité pages 14, 20, 182
Harald C. Gall, René R. Klösch, and Roland T. Mittermeir. Application patterns in re-engineering: Identifying and using reusable concepts. In Bernardette Bouchon-Meunier, Miguel Delgado, Jose Luis Verdegay, Maria Amparo Vila, and Ronald R. Yager, editors, *proceedings of the 6th international conference on Information Processing and Management of Uncertainty in knowledge-based systems*, pages 1099–1106. Springer-Verlag, July 1996.
- [Gamma *et al.*, 1994] cité pages 7, 13, 16, 17, 18, 20, 23, 128, 143, 148, 151, 166, 181, 184, 198, 204
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994. ISBN: 0-201-63361-2.
- [Gamma et Beck, 1998] cité pages 27, 129
Erich Gamma and Kent Beck. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50. SIGS Publications, July 1998.
- [Gamma et Beck, 2002] cité page 129
Erich Gamma and Kent Beck. JUnit. Web site, 2002.
- [Gamma et Eggenschwiler, 1998] cité pages 22, 27, 129
Erich Gamma and Thomas Eggenschwiler. JHotDraw. Web site, 1998. members.pingnet.ch/gamma/JHD-5.1.zip.
- [Gannod et Cheng, 1999] cité pages 32, 33, 36, 73
Gerald C. Gannod and Betty H. C. Cheng. A framework for classifying and comparing

- software reverse engineering and design recovery techniques. In Françoise Balmas, Michael Blaha, and Spencer Rugaber, editors, *proceedings of the 6th Working Conference on Reverse Engineering*, pages 77–88. IEEE Computer Society Press, October 1999.
- [Ghoniem et Fekete, 2002] cité page 282
 Mohammad Ghoniem et Jean-Daniel Fekete. Visualisation de graphes de co-activité par matrices d'adjacence. Éric Lecolinet et Dominique L. Scapin, éditeurs, *actes de la 14^e conférence sur l'Interaction Homme-Machine*, pages 279–282. ACM Press, octobre 2002.
- [Ghoniem et Fekete, 2003] cité page 282
 Mohammad Ghoniem et Jean-Daniel Fekete. Visualisation matricielle des graphes et manipulation directe de hiérarchies de clusters. Éric Brangier et Christophe Kolski, éditeurs, *actes de la 15^e conférence sur l'Interaction Homme-Machine*. ACM Press, novembre 2003. Soumis pour publication.
- [Golomb et Baumert, 1965] cité page 159
 Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. In Richard W. Hamming, editor, *Journal of the ACM*, 12(4):516–524. ACM Press, October 1965.
- [Greenwood, 2000] cité page 214
 Matt Greenwood. *CFParse Distribution*. IBM AlphaWorks, September 2000.
- [Guéhéneuc *et al.*, 2002a] cité pages 277, 287
 Yann-Gaël Guéhéneuc, Hervé Albin-Amiot, Rémi Douence, and Pierre Cointe. Bridging the gap between modeling and programming languages. Technical report 02/09/INFO, Computer Science Department, École des Mines de Nantes, July 2002.
- [Guéhéneuc *et al.*, 2002b] cité pages 217, 277, 287
 Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caféine – A tool for dynamic analysis of Java programs. Technical report 02/07/INFO, Computer Science Department, École des Mines de Nantes, May 2002.
- [Guéhéneuc *et al.*, 2002c] cité pages 277, 278, 287
 Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caféine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the 17th conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, September 2002.
- [Guéhéneuc et Albin-Amiot, 2001] cité pages 280, 287
 Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.
- [Guéhéneuc et Albin-Amiot, 2003] cité pages 277, 287
 Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. A pragmatic study of binary class relationships. In John Grundy and John Penix, editors, *proceedings of the 18th conference on Automated Software Engineering*, pages 277–280. IEEE Computer Society Press, September 2003. Short paper.

- [Guéhéneuc et Jussien, 2001a] cité pages 277, 278, 287
Yann-Gaël Guéhéneuc et Narendra Jussien. Quelques explications pour les patrons – Une application de la PPC avec explications pour l’identification de patrons de conception. Bertrand Neveu, éditeur, *actes des 7^e Journées Nationales sur la résolution de Problèmes NP-Complets*, pages 111–122. ONERA, juin 2001.
- [Guéhéneuc et Jussien, 2001b] cité pages 277, 278, 287
Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *proceedings of the 1st IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.
- [Guéhéneuc, 2002] cité pages 278, 287
Yann-Gaël Guéhéneuc. Three musketeers to the rescue – Meta-modelling, logic programming, and explanation-based constraint programming for pattern description and detection. In Kris De Volder, Kim Mens, Tom Mens, and Roel Wuyts, editors, *proceedings of the 1st ASE workshop on Declarative Meta-Programming*. Computer Science Department, University of British Columbia, September 2002.
- [Guéret *et al.*, 2000] cité page 155
Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: An application to open-shop problems. In Gunduz Ulusoy and Selçuk Karabati, editors, *European Journal of Operational Research*, 127(2):344–354. Elsevier Science, December 2000.
- [Hannemann et Kiczales, 2002] cité page 13
Jan Hannemann and Gregor Kiczales. Design pattern implementation in JAVA and AspectJ. In Satoshi Matsuoka, editor, *proceedings of the 17th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, November 2002.
- [Harrison *et al.*, 2000] cité page 58
William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML designs to JAVA. In Doug Lea, editor, *proceedings of the 15th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 178–188. ACM Press, October 2000.
- [Hartmann *et al.*, 1992] cité page 56
Thorsten Hartmann, Ralf Jungclaus, and Gunter Saake. Aggregation in a behavior oriented object model. In Ole Lehrmann Madsen, editor, *proceedings of 6th European Conference for Object-Oriented Programming*, pages 57–77. Springer-Verlag, June–July 1992.
- [Hedin, 1997] cité page 69
Görel Hedin. Language support for design patterns using attribute extension. In Jan Bosch and Stuart Mitchell, editors, *proceedings of the 1st ECOOP workshop on Language Support for Design Patterns and Frameworks*, pages 137–140. Springer-Verlag, June 1997.
- [Henderson-Sellers et Barbier, 1999] cité pages 19, 55, 85, 100, 105, 106
Brian Henderson-Sellers and Franck Barbier. A survey of the UML’s aggregation and

- composition relationships. In Jean-Claude Royer, editor, *L'objet : Logiciel, Base de données, Réseaux*, 5(3/4):339–366. Hermès Science Publications, December 1999.
- [Henderson-Sellers, 2001] cité page 55
 Brian Henderson-Sellers. Some problems with the UML v1.3 metamodel. In Ralph H. Sprague Jr., editor, *proceedings of the 34th annual Hawaii International Conference on System Sciences*, pages 3052–3064. IEEE Computer Society Press, January 2001.
- [Heuzeroth *et al.*, 2002] cité pages 122, 178
 Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In Hartmut Ehrig, Bernd J. Krämer, and Atila Ertas, editors, *proceedings the 6th world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.
- [Ho *et al.*, 2002] cité page 16
 Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau. A toolkit for weaving aspect oriented designs. In Gregor Kiczales, editor, *proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 99–105. ACM Press, April 2002.
- [Isazadeh *et al.*, 1995] cité page 13
 Ayaz Isazadeh, Glenn H. MacEwen, and Andrew J. Malton. Behavioral patterns for software requirement engineering. In Susan Puglia and Morven Gentleman, editors, *CD-Rom of the 5th annual IBM Centers for Advanced Studies CONference*. Centre for Advanced Studies of IBM Toronto Laboratory and the Institute for Information Technology of the National Research Council of Canada, November 1995.
- [Jackson et Rinard, 2000] cité pages 17, 80
 Daniel Jackson and Martin C. Rinard. Software analysis: A roadmap. In Mehdi Jazayeri and Alexander Wolf, editors, *proceedings of the 22nd International Conference on Software Engineering, future of software engineering track*, pages 133–145. ACM Press, June 2000.
- [Jackson et Waingold, 1999] cité pages 43, 57, 93, 117, 118
 Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In David Garlan and Jeff Kramer, editors, *proceedings of the 21st International Conference on Software Engineering*, pages 194–202. ACM Press, May 1999.
- [Jahnke *et al.*, 1997] cité pages 38, 126
 Jens H. Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In Mehdi Jazayeri, editor, *proceedings of the 6th European Software Engineering Conference*, pages 193–210. ACM Press, September 1997.
- [Jahnke et Zündorf, 1997] cité page 66
 Jens H. Jahnke and Albert Zündorf. Rewriting poor design patterns by good design patterns. In Serge Demeyer and Harald C. Gall, editors, *proceedings the 1st ESEC/FSE workshop on Object-Oriented Reengineering*. Distributed Systems Group, Technical University of Vienna, September 1997. TUV-1841-97-10.

- [Johnson et Erdem, 1995] cité page 13
W. Lewis Johnson and Ali Erdem. Interactive explanation of software systems. In Dorothy Setliff, editor, *proceedings of the 10th Knowledge-Based Software Engineering conference*, pages 155–164. IEEE Computer Society Press, November 1995.
- [Johnson et Opdyke, 1993] cité page 280
Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In Shojiro Nishio and Akinori Yonezawa, editors, *proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, pages 264–278. Springer-Verlag, November 1993.
- [Jussien *et al.*, 2000] cité page 160
Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In Rina Dechter, editor, *proceedings of the 6th conference on principles and practice of Constraint Programming*, pages 249–261. Springer-Verlag, September 2000.
- [Jussien et Barichard, 2000] cité page 235
Narendra Jussien and Vincent Barichard. The PaLM system: Explanation-based constraint programming. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques for Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.
- [Jussien, 1997] cité pages 160, 161
Narendra Jussien. *Relaxation de Contraintes Pour Les Problèmes Dynamiques*. Thèse de doctorat, université de Rennes I, octobre 1997.
- [Jussien, 2001a] cité page 160
Narendra Jussien. e-Constraints: Explanation-based constraint programming. In Barry O’Sullivan and Eugene Freuder, editors, *1st CP workshop on User-Interaction in Constraint Satisfaction*, December 2001.
- [Jussien, 2001b] cité pages 155, 160
Narendra Jussien. Programmation par contraintes avec explications. Bertrand Neveu, éditeur, *actes des 7^e Journées Nationales sur la résolution de Problèmes NP-Complets*, pages 147–158. ONERA, juin 2001.
- [Jussien, 2003] cité page 160
Narendra Jussien. Programmation par contraintes pour les technologies logicielles. Gilles Muller, éditeur, *actes du colloque GEMSTIC*. Groupe des Ecoles des Mines, avril 2003.
- [Kaiser, 2001] cité pages 129, 289
Wolfram Kaiser. Become a programming picasso with JHotDraw – Use the highly customizable GUI framework to simplify draw application development. In Carolyn Wong, editor, *JavaWorld*. IDG Publications, February 2001.

