Université de Montréal

Développement logiciel par transformation de modèles

par
Ghizlane El boussaidi

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des arts et des sciences
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en Informatique

Juillet, 2009

Université de Montréal
Faculté des arts et des sciences

Cette thèse intitulée :

Développement logiciel par transformation de modèles

présentée par :
Ghizlane El boussaidi

a été évaluée par un jury composé des personnes suivantes :

Président-rapporteur:	Philippe Langlais, Professeur, Université de Montréal
Directeur de recherche:	Houari Sahraoui, Professeur, Université de Montréal
Co-directeur:	Hafedh Mili, Professeur, Université du Québec à Montréal
Membre du jury:	Sylvie Hamel, Professeur, Université de Montréal
Examinateur externe:	Hans Vangheluwe, Professeur, McGill University
Représentant du doyen:	Olivier Gerbé, HEC

Résumé

La recherche en génie logiciel a depuis longtemps tenté de mieux comprendre le processus de développement logiciel, minimalement, pour en reproduire les bonnes pratiques, et idéalement, pour pouvoir le mécaniser. On peut identifier deux approches majeures pour caractériser le processus. La première approche, dite *transformationnelle*, perçoit le processus comme une séquence de transformations préservant certaines propriétés des données à l'entrée. Cette idée a été récemment reprise par l'architecture dirigée par les modèles de l'OMG. La deuxième approche consiste à répertorier et à codifier des solutions éprouvées à des problèmes récurrents. Les recherches sur les styles architecturaux, les patrons de conception, ou les cadres d'applications s'inscrivent dans cette approche. Notre travail de recherche reconnaît la complémentarité des deux approches, notamment pour l'étape de conception: dans le cadre du développement dirigé par les modèles, nous percevons l'étape de conception comme l'application de patrons de solutions aux modèles reçus en entrée.

Il est coutume de définir l'étape de conception en termes de *conception architecturale*, et *conception détaillée*. La conception architecturale se préoccupe d'organiser un logiciel en composants répondant à un ensemble d'exigences non-fonctionnelles, alors que la conception détaillée se préoccupe, en quelque sorte, du contenu de ces composants. La conception architecturale s'appuie sur des *styles architecturaux* qui sont des principes d'organisation permettant d'optimiser certaines qualités, alors que la conception détaillée s'appuie sur des *patrons de conception* pour attribuer les responsabilités aux classes. Les styles architecturaux et les patrons de conception sont des artefacts qui codifient des solutions éprouvées à des problèmes récurrents de conception. Alors que ces artefacts sont bien documentés, la décision de les appliquer reste essentiellement manuelle. De plus, les outils proposés n'offrent pas un support adéquat pour les appliquer à des modèles existants.

Dans cette thèse, nous nous attaquons à la conception détaillée, et plus particulièrement, à la transformation de modèles par application de patrons de conception, en partie parce que les patrons de conception sont moins complexes, et en partie parce que l'implémentation des styles architecturaux passe souvent par les patrons de conception. Ainsi, nous proposons une approche pour représenter et appliquer les patrons de conception. Notre approche se base sur la représentation explicite des problèmes résolus par ces patrons. En effet, la représentation explicite du problème résolu par un patron permet : (1) de mieux comprendre le patron, (2) de reconnaître l'opportunité d'appliquer le patron en détectant une instance de la représentation du problème dans les modèles du système considéré, et (3) d'automatiser l'application du patron en la représentant, de façon déclarative, par une transformation d'une instance du problème en une instance de la solution.

Pour vérifier et valider notre approche, nous l'avons utilisée pour représenter et appliquer différents patrons de conception et nous avons effectué des tests pratiques sur des modèles générés à partir de logiciels libres.

Mots-clés : Patrons de conception, Problèmes de conception, Détection, Marquage de modèles, Transformation de modèles, Méta-modélisation.

Abstract

Software engineering researchers have long tried to understand the software process development to mechanize it or at least to codify its good practices. We identify two major approaches to characterize the process. The first approach—known as transformational—sees the process as a sequence of property-preserving transformations. This idea was recently adopted by the OMG's model-driven architecture (MDA). The second approach consists in identifying and codifying proven solutions to recurring problems. Research on architectural styles, frameworks and design patterns are part of this approach. Our research recognizes the complementarity of these two approaches, in particular in the design step. Indeed within the model-driven development context, we view software design as the process of applying codified solution patterns to input models.

Software design is typically defined in terms of *architectural design* and *detailed design*. Architectural design aims at organizing the software in modules or components that meet a set of non-functional requirements while detailed design is—in some way—concerned by the contents of the identified components. Architectural design relies on architectural styles which are principles of organization to optimize certain quality requirements, whereas detailed design relies on design patterns to assign responsibilities to classes. Both architectural styles and design patterns are design artifacts that encode proven solutions to recurring design problems. While these design artifacts are documented, the decision to apply them remains essentially manual. Besides, once a decision has been made to use a design artifact, there is no adequate support to apply it to existing models.

As design patterns present an ''*easier*'' problem to solve, and because architectural styles implementation relies on design patterns, our strategy for addressing these issues was to try to solve the problem for design patterns first, and then tackle architectural styles. Hence, in this thesis, we propose an approach for representing and applying design patterns. Our approach is based on an explicit representation of the problems solved by design patterns. Indeed, and explicit representation of the problem solved by a pattern enables to: 1) better understand the pattern, 2) recognize the opportunity of applying the pattern by matching the representation of the problem against the models of the considered system, and 3) specify declaratively the application of the pattern as a transformation of an instance of the problem into an instance of the solution.

To verify and validate the proposed approach, we used it to represent and apply several design patterns. We also conducted practical tests on models generated from open source systems.

Keywords: Design patterns, Design problems, Pattern matching, Model marquing, Model transformation, Meta-modelling.

Remerciements

Je tiens à remercier vivement Hafedh Mili, professeur à l'université du Québec à Montréal et Houari Sahraoui, professeur à l'université de Montréal, pour m'avoir dirigée dans mes travaux de recherche. Tout au long de cette thèse, ils ont su me guider et me conseiller de façon pertinente. Je les remercie pour leur encadrement scientifique et humain.

Le soutien financier dont j'ai bénéficié m'a permise de me concentrer sur mon travail de recherche. Je remercie beaucoup Hafedh Mili pour son soutien financier tout au long de mon projet de thèse. Cette thèse a aussi bénéficié d'une bourse industrielle du CRSNG sponsorisée par SAP Labs Canada. J'aimerai remercier Mme Nolwen Mahé et SAP Labs Canada pour leur collaboration et leur contribution financière à ce projet. Je remercie tous les membres de l'équipe de recherche à SAP Labs Canada pour m'avoir accueillie chaleureusement et intégrée dans leur équipe.

Je remercie les membres du jury pour avoir accepté d'évaluer cette thèse. J'aimerai également remercier Hans Vangheluwe, professeur adjoint à l'université McGill, d'avoir accepté d'être examinateur externe de ma thèse.

Une grande partie de mon travail de recherche s'est faite au sein du laboratoire LATECE où l'ambiance a toujours été sympathique, cordiale et ouverte. Je voudrais remercier mes amis et collègues du LATECE qui ont contribué énormément à mon enrichissement professionnel et personnel. Je remercie mon amie Mélanie pour le temps qu'elle m'a consacrée à discuter de certaines parties de ma thèse.

Mes remerciements vont aussi à toute ma famille. Je remercie, en particulier, ma mère, mes sœurs et mes frères pour leur amour, leur soutien et leur encouragement. Je pense aussi à mon père, décédé en 1978, qui m'a tant donnée en si peu de temps. Je remercie de tout cœur mes deux grand-mères décédées qui ont bien pris soin de moi et qui m'ont donnée énormément.

Enfin, je remercie Robert, mon conjoint et ami, pour son amour, son appui inconditionnel, ses encouragements et sa compréhension.

Table des matières

Liste des tableaux

Liste des figures

Liste des abréviations

ADL	*Architecture Description Language*
CORBA	*Common Object Request Broker Architecture*
CSP	*Constraint Satisfaction Problem*
EMF	*Eclipse Modeling Framework*
LHS	*Left-Hand Side*
MDA	*Model Driven Architecture*
MOF	*Meta Object Facility*
NAC	*Negative Application Condition*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
QVT	*Query View Transformation*
RHS	*Right-Hand Side*
UML	*Unified Modeling Language*
XMI	*XML Metadata Interchange*

Chapitre 1

Introduction

1.1 Problématique

Un des défis majeurs du génie logiciel est de produire un système dont la gestion et la maintenance sont faciles, dont la complexité est contrôlée et dont la réutilisation est favorisée. En effet, en pratique, quel que soit le processus de développement adopté, le passage des spécifications d'un logiciel à sa conception reste ardu et se fait d'une façon qui ne favorise pas la réutilisation. Nous pensons que cela résulte du fait que le processus de développement est un processus complexe qui nécessite beaucoup de connaissances et compétences autant dans le développement logiciel que dans le domaine de l'application. En outre, le processus de développement n'est pas formel. En effet, la construction des logiciels se fait toujours avec la perte des connaissances et informations qui ont permis de passer d'une étape du processus à une autre (i.e. choix de conception). Selon Baxter (Baxter, 1992), la spécification du problème et la justification de la conception sont aussi perdues. Les approches basées sur le développement pour la réutilisation et celles basées sur le développement transformationnel ont été les premières approches proposées pour gérer cette complexité et pour permettre la construction rapide et à moindre coût de logiciels de meilleure qualité. Les approches pour la réutilisation se sont concentrées sur la réutilisation des différents artefacts produits lors du développement logiciel tandis que les approches transformationnelles sont plutôt des approches hybrides, i.e.

elles visent à réutiliser aussi bien les artefacts générés lors du développement que les processus de développement lui-même.

Les approches transformationnelles ont modélisé le processus de développement par une suite de transformations appliquées à des spécifications formelles pour produire à la fin un code exécutable (Partsch et Steinbruggen, 1983) (Baxter, 1992). Plus généralement, le processus de développement peut être vu comme un ensemble de transformations partant des exigences utilisateur et aboutissant à une implémentation spécifique à une plateforme et conforme aux exigences de qualité. Les choix faits durant ce processus mènent à différentes chaînes de transformations. Cette idée a été reprise par l'OMG (*Object Management Group*) avec l'architecture MDA (*model driven architecture*) (MDA, 2003). Alors qu'initialement l'OMG se préoccupait de l'interopérabilité entre systèmes via des mécanismes techniques, il s'oriente maintenant vers la réutilisation, la transformation et l'échange des modèles et de ce fait se positionne non plus en aval mais en amont du développement (Bézivin et Blanc, 2002). En effet, l'architecture MDA se base sur le fait qu'une application est représentée par plusieurs modèles et cela à différentes étapes de sa construction et à différents niveaux d'abstraction, et que les modèles peuvent être obtenus par transformation ou fusion d'autres modèles. Mais en pratique, les transformations ne sont pas complètement automatiques car la correspondance entre les modèles source et cible est incomplète ou non déterministe (Mili et al., 1995). Ainsi, un des défis majeurs est de caractériser les transformations de façon précise pour pouvoir les encoder dans des procédures ou processeurs systématiques que l'on peut appliquer à de nouvelles exigences.

Notre travail de recherche s'intéresse plus particulièrement à l'étape de conception. Il est coutume en génie logiciel de définir la conception en termes de *conception architecturale*, et de *conception détaillée*. Dans la phase de conception architecturale, il s'agit de structurer les éléments du système à concevoir. Les exigences logicielles sont alors transformées en un ensemble de spécifications de composants et de leurs interactions. Cette phase peut être vue comme la projection des modèles d'analyse selon des styles architecturaux. Elle produit en sortie un

modèle architectural du système considéré. En fait, un style architectural peut être considéré comme un patron d'organisation (e.g. organisation par couches, filtres et canaux, etc.) puisqu'il détermine le vocabulaire des composants et connecteurs utilisés dans des instances de ce style, et aussi les contraintes sur la façon de les combiner (Garlan et Shaw, 1994).

Durant la phase de conception détaillée, il faut définir l'ensemble des éléments nécessaires pour guider l'implémentation du système à construire, i.e. définition des interfaces, choix des structures de données et des algorithmes. Il s'agit donc de spécifier les composants à un niveau plus bas et plus concret que la conception architecturale. La conception détaillée implique aussi le choix et l'application de patrons de conception (Gamma et al., 1995). Les patrons de conception représentent les connaissances d'experts en orienté objet dans une forme qui peut être réutilisée par les moins experts. Par leur aspect générique, ils sont considérés comme des micro-architectures qui visent à réduire la complexité, à promouvoir la réutilisation et à fournir un vocabulaire commun aux concepteurs (Johnson, 1997).

Autant les styles architecturaux que les patrons de conception représentent des *moules de conception* qu'on peut appliquer à des modèles. Cependant, la décision d'appliquer un style architectural ou un patron de conception reste essentiellement manuelle. De plus, il n'y a pas de support adéquat pour leur application à un modèle d'analyse donné. Pour cela, nous nous posons les questions suivantes : (1) comment caractériser le type de problème auquel un moule de conception (style architectural ou patron de conception) est associé ?, (2) comment décrire le moule?, et (3) comment l'appliquer à un modèle ?

En pratique, l'utilisation d'un moule de conception consiste à choisir d'abord le moule approprié, ce qui implique sa reconnaissance comme une solution potentielle au problème de conception qu'on essaie de résoudre, et ensuite appliquer le moule choisi au problème considéré, ce qui implique la transformation d'un modèle pour qu'il devienne conforme au moule de conception. Ainsi, il y a deux aspects essentiels à considérer. D'abord, il faut comprendre et caractériser les

problèmes de conception. Ensuite il faut comprendre et maîtriser la mécanique de l'approche par transformation de modèles.

Notons que MDA ne parle pas spécifiquement d'analyse et de conception mais parle plutôt de modèle PIM (*platform independent model*) et de modèle PSM (*platform specific model*). Le modèle PIM est un modèle indépendant des plate-formes, il décrit les fonctions d'un système. Le modèle PSM combine les spécifications du PIM avec des détails spécifiques à une plate-forme particulière. L'architecture MDA recommande de spécifier un modèle PIM, et de le transformer en un modèle PSM spécifique à la plate-forme ciblée (Figure 1.1a). Par rapport au modèle MDA, notre travail se situe dans la transformation du PIM en PSM puisque le modèle PIM n'inclut aucun détail relié à l'architecture du système concerné, alors que le modèle PSM inclut aussi bien l'information concernant le choix des styles architecturaux que les détails d'implémentation reliés à la plate-forme cible. Or, le choix d'un style architectural ne dépend pas d'une plate-forme particulière. C'est un patron abstrait d'organisation qui peut avoir des réalisations sous forme de profils technologiques (e.g. J2EE, .NET). Pour cela, nous pensons qu'il est justifié d'introduire un niveau intermédiaire d'abstraction entre les modèles PIM et PSM (Figure 1.1b). Ce niveau correspondra à un modèle spécifique à un style architectural (ASSM : *Architectural-Style-Specific-Model*). Ce modèle sera obtenu par transformation du modèle PIM par application d'un style architectural. La mécanisation de cette transformation fait partie de la mécanisation du processus de conception architecturale. Le défi est de spécifier précisément un style architectural pour le représenter par un modèle (ASDM : *Architectural-Style-Description-Model*) et caractériser les transformations qui sont associées à son application.

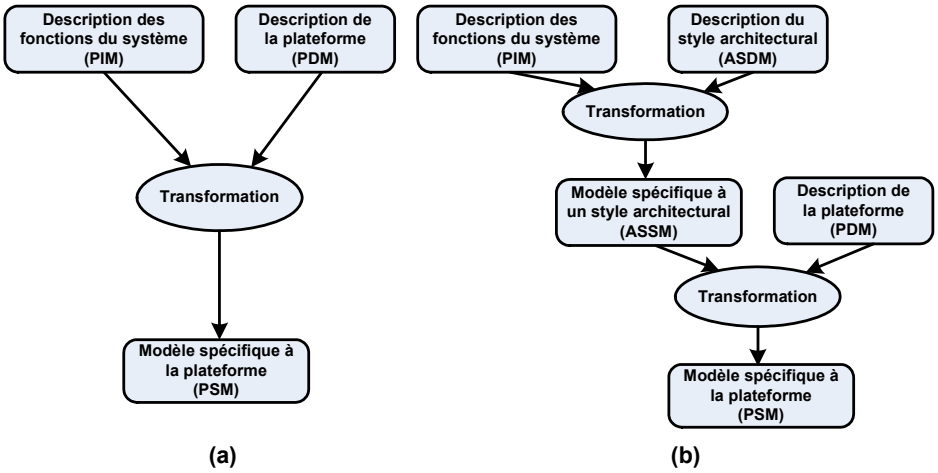


Figure 1.1. Vue globale de l'approche MDA et du contexte de notre recherche

1.2 Objectifs de la recherche

À long terme, le but de cette recherche est de contribuer à la formalisation de la conception de logiciels sous la forme de l'application de patrons de solutions (e.g. style architectural, patron de conception) à des modèles de logiciels. Cette application passe par trois tâches: 1) la compréhension du problème résolu par le patron de solution, 2) la reconnaissance d'une instance du problème dans le modèle logiciel sous considération, et 3) la mise en œuvre systématique de la solution.

Notre stratégie consiste à nous attaquer en premier au problème de ***représentation et mise en œuvre de patrons de conception*** pour, 1) proposer une représentation des patrons qui considère aussi bien le problème résolu par le patron que la solution qu'il propose, 2) comprendre les problèmes inhérents à la représentation des problèmes de conception, et 3) développer une infrastructure générique de transformation de modèles. Les leçons apprises nous suggèreront des façons d'aborder le problème plus complexe des styles architecturaux.

1.3 Contributions de la thèse

La contibution principale de cette thèse est une approche qui s'appuie sur, 1) ***la représentation explicite du problème de conception résolu par le patron de solution***, et 2) ***la mise en œuvre de la solution en tant que transformation de modèle***

de problème à modèle de solution. En effet, la représentation formelle du contexte d'application d'un patron est cruciale pour les trois tâches nécessaires à son application:

1. on ne peut certifier la pertinence d'un patron à une situation donnée sans avoir un modèle du problème; les rubriques textuelles que l'on retrouve dans la documentation des patrons (Gamma et al., 1995) sont insuffisantes, et ne peuvent servir à la mécanisation de cette étape,
2. la compréhension du patron doit illustrer l'« *avant* » et l' « *après* » application du patron. La majorité des approches qui se sont intéressées à la mise en œuvre des patrons de conception se contentent de représenter l' « *après* »,
3. l'application du patron à un fragment de modèle n'est rien d'autre que l'application de transformations à ce fragment. Ces transformations peuvent être encodées, d'une façon générique, comme des transformations du modèle du problème en un modèle de solution.

Par la représentation explicite du problème de conception résolu par un patron sous forme de modèle, notre approche permet de :

- ***Mieux caractériser un patron et son contexte d'utilisation*** : En représentant un patron par une paire de modèles (problème, solution), nous décrivons aussi bien l'état d'un modèle *avant* l'application du patron que l'état *après* son application.
- ***Détecter dans un modèle reçu en entrée les opportunités d'application d'un patron*** : En représentant explicitement le problème résolu par un patron, il est possible d'en détecter les instances dans un modèle donné en entrée.
- ***Automatiser l'application d'un patron*** : En représentant de façon déclarative l'application d'un patron sous forme d'une transformation d'une instance du modèle de problème en une instance du modèle correspondant de la solution.

1.4 Organisation de la thèse

Dans le chapitre 2, nous présentons une revue de littérature qui fait le tour des différents aspects reliés à notre recherche. En particulier, nous parlerons des

approches qui se sont penchées sur l'automatisation du processus de développement, des approches de modélisation des styles architecturaux et des approches pour la représentation et la mise en œuvre des patrons de conception.

Nous donnons une vue globale de notre approche pour la représentation et la mise en oeuvre des patrons au chapitre 3. Nous introduisons ensuite notre représentation des patrons en utilisant des exemples. Nous décrivons aussi les constructions que notre langage doit supporter et la façon dont nous avons implémenté ce langage.

Dans le chapitre 4, nous présentons notre approche pour la détection des instances de modèles de problèmes résolus par les patrons de conception dans des modèles UML. Nous décrivons aussi notre implémentation de cette approche.

Dans le chapitre 5, nous présentons d'abord une première tentative de représentation des transformations basée sur un modèle structurel et discutons des limites de cette représentation. Ensuite, nous présentons notre nouvelle approche pour la représentation et la mise en œuvre des transformations inhérentes à l'application des patrons de conception et nous en décrivons l'implémentation.

Nous présentons au chapitre 6 les résultats de la validation de notre approche. En particulier, nous donnons les résultats de la validation de l'aspect représentation sous forme d'une classification des problèmes résolus par les patrons. Nous présentons la démarche que nous avons adoptée pour valider les aspects détection et transformation et les résultats de cette validation. Nous discutons aussi des modifications que nous avons apportées à l'approche suite aux tests de validation, et des problèmes que nous avons rencontrés lorsque nous transformons des instances chevauchantes de patrons.

Finalement, dans le chapitre 7, nous présentons un bilan de la thèse et évoquons quelques directions futures pour ce travail de recherche.

Chapitre 2

État de l'art

Notre objectif étant d'automatiser le processus de conception, notre revue de littérature commence par donner un aperçu des premières approches qui se sont penchées sur l'automatisation du processus de développement, notamment les approches de programmation automatique et les approches transformationnelles. Nous donnons aussi un aperçu de l'approche MDA qui préconise le développement par transformation de modèles et qui représente un cadre général pour notre travail de recherche. Nous abordons ensuite les phases de conception architecturale et de conception détaillée qui constituent l'essentiel de la phase de conception à laquelle nous nous intéressons. En particulier, nous passons en revue les approches de modélisation des styles architecturaux et des patrons de conception.

2.1 Approche transformationnelle au processus de développement

Le processus de développement logiciel est un ensemble d'activités et de résultats associés qui produisent un logiciel (Sommerville, 2001). La plupart des processus de développement comportent les phases d'analyse, de conception et d'implantation. La durée, les résultats et la décomposition de ces phases varient d'un processus de développement à un autre. Une des premières approches visant l'automatisation d'une partie du processus de développement est la programmation automatique. Cette dernière décrit un processus synthétique qui permet de générer un

programme à partir d'une spécification. En effet, les difficultés rencontrées lors de la construction de programmes peuvent être surmontées si la tâche entière est divisée en des étapes suffisamment petites et formellement justifiées (Partsch et Steinbruggen, 1983). Le but des transformations, dans ce contexte, est d'augmenter la productivité d'un programmeur en automatisant les tâches de programmation. Les systèmes de programmation automatique font une correspondance entre une configuration de termes spécifiques à un domaine et une configuration de termes spécifiques à une implantation (Rich et Waters, 1993). Les méthodes qui ont été les plus utilisées pour réaliser de tels systèmes sont les méthodes transformationnelles. Ces méthodes nécessitent généralement une spécification formelle. Un système de programmation automatique basé sur une méthode transformationnelle reçoit un « programme » écrit en un langage de haut niveau et le convertit en une implémentation de bas niveau en lui appliquant une série de transformations dont la correction est préservée. Le système fonctionne dans ce cas par cycles. À chaque cycle, une transformation est choisie et appliquée. Le processus se termine lorsqu'une condition spécifique est satisfaite. Dans ce contexte, on peut distinguer deux types de transformations : celles qui encodent certaines connaissances nécessaires pour remplacer les constructions de la spécification par les constructions de l'implémentation et celles qui ne changent pas de niveau d'abstraction mais qui visent plutôt à faire des réorganisations ou des optimisations (Rich et Waters, 1993).

Les transformations telles que présentées dans les systèmes de programmation automatique peuvent être généralisées de la programmation à l'ensemble du processus de développement. Le modèle d'un tel système de transformation a été proposé par Baxter (1992). Ce système utilise plusieurs librairies dont les librairies de transformation qui servent à convertir une spécification abstraire en un programme concret. Le système utilise aussi des mesures de performances pour déterminer quelles propriétés sont satisfaites par un programme partiellement transformé. Des relations de substitution sont établies entre ces mesures pour définir la notion d'équivalence, i.e. « au moins aussi bon que ». Une librairie de prédicats combine les mesures de performance et les relations de substitution pour définir des niveaux intéressants de performance. Enfin, une librairie de méthode contient des plans pour

appliquer les transformations pour atteindre des niveaux définis de performance pour des parties de programmes. Ces plans de transformations fournissent un historique de conception qui permet de garder une trace des décisions qui ont été prises tout au long du processus de développement.

Généralement parlant, selon une vue transformationnelle, le développement logiciel est un processus qui consiste en plusieurs étapes. Chaque étape est une transformation (Figure 2.1) de la description d'un problème du niveau i en une description d'un problème du niveau i+1 (Mili et al., 1995). La distance entre ces transformations doit être assez courte pour minimiser les efforts de vérification de ces transformations. Si les vérifications ne détectent pas d'erreurs, il est garanti que l'implantation résultant du processus est conforme à la spécification. Comparée aux approches classiques de développement, l'approche transformationnelle a l'avantage de rendre la distance entre les transformations plus courte que celle existant entre une spécification et un programme (Sommerville, 2001). Ces transformations sont donc plus faciles à encoder.

Figure 2.1. Vue transformationnelle du processus de développement

En outre, une méthode transformationnelle permet aussi bien la réutilisation des différents artefacts générés tout au long du processus de développement que la réutilisation du processus pour le développement d'autres systèmes. Cette idée a été restituée par l'OMG avec l'architecture MDA qui adopte une démarche transformationnelle pour construire les systèmes.

2.2 L'approche MDA

2.2.1 Principes

L'approche MDA peut être considérée comme un cadre "pragmatique" pour la mise en œuvre de l'approche transformationnelle. Elle préconise le développement de logiciels par transformations de modèles (Figure 2.2). L'idée de base est de séparer la spécification des fonctions d'un système de l'implémentation de ces fonctions sur une plateforme spécifique. Ainsi lors du développement d'un système, les fonctionnalités désirées doivent être spécifiées indépendamment de la plate-forme (i.e. modèle métier). Les plateformes doivent aussi être décrites par des modèles. On choisit une plateforme particulière pour le système et on transforme la spécification métier du système (PIM) en une spécification particulière à la plate-forme choisie (PSM). MDA propose donc de monter en abstraction en manipulant des modèles et en construisant des applications par transformation de modèles.

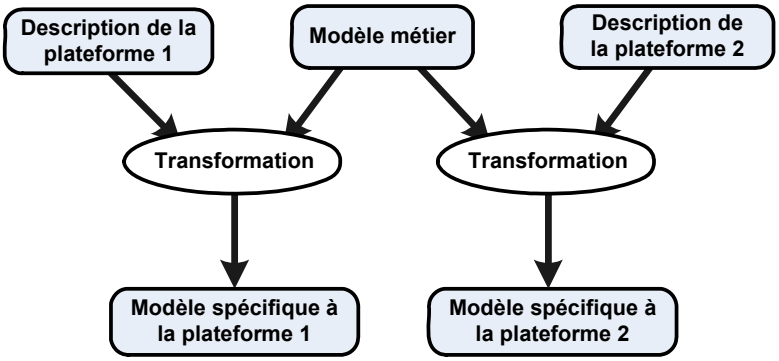


Figure 2.2. L'approche MDA

MDA résulte de la maturation d'un certain nombre de normes et de technologies qui ont permis de reconsidérer l'approche transformationnelle. Elle s'appuie notamment sur les techniques de modélisation et de transformation de modèles. Toutefois, les techniques de transformation de modèles en sont à leurs débuts et ne sont pas encore aussi matures que les techniques de modélisation. Parmi les normes clés qui forment la base de MDA, on trouve les langages de modélisation MOF et UML. MOF (*Meta Object Facility*) (MOF, 2003) fournit un ensemble de services de modélisation pour permettre l'interopérabilité entre systèmes et cela en

définissant la structure de tous les langages de modélisation. UML (*Unified Modeling Language*) (UML, 2006) (UML, 2005) est le langage de modélisation pour visualiser, spécifier et documenter les systèmes informatiques. Les modèles PIM et PSM peuvent être exprimés avec UML. UML utilise OCL (*Object Constraint Language*) (OCL, 2005) pour ajouter des contraintes formelles aux modèles. MDA s'appuie aussi sur la norme XMI (*XML Metada Interchange*) (XMI, 2003) pour sérialiser et échanger des modèles. Enfin, la plus récente des spécifications adoptées par l'OMG est la norme QVT (*Query View Transformation*) (QVT, 2005) qui vise à définir un langage pour manipuler les modèles. L'objectif de QVT est de fournir un langage qui permet d'exprimer des requêtes sur les modèles, d'extraire des vues à partir des modèles, et de spécifier de façon déclarative les transformations de modèles. La norme QVT se base sur MOF et OCL. Elle utilise OCL pour exprimer les contraintes sur les modèles et MOF pour décrire les transformations sous forme de modèles.

2.2.2 L'architecture de méta-modélisation

L'OMG a défini une architecture de méta-modélisation à quatre niveaux (Figure 2.3). Les éléments de chaque couche sont décrits par les éléments de la couche en dessus. La couche de plus bas niveau (M0) représente les données du monde réel. Ce sont les données que l'on désire modéliser. Par exemple, le livre « `Software Engineering` » et son auteur « `Ian Sommerville` » font partie des données du monde réel (Figure 2.3). Le niveau M1 correspond aux modèles d'informations. Ces modèles décrivent les informations du niveau M0. Un simple modèle UML fait partie de ce niveau. Par exemple, les données citées en exemple dans le niveau M0, sont décrites par un modèle UML qui contient deux classes (`Livre` et `Auteur`) ayant chacune un attribut `nom`.

Le niveau M2 fournit des modèles ou langages qui définissent les modèles d'informations (niveau M1). Les modèles de ce niveau sont appelés méta-modèles. Le méta-modèle d'UML, par exemple, appartient à ce niveau. Les modèles PIM et PSM se positionnent au niveau M1 de cette architecture. Les classes `Livre` et `Auteur` de notre exemple, sont toutes les deux représentées au niveau M2 par la

méta-classe `Class` et leur attribut « `nom` » est représenté par la méta-classe `Attribute`. Le plus haut niveau (M3) contient un seul et unique modèle ou langage qui est le MOF. Ce dernier définit aussi bien la structure de tous les méta-modèles du niveau M2 que sa propre structure. Par exemple, les classes `Class` et `Attribute` du niveau M2, sont toutes les deux représentées au niveau M3 par la classe `Class`. En fait, toutes les classes du niveau méta-modèle sont représentées par la classe `Class` au niveau méta-méta-modèle. La classe `Class` est représentée par elle-même, ce qui signifie qu'elle décrit sa propre structure.

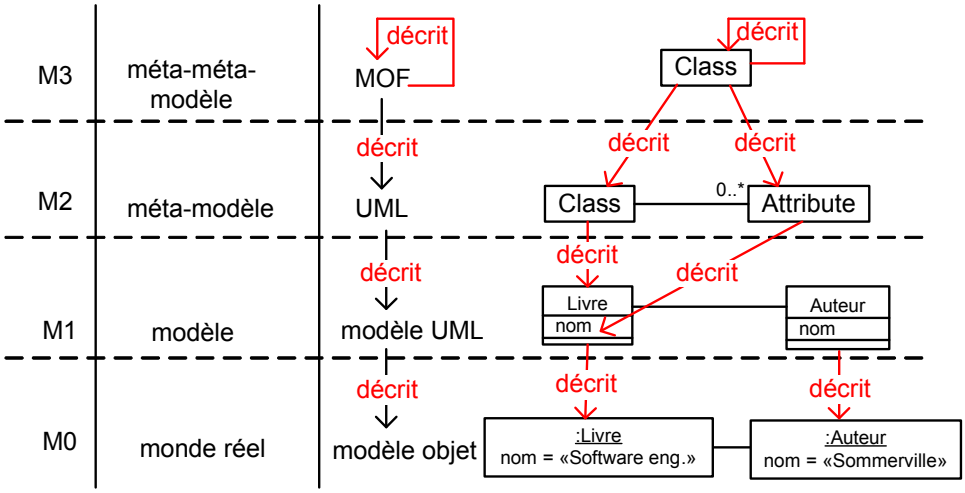


Figure 2.3. L'architecture de méta-modélisation

2.2.3 Les transformations dans MDA

MDA propose différentes approches de transformations de modèles. En effet, les transformations peuvent être spécifiées par une correspondance (*mapping*) au niveau des méta-modèles, des modèles ou par marquage. La transformation peut être spécifiée de façon déclarative par une correspondance entre les méta-modèles décrivant les modèles PIM et PSM, comme elle peut être spécifiée par une correspondance entre les modèles PIM et PSM (MDA, 2003). Une transformation définie au niveau méta-modèle est plus générique que celle définie au niveau modèle. Mais, la spécification au niveau modèle est plus précise car une entité du méta-modèle peut avoir plusieurs instances. Toutefois, les transformations de ce niveau ne sont pas réutilisables.

Une autre approche de transformation consiste à marquer d'abord le modèle PIM puis le transformer. Une marque représente un concept d'une plateforme et elle est utilisée pour marquer un élément du PIM pour indiquer comment l'élément doit être transformé. Ces marques sont définies dans un modèle qui décrit la plate-forme ciblée. Lorsqu'une plate-forme est choisie, les marques qui lui sont associées sont alors utilisées pour marquer les éléments du modèle PIM pour guider la transformation vers le PSM. On obtient un modèle PIM marqué qui est ensuite transformé pour obtenir le modèle PSM. Pour illustrer cette approche de transformation, nous présentons un exemple (Figure 2.4) extrait de (MDA, 2003). Cet exemple montre comment un modèle PIM est marqué pour être transformé en un modèle PSM spécifique à la plate-forme CORBA.

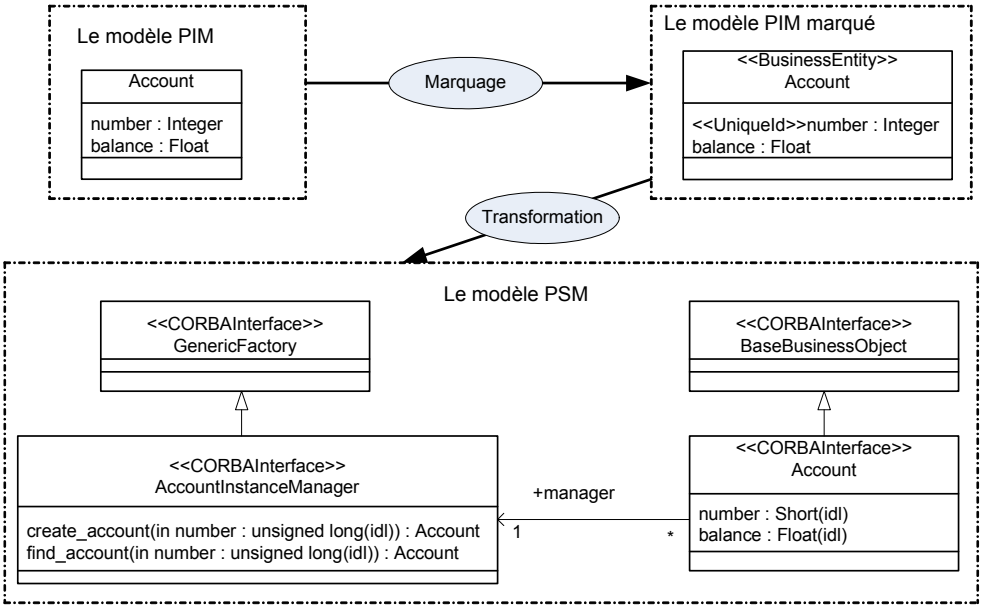


Figure 2.4. Marquage et transformation d'un modèle PIM

Pour marquer la classe `Account`, nous devons décider à quel concept CORBA (e.g. `interface`, `businessEntity`, etc.) elle correspond. Dans ce cas, elle correspond à une entité de type `businessEntity` au sens CORBA, et elle a été marquée ainsi. La transformation d'une entité de type `businessEntity` consiste à créer deux interfaces CORBA: une interface qui hérite de l'interface `BaseBusinessObject`, et une interface—appelée `AccountInstanceManager` dans notre exemple—qui gère l'instance de la classe originale et qui hérite de

l'interface `GenericFactory`. Cette transformation a été donc appliquée à la classe `Account` pour générer le modèle PSM spécifique à CORBA.

Dans tous les cas, la transition du modèle PIM vers le modèle PSM inclut aussi bien la phase de conception architecturale que la phase de conception détaillée. Chacune de ces phases consiste à raffiner le modèle PIM en le modifiant pour refléter les décisions de conception qui ont été prises durant le processus de conception, i.e. choix d'un style architectural, application de patrons de conception, etc. Nous abordons dans le reste de notre revue de littérature la conception architecturale (section 2.3) et la conception détaillée (section 2.3).

2.3 Conception architecturale et styles architecturaux

Une étape des plus importantes et critiques dans le processus de conception d'un système est la conception architecturale (Garlan et al., 2002). Simplement parlant, l'architecture fournit la description de haut niveau de la structure d'un système. « L'architecture logicielle inclut la description des éléments à partir desquels les systèmes sont construits, les interactions entre ces éléments, les patrons qui guident leur composition et les contraintes sur ses patrons » (Garlan et Shaw, 1994). La conception architecturale se base sur des patrons récurrents d'organisation appelés styles architecturaux. En effet, plusieurs systèmes sont construits et organisés de façon similaire en se basant sur les styles architecturaux. Cela explique l'importance que revêtent la compréhension et la spécification des styles architecturaux dans la conception architecturale. Par ailleurs, il est aussi très important de disposer de langages qui permettent de spécifier et formaliser les architectures.

C'est pour cette raison que nous aborderons dans cette section aussi bien la notion de style architectural que les langages qui ont été proposés pour modéliser les architectures. Nous commencerons par définir ce qu'est un style architectural et en donner un exemple. Ensuite, nous définirons les concepts sous-jacents à la description architecturale avant de donner un exemple de langage de description

d'architectures, et de faire un survol des langages les plus communs. Nous aborderons aussi la problématique d'utiliser UML pour la conception architecturale.

2.3.1 Style architectural

Un style architectural peut être considéré comme une classe générique d'architecture ou un patron d'organisation, e.g. l'architecture client-serveur. Selon Garlan et Shaw(1994), un style architectural est précisément défini par son vocabulaire (types de composants et de connecteurs), ses contraintes de configuration (i.e. contraintes topologiques appelées aussi patrons structurels), ses invariants, ses exemples d'utilisation, ses avantages et inconvénients, et ses spécialisations les plus communes. Un style est aussi caractérisé par le type d'analyses qu'on peut effectuer sur les systèmes l'adoptant. La spécification d'un style architectural inclut une partie statique et une partie dynamique. La partie statique englobe un ensemble de composants et connecteurs et des contraintes sur ces composants et ces connecteurs. La partie dynamique décrit l'évolution possible d'une architecture en réaction à des changements prévisibles ou imprévisibles de l'environnement.

Un des styles architecturaux les plus communs est le style « Filtres et Canaux » (*pipes and filters*). Dans ce style, un composant reçoit un ensemble de données en entrée et produit un ensemble de données en sortie. Le composant lit continuellement les entrées sur lesquelles il exécute un traitement ou une sorte de *filtrage* pour produire les sorties. Pour cela, les composants sont appelés filtres (Garlan et Shaw, 1994). Les spécifications des filtres peuvent contraindre les entrées et les sorties. Un connecteur, quant à lui, représente une sorte de conduit—appelé «Canal»—qui permet d'acheminer les sorties d'un filtre vers les entrées du suivant. Le style « Filtres et Canaux » exige que les filtres soient des entités indépendantes et qu'ils ne connaissent pas l'identité des autres filtres. De plus, la validité d'un système conforme à ce style ne doit pas dépendre de l'ordre dans lequel les filtres exécutent leur traitement. Parmi les exemples les plus connus de systèmes conformes à ce style, on peut citer les programmes écrits en Unix. La Figure 2.5 représente un exemple simple de ce style où on a deux filtres, le premier reçoit un flux de caractères en entrée et le transmet en sortie au second.

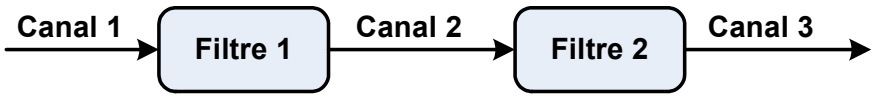


Figure 2.5. Exemple de filtres et canaux

Chaque style architectural se distingue par un ensemble de qualités architecturales qu'il favorise. Par exemple, le style « Filtres et Canaux » facilite la compréhension du comportement d'un système en le décomposant en un ensemble de comportements spécifiés par les filtres. Ainsi, un système conforme à ce style est facile à maintenir. Ce style favorise aussi la réutilisation puisqu'un même filtre peut être inséré dans plusieurs chaînes de traitements. De plus, de par son organisation, ce style supporte l'exécution concurrente et permet l'analyse des sorties et des analyses d'inter-blocage. Cependant, l'application de ce style peut dégénérer et donner une décomposition purement séquentielle d'un système.

2.3.2 Les langages de description d'architectures (ADLs)

Plusieurs langages de description d'architectures (ADL : *Architecture Description Language*) ont été proposés dans la littérature. Ces langages proposent des notations formelles pour décrire les concepts de la description architecturale, et permettent des analyses rigoureuses des architectures. Les plus connus de ces langages sont Wright (Allen, 1997), ACME (Garlan et al., 2000), Rapide (Luckham et al., 1995), UniCon (Shaw et al., 1995), C2 (Taylor et al., 1996), Darwin (Magee et al., 1995) et AESOP (Garlan et al., 1994). Un ADL fournit une syntaxe et une sémantique formelle pour modéliser l'architecture conceptuelle d'un système. Il permet de spécifier de façon abstraite les éléments d'une architecture sans définir leur implémentation. La majorité des ADLs partagent un ensemble de concepts essentiels à la modélisation architecturale tels que les composants, les connecteurs et les configurations (Medvidovic et Taylor, 2000). Ils visent généralement à spécifier explicitement les composants et la configuration d'un système. Cependant, chacun des ADLs existants s'est intéressé à une famille spécifique d'architectures ou à un domaine particulier, et de ce fait, représente une approche particulière de spécification. Ainsi, les ADLs existants se concentrent sur des aspects spécifiques de

l'architecture, et fournissent des techniques puissantes mais spécifiques d'analyse. Avant de présenter un survol des ADLs les plus communs, nous allons introduire les concepts sous-jacents à la description architecturale et qui doivent être modélisés par un ADL.

2.3.2.1 Concepts de base

À la base de la notion d'architecture, on retrouve les concepts de composant, connecteur et configuration. Un ADL doit permettre au moins de modéliser ces concepts. En nous basant sur les études faites sur les ADLs (Clements, 1996) (Medvidovic et Taylor, 2000) (Shaw et Garlan, 1994) (Vestal, 1993), nous présentons brièvement les propriétés principales qu'un ADL doit avoir. Certaines propriétés sont des exigences fondamentales que tout ADL doit supporter :

Spécification des composants : «Un composant est une unité de calcul localisée et indépendante» (Allen, 1997). C'est une unité de calcul ou de données qui peut être atomique ou composite et qui possède une interface décrivant les points d'interaction du composant avec l'environnement. Un ADL doit permettre la spécification d'un type de composant et de son interface.

Spécification des connecteurs : Le connecteur modélise les interactions entre les composants et aussi les règles qui régissent ces interactions. Il peut être implanté sous forme d'un appel de procédure, d'une variable partagée, etc. Un connecteur a une interface qui regroupe l'ensemble des points d'interaction entre le connecteur et les composants ou connecteurs qu'il relie. Il possède aussi un type qui représente une abstraction de l'interaction qu'il représente. Un ADL doit permettre de spécifier un connecteur comme une entité de première classe pour pouvoir lui associer une interface et le réutiliser.

Spécification des configurations : La configuration architecturale, appelée aussi topologie, décrit la structure complète d'un système. Elle se présente souvent sous forme de graphe connexe regroupant des composants et des connecteurs. Un ADL doit fournir un support adéquat pour modéliser une configuration.

Spécification des styles architecturaux : Une architecture est conforme à un style si ses composants et connecteurs sont ceux définis par le style considéré, et si toutes les configurations qu'elle comporte sont conformes aux contraintes définies par le style. Il est important de pouvoir spécifier un style architectural pour pouvoir faire des analyses et des vérifications sur l'architecture d'un système.

D'autres propriétés sont des exigences souhaitables qu'un ADL doit supporter pour permettre des implémentations correctes des architectures. Ces propriétés comprennent la composition hiérarchique ainsi que les spécifications de la sémantique, des contraintes et des propriétés non fonctionnelles. La sémantique des composants et des connecteurs est décrite par des modèles de comportement. Le comportement d'un composant inclut l'aspect dynamique du composant alors que celle du connecteur comprend la spécification des protocoles d'interaction qu'il décrit (Medvidovic et Taylor, 2000). Des contraintes peuvent être associées à un composant, à un connecteur ou à une configuration entière. Les contraintes sur une configuration architecturale sont des contraintes globales qui découlent souvent des contraintes sur les composants et les connecteurs participant à la configuration. Les propriétés non fonctionnelles (e.g. sécurité, performance) représentent des exigences qui ne découlent pas directement de la sémantique des concepts architecturaux (composant, connecteur, configuration). Enfin, la composition hiérarchique est une propriété qui permet de décrire une architecture à différents niveaux d'abstraction. Un composant ou un connecteur peut lui-même être un ensemble de composants et connecteurs.

La dernière catégorie de propriétés inclut des exigences désirables mais non fondamentales telle que la réutilisation des composants, des connecteurs et des configurations. La disponibilité d'un outil peut aussi faciliter la conception architecturale d'un système ainsi que l'évaluation de ses propriétés. L'outil peut même aller jusqu'à la génération d'un système exécutable à partir de la description architecturale.

2.3.2.2 Exemple : le langage Wright

Le langage Wright (Allen, 1997) (Allen et Garlan, 1997) est un des ADLs les plus communs. Il fournit une spécification explicite pour un composant, un connecteur, une configuration et même un style. Wright utilise une notation basée sur CSP (*Communicating Sequential Processes*) (Hoare, 1985) pour décrire le comportement et les interactions, ce qui permet de définir une sémantique formelle, et de rendre possible un certain nombre d'analyses et de vérifications des architectures. La configuration d'un système est décrite avec le langage Wright en trois parties. Un exemple simplifié d'une architecture Client/serveur est montré à la Figure 2.6.