- [Keene, 1989] cité page 76
Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison Wesley, 1st edition, December 1989. ISBN: 0-201-17589-4.
- [Keller *et al.*, 1999] cité page 65
Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *proceedings of the 21st International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.
- [Keller et Schauer, 2000] cité page 65
Rudolf K. Keller and Reinhard Schauer. Towards a quantitative assessment of method replacement. In Jürgen Ebert and Chris Verhoef, editors, *proceedings of the 4th Conference on Software Maintenance and Reengineering*, pages 141–150. IEEE Computer Society Press, February 2000.
- [Kent *et al.*, 1999] cité page 51
Stuart Kent, Andy Evans, and Bernhard Rumpe. UML semantics FAQ. In Ana M. D. Moreira and Serge Demeyer, editors, *ECOOOP workshop reader*, pages 33–56. Springer-Verlag, June 1999.
- [Kiczales *et al.*, 1997] cité page 13
Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *proceedings of 11th European Conference for Object-Oriented Programming*, pages 220–242. Springer-Verlag, June 1997.
- [Korn *et al.*, 1999] cité page 42
Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of JAVA applets. In Kostas Kontogiannis and Françoise Balmas, editors, *proceedings of the 6th Working Conference on Reverse Engineering*, pages 314–325. IEEE Computer Society Press, November 1999.
- [Krämer et Prechelt, 1996] cité page 61
Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *proceedings of the 3rd Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.
- [Kristensen, 1994] cité page 57
Bent Bruun Kristensen. Complex associations: Abstractions in object-oriented modeling. In J. Eliot B. Moss, editor, *proceedings of the 9th conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 272–283. ACM Press, October 1994.
- [Kullbach et Winter, 1999] cité page 65
Bernt Kullbach and Andreas Winter. Querying as an enabling technology in software reengineering. In Paolo Nesi and Chris Verhoef, editors, *proceedings of the 3rd Conference on Software Maintenance and Reengineering*, pages 42–50. IEEE Computer Society Press, March 1999.

- [Kunz et Black, 1995] cité page 67
Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging. In John Knight, editor, *Transactions on Software Engineering*, 21(6):515–527. IEEE Computer Society Press, June 1995.
- [Laburthe et Le Projet OCRE, 2000] cité page 234
François Laburthe et Le Projet OCRE. Choco : implémentation du noyau d’un système de contraintes. Christian Bessière, éditeur, *actes des 6^e Journées Nationales sur la résolution de Problèmes NP-Complets*, pages 151–165. ONERA, juin 2000.
- [Laburthe, 2000] cité page 234
François Laburthe. *Choco’s API*. OCRE commitee, October 2000.
- [Larousse, 1995]
Larousse. *Le Petit Larousse Illustré*. Larousse, 90^e édition, 1995. ISBN : 2-03-301195-X.
- [Lauder et Kent, 1998] cité pages 19, 179
Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Serge Demeyer and Jan Bosch, editors, *proceedings of 12th European Conference for Object-Oriented Programming*, pages 114–134. Springer-Verlag, July 1998.
- [Lemesle, 2000] cité pages 11, 127
Richard Lemesle. *Techniques de Modélisation et de Métamodélisation*. Thèse de doctorat, université de Nantes, octobre 2000.
- [Lévy et Losavio, 1998] cité page 179
Nicole Lévy and Francis Losavio. Analyzing and comparing architectural styles. In Raul Monge and Marcello Visconti, editors, *proceedings of the 19th international Conference of the Chilean Computer Science Society*. IEEE Computer Society Press, November 1998.
- [Lieberman et Fry, 1995] cité page 79
Henry Lieberman and Christopher Fry. Bridging the gulf between code and behavior in programming. In Irvin R. Katz, Robert L. Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen, editors, *proceedings of the 13th Conference on Human Factors and Computing Systems*, pages 480–486. ACM Press, May 1995.
- [Lieberman, 1987] cité pages 13, 16, 79
Henry Lieberman. Reversible object-oriented interpreters. In Jean Bézivin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, *proceedings of 1st European Conference for Object-Oriented Programming*, pages 11–19. Springer-Verlag, June 1987.
- [Lindholm et Yellin, 1999] cité page 42
Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999. ISBN: 0-201-43294-3.
- [Loudni, 2002] cité pages 155, 158
Samir Loudni. *Conception et Mise En Œuvre d’Algorithmes Anytime : Une Approche à Base de Contraintes*. Thèse de doctorat, École des Mines de Nantes, décembre 2002.
- [Mancoridis et al., 1998] cité pages 14, 16
Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner. Bunch:

- A clustering tool for the recovery and maintenance of software system structures. In Taghi M. Khoshgoftaar and Keith Bennett, editors, *proceedings of the 6th International Conference on Software Maintenance*, pages 50–59. IEEE Computer Society Press, August 1998.
- [Marcos *et al.*, 2001] cité page 58
Esperanza Marcos, Belen Vela, José M. Caverio, and Paloma Cáceres. Aggregation and composition in object-Relational database design. In Albertas Caplinskas and Johann Eder, editors, *proceedings of the 5th east-european conference on Advances in Databases and Information Systems*, pages 195–209. Springer-Verlag, September 2001.
- [Martin, 1998] cité page 53
Robert C. Martin. Association, aggregation, and composition relationships, 1998.
- [Mens *et al.*, 2002a] cité page 61
Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In Filomena Ferrucci and Giuliana Vitiello, editors, *proceedings of the 14th international conference on Software Engineering and Knowledge Engineering*, pages 289–296. ACM Press, July 2002.
- [Mens *et al.*, 2002b] cité page 61
Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. In Jay Liebowitz, editor, *journal on Expert Systems with Applications*, 23(4):405–413. Elsevier Science, November 2002.
- [Merriam-Webster, 2003]
Merriam-Webster. Merriam-webster online dictionnary, March 2003.
- [Mikkonen, 1998] cité page 180
Tommi Mikkonen. Formalizing design patterns. In Takuya Katayama and David Notkin, editors, *proceedings of the 20th International Conference on Software Engineering*, pages 115–124. IEEE Computer Society Press, April 1998.
- [Miller et Mukerji, 2001] cité page 280
Joaquin Miller and Jishnu Mukerji. *Model Driven Architecture – A Technical Perspective*. Object Management Group, Inc. Architecture Board MDA Drafting Team, July 2001.
- [Montanari, 1974] cité pages 158, 160
Ugo Montanari. Networks of constraints: Fundamental properties and application to picture processing. In Paul P. Wang, editor, *Information Sciences*, 7(2):95–132. Elsevier Science, April 1974.
- [Niere *et al.*, 2001] cité pages 15, 38
Jörg Niere, Jörg P. Wadsack, and Albert Zündorf. Recovering UML diagrams from JAVA code using patterns. In Jens H. Jahnke and Conor Ryan, editors, *proceedings of the 2nd workshop on Soft Computing Applied to Software Engineering*, pages 89–97. Springer-Verlag, February 2001.
- [Niere, 2002] cité page 38
Jörg Niere. Fuzzy logic based interactive recovery of software design. Presented at the ICSE Doctoral Symposium, May 2002.

- [Noble et Grundy, 1995] cité page 53
James Noble and John Grundy. Explicit relationships in object-oriented development. In Bertrand Meyer, editor, *proceedings of the 18th conference on the Technology of Object-Oriented Languages and Systems*, pages 211–226. Prentice-Hall, November 1995.
- [Object Management Group, Inc., 1997] cité page 69
Object Management Group, Inc. *Object Constraint Language Specification*, September 1997.
- [Object Management Group, Inc., 1999] cité page 51
Object Management Group, Inc. *UML v1.3 Specification*, June 1999.
- [Object Management Group, Inc., 2001] cité pages 11, 51
Object Management Group, Inc. *UML v1.4 Specification*, September 2001.
- [Object Management Group, Inc., 2003] cité pages 19, 46, 47, 51, 52
Object Management Group, Inc. *UML v1.5 Specification*, March 2003.
- [Object Technology International, Inc. / IBM, 2001] cité page 28
Object Technology International, Inc. / IBM. Eclipse platform – A universal tool platform, July 2001.
- [Office québécois de la langue française, 2003] cité pages 13, 26, 76, 130, 155, 159, 179, 181, 216
Office québécois de la langue française. Grand dictionnaire terminologique en ligne, février 2003.
- [Opdyke, 1997] cité page 280
William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.
- [Pagel et Winter, 1996] cité pages 80, 148
Bernd-Uwe Pagel and Mario Winter. Towards pattern-based tools. In Frank Buschmann, editor, *proceedings of 1st european conference on Pattern Languages of Programs*. Preliminary conference proceedings, July 1996.
- [Pal et Minsky, 1996] cité page 279
Partha Pratim Pal and Naftaly H. Minsky. Imposing the Law of Demeter and its variations. In Raimund Ege, editor, *proceedings of the 20th conference on the Technology of Object-Oriented Languages and Systems*. Prentice-Hall, August 1996.
- [Perrochon et Mann, 1999] cité page 11
Louis Perrochon and Walter Mann. Inferred designs. In James O. Coplien, editor, *Software*, 16(5):46–51. IEEE Computer Society Press, September–October 1999.
- [Petit, 2002] cité page 159
Thierry Petit. *Modélisation et Algorithmes de Résolution de Problèmes Sur-Contraints*. Thèse de doctorat, université du Languedoc, novembre 2002.
- [Piechowiak et Rodriguez, 2000] cité page 161
Sylvain Piechowiak and Jeffrey J. Rodriguez. Constraint compiling into rules formalism for dynamic CSPs computing. In Thom Frühwirth, editor, *proceedings of the 1st workshop on Rule-Based Constraint Reasoning and Programming*. Computing Research Repository, July 2000.