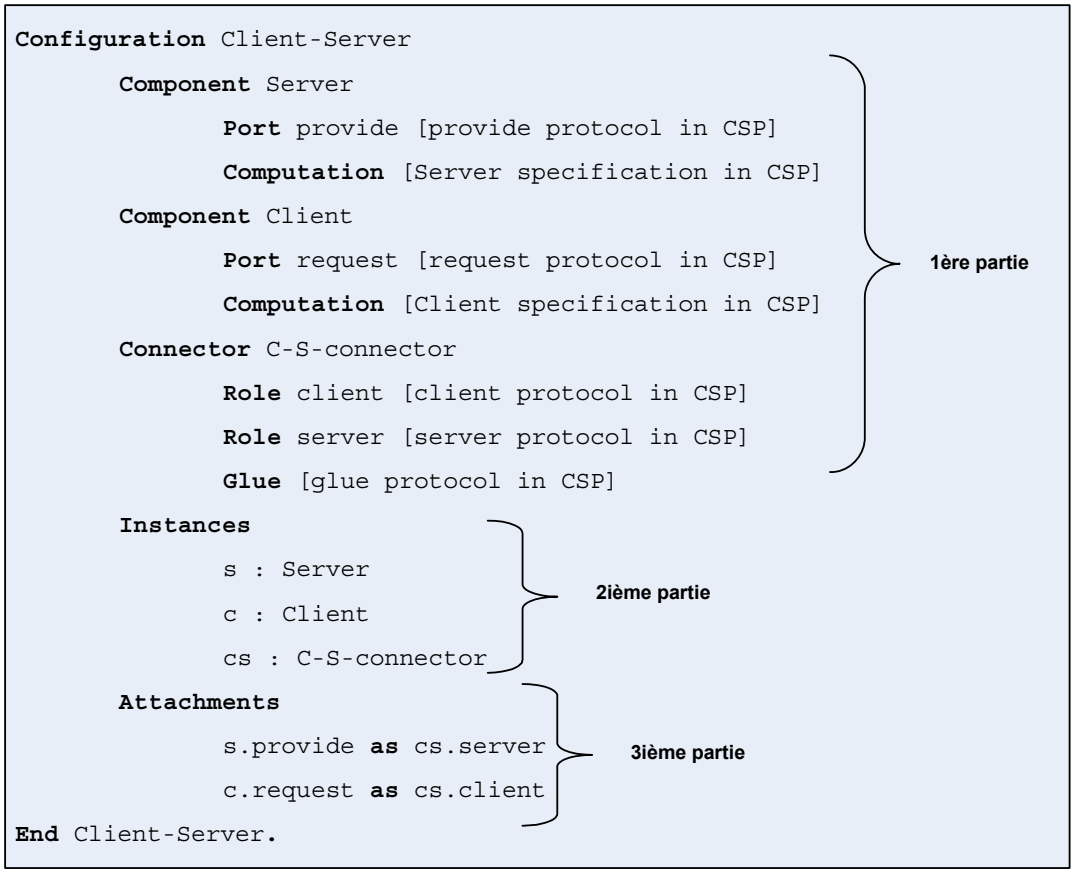


Figure 2.6. Une architecture Client/serveur exprimée avec le langage Wright

La première partie de la configuration définit les types des composants et des connecteurs. Un composant est décrit par un ensemble de ports représentant son interface, et une spécification appelée `computation` décrivant son comportement. Chaque port représente un point d'interaction entre le composant et son

environnement, et possède une description formelle exprimée en CSP qui spécifie les propriétés et les attentes du composant vu à travers ce port. La partie `computation` fournit une description complète du comportement et des propriétés du composant en montrant comment les ports sont regroupés et utilisés. Un connecteur, quant à lui, est défini par un ensemble de rôles et une spécification d'assemblage (`Glue`). Chaque rôle correspond à une partie intervenante dans l'interaction, et sa spécification–exprimée en CSP–décrit le comportement attendu de cet intervenant. La partie `Glue` décrit comment les activités des rôles des composants sont coordonnées. Par exemple, le connecteur de la configuration `Client-Server` peut être exprimé comme à la Figure 2.7.

connector C-S-connector =
  role Client = (request!x → result?y → Client) ⊓ §
  role Server = (invoke?x → return!y → Server) □ §
  glue = (Client.request?x → Service.invoke!x → Service.return?y →
    Client.result!y → **glue**)
  □ §

Figure 2.7. Exemple de connecteur extrait de (Allen, 1997)

Le rôle du serveur est décrit par un processus qui accepte une invocation et envoie un retour ou qui se termine avec succès (§ représente un processus qui se termine avec succès). Le rôle du client est aussi décrit par un processus qui appelle un service et reçoit le résultat ou se termine avec succès. La différence dans la spécification des deux rôles est le choix de l'opérateur—interne Π ou externe —qui permet de distinguer entre une situation où le comportement du processus est dicté par l'environnement (ici le serveur est obligé de fournir un service lorsqu'il y a une requête), et la situation ou un rôle peut choisir d'utiliser des services ou non (ici le client qui peut émettre une requête ou pas). Le processus `glue` coordonne le comportement des deux rôles en indiquant comment leurs événements interagissent.

La seconde partie d'une configuration –appelée `Instances`– spécifie un ensemble d'instances de composants et de connecteurs (Figure 2.6). Plusieurs instances d'un même type peuvent exister dans la même configuration. Chaque instance est nommée de façon explicite et unique. Dans la troisième partie de la

description –appelée `Attachments`– la topologie de la configuration est décrite: des instances de composants et de connecteurs sont combinées en précisant quels ports sont attachés à quels rôles. Les ports et les rôles associés doivent être compatibles, i.e. un port doit être conforme aux règles imposées par un rôle pour pouvoir participer à l'interaction spécifiée par le connecteur définissant ce rôle.

Le langage Wright permet la composition hiérarchique. En effet, la partie `computation` d'un composant et la partie `glue` d'un connecteur peuvent correspondre à des configurations. De plus, Wright permet la spécification d'un vocabulaire en utilisant des interfaces pour spécifier les contraintes sur les ports d'un composant ou les rôles d'un connecteur. Wright utilise des prédicats logiques de premier ordre pour exprimer les contraintes associées à un style.

2.3.3 Synthèse sur les ADLs

Les ADLs couvrent différents domaines et se concentrent sur différents aspects de la description architecturale. Le langage Darwin, par exemple, a pour objectif la construction de systèmes distribués tandis que le langage Rapide se concentre sur la spécification et l'analyse de l'aspect dynamique dans les architectures. Aesop supporte l'utilisation des styles architecturaux et vise principalement à fournir une plateforme de développement basée sur les styles. ACME a pour but de permettre l'échange de descriptions architecturales exprimées par différents ADLs. Il représente plutôt un langage unificateur pour l'ensemble des ADLs. Le langage C2 supporte la description des systèmes distribués dynamiques, en particulier les systèmes d'interface utilisateur. UniCon permet la composition et la vérification des systèmes à partir d'éléments prédéfinis. Enfin, le langage Wright se concentre plus sur la modélisation de tous les concepts architecturaux (composant, connecteur, configuration et style) et sur les analyses statiques dont les analyses des inter-blocages dans les systèmes concurrents.

En étudiant ces différents langages, nous avons pu identifier deux grandes familles d'ADLs. La première famille d'ADLs se concentre sur la spécification explicite des propriétés architecturales structurelles. En particulier, ces ADLs

décrivent les connecteurs par des entités de première classe. Cette famille comprend les langages Wright, ACME, Aesop, C2 et UniCon. La seconde famille se concentre plutôt sur la spécification de l'aspect dynamique des architectures et ne spécifie pas de façon explicite les connecteurs. Cette famille inclut les langages Darwin et Rapide. Nous donnons une description détaillée de chacun de ces ADLs à (El-boussaidi et Mili, 2006).

Même s'ils emploient parfois des terminologies différentes, tous les ADLs considérés permettent de spécifier un composant, son interface et son type. Par contre, ils ne spécifient pas tous de façon explicite un connecteur (e.g. Darwin et Rapide). Les ADLs qui ne permettent pas de définir un connecteur de façon explicite ne permettent pas non plus de spécifier son interface et ne favorisent pas sa réutilisation. Ils ne différencient pas entre type et instance d'un connecteur et n'offrent pas de support à son évolution (Medvidovic et al., 2002). La majorité des ADLs ne supportent pas non plus l'évolution d'un composant ou d'une architecture mais les décrivent de façon statique.

Certains ADLs permettent de spécifier la sémantique des composants (e.g. Rapide, UniCon, Wright) et des connecteurs (e.g. UniCon, Wright). La sémantique est exprimée de façon très variée. Dans UniCon, elle se limite à une liste de propriétés alors que dans Wright elle est exprimée formellement avec le langage CSP. De façon générale, les langages "génériques" (e.g. Aesop, ACME) se concentrent sur l'aspect structurel et négligent l'aspect sémantique parce qu'ils visent plutôt à fournir un modèle architectural générique. Par contre, les langages qui traitent l'aspect dynamique des architectures (e.g. Darwin, Rapide) ne modélisent pas explicitement tous les concepts architecturaux, notamment les connecteurs. À l'exception d'UniCon, aucun ADL ne supporte vraiment la spécification des propriétés non fonctionnelles des composants et des connecteurs.

2.3.4 La description architecturale basée sur UML

Parce que les modèles que nous aurons à manipuler aussi bien que les modèles manipulés dans le contexte MDA sont exprimés avec le langage UML, et parce

qu'UML a l'avantage d'être supporté par plusieurs outils, nous nous penchons sur la problématique d'utiliser UML pour la conception architecturale. Partant de l'idée qu'une description architecturale doit être comprise et manipulée par plusieurs intervenants qui ont peu de connaissances dans le domaine des spécifications formelles, plusieurs travaux ont tenté d'utiliser UML comme un langage de description des architectures (Garlan et al., 2002) (Medvidovic et al., 2002) (Baresi et al., 2003) (Zarras et al., 2001) (Perez-Martinez, 2003).

La majorité de ces travaux ont souligné l'inadaptabilité d'UML1.x à la spécification explicite et complète des architectures. En effet, certains concepts architecturaux n'ont pas de constructions équivalentes en UML, e.g. les connecteurs. Même quand il y'en a une, la sémantique n'est pas la même au sens des ADLs. Par exemple, un composant dans UML1.x décrit un artefact d'implantation et non un composant au sens des ADLs. En outre, il n'y a pas de démarche unique pour décrire les concepts architecturaux en utilisant UML1.x (Garlan et al., 2002). Certaines stratégies proposées utilisent des constructions différentes (e.g. Classe, Association) pour spécifier le même concept architectural (e.g. Connecteur). Parfois, certaines entités architecturales ayant différentes responsabilités (e.g. composant et connecteur) sont représentées avec le même concept (e.g. Classe) (Medvidovic et al., 2002). D'autres stratégies utilisent les mécanismes standards d'extension offerts par UML— tags, contraintes, stéréotypes et profils— pour spécifier les concepts architecturaux. L'utilisation de ces mécanismes a l'avantage de permettre l'utilisation des différents outils existants supportant UML. Cela n'est pas le cas quand on étend le méta-modèle UML en introduisant de nouveaux concepts (e.g. (Perez-Martinez, 2003)). Les approches qui utilisent les mécanismes standards d'UML représentent souvent les règles inhérentes à un style architectural par des contraintes exprimées en OCL. Le problème est que ces contraintes ne sont pas directement reflétées dans le modèle UML comme cela se fait avec un ADL. En outre, les modifications ultérieures faites sur le modèle UML peuvent être non conformes aux règles du style. L'aspect dynamique de l'architecture est aussi difficile à modéliser avec les diagrammes UML et il n'est pas garanti que cet aspect soit correctement et fidèlement modélisé

(Medvidovic et al., 2002). Enfin, le modèle architectural obtenu n'est pas toujours facilement lisible.

Par rapport aux versions précédentes d'UML, UML 2.0 a introduit de nouvelles constructions qui visent à le rendre plus adapté au développement à base de composants et à améliorer le support offert pour les descriptions architecturales. Cependant, il y a encore différentes façons de représenter certains concepts architecturaux avec UML2.0 (Ivers et al., 2004). De plus, le concept de Connecteur n'est pas modélisé comme une entité de première classe: la façon dont un connecteur est spécifié ne permet pas de lui attribuer une interface ni une sémantique comme le font certains ADLs, e.g. Wright. Finalement, UML2.0 ne décrit pas les propriétés fonctionnelles et n'offre pas un support explicite pour la modélisation des styles architecturaux. Toutefois, on peut représenter un style en utilisant les mécanismes standards d'extensions offerts par UML, en particulier les profils.

2.4 Conception détaillée et patrons de conception

La conception détaillée a pour objectif de spécifier les composants à un niveau plus bas et moins abstrait que la conception architecturale. Durant cette phase, le concepteur définit l'ensemble des éléments qui vont guider l'implantation des composants, i.e. définition des interfaces, des structures de données et des algorithmes. Cette phase comprend aussi le choix et l'application de patrons de conception (Gamma et al., 1995). Les patrons de conception sont des solutions éprouvées à des problèmes spécifiques et récurrents de conception. Un patron décrit un problème devant être résolu, une solution, et le contexte dans lequel cette solution est considérée (Johnson, 1997). En pratique, un patron décrit une technique pour régler un problème ainsi que les avantages et les coûts de l'utilisation de cette technique. De plus, il définit un vocabulaire commun aux concepteurs.

Plusieurs travaux se sont intéressés à l'étude et à la documentation des patrons (Coad, 1992), (Gamma et al., 1995), (Buschmann et al., 1996), (Fowler, 1997), (Johnson, 1997). Ces travaux ont proposé différents formalismes pour la représentation des patrons. En ce qui concerne les patrons de conception, le catalogue

de Gamma et al. a énormément contribué à leur émergence et à leur acceptation. Notre objectif de recherche étant de transformer des modèles en leur appliquant des patrons de conception, nous nous limiterons, donc, à l'étude des patrons de ce catalogue. La section suivante donne une brève description de ce catalogue et aussi un exemple de patron extrait de ce catalogue. Nous discuterons ensuite des problèmes et des exigences reliés à la mise en œuvre de ces patrons (section 2.4.2) avant de présenter une synthèse des travaux qui se sont penchés sur cette problématique (section 2.4.3 et 2.4.4).

2.4.1 Les patrons de conception : introduction

Dans le catalogue de Gamma et al., les patrons de conception sont décrits d'une façon uniforme et structurée. Un patron est décrit par des propriétés dont le nom, l'objectif, un exemple de problème de conception qu'il résout, et la structure de la solution proposée sous forme de diagramme de classes. Les patrons se distinguent donc par les problèmes qu'ils résolvent et leur domaine d'application (i.e. classe versus objet). En effet, Gamma et al. les classent selon leur rôle (Gamma et al., 1995). Ils distinguent entre patrons créateurs, patrons structurels et patrons comportementaux. Les patrons créateurs concernent la création de classes ou d'objets (e.g. « Singleton » et « Fabrication abstraite »). Les patrons structurels s'intéressent à la composition d'objets ou classes pour réaliser de nouvelles fonctionnalités (e.g. « Composite », « Pont » et « Proxy »). Finalement, les patrons comportementaux concernent les interactions entre classes et l'affectation des responsabilités (e.g. « Médiateur » et « Stratégie »). Nous avons choisi de présenter l'exemple du patron « Visiteur » qui est un des patrons que nous utiliserons aussi pour illustrer notre approche présentée aux chapitres suivants.

Le patron Visiteur appartient à la classe des patrons comportementaux. Nous n'aborderons pas ici toutes les rubriques du catalogue, mais nous présentons une description sommaire et suffisante pour la compréhension de ce patron. Considérons l'exemple d'un compilateur qui représente un programme par un arbre syntaxique abstrait. Le compilateur a besoin d'exécuter des opérations d'analyse (e.g. vérification de type ou vérification de l'initialisation), comme il a besoin de générer

du code. Les nœuds de l'arbre peuvent représenter des instructions d'affectation, des variables ou des expressions arithmétiques. La plupart des opérations du compilateur auront donc à traiter les nœuds différemment selon leur type. On représente donc les nœuds par une hiérarchie de classes dont une partie est montrée à la Figure 2.8.

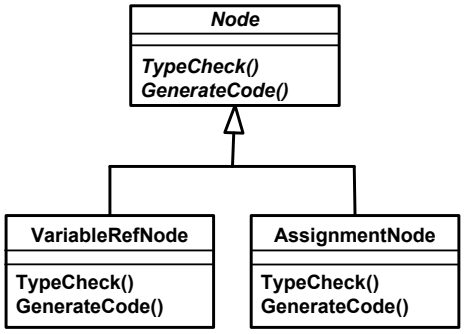


Figure 2.8. Extrait de la hiérarchie des nœuds

Le problème ici est que même si la hiérarchie de classes est bien stable (i.e. les différents types de nœuds sont connus et peuvent être définis à l'avance), l'implantation de toutes ces opérations à travers toutes les classes de la hiérarchie conduit à un système difficile à maintenir. En effet, l'ajout d'une nouvelle opération nécessite la modification des interfaces de toutes les classes de la hiérarchie. Par exemple, si on veut ajouter la fonction `print` à la hiérarchie de la Figure 2.8, il faudra définir une méthode abstraite au niveau de la racine (`Node`) et ajouter la méthode à chaque classe de la hiérarchie.

La solution proposée par le patron Visiteur consiste à implémenter chacune des opérations par une classe distincte qui regroupera les différentes implémentations de l'opération pour les différentes classes de la hiérarchie. La structure de cette solution telle qu'introduite par (Gamma et al., 1995) est représentée à la Figure 2.9. Dans ce modèle, une hiérarchie de classes représente les opérations (les objets visiteurs) et une autre les éléments (objets visités) sur lesquels ces opérations s'appliquent. Chaque classe de type `Visitor` (e.g. `ConcreteVisitor1`) regroupe l'ensemble des implantations d'une opération donnée. Les classes d'objets sur lesquels les opérations doivent s'appliquer sont représentées par les classes de type `Element` (e.g. `ConcreteElementA`). Avec cette nouvelle conception, ajouter une

nouvelle opération consiste à ajouter une nouvelle classe de type `Visitor` (i.e. `ConcreteVisitorC`) qui regroupe toutes les implantations possibles de cette nouvelle opération.

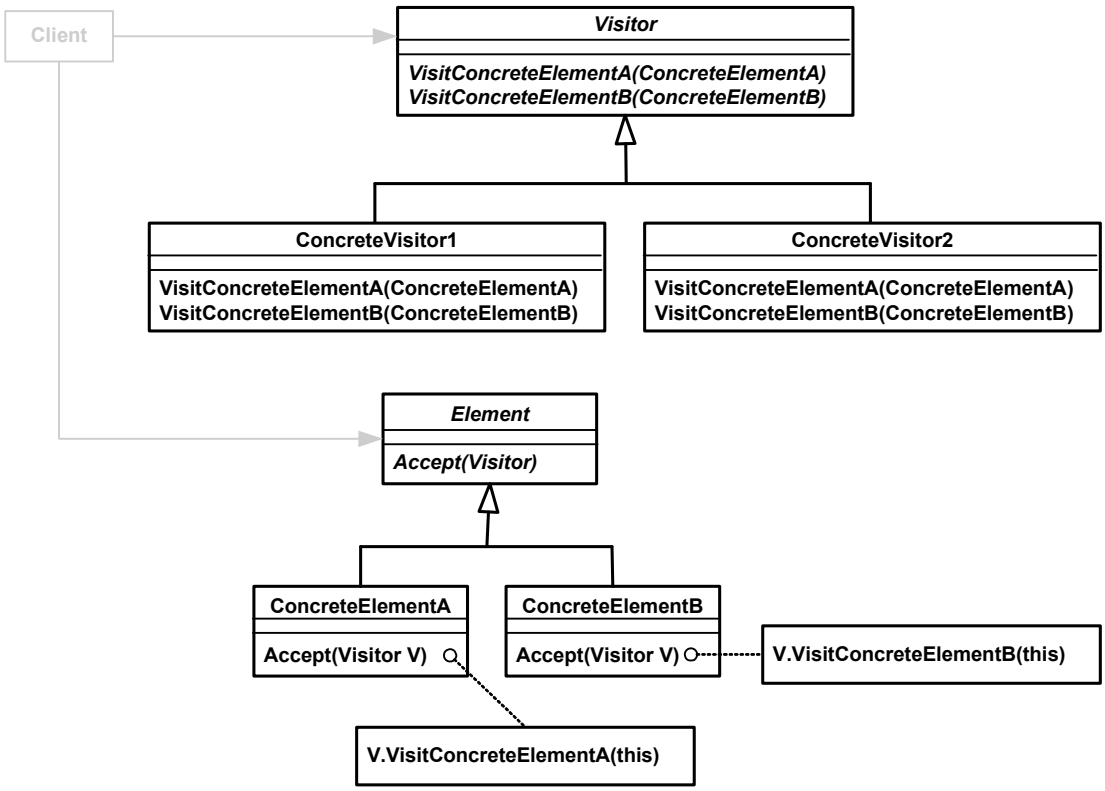


Figure 2.9. Structure de la solution proposée par le patron Visiteur

L'application de ce patron à l'instance de la Figure 2.8 donne donc deux hiérarchies de classes montrées à la Figure 2.10. L'ajout de la méthode `print` ne nécessite plus la modification de toute la hiérarchie des nœuds mais uniquement l'ajout d'une classe `PrintVisitor` (à la hiérarchie des visiteurs) qui implémente les opérations d'impression pour chaque type de nœud. La nouvelle structure est donc plus flexible et plus facile à maintenir.

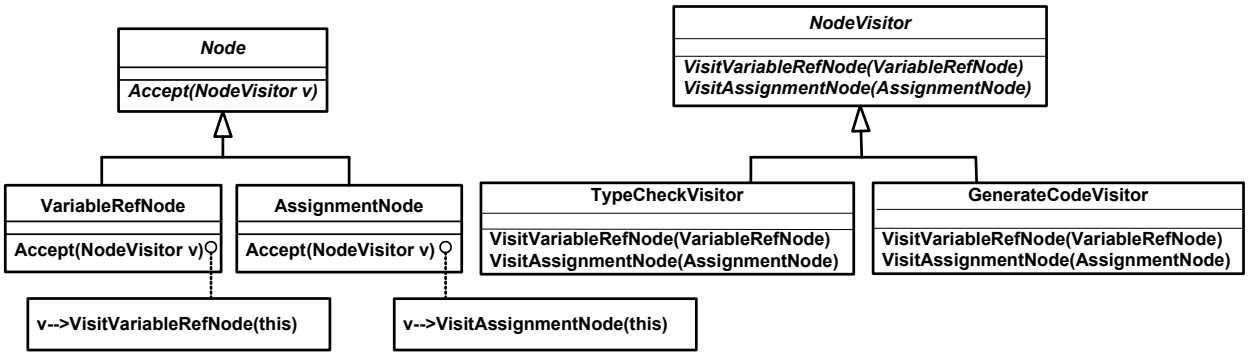


Figure 2.10. Application du patron Visiteur au problème de la Figure 2.8.

2.4.2 Mise en œuvre de patrons de conception : problèmes et exigences

L'illustration de la motivation d'un patron par un exemple facilite la compréhension de la solution proposée par le patron. Cependant, l'utilisation ou l'application d'un patron peut être difficile ou inexacte. En effet, pour un concepteur ou un programmeur expérimenté ayant déjà utilisé les patrons, l'application d'un patron dans différents contextes peut se faire sans trop de problèmes et assez rapidement alors que pour des débutants, l'opération d'adaptation d'un patron à un contexte donné peut s'avérer complexe et coûteuse. D'abord, le concepteur doit comprendre le problème de conception pour pouvoir identifier la solution adéquate et donc le patron adéquat. Une fois que le patron à utiliser est identifié, il faut l'appliquer au modèle concerné. Pour cela, le concepteur doit pouvoir définir le rôle de chaque élément du modèle pour établir une correspondance avec les éléments du patron. Une fois le patron appliqué, il faudra s'assurer que la sémantique du modèle est toujours respectée, comme il faudra s'assurer que de futures modifications du modèle ne violent pas les contraintes sémantiques et structurelles prescrites par le patron.

De façon générale la description des patrons se fait par une combinaison de diagrammes, de texte et d'exemples de code comme c'est le cas dans le catalogue des patrons de conception de (Gamma et al., 1995). Les diagrammes ont été généralement exprimés avec des notations objet non formelles comme OMT (Rumbaugh et al.,

1996) ou UML. Or, ces diagrammes ne permettent ni la spécification formelle des patrons ni la mécanisation de leur application. Enfin, l'implantation des patrons dans un langage de programmation nécessite une correspondance entre les éléments de conception du patron et les constructions du langage de programmation. Cela n'est pas toujours aussi évident, dans le cas du langage Java, par exemple, on ne peut pas implémenter l'héritage multiple.

De nombreux travaux se sont penchés sur la problématique de la représentation et la mise en œuvre des patrons de conception. Des approches ont proposé des outils ou modifié des outils existants pour permettre la représentation ou la mise en œuvre des patrons de conception. Cependant, les représentations adoptées pour les patrons sont diverses et dépendent des objectifs de chaque approche et des niveaux d'abstraction concernés. L'objectif d'une approche peut être de fournir un support d'aide à la conception, un support pour générer du code à partir de modèles conformes à des patrons, un outil de validation des modèles conformes à des patrons, ou un support de rétro-conception pour appliquer des patrons à des modèles existants (Borne et Revault, 1999).

Nous croyons que pour pouvoir automatiser l'application d'un patron de conception, une approche doit représenter explicitement aussi bien la structure d'un patron que la transformation inhérente à son application. La structure d'un patron est composée d'un ensemble d'éléments qui coopèrent pour résoudre un problème de conception. Il est très important de représenter explicitement ces éléments et les relations entre eux car cela permet de mieux comprendre le patron et fournit aussi une aide pour son utilisation. En effet, l'application d'un patron à un modèle consiste à reproduire la structure générique du patron concerné en utilisant un ensemble d'éléments du modèle considéré. L'application d'un patron à un modèle implique donc l'application d'une transformation au modèle en question. Cette transformation doit être spécifiée d'une façon explicite et déclarative pour permettre sa réutilisation dans différents contextes d'application du patron. Lorsque la transformation est spécifiée de façon explicite, le processus de sa mise en œuvre doit également être défini explicitement pour permettre aussi bien la réutilisation de la transformation que

celle du processus de son application. D'autres informations peuvent être spécifiées pour compléter la description des patrons comme la représentation de l'aspect collaboration et la spécification des contraintes reliées aux patrons.

2.4.3 Approches de mise en œuvre de patrons de conception

De façon générale, les approches peuvent être divisées en deux familles. Une première famille qui représente les patrons de façon explicite ce qui facilite leur manipulation par les concepteurs (modification de patron, ajout, etc.) mais qui néglige l'aspect mise en œuvre des patrons. Une seconde famille qui représente les patrons implicitement en se concentrant plutôt sur les transformations inhérentes à leur application. Dans la famille des travaux qui proposent une représentation explicite du patron, certains proposent un méta-modèle pour représenter les patrons, alors que d'autres proposent un ensemble de modèles ou un reposoir de patrons.

Pratiquement toutes les approches que nous avons étudiées –et qui représentent explicitement les patrons– adoptent une démarche *descendante* pour appliquer les patrons. Dans une démarche descendante, l'utilisateur choisit un patron et l'outil utilisé génère les éléments de conception (e.g. classes, méthodes, associations) composant le patron (Florijin et al., 1997). Dans la démarche *ascendante*, par contre, un ensemble d'éléments, reflétant la structure d'un patron particulier, est associé à une instance de ce patron. Certaines approches adoptent cette démarche pour la détection d'instances de patrons et cela généralement dans du code (e.g. (Albin-Amiot et Guéhéneuc, 2001)). Une démarche *mixte* consiste à compléter un ensemble d'éléments reflétant partiellement la structure d'un patron avec d'autres éléments et associations de façon à ce que le nouvel ensemble reflète la structure entière du patron. Aucune approche parmi celles que nous avons étudiées n'adopte cette dernière démarche.

Nous avons donc regroupé les travaux que nous avons étudiés en trois catégories de famille :

- La première catégorie d'approches représente un patron de façon implicite et ne dissocie pas cette représentation de l'application du patron.

- La seconde catégorie représente explicitement un patron, mais ne sépare pas le mécanisme de son application de cette représentation. Ce sont souvent des approches qui raisonnent dans un niveau unique de modélisation (abstraction).
- La dernière famille d'approches donne une représentation explicite d'un patron et le mécanisme d'application du patron est indépendant du patron lui-même. Ce sont des approches déclaratives où les patrons sont représentés généralement par des méta-modèles ou des profils UML.

Nous donnons une description sommaire des trois catégories de famille dans les sections suivantes. Une description détaillée de ces approches est donnée à (El-boussaidi et Mili, 2004).

2.4.3.1 Approches avec représentation implicite des patrons

On retrouve, par exemple, dans cette catégorie, les approches de Budinsky et al. (1996), Alencar et al. (1997) et Sunyé et al. (2000). Budinsky et al. présentent un outil qui permet de générer automatiquement le code d'un patron à partir de quelques informations fournies par l'utilisateur. Les patrons sont présentés comme un système hypertexte formé de pages reliées. Alencar et al. modélisent un patron de conception comme une transformation appliquée à un modèle et cela en utilisant un langage de processus. Les processus de transformation sont appliqués aux spécifications exprimées sous forme de schémas. Des outils peuvent traduire les descriptions faites dans le langage de processus aux langages de programmation orientés objet. Cette approche s'est plus concentrée sur l'aspect syntaxique des transformations alors que l'aspect sémantique n'a pas été approfondi. De leur côté, Sunyé et al. proposent un cadre d'application UMLAUT (*Unified Modeling Language All pUrposes Transformer*) permettant l'application des transformations sur des modèles UML. Ils utilisent une combinaison du paradigme orienté objet et du paradigme fonctionnel. Cette approche ne permet pas la représentation explicite d'un patron. Comme dans (Alencar et al., 1997), un patron est représenté d'une façon implicite dans les transformations appliquées à un modèle lors de son utilisation.

2.4.3.2 Approches avec représentation explicite des patrons

Dans cette catégorie d'approches, on peut citer (Fontoura et Lucena, 2001) (Eden et al., 1999) (Meijler et al., 1997) (Florijin et al., 1997) (Taibi et Taibi, 2006) (Sanada et Adams, 2002) (Maplesden et al., 2002). Certaines de ces approches (Eden et al., 1999) (Taibi et Taibi, 2006) se basent sur des langages formels pour la spécification des patrons. Eden et al. proposent un langage formel LePUS (*LanguagE for Patterns Uniform Specification*) pour exprimer les patrons. C'est un fragment de logique qui utilise un vocabulaire limité d'entités et de relations pour définir un patron de façon précise et complète sous forme de formule accompagnée d'une représentation graphique. Taibi et Taibi proposent un langage BPSL (*Balanced Pattern Specification Language*) qui utilise la logique de premier ordre pour formaliser l'aspect structurel des patrons et la logique temporelle d'actions pour spécifier l'aspect comportemental. De telles spécifications peuvent complémenter d'autres approches. Le problème avec ces deux approches est qu'elles proposent des langages qui ne sont pas faciles à maîtriser.

Les approches (Fontoura et Lucena, 2001) (Sanada et Adams, 2002) proposent d'étendre UML pour représenter et instancier les patrons. Ils utilisent les mécanismes d'extension offerts par UML (i.e. stéréotype, tag et contrainte) pour modifier les diagrammes de classes de façon à ce qu'ils puissent représenter explicitement les points de variation et les classes d'application. Toutefois, la représentation fournie par Sanada et Adams est utilisée plutôt comme documentation du patron et non pour son instanciation. Fontoura et Lucena utilisent des diagrammes d'activité pour décrire les étapes à exécuter durant l'instanciation d'un patron. Chaque action dans un diagramme d'activité est exprimée par un langage de programmation logique qui a une sémantique assez intuitive. Cependant, l'approche ne fournit pas une démarche pour appliquer un patron à un modèle déjà existant.

Meijler et al. (1997), Florijin et al. (1997), et Maplesden et al. (2002) représentent un patron par une structure regroupant un ensemble de participants ayant des rôles. Cette structure est spécifiée par un schéma dans (Meijler et al., 1997), un modèle de fragment dans (Florijin et al., 1997), et un diagramme de spécification

dans (Maplesden et al., 2002). Meijler et al. proposent en plus des schémas d'instanciation pour les patrons. Cependant, ces schémas ne sont pas génériques car ils incluent des informations spécifiques au cas concret pris comme exemple. Il en est de même dans (Maplesden et al., 2002) car il faut spécifier un diagramme d'instanciation pour chaque modèle ou partie de modèle auquel on veut appliquer un patron. Enfin, dans (Florijin et al., 1997), l'instanciation d'un patron se fait de façon descendante et il n'est donc pas possible de transformer des modèles déjà existants par application d'un patron.

2.4.3.3 Approches déclaratives

Dans la famille des approches déclaratives, plusieurs chercheurs proposent des méta-modèles pour représenter les patrons (Pagel et Winter, 1996) (Albin-Amiot et Guéhéneuc, 2001) (France et al., 2004) (Elaasar et al., 2006). En particulier, France et al. proposent un méta-modèle qui étend le méta-modèle UML tandis que Pagel et Winter, Albin-Amiot et Guéhéneuc proposent des méta-modèles dédiés aux patrons. Elaasar et al. proposent un cadre de modélisation des patrons qui se base sur la norme de méta-modélisation MOF, et qui adopte une architecture compatible avec l'architecture à 4 couches de méta-modélisation de l'OMG. Debnath et al. (2006) proposent aussi une architecture à 4 couches mais qui est basée sur les profils UML et où chaque patron est représenté par un profil. D'autres approches (Lauder et Kent, 1998) (Mens et Tourwé, 2001) ont utilisé des notations autres qu'UML pour représenter les patrons. Mens et Tourwé utilisent la technique de méta-programmation déclarative pour représenter un patron et ses contraintes par des prédicats. Lauder et Kent proposent d'utiliser en conjonction la notation des diagrammes de contraintes et la notation UML pour spécifier les patrons.

Le processus d'instanciation des patrons diffère d'une approche à l'autre. Pagel et Winter définissent le processus d'instanciation comme un ensemble de relations entre les différents éléments du patron décrit abstraitement et ceux du patron appliqué dans un contexte particulier. Albin-Amiot et Guéhéneuc visent plutôt à générer du code et proposent donc d'instancier un patron en plusieurs étapes en commençant par le méta-modèle pour aboutir à la génération du code. Dans (Lauder

et Kent, 1998), le modèle de rôles (Méta-modèle) peut être instancié par un modèle appelé le modèle de types qui est réalisé par un modèle de classes contenant les détails de l'implémentation concrète du patron. Mens et Tourwé proposent d'instancier un patron par une requête de création avec des arguments adéquats.

D'une façon analogue à (Gamma et al., 1995), certaines approches utilisent des diagrammes de séquence pour spécifier les collaborations entre les entités composant un patron (France et al., 2004) (Lauder et Kent, 1998). Toutefois, les diagrammes de contraintes proposés par Lauder et Kent (1998) ne font pas partie du standard UML, et donc les outils UML ne pourront pas être utilisés pour appliquer cette approche à moins qu'ils soient modifiés ou étendus. Aussi, l'outil proposé par France et al. (2004) ne supporte que la spécification structurelle des patrons.

En outre, l'approche de Mens et Tourwé est l'une des rares approches qui s'est penchée sur les transformations valides qui peuvent être faites sur du code existant (*refactoring*) et sur des instances de patrons. Ces transformations sont spécifiées sous forme d'un ensemble de règles exprimées dans un langage de programmation logique. Cependant, elles s'appliquent au code et non pas à des modèles. De façon générale, les approches que nous avons étudiées visent à spécifier la structure de la solution proposée par un patron et à générer du code à partir de cette structure. La majorité de ces approches ne permettent pas la rétro-conception dans la mesure où elles ne s'appliquent pas à des modèles existants.

2.4.4 Synthèse sur les approches de représentation et mise en œuvre des patrons

Généralement parlant, il y a un consensus sur les éléments qui composent et définissent un patron. Cependant, la majorité des approches se sont plus concentrées sur l'aspect structurel du patron, négligeant l'aspect collaboration entre éléments du patron. Seules les approches de Lauder et Kent et de France et al. ont abordé cet aspect. Les approches qui spécifient explicitement les patrons ne permettent pas toutes de spécifier les contraintes reliées aux patrons (e.g. (Fontoura et Lucena, 2001), (Albin-Amiot et Guéhéneuc, 2001)). Les travaux qui proposent des langages

formels basés sur des notations mathématiques (e.g. (Eden et al., 1999), (Taibi et Taibi, 2006)) permettent une spécification précise et générique des patrons et de leur application mais elles nécessitent des connaissances approfondies en mathématique.

Parmi les quelques approches qui se sont intéressées à la représentation des contraintes sur les éléments du patron, Florijin et al. (1997) ont aussi examiné les méthodes de validation et de vérification d'un modèle après application du patron. En effet, lors des modifications manuelles ou automatiques d'un modèle, la structure d'une instance de patron peut être rompue soit par une suppression d'un élément ou par l'ajout d'éléments, et il convient alors de vérifier que la structure du patron est toujours respectée. Cela peut être pris en charge par l'ajout de contraintes dont il faudra tenir compte lors de l'application du patron et aussi lors des modifications ultérieures. Mens et Tourwé suggèrent d'énumérer toutes les transformations possibles pour chaque patron (Mens et Tourwé, 2001).

Quelques aspects dans les descriptions des patrons ne sont pas faciles à spécifier tels que l'intention, la motivation, les conséquences et les indications d'utilisation. Budinsky et al. (1996) sont les seuls qui se sont intéressés à l'ensemble des rubriques employées dans (Gamma et al., 1995) pour décrire un patron. Toutefois, l'approche s'est limitée à fournir une description textuelle de ces rubriques. Concernant la convenance du patron à appliquer, seuls Mens et Tourwé (2001) et Sunyé et al. (2000) ont proposé d'appliquer des transformations avec des pré-conditions.

D'autres travaux se sont intéressés aux relations entre différents patrons (Noble, 1998) (Zimmer, 1994). Selon Noble, certains patrons peuvent être en conflit et ne peuvent être utilisés ensemble dans un même contexte (Noble, 1998). Seuls Eden et al. ont abordé cet aspect lors de l'utilisation des patrons (Eden et al., 1999). Toutefois, ils se sont limités à la spécification de la relation de raffinement entre deux patrons. Finalement, en dehors des approches utilisant des valeurs marquées (tags), la traçabilité des éléments composant une instance de patron n'est pas toujours possible. Ceci rend difficile la modification ultérieure de cette instance.

En ce qui concerne les transformations inhérentes à l'application des patrons, très peu d'approches les représentent explicitement (e.g. (Alencar et al., 1997), (Sunyé et al., 2000)). En effet, la majorité des approches se concentrent plutôt sur la représentation explicite des patrons et visent à fournir un support d'aide à la conception et non pas un support pour l'application des patrons (i.e. la transformation de modèles existants par application des patrons). En outre, dans le cas des approches qui représentent explicitement les transformations, la représentation d'un patron est implicite et indissociable de la transformation qui lui est associée (e.g. (Alencar et al., 1997), (Sunyé et al., 2000)).

2.5 Discussion

Pour la conception architecturale, la communauté de recherche s'est plus concentrée sur les langages de description d'architecture (ADLs), et les langages proposés étaient souvent spécialisés. En particulier, ces langages sont généralement difficiles pour un concepteur non expérimenté. Ceci limite leur utilisation et leur intégration dans le processus de développement. Pour sa part, la communauté industrielle utilise généralement la norme UML qui vise à modéliser l'architecture sous une perspective orientée objet. UML a l'avantage de fournir une notation flexible et facile à utiliser par les concepteurs. Il est aussi supporté par plusieurs outils dont certains vont jusqu'à la génération du code. Cependant, c'est un langage moins formel et moins rigoureux que les ADLs. De plus, UML ne permet pas de représenter des concepts architecturaux tels les styles architecturaux, à moins de l'étendre à cet effet.

Concernant les patrons de conception, les travaux de recherche se sont penchés sur la représentation de la structure du patron et ont négligé l'aspect mise en œuvre du patron. Cet aspect nécessite la représentation explicite des transformations inhérentes à l'application des patrons. Comme nous l'avons souligné à la section précédente, ces approches ne permettent pas d'appliquer un patron à un modèle existant.

Que ce soit pour représenter un style architectural ou un patron de conception, toutes les approches sans exception se sont limitées à la représentation de la solution que les styles ou patrons proposent et, dans une moindre mesure, à maintenir l'intégrité de la solution en cas d'évolution du système. La démarche adoptée par toutes les approches est descendante que ce soit pour appliquer un style architectural ou un patron de conception. Aucune approche ne s'est attaquée au problème de l'indentification de l'opportunité ou du contexte d'application de ces patrons— les styles architecturaux sont des patrons d'organisation. Le problème de mécanisation de la mise en œuvre des solutions proposées a aussi été négligé.

Nous croyons que la représentation formelle du contexte d'application d'un patron (i.e. patron de conception et style architectural) est la clé pour mieux le comprendre et l'appliquer. En effet, l'utilisation d'un patron passe par trois étapes :

1. la reconnaissance du patron comme une solution potentielle au problème de conception qu'on essaye de résoudre,
2. la compréhension du patron et la maîtrise des principes sous-jacents au patron pour pouvoir l'utiliser adéquatement dans le contexte considéré, et
3. l'application du patron choisi au problème considéré.

Or, ces étapes supposent une bonne compréhension du problème de conception, de la solution proposée par un patron au problème en question, ainsi que la transformation nécessaire pour passer du problème vers la solution.

Chapitre 3

Représentation et mise en œuvre des patrons de conception par représentation explicite du problème

3.1 Principes de notre approche

Notre approche pour la représentation et la mise en œuvre des patrons de conception se base sur la représentation explicite des problèmes de conception résolus par les patrons. En effet, la représentation explicite du problème de conception résolu par un patron permet de caractériser et de documenter d'une façon plus précise un patron et ses indications d'utilisation. Elle permet aussi d'évaluer la pertinence du patron à un modèle donné et de mieux comprendre l'effet de l'application du patron, en illustrant aussi bien l'état d'un modèle *avant* son application que l'état *après* son application. Finalement, elle permet d'automatiser l'application d'un patron en la représentant par une transformation d'une instance du problème en une instance de la solution.

Ainsi, nous proposons une approche où un patron de conception est représenté par un triplet <MP, MS, T> où :

- MP est une caractérisation du problème résolu par le patron sous la forme d'un méta-modèle dont les instances correspondent à des situations où le patron est applicable. Nous appelons MP modèle[1] de problème,
- MS est une description de la structure proposée par le patron pour résoudre le problème de conception en question. Cette description est aussi sous la forme d'un méta-modèle appelé modèle de solution,
- T est une transformation reflétant la mise en œuvre du patron, qui consiste à transformer une instance d'un problème de conception en une instance de la solution correspondante. T fait référence aux (méta-)modèles MP et MS.

Notre approche se base sur l'existence dans un environnement de développement d'un catalogue de ces triplets <MP, MS, T> (Figure 3.1). Étant donné un modèle en entrée, nous essayons d'abord de détecter des instances de problèmes résolus par les patrons de conception dans ce modèle. Cela consiste à chercher pour chaque modèle de problème du catalogue, des configurations ou fragments du modèle en entrée, qui lui sont conformes. Si on détecte une instance d'un problème dans le modèle en entrée, les entités de cette instance sont annotées par le rôle qu'elles jouent dans le problème de conception détecté ; les rôles correspondent aux entités définies dans le modèle de problème détecté. Cela suggère que le patron de conception associé au problème détecté peut être applicable au modèle reçu en entrée. Donc à la fin de cette étape que nous appelons *marquage,* on obtient un modèle marqué. Le concepteur peut alors appliquer la transformation associée au patron en question au modèle marqué. Le résultat de cette transformation est le modèle en entrée où une instance d'un problème de conception associé à un patron a été remplacée par la solution proposée par le patron en question.

---

[1] MP est un méta-modèle au sens que c'est un modèle de niveau M2, i.e. ses instances sont des modèles. MP est un modèle de problème au sens étymologique du terme puisqu'il représente la structure typique ou caractéristique du problème de conception. Idem pour MS.

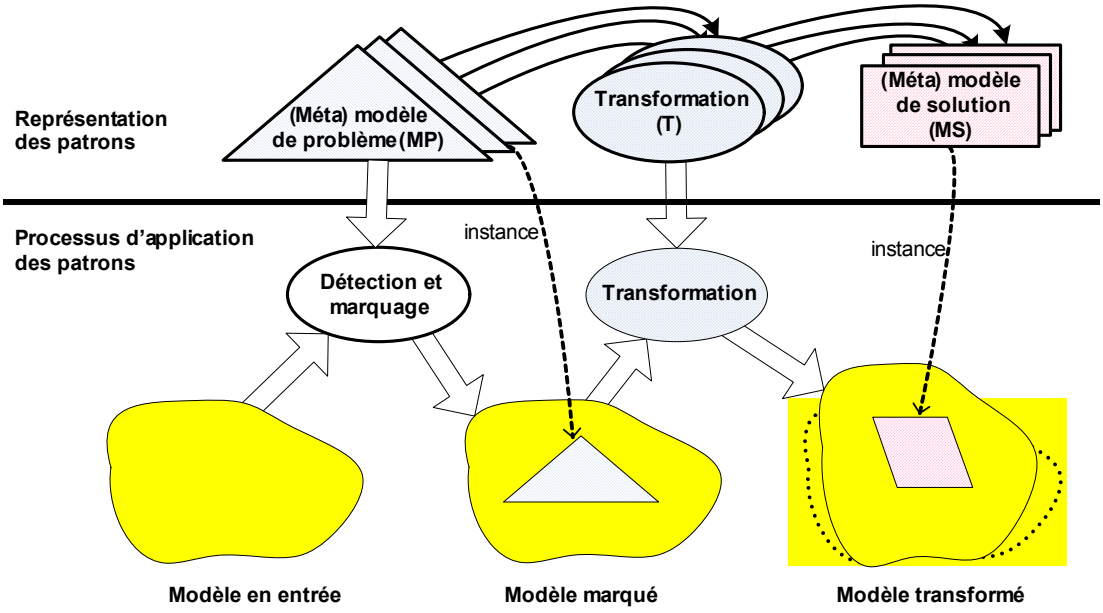


Figure 3.1. Vue globale de notre approche

Ce chapitre est dédié l'aspect représentation des patrons, en particulier, la modélisation des problèmes de conception résolus par les patrons. Nous illustrerons notre approche à la représentation des problèmes de conception à l'aide de trois exemples de patrons de Gamma et al.(1995) : le patron Visiteur (sections 3.2 et 3.3), le patron Composite (section 3.4) et le patron Pont (annexe 1). La section 3.2 introduit le langage de représentation des problèmes par l'exemple du patron Visiteur. La représentation de la solution est présentée à la section 3.3. La section 3.4 illustre le modèle de problème et le modèle de solution pour le patron Composite. Nous décrivons à la section 3.5 les principales constructions que nous avons du ajouter à UML pour représenter les modèles de problèmes et de solutions. Notre implémentation est décrite à la section 3.6.