- [Prechelt et Krämer, 1998] cité page 61
Lutz Prechelt and Christian Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. In Maurer Hermann, editor, *Journal of Universal Computer Science*, 4(12):866–883. Know-Center, December 1998.
- [Pressman, 2001] cité page 11
Roger S. Pressman. *Software Engineering – A Practitioner’s Approach*. McGraw-Hill Higher Education, 5th edition, November 2001. ISBN: 0-07-249668-1.
- [Quilici *et al.*, 1997] cité page 64
Alex Quilici, Quing Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. In Bashar Nuseibeh, editor, *Journal of Automated Software Engineering*, 5(3):347–372. Kluwer Academic Publishers, July 1997.
- [Rapicault et Fornarino, 2000] cité pages 69, 80, 180
Pascal Rapicault et Mireille Fornarino. Instanciation et vérification de patterns de conception : un méta-protocole. Christophe Dony et Houari Sahraoui, éditeurs, *actes du 6^e colloque Langages et Modèles à Objets*, pages 43–58. Hermès Science Publications, janvier 2000.
- [Rational Software Technical Support, 2000] cité pages 46, 52
Rational Software Technical Support. What is the difference between aggregation and composition and how are they represented in rose?, March 2000.
- [ReiBing, 2001] cité page 280
Ralf ReiBing. Assessing the quality of object-oriented designs. In Doug Lea, editor, *proceedings of the 16th OOPSLA Doctoral Symposium*. ACM Press, October 2001.
- [Rich et Waters, 1990] cité pages 13, 15, 16, 64
Charles Rich and Richard C. Waters. *The Programmer’s Apprentice*. ACM Press Frontier Series and Addison-Wesley, 1st edition, January 1990. ISBN: 0-201-52425-2.
- [Richner et Ducasse, 1999] cité pages 14, 61, 80
Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *proceedings of 7th International Conference on Software Maintenance*, pages 13–22. IEEE Computer Society Press, August 1999.
- [Rousseau *et al.*, 1995] cité page 58
Bertrand Rousseau, Alberto Aimar, Arash Khodabandeh, and Paolo Palazzi. Filling the gap between OO methodologies and programming languages. Programming Techniques Group, CERN ECP Division, 1211 Geneva 23, Switzerland., March 1995.
- [Rumbaugh *et al.*, 1991] cité page 51
James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Inc., 1st edition, October 1991. ISBN: 0-13-629841-9.
- [Rumbaugh *et al.*, 1999] cité page 58
James Rumbaugh, Robert Jacobson, and Grady Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1st edition, January 1999. ISBN: 0-201-30998-X.

- [Rumbaugh, 1987] cité page 57
James Rumbaugh. Relations as semantic constructs in an object-oriented language. In Norman K. Meyrowitz, editor, *proceedings of the 2nd conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 466–481. ACM Press, December 1987.
- [Saksena *et al.*, 1998] cité page 55
Monika Saksena, Robert B. France, and Maria M. Larrondo-Petrie. A characterization of aggregation. In Colette Rolland, editor, *proceedings of the 5th international conference on Object-Oriented Information Systems*, pages 363–372. Springer-Verlag, September 1998.
- [Samuelson, 1990] cité pages 11, 214
Pamela Samuelson. Reverse-engineering someone else’s software: Is it legal? In Wilma M. Osborne and Elliot J. Chikofsky, editors, *Software*, 7(1):90–96. IEEE Computer Society Press, January/February 1990.
- [Samuelson, 2002] cité pages 11, 214
Pamela Samuelson. Reverse engineering under siege. In Diane Crawford, editor, *Communications of the ACM*, 45(10):15–20. ACM Press, October 2002.
- [Sartipi *et al.*, 2000] cité pages 12, 178
Kamran Sartipi, Kostas Kontogiannis, and Farhad Mavaddat. Architectural design recovery using data mining techniques. In Jürgen Ebert and Chris Verhoef, editors, *proceedings of the 4th Conference on Software Maintenance and Reengineering*, pages 129–140. IEEE Computer Society Press, March 2000.
- [Schauer et Keller, 1999] cité pages 65, 282
Reinhard Schauer and Rudolf K. Keller. Pattern visualization for software comprehension. In Scott Tilley and Giuseppe Visaggio, editors, *proceedings of the 6th International Workshop on Program Comprehension*, pages 4–12. IEEE Computer Society Press, June 1999.
- [Schiex *et al.*, 1995]
Thomas Schiex, Hélène Fargier, and Gérard Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In Chris S. Mellish, editor, *proceedings of 14th International Joint Conference on Artificial Intelligence*, pages 631–637. AAAI Press, August 1995.
- [Schiex et Verfaillie, 1994] cité page 160
Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. In Nikolaos G. Bourbakis, editor, *International Journal of Artificial Intelligence Tools*, 3(2):187–207. World Scientific, February 1994.
- [Seemann et von Gudenberg, 1998] cité pages 17, 70, 79, 127, 129, 178
Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of JAVA software. In Bill Scherlis, editor, *proceedings of 5th international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.
- [Seemann, 1997] cité page 282
Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams:

- Towards automatic layout of object-oriented software diagrams. In Giuseppe Di Battista, editor, *proceedings of the 5th international symposium on Graph Drawing*, pages 415–424. Springer-Verlag, September 1997.
- [Sefika *et al.*, 1996] cité pages 61, 80
Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In Tom Maibaum and Marvin V. Zelkowitz, editors, *proceedings of the 18th International Conference on Software Engineering*, pages 387–397. ACM Press, March 1996.
- [Sharon, 1996] cité page 11
David Sharon. Meeting the challenge of software maintenance. In Ted J. Biggerstaff, editor, *Software*, 13(1):122–126. IEEE Computer Society Press, January 1996.
- [Shull *et al.*, 1996] cité pages 17, 19
Forrest Shull, Walcélio Melo, and Victor R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical report CS-TR-3597, Computer Science Department, University of Maryland, January 1996.
- [Simons et Graham, 1999] cité pages 51, 282
Anthony J. H. Simons and Ian Graham. 30 things that go wrong in object modelling with UML v1.3. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 17, pages 237–257. Kluwer Academic Publishers, 1st edition, October 1999.
- [Skogan, 1999] cité page 52
David Skogan. Application schema specific data structure, May 1999.
- [Soloway, 1986] cité pages 20, 126, 216
Elliot Soloway. Learning to program = Learning to construct mechanisms and explanations. In Peter J. Denning, editor, *Communications of the ACM*, 29(9):850–858. ACM Press, September 1986.
- [Soukup, 1995] cité page 16
Jiri Soukup. Implementing patterns. In Jim O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 20, pages 395–412. Addison-Wesley, 1st edition, May 1995.
- [Sun Microsystems, Inc., 2002] cité pages 27, 129
Sun Microsystems, Inc. *Java Abstract Window Toolkit*, May 2002.
- [Sunyé *et al.*, 2000] cité pages 23, 80, 148, 180
Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In Elisa Bertino, editor, *proceedings of the 14th European Conference for Object-Oriented Programming*, pages 44–62. Springer-Verlag, June 2000.
- [Sunyé, 1999] cité pages 20, 80
Gerson Sunyé. *Mise en Oeuvre de Patterns de Conception : un Outil*. Thèse de doctorat, université Pierre et Marie Curie, juillet 1999.
- [Taibi et Ngo, 2003] cité page 17
Toufic Taibi and David Chek Ling Ngo. Formal specification of design pattern combina-

- tion using BPSL. In Martin Dyer, Martin Shepperd, and Claes Wohlin, editors, *journal of Information and Software Technology*, 45(3):157–170. Elsevier Science, March 2003.
- [Takang et Grubb, 1996] cité page 11
Armstrong A. Takang and Penny A. Grubb. *Software Maintenance: Concepts and Practice*. International Thomson Computer Press, 1st edition, January 1996. ISBN: 1-85032-192-2.
- [Tatsubori et Chiba, 1998] cité page 69
Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. In Jean-Charles Fabre and Shigeru Chiba, editors, *proceedings of the 1st OOPSLA workshop on Reflective Programming in C++ and JAVA*, pages 56–60. Center for Computational Physics, University of Tsukuba, October 1998. UTCCP Report 98-4.
- [Tatsubori, 1999] cité pages 17, 79
Michiaki Tatsubori. An extension mechanism for the JAVA language. Master’s thesis, Graduate School of Engineering, University of Tsukuba, 1999.
- [Thimbleby, 1999] cité pages 17, 79
Harold W. Thimbleby. A critique of JAVA. In Douglas Comer and Andy J. Wellings, editors, *Software – Practice and Experience*, 29(5):457–478. John Wiley & Sons, April 1999.
- [Thomas, 2001] cité page 51
David Thomas. UML – The universal modeling *and programming* language? *LogOn Expert’s Corner*. LogOn, September 2001.
- [Thomas, 2002] cité pages 7, 80
Dave Thomas. Reflective software engineering – From MOPS to AOSD. In Richard Wiener, editor, *Journal of Object Technology*, 1(4):17–26. ETH Zürich, September 2002.
- [Tikir et Hollingsworth, 2002] cité page 133
Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In Phyllis G. Frankl, editor, *proceedings of the 6th International Symposium on Software Testing and Analysis*, pages 86–96. ACM Press, July 2002.
- [Tonella et Antoniol, 1999] cité page 17
Paolo Tonella and Giulio Antoniol. Object oriented design pattern inference. In Hongji Yang and Lee White, editors, *proceedings of the 7th International Conference on Software Maintenance*, pages 230–240. IEEE Computer Society Press, August 1999.
- [Tonella et Potrich, 2001] cité pages 17, 118
Paolo Tonella and Alessandra Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In Gerardo Canfora and Anneliese Amschler Andrews-Von Maryhauser, editors, *proceedings of the 9st International Conference on Software Maintenance*, pages 376–385. IEEE Computer Society Press, November 2001.
- [Torán, 1996] cité page 71
Jacobo Torán. On the complexity of the graph isomorphism problem, April 1996. Tutorial given at the British Colloquium for Theoretical Computer Science.

- [Traverso et Mancoridis, 2002] cité page 16
Martin Traverso and Spiros Mancoridis. On the automatic recovery of style-specific structural dependencies in software systems. In Bashar Nuseibeh, editor, *journal of Automated Software Engineering*, 9(4):331–359. Kluwer Academic Publishers, July 2002.
- [Tsang, 1993] cité pages 158, 159, 160
Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1st edition, August 1993. ISBN: 0-12-701610-4.
- [Vauttier, 1999] cité page 55
Sylvain Vauttier. Une nouvelle approche de la spécification du comportement des objets composites en UML. Roger Rousseau, éditeur, *actes du 5^e colloque Langages et Modèles à Objets*, pages 277–292. Hermès Science Publications, janvier 1999.
- [Walenstein, 2002] cité pages 32, 279
Andrew Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. Ph.D. thesis, School of Computing Science, Simon Fraser University, May 2002.
- [Wendorff, 2001] cité page 280
Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of 5th Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.
- [Wien, 2000] cité page 53
Universität Wien. Aggregation and composite objects., July 2000.
- [Wolczko, 1992] cité page 19
Mario Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *Software Engineering Journal*, 7(2):95–101. IEEE Xplore, March 1992.
- [Wuyts *et al.*, 1999] cité pages 20, 61
Roel Wuyts, Kim Mens, and Theo D’Hondt. Explicit support for software development styles throughout the complete life cycle. Technical report Vub-Prog-TR-99-07, Programming Technology Lab, Vrije Universiteit Brussel, April 1999.
- [Wuyts, 1998] cité pages 20, 23, 61, 62, 126, 281
Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the 26th conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

Glossaire

NOUS définissons les mots dont le sens nous semble important. Nous avons utilisé le sens admis dans le langage courant [Larousse, 1995] ou en informatique [Office québécois de la langue française, 2003] pour la plupart des mots. Cependant, nous avons parfois restreint le sens d'un mot ou proposé une définition spécifique lorsque le sens admis portait à confusion ou était trop imprécis, par exemple les sens des mots *patron* et *motif*.

A

Aller-retour

De [Office québécois de la langue française, 2003], *voyage aller-retour* : tout voyage dont la destination finale est le point d'origine, et dont le parcours est le même dans les deux sens.

En anglais : *round-trip*.

Un aller-retour [Demeyer *et al.*, 1999b ; Demeyer *et al.*, 1999a ; Niere *et al.*, 2001 ; Booch, 1993, page 517] consiste pour un mainteneur à suivre les constituants ou les motifs depuis un niveau d'abstraction vers un autre niveau plus abstrait pour comprendre leurs raisons d'être et leurs interactions, puis à retourner au niveau d'abstraction originel pour effectuer les modifications requises.

Voir aussi section 1.2 page 14.

Architecture

De [Larousse, 1995, page 85], *architecture* : **2.** Litt. Structure, organisation.

En anglais : *architecture*.

L'architecture d'un programme à objets représente la structure, l'organisation et le comportement global du programme [Jackson et Rinard, 2000] quelque soit le modèle employé pour la représenter : code source, modèle UML ou instance d'un métamodèle spécifique.

Voir aussi *diagramme de classe**.

Voir aussi la méthode OMT, [Rumbaugh *et al.*, 1991, page 455], dans laquelle l'architecture d'un programme est la “*structure d'ensemble du système, comprenant sa partition en sous-système et leur allocation à des tâches et à des processeurs.*”

Artefact

De [Larousse, 1995, page 92], *artefact* : **Didact.** Phénomène d'origine accidentelle ou artificielle, rencontré au cours d'une observation ou d'une expérience.

De [Office québécois de la langue française, 2003], *artefact* : partie d'information utilisée ou produite lors du processus de développement d'un logiciel.

Ce mot est parfois utilisé dans la littérature pour décrire les *constituants** d'un méta-modèle, d'un *modèle** ou d'un *formalisme**. Nous pensons que ce mot ne convient pas pour décrire les constituants de modèles car sa définition connote un phénomène dynamique. Pour les phénomènes dynamiques (résultats d'analyses dynamiques, par exemple), nous préférons le mot *événement**.

Le mot anglais *artifact*[Merriam-Webster, 2003] se rapproche du mot français *constituant*. Il est défini comme : **1 a** : Something created by humans usually for a practical purpose; especially : an object remaining from a particular period <caves containing prehistoric artifacts> **b** : Something characteristic of or resulting from a human institution or activity <self-consciousness... turns out to be an artifact of our education system – Times Literary Supplement> **2** : A product of artificial character (as in a scientific test) due usually to extraneous (as human) agency.

Aussi, le mot anglais *artifact* est utilisé dans la méthode RATIONAL UNIFIED PROCESS avec le sens particulier de : “[...] *some document, report, or executable that is produced, manipulated, or consumed.*” [Booch *et al.*, 1999, page 454].

C

Code octal

En anglais : *byte-code*.

Le langage de programmation JAVA est compilé vers un langage intermédiaire, appelé code octal, dont les instructions sont codées sur un octet. Ce code octal est alors exécuté par la machine virtuelle JAVA.

Compréhension de programme

En anglais : *program understanding*.

La sous-phase de compréhension, lors de la maintenance, représente le moment où les mainteneurs étudient un programme pour le comprendre. Comprendre un programme consiste à comprendre ce que chaque instruction signifie, comment le flot de contrôle passe d'une instruction à une autre, quels algorithmes ont été utilisés, comment l'information est représentée et transformée en structures de données, quels sous-programmes invoquent quels autres sous-programmes et comment le programme interagit avec son environnement [Deimel et Naveda, 1990, page 9].

Voir aussi Ted J. Biggerstaff *et al.* [Biggerstaff *et al.*, 1994] pour lesquels “*a person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.*”

Constituant

De [Larousse, 1995, page 263], *constituant* : **1.** Qui entre dans la constitution, la composition de quelque chose.

En anglais : *constituent*.

Nous utilisons le mot *constituant* pour unifier la désignation de ce qui entre dans la définition d'un métamodèle, d'un *modèle** et d'un *formalisme**, parfois appelés méta-entité, *entité** et *artefact** dans la littérature, respectivement. Nous n'utilisons pas le mot *artefact* car sa définition ne correspond pas à ce que nous exprimons. Nous utilisons parfois le mot *entité*.

D**Développement**

De [Office québécois de la langue française, 2003], *développement* : création, réalisation et mise au point d'un programme, d'un logiciel, d'une application ou d'un système.

En anglais : *development, programming*.

Le sens du mot *développement* est similaire à celui du mot *programmation* [Office québécois de la langue française, 2003] : “*ensemble des activités techniques reliées à l'élaboration d'un programme informatique. La programmation comprend des activités de conception, d'écriture, de test et de maintenance de programmes pour ordinateurs.*”

Cependant, les sens des mots *développeur** et *programmeur** sont différents et nous préférons les mots *développement* et *développeur*.

Développeur

De [Office québécois de la langue française, 2003], *développeur* : personne responsable de la définition d'un problème qui sera résolu grâce à l'informatique, de la conception des procédures qui seront utilisées pour traiter ce problème et de l'écriture du programme nécessaire, qui peut également prendre en charge la maintenance et l'évolution du programme.

En anglais : *developer*.

Nous utilisons le mot *développeur* pour désigner indifféremment la personne qui définit l'architecture d'un programme, qui le conçoit, et qui l'implémente. Nous préférons le mot *développeur* au mot *programmeur** qui est plus spécifique.

Diagramme

De [Larousse, 1995, page 338], *diagramme* : représentation graphique ou schématique permettant de décrire l'évolution d'un phénomène, la corrélation de deux facteurs, la disposition des parties d'un ensemble.

De [Office québécois de la langue française, 2003], *diagramme* : représentation graphique d'une séquence d'opérations ou de la structure d'un système. Le but du diagramme est de montrer l'évolution d'un processus ou les relations qui existent entre plusieurs ensembles interdépendants.

En anglais : *diagram*.

Voir aussi la notation UML, [Rumbaugh *et al.*, 1999, page 260], qui définit un dia-

gramme comme “*A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).* UML supports class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, statechart diagram, activity diagram, component diagram, and deployment diagram.”

Diagramme de classes

Dans la notation UML, [Rumbaugh *et al.*, 1999, page 260], un diagramme de classes est défini comme “[...] *a graphic presentation of the static view that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships. A class diagram may show a view of a package and may contain symbols for nested packages. A class diagram contains certain reified behavioral elements, such as operations, but their dynamics are expressed in other diagrams, such as statechart diagrams and collaboration diagrams.*”

En anglais : *class diagram*.

La définition d’un diagramme de classe indique qu’il donne une vision statique d’un système mais nous pensons, comme Daniel Jackson et Martin Rinard, qu’un diagramme de classes décrit la structure *et* le comportement *global* du système : “*Object models are often no more than class diagrams, showing the classes, their fields, and the subclassing relationships between them. [...] The object model describes the global relationships amongst the elements of the system; its states are global configuration of objects, and its transition are the global changes that results as objects come and go and as relationships are altered.*” [Jackson et Rinard, 2000].