Notre approche pour la détection des problèmes de conception et le marquage des modèles est décrite au chapitre 4, alors que l'aspect représentation et mise en œuvre des transformations est présenté au chapitre 5.

3.2 Modèle du problème résolu par le patron Visiteur

Le problème résolu par le patron Visiteur a été décrit textuellement dans (Gamma et al., 1995). L'exemple de la Figure 2.8 représente un exemple de situation où le problème de conception—résolu par le patron Visiteur—est présent. Notre objectif est de capturer l'essence de ce problème dans un modèle formel que l'on peut lire et manipuler facilement. Pour ce faire, nous proposons de définir un méta-modèle de ce problème, i.e. un modèle dont les instances seront des modèles tels que celui de la Figure 2.8. En fait, les instances de ce modèle sont des modèles de niveau analyse ou conception présentant les symptômes du problème résolu par le patron Visiteur.

Nous présentons une première esquisse de notre modèle de problème à la Figure 3.2. C'est un modèle UML avec quelques extensions que nous discuterons au fur et à mesure. Ce modèle représente une hiérarchie de classes implémentant le même comportement. La racine de cette hiérarchie est représentée par la méta-classe `AbstractClass` qui définit un certain nombre d'opérations abstraites représentées par l'entité `AbstractOperation`. `AbstractClass` a un certain nombre de sous-classes (`ConcreteClass`) qui implémentent toutes les opérations abstraites de `AbstractClass`. L'association `Inherits_from` représente la relation d'héritage qui peut exister entre deux classes d'un modèle en entrée. Cette relation a une sémantique différente de celle du méta-modèle UML. En fait, nous avons introduit notre propre relation d'héritage car nous avons besoin de spécifier l'héritage entre un ensemble de classes et une super-classe; un héritage avec une multiplicité. La multiplicité 1..* associée à `Inherits_from` spécifie que `AbstractClass` peut avoir une ou plusieurs sous-classes de type `ConcreteClass`. Cette dernière peut aussi avoir plusieurs sous-classes correspondant à un niveau plus bas d'abstraction, cela est représenté par la relation `Inherits_from` qui relie `ConcreteClass` à elle même.

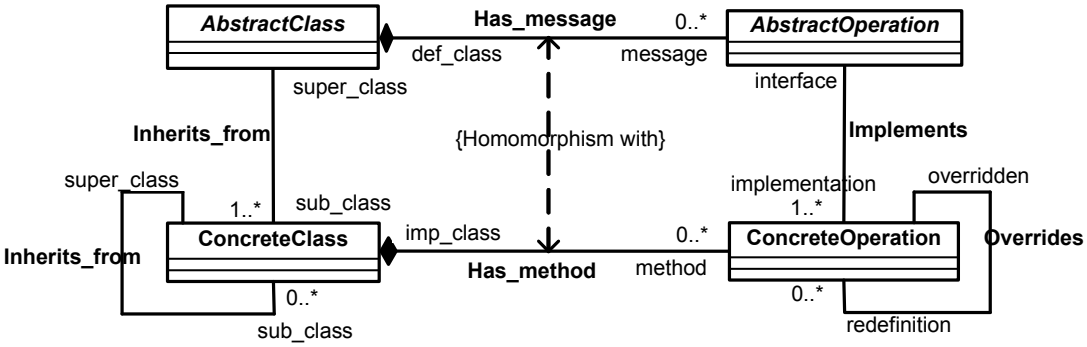


Figure 3.2. Modèle statique du problème résolu par le patron « Visiteur »

Pour exprimer le fait que chaque classe concrète a une opération `ConcreteOperation` (le lien `has_method`) pour chaque `AbstractOperation` supporté par (lien `has_message`) sa super-classe `AbstractClass`, nous utilisons une contrainte appelée `Homomorphism_with`. Pour garder le modèle simple, nous ne représentons pas les paramètres et les types de retour des opérations.

Le modèle de la Figure 3.2 est appelé modèle *statique* du problème résolu par le patron Visiteur. En effet, tel quel, ce modèle ne représente pas encore des situations où le patron Visiteur peut être appliqué; il représente une simple hiérarchie de classes. En fait, l'information manquante dans ce modèle est le type d'évolution future de la hiérarchie de classes que le patron Visiteur résout, i.e. le scénario selon lequel la fonctionnalité offerte par la hiérarchie de classes est appelée à évoluer. En effet, comme nous l'avons déjà évoqué à la section 2.4.1, le problème de conception auquel s'attaque ce patron est l'existence d'une hiérarchie de classes dans laquelle les classes offrent sensiblement la même fonctionnalité, mais avec une implémentation différente. Cette configuration n'est pas un problème en soi mais c'est son évolution possible qui pose problème. En fait, lorsque la hiérarchie des classes est stable, alors que les traitements (l'ensemble des méthodes) ne le sont pas, l'ajout d'une nouvelle fonctionnalité devient une opération pénible et coûteuse. En effet, pour chaque nouvelle fonctionnalité il faudra définir une méthode abstraite au niveau de la racine et ajouter la méthode à chaque classe de la hiérarchie. En pratique, un bon nombre des patrons de conception proposés par Gamma et al. visent à augmenter la résilience

de modèles (ou applications) face à des scénarios d'évolution, et le patron Visiteur en est un exemple.

Nous avons donc complété le modèle de la Figure 3.2 pour spécifier que le nombre d'opérations/de méthodes est susceptible d'augmenter. En fait, nous avons pu représenter les différents types de changements que nous avons rencontrés en étudiant les patrons de conception de (Gamma et al., 1995) en terme de changements de cardinalités des associations au niveau méta-modèle. Par exemple, dans le cas du problème résolu par le Visiteur, la classe `AbstractClass` a un nombre changeant (i.e. croissant) d'opérations. Pour cela, nous avons modifié la cardinalité « 1..* » de l'association `Has_message` entre `AbstractClass` et `AbstractOperation` en lui ajoutant le qualificateur « ++ » qui indique que cette cardinalité est susceptible de croître avec le temps. La Figure 3.3 montre le nouveau modèle de problème qui inclut l'information concernant l'évolution dans le temps de la hiérarchie de classes. L'extrémité changeante d'une association (i.e. celle ayant une cardinalité contenant le qualificateur « ++ ») est appelée point de variation dans le temps (*time hotspot*).

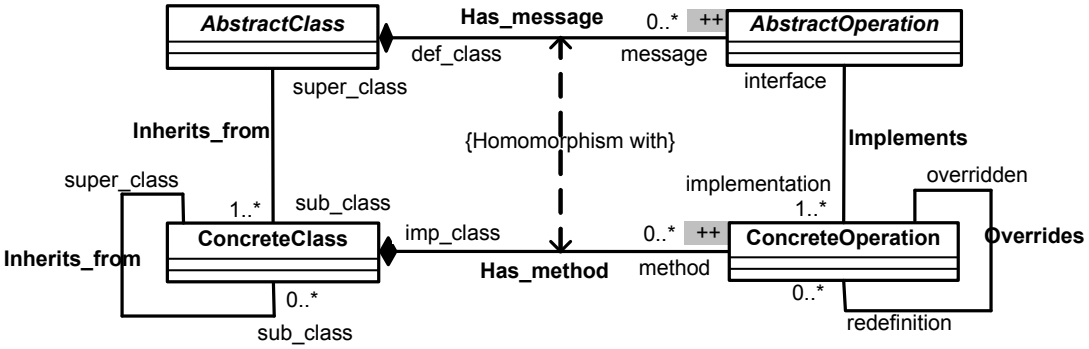


Figure 3.3. Modèle du problème résolu par le patron « Visiteur »

Notons que sans les points de variation, les modèles de problèmes (i.e. modèles statiques) des patrons Visiteur, Pont et Stratégie sont identiques : une hiérarchie de classes qui implémentent les mêmes opérations ! Une brève description du patron Pont ainsi que des représentations du problème de conception qu'il résout et de la solution qu'il propose sont données dans l'annexe 1.

Notons que nous ne représentons que les éléments concernés par l'application du patron, i.e. les éléments faisant partie de la spécification du problème. Par exemple, rien n'empêche que dans une instance du modèle de problème, la classe correspondant à `AbstractClass` puisse avoir des méthodes concrètes qui ne sont pas redéfinies par ses sous-classes—auquel cas la classe n'est pas une classe abstraite pure. Ces méthodes concrètes ne sont pas représentées par notre modèle de problème puisqu'elles ne font pas partie du problème à résoudre et ne subissent aucune transformation lors de l'application du patron. Nous discuterons à la section 3.5 l'ensemble des constructions que nous avons ajoutées au méta-modèle UML pour pouvoir spécifier les modèles des problèmes.

3.3 Modèle de la solution proposée par le patron Visiteur

Le patron « Visiteur » propose une solution au problème décrit dans la section précédente. Cette solution consiste à implanter chacune des opérations par une classe distincte qui regroupera les différentes implémentations de l'opération pour les différentes classes de la hiérarchie. Pour représenter le modèle de solution, nous avons utilisé une approche similaire à celle que nous avons utilisée pour représenter le modèle du problème. La Figure 3.4 montre notre représentation de la solution proposée par le patron Visiteur.

Dans ce modèle, une hiérarchie de classes représente les opérations (les objets visiteurs) et une autre les éléments (objets visités) sur lesquels ces opérations s'appliquent. Toutefois, contrairement à la structure de la solution telle qu'introduite par (Gamma et al., 1995) (Figure 2.9), notre modèle n'indique pas un nombre précis d'éléments concrets ni de visiteurs concrets. En effet, les classes d'objets sur lesquels les opérations doivent s'appliquer sont représentées par la famille de classes `ConcreteElement`. Les classes de type « Visiteur » sont aussi regroupées dans une famille de classes `ConcreteVisitor`. Le lien entre les deux hiérarchies est établi par le fait que les classes de type `ConcreteElement` sont les types des paramètres des opérations de la classe de type `Visitor`. La cardinalité « 1..* » de l'association

`Inherits_from` qui existe entre `ConcreteElement` et `Element` indique qu'il peut y avoir plusieurs classes de type `ConcreteElement` dans une instance de ce modèle.

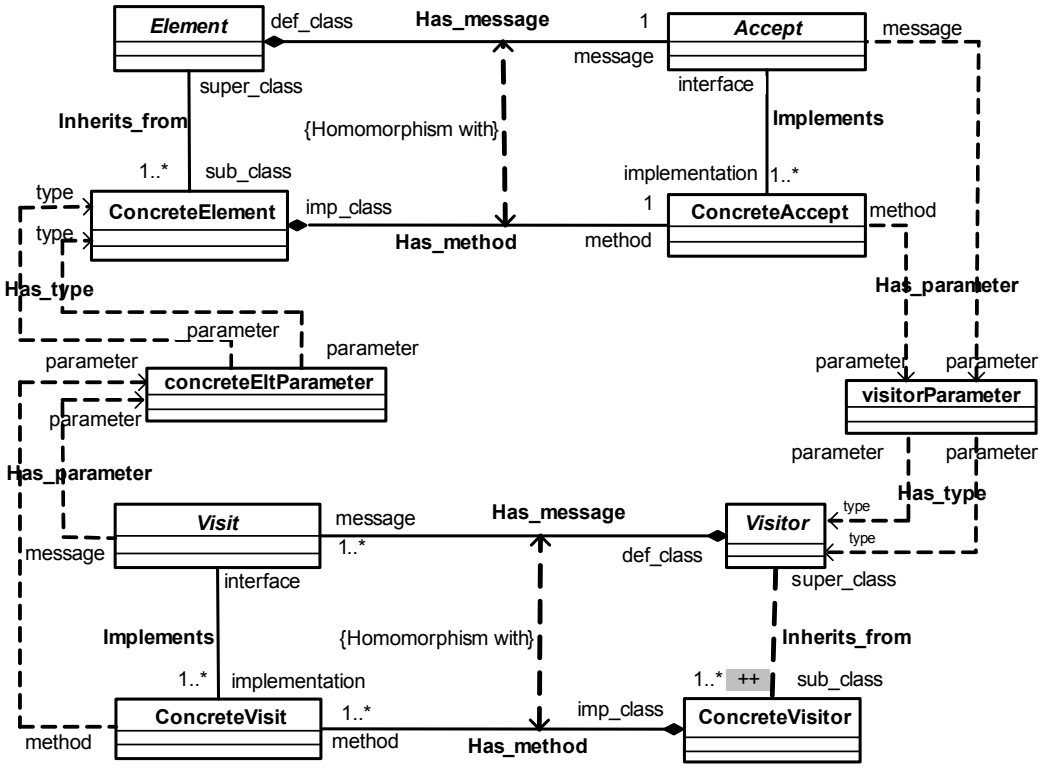


Figure 3.4. Représentation de la solution proposée par le patron « Visiteur »

Chaque classe de type visiteur (`ConcreteVisitor`) regroupe l'ensemble des implémentations d'une opération donnée. De ce fait, chaque ajout d'opération dont l'implémentation dépend des classes `ConcreteElement`, se traduit par l'ajout d'une classe à la famille `ConcreteVisitor`. La classe ajoutée regroupe toutes les implémentations possibles de cette nouvelle opération. La famille des classes de type visiteur (`ConcreteVisitor`) est donc extensible, i.e. le nombre de classes correspondant à `ConcreteVisitor` peut croître. Cela est spécifié dans notre modèle par la cardinalité « 1..*,++ » de l'association `Inherits_from` que la famille des `ConcreteVisitor` a avec la classe `Visitor`.

De façon similaire au modèle de problème, nous ne représentons dans le modèle de solution que les éléments concernés par l'application du patron, i.e. les éléments faisant partie de la mise en œuvre du patron. Ces éléments incluent donc les

classes (`Visitor`, `Element`, `ConcreteVisitor` et `ConcreteElement`), les opérations (`Visit`, `ConcreteVisit`, `Accept` et `ConcreteAccept`), les paramètres de ces opérations (`concreteEltParameter` et `visitorParameter`) et leur type. Toutefois, nous avons aussi ajouté deux contraintes « `Homomorphism_with` » pour spécifier, d'une part, que chaque classe de type `ConcreteElement` a une opération `ConcreteAccept` pour chaque opération abstraite `Accept` supportée par sa super-classe `Element` et, d'autre part, que chaque classe de type `ConcreteVisitor` a une opération `ConcreteVisit` pour chaque opération abstraite `Visit` supportée par sa super-classe `Visitor`.

3.4 Représentation du patron Composite

3.4.1 Description

Il existe des situations où des composants peuvent être regroupés pour former d'autres composants lesquels peuvent être regroupés pour en former d'autres. On peut parler de composition récursive. Le problème est qu'on a aussi besoin de permettre aux objets clients de traiter ces différents composants et composites de la même manière. Supposons que nous désirions développer une application financière qui supporte la gestion des portefeuilles. Cette application doit permettre de construire des portefeuilles composites à partir d'actifs individuels. Ces actifs peuvent être groupés pour former des portefeuilles, lesquels peuvent à leur tour être groupés pour former de plus larges portefeuilles ou comptes. Une simple implémentation de cette application pourrait définir des classes pour les actifs individuels telles que des *actions* (`Stock`) et des *obligations* (`Bond`) et des classes (e.g. `Portfolio`, `Account`) qui sont des conteneurs de ces actifs primitifs (Figure 3.5).

Le problème avec une telle implémentation est que le code client, qui utilise ces classes, doit traiter les objets de type *action simple* et les objets de type *portefeuille* différemment, malgré que l'utilisateur les traite identiquement, e.g. un utilisateur voudrait afficher l'état actuel de ces actifs qu'ils soient de *simples actions* ou des *portefeuilles*. Ainsi une telle approche rendra le code client plus complexe et

plus difficile à maintenir et à étendre, i.e. plusieurs blocks conditionnels (`case` ou `if`) comme le montre la Figure 3.5.

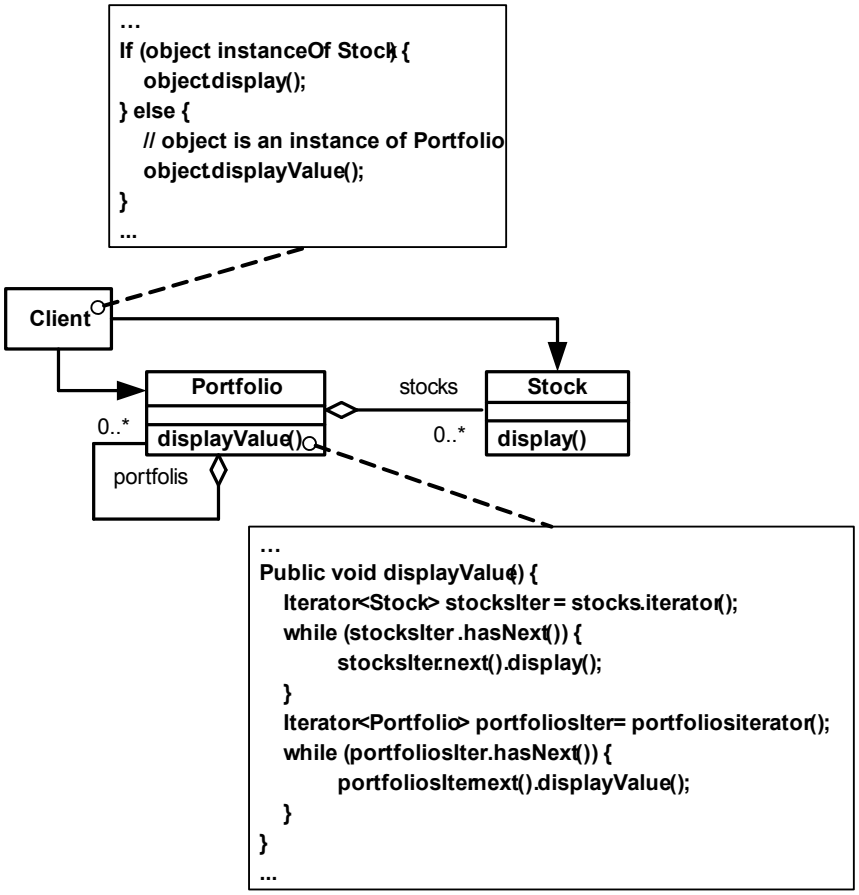


Figure 3.5. Une simple implémentation de l'application de gestion des portefeuilles

Le patron Composite décrit comment utiliser la composition récursive de façon à ce que les clients traitent de façon uniforme les objets individuels et les objets composites. L'élément clé dans la solution proposée par le patron Composite est une classe abstraite qui représente autant les composants simples (primitifs) que les composites (conteneurs de composants). Dans notre exemple, cette classe appelée `Asset` représente un *actif* (Figure 3.6). Elle définit les opérations communes à tous les objets de type `Asset`, e.g. l'opération `display()`.

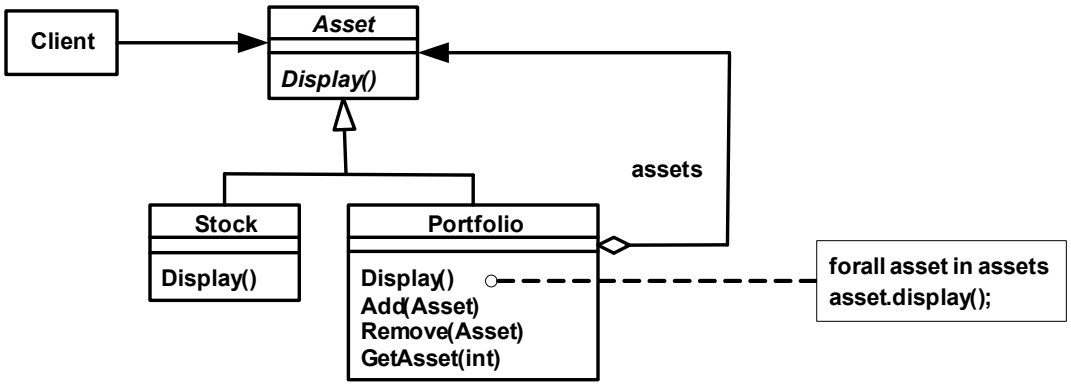


Figure 3.6. Utilisation du Composite dans l'application de gestion des portefeuilles

La classe `Stock` représente un objet primitif (i.e. actif individuel). Elle implémente l'opération `display()` pour afficher sa valeur actuelle. La classe `Portfolio` est une agrégation d'objets `Asset`. L'opération `display()` de la classe `Portfolio` appelle l'opération `display()` des actifs la composant sans se préoccuper de leur nature (individuelle ou composite). La classe `Portfolio` déclare aussi des opérations pour accéder et gérer les actifs qui la composent. La Figure 3.7 montre la structure typique de ces objets récursivement composés.

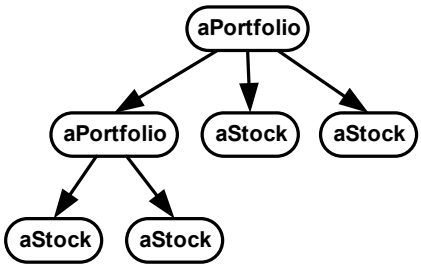


Figure 3.7. Objets composés récursivement.

La Figure 3.8 montre la structure générique du patron Composite. Les classes clients utilisent l'interface `Component` pour interagir avec les objets dans la hiérarchie de composition (i.e. `Composite` et `Leaf`). Les requêtes reçues par les objets feuilles (`Leaf`) sont directement traitées alors que celles reçues par un objet `Composite` sont transmises à ses composants (`Component`), qu'ils soient des objets de type `Leaf` ou `Composite`.

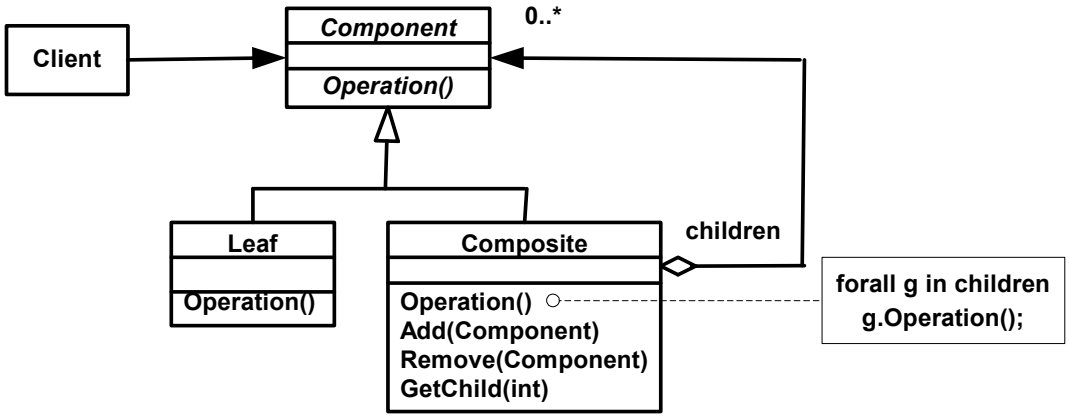


Figure 3.8. Structure du patron Composite.

3.4.2 Modèle du problème résolu par le patron Composite

Pratiquement, la solution proposée par le patron Composite se base sur deux éléments clés : i) l'héritage entre Component et Composite qui fait que l'un peut se substituer à l'autre dans une variable, et ii) l'uniformisation de l'API entre le composite et les composants, ce qui permet de les utiliser de façon transparente. Ainsi et de façon intuitive, la représentation du problème de conception associé à ce patron serait un modèle où on a failli à un de ces deux éléments ou les deux. Le (méta)modèle de la Figure 3.9 est une abstraction du problème de conception lorsque les deux éléments font défaut : pas d'héritage et pas d'API commune. Ce modèle présente deux classes : une décrivant les objets composites (`CompositeComponent`) et une décrivant les objets primitifs (`SimpleComponent`). Les deux classes exhibent un comportement similaire. Cela est représenté par la contrainte `Has_same_behavior` existant entre les opérations définies par ses classes (`CompositeOperation` et `SimpleOperation`).

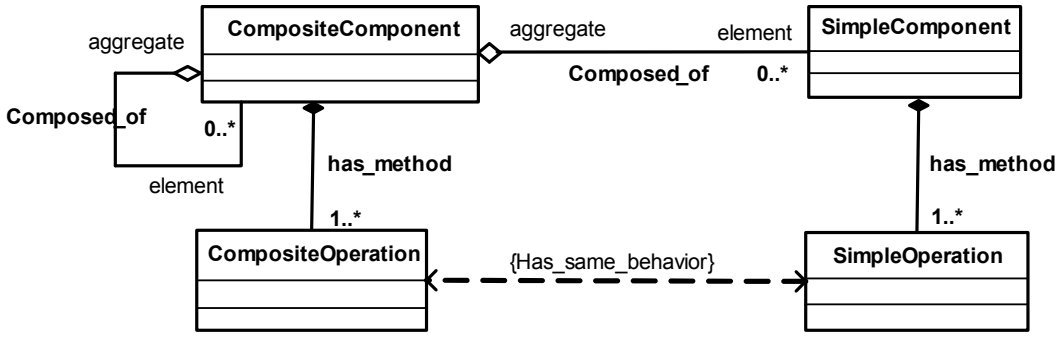


Figure 3.9. Modèle du problème résolu par le patron Composite (1ère esquisse)

Le problème avec la représentation de la Figure 3.9 est qu'il n'est pas possible d'en détecter les instances dans des modèles et de les marquer automatiquement (première phase de notre approche, voir Figure 3.1). En effet, il n'est pas possible de vérifier à partir d'un simple diagramme de classes que deux méthodes, ayant des signatures différentes, implémentent le même comportement à moins d'avoir accès au code ou au modèle comportemental. Dans notre exemple de la Figure 3.5, sans analyse du code, nous n'aurions pas été capables de reconnaitre la méthode `display()` comme faisant partie de l'interface commune aux composants et aux agrégats, malgré le fait qu'elle porte le nom `displayValue()` dans l'agrégat (`Portfolio`), et `display()` dans le composant simple (`Stock`).

En pratique, seule une représentation du problème où l'héritage fait défaut, donnerait un modèle de problème dont les instances peuvent être détectées dans un modèle en entrée, i.e. il est plus facile de reconnaître une API commune aux composants et le composite. Ainsi, notre deuxième esquisse du modèle de problème résolu par le patron Composite, est en fait une implémentation incomplète du patron Composite (Figure 3.10). Nous discutons plus en détail au chapitre 6 des différentes catégories de problèmes de conception que nous sommes aptes à représenter et détecter dans les modèles en entrée. Dans le nouveau modèle de problème, qui décrit en pratique une mauvaise implémentation du patron, la contrainte `Has_same_behavior` a été remplacée par la contrainte `Has_same_signature`.

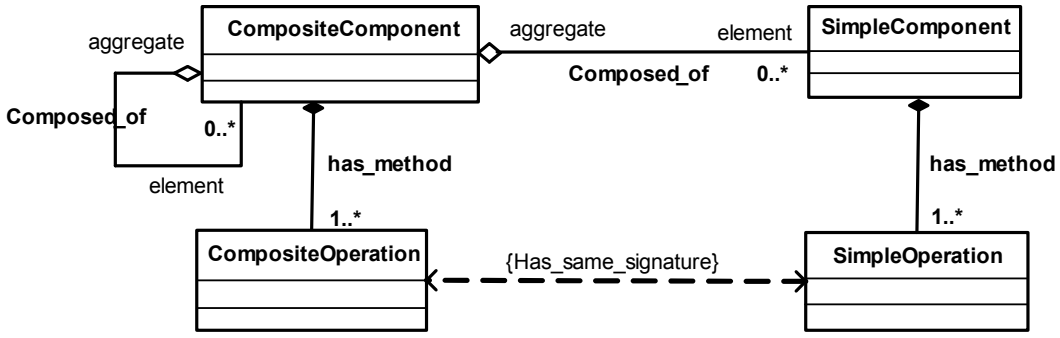


Figure 3.10. Modèle du problème résolu par le patron Composite (2ième esquisse)

3.4.3 Modèle de la solution proposée par le patron Composite

La Figure 3.11 montre notre (méta)modèle de la solution proposée par le patron Composite. Ce modèle représente une hiérarchie de classes dont la racine est la classe `Component` et les sous-classes sont `Composite` et `Leaf`. Notons que le modèle spécifie qu'il peut y avoir plusieurs sous-classes de type `Leaf` (i.e. la cardinalité « 1..* » de la relation `Inherits_from` existant entre `Leaf` et `Component`).

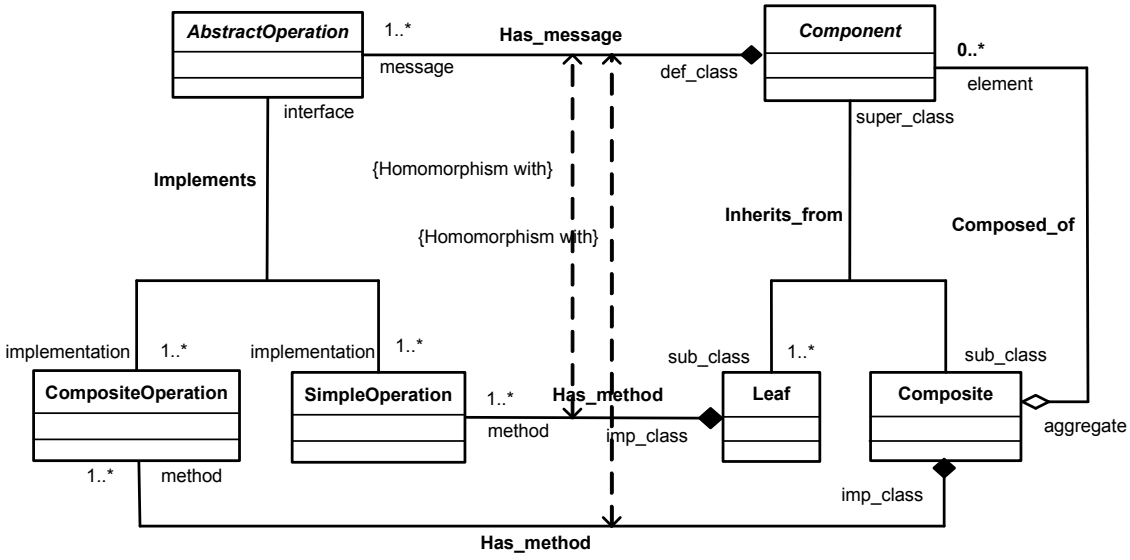


Figure 3.11. Représentation de la solution proposée par le patron Composite

De façon similaire au patron Visiteur, nous ne représentons dans le modèle de solution que les éléments concernés par l'application du patron, i.e. les éléments faisant partie de la mise en œuvre du patron Composite. Ces éléments incluent donc

les classes (`Component`, `Composite`, et `Leaf`) et les opérations (`AbstractOperation`, `SimpleOperation`, et `CompositeOperation`). Nous avons ajouté deux contraintes « `Homomorphism_with` » pour spécifier, d'une part, que chaque classe de type `Leaf` a une opération `SimpleOperation` pour chaque opération abstraite `AbstractOperation` supportée par sa super-classe `Component` et, d'autre part, chaque classe de type `Composite` a une opération `CompositeOperation` pour chaque opération abstraite `AbstractOperation` supportée par sa super-classe `Component`.

3.5 Constructions nécessaires pour notre langage de spécification des patrons

L'étude des patrons présentés dans (Gamma et al., 1995) nous a amené à déduire les constructions à ajouter au méta-modèle de base d'UML pour permettre la spécification de nos modèles de problèmes et de solutions. Les principales constructions que notre langage pour la spécification des patrons doit supporter sont :

- La notion d'« évolutivité » qui est une des notions les plus importantes à spécifier. En effet, comme nous l'avons déjà évoqué, plusieurs patrons visent à protéger des classes clientes de l'évolution ou la variation possible et probable d'une application. En fait, un point de variation (*time hotspot*) résolu par un patron consiste en d'éventuels ajouts d'entités à une famille existante d'entités. Il peut représenter l'évolution du nombre de sous-classes d'une classe donnée (e.g. les patrons « Pont », « Abstract Factory »), ou du nombre d'opérations d'une classe donnée (e.g. « Visiteur », « Décorateur »), ou encore, du nombre d'implémentations d'une opération donnée (e.g. « Stratégie », « Patron de méthode »). Après étude de ces différents types d'évolution, nous les avons spécifiés sous forme de changements des cardinalités d'associations au niveau des modèles de problème. Nous avons donc représenté les points de variation en ajoutant le symbole « ++ » aux cardinalités des associations concernées.
- La notion de "famille" qui représente un ensemble d'entités du même type (de niveau méta-méta-modèle) qui partagent certaines propriétés, et qu'on peut

traiter comme une seule entité. Ces entités peuvent être des classes ou des méthodes. Par exemple, un ensemble de sous-classes d'une classe donnée (e.g. « Visiteur », « Pont », « Abstract Factory »), un ensemble de classes fortement couplées (e.g. « Médiateur »), un ensemble de classes appartenant au même paquetage (e.g. « Façade »), un ensemble de classes utilisant à peu près la même interface (e.g. « Façade »), etc. Dans plusieurs cas, on a eu aussi besoin de représenter un ensemble d'opérations appartenant à une classe ou à l'ensemble des classes appartenant à une même famille (e.g. « Visiteur », « Pont »).

- Une sémantique particulière pour les associations impliquant les familles. En fait, nous avons introduit de nouvelles associations pour spécifier des relations d'héritage et d'implémentation ayant des cardinalités. En effet, une famille de classes peut hériter d'une classe ou implanter une interface (e.g. « Pont », « Visiteur », « Fabrication abstraite », « Stratégie », « Composite », etc.). Généralement parlant, une famille de classes peut être reliée à une classe ou à une autre famille de classes. Cette relation peut être de type héritage, dépendance, implémentation ou agrégation.
- La notation « 0..** » (*zero-to-too-many*) qui caractérise une cardinalité trop élevée ou une interaction trop complexe. En effet, nous avons constaté que dans certains cas, les problèmes de conception résultent de la présence d'interactions trop complexes, comme par exemple le cas où un objet est couplé à un trop grand nombre d'objets. De tels problèmes sont traités, par exemple, par les patrons « Médiateur » et « Chaîne de responsabilité ». Le seuil pour dire que la nature de la cardinalité est « 0..* » (*many*) ou « 0..** » (*too-many*) est une valeur à définir par le concepteur ou par des métriques.
- La représentation d'un certain nombre de contraintes qui seraient autrement trop laborieuses à décrire directement en UML, dont la contrainte « `Homomorphism_with` » entre associations (e.g. « Visiteur », « Pont », etc.), ou la contrainte « `Has_same_signature` » entre opérations signifiant que des opérations ont la même signature (e.g. « Composite », « Stratégie », etc.).

- La représentation des "constantes". En effet, quelques patrons ont nécessité l'introduction de la notion de constantes lors de la modélisation des solutions qu'ils proposent. Nous distinguons donc les entités (classes, méthodes ou attributs) *littérales* spécifiées par un modèle de solution et qui vont apparaître telles quelles durant l'instanciation du patron de celles qui sont des entités *paramètres*. Par exemple, dans le cas du patron « Observateur », le modèle de solution spécifie une classe *littérale* nommée `Observable` qui est la super-classe de tous les objets à observer et qui définit un certain nombre d'opérations pour attacher, détacher et notifier les objets qui sont des observateurs. La méthode « `notify` » du patron « Observateur » est un exemple de méthode spécifiée comme étant *littérale*. Dans la plupart des patrons (e.g. « Pont », « Stratégie », « Facade », « Proxy », « Adaptateur », etc.), toutes les opérations qui apparaissent dans le modèle de la solution proviennent directement des données du problème, ce sont des opérations *paramètres*. Enfin, dans le cas du patron « Singleton », par exemple, on a besoin d'ajouter—s'il n'existe pas déjà—un attribut qui correspond à l'instance unique de cette classe. Cet attribut est défini *littéralement* dans le modèle de la solution.

Hormis les modifications et ajouts que l'on vient de citer, notre langage pour décrire les modèles de problèmes et de solutions reste sensiblement UML. Un résumé de la notation que nous utilisons dans notre langage est présenté à l'annexe 2.

Notons que les concepts `AbstractClass`, `ConcreteClass`, `AbstractOperation`, `ConcreteOperation`, `Element`, `ConcreteElement` vus précédemment ne font pas partie des primitives du langage. Ce sont des concepts dont la sémantique est définie par le concepteur qui décrit les patrons. Par contre, ces concepts doivent hériter du noyau d'UML qui est compatible avec MOF. Par exemple, `AbstractClass`, `ConcreteClass`, `Element`, `ConcreteElement`, sont des classifieurs (*Classifiers*), et `AbstractOperation` et `ConcreteOperation` sont des opérations.

3.6 Implémentation

Nous avons implanté notre langage de représentation des patrons, ainsi que les procédures de détection et d'application des patrons (présentées aux chapitres 4 et 5) au sein de l'environnement de développement Eclipse™. Plus particulièrement, nous avons construit un méta-modèle pour spécifier les modèles des problèmes et des solutions tels que présentés aux sections précédentes et cela en utilisant le cadre d'application Eclipse Modeling Framework™ (EMF) (EMF, 2005) (Budinsky et al., 2003). Nous prévoyons de déployer notre outil sous forme d'un module (plug-in) Eclipse offrant un catalogue de patrons et permettant de les appliquer.

Ainsi, pour implanter et utiliser notre méta-modèle, nous avons choisi d'étendre le méta-modèle EMF qui est basé sur MOF (*Meta-Object Facility*) (MOF, 2003), et cela pour plusieurs raisons. D'abord, le désir de travailler dans un environnement de développement ouvert, du domaine public (*open source*) et extensible. Ensuite, permettre la visualisation graphique des modèles manipulés. Finalement, supporter la sérialisation des modèles de problèmes et de solutions, et des modèles manipulés sous la forme du standard d'échange XMI (*XML Metadata Interchange*).

Nous commencerons d'abord par une description du cadre EMF (section 3.6.1). Ensuite, nous décrirons en détail notre méta-modèle pour la représentation des patrons et son implémentation (section 3.6.2). Finalement, nous présenterons un exemple d'instanciation d'un modèle de problème, en l'occurrence celui du patron Visiteur (section 3.6.3).

3.6.1 Description du cadre EMF

3.6.1.1 Aperçu

EMF est un cadre d'application (*framework*) développé en Java destiné à l'environnement Eclipse. Il relie les concepts de modélisation directement à leur implémentation. En effet, EMF supporte des fonctionnalités pour la génération de code qui permettent de traduire rapidement les modèles en code Java. Cela permet de

se concentrer sur le modèle plutôt que son implémentation. En fait, EMF a été introduit comme une implémentation de la spécification MOF (*Meta Object Facility*). Il peut être considéré comme une implémentation en Java d'un sous-ensemble noyau de l'API MOF. Cependant, du fait que EMF est orienté vers la génération de code, il présente quelques différences conceptuelles par rapport à MOF (Gerber et Raymond, 2003).

EMF est compatible avec l'approche MDA. Il est considéré comme une implémentation d'une partie de MDA dans la famille des outils basés sur Eclipse. EMF utilise XMI (*XML Metadata Interchange*) comme sa forme canonique pour définir les modèles. Il permet aussi d'accéder d'une façon réflexive aux modèles. Il y a plusieurs façons pour créer un modèle EMF :

- Créer un document XMI directement;
- Exporter un document XMI à partir d'un outil de modélisation tel que Rational Rose;
- Créer des interfaces Java et les annoter avec les propriétés du modèle; ou
- Utiliser un schéma XML pour décrire le modèle.

Modélisation et programmation sont donc considérées au même niveau. En effet, selon l'expertise du concepteur, il peut utiliser une forme pour créer son modèle et générer les autres grâce à EMF. Ce dernier peut être vu comme un unificateur des trois importantes technologies : Java, XML et UML. Ainsi, EMF définit une terminologie commune pour décrire ces différents types de modèles (i.e. les aspects statiques de java, les modèles objets UML, et les dialectes XML) et un modèle pour décrire les modèles EMF.

3.6.1.2 Le méta-modèle EMF (*Ecore Model*)

Le modèle utilisé pour représenter les modèles EMF est appelé le «*Ecore Model*». Les interfaces des éléments de ce modèle sont regroupées dans un paquetage appelé « `org.eclipse.emf.ecore` » dont la hiérarchie est présentée à la Figure 3.12. Cette hiérarchie a pour racine l'interface `EObject`. Les classes qui implantent les interfaces représentées en gris à la Figure 3.12, sont des classes abstraites

(`EModelElement`, `ENamedElement`, `ETypedElement`, `EClassifier`, et `EStructuralFeature`).

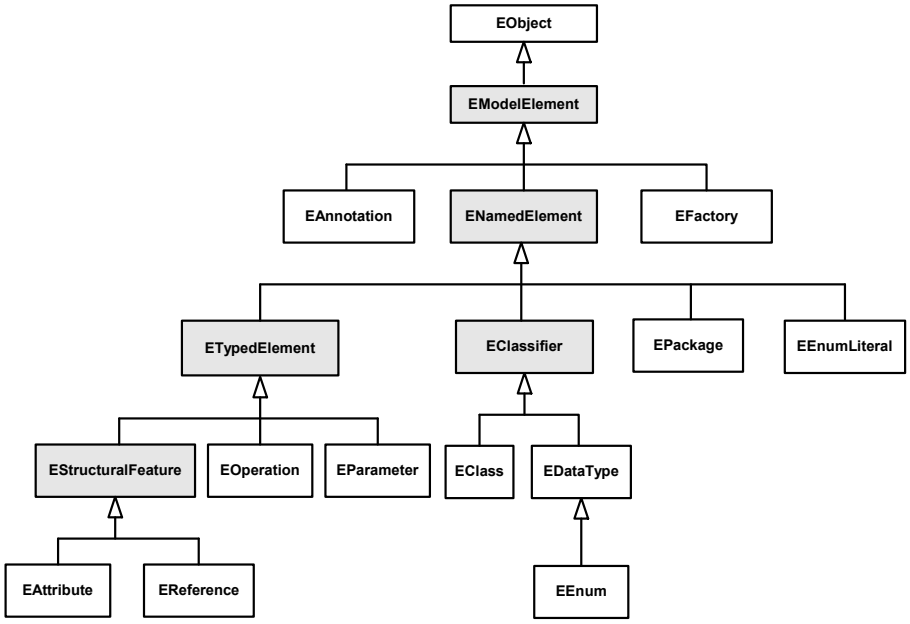


Figure 3.12. Hiérarchie des classes du paquetage *ECore* du cadre EMF.

La Figure 3.13 montre les détails du sous-ensemble de EMF dont nous aurons besoin, par la suite, pour implémenter notre langage de description des patrons. Dans ce modèle, `EClass` représente les classes. Une classe est identifiée par un nom et peut contenir des attributs (`eAttributes`), des opérations (`eOperations`) et des références (`eReferences`). Elle peut aussi référer à d'autres classes comme étant ses super-classes (`eSuperTypes`). Un attribut est une instance de `EAttribute`, et a un nom et un type.

Une instance de `EReference` représente une extrémité d'association entre deux classes, l'équivalent de la méta-classe *AssociationEnd* dans UML. Ainsi, le concept *Association* tel que défini dans UML ou MOF (relation entre deux classes) n'existe pas dans EMF. Une référence, tout comme un attribut, a aussi un nom et un type. Ce dernier réfère à l'instance de `EClass` qui est à l'autre extrémité de l'association. Si l'association est navigable dans la direction opposée, il y aura une autre référence correspondante (`eOpposite`). Dans un tel cas, une association est définie par deux instances de `EReference` chacune définissant l'autre comme sa

référence opposée (`eOpposite`). Une référence est de type agrégation quand son attribut `containment` est égal à *true*.

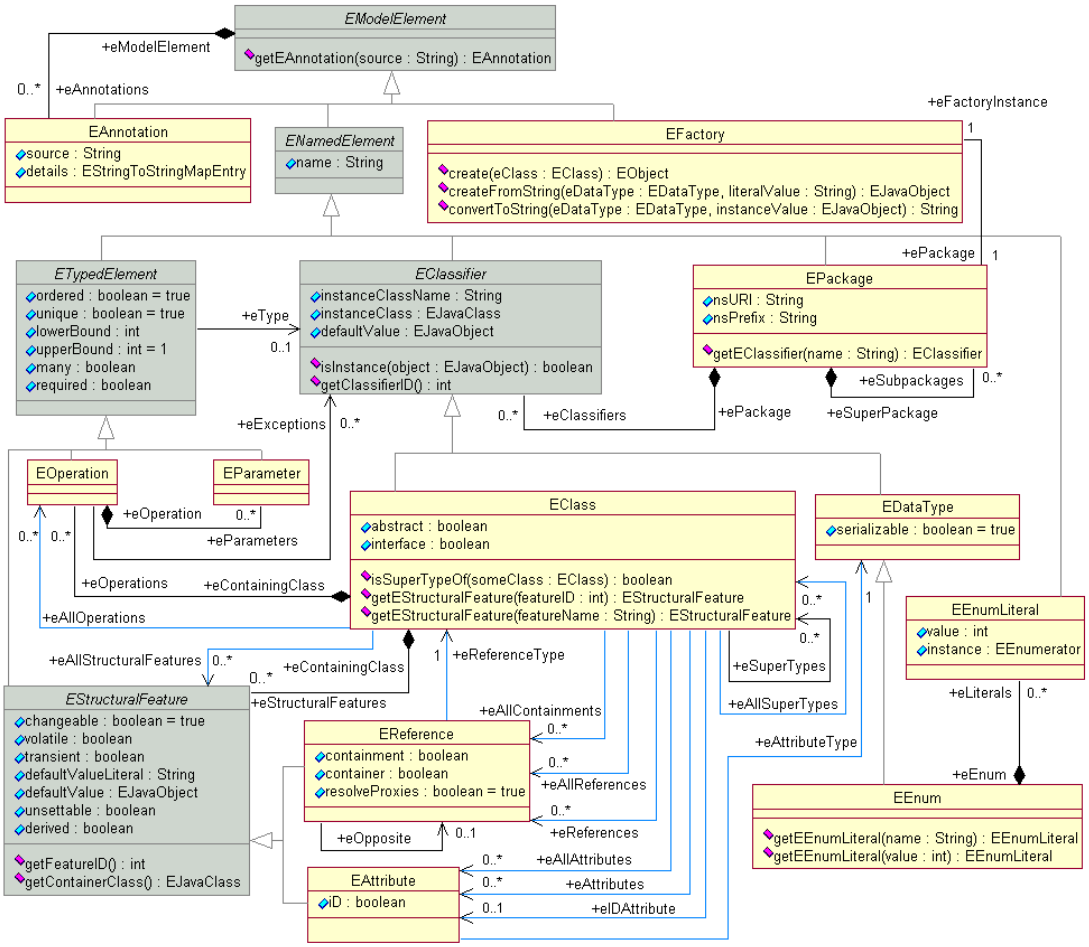


Figure 3.13. Le méta-modèle EMF

Un attribut et une référence ont plusieurs propriétés similaires, comme le nom, le type, la classe qui les contient, la multiplicité, etc. Pour regrouper ces similarités, le méta-modèle inclut une super-classe commune à `EAttribute` et `EReference` appelée `EStructuralFeature`. Cette dernière est elle-même dérivée de plusieurs super-types comme `ETypedElement` et `ENamedElement`. La classe `ENamedElement` contient un seul attribut `name` et la majorité des classes du méta-modèle étendent cette classe pour hériter de cet attribut. La classe `ETypedElement`, quant à elle, regroupe les propriétés communes aux éléments typés comme `EAttribute`, `EReference`, `EParameter` et `EOperation`. Un type d'une entité typée est défini par la référence `eType` de

`ETypedElement`. Cette référence est de type `EClassifier` laquelle définit une super-classe commune à `EClass` et `EDataType` permettant ainsi à un type d'être une classe ou un type de données. Pour ce qui est des types de données, le cadre EMF réutilise les types définis en Java. Il a cependant ajouté un type `EEnum` pour spécifier les énumérations. Une instance de `EEnum` spécifie une liste explicite de valeurs. Ces valeurs sont appelées des littéraux et sont des instances de la classe `EEnumLiterals`. La classe `ETypedElement` définit aussi les attributs `lowerBound` et `upperBound` servant à spécifier les cardinalités d'un élément typé. Il peut s'agir de la multiplicité d'une référence, d'un attribut, d'un paramètre ou de l'élément retourné par une opération.