E

EDI

De [Office québécois de la langue française, 2003], *environnement de développement intégré* : dans un système de développement, ensemble d’outils intégrés qui sont utilisés pour le développement de logiciels et qui sont directement accessibles à partir de l’interface utilisateur. Un environnement de développement intégré comprend habituellement un éditeur de texte, un compilateur et un débogueur.

En anglais : *IDE* pour *Integrated Development Environment*.

Entité

De [Larousse, 1995, page 393], *entité* : **1.** Réalité abstraite qui n’est conçue que par l’esprit.

De [Office québécois de la langue française, 2003], *entité* : élément représentant un phénomène (personne, concept, événement) et qui peut être traité comme une unité indépendante ou un membre d’une catégorie particulière, et à propos duquel des données peuvent être stockées.

En anglais : *entity*.

Nous utilisons le mot entité quand le seul mot *constituant** pourrait être ambiguë.

Événement

De [Larousse, 1995, page 415], *événement* : **1.** Ce qui se produit, arrive ou apparaît ;

fait, circonstance. **3. STAT.** Éventualité qui se réalise dans un univers donné.

De [Office québécois de la langue française, 2003], *événement* : signal qui permet, par ses différents états, d'indiquer la situation ou l'évolution d'une partie d'un système. Tout fait significatif pour un traitement. Désigne en général un fait dans le traitement des données ; s'il est attendu par une tâche, il permet à celle-ci de commencer ou de se poursuivre au moment où il se produit. Par exemple, la fin de l'exécution d'une opération asynchrone, telle qu'une opération d'entrée-sortie, est un événement.

En anglais : *event*.

Explication de contradiction

En anglais : *nogood*.

Dans la programmation par contraintes avec explications, une explication de contradiction est un sous-ensemble des contraintes du problème (contraintes originelles définies par le système de contraintes *et* contraintes de décision ajoutées lors de la recherche de solutions) qui conduit à une contradiction : aucune solution au problème ne peut être trouvée avec cet ensemble de contraintes originelles et de contraintes de décisions [Jussien, 2001a ; Jussien, 2001b].

Voir aussi sous-section 4.3.2 page 160.

Explication de retrait

En anglais : *removal explanation*.

Dans la programmation par contraintes avec explications, une explication de retrait est un sous-ensemble des contraintes du problème (contraintes originelles définies par le système de contraintes *et* contraintes de décision ajoutées lors de la recherche de solutions) qui justifie le retrait d'une valeur du domaine d'une variable : cette valeur ne permet pas de satisfaire les contraintes de l'explication de retrait.

Voir aussi sous-section 4.3.2 page 160.

F

Filtrage

En anglais : *filtering*.

Le filtrage est la technique utilisée pour maintenir la cohérence d'un problème de satisfaction de contraintes. Le filtrage retire les valeurs des domaines des variables du problème qui ne satisfont pas les contraintes du problème.

Voir aussi sous-section 4.3.1 page 155.

Formalisme

De [Office québécois de la langue française, 2003], *formalisme à base d'objets* : formalisme utilisé en représentation des connaissances dans lequel les objets et leurs attributs constituent les éléments de base de la connaissance.

En anglais : *formalism*.

Nous utilisons le mot *formalisme* pour parler du moyen utilisé pour représenter un programme : un formalisme peut être un langage de programmation, tel JAVA, un diagramme de classes, comme dans le notation UML, etc.

Forme approchée

En anglais : *weaker form*.

Une micro-architecture similaire au modèle d'un motif de conception est une forme approchée si ses constituants et leurs relations ne sont pas en tout point similaires au modèle du motif, par exemple si deux constituants de la micro-architecture ne sont pas liés par la relation préconisée par le motif.

Voir aussi section 1.2 page 14.

Forme complète

En anglais : *complete form*.

Une micro-architecture similaire au modèle d'un motif de conception est une forme complète du motif si les constituants de la micro-architecture et leurs relations sont en tout point similaires aux constituants du modèle du motif et à leurs relations.

Voir aussi section 1.2 page 14.

M**Maintenance**

De [Larousse, 1995, page 621], *maintenance* : **1.** Ensemble des opérations permettant de maintenir ou de rétablir un système, un matériel, un appareil, etc. dans un état donné ou de lui restituer des caractéristiques de fonctionnement spécifiées.

De [Office québécois de la langue française, 2003], *maintenance* : ensemble des opérations jugées nécessaires pour garantir en tout temps le bon fonctionnement d'un système informatique, conformément à des spécifications définies.

D'après le standard ANSI/IEEE 729-1983, la phase de maintenance consiste en la modification d'un logiciel après livraison pour corriger des erreurs, pour améliorer les performances et autres attributs, ou pour adapter le logiciel à son environnement.

Mainteneur

De [Larousse, 1995, page 621], *mainteneur* : **1.** Litt. Personne soutenant, maintenant quelque chose qui est menacé de disparaître.

En anglais : *maintainer*.

Les mainteneurs sont des développeurs spécialisés qui travaillent sur les programmes après leur déploiement, pendant leur maintenance.

Micro-architecture

De [Office québécois de la langue française, 2003], *micro-architecture* : architecture logicielle déterminée par la perspective fragmentée selon laquelle on considère la *structure** des programmes. La micro-architecture est associée aux modèles de conception qui répondent à un problème technique donné, par opposition à la macro-architecture qui considère les problèmes dans leur ensemble.

En anglais : *micro-architecture*.

Une micro-architecture est un sous-ensemble des *constituants** du *modèle** d'un programme et leur relations.

Modèle

De [Larousse, 1995, page 664], *modèle* : **II. Didact. 1.** Structure formalisée utilisée pour rendre compte d'un ensemble de phénomènes qui possèdent entre eux certaines relations.

En anglais : *model*.

Modèle abstrait

En anglais : *abstract model*.

Le modèle abstrait d'un motif de conception décrit les participants du motif et leurs relations dans "l'absolu", avant tout paramétrage ou adaptation d'un modèle du motif à un contexte particulier, par opposition à un *modèle concret**.

Voir aussi section 4.2 page 148.

Modèle concret

En anglais : *concrete model*.

Le modèle concret d'un motif de conception décrit les participants du motif et leurs relations adaptés à un contexte particulier, par opposition à un *modèle abstrait**.

Voir aussi section 4.2 page 148.

Modèle dynamique

En anglais : *dynamic model*.

Un modèle dynamique décrit le comportement du programme au niveau implémentation, par exemple comme une trace des événements d'exécution du programme, par opposition à un *modèle statique**.

Voir aussi 3.1 page 79.

Modèle statique

En anglais : *static model*.

Un modèle statique décrit la structure du programme au niveau implémentation, par exemple sous la forme d'une suite d'instructions en JAVA, par opposition à un *modèle dynamique**.

Voir aussi 3.1 page 79.

Motif

De [Larousse, 1995, page 676], *motif* : **II. 1. a.** Thème, structure ornementale qui, le plus souvent, se répète. **b.** MUS. Dessin mélodique ou rythmique, plus ou moins long et pouvant, dans le développement de l'œuvre, subir des modifications ou des transpositions.

En anglais : *pattern*.

Motif de conception

De [Office québécois de la langue française, 2003], *motif de conception* : en programmation par objets, infrastructure logicielle constituée d'un petit ensemble de classes portant un nom unique et qui répond à un problème technique connu.

En anglais : *design motif*.

La définition proposée par [Office québécois de la langue française, 2003] est à rapprocher de la définition donnée dans [Gamma *et al.*, 1994, page 2] : "A *design pattern*

names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.”

Cependant, nous utilisons *motif de conception* avec un sens bien particulier pour distinguer un ensemble {*nom, problème, solution, compromis*}, un *patron de conception*^{*}, et la solution proposée par ce patron de conception, un motif de conception. Ainsi, un motif de conception désigne spécifiquement la solution préconisée par un patron de conception.

Voir aussi *motif*^{*}, *patron de conception*^{*} et section 1.1 page 11.

Motif Agrégation

En anglais : *aggregation*.

La relation d'agrégation proposée par les méthodes et les notations de conception, comme UML, est un motif au niveau **idiomatique** : cette relation s'exprime par un unique constituant au niveau **idiomatique** (représenté par un symbole unique) et abstrait un ensemble de constituants du niveau **implémentation**.

Voir aussi section 3.2 page 84 pour les définitions du motif **Agrégation** aux niveaux **implémentation** et **conception** et section 3.4 page 101 pour la définition au niveau **implémentation** avec les propriétés définies dans la section 3.3 page 92.

Motif Association

En anglais : *association*.

La relation d'association proposée par les méthodes et les notations de conception, comme UML, est un motif au niveau **idiomatique** : cette relation s'exprime par un unique constituant au niveau **idiomatique** (représenté par un symbole unique) et abstrait un ensemble de constituants du niveau **implémentation**.

Voir aussi section 3.2 page 84 pour les définitions du motif **Association** aux niveaux **implémentation** et **conception** et section 3.4 page 101 pour la définition au niveau **implémentation** avec les propriétés définies dans la section 3.3 page 92.

Motif Composition

En anglais : *composition*.

La relation de composition proposée par les méthodes et les notations de conception, comme UML, est un motif au niveau **idiomatique** : cette relation s'exprime par un unique constituant au niveau **idiomatique** (représenté par un symbole unique) et abstrait un ensemble de constituants du niveau **implémentation**.

Voir aussi section 3.2 page 84 pour les définitions du motif **Composition** aux niveaux **implémentation** et **conception** et section 3.4 page 101 pour la définition au niveau **implémentation** avec les propriétés définies dans la section 3.3 page 92.

N

Niveau d'abstraction

De [Larousse, 1995, page 699], *niveau* : **II. 1.** Valeur de quelque chose, de quelqu'un ;

degré atteint dans un domaine. **4.** Échelon d'un ensemble organisé, position dans une hiérarchie ; et de [Larousse, 1995, page 29], *abstraction* : **1.** Action d'abstraire ; résultat de cette action.