Dans le modèle EMF, la classe `EPackage` regroupe, via la référence `eClassifiers`, un ensemble de classes ou de types de données. Une instance de `EPackage` représente donc un paquetage qui peut correspondre à un modèle ou un fragment de modèle. Lorsqu'un modèle EMF est sérialisé, l'élément de niveau document (*document-level*) autrement dit la racine représente un paquetage. Une instance de `EPackage` a un identificateur unique lequel est aussi sérialisé sous format XMI. Cet identificateur est défini par l'attribut `nsURI` (*namespace Uniform Resource Identifier*) qui correspond à l'espace de nom associé à cette instance.

Il existe plusieurs autres paquetages qui font partie de EMF mais que nous ne décrirons pas ici (e.g. paquetage de génération de code, paquetage des utilitaires, etc.). Nous montrerons, par contre, des exemples de modèles EMF (XMI) et de codes au fur et à mesure que nous présentons l'implémentation de notre méta-modèle.

3.6.2 Méta-modèle pour la spécification des patrons

Nous avons construit un méta-modèle commun aux modèles de problèmes et de solutions des patrons de conception. Nous l'avons implémenté comme une extension au modèle « `Ecore` ». Notre méta-modèle, présenté à la Figure 3.14, définit tous les concepts nécessaires pour représenter les modèles de problèmes et de solutions tels que nous les avons présentés à la section 3.5. Dans la Figure 3.14, les classes appartenant au méta-modèle EMF (`Ecore`) apparaissent en gris, alors que les

classes que nous avons ajoutées sont en blanc. Nous allons adopter dans le reste du chapitre une convention typographique qui permet de distinguer entre les classes de EMF (`courrier`), les classes de notre méta-modèle (***`courrier bold italic`***) et les classes de nos modèles de problèmes ou de solutions (*`courrier italic`*).

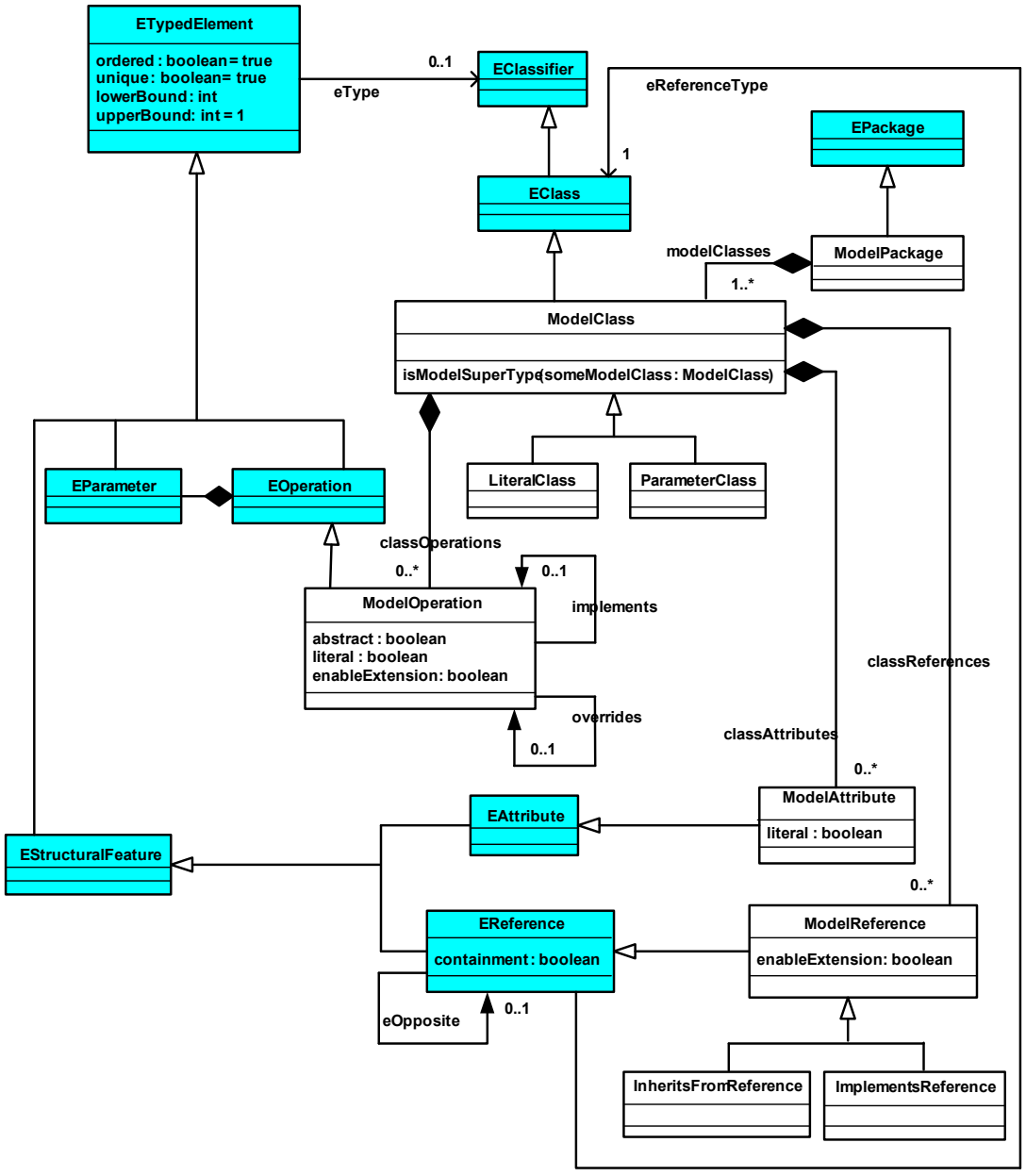


Figure 3.14. Méta-modèle pour la représentation des problèmes et des solutions

Sachant que EMF utilise la classe `EPackage` pour représenter un modèle ou un fragment de modèle, alors pour définir un nouveau méta-modèle (et donc, définir de

nouveaux types de modèles), il nous faut définir une sous-classe de `EPackage`, auprès de laquelle nous enregistrons les nouvelles méta-classes. Cette sous-classe s'appelle ***`ModelPackage`***. Un modèle de problème ou de solution est donc une instance de ***`ModelPackage`***.

Comme le montre la Figure 3.14, une instance de ***`ModelPackage`*** contient des instances de la classe ***`ModelClass`*** qui est une sous-classe de `EClass` du modèle « `Ecore` ». ***`ModelClass`*** représente toutes les classes apparaissant dans les modèles de problèmes et dans les modèles de solutions. Nous avons étendu la classe ***`ModelClass`*** en introduisant deux sous-classes ***`ParameterClass`*** et ***`LiteralClass`***. La classe ***`LiteralClass`*** représente une classe "*constante*" dont les propriétés sont entièrement définies (son nom, ses attributs, ses opérations, etc.) et qui apparaîtra généralement dans un modèle de solution. Lors de l'application d'un patron, une classe du modèle de solution de type ***`LiteralClass`***, doit être ajoutée telle quelle au modèle sur lequel le patron est appliqué (e.g. la classe *`Observable`* du patron « Observateur »). La classe ***`ParameterClass`*** représente des classes "*paramètres*" des modèles de problèmes ou de solutions. Dans l'exemple du patron Visiteur, les classes *`AbstractClass`* et *`ConcreteClass`* du modèle de problème (Figure 3.3) et les classes *`Element`*, *`ConcreteElement`*, *`Visitor`* et *`ConcreteVisitor`* du modèle de la solution (Figure 3.4) sont toutes des instances de ***`ParameterClass`***.

La Figure 3.15 montre les liens entre le modèle `Ecore`, notre méta-modèle et nos modèles de problèmes ou de solutions. ***`ModelPackage`*** et ***`ModelClass`*** appartiennent au niveau méta-modèle, ***`ModelClass`*** étend `EClass` et ***`ModelPackage`*** étend `EPackage`. Toutes les classes du niveau méta-modèle sont représentées par la classe `EClass` au niveau méta-méta-modèle. Donc, les classes ***`ModelPackage`*** et ***`ModelClass`*** sont des instances de `EClass`. La classe `EClass` est représentée par elle-même, ce qui signifie qu'elle est une instance d'elle même. Une instance de ***`ModelPackage`*** est un paquetage correspondant à un modèle. Par exemple, le modèle de solution du patron Visiteur appelé *`VisitorSolution`* est une instance de ***`ModelPackage`***. Une instance de ***`ModelClass`*** est une classe d'un modèle de problème

ou de solution, par exemple, la classe `Element` du modèle de solution du patron Visiteur.

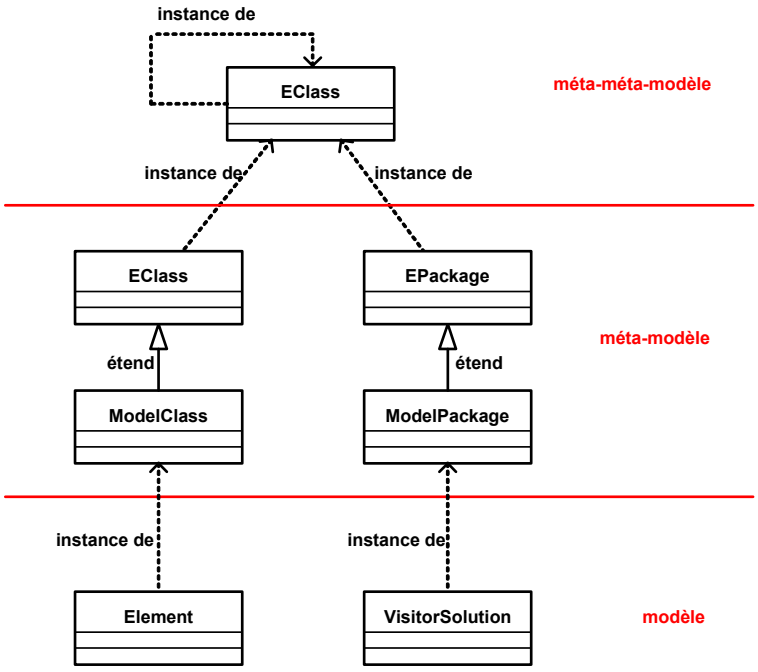


Figure 3.15. Couches de méta-modélisation

Une instance de ***ModelClass*** peut contenir des attributs, des opérations et des références (Figure 3.14). Toutes les opérations sont des instances de la classe ***ModelOperation*** qui étend la classe `EOperation` de EMF. Au lieu de définir une méta-classe pour les méthodes abstraites, une pour les méthodes concrètes, et une pour les méthodes littérales (*constantes*), nous avons défini dans ***ModelOperation*** des variables d'instances « ***abstract*** » et « ***literal*** », toutes les deux booléennes. L'attribut « ***enableExtension*** », quant à lui, permet de spécifier si l'opération correspond à une opération paramètre dont la famille est appelée à croître ou non (e.g. le modèle du problème résolu par le « Visiteur »). Dans le cas du patron Visiteur, par exemple, les opérations `Accept` et `Visit`, du modèle de solution (repris à la Figure 3.16) sont des instances de ***ModelOperation*** dont l'attribut ***literal*** est égal à *true*. En plus, la classe ***ModelOperation*** possède deux références sur elle-même : une référence ***implements*** qui permet d'indiquer qu'une opération en implémente une

autre et une référence ***overrides*** qui permet d'indiquer qu'une opération en redéfinit une autre.

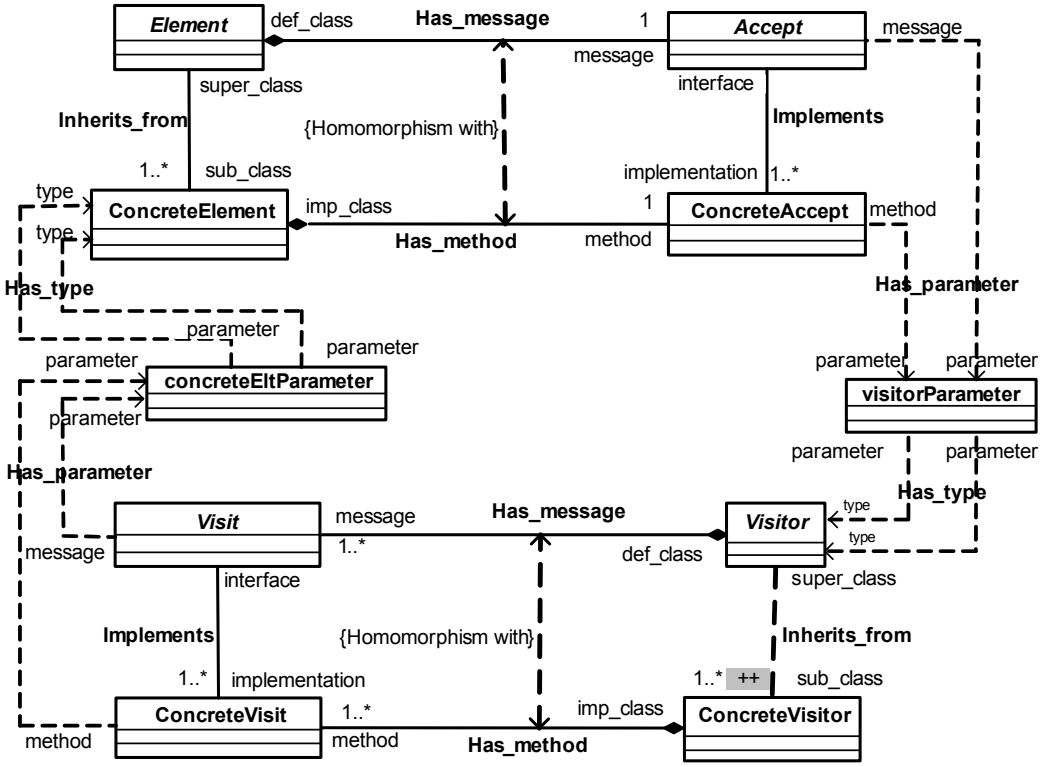


Figure 3.16. Représentation de la solution proposée par le patron « Visiteur »

Pour ce qui est des attributs, ils sont représentés par la classe ***ModelAttribute*** qui est une sous-classe de la classe `EAttribute` de EMF. De façon similaire à ***ModelOperation***, la classe ***ModelAttribute*** définit une variable d'instance appelée « ***literal*** » qui permet de spécifier si l'attribut est *littéral* ou non (c'est un *paramètre*). Nous avons aussi étendu la classe `EReference` par la classe ***ModelReference*** pour avoir notre propre modèle d'associations et cela pour plusieurs raisons. D'abord, nous avons besoin de représenter divers types de relations entre une famille de classes, d'une part, et une classe ou une autre famille de classes, d'autre part, au niveau méta-modèle. Ensuite, nous avons besoin de représenter aussi l'évolutivité de la cardinalité de la référence, ce que nous avons noté «++» dans les modèles de problèmes et de solutions. L'éventuelle évolution de la cardinalité d'une référence est spécifiée par la variable d'instance ***enableExtension*** qui est de type

booléen. Quand ***enableExtension*** est égal à *true*, cela signifie que cette référence instance de ***ModelReference*** pointe sur une famille de classes dont le nombre va probablement augmenter.

Dans le cas particulier d'une relation d'héritage entre une famille de classes et une classe, nous avons étendu la classe ***ModelReference*** par une autre classe ***InheritsFromReference*** pour pouvoir représenter cette catégorie d'héritage au niveau méta-modèle. De la même façon, nous avons introduit la sous-classe ***ImplementsReference*** pour représenter la relation `Implements` existante entre une famille de classes et une interface. Dans le modèle de solution du patron Visiteur (Figure 3.16), par exemple, la classe *`ConcreteVisitor`* a une relation *`Inherits_from`*, avec la classe *`Visitor`*, dont la cardinalité est « 1..*,++ ». Cette relation est, en fait, une instance de la classe ***InheritsFromReference*** dont l'attribut ***enableExtension*** est égal à *true* car le nombre de classes de type *`ConcreteVisitor`* est susceptible d'augmenter.

Pour implémenter notre méta-modèle, nous avons choisi d'utiliser des interfaces Java annotées. Nous rappelons au lecteur qu'il y a plusieurs façons de créer un modèle EMF (voir section 3.6.1.1). Nous avons créé pour chaque classe du méta-modèle une interface annotée en Java. Ces interfaces sont données en entrée au générateur EMF pour générer un modèle EMF et, par la suite, les classes d'implémentation de notre méta-modèle. Nous décrivons les détails de cette implémentation à l'annexe 3. Notre méta-modèle, sérialisé sous format XMI, est donné à l'annexe 4.

3.6.3 Exemple d'instanciation d'un modèle de problème

Nous avons utilisé notre méta-modèle pour instancier différents modèles de problèmes et de solutions. Des extraits du code qui permet de créer le modèle du problème du patron Visiteur sont présentés à la Figure 3.17. Pour créer une instance de notre méta-modèle, il faut enregistrer l'espace de nom lui correspondant. Nous n'entrerons pas dans ces détails. Une fois l'enregistrement fait, nous créons une instance de la classe de fabrication ***PatternMetamodelFactoryImpl*** associée à notre

méta-modèle (ligne 1) et qui a été générée par le générateur EMF (voir Annexe 3). Nous utilisons cette instance pour créer un modèle vide et ensuite tous les éléments que le modèle doit contenir. Ainsi, nous créons un modèle dont le nom et l'espace de nom sont nommés *VisitorProblem* (lignes 2, 3, 4 et 5). Ensuite, nous créons et initialisons une instance de ***ParameterClass*** correspondant à la classe *AbstractClass* (lignes 6, 7 et 8) du modèle du problème, et une instance de ***ModelOperation*** nommée *AbstractOperation* (lignes 9, 10, 11 et 12). Cette dernière est ajoutée à la liste des opérations de la classe *AbstractClass* (ligne 13).

```
    //Créer une instance de la classe de fabrication
1.  PatternMetamodelFactory modelFactory = patternMetamodelFactory.eINSTANCE;
    //Créer un modèle dont le nom est VisitorProblem
2.  ModelPackage visitorProblem = modelFactory.createModelPackage();
3.  visitorProblem.setName("VisitorProblem");
4.  visitorProblem.setNsPrefix("VisitorProblem");
5.  visitorProblem.setNsURI("http://VisitorProblem");
    //Créer la classe AbstractClass
6.  ParameterClass abstract_class = modelFactory.createParameterClass();
7.  abstract_class.setName("AbstractClass");
8.  abstract_class.setAbstract(true);
    //Créer l'opération AbstractOperation
9.  ModelOperation abstract_operation = modelFactory.createModelOperation();
10. abstract_operation.setName("AbstractOperation");
11. abstract_operation.setAbstract(true);
12. abstract_operation.setEnableExtension(true);
    //Ajouter AbstractOperation aux opérations de la classe AbstractClass
13. abstract_class.getClassOperations().add(abstract_operation);
…
    //Créer la relation Inherits_from entre ConcreteClass et AbstractClass
14. InheritsFromReference inheritsRef = modelFactory.createInheritsFromReference();
15. inheritsRef.setName("Inherits_from");
16. inheritsRef.setEType(abstract_class);
17. inheritsRef.setLowerBound(1);
18. inheritsRef.setUpperBound(-1);
    //Ajouter Inherits_from aux références de la classe ConcreteClass
19. concrete_class.getClassReferences().add(inheritsRef);
…
    //Ajouter toutes les classes créées au paquetage visitorProblem
20. visitorProblem.getModelClasses().add(abstract_class);
21. visitorProblem.getModelClasses().add(concrete_class);
    //Sauvegarder le paquetage visitorProblem
22. Resource outputResource = (new ResourceSetImpl()).createResource(
       URI.createURI(visitorProblem.getName() + ".ecore"));
23. outputResource.getContents().add(visitorProblem);
24. try {
25.    outputResource.save(null);
26. } catch (Exception e) {
27.    e.printStackTrace();
28. }
```

Figure 3.17. Extraits du code permettant de générer le modèle du problème associé au patron Visiteur.

Nous rappelons au lecteur que nous avons choisi de définir notre propre relation d'héritage ***InheritsFromReference*** et cela pour pouvoir lui associer une multiplicité et aussi pour spécifier la notion d'évolutivité. Dans le modèle du problème du Visiteur (Figure 3.3), par exemple, la classe *ConcreteClass* hérite de la classe *AbstractClass*. Cette relation d'héritage a une multiplicité « 1..* » qui indique qu'une ou plusieurs classes de type *ConcreteClass* peuvent hériter de la classe *AbstractClass*. Cette relation est créée par le code montré à la Figure 3.17. Cette relation d'héritage est définie comme une instance de ***InheritsFromReference*** dont le nom est *Inherits_from* (lignes 14 et 15). Ses attributs `lowerBound` et `upperBound` sont initialisés (lignes 17 et 18) pour spécifier la cardinalité « 1..* ». La méthode `setEType` (ligne 16) est utilisée pour spécifier que le type de cette référence est la classe *AbstractClass* qui est la super-classe de toutes les classes de type *ConcreteClass*. Finalement, cette référence est ajoutée à la liste des références de *ConcreteClass* (ligne 19). Si la cardinalité de cette relation avait été « 1..*, ++ » comme cela est le cas dans le modèle du problème du patron Pont où le nombre de sous-classes d'une classe est susceptible d'augmenter (voir annexe 1), son attribut ***enableExtension*** aurait été initialisé à *true*.

Une fois toutes les entités de notre modèle du problème créées, nous les ajoutons à notre modèle *VisitorProblem* (lignes 20 et 21). Pour pouvoir sérialiser ce modèle, nous l'ajoutons à une ressource qui se chargera de la sérialisation (lignes 22 à 28). La ressource a le même non que le paquetage *VisitorProblem* et l'extension « `.ecore` ». En fait, l'API de persistance d'EMF nous permet de sérialiser les instances de notre méta-modèle (i.e. des modèles de problèmes ou de solutions) sous format XMI. Cette API est décrite brièvement à l'annexe 5.

Nous avons procédé de la même façon pour instancier le modèle de la solution du patron Visiteur ainsi que les modèles de problèmes et de solutions d'autres patrons (e.g. Pont, Composite, Stratégie). La sérialisation sous format XMI du modèle de problème du patron Visiteur est montrée à la Figure 3.18.

```
<?xml version="1.0" encoding="UTF-8"?>
<org.eclipse.PatternMetamodel:ModelPackage xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xmlns:org.eclipse.PatternMetamodel="http:///org/eclipse/
   PatternMetamodel.ecore" name="VisitorProblem" nsURI="http://VisitorProblem"
   nsPrefix="VisitorProblem">
   <modelClasses xsi:type="org.eclipse.PatternMetamodel:ParameterClass"
      name="AbstractClass" abstract="true">
      <classOperations name="AbstractOperation" abstract="true"
         enableExtension="true"/>
   </modelClasses>
   <modelClasses xsi:type="org.eclipse.PatternMetamodel:ParameterClass"
      name="ConcreteClass">
      <classOperations name="ConcreteOperation"
         implements="VisitorProblem.ecore#//AbstractClass/AbstractOperation"
         overrides="VisitorProblem.ecore#//ConcreteClass/ConcreteOperation"
         enableExtension="true"/>
      <classReferences xsi:type="org.eclipse.PatternMetamodel:InheritsFromReference"
         name="Inherits_from" lowerBound="1" upperBound="-1"
         eType="org.eclipse.PatternMetamodel:ParameterClass VisitorProblem.ecore#//
         AbstractClass"/>
      <classReferences xsi:type="org.eclipse.PatternMetamodel:InheritsFromReference"
         name="Inherits_from" lowerBound="1" upperBound="-1"
         eType="org.eclipse.PatternMetamodel:ParameterClass VisitorProblem.ecore#//
         ConcreteClass"/>
   </modelClasses>
</org.eclipse.PatternMetamodel:ModelPackage>
```

Figure 3.18. Modèle du problème du patron Visiteur sous format XMI.

3.7 Conclusion

Nous avons introduit et décrit, dans ce chapitre, notre approche pour la représentation et la mise en œuvre des patrons. Nous avons essayé de représenter d'une façon précise les problèmes de conception résolus par les patrons. Cette représentation nous permet de mieux comprendre les patrons, de les spécifier d'une façon plus complète, d'évaluer leur opportunité d'utilisation (étape décrite au chapitre 4) et de supporter leur application d'une façon automatique (voir chapitre 5). En plus d'être générique, notre approche permet de promouvoir la réutilisation des patrons.

Pour ce qui est de l'implémentation, nous avons construit un méta-modèle pour la représentation des modèles de problèmes et de solutions reliés aux patrons. Nous avons implanté ce méta-modèle au sein de l'environnement de développement Eclipse. Nous l'avons utilisé pour instancier plusieurs exemples de modèles de problèmes et de solutions que nous avons sérialisés sous format XMI.

Chapitre 4

Le processus de marquage

4.1 Vue globale de notre approche pour le marquage

Le processus de marquage est le processus qui permet de reconnaître des instances des modèles de problèmes dans un modèle donné en entrée par l'utilisateur. Comme démontré par l'exemple du patron Visiteur décrit dans le chapitre précédent, un modèle de problème est composé d'une partie statique et des points de variation temporels (*time hotspots*). Le modèle statique de problème peut être comparé statiquement au modèle reçu en entrée, par contre l'information concernant les points de variation n'est pas représentée dans les modèles en entrée. Pour cela, nous avons divisé le processus de marquage en trois étapes (Figure 4.1) :

1. **Correspondance statique au modèle de problème** (***static pattern matching***) : dans cette étape, nous essayons de reconnaître dans un modèle reçu en entrée des configurations ou structures qui sont conformes à des modèles statiques de problèmes. Ce problème est fondamentalement un problème de correspondance de graphes (*graph pattern matching*) puisque les modèles sont des graphes. La résolution de ce problème se fait par le biais de techniques de *pattern matching*. En particulier, nous utilisons les techniques de satisfaction de contraintes pour résoudre ce problème. Notre approche est expliquée en détails à la section 4.5. À la fin de cette étape, nous indiquons au concepteur toutes les structures—fragments faisant partie du modèle en entrée—qui sont conformes à des modèles statiques de problèmes.

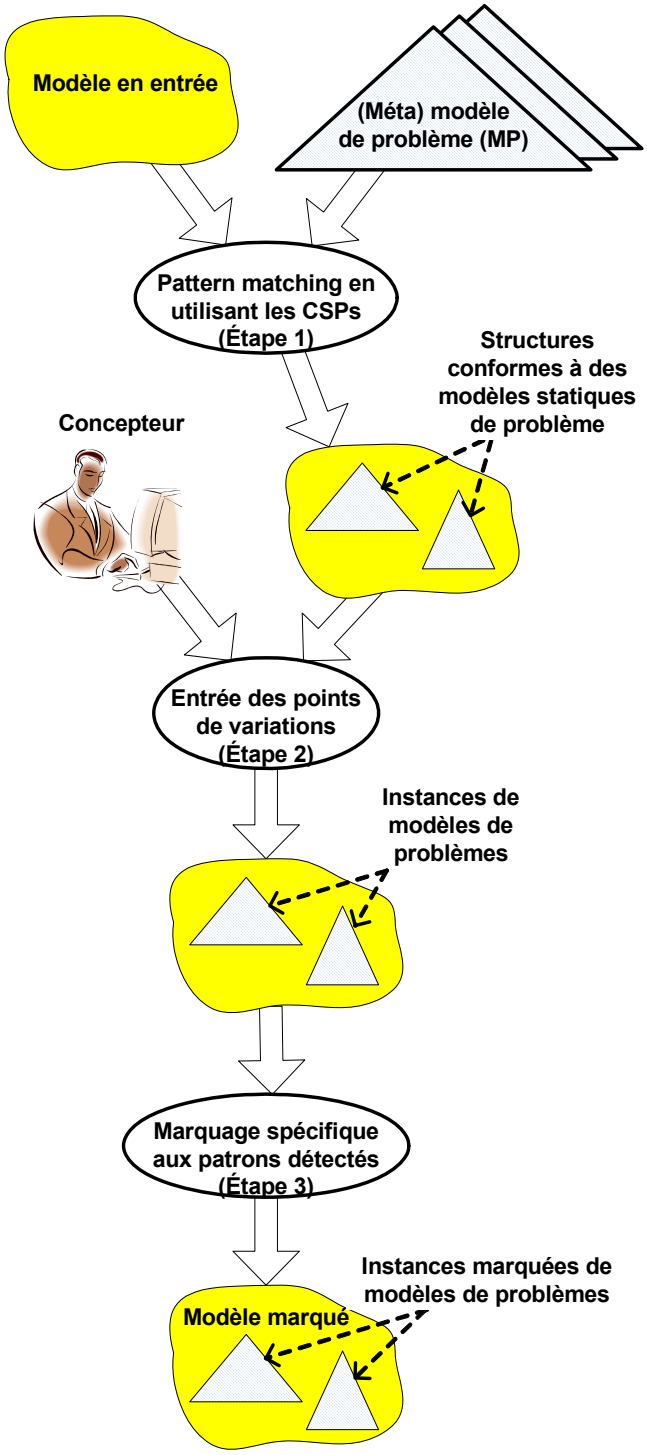


Figure 4.1 Le processus de marquage

2. **Entrée des points de variation** : La conformité d'un fragment du modèle en entrée à un modèle statique de problème ne signifie pas que le patron de conception s'applique à ce fragment. En effet, certains patrons de conception

ont le même modèle statique de problème, et ne diffèrent que par les points de variation. Par exemple, les problèmes résolus par les patrons Visiteur, Pont et Stratégie sont représentés par le même modèle statique: une hiérarchie de classes ! Ainsi, lorsqu'une structure (i.e. un fragment du modèle en entrée) est conforme à la partie statique de plus d'un modèle de problème, nous avons besoin d'identifier les éventuels points de variation présents dans le modèle en entrée. Si nous disposons de différentes versions historiques du modèle décrivant le système considéré, nous pourrons comparer les modèles de classes des différentes versions pour déduire les parties qui ont changé d'une version à l'autre (voir par exemple (Sahraoui et al., 2000)). Par contre, pour un nouveau système ou un système dont les données historiques ne sont pas disponibles, nous devons demander au concepteur d'identifier les éventuels points de variation (e.g. le nombre de sous-classes d'une classe donnée, le nombre de méthodes, etc.). Bien sûr, la précision de ces points de variation dépend du niveau d'expertise du concepteur et de son expérience dans le domaine du problème en cours.

3. **Marquage complet des instances de modèles de problème**: Dans cette étape, nous utilisons les points de variation fournis par le concepteur pour compléter la correspondance—statique de problème—établie dans la première étape du processus de marquage. Cela nous permet de confirmer ou non la conformité d'un fragment du modèle en entrée au modèle complet du problème d'un patron de conception. Une fois un tel fragment est trouvé, nous marquons ses éléments (e.g. classes, méthodes, attributs, associations, relations d'héritage, etc.) avec les rôles auxquels ils correspondent dans le modèle du problème du patron en question. Par exemple, si un fragment est conforme au modèle du problème associé au patron Visiteur, une classe de ce fragment sera annotée par le rôle `AbstractClass`, ses sous-classes seront annotées par le rôle `ConcreteClass`, ses méthodes abstraites seront annotées par le rôle `AbstractOperation`, etc. Vu que différents modèles de problèmes peuvent référer aux mêmes rôles (e.g. la notion de `AbstractClass` apparaît dans plusieurs modèles de problèmes), nous avons besoin de qualifier les rôles

par le patron auquel ils réfèrent. Ainsi, la même classe abstraite dans le modèle en entrée pourrait être marquée par l'annotation `Visitor.AbstractClass` et `Strategy.AbstractClass`. De plus, parce que le même modèle en entrée peut inclure différentes instances du même modèle de problème, les marques sont aussi qualifiées par un numéro d'instance du patron relié. Ainsi, une classe concrète peut avoir une marque `Strategy.1.ConcreteClass`, une marque `Strategy.2.ConcreteClass` et une marque `Visitor.1.ConcreteClass`. Ces marques sont utilisées par nos règles de transformation pour éviter que différentes instances des patrons de conception n'interfèrent les unes avec les autres durant la phase de transformation.

Dans ce chapitre, nous nous intéressons plus particulièrement à la première étape du processus de marquage à savoir, comment trouver dans un modèle reçu en entrée, des fragments qui correspondent à un modèle de problème. En pratique, les modèles sont des graphes et le problème de correspondance entre modèles peut être considéré comme un problème de correspondance de patrons de graphes (*graph pattern matching*). Il a été démontré dans la littérature que ce problème—connu aussi comme étant le problème d'homomorphisme de graphes—est NP-complet (Mehlhorn, 84).

Plusieurs approches de transformations de modèles se sont intéressées au problème de *pattern matching*. Dans la majorité de ces approches, l'algorithme de recherche de *pattern matching* dépend du graphe/modèle recherché ce qui ne favorise pas sa réutilisation. Cependant, l'algorithme de recherche peut être dissocié du graphe considéré en traduisant le problème de *pattern matching* en un problème de satisfaction de contraintes (Rudolf, 98). Pour cette raison, et aussi pour bénéficier des résultats de la recherche dans le domaine des techniques de satisfaction de contraintes, nous avons appliqué cette stratégie à notre problème de détection d'instances des modèles de problèmes.

Dans la suite du chapitre, nous allons définir, de façon simplifiée, les concepts de graphe, d'homomorphisme de graphes et de transformation de graphe. Nous

survolerons quelques approches qui se sont intéressées à la détection de patrons ou motifs dans les systèmes orientés objet. Nous présenterons brièvement des notions de base de la théorie des problèmes de satisfaction de contraintes puis nous introduirons l'approche proposée par Rudolf (98) pour résoudre le problème de *pattern matching* en utilisant les problèmes de satisfaction de contraintes. Nous présenterons à la section 4.5 notre approche pour la détection des instances de modèles de problèmes résolus par les patrons de conception dans des modèles UML. Nous illustrerons notre approche par l'exemple du patron Visiteur. Finalement, nous décrirons notre implémentation à la section 4.6 avant de conclure dans la section 4.7.

4.2 Concepts de base

4.2.1 Graphe

Un graphe G est généralement défini par un ensemble de nœuds G_V et un ensemble d'arêtes G_E telles que chaque arête appartenant à G_E relie deux nœuds appartenant à G_V. Plus formellement, un graphe G est défini selon l'approche algébrique (Corradini et al., 96) (Rudolf, 98) par un *tuple* (G_V, G_E, L, s, t, l) où

- G_V est un ensemble fini de nœuds, G_E est un ensemble fini d'arêtes, et $G_V \cap G_E = \varnothing$,
- s, t : $G_E \rightarrow G_V$ sont des fonctions qui associent respectivement des nœuds sources et des nœuds cibles aux arêtes,
- l : $G_V \cup G_E \rightarrow L$ est une fonction associant à un nœud ou une arête un type ou libellé. L est un ensemble de libellés/types.

Les graphes offrent un formalisme mathématique très riche pour la modélisation. En orienté objet, les graphes de modélisation (i.e. modèles) se situent à deux niveaux: le niveau de type et le niveau d'instance (Baresi et Heckel, 2000). Un graphe de niveau instance, est un graphe typé, i.e. il est décrit de façon abstraite par un graphe de types appelé aussi graphe de schéma (Engels et al., 1997). Il existe un homomorphisme entre un graphe typé (e.g. diagramme d'objets) et son graphe de

schéma (diagramme de classes). Un modèle est un graphe typé et attribué puisqu'un élément (nœud ou arête) du modèle a un type et peut avoir des attributs.

4.2.2 Homomorphisme de graphes

Un homomorphisme de graphes est une correspondance (*mapping*) entre deux graphes qui préserve la structure et le type. Plus formellement et en se basant sur la définition donnée dans (Rudolf, 98), un homomorphisme d'un graphe $H = (H_V, H_E, H_L, s_H, t_H, l_H)$ vers un graphe $G = (G_V, G_E, L_G, s_G, t_G, l_G)$ peut être défini par deux correspondances (*mappings*) $M_V : H_V \rightarrow G_V$ et $M_E : H_E \rightarrow G_E$ telles que :

- $\forall v \in H_V$, $l_H(v) = l_G(M_V(v))$, i.e. un nœud v de H_V et son image dans G_V par le mapping M_V ont le même type.
- $\forall e \in H_E$, $l_H(e) = l_G(M_E(e))$, i.e. une arête e de H_E et son image dans G_E par le mapping M_E ont le même type.
- $\forall e \in H_E$, $M_V(s_H(e)) = s_G(M_E(e))$ et $M_V(t_H(e)) = t_G(M_E(e))$, i.e. l'image par le mapping M_V de la source (respectivement la cible) d'une arête e est égale à la source (respectivement la cible) de l'image par le mapping M_E de l'arête e.

Pour illustrer la notion d'homomorphisme de graphes, considérons les graphes H et G de la Figure 4.2. Pour simplifier, les libellées des nœuds et des arêtes définissent leurs types. En appliquant la définition précédente d'homomorphisme, il existe une seule instance valide d'homomorphisme entre H et G. Autrement dit, il y a une seule instance valide de H dans G (représentée en gras dans G).

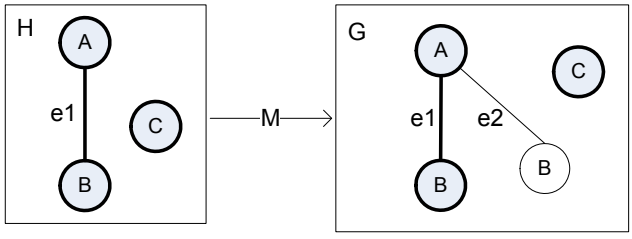


Figure 4.2. Un exemple d'homomorphisme de graphes

4.2.3 Transformation de graphe

La transformation d'un graphe est décrite par un ensemble de règles. Une règle de transformation de graphe—appelée aussi production—est définie par une pré-condition et une post-condition. La pré-condition est en fait composée du graphe source et potentiellement de conditions négatives d'application (NAC : *negative application condition*). Les conditions négatives d'application interdisent, si elles sont vérifiées, l'application de la règle en question. La post-condition correspond au graphe cible de la règle.

Selon Corradini et al. (96), une production P : L → R définit une correspondance partielle entre les éléments du graphe source L et ceux du graphe cible R (Figure 4.3). Cette correspondance détermine quels nœuds et arcs sont préservés par l'application de P, lesquels doivent être supprimés et lesquels doivent être créés. Ainsi, le graphe H (Figure 4.3) dérivé du graphe G par application de la production P, est construit par la formule G\(L\R)∪(R\L). Autrement dit, le graphe H est construit en supprimant la partie du graphe G qui existe dans L mais pas dans R (la partie L\R) et en ajoutant de nouveaux éléments correspondant à des éléments de R qui n'existaient pas dans L (la partie R\L). m et m' sont des homomorphismes de graphes. Notons que cette définition est basée sur l'approche algébrique SPO (*single-pushout*) (Corradini et al., 96).

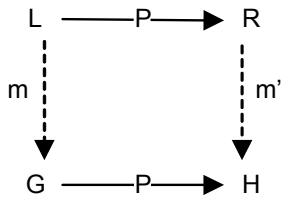


Figure 4.3. Représentation schématique d'une production

La Figure 4.4 montre un exemple de règle et de son application. L'application d'une règle P à un graphe G (ou modèle) se déroule ainsi en trois étapes: (1) trouver les correspondances de la partie gauche L de la production P dans G, (2) vérifier les NAC qui interdisent la présence de certains objets et liens dans G, et (3) remplacer

dans G la partie LHS (i.e. L) par la partie RHS (i.e. R) correspondante pour obtenir le graphe transformé H.

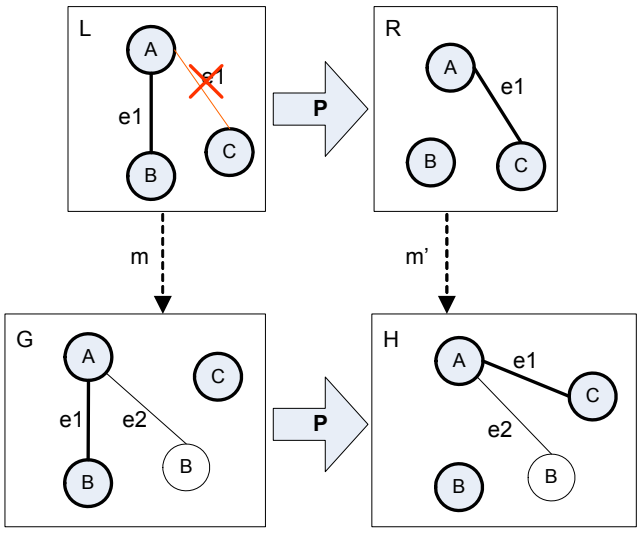


Figure 4.4. Exemple d'une règle et de son application à un graphe

4.3 Approches pour la détection de patrons dans les systèmes orientés objet

Notre problème de détection des instances de modèles de problèmes dans des modèles donnés en entrée, est relié, en général, au problème de détection de patrons dans les systèmes orientés objet et, en particulier, au problème de *pattern matching* dans les approches transformationnelles dirigées par les modèles.

4.3.1 Détection de patrons dans le code source

Plusieurs approches se sont intéressées au problème de détection de patrons ou motifs dans les systèmes orientés objet. Ces approches peuvent être classées par les patrons ciblés ou les techniques utilisées. En effet, ces approches visent à détecter soit des problèmes de conception (Alikacem et Sahraoui, 2006) (Ciupke, 1999), soit des micro-patrons appelés aussi des patrons élémentaires de conception (Arcelli et al., 2005) (Gil et Maman, 2005) (Smith et Stotts, 2003), ou des instances des patrons de conception (Wuyts, 1998) (Prechelt et Kramer, 1998) (Guéhéneuc et Jussien, 2001). Alors que l'approche de (Alikacem et Sahraoui, 2006) utilise des métriques pour

évaluer la qualité du code source et en déduire ainsi les problèmes de conception, l'approche de (Ciupke, 1999) utilise la programmation logique. Dans les deux cas, les problèmes de conception sont définis comme des violations d'heuristiques orientées objet. Ils sont donc d'un niveau d'abstraction plus bas que les problèmes de conception résolus par les patrons de conception.

L'approche dans (Smith et Stotts, 2003) propose un outil qui utilise le calcul Sigma (σ-calculus) pour représenter et détecter des patrons élémentaires de conception. Étant plus petits et moins complexes, ces patrons élémentaires (e.g. abstraction d'interface, délégation d'implémentation) sont plus faciles à détecter que les patrons de conception. Des règles sont utilisées pour composer les patrons élémentaires et construire des patrons plus larges et plus complexes. Cependant, très peu de concepteurs ont les connaissances nécessaires pour utiliser cet outil. Arcelli et al. (2005) proposent une autre méthode pour détecter les patrons élémentaires de conception dans le code Java. Ils exploitent les particularités du langage Java dans la définition de ces patrons et utilisent des fonctions logiques pour les détecter. Les patrons élémentaires sont plus faciles à automatiser mais ne fournissent pas autant d'information sur la conception que les patrons de conception (Gil et Maman, 2005).

Les approches dans (Wuyts, 1998) et (Prechelt et Kramer, 1998) utilisent la programmation logique pour spécifier et détecter les patrons de conception dans le code source alors que l'approche (Guéhéneuc et Jussien, 2001) utilise la programmation par contraintes combinée avec un système d'explications pour détecter des instances approximatives de patrons de conception. Notre approche (décrite à la section 4.5) est similaire à celle de (Guéhéneuc et Jussien, 2001) dans la mesure où elle utilise la programmation par contraintes. Cependant, au niveau de l'implémentation, nous utilisons de façon différente les problèmes de satisfaction de contraintes. En effet, dans (Guéhéneuc et Jussien, 2001), les relations telles que la composition et l'héritage sont considérées comme des contraintes sur les classes extraites du code source. Cela nécessite le traitement des instances détectées, par exemple, il peut y avoir plus d'une relation de composition entre deux classes et nous avons besoin de les retourner toutes.

Au meilleur de notre connaissance, toutes ces approches se sont penchées sur la détection dans le code source d'instances complètes ou approximatives des solutions proposées par les patrons de conception et non des problèmes qu'ils résolvent.

4.3.2 Pattern matching dans le contexte des approches dirigées par les modèles

Le problème d'homomorphisme de graphes est un problème commun dans les approches de transformations de modèles basées sur les transformations de graphes. En effet, pour appliquer une règle à un graphe donné G (Figure 4.4), il faut d'abord trouver une instance de L dans G et puis la remplacer par une instance de R. Alors que l'opération de remplacement peut être faite en un temps linéaire, l'opération de recherche d'un sous-graphe de G qui correspond à L, est dans le pire des cas, exponentielle en fonction de la taille de L (nombre de nœuds et d'arêtes). Ainsi, la phase de *graph pattern matching* est la phase la plus critique dans le processus d'application d'une transformation de graphe puisque la qualité de l'application dépend principalement de la qualité du *pattern matching*.