En anglais : *abstract level*.

Un niveau d'abstraction correspond à une *phase** du cycle de développement d'un programme. Le niveau d'abstraction le plus concret (bas niveau) est le code source du programme (bien que l'on puisse considérer des abstractions de niveaux encore plus bas, tel le code compilé) ; ensuite, les niveaux d'abstraction se succèdent du plus concret au plus abstrait : architecture, conception, analyse du domaine, analyse des besoins, etc. [Booch, 1993].

Voir aussi la méthode OMT, [Rumbaugh *et al.*, 1991, page 455], qui définit une abstraction comme la “*capacité mentale permettant aux humains de se représenter les problèmes du monde réel avec un degré variable de détail en fonction du contexte courant du problème.*” et section 1.1 page 11.

Niveau idiomatique

En anglais : *interclass level*.

Nous introduisons le niveau d'abstraction **idiomatique** entre les niveaux **implémentation** et **conception**. Ce niveau nous permet de décomposer le problème de la traçabilité des motifs de conception en deux phases : une première phase d'analyse du programme pour en construire un modèle au niveau **idiomatique**, dans lequel les relations d'association, d'agrégation et de composition entre classes sont précisément décrites, et une seconde phase d'analyse du modèle du programme au niveau **idiomatique** pour identifier les micro-architectures similaires à des motifs de conception et pour construire un modèle du programme au niveau **conception**, dans lequel les motifs de conception identifiés sont représentés.

P

Patron

De [Larousse, 1995, page 756], *patron* : **1.** Modèle (en tissu, en papier fort, etc.) d'après lequel on taille un vêtement. **2.** Modèle servant à exécuter certains travaux d'artisanat, d'arts décoratifs. **3.** Pochoir pour le coloriage.

En anglais : *pattern*.

Patron de conception

En anglais : *design pattern*.

Un patron de conception identifie, nomme et décrit une solution simple et élégante à un problème récurrent de conception en programmation par objets [Gamma *et al.*, 1994]. Nous utilisons *patron de conception* pour distinguer un ensemble {*nom, problème, solution, compromis*}, qui définit un patron de conception et son utilisation [Coplien, 1998], du résultat de ce processus, la solution proposée, qui est un *motif de conception**. Cette utilisation des mots *patron* et *motif* essaie de concilier la terminologie anglaise (*pattern*), la distinction nécessaire entre un processus (un *patron*) et le résultat du

processus (un *motif*) et notre goût pour la musique (voir *motif**).

PDL

PDL : langage de description des motifs de conception.

En anglais : *Pattern Description Language*.

Phase

De [Larousse, 1995, page 774], *phase* : **1.** Chacun des changements, des aspects successifs d'un phénomène en évolution ; chacun des intervalles de temps marqués par ces changements.

En anglais : *phase*.

Partie distincte d'un processus dans lequel diverses opérations sont successivement effectuées.

Problème de satisfaction de contraintes

En anglais : *constraint satisfaction problem*.

Un problème de satisfaction de contraintes est un triplet $\mathcal{R} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, où :

- $\mathcal{V} = \{v_1, \dots, v_n\}$ est un ensemble de n variables ;
- $\mathcal{D} = \{D_1, \dots, D_n\}$ est un ensemble de n domaines finis. Un domaine D_i représente l'ensemble des valeurs possibles pour la variable v_i ;
- $\mathcal{C} = \{C_1, \dots, C_e\}$ est un ensemble de e contraintes entre les variables du problème.

Ce triplet est la base du raisonnement par contraintes.

Voir aussi sous-section 4.3.1 page 155.

Programmation par contraintes

De [Office québécois de la langue française, 2003], *programmation par contraintes* : en programmation par contraintes, la connaissance est exprimée sous la forme d'ensembles de contraintes et la résolution de problèmes est basée sur des techniques de propagation (renforcement ou relaxation de contraintes).

En anglais : *constraint programming*.

La programmation par contraintes est un sujet de recherche au carrefour des travaux de divers domaines, comme les mathématiques discrètes, l'analyse numérique, l'intelligence artificielle, la programmation mathématique (ou plus généralement la recherche opérationnelle) et le calcul formel. L'hypothèse de base de la programmation par contraintes, qui est la recherche de solutions dans un domaine borné, est souvent vérifiée dans la pratique où les inconnues sont des grandeurs physiques [Jussien, 2003].

Voir aussi sous-section 4.3.1 page 155.

Programmation par contraintes avec explications

En anglais : *explanation-based constraint programming (e-Constraints)*.

La programmation par contraintes avec explications est un nouveau paradigme de résolution des *problèmes de satisfaction de contraintes**. Dans ce paradigme, qui raffine le paradigme de *programmation par contraintes**, le solveur fournit une explication de son comportement à chaque phase de la résolution d'un problème.

Ainsi, la programmation par contraintes avec explications rend possible l'explication du comportement du solveur, la justification des solutions trouvées et la construction

de nouvelles stratégies de recherche [Jussien, 2001b].

Voir aussi sous-section 4.3.2 page 160.

Programmeur

De [Office québécois de la langue française, 2003], *programmeur* : spécialiste qui traduit les opérations que l'ordinateur doit effectuer en instructions que ce dernier peut comprendre.

En anglais : *programmer*.

Nous préférons le mot *développeur*^{*} au mot *programmeur* en relation au mot *développement*^{*}.

Propriété

De [Larousse, 1995, page 830], *proptiété* : **3.** Ce qui est le propre, la qualité particulière de quelque chose.

En anglais : *property*.

Nous utilisons le mot propriété pour parler des caractéristiques des motifs interclasses : durée de vie, exclusivité, multiplicité et site d'invocation. Nous préférons le mot *propriété* au mot *composante* [Larousse, 1995, page 252] : “*Elément constitutif. 2. MÉCAN. Chacune des forces qui interviennent dans la formation d’une résultante. 3. MATH. Coordonnées d’un vecteur dans une base.*”

Propriété de durée de vie

En anglais : *lifetime property*.

La *propriété*^{*} de durée de vie est une des quatre propriétés minimales qui caractérisent les motifs interclasses **Association**, **Agrégation** et **Composition**. Elle contraint les durées de vie des instances des classes liées par le motif **Composition**.

Voir aussi les sections 3.3 page 92 et 3.4 page 101.

Propriété de multiplicité

En anglais : *multiplicity property*.

La *propriété*^{*} de multiplicité est une des quatre propriétés minimales qui caractérisent les motifs interclasses **Association**, **Agrégation** et **Composition**. Elle contraint les nombres des instances des classes liées par ces motifs interclasses.

Voir aussi les sections 3.3 page 92 et 3.4 page 101.

Propriété de site d'invocation

En anglais : *invocation site property*.

La *propriété*^{*} de site d'invocation est une des quatre propriétés minimales qui caractérisent les motifs interclasses **Association**, **Agrégation** et **Composition**. Elle indique si les instances de classes liées par ces motifs interclasses peuvent s'envoyer des messages et les sites d'invocation des méthodes correspondantes (champs, variable locale, etc.).

Voir aussi les sections 3.3 page 92 et 3.4 page 101.

Propriété d'exclusivité

En anglais : *exclusivity property*.

La *propriété*^{*} d'exclusivité est une des quatre propriétés minimales qui caractérisent les motifs interclasses **Association**, **Agrégation** et **Composition**. Elle contraint l'appartenance

des instances d'une classe aux instances d'une autre classe quand les deux classes sont liées par le motif **Agrégation** ou **Composition**.

Voir aussi les sections 3.3 page 92 et 3.4 page 101.

Ptidej

PTIDEJ : identification (de donner une identité, modéliser), détection et amélioration de motifs en JAVA.

En anglais : *Pattern Trace Identification, Detection, and Enhancement in JAVA*.

R

Rétroconception

De [Office québécois de la langue française, 2003], *rétroconception* : pratique qui consiste à analyser un produit fini (comme un logiciel d'application ou une puce) pour connaître la manière dont celui-ci a été conçu ou fabriqué. Bien que l'on utilise le plus souvent la rétroconception dans le but de copier un produit, en tout ou en partie, ou de s'en inspirer pour réaliser un produit concurrent, on peut également la pratiquer pour transférer un logiciel existant d'une plate-forme à une autre et dans un autre langage de programmation. En effet, lorsque la documentation d'un logiciel est déficiente, la rétroconception peut être utile pour restituer les exigences conceptuelles originelles d'un produit. La rétroconception, courante dans l'industrie de l'informatique, et parfaitement légale, est considérée par certains comme de l'espionnage industriel et du piratage.

En anglais : *reverse engineering*.

Voir aussi [Samuelson, 1990 ; Samuelson, 2002].

S

Solution approchée

En anglais : *weaker solution*.

Une solution approchée à un *problème de satisfaction de contraintes** est une solution obtenue après la relaxation de certaines contraintes ou du problème, par opposition à une *solution complète**. Cette notion de solution approchée est à rapprocher de la notation de solution de mauvaise qualité dans les problèmes de satisfaction de contraintes valués [Schiex *et al.*, 1995].

Voir aussi section 4.5 page 169.

Solution complète

En anglais : *complete solution*.

Une solution complète à un *problème de satisfaction de contraintes** est une solution qui satisfait toutes les contraintes du problème, par opposition à une *solution approchée**. Cette notion de solution complète est à rapprocher de la notation de solution de bonne qualité dans les problèmes de satisfaction de contraintes valués [Schiex *et al.*, 1995].

Voir aussi section 4.5 page 169.

Structure

De [Larousse, 1995, page 965], *structure* : **1.** Manière dont les parties d'un ensemble concret ou abstrait sont arrangées entre elles ; disposition. **2.** Organisation, système complexe considéré dans ses éléments fondamentaux.

De [Office québécois de la langue française, 2003], *structure* : organisation d'objets, matériels, logiciels ou d'information. Dans le cas de l'information, par exemple, on dit que les structures de données sont les vecteurs, les matrices, les files, les listes, etc.