Beaucoup de travaux se sont donc intéressés au problème de *graph pattern matching* pour en réduire la complexité moyenne. Hormis quelques approches qui proposent des implémentations très spécifiques (e.g. (Kalnins et al., 2006)), on peut identifier deux orientations principales dans les stratégies de *pattern matching*. Dans la première orientation, les approches basées sur les transformations de graphes utilisent des algorithmes complexes basés sur des plans de recherche (Zündorf, 1994) permettant de traverser la partie LHS d'une règle (e.g. GReAT (Agrawal et al, 2005), FUJABA (Grunske et al., 2005), PROGRES (Zündorf, 1994) et VIATRA (Balogh et Varró, 2006)). Dans la deuxième orientation, les approches utilisent les techniques de satisfaction de contraintes (e.g. AGG (Rudolf et Taentzer, 1999)).

De façon informelle, un plan de recherche définit une séquence de nœuds du patron recherché. Cette séquence peut être utilisée durant le processus de *pattern matching* pour contrôler l'ordre de parcours des objets dans le modèle source (Varro,

2008). Pour réduire la complexité de la recherche de correspondances, on exploite généralement les propriétés du graphe/modèle dont on cherche une instance pour développer un algorithme adéquat. Par exemple, FUJABA, PROGRES et GReAT utilisent les contraintes de cardinalités et de types imposées par le graphe de type/méta-modèle. Cependant, cette démarche ne permet pas la réutilisation de l'algorithme de recherche en passant d'un modèle à un autre.

Des approches plus adaptatives (e.g. (Varro et al., 2006) ou génériques (e.g. (Horvath et al., 2007)) ont été alors proposées. Varro et al.(2006) proposent une approche où un plan de recherche est sélectionné à l'exécution à partir de plans précédemment générés et cela en se basant sur des données statistiques issues du modèle courant. Le problème avec cette approche est qu'elle se base sur une forte hypothèse qui est l'existence de modèles prédéfinis typiques du domaine. L'approche suppose aussi que toutes les associations entre les nœuds du modèle sont de type un-à-un et que toutes les extrémités des associations sont navigables. Horvath et al.(2007) proposent une représentation qui permet de comparer différents plans de recherche en plaçant à différentes positions une opération complexe de recherche telle que la vérification des NAC d'une règle, et de choisir le plan ayant le coût minimal. Le problème avec cette approche est qu'il y a une surcharge due au calcul des coûts de tous les plans de recherche possibles. Finalement, dans les deux approches, la représentation des plans de recherche est complexe et non intuitive.

D'une façon ou d'une autre, les stratégies de *pattern matching* sont basées sur des algorithmes de *backtracking*. Rudolf (98) argumente que la majorité des travaux entrepris pour optimiser les algorithmes de *backtracking* ont été faits dans le contexte des problèmes de satisfaction de contraintes. Ainsi pour bénéficier des résultats de la recherche dans ce domaine, Rudolf a proposé de traduire le problème d'homomorphisme de graphes en un problème de satisfaction de contraintes. Contrairement aux autres approches utilisant les plans de recherche, l'utilisation des problèmes de satisfaction de contraintes permet de dissocier l'algorithme de recherche de motifs, de la structure du motif considéré. De plus, les outils permettant de résoudre les problèmes de satisfaction offrent des mécanismes permettant de

guider le processus de recherche des solutions de façon à l'optimiser. Nous avons donc choisi d'appliquer cette stratégie à notre problème de détection d'instances des modèles de problèmes.

4.4 Utilisation des problèmes de satisfaction de contraintes pour le pattern matching

Le problème d'homomorphisme de graphes/modèles peut être reformulé et résolu comme un problème de satisfaction de contraintes (Rudolf, 98). Nous présentons d'abord quelques notions de base des problèmes de satisfaction de contraintes. Ensuite nous présenterons l'approche proposée par Rudolf (98) pour résoudre le problème d'homomorphisme de graphes en utilisant les problèmes de satisfaction de contraintes.

4.4.1 Concepts de base des problèmes de satisfaction de contraintes

Un problème de satisfaction de contraintes (CSP : *Constraint Satisfaction Problem*) consiste en un ensemble de variables ayant chacune un domaine fini, et un ensemble de contraintes sur les valeurs que les variables peuvent avoir simultanément (Tsang, 93). Il s'agit de trouver un ensemble de valeurs qui peuvent être affectées aux différentes variables tout en satisfaisant toutes les contraintes. Formellement, un problème CSP est décrit par (Freuder, 78) (Tsang, 93) (Bacchus et van Beek, 98) (Rudolf, 98) :

- Un ensemble fini de variables $V = \{v_1,\ldots,v_n\}$.
- Leurs domaines respectifs. Le domaine d'une variable est composé d'un ensemble fini de toutes les valeurs qui peuvent être affectées à cette variable. Soit D_i le domaine de la variable v_i.
- Un ensemble R de contraintes qui est un ensemble fini de restrictions sur les valeurs que peuvent prendre les variables (Barták, 2003).

Une contrainte C appartenant à R est définie sur un sous-ensemble ou un *tuple* de variables $(v_1,\ldots v_r)$. Elle restreint les valeurs pouvant être affectées à ces variables. La contrainte C peut être considérée comme une relation du *tuple* $(v_1,\ldots v_r)$ vers le

produit cartésien des domaines respectifs de ces variables, i.e. $C(v_1,\ldots v_r) \subseteq D_1 \times \ldots \times D_r$. La taille du *tuple* $(v_1,\ldots v_r)$ est appelée l'*arité* de la contrainte C. Une contrainte C sur un *tuple* $(v_1,\ldots v_r)$ est satisfaite par un *tuple* de valeurs $(d_1,\ldots d_r)$ si et seulement si :

- $\forall(1\leq i \leq r)\ d_i \in D_i$ (le domaine de v_i) et,
- $(d_1,\ldots d_r) \in C(v_1,\ldots v_r)$

La majorité des travaux de recherche dans la littérature se sont concentrés sur des problèmes avec des contraintes binaires. En effet, il a été démontré qu'on pouvait traduire n'importe quel problème CSP discret dont les contraintes sont d'*arité* n ($n \geq 3$) en un problème CSP équivalent avec des contraintes unaires et binaires (Rossi et al., 90) (Bacchus et van Beek, 98). Une solution $(d_1,\ldots d_n)$ à un problème CSP—si elle existe—affecte une valeur d_i de son domaine à chaque variable v_i de façon que toutes le contraintes du problème soient satisfaites (Barták, 99).

Pour illustrer ces concepts, considérons le problème des N-Reines qui est souvent utilisé pour illustrer les problèmes de satisfaction de contraintes et la programmation par contraintes. Le problème des N-Reines consiste à placer N reines sur un échiquier de N*N cases de façon telle qu'aucune reine n'en menace une autre. Selon les règles des échecs, une reine menace n'importe quelle autre pièce qui se trouve dans la même rangée, la même colonne ou la même diagonale. La Figure 4.5 montre une solution à ce problème dans le cas où N est égal à 8.

Une formalisation possible de ce problème en un problème CSP utilise N variables $\{v_1,\ldots v_n\}$ pour représenter les rangées de l'échiquier (Tsang, 93). Si nous libellons les colonnes de 1 à N, chaque variable peut avoir une valeur de 1 à N correspondant à la colonne (position) sur laquelle la reine est placée sur cette rangée. Ainsi $D_{v1} = D_{v2} = \ldots = D_{vN} = \{1,2,\ldots,N\}$. Pour assurer qu'aucune paire de reines n'est sur la même colonne et qu'aucune paire de reines n'est sur la même diagonale, on utilise les contraintes suivantes :

- $\forall\ i \neq j,\ v_i \neq v_j$ et,
- $\forall i \neq j$, si $v_i = a$ et $v_j = b$ alors $|i - j| \neq |a - b|$.

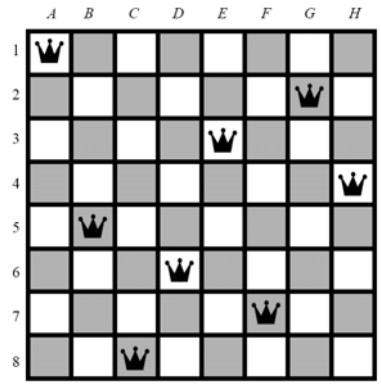


Figure 4.5. Une solution possible au problème des 8-Reines (Tsang, 93)

Selon cette formalisation du problème des N-Reines, il y a N^N combinaisons possibles à considérer pendant la recherche de la solution (e.g. 8^8 dans le cas des 8-Reines). Par rapport à d'autres formalisations du problème, elle réduit la taille/difficulté du problème car sa définition des variables intègre déjà les contraintes sur les rangées (Tsang, 93).

Les problèmes CSPs sont généralement résolus en utilisant une recherche exhaustive basée sur le *backtracking*. « Dans le *backtracking*, les variables sont instanciées de façon séquentielle. Dès que toutes les variables associées à une contrainte sont instanciées, la validité de la contrainte est vérifiée. Si une instanciation partielle viole une contrainte, l'algorithme revient sur ses pas (exécute le *backtracking*) jusqu'à la variable la plus récemment instanciée et qui possède encore des valeurs alternatives. Le *backtracking* permet donc d'éliminer un sous-espace du produit cartésien de tous les domaines des variables chaque fois qu'une instanciation partielle viole une contrainte. » (Kumar, 92).

L'algorithme de *backtracking* présente donc une nette amélioration par rapport à l'algorithme naïf qui consiste à générer toutes les combinaisons possibles de valeurs et les tester. Cependant, pour la majorité des problèmes intéressants, le backtracking ne réduit pas la complexité exponentielle du problème. Dans la littérature, on recense plusieurs problèmes reliés à l'algorithme standard de backtracking notamment le *thrashing* qui fait référence à des situations où la recherche échoue dans différentes parties de l'espace de solution et cela pour la même

raison (Barták, 99). L'exemple le plus simple du thrashing, qu'on appelle *incohérence de nœud* (*node inconsistency*) se présente quand une valeur d_i dans le domaine d'une variable v_i ne satisfait pas une contrainte unaire sur cette variable : chaque fois qu'on affecte d_i à v_i il y a échec. Ce type d'incohérence peut être réglé en éliminant tout simplement les valeurs qui ne satisfont pas les contraintes unaires du domaine de la variable concernée (Kumar, 92). Un autre type de thrashing, appelé *incohérence d'arc* (arc fait référence à une contrainte entre deux variables) se présente quand des valeurs affectées à deux variables (v_i, v_j) sont conflictuelles, i.e. violent une contrainte. En effet, l'algorithme standard de backtracking ne se souvient pas de ces valeurs conflictuelles et les réessaye à chaque fois qu'il repasse par ses deux variables. De plus, le conflit entre des valeurs n'est détecté que lorsqu'il a lieu réellement (Barták, 99).

Plusieurs approches, regroupées sous le nom de *backtracking* intelligent, ont été proposées pour résoudre les différents types de *thrashing*. La méthode du *backjumping*, par exemple, propose lors d'un échec durant l'instanciation d'une variable v_y de déterminer la variable v_x responsable de l'échec (i.e. variable impliquée dans la contrainte que toutes les valeurs du domaine de v_y violent) et de faire un saut en arrière directement à la variable v_x sans repasser par les variables intermédiaires (Rudolf, 98). Une autre technique qui permet d'éviter l'incohérence d'arc consiste à supprimer *à l'avance*, des domaines des variables, les valeurs qui sont incohérentes avec les contraintes binaires. Par exemple, la paire de variables (v_i,v_j) est cohérente si et seulement si pour chaque valeur x dans le domaine de v_i, il existe une valeur y dans le domaine de v_j telle que la paire (x , y) satisfait la contrainte binaire existante entre v_i et v_j (Kumar, 92). Ainsi, lorsqu'on affecte une valeur à une variable, toutes les contraintes impliquant cette variable sont examinées. Cela est connu sous le nom de *propagation de contraintes* puisque les contraintes examinées réduiront les domaines de toutes les variables reliées. La propagation de contraintes permet de réduire la taille du problème à résoudre (Kumar, 92).

En pratique, les problèmes CSPs sont généralement résolus en combinant le *backtracking* intelligent et la propagation de contraintes. Dans notre exemple des N-

Reines, si on affecte la valeur j à la reine de la rangée i (i.e. on place la reine de la rangée i sur la colonne à la position j), la valeur j est supprimée des domaines de toutes les autres variables. Durant la propagation de contraintes, si le domaine d'une variable devient vide, l'algorithme de recherche revient sur ses pas (*backtrack*) jusqu'au dernier point où une valeur a été affectée à une variable (appelé *point de choix*) et il essaye une valeur qui n'a pas encore été essayée. S'il n'y a plus de valeurs à essayer, l'algorithme revient au point de choix précédent, et ainsi de suite. S'il n'y a plus de points de choix et le domaine d'une variable est vide alors le problème n'a pas de solution.

4.4.2 Le problème de *Graph Pattern Matching* traduit en CSP

Rudolf (98) a proposé de traduire les problèmes de *graph pattern matching* en des problèmes CSP équivalents. En utilisant cette approche pour un graphe non typé, on fait correspondre à chaque nœud et à chaque arête d'un graphe patron (i.e. le graphe recherché) une variable distincte. Le domaine de chaque variable associée à un nœud est l'ensemble des nœuds appartenant au graphe cible (i.e. le graphe dans lequel nous recherchons une instance du patron), et le domaine de chaque variable associée à une arête est l'ensemble des arêtes appartenant au graphe cible. Les restrictions reliées à l'homomorphisme de graphes sont exprimées en utilisant des contraintes sur les variables.

Considérons l'exemple de la Figure 4.6, extrait de (Rudolf, 1998), où nous cherchons des occurrences du graphe L dans le graphe cible G. Selon cette approche, nous avons trois variables x_1, x_2, x_3 dont les domaines sont $D_1 = D_3 = \{d_1, d_3, d_4, d_6\}$ et $D_2 = \{d_2, d_5\}$ où D_i est le domaine de la variable x_i.

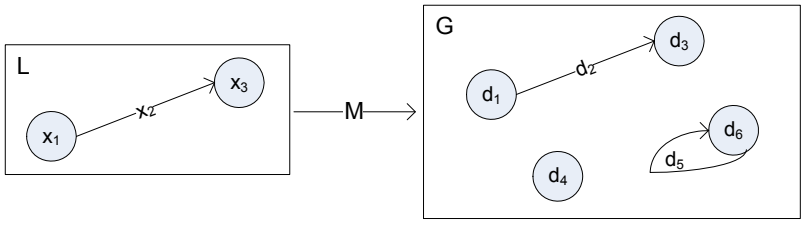


Figure 4.6. Un exemple de graph pattern matching

Les propriétés d'homomorphisme sont exprimées par deux contraintes qui spécifient que lorsque des valeurs ont été affectées aux variables x_1, x_2 et x_3, la valeur affectée à x_2—laquelle est une arête—a comme nœud source la valeur affectée à x_1 et comme nœud cible la valeur affectée à x_3 :

- $C_{src}(x_2, x_1) = \{ (d_e, d_v) \in D_2 \times D_1 \mid source(d_e) = d_v \}$
- $C_{tar}(x_2, x_3) = \{ (d_e, d_v) \in D_2 \times D_3 \mid target(d_e) = d_v \}$

La contrainte $C_{src}(x_2, x_1)$ réduit l'ensemble des paires de valeurs pouvant être affectées simultanément à x_1 et x_2 à l'ensemble $\{(d_2, d_1), (d_5, d_6)\}$, et la contrainte $C_{tar}(x_2, x_3)$ réduit l'ensemble des paires de valeurs pouvant être affectées simultanément à x_2 et x_3 à l'ensemble $\{(d_2, d_3), (d_5, d_6)\}$. Ainsi la réduction des domaines des variables par ces deux contraintes aboutit à deux solutions possibles ($x_1=d_1$, $x_2=d_3$, $x_3=d_2$) et ($x_1=d_6$, $x_2=d_5$, $x_3=d_6$).

4.5 Construction des CSPs à partir des modèles de problèmes

Notre objectif est d'utiliser l'approche proposée par Rudolf (Rudolf, 98) pour trouver des instances de nos modèles de problèmes dans des modèles donnés en entrée par le concepteur. Ainsi nous avons besoin de traduire notre problème de *pattern matching* en un problème CSP. Pratiquement, cela signifie la génération d'un ensemble de variables et d'un ensemble de contraintes pour un modèle de problème donné, et le calcul des domaines des variables à partir du modèle donné en entrée. Le processus de génération des CSPs à partir des modèles de problèmes et des modèles en entrée est illustré par la Figure 4.7. Nous en décrivons les principales étapes dans les sous-sections qui suivent.

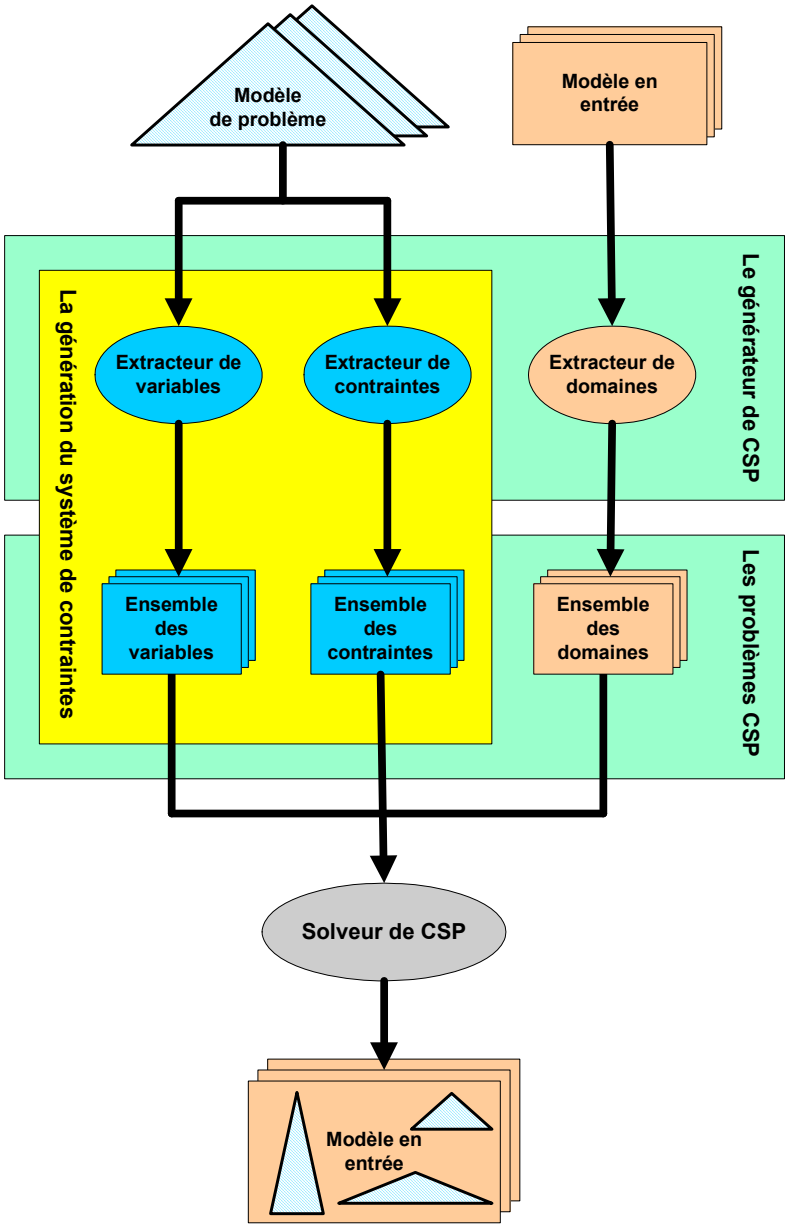


Figure 4.7. Le processus de génération des CSPs à partir des modèles de problèmes

4.5.1 Extraction des variables

Pour un modèle de problème donné, les variables correspondent aux entités qui composent le modèle. Cela comprend les classes, les opérations, les attributs et toutes les relations entre eux. Les relations entre les différentes entités peuvent être des entités du niveau modèle UML (e.g. association), ou des entités du niveau méta-modèle UML (e.g. héritage, relation entre les méta-classes `Class` et `Operation` du

méta-modèle UML) ou des relations qui ne sont pas représentées ni au niveau modèle UML ni au niveau méta-modèle UML (e.g. la relation d'implémentation existante entre deux opérations).

Ainsi, dans le cas du modèle de problème du patron Visiteur, les variables extraites (Figure 4.8) sont: `var_AbstractClass`, `var_ConcreteClass`, `var_AbstractOperation`, `var_ConcreteOperation`, `var_inherits_from_1`, `var_inherits_from_2`, `var_has_method`, `var_has_message`, `var_implements`, et `var_overrides`. Les quatre premières variables correspondent aux entités du modèle de problème, i.e. les classes et les opérations. Les six dernières variables correspondent aux associations entre ces entités. Nous avons utilisé deux variables `var_inherits_from_1` et `var_inherits_from_2` pour distinguer la relation d'héritage entre une classe concrète et une classe abstraite de la relation d'héritage entre une classe concrète et une classe concrète.

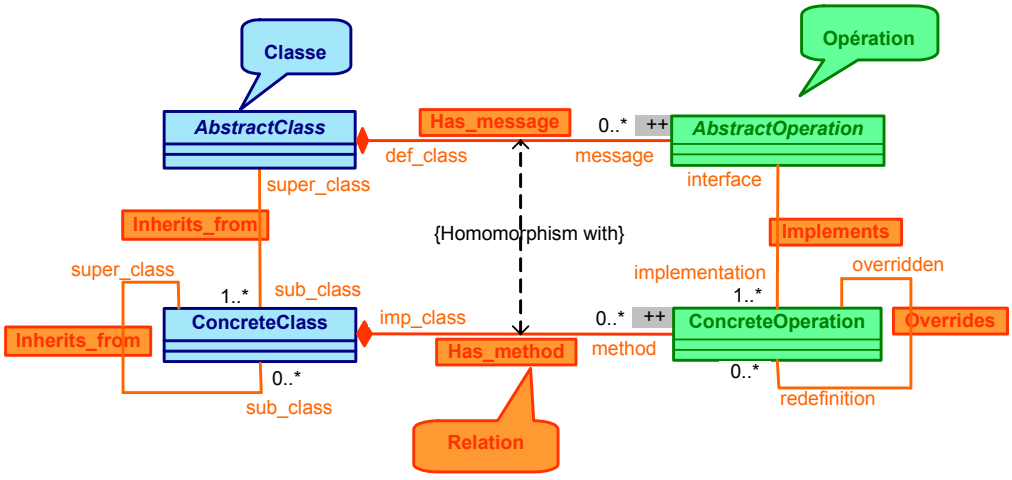


Figure 4.8. Les variables extraites du modèle du problème du patron Visiteur

4.5.2 Extraction des domaines

Étant donné un modèle en entrée et en se référant à notre exemple du patron Visiteur, nous avons besoin d'identifier les domaines pour les dix variables qui ont été extraites du modèle de problème du patron Visiteur. L'identification des domaines soulève un problème commun à tous les problèmes CSPs et qui est relié au niveau de précision des domaines et à l'effort investi dans l'extraction des domaines. En effet,

considérons l'exemple de la variable `var_AbstractClass`. Nous pouvons définir le domaine de cette variable comme étant l'ensemble des classes abstraites du modèle à l'entrée, ce qui signifie que nous devons développer une procédure pour les identifier dans le modèle en entrée. Autrement, nous pouvons aussi définir le domaine de cette variable comme étant l'ensemble des classes du modèle à l'entrée, mais ajouter à l'ensemble des contraintes une contrainte unaire qui identifie les classes abstraites parmi l'ensemble des classes. En fait, nous pourrions même utiliser l'ensemble des entités du modèle en entrée comme domaine de la variable `var_AbstractClass` et ajouter deux contraintes unaires pour spécifier que c'est une entité de type classe et qu'elle est abstraite.

L'avantage d'avoir un domaine précis (i.e. implicitement inclure la contrainte unaire dans le domaine) est que sa taille est réduite ce qui améliore le processus de recherche de solution des CSPs. C'est généralement la démarche utilisée pour traiter les contraintes unaires puisqu'elle évite la vérification dynamique et répétitive de ces contraintes dans le processus de *backtracking* (Rudolf, 98). L'inconvénient de cette démarche est que, du point de vue de la représentation, une certaine connaissance essentielle du problème est enfouie dans des procédures implicites et non déclaratives (i.e. dans les extracteurs de domaines) au lieu d'être exprimée de façon explicite et déclarative avec des contraintes (El-boussaidi et Mili, 2008).

En se basant sur les variables extraites de différents modèles de problème (e.g. Visiteur, Composite, Pont), nous distinguons deux familles de domaines à extraire d'un modèle en entrée: (1) les domaines qui sont directement extraits à partir du modèle en entrée, i.e. ceux ne nécessitant pas de calculs pour les déduire, et (2) les domaines qui sont calculés à partir du modèle en entrée. En effet, l'utilisation de modèles d'analyse en UML offre des avantages mais aussi des inconvénients. Pour illustrer cela, un extrait du méta-modèle UML est montré à la Figure 4.9.

Comme le montre le méta-modèle UML, les concepts de « classe » et « classe abstraite » sont explicitement représentés et donc le calcul des domaines des variables `var_AbstractClass` et `var_ConcreteClass` est simple :

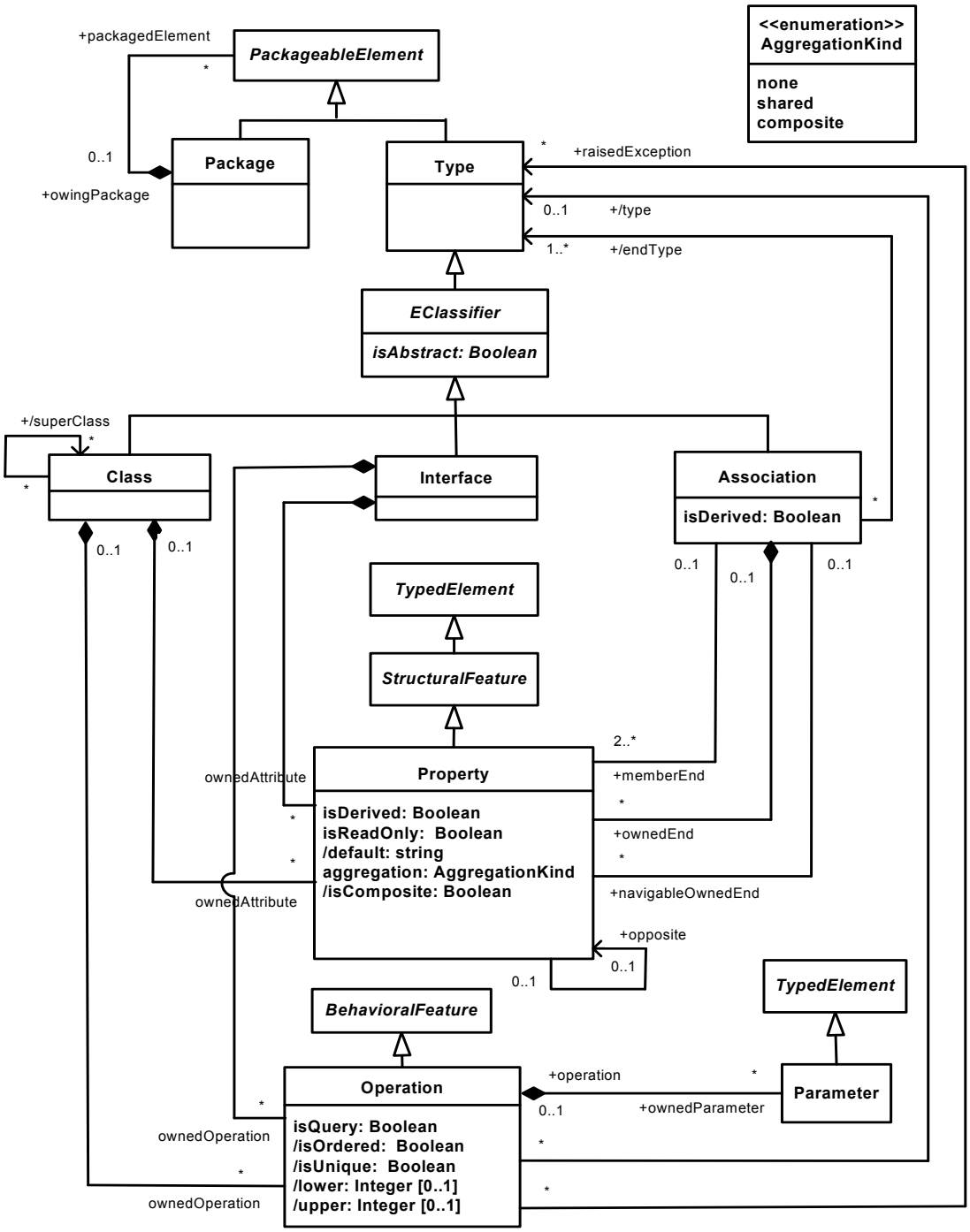


Figure 4.9. Extrait du méta-modèle UML

- Domain(var_AbstractClass) = {entity ∈ InputModel[2] | entity **isInstanceOf** Class ∧ entity.abstract=true}
- Domain(var_ConcreteClass) = {entity ∈ InputModel | entity **isInstanceOf** Class ∧ entity.abstract=false}

Par contre, les domaines des variables `var_AbstractOperation` et `var_ConcreteOperation` nécessitent un calcul plus élaboré. En effet, même si UML définit dans son méta-modèle la classe `Operation`, il ne définit pas la notion d'abstraction pour les opérations : l'abstraction des opérations est un concept de programmation. Ainsi, nous pouvons utiliser l'ensemble des opérations du modèle en entrée comme domaine autant pour la variable `var_AbstractOperation` que pour la variable `var_ConcreteOperation` et ajouter des contraintes pour identifier celles qui sont abstraites. Alternativement, nous pouvons enrichir la procédure d'extraction du domaine pour générer des domaines différents. Dans les deux cas, on ne peut disposer de cette information que si nous disposons du code source associé au modèle.

L'extraction des domaines pour les variables `var_inherits_from_1`, `var_inherits_from_2`, `var_has_method` et `var_has_message` est encore moins évidente. Ces variables correspondent à des relations qui sont implicites dans un modèle UML puisqu'elles sont représentées au niveau du méta-modèle UML par des associations et non pas par des entités. En effet, dans un modèle UML, la relation ''`Has_method`'' entre une classe et ses méthodes est implicite dans le fait que la méthode fait partie de la définition de la classe : elle n'est pas représentée par une association ou une entité. De la même façon, la relation d'héritage entre deux classes est implicite dans un modèle UML malgré qu'elle soit couramment représentée par un lien dans les outils de modélisation. En fait, la relation d'héritage n'est ni une association ni une entité au niveau modèle, i.e. une classe a une liste de super-classes

[2] `InputModel` est une instance de la méta-classe `Package`. Il correspond au modèle en entrée. En pratique, il faut parcourir toute l'arborescence du modèle en entrée puisque c'est un paquetage pouvant contenir d'autres paquetages.

comme elle a une liste de méthodes. L'héritage est aussi représenté par une association entre les classes au niveau méta-modèle.

Les choses se compliquent encore plus avec les variables `var_implements` et `var_overrides` et les relations sous-jacentes entre opérations. Ces relations n'ont aucune représentation dans le méta-modèle UML et doivent être construites en se basant sur l'information donnée dans le modèle en entrée. Nous expliquerons plus concrètement la façon dont nous calculons les différents domaines lors de la description de l'implémentation (section 4.6.2).

4.5.3 Extraction des contraintes

Les contraintes spécifient quelles valeurs peuvent être affectées simultanément aux variables. Dans notre contexte, nous avons besoin de spécifier que lorsque des valeurs sont affectées à des variables, ces valeurs ont le type approprié (e.g. seulement une classe du modèle peut être affectée à la variable `var_ConcreteClass`) et qu'elles satisfont les relations existant entre les variables. Comme nous l'avons souligné dans la section précédente (4.5.2), la compatibilité de type peut être spécifiée explicitement en utilisant une contrainte unaire sur une variable ou implicitement en réduisant le domaine de la variable en question. Pour réduire le nombre de contraintes à vérifier dynamiquement et aussi la taille des domaines associés aux variables, nous avons adopté la seconde approche.

Les contraintes représentent, dans notre contexte, les relations entre les variables du problème : elles reflètent les liens entre les différentes entités du modèle de problème. Donc, les contraintes sont utilisées pour faire la correspondance entre des extrémités d'associations et les entités qui leur sont associées. Pour illustrer cette idée, la Figure 4.10 montre un fragment du modèle de problème du patron Visiteur ainsi que les variables et les contraintes construites à partir de ce fragment.

Comme nous l'avons expliqué à la section 4.5.1, les entités `AbstractClass`, `ConcreteClass` et `Inherits_from` sont représentées par les variables `var_AbstractClass`, `var_ConcreteClass` et `var_inherits_from_1`. Les domaines de ces variables sont respectivement l'ensemble de toutes les classes abstraites du

modèle en entrée, l'ensemble de toutes les classes concrètes du modèle en entrée et l'ensemble de toutes les relations d'héritage calculées à partir du modèle en entrée. Notons que les relations d'héritage sont représentées par la classe `InhertisFromRelationship` ayant les attributs `super-class` et `sub_class`. Des instances de cette classe sont créées pendant la phase d'extraction des domaines. Cette classe est décrite plus en détails dans la section 4.6.2.1.

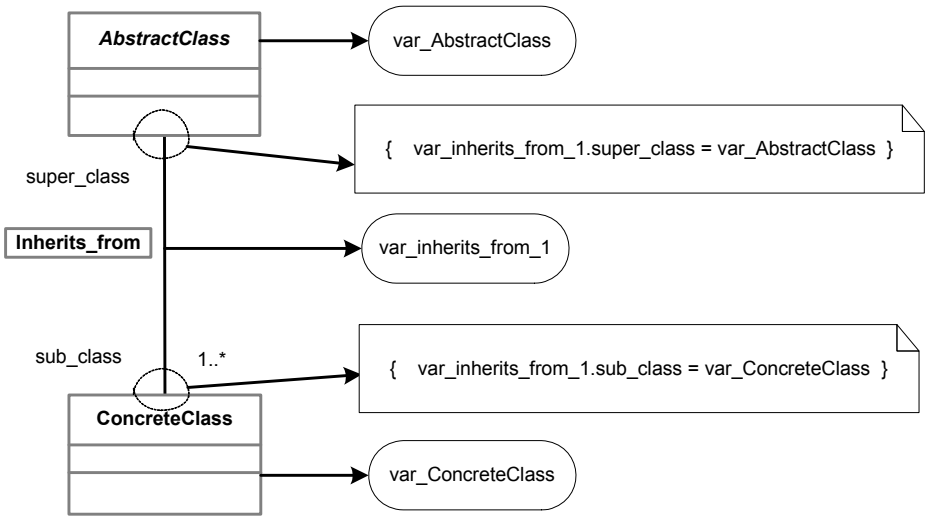


Figure 4.10. Illustration de la correspondance entre le modèle du problème et les variables et les contraintes.

Comme l'illustre la Figure 4.10, lorsqu'une classe $\texttt{class}_j$ est affectée à la variable `var_AbstractClass`, la relation d'héritage $\texttt{inherits}_i$ affectée à la variable `var_inherits_from_1` doit avoir son attribut `super_class` égal à $\texttt{class}_j$. Inversement, lorsqu'une valeur $\texttt{inherits}_i$ est affectée à la variable `var_inherits_from_1`, la classe $\texttt{class}_j$ affectée à la variable `var_AbstractClass` doit être égale à l'attribut `super_class` de $\texttt{inherits}_i$. Formellement, nous exprimons ça par une contrainte binaire `Cons_Inherits_tar` qui réduit les domaines des variables `var_AbstractClass` et `var_inherits_from_1` comme suit :

- Cons_Inherits_tar(var_inherits_from_1, var_AbstractClass) = { ($inherits_i$, $class_j$) ∈ Domain(var_inherits_from_1) × Domain(var_AbstractClass) | $inherits_i$.super_class = $class_j$}

De façon similaire, nous utilisons une contrainte binaire `Cons_Inherits_src` pour restreindre les valeurs pouvant être affectées simultanément aux variables `var_ConcreteClass` et `var_inherits_from_1` :

- Cons_Inherits_src(var_inherits_from_1, var_ConcreteClass) = { ($inherits_i$, $class_j$) ∈ Domain(var_inherits_from_1) × Domain(var_ConcreteClass) | $inherits_i$.sub_class = $class_j$}

De façon générale, pour chaque variable qui représente une ‘‘*arête*’’ dans notre modèle de problème (e.g. `var_has_method`, `var_has_message`, `var_implements` et `var_overrides`), nous devons spécifier deux contraintes comme nous l’avons fait pour la variable `var_inherits_from_1` : une contrainte pour assurer la compatibilité de la source de l’arête (e.g. `Cons_Inherits_src`) et une contrainte pour assurer la compatibilité de la cible de l’arête (e.g. `Cons_Inherits_tar`). Le Tableau 4-1 montre tous les types de contraintes qui ont été extraites à partir du modèle de problème du patron Visiteur.

Contrainte	**Variables**	**Réduction correspondante des domaines des variables**
Cons_Inherits_src	var_inherits_from var_class	{ (inherits$_i$, class$_j$) ∈ Domain(var_inherits_from) × Domain(var_class) | inherits$_i$.sub_class = class$_j$}
Cons_Inherits_tar	var_inherits_from var_class	{ (inherits$_i$, class$_j$) ∈ Domain(var_inherits_from_1) × Domain(var_class) | inherits$_i$.super_class = class$_j$}
Cons_HasMethod_src	var_has_method var_class	{ (hasMethod$_i$, class$_j$) ∈ Domain(var_has_method) × Domain(var_class) | hasMethod$_i$.imp_class = class$_j$ }
Cons_HasMethod_tar	var_has_method var_operation	{ (hasMethod$_i$, operation$_j$) ∈ Domain(var_has_method) × Domain(var_operation) | hasMethod$_i$.method = operation$_j$ }
Cons_HasMessage_src	var_has_message var_abstract_class	{ (hasMessage$_i$, class$_j$) ∈ Domain(var_has_message) × Domain(var_abstract_class) | hasMessage$_i$.def_class = class$_j$ }
Cons_HasMessage_tar	var_has_message var_abst_operation	{ (hasMessage$_i$, operation$_j$) ∈ Domain(var_has_message) × Domain(var_abst_operation) | hasMessage$_i$.message = operation$_j$ }
Cons_Implements_src	var_implements var_operation	{ (implements$_i$, operation$_j$) ∈ Domain(var_implements) × Domain(var_operation) | implements$_i$.implementation = operation$_j$}
Cons_Implements_tar	var_implements var_abst_operation	{ (implements$_i$, operation$_j$) ∈ Domain(var_implements) × Domain(var_abst_operation) | implements$_i$.interface = operation$_j$}
Cons_Overrides_src	var_overrides var_operation	{ (overrides$_i$, operation$_j$) ∈ Domain(var_overrides) × Domain(var_operation) | overrides$_i$.redefinition = operation$_j$}
Cons_Overrides_tar	var_overrides var_operation	{ (overrides$_i$, operation$_j$) ∈ Domain(var_overrides) × Domain(var_operation) | overrides$_i$.overriden = operation$_j$}

Tableau 4-1. Différents types de contraintes extraites du modèle de problème du patron Visiteur

4.6 Implémentation

Nous avons implémenté notre stratégie de *pattern matching* en utilisant le solveur de contraintes ILOG JSolver™. Le fait d'avoir des variables dont les domaines sont des objets et non des valeurs scalaires, a été un facteur déterminant dans le choix du solveur de contraintes. En effet, ILOG JSolver est un solveur de contraintes écrit entièrement en Java et conçu pour manipuler des modèles objet en Java. Il permet de définir de nouvelles heuristiques pour modifier l'algorithme de

recherche de solutions à un problème CSP, e.g. une heuristique pour l'ordre des variables ou des valeurs. Un autre avantage important de JSolver est qu'il permet aux utilisateurs de définir leurs propres contraintes.

Dans le reste de la section, nous donnerons d'abord une description sommaire de JSolver (section 4.6.1) puis nous présenterons les détails de notre implémentation (sections 4.6.2 et 4.6.3).

4.6.1 Description de JSolver

ILOG JSolver consiste en une API complète de programmation par contraintes pour modéliser le problème à résoudre (i.e. les variables et les contraintes) et trouver les solutions. La classe noyau de l'API est la classe `IlcSolver`. Une instance de cette classe est responsable de la modélisation et de la résolution d'un problème CSP. Elle est une sorte de fabrique (i.e. *Factory*) qui offre des méthodes pour créer des variables et des contraintes. Elle offre aussi des méthodes pour lancer la recherche d'une solution et parcourir l'ensemble des solutions trouvées, s'il en existe plus d'une. La Figure 4.11 montre du code illustrant le style de programmation déclarative de l'API de JSolver.

Dans cet exemple, nous créons une variable `x` dont le domaine est l'ensemble des nombres entiers entre 0 et 10. Nous créons ensuite une contrainte (`x ≠ 5`) que nous ajoutons au solveur avant de lancer la recherche d'une solution à ce problème en utilisant la méthode `newSearch()` du solveur. Dans la boucle `while`, nous parcourons les différentes solutions trouvées grâce à la méthode `next()` qui permet de passer d'une solution à la suivante. Chaque solution trouvée est affichée à l'écran. Dans ce cas-ci, le programme affichera les valeurs 0, 1, 2, 3, 4, 6, 7, 8, 9 et 10. Finalement, la méthode `endSearch()` termine la recherche et supprime les objets internes créés par JSolver tels que l'arbre de recherche, les points de choix, etc.

```
//create an instance of the IlcSolver class
IlcSolver solver = new IlcSolver();
//create a constrained integer variable
IlcIntVar x = solver.intVar(0, 10);
//create a constraint on the variable
IlcConstraint ct = solver.neq(x, 5);
//add the constraint to the instance of IlcSolver
solver.add(ct);
//start searching for solutions
solver.newSearch();
//searching for and printing all solutions
while (solver.next()) {
        System.out.println(x.getDomainValue());
}
//ending the search
solver.endSearch();
```

Figure 4.11. Simple illustration du fonctionnement de JSolver.

JSolver offre un grand nombre de contraintes prédéfinies et un ensemble d'opérateurs logiques pour les combiner pour définir des contraintes plus complexes. Cependant, dans certains contextes, il est nécessaire de définir des contraintes qui ne peuvent être exprimées en terme des contraintes prédéfinies. Pour cela, JSolver offre aux utilisateurs la possibilité de personnaliser leurs applications en définissant leurs propres contraintes. Nous décrivons la façon de définir de nouvelles contraintes dans JSolver à la section 4.6.2.2 et nous donnons un exemple de contrainte.

4.6.2 Description de notre implémentation

La Figure 4.12 montre les paquetages principaux de notre implémentation du processus de marquage.

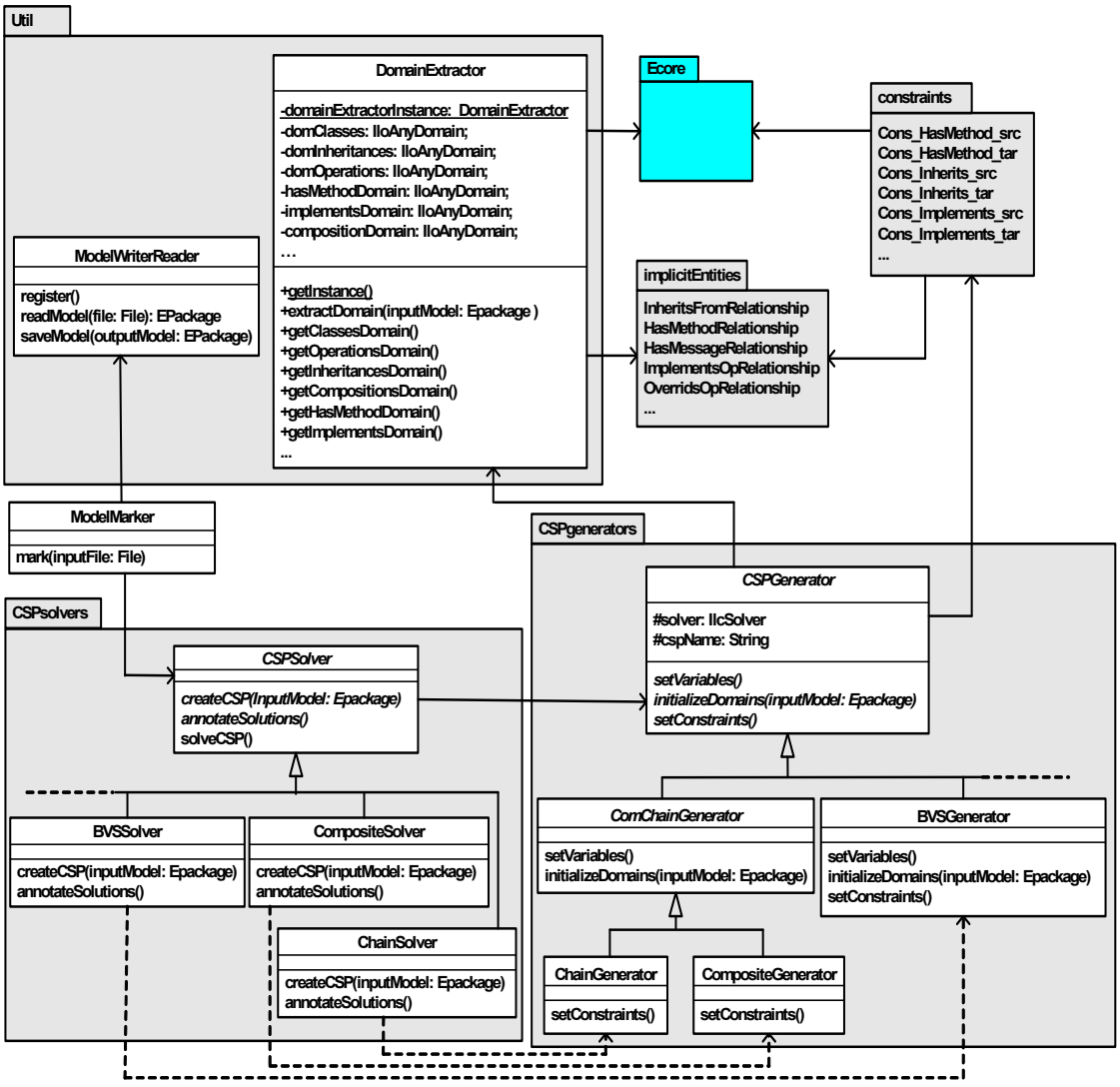


Figure 4.12. Principales classes de notre implémentation du processus de marquage.