En anglais : *structure*.

La structure d'un programme décrit le type et l'organisation de ses constituants et de leurs relations.

T**Traçabilité**

De [Office québécois de la langue française, 2003], *traçabilité* : procédé visant à retracer et identifier, au moyen d'identifications enregistrées, l'origine des matériaux et les pièces inhérentes à un système robotique complexe.

En anglais : *traceability*.

La *traçabilité* [Soukup, 1995 ; Bosch, 1998 ; Antoniol *et al.*, 2001] est notre préoccupation principale : d'une part l'identification, dans les modèles d'un programme aux niveaux **implémentation** et **idiomatique**, des constituants qui forment des micro-architectures similaires à des motifs interclasses et de conception et, d'autre part, les liens entre ces constituants entre les niveaux d'abstraction **implémentation**, **idiomatique** et **conception**. La traçabilité aide les mainteneurs à comprendre le programme en maintenance, les choix d'implémentation et de conception réalisés.

Voir aussi section 1.1 page 11.

Listes

Liste des algorithmes

3.1	Calcul de la valeur de la propriété $MU(A, B)$	117
3.2	Calcul basé sur les accesseurs de la valeur de la propriété $MU(A, B)$ pour une collection homogène	118
3.3	Calcul de la valeur de la propriété $SI(A, B)$	119
3.4	Calcul de la valeur de la propriété $EX(A, B)$	120
3.5	Calcul de la valeur de la propriété $DV(A, B)$	121
3.6	Construction du modèle statique du programme au niveau idiomatique . . .	124
3.7	Construction du modèle complet du programme au niveau idiomatique . . .	125
4.1	Résolution générique d'un problème de satisfaction de contraintes avec ex- plications	163
4.2	Gestion des contradictions	164
4.3	Gestion générique des contradictions	171
4.4	Recherche automatique de toutes les solutions	172
4.5	Choix des contraintes à ajouter et à retirer	173
4.6	Algorithme de relaxation des contraintes	175
4.7	Algorithme de création d'une contrainte affaiblie	175
4.8	Construction du modèle du programme au niveau conception	177

Liste des codes source

2.1	Extrait de la classe CompositeFigure	43
2.2	Deux classes liées par une relation d'héritage	45
2.3	Code source généré par ROSE pour les motifs interclasses Association , Agré- gation et Composition , avec les motifs interclasses représentés sur les fi- gures 2.10(a), 2.10(b) et 2.10(c)	48
2.4	Code source généré par ROSE pour les relations d'association, d'agrégation et de composition, modifié pour utiliser une collection	48
2.5	Prédicat d'identification du motif de conception Composite avec SOUL . . .	62

3.1	Implantation d'une relation d'association	85
3.2	Implantation d'une relation d'agrégation	86
3.3	Implantation d'une relation de composition en JAVA	87
3.4	Implantation d'une relation de composition en C++	89
5.1	Description d'un sous-ensemble du modèle du programme JHOTDRAW au niveau idiomatique	204
5.2	Visualisation d'un sous-ensemble du modèle du programme JHOTDRAW au niveau idiomatique	207
6.1	Méthode recognize() simplifiée du constituant Classe	213
6.2	Méthode build() simplifiée du constituant IdiomLevelModel	214
6.3	Prédicat de vérification de la propriété de durée de vie $DV(A, B)$	220
6.4	Prédicat de vérification des affectations à la réception d'un événement de ramassage	222
6.5	Prédicat de conversion des affectations à la réception d'un événement de ramassage ou de terminaison du programme	222
6.6	Prédicat de vérification de la propriété d'exclusivité $EX(A, B)$	224
6.7	Prédicat de mise à jour des termes exclusivityProperty/5	225
6.8	Prédicats simplifiés pour vérifier l'existence d'un motif Composition	226
6.9	Lanceur CAFFEINE pour identifier le motif Composite entre les classes DrawApplication et StandardDrawingView de JHOTDRAW	229
6.10	Extrait du code source simplifié pour remplacer un motif Agrégation par un motif Composition	231
7.1	Programme de résolution d'un problème de satisfaction de contraintes en CHOCO	236
7.2	Programme de résolution d'un problème de satisfaction de contraintes avec explications en PALM	236
7.3	Constructeur de la classe PtidejVar	239
7.4	Définition simplifiée de la classe Entity	241
7.5	Implantation de la méthode awake()	245
7.6	Implantation simplifiée de la méthode selectDecisionToUndo() pour le solveur interactif	247
7.7	Implantation simplifiée de la méthode combinatorialAutomaticSolve() pour le solveur automatique	249
7.8	Implantation simplifiée de la méthode selectDecisionToUndo() pour le solveur automatique	251
8.1	Déclaration simplifiée du constituant Composite	254
8.2	Extrait de l'implantation du visiteur pour générer le domaine des variables d'un modèle du programme au niveau idiomatique	257
8.3	Extrait du domaine des variables généré pour le modèle du programme JHOTDRAW au niveau idiomatique	258

8.4	Extrait de l'implantation du visiteur pour générer le système de contraintes du modèle d'un motif	259
8.5	Extrait du problème de satisfaction de contraintes généré pour le modèle du motif de conception Composite	262
8.6	Algorithme simplifié d'instanciation des micro-architectures	269

Liste des définitions

3.1	Relation d'association au niveau idiomatique	84
3.2	Relation d'association au niveau implémentation	85
3.3	Relation d'agrégation au niveau idiomatique	85
3.4	Relation d'agrégation au niveau implémentation	86
3.5	Relation de composition au niveau idiomatique	86
3.6	Relation de composition au niveau implémentation	87
3.7	Motif Association $AS(A, B)$	101
3.8	Motif Agrégation $AG(A, B)$	102
3.9	Motif Composition $CO(A, B)$	103
4.1	Problème combinatoire	155
4.2	Problème de satisfaction de contraintes	156
4.3	Instanciation	156
4.4	Solution	158
4.5	Résolution	158
4.6	Cohérence	158
4.7	Problème sur-contraint et contradiction	159
4.8	Relaxation du problème et de contraintes	159

Liste des exemples

3.1	Exemples + et – de la propriété de durée de vie	92
3.2	Exemple <i>faux</i> de la propriété d'exclusivité	93
3.3	Exemple <i>vrai</i> de la propriété d'exclusivité	93
3.4	Exemples de la propriété de multiplicité	94
3.5	Exemples de la propriété de site d'invocation	95
3.6	Durée de vie $DV(Classe)$	97
3.7	Exclusivité $EX(A, B)$	98
3.8	Multiplicité $MU(A, B)$	98
3.9	Site d'invocation $SI(A, B)$	100
3.10	Implantation du motif Association	107
3.11	Autre implantation du motif Association	107
3.12	Implantation du motif Agrégation	108
3.13	Implantation du motif Composition	109

3.14	Autre implantation du motif Composition	111
3.15	Implantation de deux motifs Composition croisés	111
3.16	Contre-exemple au motif Composition	113
3.17	Contre-exemple à deux motifs Composition croisés	113
3.18	Autre contre-exemple à deux motifs Composition croisés	114
4.1	Contrainte d'héritage strict	166
4.2	Contrainte de connaissance	167

Table des figures

1.1	Exemples de formalismes pour décrire les niveaux d'abstraction	12
1.2	Phases de la compréhension avec les patrons	15
1.3	Formalismes pour décrire les niveaux implémentation , idiomatique , conception et analyse	18
1.4	Modèle du motif de conception Médiateur au niveau idiomatique décrit par un diagramme de classes	20
1.5	Identification et traçabilité des relations interclasses et des motifs de conception	21
1.6	Modèle de JHOTDRAW au niveau conception , le diagramme de collaboration représente le motif de conception Composite	24
1.7	Modèles du programme JHOTDRAW : du niveau implémentation au niveau idiomatique ; du niveau idiomatique au niveau conception	25
(a)	Modèle de JHOTDRAW au niveau implémentation	25
(b)	Modèle de JHOTDRAW au niveau idiomatique	25
(c)	Modèle du motif de conception Composite au niveau idiomatique	25
(d)	Modèle de JHOTDRAW au niveau conception avec le motif Composite	25
1.8	Modèle de JHOTDRAW au niveau conception	29
1.9	Modèle de JHOTDRAW obtenu automatiquement	30
2.1	Taxonomie des techniques de rétroconception	33
2.2	Dimensions sémantiques de la qualité des résultats de la rétroconception	35
(a)	Distance sémantique	35
(b)	Précision sémantique	35
(c)	Niveau de détails sémantiques	35
(d)	Continuité sémantique	35
2.3	Quantification de la distance sémantique	36
2.4	Techniques de rétroconception des motifs interclasses	37
2.5	Diagramme de collaboration représentant le cliché d'identification de liens de lecture vers un objet	39
2.6	Diagramme de collaboration représentant le cliché d'identification d'itération sur des ensembles d'objets	41

2.7	Diagramme de collaboration représentant le cliché flou d'identification de liens d'accès à un ou plusieurs objets	41
2.8	Modèle à objets obtenu par la rétroconception de la classe CompositeFigure avec WOMBLE	43
2.9	Modèle au niveau idiomatique des classes présentées sur l'extrait de code source 2.2 par rétroconception avec ARGOUML	46
2.10	Motifs interclasses	47
(a)	Association.	47
(b)	Agrégation.	47
(c)	Composition.	47
(d)	Diagramme de classes erroné.	47
2.11	Techniques de rétroconception des motifs de conception	60
3.1	Sous-ensemble du métamodèle pour décrire l'architecture d'un programme au niveau idiomatique	81
3.2	Notation pour représenter un modèle au niveau idiomatique	84
(a)	Entité, méthodes et champs.	84
(b)	Association.	84
(c)	Agrégation.	84
(d)	Composition.	84
3.3	Relation d'inclusion entre les motifs interclasses	105
3.4	Modèle de JHOTDRAW au niveau implémentation	137
3.5	Modèle de JHOTDRAW au niveau idiomatique obtenu automatiquement	138
3.6	Modèle de JHOTDRAW au niveau idiomatique fourni avec la documentation du programme	139
3.7	Correspondances entre les modèles de JHOTDRAW au niveau idiomatique fourni avec la documentation du programme et obtenu automatiquement	140
4.1	Sous-ensemble du métamodèle pour décrire le modèle d'un programme au niveau conception	146
4.2	Sous-ensemble du métamodèle pour décrire un motif de conception	150
4.3	Processus de description des modèles abstraits des motifs de conception	152
4.4	Modèle du motif de conception Composite et notation	152
(a)	Description du motif de conception Composite	152
(b)	Notation du métamodèle	152
4.5	Modèle de JHOTDRAW au niveau idiomatique obtenu automatiquement	185
4.6	Motif de conception Composite	186
(a)	Motif de conception Composite	186
(b)	Modèle du motif de conception Composite au niveau idiomatique	186
4.7	Interactions avec le solveur de contraintes pour identifier les micro-architectures similaires au modèle du motif de conception Composite	186
4.8	Modèle de JHOTDRAW au niveau conception obtenu automatiquement avec une micro-architecture similaire au motif de conception Composite mise en valeur	187

4.9	Modèle de JHOTDRAW au niveau conception fourni avec la documentation du programme	188
4.10	Correspondances entre les modèles de JHOTDRAW au niveau conception fourni avec la documentation du programme et obtenu automatiquement . .	189
5.1	Noyau du métamodèle PDL	198
5.2	Sous-ensemble simplifié du métamodèle PDL	200
5.3	Sous-ensemble simplifié du métamodèle PADL après restructuration du métamodèle PDL	209
5.4	Sous-ensemble simplifié du métamodèle PADL après extension	210
5.5	Modèle graphique d'un sous-ensemble du modèle du programme JHOTDRAW au niveau idiomatique	210
6.1	Démarrage de CAFFEINE pour analyser le programme JHOTDRAW	230
6.2	Utilisation du programme JHOTDRAW, CAFFEINE analyse les événements générés en arrière-plan	230
6.3	Résultats de l'analyse avec CAFFEINE du programme JHOTDRAW	231
8.1	Système de contraintes en contradiction, le mainteneur guide la recherche .	264
8.2	Reprise de la recherche après rétraction de la contrainte de composition .	266
8.3	Système de contraintes en contradiction, le mainteneur doit retirer une autre contrainte	266
8.4	Recherche des solutions complètes	267
8.5	Recherche des solutions approchées	268
A.6	Approche logicielle	286
A.7	Références	288

Liste des propriétés

3.1	Durée de vie	92
3.2	Exclusivité	92
3.3	Multiplicité	93
3.4	Site d'invocation	94

Liste des tableaux

2.1	Dimensions de la qualité sémantique des techniques présentées d'identification des motifs interclasses	50
2.2	Constructeur du motif de conception Composite	68
3.1	Résultats de l'identification du motif Agrégation	129
3.2	Identification du motif Agrégation dans AWT	131
3.3	Identification du motif Agrégation dans JHOTDRAW	132
3.4	Identification du motif Agrégation dans JUNIT	132

3.5	Identification du motif Composition dans JUNIT avec l'interface textuelle . .	134
3.6	Identification du motif Composition dans JUNIT avec l'interface AWT . . .	134
3.7	Identification du motif Composition dans JUNIT avec l'interface SWING . . .	134
4.1	Schéma général des patrons de conception	153
4.2	Résultats de l'identification de motifs de conception dans le programme JHOTDRAW	182
5.1	Correspondance des relations entre les métamodèles PDL et PADL	201
5.2	Correspondance des constituants entre les métamodèles PDL et PADL . .	202
6.1	Liste des événements possibles dans la trace d'exécution d'un programme .	218
8.1	Solutions obtenues avec le solveur automatique combinatoire	265

Index

A	
Agrégation	92
Partie	92
Tout	92
Aller-retour	16
Architecture	12
ARGOUML	49
C	
CFPARSE	215
CHAVA	46
Code octal	46
Composition	
Partie, Tout	<i>see</i> Agrégation
Compréhension	12
Constituant	13
D	
Découverte	18
Développement	12
Diagramme	18, 131
De Classes	18
E	
EDI	30
Événement	86
Explication	
De contradiction	162
De retrait	163
F	
Filtrage	160
Formalisme	12
Forme	
Approchée	17
Complète	17
J	
BORLAND JBUILDER	50
M	
Métaprogrammation déclarative	65
Maintenance	12
Mainteneur	14
Micro-architecture	14
Modèle	12
Abstrait	152, 255
Concret	152
Dynamique	86, 119, 217
Statique	86, 119, 213
Motif	13, 95
Motif de conception	17
Association	91
Agrégation	92
Composition	93
N	
Niveau d'abstraction	12
idiomatique	19
P	
Patron	13
Patron de conception	17
PDL	29
Programmation	
Par contraintes	158
Avec explications	162
Propriété	97
Durée de vie	97
Exclusivité	98

Multiplicité	98
Site d'invocation	99
PSC	158
PTIDEJ	196

R

Rétroconception	12
RATIONAL ROSE	50

S

Solution	
Approchée	169
Complète	169
Solveur de contraintes	160, 169
PALM	235
PTIDEJ SOLVER	238
Structure	14

T

TOGETHERSOFT TOGETHER	50
Traçabilité	16

W

WOMBLE	47
--------------	----

*I have seen things you people would not believe.
Attack ships on fire off the shoulder of Orion.
I watched C-beams glitter in the dark near the
Tannhauser gate.
All those moments will be lost in time like tears
in the rain.
Time to die.*

Roy Batty, extrait du film “*Blade Runner*” (1982) inspiré
du livre “*Do androids dream of electric sheep*” (1968) de
Philip K. Dick.

JAVA et toutes les marques basées sur JAVA sont des
marques déposées par SUN MICROSYSTEMS, INC. aux
États-Unis d’Amérique et dans les autres pays.
Tous les autres noms de marques, noms de produits et
marques déposées appartiennent à leurs propriétaires
respectifs.

Ce mémoire a été composé en 637 heures avec ECLIPSE
v2.1, IMAGEMAGICK v5.2.9, LGRIND v3.6, PAINT v5.0,
SMARTDRAW v6.11, T_EX v3.14159 (MikTEX v2),
WINBIBDB v2.2 et WINEDT v5.4 (20030401) sur WIN-
DOWS 2000 v5.00.2195SP3.

Version 1.0.6, le 22 novembre 2005 à 15 h 19.

Tous droits réservés.

Un cadre pour la traçabilité des motifs de conception

Les patrons de conception sont importants en génie logiciel à objets car ils contribuent à la qualité des programmes. Ils proposent des solutions élégantes à des problèmes récurrents de conception, des motifs utilisés pendant l'implantation. À l'usage, ces motifs de conception sont disséminés dans le code source et ne sont plus explicites lors de la maintenance; pourtant, ils aideraient à comprendre l'implantation et la conception des programmes, assurant leur qualité. Ce mémoire propose des modèles et des algorithmes pour garantir la traçabilité des motifs de conception entre les phases d'implantation et de rétroconception des programmes par l'identification semi-automatique des micro-architectures similaires à ces motifs dans le code source.

La métamodélisation est utilisée pour décrire les motifs de conception et les programmes JAVA. Elle amène à expliciter certaines relations interclasses (association, agrégation et composition) offertes par les langages de conception comme UML et à préciser leurs propriétés (durée de vie, exclusivité, multiplicité et site d'invocation) pour les identifier avec des algorithmes d'analyses statiques et dynamiques. Elle conduit aussi à traduire les motifs en systèmes de contraintes et à identifier les micro-architectures similaires, formes complètes et approchées, par la résolution de problèmes de satisfaction de contraintes. La programmation par contraintes avec explications permet de guider la résolution et d'expliquer les micro-architectures identifiées.

La suite d'outils PTIDEJ est une implantation des modèles et des algorithmes proposés. Elle est intégrée à l'environnement ECLIPSE de développement en JAVA. Elle inclut le métamodèle PADL, dérivé du métamodèle PDL; des outils d'analyses statiques et dynamiques, INTROSPECTOR et CAFFEINE; et un solveur de contraintes, PTIDEJ SOLVER, dérivé du solveur de contraintes avec explications de référence PALM.

Mots clé : génie logiciel à objets, rétroconception, qualité, patrons de conception, traçabilité, relations interclasses, programmation par contraintes avec explications, PTIDEJ.

A framework for design motif traceability

Design patterns are important in object-oriented software engineering. They contribute to the quality of programs. They offer design motifs, elegant solutions to recurrent design problems. After implementation, the motifs are disseminated in source code. They are not available directly for maintenance; however they would help in understanding program implementation and design and in ensuring the quality of programs after maintenance. This thesis explores models and algorithms to identify semi-automatically micro-architectures in source code, which are similar to motifs, and to ensure their traceability between implementation and reverse engineering phases.

Meta-modeling is used to describe design motifs and JAVA programs. It leads to characterize certain interclass relations (association, aggregation, and composition) offered by design languages, such as UML, to precise their properties (access type, lifetime, exclusivity, and multiplicity), and to identify them with static and dynamic analyses. It also leads to translate motifs into constraint systems and to identify micro-architectures, which are similar to motifs (complete and distorted forms), by solving constraint satisfaction problems. Explanation-based constraint programming allows guiding the solve interactively and explaining identified micro-architectures.

The PTIDEJ tool suite implements the proposed models and algorithms and is integrated with the ECLIPSE development environment for JAVA. It includes the PADL meta-model, derived from the PDL meta-model; Static and dynamic analysis tools, INTROSPECTOR and CAFFEINE; A constraint solver, PTIDEJ SOLVER, derived from the PALM explanation-based constraint solver.

Keywords: Software engineering, reverse engineering, quality, design patterns, traceability, inter-class relationships, explanation-based constraint programming, PTIDEJ.