En pratique, nous avons eu besoin d'implémenter un générateur de CSP pour chaque modèle de problème associé à un patron de conception. Cependant, ces générateurs partagent une même interface regroupant les fonctions pour extraire les variables et leur affecter les domaines appropriés et pour extraire les contraintes. Les classes génératrices des CSPs pour les différents patrons sont regroupées dans le paquetage `CSPgenerators`.

Le diagramme de séquence montrée à la Figure 4.13 décrit les interactions entre les différentes classes de notre framework. La classe d'entrée à notre application

est la classe `ModelMarker`. Cette classe permet à travers sa méthode `mark(inputFile: File)` de lire le modèle reçu en entrée sous format XMI, de lancer le processus de marquage et de sauvegarder le modèle marqué. Pour lire un modèle à partir d'un fichier XMI et sérialiser le modèle à la fin du processus de marquage, la méthode `mark` fait appel aux services de la classe utilitaire `ModelWriterReader`.

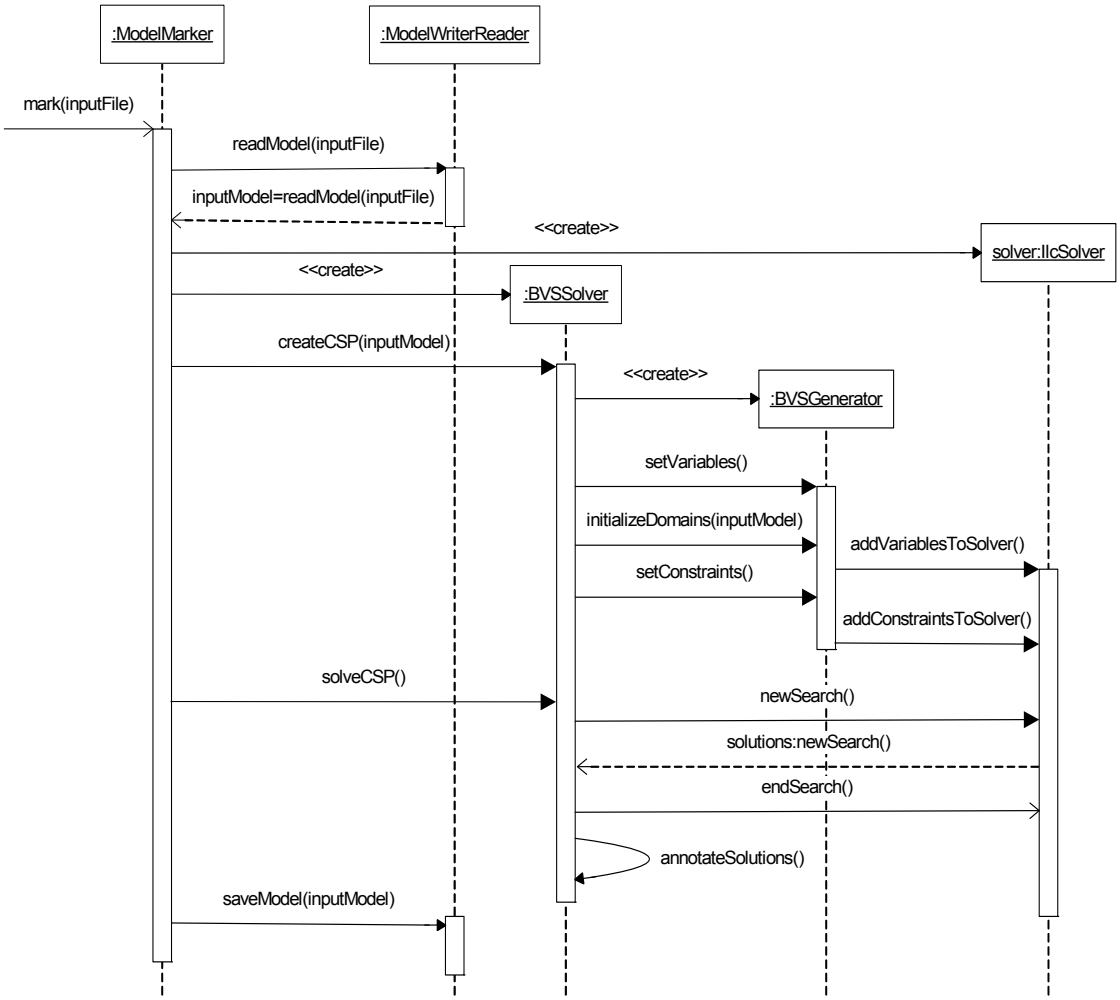


Figure 4.13. Diagramme de séquence décrivant les interactions entre les différentes classes de notre framework.

La classe `ModelMarker` délègue la création des générateurs des CSPs (et donc la création des CSPs), la résolution des CSPs générés et l'annotation des solutions obtenues à des classes de type ''*solveurs*'' regroupées dans le paquetage `CSPsolvers`

(Figure 4.12). En fait, nous avons utilisé le patron « Fabrication abstraite » pour implémenter les solveurs et les générateurs de systèmes de contraintes associés aux différents patrons de conception.

Une classe de type `CSPsolver` crée une instance de la classe associée de type `CSPGenerator`. Le préfixe du nom des classes de type `CSPsolver` et `CSPGenerator` correspond au patron concerné. Par exemple, la classe `BVSSolver` crée une instance de la classe `BVSGenerator`, le préfixe `BVS` fait référence aux patrons Pont (B pour *Bridge*), Visiteur (V) et Stratégie (S) qui ont les mêmes modèles statiques de problèmes comme nous l'avons expliqué au chapitre 3. Ainsi, nous utilisons le même générateur et solveur de contraintes pour ces trois patrons. Les générateurs de CSPs associés aux patrons Composite et Chaîne de responsabilité étendent, aussi, un générateur commun plus général puisque leurs modèles respectifs de problèmes sont très similaires.

Chaque classe de type `CSPGenerator` implémente les méthodes suivantes :

- La méthode `setVariables()` qui crée l'ensemble des variables associées au modèle de problème relié (i.e. le ou les patrons associés à cette classe).
- La méthode `initializeDomains()` qui reçoit en paramètre le modèle en entrée et fait appel aux méthodes de la classe `DomainExtractor` pour initialiser chacune des variables.
- La méthode `setContraints()` qui définit les contraintes sur les variables.

Une classe de type `CSPGenerator` est aussi responsable d'ajouter les variables et les contraintes créées à l'instance courrante de la classe `IlcSolver` laquelle a été créée par l'instance de `ModelMarker`. Pour simplifier le diagramme de séquence, nous avons utilisé des méthodes *fictives* que nous avons appelées `addVariablesToSolver` et `addConstraintsToSolver`. La façon dont on ajoute les variables et les contraintes au solveur `IlcSolver` a été décrite à la sous-section précédente.

Chaque classe de type `CSPSolver` (i.e. les classes de fabrication) doit implémenter deux méthodes :

- `createCSP()` qui crée une instance du générateur approprié de CSPs et fait appel aux méthodes de cette instance (i.e. `setVariables()`, `initializeDomains()` et `setContraints()`) pour initialiser le problème créé.
- `annotateSolutions()` qui marque dans le modèle reçu en entrée les instances trouvées par le solveur.

C'est la méthode `solveCSP()`, implémentée par la super-classe `CSPSolver`, qui permet de lancer la recherche des solutions à tous les problèmes qui ont été créés. Elle fait ensuite appel à la méthode `annotateSolutions()` pour marquer les entités faisant partie des instances retournées par le solveur IlcSolver. Finalement, une fois le processus d'annotation est fini, le modèle est sérialisé sous format XMI.

Nous décrirons dans la section 4.6.2.1 comment fonctionne l'extracteur de domaines et comment il utilise les classes définies dans le paquetage `implicitEntities` pour extraire les domaines des variables qui ne correspondent pas à des entités explicites d'un modèle UML. Nous décrirons ensuite (section 4.6.2.2) comment nous avons implémenté nos contraintes. Enfin à la section 4.6.3, nous illustrerons le fonctionnement de notre application de marquage en utilisant l'exemple du patron Visiteur.

4.6.2.1 Extraction des domaines à partir des modèles EMF

L'extraction des domaines à partir d'un modèle reçu en entrée est faite de façon indépendante du problème de conception dont nous cherchons une instance. La classe `DomainExtractor` implémente l'ensemble des méthodes nécessaires pour extraire les domaines des différentes variables. Les méthodes de la classe `DomainExtractor` manipulent des objets EMF puisque nos modèles sont des modèles EMF (notre méta-modèle est une extension du méta-modèle EMF).

Comme nous l'avons expliqué à la section 4.5.2, nous distinguons les domaines directement extraits à partir d'un modèle de ceux que nous devons calculer en utilisant les données du modèle. Les domaines directement extraits des modèles correspondent aux domaines des variables explicitement représentées dans un modèle

UML (i.e. instances des classes du méta-modèle UML). Ces domaines sont décrits en termes des objets EMF dans la table suivante (Tableau 4-2).

Nom du domaine	L'ensemble correspondant extrait du modèle
AllClassesDomain	$\{class_i \in InputModel.Classifiers\}$
AbstractClassesDomain	$\{class_i \in InputModel.Classifiers \mid class_i.abstract = true\}$
InterfacesDomain	$\{class_i \in InputModel.Classifiers \mid class_i.interface = true\}$
ConcreteClassesDomain	$\{class_i \in InputModel.Classifiers \mid class_i.abstract = false \wedge class_i.interface = false \}$
OperationsDomain	$\{operation_i \in InputModel.Classifiers.Operations\}$
AttributesDomain	$\{atttribute_i \in InputModel.Classifiers.Attributes\}$
CompositionsDomain	$\{reference_i \in InputModel.Classifiers.References \mid reference_i.containment = true\}$
AssociationsDomain	$\{reference_i \in InputModel.Classifiers.References \mid reference_i.containment = false\}$

Tableau 4-2. Les domaines directement extraits d'un modèle

Les domaines qui sont inférés moyennant un calcul, sont les domaines des variables qui correspondent à des entités non explicites d'un modèle UML. Un domaine de cette catégorie comprend un ensemble d'entités que nous devons créer en nous basant sur les informations du modèle en entrée. Par exemple, dans le cas du patron Visiteur, nous avons besoin de déduire à partir du modèle en entrée les entités qui vont former le domaine de la variable `var_inherits_from_1`. En fait, nous avons besoin de calculer les domaines des variables `var_inherits_from_1`, `var_inherits_from_2`, `var_has_method`, `var_has_message`, `var_implements`, et `var_overrides`.

Pour ce faire, nous avons représenté chacune de ces entités/variables explicitement par une classe que nous instancions en nous basant sur l'information

disponible dans le modèle en entrée pour former le domaine de la variable correspondante. Le paquetage `implicitEntities` regroupe l'ensemble de ces classes qui décrivent des objets n'existant pas explicitement dans un modèle UML.

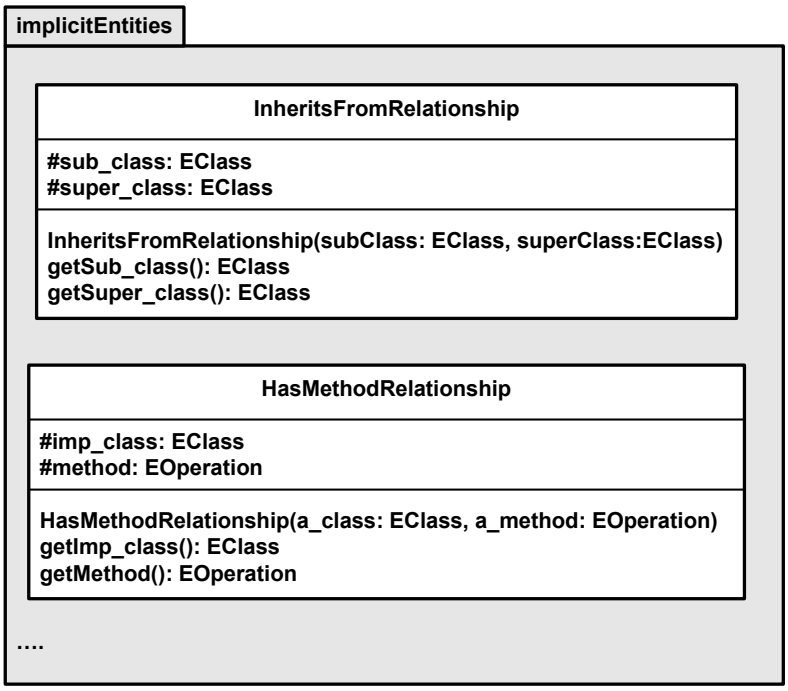


Figure 4.14. Extrait du paquetage `implicitEntities`

Par exemple, ce paquetage contient une classe appelée `InheritsFromRelationship` et dont l'interface est montrée à la Figure 4.14. Cette classe possède deux attributs correspondant aux classes qu'elle relie. En l'occurrence, une instance de `InheritsFromRelationship` est une relation d'héritage entre les classes référencées par les attributs `sub_class` et `super_class`. De façon générale, chaque classe du paquetage `implicitEntities` a deux attributs correspondant respectivement à la *source* et à la *cible* de la relation qu'elle représente, et un ensemble de méthodes pour accéder à ces attributs.

Concrètement, la classe `DomainExtractor` créé une instance de `InhertisFromRelationship` pour chaque relation d'héritage déduite du modèle en entrée, en procédant de la façon suivante :

- $\forall$ $class_i \in$ InputModel.Classifiers, if($class_i$.superType # null)[3] then create(InheritsFromRelation(

  sub_class = $class_i \wedge$ super_class = $class_i$. superType)).

4.6.2.2 Implémentation des contraintes

JSolver définit une classe générique `IlcUserConstraint` qui peut être étendue par l'utilisateur pour définir de nouvelles contraintes. Une sous-classe de la classe `IlcUserConstraint` doit implémenter obligatoirement deux méthodes : `propagate()` et `post()`. En fait, lorsqu'on définit une nouvelle contrainte sur des variables, on doit implémenter les réductions des domaines de ces variables en éliminant les valeurs qui ne peuvent faire partie d'une solution. Ces réductions de domaines sont réalisées par des modificateurs élémentaires de domaines. Pratiquement, dans la méthode `propagate()`, on utilise ces modificateurs pour supprimer les valeurs incohérentes des domaines des variables que la contrainte relie. Autrement dit, les modificateurs de domaines permettent d'implémenter l'*invariant* d'une contrainte. Quelques exemples de modificateurs élémentaires de domaines utilisés dans la propagation sont : `setDomainValue()`, `removeDomainValue()`, `setDomainRange()`, `setDomainMin()`, etc.

La méthode `post()` permet de définir les événements de propagation reliés à la contrainte. Un événement de propagation est déclenché par la modification du domaine d'une variable. Les mots clés associés aux événements de propagation sont :

- `whenValue()` : une valeur a été affectée à la variable.
- `whenRange()` : indique qu'au moins une des limites (minimale ou maximale) du domaine de la variable, a été changée.
- `whenDomain()` : indique que le domaine de la variable a été modifié.

---

[3] En réalité, nous manipulons une liste de super-classes (appelée `eSuperTypes` dans EMF). Toutefois, cette liste ne contient qu'une seule super-classe puisque le langage Java ne supporte pas l'héritage multiple.

Le paquetage `constraints` (Figure 4.12) regroupe l'ensemble des contraintes que nous avons définies et implémentées comme extension aux contraintes fournies par le solveur. Un extrait de la classe implémentant la contrainte `Cons_inherits_from_src` est montré à la Figure 4.15. Cette classe définit des attributs (`var_inherits_from` et `var_class`) correspondant aux variables sur lesquelles la contrainte s'applique (lignes 2 et 3). Le constructeur de la classe (lignes 4 à 8) permet d'initialiser ces attributs par les paramètres donnés en entrée à la contrainte. L'invariant de cette contrainte—implémentée par sa méthode `propagate()` (lignes 13 à 34)–peut s'exprimer ainsi :

- Si on affecte une valeur à `var_inherits_from` (`isBound()=true`) (ligne 14), le domaine de la variable `var_class` est réduit à une seule valeur (ligne 17) et c'est la valeur contenue dans l'attribut `sub_class` de l'instance de `InheritsFromRelationship` affectée à `var_inherits_from`.
- Si on affecte une valeur à `var_class` (`isBound()=true`) (ligne 19), le domaine de la variable `var_inherits_from` est réduit à l'ensemble des instances de `InheritsFromRelationship` telles que leur attribut `sub_class` est égal à la valeur affectée à `var_class` (lignes 21 à 32). Dans notre contexte, cet ensemble est réduit à une seule instance de `InheritsFromRelationship` puisqu'il n'y a pas d'héritage multiple dans Java.

La contrainte est propagée à chaque fois que nous affectons une valeur à une des deux variables (lignes 10 et 11). En pratique, en plus des méthodes `post()` et `propagate()`, nous avons défini dans les classes implémentant nos contraintes d'autres méthodes qui nous permettent de manipuler les contraintes, i.e. combiner nos contraintes en utilisant les opérateurs logiques.

```
1. public class Cons_Inherits_src extends IlcUserConstraint {
2.  IlcAnyExpr var_inherits_from;
3.  IlcAnyExpr var_class;
4.  public Cons_Inherits_src(IlcAnyExpr _inherits, IlcAnyExpr _class){
5.     super(_inherits.getSolver());
6.     var_inherits_from = _inherits;
7.     var_class = _class;
8.  }
9.  public void post() {
10.    var_inherits_from.whenValue(this);
11.    var_class.whenValue(this);
12. }
13. public void propagate() {
14.    if (var_inherits_from.isBound()) {
15.      InheritsFromRelationship currentInherits =
16.           (InheritsFromRelationship) var_inherits_from.getDomainValue();
17.      var_class.setDomainValue(currentInherits.getSub_class());
18.    }
19.    if (var_class.isBound()){
20.      EClass currentClass = (EClass)var_class.getDomainValue();
21.      InheritsFromRelationship currentInherits = null;
22.      try {
23.           for (Iterator<InheritsFromRelationship> i=
24.                  var_inherits_from.domainIterator(); i.hasNext();){
25.             currentInherits = i.next();
26.             if (!(currentInherits.getSub_class().equals(currentClass))){
27.                  var_inherits_from.removeDomainValue(currentInherits);
28.             }
29.           }
30.      } catch (IlcFailException er) {
31.           er.printStackTrace();
32.      }
33.    }
34.  }
...
   }
```

Figure 4.15. Extrait de la classe `Cons_Inherits_src`

4.6.3 Détection des instances de modèles de problèmes dans des modèles UML

Nous avons utilisé notre approche pour détecter des instances de modèles de problèmes notamment pour les patrons Visiteur, Pont, Stratégie, Composite et Chaîne de responsabilité. Nous présentons et discutons en détails au chapitre 6 les résultats des tests que nous avons exécutés sur des modèles réels. Pour illustrer brièvement les résultats du processus de marquage, nous utiliserons ici notre exemple de la hiérarchie de nœuds (Figure 4.16) comme modèle donné en entrée à notre application.

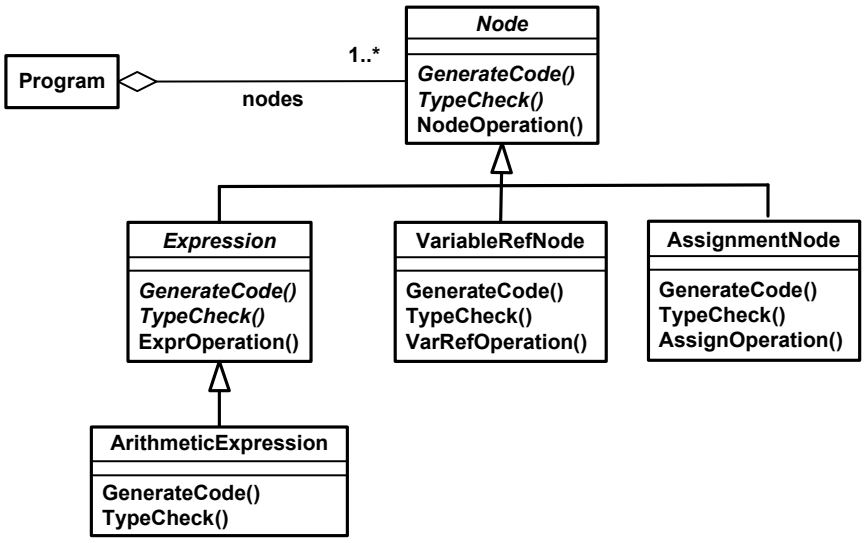


Figure 4.16. Modèle donné en entrée à l'application

En pratique, notre application de marquage reçoit le modèle en format XMI et le sérialise à la fin du processus de marquage en format XMI. La Figure 4.17 montre un extrait en format XMI du modèle de la Figure 4.16. Cet extrait montre les classes `Node` et `VariableRefNode` et leurs méthodes respectives.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="modelForIllustration"
    nsURI="http://modelForIllustration" nsPrefix="modelForIllustration">
    <eClassifiers xsi:type="ecore:EClass" name="Node">
       <eOperations name="TypeCheck"/>
       <eOperations name="GenerateCode"/>
       <eOperations name="NodeOperation"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="VariableRefNode" eSuperTypes="#//Node">
       <eOperations name="TypeCheck"/>
       <eOperations name="GenerateCode"/>
       <eOperations name="VarRefOperation"/>
    </eClassifiers>
…
</ecore:EPackage>
```

Figure 4.17. Extrait du modèle donné en entrée en format XMI

Un extrait du résultat de la première étape du processus de marquage (i.e. *pattern matching avec les CSPs*) appliqué au modèle de la Figure 4.16, est montré à la Figure 4.18. Cet extrait montre les mêmes entités que celles montrées à la Figure 4.17 (i.e. les classes `Node` et `VariableRefNode` et leur méthodes) et qui ont été

marquées. En fait, chaque fois qu'une solution à un problème CSP est trouvée, le solveur (e.g. `BVSSolver`) associé au générateur de ce problème (e.g. `BVSGenerator`) procède à un marquage préliminaire de l'ensemble des entités composant cette solution.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="modelForIllustrationMarked"
    nsURI="http://modelForIllustrationMarked" nsPrefix="modelForIllustrationMarked">
    <eClassifiers xsi:type="ecore:EClass" name="Node">
        <eAnnotations>
          <details key="pattern" value="BVS"/>
          <details key="role" value="abstractClass"/>
          <details key="instance" value="1"/>
        </eAnnotations>
        <eOperations name="TypeCheck">
          <eAnnotations>
            <details key="pattern" value="BVS"/>
            <details key="role" value="abstractOperation"/>
            <details key="instance" value="1"/>
          </eAnnotations>
        </eOperations>
        <eOperations name="GenerateCode">
          <eAnnotations>
            <details key="pattern" value="BVS"/>
            <details key="role" value="abstractOperation"/>
            <details key="instance" value="1"/>
          </eAnnotations>
        </eOperations>
        <eOperations name="NodeOperation"/>
    </eClassifiers>
    <eClassifiers xsi:type="ecore:EClass" name="VariableRefNode" eSuperTypes="#//Node">
        <eAnnotations>
          <details key="pattern" value="BVS"/>
          <details key="role" value="concreteClass"/>
          <details key="instance" value="1"/>
        </eAnnotations>
        <eOperations name="TypeCheck">
          <eAnnotations>
            <details key="pattern" value="BVS"/>
            <details key="role" value="concreteOperation"/>
            <details key="instance" value="1"/>
          </eAnnotations>
        </eOperations>
        <eOperations name="GenerateCode">
          <eAnnotations>
            <details key="pattern" value="BVS"/>
            <details key="role" value="concreteOperation"/>
            <details key="instance" value="1"/>
          </eAnnotations>
        </eOperations>
        <eOperations name="VarRefOperation"/>
    </eClassifiers>
...
</ecore:EPackage>
```

Figure 4.18. Extrait du modèle après la première étape du marquage.

Pour implémenter une marque, nous avons utilisé la classe `EAnnotation` de EMF. Cette classe constitue un mécanisme par lequel des informations additionnelles peuvent être attachées à n'importe quel objet d'un modèle. En effet, dans le méta-modèle EMF (Figure 3.13), toutes les classes sont des sous-classes de la classe `EModelElement` dont une instance peut contenir zéro ou plusieurs instances de la classe `EAnnotation` à travers la référence `eAnnotations`. Une instance de `EAnnotation` permet à travers son attribut `Details` (de type `Map`) de définir différentes clés et de leur associer des valeurs.

Comme le montre le fragment de la Figure 4.18, une marque, associée à une entité, est composée de trois informations : 1) le nom du patron détecté, i.e. quel modèle de problème a été détecté; 2) le rôle de l'entité dans le problème de conception; et 3) un numéro d'instance. Par exemple, la classe `Node` a été marquée avec une annotation dont la clé `pattern` a une valeur `BVS` signifiant qu'elle fait partie d'une instance du modèle statique de problème correspondant aux patrons Pont (*Bridge*), Visiteur et Stratégie. La clé `role` a une valeur `abstractClass` qui correspond au rôle joué par la classe `Node` dans le problème. Finalement, la clé `instance` est utilisée pour spécifier le numéro de l'instance du modèle de problème dont la classe `Node` fait partie, ici la valeur est égale à `1` car le modèle donné en entrée ne contient qu'une seule instance de ce modèle de problème. Nous avons numéroté les instances d'un modèle de problème pour gérer les conflits dus au chevauchement des problèmes de conception. Ce sont les solveurs (e.g. `BVSSolver`, `CompositeSolver`) qui gèrent cette numérotation.

La classe `VariableRefNode` a été marquée de la même façon avec le rôle `concreteClass`. Pour les deux classes (`Node` et `VariableRefNode`), seules les opérations `TypeCheck()` et `GenerateCode()` ont été marquées comme faisant partie du problème détecté.

À ce stade du processus de marquage, dans le cas de certains patrons tels que les patrons Composite et Chaîne de responsabilité, une instance détectée du modèle de problème associé est une instance confirmée, i.e. nous n'avons pas besoin d'une étape ou information supplémentaire. Par contre, pour les patrons de conception dont

l'objectif est de protéger les classes clientes des variations, les instances détectées sont des instances correspondant à leurs modèles *statiques* de problèmes et nous avons besoin de l'information à propos des points de variation pour compléter le marquage. Par exemple, dans le cas où nous avons détecté une instance du modèle statique de problème correspondant aux patrons Pont, Visiteur et Stratégie, nous interagissons avec le concepteur à travers des questions/réponses pour identifier les points de variation dans cet instance, e.g. le nombre de sous-classes d'une classe, le nombre de méthodes d'une classe, ou le nombre d'algorithmes associés aux méthodes.

Ainsi, selon le modèle statique détecté et les points de variation entrés par le concepteur, nous procédons au marquage spécifique d'une instance. Pour notre exemple de la Figure 4.18, si le concepteur confirme que le nombre de méthodes de la classe `Node` est susceptible d'augmenter, nous en déduisons que cette classe fait partie d'une instance du problème résolu par le patron Visiteur et nous générons une annotation spécifique pour ce patron. L'annotation générée est identique à celle que la classe `Node` avait, sauf pour la clé `pattern` qui fait référence au patron Visiteur (Figure 4.19).

```
…
<eClassifiers xsi:type="ecore:EClass" name="Node">
       <eAnnotations>
              <details key="pattern" value="Visitor"/>
              <details key="role" value="abstractClass"/>
              <details key="instance" value="1"/>
       </eAnnotations>
       <eOperations name="TypeCheck">
              <eAnnotations>
                     <details key="pattern" value="Visitor"/>
                     <details key="role" value="abstractOperation"/>
                     <details key="instance" value="1"/>
              </eAnnotations>
       </eOperations>
…
</eClassifiers>
…
```

Figure 4.19. Extrait du résultat du marquage spécifique.

Le concepteur peut très bien spécifier que l'instance détectée d'un modèle statique présente différents types de variation. Par exemple, si le concepteur décidait que le nombre de méthodes de la classe `Node` peut augmenter aussi bien que le

nombre de sous-types de `Node` et de leur classes d'implémentation, nous en déduirions que la classe `Node` fait partie de deux problèmes de conception : un résolu par le patron Visiteur et l'autre résolu par le patron Pont. Dans ce cas, la classe `Node` aurait deux marques (Figure 4.20) : une pour chaque patron (`Visitor.AbstractClass.1` et `Bridge.AbstractClass.1`). Ces marques seront utilisées pour guider les règles de transformation associées aux patrons détectés (présentées au chapitre 5), et aussi pour gérer les interférences existantes entre les instances chevauchantes du même patron ou de différents patrons de conception. Le problème de chevauchement des instances est relié à la définition des problèmes de conception, il sera discuté plus en détails au chapitre 6.

```
…
<eClassifiers xsi:type="ecore:EClass" name="Node">
       <eAnnotations>
              <details key="pattern" value="Visitor"/>
              <details key="role" value="abstractClass"/>
              <details key="instance" value="1"/>
       </eAnnotations>
       <eAnnotations>
              <details key="pattern" value="Bridge"/>
              <details key="role" value="abstractClass"/>
              <details key="instance" value="1"/>
       </eAnnotations>
       <eOperations name="TypeCheck">
              <eAnnotations>
                     <details key="pattern" value="Visitor"/>
                     <details key="role" value="abstractOperation"/>
                     <details key="instance" value="1"/>
              </eAnnotations>
              <eAnnotations>
                     <details key="pattern" value="Bridge"/>
                     <details key="role" value="abstractOperation"/>
                     <details key="instance" value="1"/>
              </eAnnotations>
       </eOperations>
…
</eClassifiers>
…
```

Figure 4.20. Exemple d'une classe avec deux annotations

4.6.4 Traitement des solutions

Nonobstant les problèmes reliés aux points de variation, les instances détectées par le solveur doivent être traitées pour regrouper celles qui font partie du même problème. En effet, pour notre exemple de la Figure 4.16, JSolver retourne

cinq (5) solutions montrées à la Figure 4.21, chaque solution est délimitée par les différentes lignes en pointillés. Ces résultats s'expliquent par deux raisons.

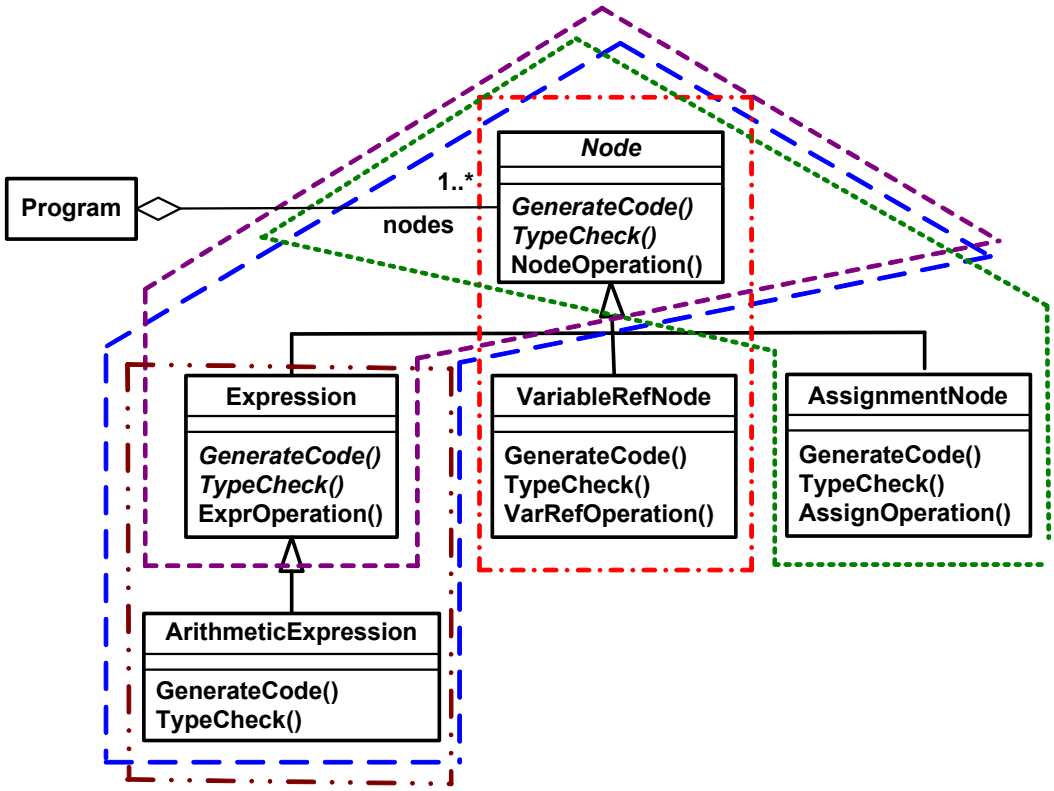


Figure 4.21. Instances retournées par le solveur

La première raison est reliée à un problème commun dans les approches de *pattern matching*. Nous illustrerons ce problème avec un exemple simple de recherche d'expressions régulières. Supposons que nous recherchons l'expression régulière a*b* dans un texte donné. Si le texte était, par exemple, la chaîne de caractères « cdfiaaabbxyz », l'expression a*b* correspondrait aux chaînes a, aa, aaa, b, bb, ab, aab, aaab, abb, aabb et aaabb. Un outil d'analyse doit être implémenté de façon à ce qu'il retourne la séquence la plus longue correspondant à l'expression régulière recherchée. De façon analogue, JSolver retourne tout fragment du modèle en entrée qui satisfait l'ensemble des contraintes associé à un modèle de problème donné; et dans les cas où il existe une relation récursive dans le modèle de problème, JSolver génère des chemins de différentes longueurs. Ceci explique pourquoi dans notre exemple de la Figure 4.21, JSolver identifie comme solutions distinctes les

chemins `ArithmeticExpression→Expression→Node`, `ArithmeticExpression→Expression`, et `Expression→Node`,

La seconde raison est reliée à un problème dû aux limitations de la version de JSolver que nous avons utilisée et elle explique pourquoi les chemins `Expression→Node`, `VariableRefNode→Node` et `AssignmentNode→Node` sont considérés comme des solutions différentes. En effet, pour avoir toutes les sous-classes d'une classe donnée en une fois, nous avons besoin de définir une variable dont la valeur serait un ensemble d'objets et donc le domaine serait un ensemble d'ensembles d'objets par opposition à un simple ensemble d'objets. Or, la version de JSolver que nous avons utilisée supporte seulement les tableaux d'ensembles numériques et elle nécessite de spécifier la taille d'un tableau dans la définition de la variable, ce qui n'est pas pratique dans notre contexte : nous ne connaissons pas à l'avance la taille de l'ensemble retourné des valeurs et, de plus, cette taille varie à l'intérieur du même fragment (i.e. dans notre exemple, `Node` a trois sous-classes et `Expression` a une sous-classe).

Nous avons donc eu besoin de traiter les solutions retournées par le solveur pour identifier le fragment le plus grand du modèle en entrée qui correspond à une seule et même instance d'un modèle de problème donné. Pour ce faire, nous récupérons les solutions au fur et à mesure qu'elles sont retournées par le solveur et nous les organisons en une forêt d'arbres en se basant sur les sous-chemins communs et la marque. Chaque arbre de la forêt correspond à une instance différente d'un problème. Nous discuterons du problème de regroupement des solutions et sa relation avec notre spécification des problèmes de conception au chapitre 6.

4.7 Conclusion

Nous avons présenté dans ce chapitre une approche semi-automatique pour marquer les modèles UML. En particulier, notre approche permet :

1. la détection, dans des modèles UML, des instances de modèles de problèmes résolus par les patrons de conception; et
2. le marquage des instances détectées.

Le marquage d'une instance détectée d'un patron donné permet une mise en œuvre automatique de la transformation inhérente à l'application du patron concerné.

En utilisant les techniques de satisfaction de contraintes, notre approche dissocie le modèle de problème recherché de l'algorithme de détection. De plus, notre approche peut être aussi utilisée pour détecter des instances des solutions proposées par les patrons de conception comme elle peut être utilisée pour détecter d'autres types de patrons.

Chapitre 5

Représentation des transformations

Nous décrivons dans ce chapitre notre approche pour la représentation et la mise en œuvre des transformations inhérentes à l'application des patrons de conception. Nous présenterons à la section 5.1 notre première tentative de représentation des transformations qui était basée sur un modèle structurel, et nous discuterons des limites de cette représentation. Nous introduirons les exigences pour représenter les transformations associées aux patrons de conception à la section 5.2. Nous survolerons ensuite les récentes approches de transformation de modèles (section 5.3). Nous présenterons notre approche pour la représentation des transformations à la section 5.4 et nous en décrirons l'implémentation à la section 5.5 avant de conclure à la section 5.6.

5.1 Un modèle structurel pour la représentation des transformations

5.1.1 Principes

En nous basant sur notre représentation des patrons, l'application d'un patron de conception consiste à transformer une instance du modèle de problème résolu par le patron considéré en une instance du modèle de la solution qu'il propose. Donc, indépendamment de l'instance sur laquelle elle s'applique, la transformation inhérente à un patron est spécifiée de façon déclarative par la correspondance entre les éléments du modèle de problème (MP) et ceux du modèle de solution (MS). Nous

avons représenté cette transformation sous forme d'une structure récursive $T(MP \rightarrow MS) = <MP, MS, \{T(A_i \rightarrow B_j) \mid A_i \in MP \text{ et } B_j \in MS\}>$. Lorsque A_i est absent (`null`) dans $T(A_i \rightarrow B_j)$ cela signifie que B_j est un nouvel élément qu'on ajoute. Inversement, lorsque B_j est `null` alors A_i est à supprimer. Cette structure peut être spécifiée de façon visuelle en établissant des liens dans un éditeur graphique entre éléments du modèle de problème et éléments du modèle de solution.

5.1.2 Représentation structurelle des transformations

Nous avons donc représenté les structures décrivant les transformations par un méta-modèle. Une instance de ce méta-modèle spécifie le modèle des transformations associées à un patron. En fait, ce modèle de transformations établit des correspondances entre les entités du modèle de problème associé à un patron et les entités du modèle correspondant de solution. Par exemple, dans le cas du patron Visiteur, le modèle de transformation doit établir une correspondance entre, d'un côté, l'entité `AbstractClass` du modèle de problème et, de l'autre côté, les entités `Element` et `Visitor` du modèle de la solution (Figure 5.1). Cela signifie que `AbstractClass` est la source pour ces deux entités de la solution. Le modèle de transformation établit aussi une correspondance entre `ConcreteClass` et `ConcreteElement` et une correspondance entre, d'une part, la relation `inherits_from` entre `ConcreteClass` et `AbstractClass` et, d'autre part, la relation `inherits_from` entre `ConcreteElement` et `Element`. Et ainsi de suite.

En plus, le modèle de correspondances inclut les propriétés et informations nécessaires pour guider l'application du patron à un modèle source dûment marqué. En effet, le modèle source marqué ainsi que le modèle de correspondances sont fournis en entrée à un moteur générique qui exécute la transformation sur le modèle marqué en procédant de façon récursive basée sur le modèle de correspondances (Figure 5.2).

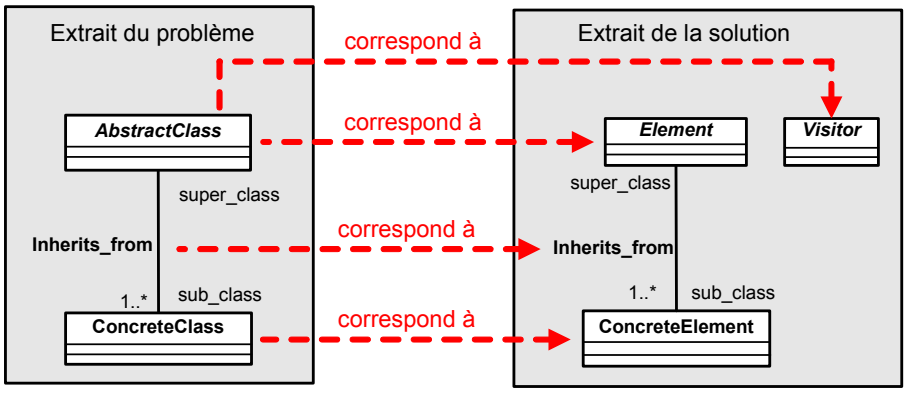


Figure 5.1. Exemple de correspondances entre entités du modèle de problème et celles du modèle de la solution.

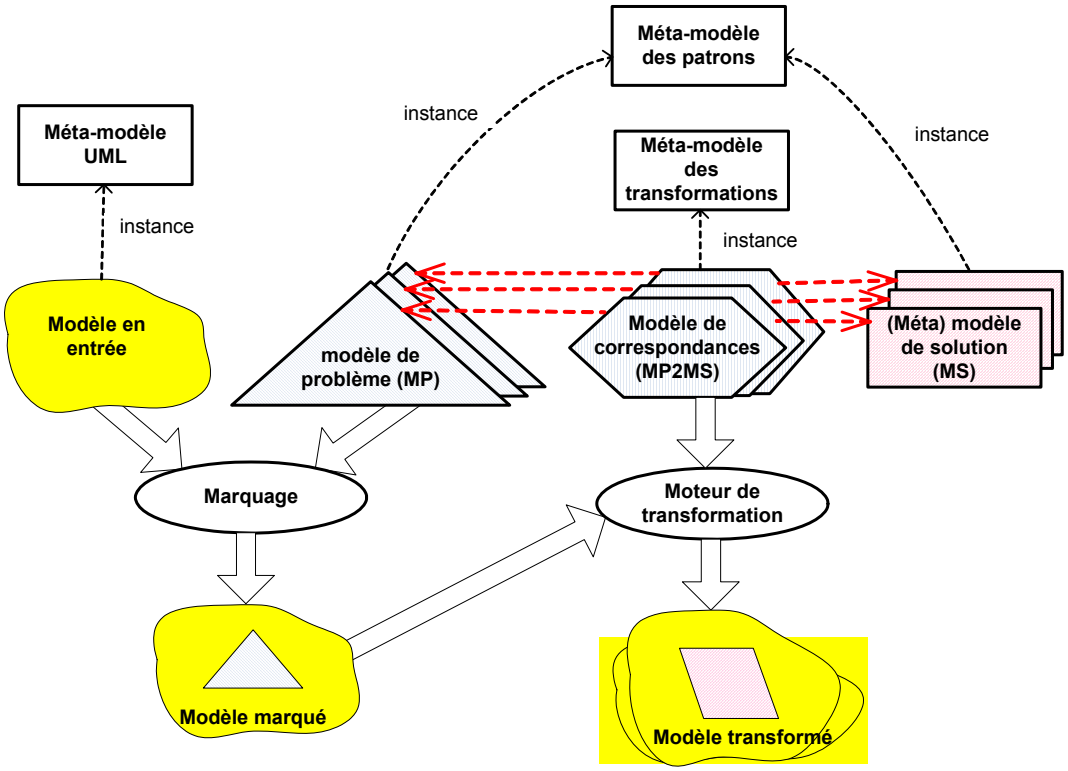


Figure 5.2. Transformations basées sur des modèles de correspondances entre problèmes et solutions.

La Figure 5.3 montre le méta-modèle des transformations permettant d'instancier des modèles de correspondances entre modèles de problèmes et modèles de solutions. Un modèle de correspondances est une instance de la classe `ModelMapping` laquelle est une agrégation d'instances des classes

`AssociationMapping` et `ClassMapping`. Une correspondance entre deux classes est une instance de `ClassMapping` laquelle est une agrégation de correspondances entre opérations (`OperationMapping`) et de correspondances entre attributs (`AttributeMapping`), etc. Toutes les correspondances héritent de la super-classe `EntityMapping`. Cette dernière associe au plus un élément source issu du modèle du problème à un élément cible appartenant au modèle de la solution. Elle décrit aussi la façon dont l'entité source doit être transformée par l'intermédiaire de son attribut `Transformation`. La suppression (respectivement l'ajout) d'un élément est représentée par l'absence de l'élément cible (respectivement l'élément source). Les classes `NamedElement` et `Model` font partie du méta-modèle UML.

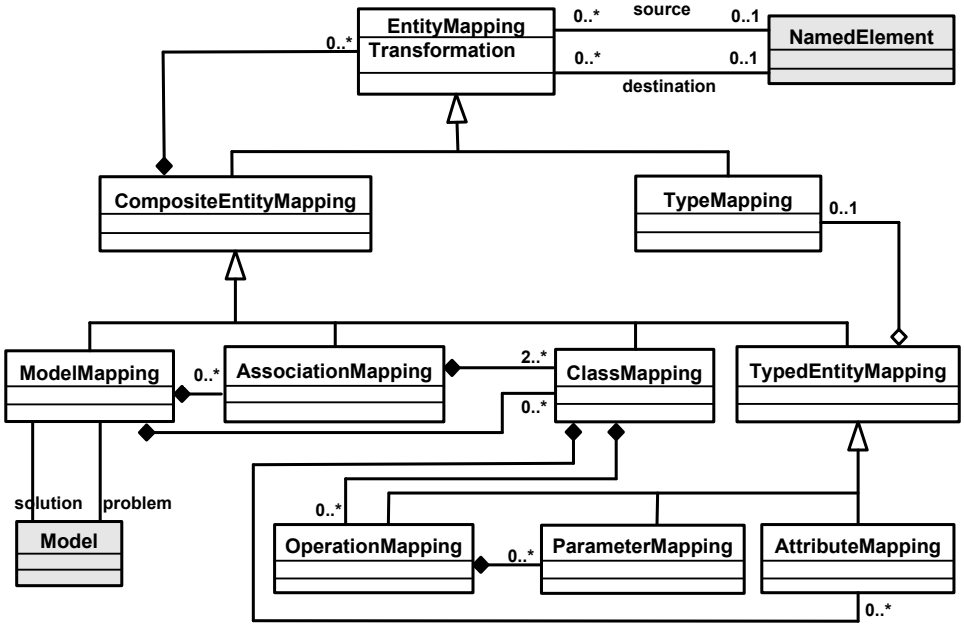


Figure 5.3. Méta-modèle des transformations

Une instance de `OperationMapping` peut comprendre des instances de la classe `ParameterMapping` lorsque les opérations qu'on essaye de mettre en correspondance ont des paramètres. Un paramètre et un attribut sont des entités qui possèdent un type. Une opération, quant à elle, peut avoir un type de retour. Dans le cas où une opération a un type de retour, il faut spécifier ce qu'il advient de ce type de retour lors de la transformation de cette opération. De même, il faut spécifier dans le cadre d'une correspondance entre attributs (respectivement paramètres) la

correspondance entre types. Pour cette raison, nous avons introduit dans notre méta-modèle la classe `TypedEntityMapping`. Cette dernière représente une généralisation de toutes les correspondances entre des entités typées (attribut, opération, paramètre). Cette classe peut contenir des instances de la classe `TypeMapping` qui spécifie la correspondance entre les types. La multiplicité « 0..1 » de la relation de composition existant entre `TypedEntityMapping` et `TypeMapping`, spécifie que la correspondance de deux entités typées a au plus une correspondance de types.

Il est très important de spécifier la portée de la correspondance (*mapping*). La portée d'une correspondance fait référence au contexte dans lequel cette correspondance est valable. La relation de correspondance entre un attribut du modèle de problème et un attribut du modèle de solution, par exemple, ne peut exister que dans le contexte de la correspondance entre les classes auxquelles appartiennent ces attributs. De la même façon, la relation de correspondance entre un paramètre du modèle de problème et un paramètre du modèle de solution ne peut exister que dans le contexte d'une correspondance entre les opérations dont ils sont des paramètres.

Un extrait d'une instance du méta-modèle de transformations est montré à la Figure 5.4. C'est un extrait du modèle de transformations associé au patron Visiteur. Le modèle de transformations, appelé ici `VisitorMapping` est une instance de `ModelMapping`. Il contient plusieurs instances de `ClassMapping` qui spécifient les transformations à appliquer sur les différentes classes du modèle de problème. Nous montrons dans cet extrait deux instances de `ClassMapping`: la classe `AbstractClass2Element` qui associe la classe `AbstratClass` du modèle de problème à la classe `Element` du modèle de solution, et la classe `AbstractClass2Visitor` qui associe la classe `AbstractClass` à l'interface `Visitor` du modèle de solution. De plus, ces deux correspondances spécifient comment certaines propriétés des entités `Element` et `Visitor` sont déduites de la classe `AbstractClass`.

Pour pouvoir utiliser les fonctionnalités de EMF, notamment la sérialisation sous format XMI, nous avons *aussi* implémenté le méta-modèle de transformations (Figure 5.3) comme une extension au méta-modèle EMF. Nous ne nous attarderons

pas sur les détails de cette implémentation car ils sont similaires à ceux donnés lors de la description de l'implémentation de notre méta-modèle pour la spécification des patrons (section 3.6 du chapitre 3).

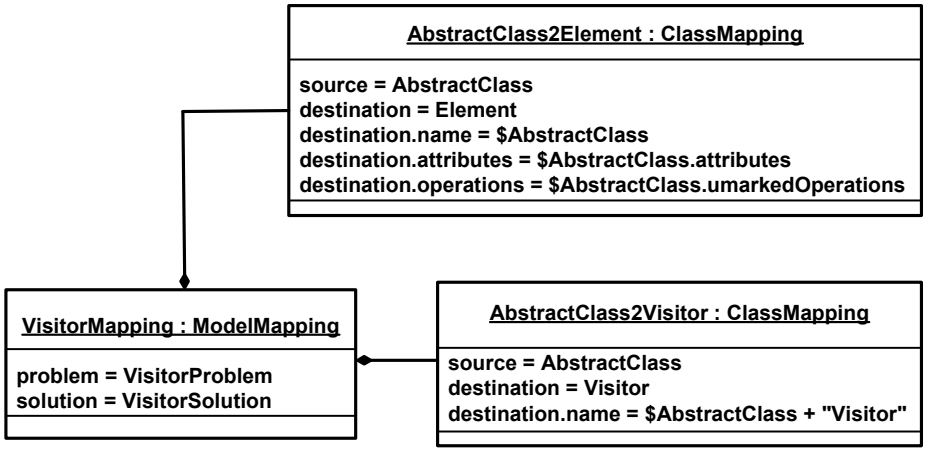


Figure 5.4. Extrait du modèle de transformations associé au patron « Visiteur »

Pour appliquer les transformations, nous avons implémenté un moteur/interprètre qui prend en entrée un modèle objet dûment marqué par un ou plusieurs instances de modèles de problèmes, et les modèles de transformations (Figure 5.2). Le moteur produit en sortie le modèle à l'entrée tel que transformé par le patron. Il procède d'une façon descendante où la transformation d'un agrégat source génère d'abord une coquille vide de l'agrégat destination, lui applique les transformations locales, puis applique récursivement les transformations sur ses composantes. Par exemple, au plus haut niveau, le transformateur génère d'abord un modèle destination vide, puis commence à transformer les classes. Même scénario au niveau des classes : on génère d'abord une classe cible vide, puis on génère les attributs et les opérations. Et ainsi de suite.

Cette démarche nous a permis de représenter et d'appliquer les transformations inhérentes aux patrons les plus simples—moyennant quelques acrobaties—mais s'est avérée inadéquate pour des patrons le moindrement plus complexes.

5.1.3 Problèmes reliés à la représentation structurelle des transformations

Le problème avec notre représentation des transformations est que le méta-modèle de transformations est incomplet. En effet, ce méta-modèle permet de spécifier uniquement des transformations de type 0..1 à 0..1 (un élément source associé à un élément cible). Or, en réalité les correspondances ne sont pas toujours de type 0..1 à 0..1. En effet, une des transformations les plus communes aux patrons de conception est celle qui produit à partir d'une classe du modèle de problème une paire d'entités du modèle de solution, lesquelles sont une classe et une interface où la classe implémente l'interface (e.g. Pont, Proxy, Décorateur).

Un autre problème auquel nous avons été confrontés est que le méta-modèle de transformations permet d'associer uniquement des entités de même type, i.e une classe à une classe, une opération à une opération, etc. Or, une classe du modèle de problème ne correspond pas toujours à une classe dans le modèle de solution. Idem pour les opérations. Dans l'exemple du patron Visiteur, les opérations `AbstractOperation` du modèle de problème (Figure 3.3) correspondent à des classes `ConcreteVisitor` dans le modèle de solution (Figure 3.4) et, inversement, les classes `ConcreteClass` (Figure 3.3) correspondent, entre autres, aux opérations `Visit` et `ConcreteVisit` (Figure 3.4).

Même si nous complétions le méta-modèle de transformations pour prendre en charge les problèmes que nous venons de citer, l'avantage de la représentation structurelle graphique est plus que douteux. En effet, la mise en œuvre des transformations est assez difficile car les correspondances sont faites en se basant sur la structure du méta-modèle. En fait, les transformations sont imbriquées selon la hiérarchie de contenu décrite par le méta-modèle (un package contient des classes, une classe contient des opérations, etc.). Or l'étendue ne fournit pas de contexte suffisant pour spécifier les correspondances. La notion d'étendue signifie qu'un élément `X` faisant partie d'un élément `Y` correspondra à un élément `X'` faisant partie d'un élément `Y'`. Dans le cas du patron Visiteur, par exemple, `AbstractOperation`

est une opération qui fait partie de `AbstractClass` mais son correspondant `ConcreteVisitor` ne fait pas partie des correspondants de `AbstractClass`.

En outre, l'application d'un patron à un modèle ne devrait se faire que lorsqu'on découvre un ensemble particulier d'éléments et de relations entre ces éléments dans le modèle considéré, i.e. une instance d'un modèle de problème. Nous avons donc besoin de transformations contextuelles (*context sensitive*). En effet, une transformation contextuelle (a R x $\rightarrow$ a R y) permet, d'une part, de décrire plus complètement les conditions dans lesquelles une entité y se subsitue à une entité x et, d'autre part, de préserver les relations qu'une entité transformée avait avec les autres entités du modèle.

5.2 Exigences pour un langage de représentation des transformations

Notre exigence initiale pour un langage de transformation était de spécifier de façon déclarative les transformations inhérentes aux patrons tout en utilisant les standards de l'OMG, notamment le langage UML. Cependant, à partir de notre expérience pratique avec les transformations de modèles et en nous basant sur la spécification QVT (*Query* / *View* / *Transformation*) (QVT, 2005) de l'OMG, nous avons pu identifier d'autres exigences auxquelles notre langage de transformation doit répondre pour permettre l'automatisation de l'application des patrons.

5.2.1 Exigences selon le standard QVT de MDA

La spécification QVT vise à définir un langage standard pour manipuler les modèles. Elle s'appuie sur le standard MOF pour la méta-modélisation et le standard OCL pour exprimer des contraintes formelles sur les modèles. Les exigences auxquelles répondait la spécification QVT sont :

- Disposer d'un langage qui permette d'exprimer des requêtes sur les modèles. Le résultat d'une requête sur un modèle est un ensemble d'éléments instances des types définis par le modèle considéré.

- Permettre d'extraire des vues à partir des méta-modèles. Une vue est un modèle dérivé d'un autre. Elle peut être le résultat d'une requête ou d'une transformation.
- Pouvoir spécifier de façon déclarative les transformations par le biais de correspondances entre éléments d'un méta-modèle source et éléments d'un méta-modèle cible. Les deux méta-modèles doivent être conformes à MOF.
- Le langage de transformation doit être suffisamment expressif pour pouvoir décrire complètement la façon dont les éléments cibles sont déduits des éléments sources.
- Permettre des transformations incrémentales La modification d'un modèle source doit être propagée vers le modèle cible et inversement. Seules les règles de transformation qui incluent les éléments modifiés dans un modèle s'exécutent.

D'autres propriétés intéressantes mais pas obligatoires d'un langage de transformation ont été introduites dans l'appel à proposition RFP/QVT (Gardner et al., 2003). En effet, il est souhaitable qu'un langage permette de spécifier des transformations qui soient bidirectionnelles. Il est également intéressant que les transformations aient un comportement transactionnel. Finalement, le langage doit fournir des mécanismes qui permettent la réutilisation des transformations.

5.2.2 Nos exigences

Notre expérience avec les transformations de modèles nous a permis d'identifier des exigences supplémentaires que notre langage de transformation doit satisfaire pour supporter l'application automatique des patrons de conception. Nous listons ci-après quelques exigences essentielles :

- Le langage doit nous permettre d'exprimer des transformations complexes. Une transformation est dite complexe si elle construit des structures dans le modèle cible qui ne correspondent pas directement à des éléments individuels du modèle source (Gardner et al., 2003). Dans notre cas, il y a très peu de transformations qu'on peut exprimer par des productions de type $A \rightarrow B$. La

majorité des transformations doivent être exprimées par des productions dont la partie gauche contient plusieurs entités et des relations entre ces entités. Autrement dit, on doit pouvoir exprimer des transformations de type plusieurs à plusieurs.

- Notre langage doit nous permettre d'établir des correspondances entre des entités de différents types. Par exemple, une classe peut être associée à une opération ou inversement (e.g. le patron Visiteur).
- La traçabilité des transformations est très importante. Cela nous permettra de vérifier le résultat obtenu par l'application d'un patron sur un modèle et établir certaines contraintes sur les transformations futures pouvant être faites sur le modèle en question.

5.2.3 Vers une approche de transformation à base de règles

Fondamentalement, la transformation de modèles objets—ou graphiques en général—est un problème de transformation de graphes typés (e.g. (Grunske et al., 2005), (Mens et al., 2005)). En effet, plusieurs transformations associent une configuration d'éléments du modèle source à une configuration d'éléments du modèle cible et ne peuvent être décomposées davantage sans perdre de leur sémantique. Pour remédier aux problèmes décrits à la section 5.1.3, nous nous sommes intéressés à la problématique de représentation des transformations de graphes typées. Nous avons choisi d'utiliser les règles de production pour représenter les transformations élémentaires dans le contexte d'un environnement hybride objets-règles. Cette solution a plusieurs avantages sur les approches existantes aux transformations de graphe, tant sur le plan pratique que sur le plan théorique. Sur le plan pratique, nous avons utilisé des technologies éprouvées (i.e. EMF pour les modèles et ILOG JRules™ pour les règles) nous épargnant d'importants efforts pour développer l'infrastructure de transformation. Sur le plan théorique, la technologie des moteurs d'inférence nous permet de régler de façon élégante plusieurs problèmes liés au contrôle d'exécution des règles de transformations.

Avant de présenter notre nouvelle approche pour les transformations (section 5.4), nous présentons un survol rapide des récentes approches de transformation de modèles en particulier celles basées sur les transformations de graphes.

5.3 Survol des approches de transformation de modèles

Plusieurs approches ont été proposées pour la mise en œuvre des transformations de modèles. Parmi celles-ci, les plus courantes sont les approches de manipulation directe, les approches basées sur les transformations de graphe et les approches relationnelles. Les approches de manipulation directe (e.g. (JMI, 2002), (Jamda, 2003) et (EMF, 2005)) offrent une représentation interne du modèle et une API pour manipuler cette représentation (Czarnecki et Helsen, 2003). Dans ces approches, l'utilisateur doit implanter les règles de transformation en utilisant un langage de programmation. Les approches relationnelles et les approches basées sur les transformations de graphe permettent des spécifications plus abstraites des transformations.

Les approches relationnelles sont des approches déclaratives où les transformations sont spécifiées par des relations mathématiques (Akehurst et Kent, 2002) (Appukuttan et al., 2003). La spécification QVT en est un cas particulier. Une transformation est donc décrite par des relations définies entre des éléments sources et des éléments cibles, et par un ensemble de contraintes. En fait, les relations sont encodées sous forme de modèle structurel (Akehurst et Kent, 2002), elles ne sont pas exécutables et sont bidirectionnelles. Mais, certaines approches leur associent une sémantique exécutable (Appukuttan et al., 2003). Cependant, les relations sont spécifiées en se basant sur la structure des modèles source et cible. Dans ce sens, les approches relationnelles sont similaires à notre première tentative de représentation des transformations et elles présentent les mêmes problèmes et limitations.

Les systèmes de transformations de graphes permettent de spécifier les transformations de façon visuelle et de les mettre en œuvre. Leurs fondements mathématiques fournissent une base rigoureuse et puissante pour spécifier les transformations de modèles. De plus, il est plus intuitif d'utiliser des graphes pour

représenter les modèles. Pour cela, plusieurs approches pour les transformations de modèles (AGG (Rudolf et Taentzer, 1999), GReAT (Agrawal et al, 2005), FUJABA (Grunske et al., 2005) et VIATRA (Balogh et Varró, 2006)) s'appuient sur la théorie des transformations de graphe. Notre nouvelle approche pour la spécification des transformations (section 5.4) s'apparente plus à ces approches même si elle n'offre pas de notation graphique. Avant de présenter certaines de ces approches (section 5.3.2), nous rappelons d'abord brièvement les concepts de graphes typés, de grammaire de graphes et de transformation de graphes. Les concepts de graphes typés et de transformation de graphes ont été déjà introduits au chapitre 4 (section 4.2).

5.3.1 Graphes typés et grammaires de graphes

Un graphe est défini par un ensemble de nœuds et d'arêtes telles que chaque arête relie deux nœuds appartenant à l'ensemble des nœuds du graphe. Un graphe typé est un graphe dont les nœuds et les arêtes sont des instances de nœuds et d'arêtes d'un graphe de types. Une règle de transformation de graphe appelée aussi production `p : L → R` est composée d'une paire de graphes typés `L` (LHS : *left hand side*) et `R` (RHS : *right hand side*). Cela signifie que la règle `p` a comme pré-condition le graphe `L` et comme post-condition le graphe `R`. L'idée de base est de considérer une production comme une description finie d'un ensemble potentiellement infini de dérivations (Corradini et al., 1996). Une production `p : L → R` est appliquée à un graphe `G` lorsque le patron représenté par le graphe `L` est détecté dans `G` ; il est alors remplacé par le patron du graphe `R`. Un exemple de transformation de graphe a été présenté au chapitre 4 (section 4.2.3).

Les grammaires de graphes sont des généralisations des grammaires textuelles. Une grammaire de graphes est composée d'un ensemble de graphes initiaux et d'un ensemble de règles de réécriture ou de transformation de graphes. Finalement, un système de transformation de graphes est défini par un graphe de type et un ensemble de règles de transformation. Une séquence de dérivations est produite par des règles de transformation pouvant être appliquées de façon séquentielle ou parallèle. L'ensemble des séquences de dérivations possibles forment une

spécification complète de la sémantique du comportement d'un système (Engels et al., 1997).

5.3.2 Transformations de modèles basées sur les grammaires de graphes

Les approches de transformation de modèles basées sur les transformations de graphes sont des approches basées sur les règles (Taentzer et al.,2005). Ces approches se distinguent d'abord par la représentation des modèles ou graphes qu'ils permettent de manipuler. Elles se distinguent aussi par la façon dont les règles sont spécifiées et par le support offert pour permettre le contrôle d'application des règles. Dans AGG (*Attributed Graph Grammar*), on utilise un système de transformation de graphes i.e. un graphe de types et un ensemble de règles (Rudolf et Taentzer, 1999). Les modèles sont spécifiés par des graphes attribués typés. Une règle est composée d'une paire de graphes typés et peut inclure des calculs d'attributs. Des filtres peuvent être associés aux règles indiquant des conditions sous lesquelles les règles ne s'appliqueraient pas. AGG permet de définir des couches de règles pour fixer l'ordre d'application des règles. Ces couches de règles permettent de définir un certain flux de contrôle des transformations. La base formelle d'AGG permet d'offrir un support de validation incluant les critères de terminaison, la vérification de la cohérence des graphes et des systèmes de transformation de graphes, et la détection de conflits et de dépendances entre règles de transformation.

Le système GRreAT (*Graph Rewriting and Transformation Language*) définit trois langages: un langage pour spécifier des patrons, un langage pour spécifier les règles de transformations et un langage pour le contrôle d'application des transformations (Agrawal et al., 2005). Le langage pour spécifier les règles utilise les méta-modèles source et cible pour décrire les parties LHS et RHS d'une règle et un ensemble d'actions. Pour le contrôle d'application des règles, GReAT fournit des opérateurs de séquence, de composition et de branchement, et il permet l'imbrication des règles. Enfin, GReAT utilise OCL pour spécifier des contraintes sur les modèles

résultants des transformations permettant ainsi de vérifier leur conformité aux méta-modèles les décrivant dans les règles.

VIATRA (Balogh et Varró, 2006) se base sur les transformations de graphes et les machines abstraites à états (ASM : *abstract state machine*). Les règles élémentaires de transformation sont décrites par des règles de transformation de graphes. Les ASMs fournissent toutes les structures de contrôle nécessaires pour spécifier des règles de transformation plus complexes à partir des règles élémentaires. VIATRA permet aussi de spécifier des patrons de graphes en utilisant des contraintes et des conditions sur les modèles et d'intégrer des appels à des méthodes natives Java dans les transformations. Ceci en fait un langage très puissant et expressif.

FUJABA (*From UML to Java And Back Again*) utilise des graphes typés appelés les *story diagrams* qui combinent la partie gauche et la partie droite d'une règle de transformation (Grunske et al., 2005). Les graphes des types sont spécifiés par des diagrammes de classes UML. Une règle est exprimée par un diagramme de collaboration d'UML où des stéréotypes permettent de marquer les éléments à ajouter ou à supprimer pendant l'application de la règle considérée. Les structures de contrôle qui gèrent l'ordre d'exécution des règles sont spécifiées en utilisant les diagrammes d'activité d'UML. Des conditions négatives d'application peuvent être associées aux règles. Les *story diagrams* permettent d'exprimer des transformations complexes, mais ils sont difficiles à lire.

De manière générale, les exemples de transformations présentées par les différentes approches ne sont pas assez complexes. Cela donne une idée limitée de chacune des approches et ne nous permet pas de savoir s'il est possible ou non d'exprimer toutes les transformations possibles d'un modèle en utilisant une seule approche. Selon Gardner et al. (2003), il faudra combiner plusieurs approches pour pourvoir exprimer n'importe quel type de transformation. Il serait aussi bénéfique de combiner un langage de transformations de graphes avec un langage textuel de contraintes (Mens et al., 2005). De plus, pour le moment, il n'existe pas de critères permettant de confirmer quel type d'approche est le plus adapté à un type d'applications.

L'approche que nous proposons ne fournit pas de notation graphique pour les règles. Cela dit, les représentations graphiques proposées ne sont pas toujours intutives (e.g. FUJABA et GReAT). De plus, une des préoccupations majeures d'une approche basée sur les règles est le contrôle d'application des règles (Baresi et Heckel, 2002). En effet, chacune des approches que nous venons de décrire (AGG, GReAT, FUJABA et VIATRA) propose ses propres mécanismes pour contrôler l'application des règles. L'avantage de notre approche est que nous n'avons pas à contrôler l'exécution des règles. D'une part, nous nous fions aux techniques de contrôle du moteur à base de règles, à savoir: a) exécution unique d'une règle pour un n-uplet donné, b) gestion des priorités, c) flux de règles, d) chaînage avant, dans le sens qu'une règle crée les conditions pour une autre. D'autre part, nous utilisons dans la spécification de nos règles des jointures et des variables de correspondance qui permettent indirectement de contrôler l'exécution des règles. Ces notions sont abordées à la prochaine section.

5.4 Notre approche pour la représentation des transformations

5.4.1 Principes

Notre approche pour la spécification des transformations peut être vue comme une représentation textuelle de grammaires de graphe. En effet, la transformation inhérente à l'application d'un patron de conception peut être représentée par une règle de production. Les parties LHS (*left hand side*) et RHS (*right hand side*) de cette règle correspondront respectivement au modèle du problème (MP) et au modèle de solution (MS) associés au patron considéré. Cette règle aurait la forme de : $\rho(A_1 \ldots A_m) \Rightarrow \varphi(B_1 \ldots B_n)$ où les A_i et B_j sont des symboles désignant respectivement les entités du modèle de problème et les entités du modèle de solution. ρ représente l'ensemble des relations entre les entités du modèle de problème (A_i) et φ représente l'ensemble des relations entre les entités du modèle de solution (B_j).

Si chaque patron devait être représenté par une seule règle, cette règle de transformation serait difficile à formuler et à comprendre. De plus, cette démarche ne nous permettrait pas de partager et de réutiliser certaines transformations élémentaires qui sont communes aux patrons de conception. Par conséquent, nous avons essayé de décomposer cette règle complexe de transformation en un ensemble de règles élémentaires tout en préservant la sémantique de la règle complexe. Pour ce faire, nous proposons deux heuristiques. La première heuristique permet de décomposer une règle de transformation en simplifiant sa partie RHS; la seconde heuristique permet de simplifier la partie LHS de la règle.

Nous illustrerons dans les sections suivantes les deux heuristiques de simplification en utilisant le patron Visiteur.

5.4.2 Simplification de la partie RHS d'une règle de transformation

Prenons l'exemple de la transformation inhérente à l'application du patron Visiteur. Cette transformation peut être décrite en utilisant une seule règle illustrée par le diagramme de la Figure 5.5. Notons que ce diagramme ne représente que les entités et leurs relations: il ne montre pas comment les propriétés des entités de la partie RHS sont calculées à partir de celles des entités de la partie LHS.

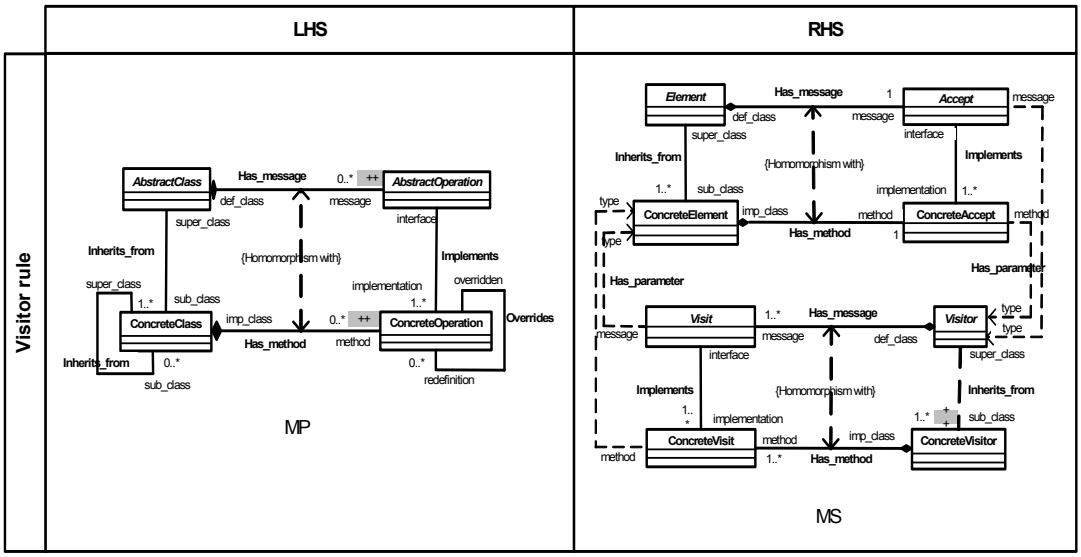


Figure 5.5. Règle de transformation associée au patron Visiteur.

Notre première heuristique décompose la règle de la Figure 5.5 en deux ensembles de règles :

- Un ensemble de règles qui génèrent les éléments de la solution, un élément à la fois. La partie droite (RHS) de ces règles contient un seul élément.
- Un ensemble de règles qui génèrent les relations entre les éléments générés par le premier ensemble de règles, une relation à la fois.

Autrement dit, nous générons le modèle ou graphe de solution en générant les nœuds, un par un, et ensuite en générant les relations entre ces nœuds. La Figure 5.6 illustre quelques unes des règles obtenues par l'application de cette première heuristique à la règle de transformation associée au patron Visiteur. La partie LHS de ces règles est la même que celle de la règle initiale, i.e. elle comprend l'ensemble du modèle de problème. Par exemple, les deux premières règles R_1 et R_2 génèrent, respectivement, les entités `Element` et `ConcreteElement` du modèle de solution à partir du modèle de problème, alors que la règle R_i génère la relation d'héritage `inherits_from` existant entre ces deux entités.

Cette heuristique préserve la sémantique puisque nous ne perdons aucune partie du contexte d'application des règles dans la simplification. De façon générale, si le modèle de solution contient M entités et N relations entre ces entités, notre première heuristique génère M+N règles de transformations : M règles pour générer les entités et N règles pour générer les relations.

5.4.3 Simplification de la partie LHS d'une règle de transformation

Notre seconde heuristique vise à simplifier la partie LHS des règles de transformation en gardant seulement le contexte nécessaire et suffisant pour générer correctement le modèle de solution. En effet, certaines des règles obtenues après application de la première heuristique peuvent être simplifiées davantage lorsque les éléments qu'elles génèrent (i.e. entités de la solution) dépendent d'un sous-ensemble des entités sources (i.e. entités du problème) et non pas de toutes les entités sources. Il reste à déterminer l'ensemble des entités et des relations dont on a besoin pour générer une entité du modèle cible.

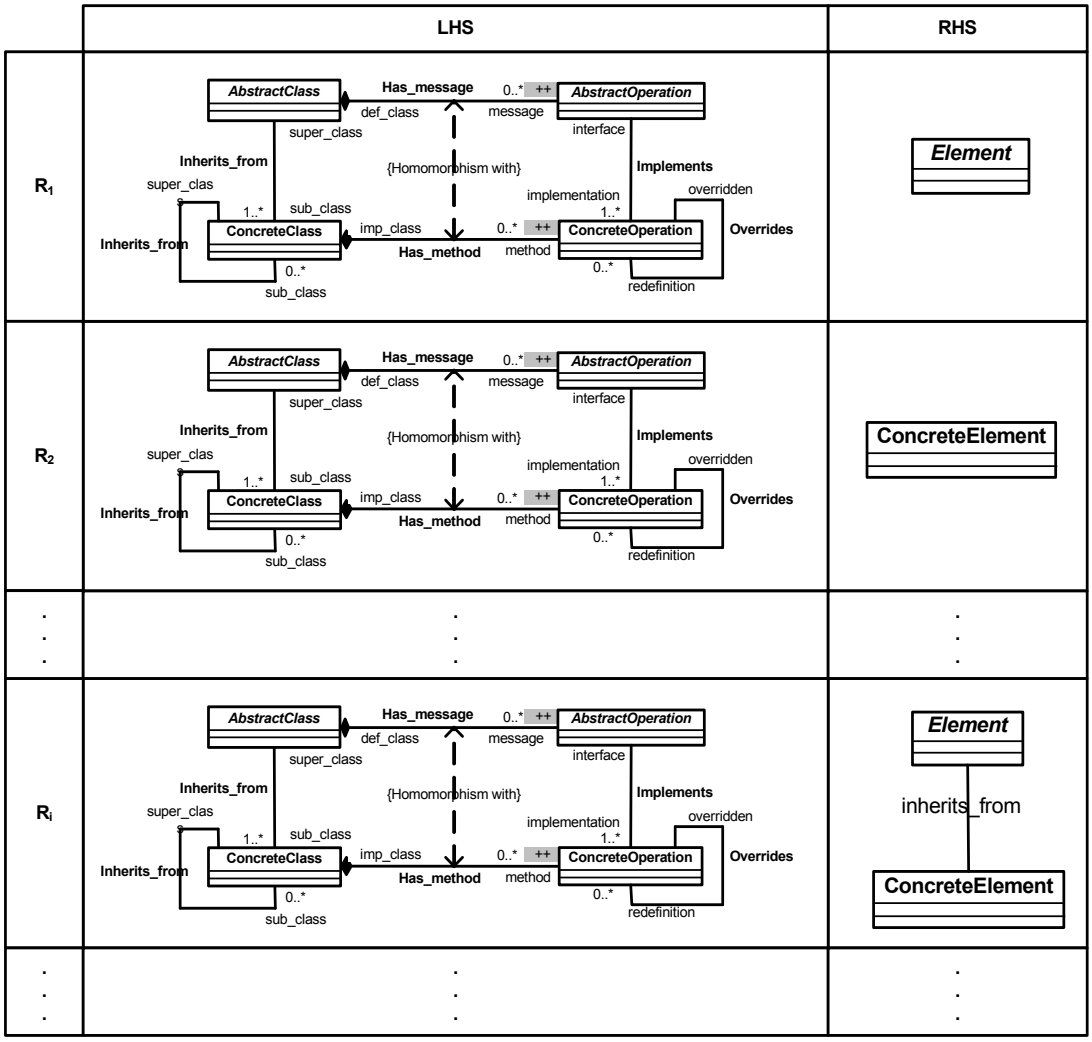


Figure 5.6. Règles de transformation obtenues par application de la première heuristique à la règle de la Figure 5.5.

Considérons la règle R_1 de la Figure 5.6. Même si cette transformation a lieu dans le contexte d'une occurrence d'une instance du problème résolu par le patron Visiteur, la classe `Element` générée dans le modèle de la solution, dépend uniquement de la classe du modèle en entrée qui correspond à l'entité `AbstractClass` du modèle de problème, i.e. la classe du modèle en entrée qui a été marquée comme étant `AbstractClass`. En effet, les propriétés de la classe `Element` sont calculées à partir des propriétés de la classe `AbstractClass`, e.g. le nom de la classe `Element` est le nom de la classe `AbstractClass` possiblement augmenté d'un prefixe ou suffixe. En fait, lorsque les propriétés (e.g. nom, attribut) d'une entité A_i de la partie LHS

contribuent au calcul d'une propriété de l'entité B_j de la partie RHS alors A_i fait partie du contexte indispensable à la génération de B_j et doit figurer dans la partie LHS de la règle qui produit B_j. L'ensemble de ces A_i représente le contexte nécessaire (*minimal*) pour la génération de B_j.

Le calcul du contexte *miminal* d'une règle de transformation est assez facile. Cependant, le contexte minimal n'est pas toujours suffisant pour garantir une exécution correcte de la règle. Par exemple, notre processus de marquage pourrait identifier plusieurs classes concrètes dans le modèle en entrée, mais seules celles héritant de la classe abstraite seront affectées par le patron. Ainsi, si nous n'associons pas un contexte suffisant à nos règles, elles pourraient transformer des entités non affectées par le patron, i.e. des entités ne faisant pas partie de l'instance du problème.

Ainsi, nous définissons, pour une règle donnée, le contexte *suffisant* comme étant l'ensemble minimal des éléments de la partie LHS qui permettraient de générer correctement l'entité B_j de la partie RHS, i.e. lorsqu'un modèle proprement marqué qui contient une seule instance du patron est transformé par l'application des règles ainsi simplifiées, il est transformé correctement. Cet aspect est relié à notre discussion au chapitre 6 (section 6.3.2). Le contexte suffisant est plus grand que le contexte minimal mais il est plus petit que le contexte représenté par le modèle intégral du problème. Nous ne disposons pas de moyens précis pour déterminer le contexte suffisant à priori. L'expérience et l'expérimentation devraient nous aider à raffiner cette deuxième heuristique.

L'application de cette heuristique aux règles R_1, R_2 et R_i de notre exemple (Figure 5.6), donne les règles montrées à la Figure 5.7. La règle R_i, par exemple, permet de générer la relation d'héritage entre les entités `ConcreteElement` et `Element` à partir de la relation d'héritage existant entre `ConcreteClass` et `AbstractClass`. Donc, le contexte suffisant de R_i contient autant les entités `ConcreteClass` et `AbstractClass` que la relation d'héritage entre elles.

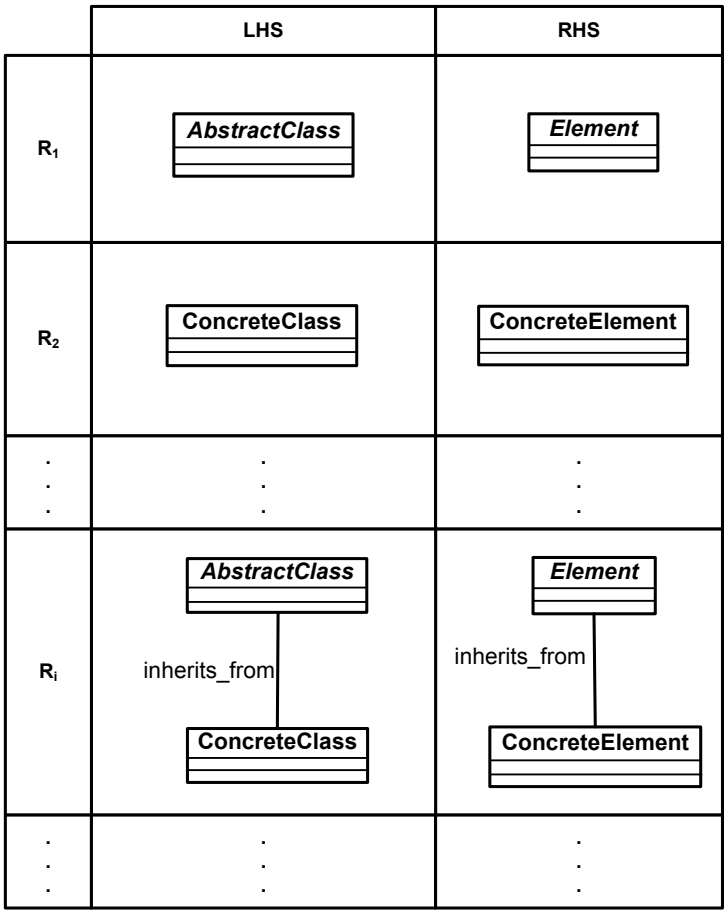


Figure 5.7. Règles de transformation obtenues par application de la seconde heuristique aux règles de la Figure 5.6.

5.4.4 Autres propriétés des règles de transformation

Reprenons l'exemple de la règle R_i vue précédemment. En pratique, la partie LHS de la règle R_i inclut aussi les entités `Element` et `ConcreteElement` pour lesquelles la règle R_i doit générer une relation d'héritage. En fait, le contexte d'application de certaines règles peut comprendre aussi bien une partie du modèle de problème qu'une partie du modèle de solution. Ainsi, comme le montre la Figure 5.8, la prémisse de la règle R_i est exprimée en termes des entités du modèle source (`AbstractClass` et `ConcreteClass`) et des entités du modèle cible qui leur ont été associées (`Element` et `ConcreteElement`). Autrement dit, la relation d'héritage existant entre `ConcreteClass` et `AbstractClass`, est transformée par la règle R_i en une relation d'héritage entre R_2`(ConcreteClass)` et R_1`(AbstractClass)` où R_1 et R_2

sont les règles de transformation illustrées par la Figure 5.7. De façon implicite, la règle R_i ne s'exécutera que lorsque les règles R_1 et R_2 ont été déjà exécutées.

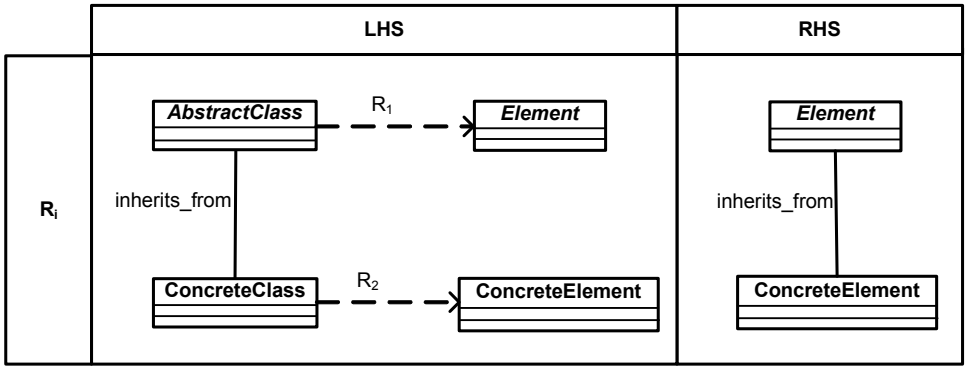


Figure 5.8. Contexte minimal et suffisant pour l'application de la règle R_i

En plus de créer des entités (classes, interfaces, opérations, attributs et relations), nos règles de transformation initialisent les propriétés des entités générées lorsque c'est possible. La règle R_1, par exemple, spécifie que tous les attributs de la classe `AbstractClass` sont récupérés par la classe `Element`. Par contre, seules les opérations non marquées de `AbstractClass` sont récupérées par `Element` : les opérations marquées vont être transformées en des instances de `ConcreteVisitor`. La Figure 5.9 montre une version plus complète de la règle R_1. Ainsi, un attribut (`sourceAttribute`) de la classe `AbstractClass` est transformé par la règle R_1 en un attribut (`targetAttribute`) ayant le même nom et le même type. Une opération non marquée (`sourceOperation`) est transformée en une opération (`targetOperation`) ayant la même signature que l'opération source. Comme nous l'avons mentionné au chapitre 3, nous ne représentons dans un modèle de problème que les éléments concernés par l'application du patron. Ainsi, les opérations non marquées (`sourceOperation`) et les attributs (`sourceAttribute`) de la classe `AbstractClass` n'ont pas été représentés dans notre modèle du problème. La version complète de la règle R_1 inclut aussi les noms et les types des attributs et les signatures des opérations (i.e. les noms, les paramètres et les types de retour).

Par ailleurs, un élément de destination ne peut être généré que par une seule règle. Cependant, la prémisse de cette règle peut représenter un contexte auquel plusieurs éléments—sources et possiblement cibles—participent. Dans le cas du

patron Composite, par exemple, l'interface `Component` appartenant au modèle de solution est générée à partir des éléments représentant des composants simples et des éléments représentant les composites dans le modèle source. Dans le cas du patron « Fabrication abstraite », une classe de fabrication concrète est générée à partir d'une collection de classes concrètes représentant une famille de produits dans le modèle source.

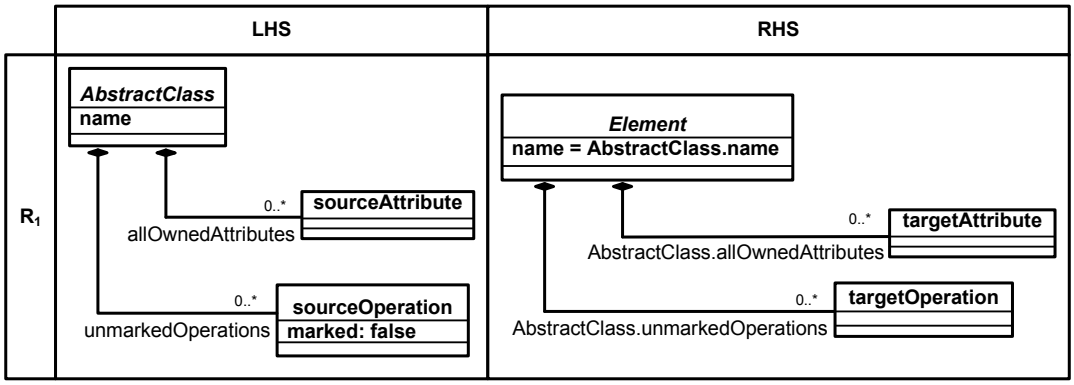


Figure 5.9. Calcul des propriétés de la classe `Element` à partir de celles de la classe `AbstractClass`.

De plus, lorsqu'une règle spécifie plusieurs éléments dans sa prémisse, elle spécifie aussi les relations entre ces éléments. Les relations peuvent faire partie du modèle source (e.g. une association, une relation d'héritage) comme elles peuvent être des relations de correspondance (*mapping*) générées précédemment par d'autres règles. Une relation de correspondance relie un élément du modèle source à un élément du modèle cible. Par exemple, la prémisse de la règle R_i comprend trois relations (Figure 5.10): (i) la relation d'héritage entre `ConcreteClass` et `AbstractClass`, (ii) la relation de correspondance ($\texttt{mapping}_{R1}$) entre `AbstractClass` et `Element` qui a été générée par la règle R_1, et (iii) la relation de correspondance ($\texttt{mapping}_{R2}$) entre `ConcreteClass` et `ConcreteElement` qui a été générée par la règle R_2. Concrètement, chacune de ces relations de correspondance est implémentée par une variable ayant deux attributs (`source` et `target`). Cette variable permet de garder la trace de l'exécution de la règle qui l'a générée.

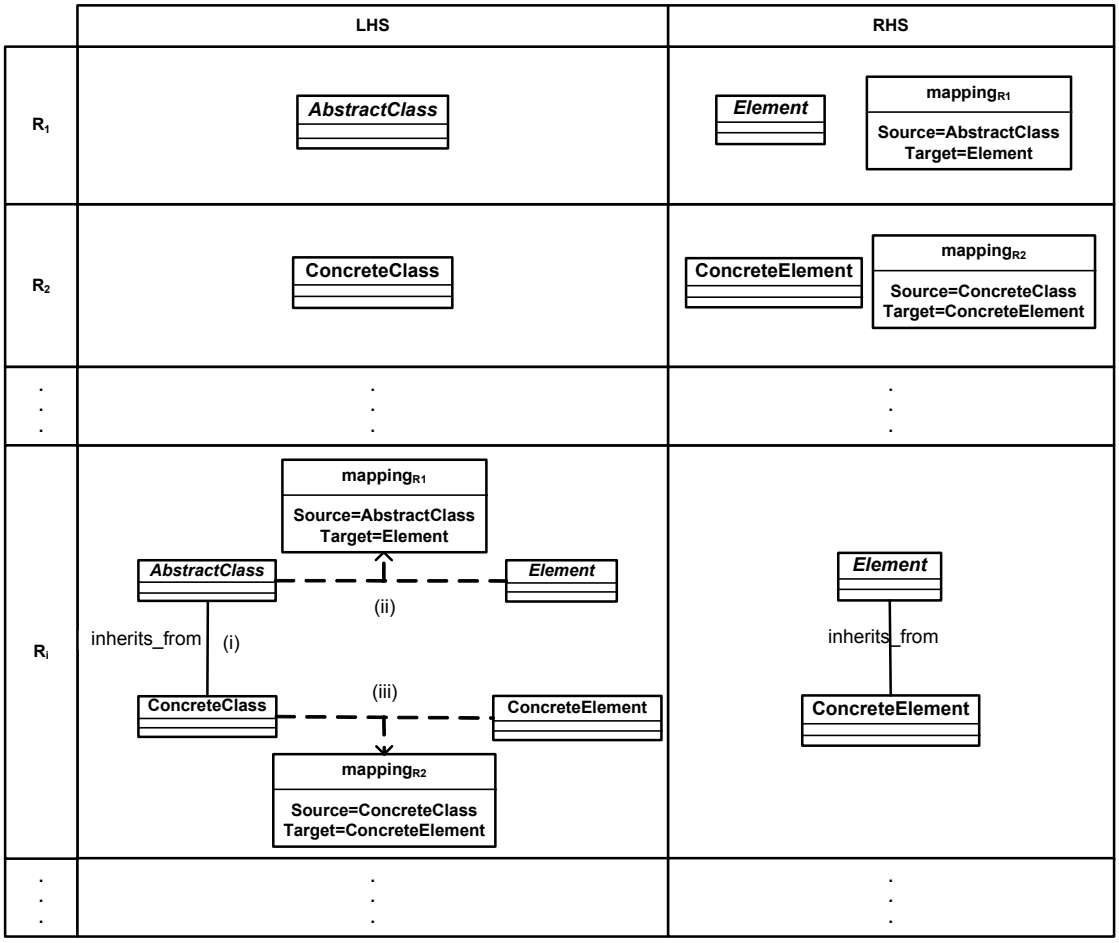


Figure 5.10. Exemple d'utilisation des variables de correspondance.

La Figure 5.11 montre un autre exemple de règle (règle R_{i+1}) dont la prémisse contient des entités du modèle source et du modèle cible. Elle montre aussi la règle R_4 où une opération `AbstractOperation` est transformée en une classe `ConcreteVisitor`. Le patron Visiteur est un des rares patrons où on transforme des classes en méthodes et des méthodes en classes.

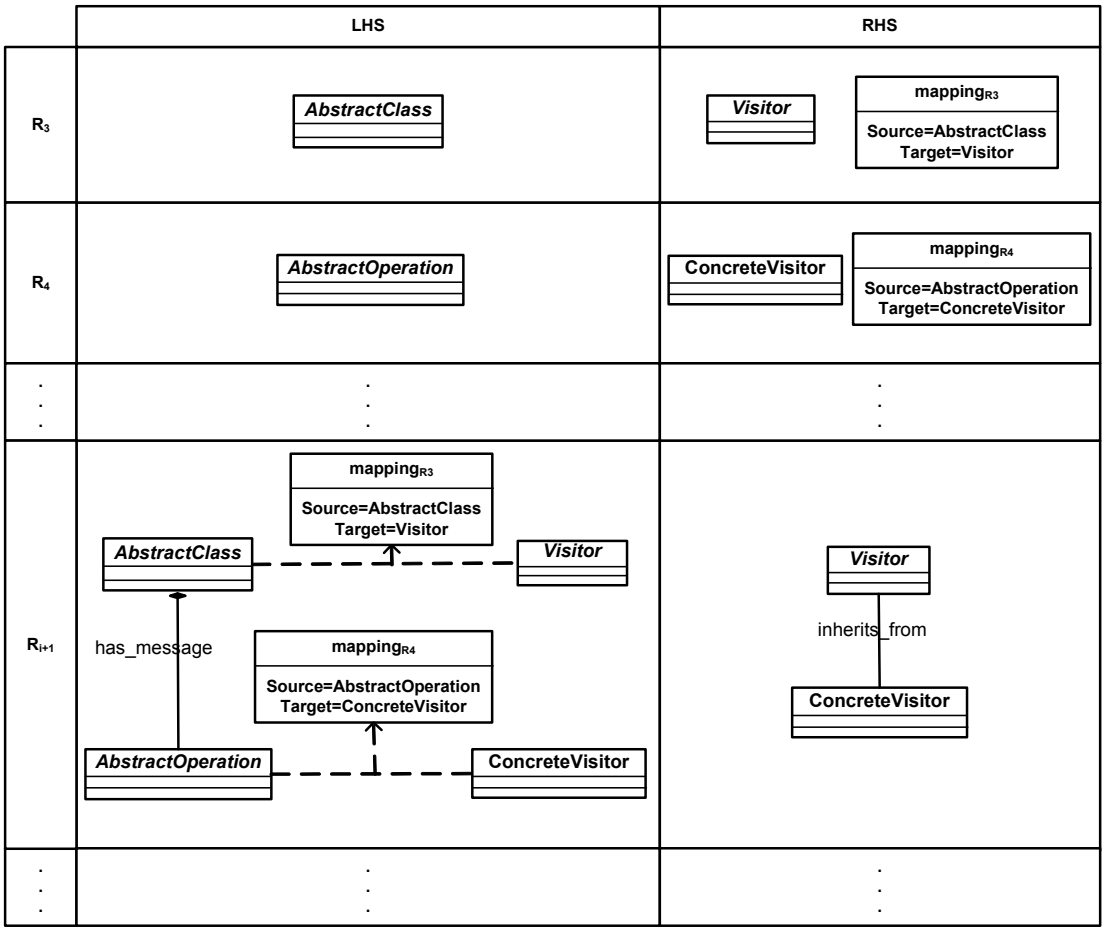


Figure 5.11. Un autre exemple de règle dont la partie LHS contient des entités sources et cibles.

5.4.5 Transformation des entités périphériques

Nous avons aussi défini un certain nombre de transformations qui s'appliquent aux entités clientes/périphériques utilisant des entités affectées par l'application d'un patron. Pour une instance détectée d'un modèle de problème donné, une entité est dite périphérique si (i) elle ne fait pas partie de l'instance en question, et (ii) elle a au moins une référence (e.g. extrémité d'association, type d'attribut, type de paramètre, etc.) à une entité de cette instance. Même si ces entités périphériques ne sont pas transformées, il faut mettre à jour leurs références aux entités transformées par l'application des patrons. Par exemple, chaque fois qu'une classe A est remplacée par une classe B, nous remplaçons toutes les références des classes périphériques à A par

des références à B. Les Figures Figure 5.12, Figure 5.13 et Figure 5.14 montrent des règles qui permettent de mettre à jour les possibles références d'une classe périphérique appelée `ClientClass` à la classe `AbstractClass`, i.e. quand le type d'une extrémité d'association, d'une opération ou d'un paramètre est `AbstractClass`, il est remplacé par le type `Element`. Notons que `ClientClass` n'est pas une marque: les classes périphériques ne sont pas marquées.

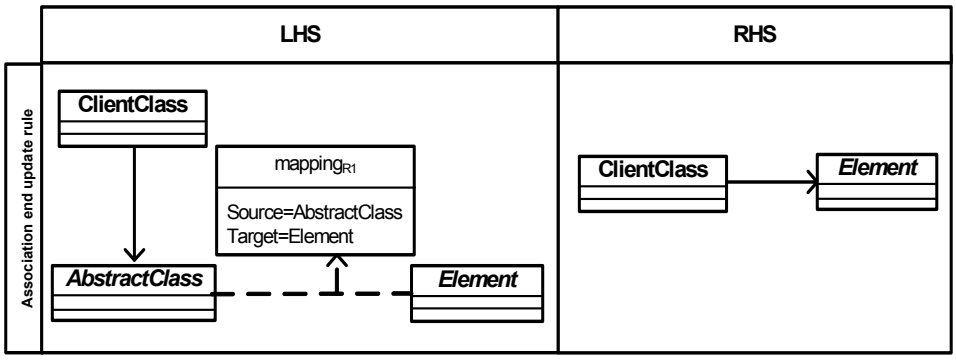


Figure 5.12. Règle de mise à jour d'une association entre une classe périphérique et la classe `AbstractClass`.

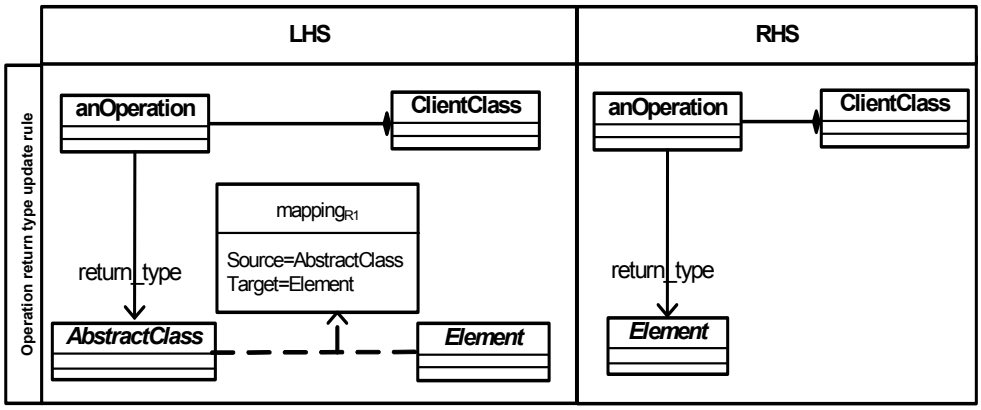


Figure 5.13. Règle de mise à jour du type d'une opération d'une classe périphérique.

En pratique, il est nécessaire d'implémenter d'autre règles de mises à jour qui sont reliées à l'aspect *dynamique*, e.g. une méthode d'une classe cliente qui appelait une méthode `AbstractOperation` de la classe `AbstractClass` doit maintenant appeler la méthode `accept` de la classe `Element` en lui passant comme paramètre la classe de type `Visitor` qui a été générée à partir de l'opération `AbstractOperation`.

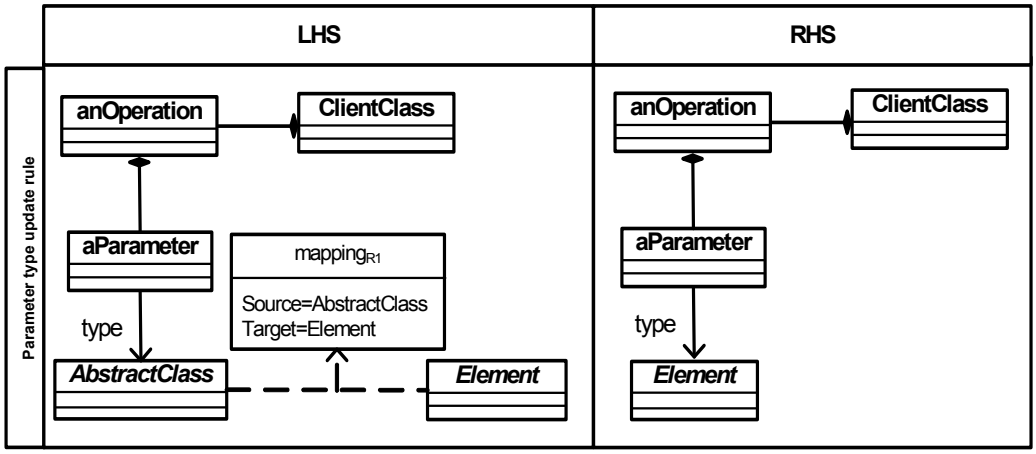


Figure 5.14. Règle de mise à jour du type d'un paramètre.

5.5 Implémentation des règles de transformation

Nous avons choisi d'implémenter nos règles de transformation en utilisant ILOG JRules™. JRules est un système hybride *objet-règles* qui permet de séparer les règles d'affaires du code d'une application, facilitant ainsi l'adaptation et la maintenance de l'application. De plus, l'algorithme optimisé de JRules permet une vitesse d'exécution rapide même pour un nombre élevé de règles et cela comparativement à d'autres systèmes à base de règles (e.g. JESS, JBoss Rules).

Dans le reste de la section, nous donnerons d'abord une description sommaire de JRules. Nous présenterons à la section 5.5.2 un exemple de règle illustrant comment nous avons implémenté nos règles avec JRules. Finalement, nous décrirons l'interface et le fonctionnement de notre transformateur à la section 5.5.3.

5.5.1 ILOG JRules pour la représentation des règles

Dans JRules, une règle est écrite avec le langage IRL (*ILOG Rule Language*) puis exécutée par le moteur de JRules. Le langage IRL a une syntaxe similaire au langage Java et il permet d'utiliser les opérateurs et les collections Java dans l'expression des règles.

Une règle est composée d'une partie *condition* et d'une partie *action*. Elle a la forme : `when {Condition} then {Action}`. Une condition peut porter sur un type,

une relation, une annotation, etc. La partie *Action* décrit les actions à faire si la partie *Condition* est satisfaite. Dans notre contexte, elle peut comprendre la création d'une entité (i.e. classe, attribut, opération ou association), la suppression d'une entité, la modification d'une entité, ou la création de liens entre deux entités. En fait, les conditions et les actions réfèrent à des objets Java. Dans notre cas, les objets manipulés dans les conditions et les actions sont des instances du modèle `ECore` (i.e. méta-modèle EMF).

Le moteur ILOG JRules est composé de trois éléments :

1. un *ruleset* qui représente l'ensemble des règles, fonctions et activités qui sont traitées par le moteur,
2. une mémoire de travail (*working memory*) qui stocke des références aux objets à traiter par les règles, et
3. un agenda qui maintient une liste de paires <règle, n-uplet> en attente d'exécution: tout n-uplet et règle tels que le n-uplet satisfait les conditions de la règle.

Typiquement, le moteur JRules analyse les objets présents dans sa mémoire de travail et les conditions des règles du *ruleset* pour déterminer parmi les instances des règles celles qui doivent être ajoutées à son agenda d'exécution et celles qui doivent en être supprimées. Une règle peut avoir plusieurs instances dans l'agenda. Ensuite, le moteur de règles décide quelle instance de règle dans l'agenda sera exécutée en premier. Une fois une règle exécutée, le moteur réorganise la mémoire de travail pour refléter le résultat de l'exécution. Cette mise à jour peut modifier l'agenda: d'autres règles peuvent s'ajouter à l'agenda.

5.5.2 Exemple de règle

La Figure 5.15 montre la règle R_1 de la Figure 5.9 telle que codée en JRules. Cette règle transforme une classe ayant la marque `AbstractClass` qui correspond à la classe `AbstractClass` du modèle de problème, en une classe qui correspond à la classe `Element` du modèle de la solution. Comme déjà expliqué au chapitre 4 (section 4.6.3), nous avons implémenté les marques sous forme d'annotations en utilisant la

classe `EAnnotation` de EMF. Nous rappelons aussi au lecteur qu'une instance de `EAnnotation` possède un attribut nommé `Details` (de type `Map`) qui permet d'associer des valeurs à différentes clés. Dans notre contexte, une marque associée à une entité, est composée de trois clés: 1) la clé `pattern` qui contient le nom du patron détecté; 2) la clé `role` qui correspond au rôle de l'entité dans le problème de conception; et 3) la clé `instance`: qui contient le numéro de l'instance detectée.

```
1.  when
2.  {
3.     ?anAbstractClass: EClass (
4.          ?visitorAnnotation: getEAnnotationForPattern(?this,"Visitor");
5.          ?attributes: ?this.getEAttributes();
6.          ?operations: ?this.getEOperations() ;
7.          ?visitorAnnotation != null ;
8.          ?visitorAnnotation.getDetails().get("role").equals("AbstractClass")) ;
9.     ?unmarkedOperations: collect EOperation (?this.getEAnnotations().size() == 0)  in ?operations ;
10. }
11. then
12. {
13.    EClass ?anElementClass = createEClass(?anAbstractClass.getName());
14.    addAttributes(?anElementClass,?attributes);
15.    addOperations(?anElementClass,?unmarkedOperations);
16.    EAnnotation ?annotation = createEAnnotation();
17.    ?annotation.getDetails().put("pattern","Visitor");
18.    ?annotation.getDetails().put("role","Element");
19.    ?annotation.getDetails().put("instance",?visitorAnnotation.getDetails().get("instance"));
20.    ?annotation.setEModelElement(?anElementClass);
21.    insert (?anElementClass) ;
22.    Map ?R1 = createMapping(?anElementClass,?anAbstractClass);
23.    insert (?R1);
24.    ?new_objects.add(?anElementClass);
25. }
```

Figure 5.15. Exemple d'une règle de transformation écrite en JRules.

Dans la partie *Condition* de la règle R_1 (Figure 5.15), la fonction `getEAnnotationForPattern(EnamedElement entity, String nameOfPattern)` (ligne 4) permet de parcourir les annotations associées à une entité donnée en paramètre pour trouver—si elle existe—l'annotation associée au patron dont le nom est donné en paramètre. Ainsi, la règle s'exécutera si une classe `?anAbstractClass` (ligne 3) a une annotation dont la clé « `pattern` » est égale à ‘‘`Visitor`’’ (`?visitorAnnotation != null;`) (ligne 7) et si sa clé « `role` » est égale à la marque ‘‘`AbstractClass`’’ (ligne 8). Nous utilisons les variables `?attributes` et `?operations` pour récupérer respectivement les attributs et les opérations de la classe en cours (lignes 5 et 6). L'ensemble des opérations non marquées, représenté par la

variable `?unmarkedOperations`, est construit à partir de l'ensemble des opérations (ligne 9).

Dans la partie *Action* de la règle, nous créons une nouvelle classe `?anElementClass` qui jouera le rôle de la classe `Element` dans la solution. La fonction `createEClass(String name)` permet de créer une classe avec le nom `name` (ligne 13). La classe `?anElementClass` a le même nom que la classe `?anAbstractClass`. Les fonctions `addAttributes(EClass aClass, Collection attributes)` et `addOperations(EClass aClass, Collection operations)` permettent respectivement d'ajouter une liste d'attributs et une liste d'opérations à la classe donnée en paramètre (lignes 14 et 15). Cette règle créé aussi une annotation (ligne 16) dont la clé `pattern` a la valeur `Visitor` (ligne 17), dont la clé `role` a une valeur égale à `Element` (ligne 18) et dont le numéro d'instance est le même que celui de la classe source (ligne 19), i.e. le numéro d'instance de la solution correspond au numéro d'instance du problème source. L'annotation ainsi créée est utilisée pour marquer la classe `?anElementClass` générée par cette règle (ligne 20).

L'opération `insert(?anElementClass)` permet d'insérer l'objet en paramètre dans la mémoire de travail (ligne 21). Si l'objet est déjà dans la mémoire de travail, elle le met à jour. Sinon, elle le crée avant de l'insérer. Pour implémenter les variables de correspondance, nous avons utilisé des tables `HashMap` dont les clés sont les éléments cibles générés et les valeurs sont les éléments sources étant donné qu'un élément cible ne peut être généré que par une seule règle, contrairement à un élément source qui peut faire partie de la prémisse de plusieurs règles. Nous rappelons au lecteur qu'une variable de correspondance est créée par une règle pour établir une correspondance entre les éléments cibles et les éléments sources qui ont servi à leur génération par cette règle. Pour notre exemple, une correspondance entre les classes `?anElementClass` et `?anAbstractClass` est créée (ligne 22) et insérée dans la mémoire de travail (ligne 23).

Nous utilisons une collection d'objets `new_objects` pour stocker les objets créés par les règles (ligne 24). En fait JRules nous permet de définir des paramètres d'entrées et sorties qui peuvent être utilisés par les règles. La collection `new_objects`

sera utilisée par le module de transformation qui permet l'exécution du processus global de transformation. Ce module est décrit à la section suivante.

JRules permet de définir et d'implémenter un ensemble fonctions (appelé `FunctionSet`) qui est associé à un `RuleSet`. Ces fonctions correspondent à des ''macros'' dans les langages traditionnels. Elles sont utilisées pour simplifier l'écriture des règles. C'est le cas de nos fonctions `createEClass`, `createEOperation`, `createEReference`, `createAnnotation`, `createMapping`, `addAttributes`, `addReference`, `getEAnnotationForPattern`, etc.

5.5.3 Processus de transformation et interface du transformateur de modèles

Concrètement, le processus d'application d'une transformation se présente comme suit :

- On commence par lire le modèle à transformer (format XMI), qui est le résultat de l'étape de marquage, et on met les objets marqués dans la mémoire de travail.
- Le moteur JRules analyse les objets de la mémoire de travail et identifie les instances potentielles de règles (i.e. celles dont les conditions sont satisfaites). Ces instances sont placées dans l'agenda d'exécution.
- Le moteur JRules itère sur le contenu de l'agenda. Il exécute les règles qui sont dans l'agenda et met à jour la mémoire de travail ce qui peut déclencher l'ajout de nouvelles règles à l'agenda d'exécution. La transformation se termine lorsqu'il n'y a plus de règles dans l'agenda.
- On sérialise le modèle obtenu (i.e. les objets générés par l'exécution des règles) en format XMI.

L'API de notre transformateur est montrée à la Figure 5.16. La classe `ModelTransformer` regroupe toutes les fonctions nécessaires pour appliquer les patrons aux modèles marqués. Le processus de transformation/application des patrons

est enclenché par l'appel de la méthode `executeTransformation`. Cette méthode reçoit comme paramètre le modèle marqué sous la forme d'un fichier XMI.

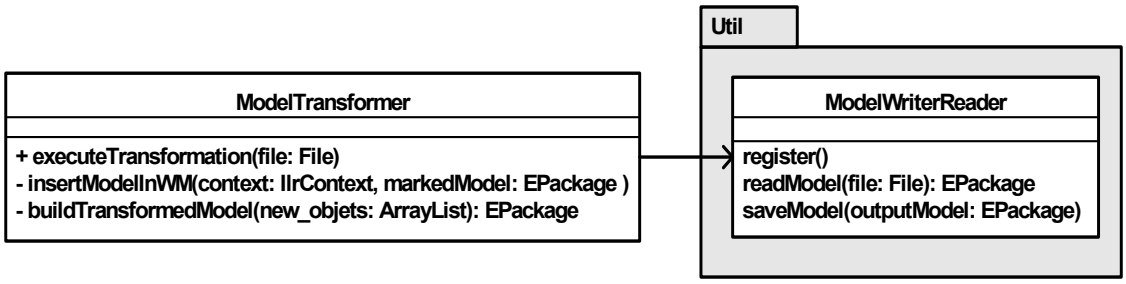


Figure 5.16. Interface de notre transformateur de modèle

Comme le montre la Figure 5.17, la méthode `executeTransformation` fait appel à la classe utilitaire `ModelWriterReader` pour lire le modèle marqué à partir du fichier reçu en entrée (lignes 2 et 3). Elle créé une instance de `EPackage` appelée `transformedModel` (ligne 4) : cette instance servira à stocker le modèle obtenu après transformation. Ensuite, elle crée une instance de la classe `IlrRuleset` qui représente, dans JRules, une entité qui gère un ensemble de règles (ligne 5). Les règles dans un *ruleset* sont extraites par l'analyse (*parsing*) d'unités d'entrées. Dans notre cas, les règles que nous avons créées avec JRules ont été exportées dans un fichier (`data/rules.irl`). Ce fichier est donc analysé et son contenu est récupéré dans l'instance de `IlrRuleset` que nous avons créée (ligne 6). Ensuite, une instance de la classe `IlrContext` est créée à partir de notre *ruleset* (ligne 10). Une instance de `IlrContext` représente un moteur d'exécution : les règles ne peuvent être exécutées qu'à l'intérieur d'un contexte d'exécution lequel est toujours attaché à un *ruleset*.

```
1. public void executeTransformation(File file) {
2.    ModelWriterReader myWriterReader = new ModelWriterReader();
3.    EPackage markedModel = myWriterReader.readModel(file);
4.    EPackage transformed_model = EcoreFactory.eINSTANCE.createEPackage();
5.    IlrRuleset ruleset = new IlrRuleset();
6.    if (! ruleset.parseFileName("data/rules.irl")){
7.       System.out.println("Ruleset parsing error");
8.       System.exit(-1);
9.    }
10.   IlrContext ruleEngine = new IlrContext(ruleset);
11.   try{
12.      insertModelInWM(ruleEngine, markedModel);
13.      IlrParameterMap results = ruleEngine.execute();
14.      ArrayList new_objects = (ArrayList)results.getObjectValue("new_objects");
15.      transformedModel  = buildTransformedModel(new_objects);
16.   } catch (Exception ex) {
17.      ex.printStackTrace();
18.   }
19.   myWriterReader.saveModel(transformedModel);
20.}
```

Figure 5.17. Implémentation de la méthode `executeTransformation`.

Nous insérons ensuite les entités du modèle marqué dans la mémoire de travail du moteur (ligne 12) ; cela est réalisé par la méthode `insertModelInWM` de notre transformateur. Nous invoquons ensuite la méthode `execute()` du moteur (ligne 13). Cette méthode exécute les règles selon le séquencement (*ruleflow*) défini dans le *ruleset*, et elle retourne une instance de la classe `IlrParameterMap`. Cette instance implémente une structure pour stocker les résultats d'exécution des règles comprenant le nombre de règles exécutées, et les noms et valeurs de chaque paramètre sortant du *ruleset* associé à notre engin d'exécution. Nous récupérons ensuite le contenu de la variable `new_objects` (ligne 14) que nous avons définie dans notre *ruleset* comme paramètre sortant, et que nous avons utilisé pour stocker les objets après transformation, comme nous l'avons montré dans la règle de la Figure 5.15.

Enfin, nous utilisons la méthode `buildTransformedModel` de notre transformateur pour construire le modèle transformé à partir de la liste d'objets contenus dans la variable `new_objects` (ligne 15). Ce modèle est ensuite sérialisé sous format XMI (ligne 19).

Nous avons implémenté les règles de transformation associées aux patrons Visiteur, Pont, Stratégie, Proxy, Singleton, Composite et Fabrication abstraite. Nous les avons organisées selon le séquencement (*ruleflow*) montré à la Figure 5.18. En effet, JRules permet de modéliser et de contrôler explicitement la séquence d'exécution des règles en utilisant des ‘‘diagrammes d'activités’’. Ainsi, nous avons créé pour chaque patron une activité qui regroupe les règles qui lui sont associées. Ces activités sont des tâches indépendantes et parallèles : une tâche est déclenchée (i.e. les règles la composant sont enclenchées) lorsqu'un fragment du modèle est marqué comme étant une instance du problème résolu par le patron associé à cette tâche. L'activité `InitializeFunction` est une fonction qui ne fait qu'annoncer le démarrage du processus de transformation.

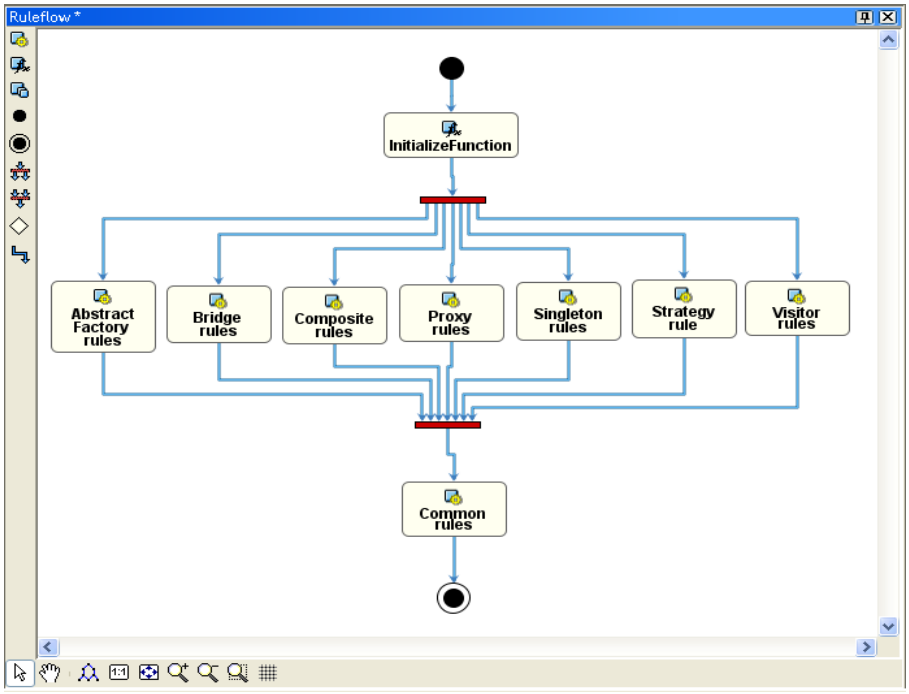


Figure 5.18. Séquencement des règles

L'activité « `Common rules` » regroupe les règles communes dont le but est de récupérer les entités qui n'ont pas été touchées par les transformations : toutes les entités du modèle source sauf celles affectées par l'application des patrons et les entités qui ont des références sur les entités affectées par l'application des patrons.

Dans un premier temps, nous avons utilisé notre transformateur pour appliquer des patrons à des fragments de modèles contenant une instance à la fois d'un problème de conception, i.e. nous appliquons un seul patron à la fois. L'application automatique du patron considéré se fait de façon correcte, i.e. le modèle obtenu après transformation est conforme au méta-modèle UML (correction syntaxique) et la sémantique du modèle initial a été préservée. Toutefois, nous avons été confrontés à certains problèmes communs aux approches de transformations de modèles dans le contexte de chevauchement des instances à transformer. Cette problématique est illustrée et discutée au chapitre 6.

5.6 Conclusion

Nous avons présenté dans ce chapitre notre approche pour la représentation et la mise en œuvre des transformations inhérentes à l'application des patrons. Nous avons pu implémenter les règles de transformation associées à plusieurs patrons de conception en utilisant cette approche, et nous avons pu appliquer automatiquement ces patrons à des modèles marqués contenant une instance de patron à la fois.

Notre approche est déclarative puisque nos règles de transformation sont exprimées en terme des entités des modèles de problèmes et de solutions associés aux patrons, et non pas en terme des entités des modèles auxquels nous désirons appliquer les patrons. L'approche peut être utilisée pour mettre en œuvre d'autres types de patrons pouvant être décrits en terme de problèmes et solutions.

Chapitre 6

Expérimentations et résultats

Le but de ce chapitre est de valider notre approche pour la représentation et l'application des patrons de conception. Dans les chapitres 3, 4 et 5, nous avons présenté des exemples qui démontrent le fonctionnement de notre approche notamment pour représenter des patrons tels que les patrons Visiteur, Pont et Composite, et pour appliquer, de façon semi-automatique, ces patrons à des modèles *synthétiques* (i.e. des modèles que nous avons fabriqués nous-mêmes). Pour vérifier et valider notre approche, il fallait réaliser des tests en utilisant des modèles de taille moyenne à grande, et, préférablement, des modèles d'applications ou de systèmes en utilisation. En pratique, nous ne disposons pas de tels modèles. En effet, un des problèmes communs aux approches dirigées par les modèles est la non disponibilité d'un reposoir de modèles ''*benchmarks*'' qui peuvent être utilisés pour tester et comparer les approches de transformation de modèles.

Nous commencerons par présenter les différents aspects que nous désirons valider (section 6.1). Nous présenterons à la section 6.2 les résultats reliés à la validation de l'aspect représentation des patrons de conception. Ces résultats sont donnés sous forme d'une classification des problèmes résolus par ces patrons (section 6.2.1). Nous discuterons aussi des modifications que nous avons apportées à la représentation de certains patrons en particulier le patron Visiteur (sections 6.2.2 et 6.2.3). Nous présenterons ensuite, à la section 6.3, la démarche que nous avons adoptée pour valider les aspects détection et transformation et tester l'approche sur de gros modèles (section 6.3.1), les problèmes que nous avons rencontrés lorsque nous

transformons des instances chevauchantes de patrons (section 6.3.2) et un résumé des résultats des tests effectués (section 6.3.3). Nous terminerons par une discussion à la section 6.4.

6.1 Aspects à valider

Nous nous sommes posé plusieurs questions sous-jacentes à la validation de l'approche. En fait, nous distinguons trois types de validation : (i) la validation de la représentation proposée pour les patrons; (ii) la validation de l'aspect détection des problèmes de conception, laquelle est intimement reliée à la validation de l'aspect représentation; et (iii) la validation de l'aspect transformationnel.

6.1.1 Questions reliées à la représentation et détection des problèmes de conception

La validation de notre représentation des patrons nous amène surtout à la validation de la caractérisation des problèmes de conception. Pour ce faire, nous devons, d'une part, évaluer notre capacité d'appliquer notre formalisme à tous les patrons de conception et, d'autre part, vérifier la correction et la complétude des modèles de problèmes que nous proposons.

Dans ce contexte, la première question à laquelle nous avons essayé de répondre est: ‘‘*Sommes-nous capables de représenter tous les problèmes résolus par les patrons de conception ?*’’. Pour répondre à cette question, nous avons étudié les vingt trois patrons du catalogue de Gamma et al. (1995). Les résultats de cette étude sont décrits à la section 6.2.

La seconde question est: ‘‘*Jusqu'à quel degré le modèle de problème que nous proposons pour un patron, est-il une représentation fidèle du problème de conception résolu par ce patron ?*’’. Dans ce contexte, il y a deux aspects à considérer :

- Est-ce que la représentation du problème de conception est correcte ? Nous devons vérifier pour chaque patron si toutes les instances détectées du modèle

de problème associé, dans un modèle donné en entrée, correspondent réellement à des contextes où le patron doit être appliqué.

- Est-ce que la représentation du problème de conception est complète ? Nous devons nous assurer qu'il n'y a pas d'instances légitimes du problème que notre modèle de problème ne représente pas, i.e. nous détectons toutes les instances légitimes.

Pour répondre à ces questions, nous avons appliqué notre processus de marquage (i.e. détection automatique) à des modèles générés à partir de logiciels libres. Notre démarche ainsi que les résultats obtenus sont décrits à la section 6.3.

6.1.2 Questions reliées à l'aspect transformation

Les principales questions que nous nous posons dans ce contexte sont reliées à la correction et à la complétude de nos transformations. Toutefois, nous nous sommes aussi penchés sur les questions de terminaison, de confluence et de symétrie des transformations.

Dans le cadre de la correction des transformations, nous avons essayé de vérifier si les modèles obtenus par l'application d'un ou de plusieurs patrons, en utilisant notre approche, sont corrects. Il y a deux aspects à considérer dans ce contexte :

- Correction syntaxique du modèle obtenu après transformation : Nous devons nous assurer que le modèle transformé est conforme au méta-modèle UML. Dans ce contexte, nous nous sommes intéressés aux problèmes de conflits lorsque différentes instances du même patron ou de différents patrons s'appliquent au même fragment de modèle. Pour gérer ce problème, comme nous l'avons décrit au chapitre 4, le marquage de chaque instance détectée fait référence au patron concerné. De plus, pour écarter les possibilités de conflits lorsque les différentes instances sont du même patron, nous indexons les instances d'un même patron—quand on en détecte plus d'une—dans le même modèle source (e.g. « `Visiteur.AbstractClass.1` », « `Visiteur.AbstractClass.2` »). Toutefois, comme nous le verrons dans les

résultats des tests, cette démarche n'est pas suffisante pour régler les problèmes qui peuvent survenir quand des instances de patrons se chevauchent.

- Correction sémantique du modèle obtenu après transformation : Le modèle transformé doit préserver la sémantique du modèle source. L'équivalence sémantique entre le modèle source et le modèle obtenu après transformation ne peut être prouvée formellement (Varro et al., 2002). Par contre, nous pouvons analyser manuellement un modèle obtenu après application automatique des patrons pour vérifier si la sémantique du modèle source a été preservée.

Dans le cadre de la complétude des transformations (i.e. on transforme tout ce qui doit être transformé), nous nous sommes intéressés au problème de correspondances partielles entre modèles sources et modèles de problèmes. Ce problème peut résulter de l'incomplétude des modèles sources. En fait, c'est un problème commun aux approches de transformation de modèles. Toutefois, dans notre implémentation actuelle du processus de marquage, nous ne marquons des entités du modèle source que lorsqu'une instance complète du modèle de problème est détectée et que ces entités en font partie. Ce qui signifie que notre transformateur n'enclenche l'exécution des règles de transformations d'un patron que lorsque tous les éléments décrits dans le modèle de problème en question sont détectés. La détection d'instances incomplètes de nos modèles de problèmes est discutée dans les travaux futurs au chapitre suivant.

Pour ce qui est de l'aspect terminaison des transformations, nous avons essayé de vérifier, à travers nos tests, si les transformations enclenchées pour appliquer les patrons dont les problèmes ont été détectés se terminent. En général, la terminaison des transformations est indécidable (Heckel et al., 2002). Cependant, dans notre contexte, le nombre de marques qui guident les transformations est assez limité et les marques utilisées dans la partie *Action* des règles pour marquer les entités transformées sont différentes de celles utilisées pour marquer les problèmes, i.e. une entité source est marquée par son rôle dans le problème et une entité cible est

marquée par son rôle dans la solution. Cette différentiation des marques utilisées *avant* et *après* transformation nous permettent d'éviter les risques de boucle infinie.

Dans le cadre de la confluence des transformations, nous avons essayé de vérifier si le modèle obtenu après l'application d'un patron est le même quel que soit l'ordre d'application des règles faisant partie de la transformation associée au patron. Cette question est reliée à la nature non déterministe des transformations. De façon simplifiée, la confluence d'une transformation signifie que le modèle obtenu à la fin de l'exécution des règles associées est le même indépendamment de l'ordre dans lequel ces règles sont exécutées. Lorsque la transformation n'est pas confluente, cela implique la possibilité que la transformation génère des modèles intermédiaires qui ne peuvent plus être transformés (Küster et Abd-El-Razik, 2006), autrement dit, des modèles qui ne sont pas syntaxiquement corrects. Dans notre contexte, vu notre démarche pour simplifier la transformation associée à un patron (section 5.4 du chapitre précédent), nous avons deux catégories de règles dans une transformation:

1. Les règles qui génèrent les entités du modèle cible (i.e. les nœuds), une entité à la fois. Ces règles sont indépendantes les unes des autres, dans la mesure où chacune génère une entité cible distincte: une entité cible ne peut être générée que par une seule règle. Quel que soit l'ordre d'application de ces règles, le résultat final est l'ensemble des entités (i.e. nœuds) du modèle cible.
2. Les règles qui génèrent les relations (i.e. les arêtes) entre les entités du modèle cible, une relation à la fois. Ces règles sont aussi indépendantes les unes des autres puisqu'une relation du modèle cible ne peut être générée que par une seule règle. Par contre, une règle de cette catégorie ne peut s'exécuter que si les entités cibles pour lesquelles elle génère une relation ont été déjà générées. Donc, chacune des règles de cette catégorie dépend de deux règles de la première catégorie et doit s'exécuter après ces deux règles. De façon implicite, l'utilisation des variables de correspondance dans une règle de cette catégorie, retarde l'exécution d'une règle jusqu'à ce que les deux règles dont elle dépend soient exécutées. Nous rappelons au lecteur qu'une relation de

correspondance est générée par une règle, et qu'elle permet de relier une entité du modèle source à une entité du modèle cible.

Le dernier aspect de notre validation concerne la symétrie des transformations : ''*Est-ce que l'application de plusieurs patrons à un fragment de modèle donne le même résultat quel que soit l'ordre dans lequel les patrons sont appliqués ?*''. En fait, dans notre contexte, le problème de symétrie des transformations ne se pose que lorsque des instances de patrons se chevauchent, i.e. les instances détectées de problèmes se chevauchent. Les tests que nous avons faits, ont démontré que ce n'est pas le cas pour tous les patrons dans certains contextes. Dans ces cas là, il faut imposer un ordre d'exécution aux patrons chevauchants. Ce problème est illustré à la section 6.3.2.

Pour répondre à ces questions, nous avons réalisé des transformations sur des modèles que nous avons générés à partir de logiciels libres et que nous avons marqué en utilisant notre framework. Les résultats sont présentés à la section 6.3.

6.2 Validation de l'aspect représentation des problèmes de conception

Le premier aspect à vérifier concerne notre capacité à représenter tous les problèmes résolus par les patrons de conception. Ainsi nous avons essayé d'identifier le type de problèmes qu'on peut caractériser en utilisant notre approche en étudiant les vingt-trois (23) patrons de conception de Gamma et al(1995). La réponse s'est avérée négative pour certains patrons. En fait, nous avons pu représenter les modèles de problèmes pour treize patrons (Pont, Visiteur, Fabrication abstraite, Composite, Stratégie, Façade, Proxy, Singleton, Adaptateur, Chaîne de responsabilité, Médiateur, Mémento et Méthode de fabrication). Les résultats de notre étude sont donnés sous forme de classification des problèmes résolus par les patrons de conception et des modèles de problèmes que nous proposons pour les représenter (section 6.2.1).

Par ailleurs, les tests que nous avons réalisés nous ont permis de constater que notre représentation des problèmes de conception, notamment pour les patrons

Visiteur, Pont et Stratégie, ne nous permet pas de détecter certaines instances possibles de ces problèmes: notre modèle de problème impose plus de contraintes sur les hiérarchies de classes que ce que les modèles sources exhibent en réalité. Ce problème est relié aux descriptions données de ces problèmes dans le catalogue des patrons de conception. Par exemple, dans les illustrations données dans Gamma et al.(1995) du patron Visiteur, d'une part, les méthodes participant au problème sont des méthodes qui sont implémentées par toutes les sous-classes de la hiérarchie concernée par le problème, et d'autre part, elles sont toutes abstraites au niveau de la classe racine et concrètes au niveau des sous-classes. En pratique, certaines méthodes concernées par le problème peuvent être concrètes même si la classe qui les contient est abstraite. De plus, une méthode n'a pas besoin d'être implémentée par toutes les sous-classes de tous les chemins de la hiérarchie; il suffit qu'elle soit implémentée ou redéfinie par une sous-classe dans chacun des chemins. Par conséquent, nous avons modifié la représentation des problèmes de conception résolus par les patrons Pont, Visiteur et Stratégie et donc, aussi les problèmes de satisfaction de contraintes associés à ces patrons et aussi la façon dont nous regroupons les instances détectées. Nous illustrons ces modifications avec le modèle de problème résolu par le patron Visiteur (sections 6.2.2 et 6.2.3).

6.2.1 Classification des problèmes résolus par les patrons de conception

Notre étude des patrons de conception nous a permis de constater que les problèmes qu'ils résolvent ne sont pas tous du même niveau. Nous avons donc regroupé en trois catégories les problèmes résolus par les patrons et les modèles que nous proposons pour les représenter :

1. Problèmes de conception de premier niveau : Certains patrons s'attaquent à des problèmes de conception qui sont apparents dans les modèles d'entrée, e.g. un ensemble de classes fortement couplées dans le cas du patron Médiateur, un trop fort couplage entre les clients et les classes d'un sous-système ou entre différents sous-systèmes dans le cas du patron Façade, etc.

Dans ces cas, notre modèle du problème est une représentation complète du problème résolu par le patron. Toutefois, dans la plupart de ces cas, nous ne sommes pas en mesure de détecter automatiquement des instances de ces problèmes en utilisant notre processus de marquage tel quel : il faut recourir à d'autres moyens pour analyser les modèles (e.g. visualisation ou métriques). Par ailleurs, même si nous arrivions à détecter ces problèmes, nous ne pourrions pas automatiser l'application de certains des patrons concernés. Par exemple, dans le cas du patron Médiateur, nous aurions pu détecter une occurrence du problème en combinant métriques et visualisation, mais nous ne pourrions pas appliquer le patron automatiquement. En effet, le patron Médiateur est un patron trop générique. Pour une interaction donnée entre les objets, on peut créer un médiateur particulier, mais d'une façon générale, le patron est paramétré par la sémantique/le comportement interne des objets participant au patron.

2. Exigences de conception : Dans la seconde catégorie des problèmes résolus par les patrons, nous regroupons des problèmes décrits par des exigences pour lesquelles il faut trouver une solution, e.g. une opération avec différents algorithmes (le patron Stratégie), une hiérarchie dont les classes sont stables mais pas le comportement (le patron Visiteur), une composition récursive avec un comportement similaire des composants et des composites (le patron Composite), des objets distants (le patron Proxy), des familles d'objets où une famille, à la fois, peut être utilisée pour configurer un système (le patron Fabrication abstraite). Dans ces cas, nous ne sommes pas toujours en mesure de détecter le problème dans un modèle source. Cependant, nous pouvons détecter—dans le modèle source—une mauvaise solution au problème considéré. Nous représentons alors les problèmes de cette catégorie en termes de mauvaises solutions. Toutefois, pour certains patrons, le modèle de problème est une représentation de la mauvaise solution au problème associé au patron (e.g. Composite, Fabrication abstraite, Adaptateur, Chaine de responsabilité, Proxy) alors que pour d'autres patrons, le modèle de problème est une représentation de la mauvaise solution augmentée des points de

variation (e.g. Pont, Visiteur, Stratégie). Notons que dans le cas du patron Proxy, même si nous avons représenté le problème en terme d'une mauvaise solution, nous ne sommes pas en mesure d'en détecter les instances, i.e. nous ne pouvons pas identifier dans un modèle des classes assujetties à des contraintes non structurelles (e.g. un objet est coûteux quand il est chargé en mémoire). Cependant, cette représentation nous permet d'automatiser l'application du patron une fois une instance est marquée manuellement.

3. Spécifications de solutions : Dans la troisième catégorie des problèmes résolus par les patrons, le problème n'est pas identifié précisément et la description donnée dans le catalogue des patrons est une spécification d'une solution. Le patron représente alors une façon de l'implémenter (e.g. le patron Singleton et le patron Memento). Dans ce cas, on ne peut pas reconnaître le problème dans le modèle en entrée. Toutefois, nous pouvons représenter le modèle de problème en terme d'éléments à transformer, e.g. une classe candidate à l'application du patron Singleton ou du patron Mémento. Cela permet d'automatiser l'application du patron même si le marquage est fait manuellement.

Pour récapituler, le modèle du problème peut représenter : (1) une description complète du problème, (2) une description d'une mauvaise solution à un problème de conception, ou (3) une description de ce qui doit être transformé uniquement. Notre méta-modèle ne décrit que les problèmes de la deuxième et troisième catégorie. Le niveau de détails de nos modèles de problème peut dans certains cas nécessiter des informations qui ne sont pas représentées explicitement dans un modèle source. C'est le cas du patron « Fabrication abstraire » qui a besoin des signatures des constructeurs de certaines classes, chose qui n'est pas toujours disponible dans un modèle d'analyse ou de conception.

Pour diverses raisons, nous n'avons pu appliquer notre approche à certains patrons tels que les patrons Décorateur, Interpréteur et le patron de méthode (*template method*) et. Dans le cas du patron *template method*, nous avons besoin de beaucoup de connaissances qui ne sont pas présentes dans le modèle source. En fait, c'est un

problème commun aux patrons qui sont orientés *codage* plutôt que *conception*. Le patron *template method* en est un exemple puisqu'il représente une technique de réutilisation du code. En effet, ce patron propose de définir pour une opération dans une classe donnée, le squelette d'un algorithme, en déléguant certaines étapes à des sous-classes, et cela lorsque l'algorithme (i.e. le comportement) associé à l'opération possède autant des parties bien définies (invariantes) que des parties ayant différentes implémentations. Cette information n'est pas évidente à représenter de façon structurelle et elle ne se trouve pas dans un modèle de niveau analyse ou conception.

Dans le cas du patron Décorateur, dont l'objectif est de permettre l'ajout de responsabilités de façon dynamique à des objets, les comportements ou variables à ajouter n'existent pas forcément dans le modèle initial. Même dans le cas d'une mauvaise conception (e.g. un ensemble de classes ayant la même *interface fonctionnelle* mais avec des variantes de comportement), nous ne sommes pas en mesure de distinguer les comportements ou variables additionnels des autres qui ne sont pas reliés à la décoration/ l'ajout.

Dans le cas du patron Interpréteur, la description donnée par Gamma et al. (1995) du problème est : « Lorsqu'un problème particulier est rencontré assez souvent, il est intéressant d'exprimer des instances du problème avec des phrases dans un langage simple. Ensuite, on peut construire un interpréteur qui résout le problème en interprétant ces phrases. » (Gamma et al., 1995). Ce problème n'est pas facile à exprimer dans notre approche. En effet, il existe une trop grand écart sémantique entre la description du problème et la description de la solution. En fait, pour pouvoir appliquer ce patron dans un contexte donné, il faut vérifier: 1) la simplicité et la régularité des instances d'un problème dans le domaine considéré, 2) la possibilité d'exprimer ce problème de façon déclarative, et 3) l'existence d'une procédure systématique de résolution de ce problème à partir de son expression. Cette procédure est le processus d'interprétation qui utilise une grammaire dépendante de la paire (domaine, problème) en question. En fait, le patron Interpréteur peut être considéré comme un méta-patron puisqu'il décrit une approche de solution plutôt qu'une solution à un problème précis.

Le Tableau 6-1 résume les résultats de notre étude de l'ensemble des patrons du catalogue de Gamma et al.(1995).

Modèle de problème / Catégorie du patron	Représentation complète	Mauvaise solution	Mauvaise solution + points de variation	Éléments à transformer	Pas de représentation proposée
Problème de 1er niveau	• Médiateur • Façade				• Poids mouche
Exigences de conception		• Adaptateur • Chaîne de responsabilité • Composite • Fabrication abstraite • Méthode de fabrication • Proxy	• Pont • Stratégie • Visiteur		• Décorateur • Monteur (*Builder*) • Observateur • Commande • Itérateur • Prototype
Spécification de solution				• Singleton • Mémento	
Patron de codage					• Patron de méthode • État
Méta-patron					• Interpréteur

Tableau 6-1. Classification des patrons de conception et des modèles proposés de problèmes

6.2.2 Modification de la représentation du problème résolu par le patron Visiteur

Le modèle de problème associé au patron Visiteur—que nous reprenons à la Figure 6.1–correspondait à un modèle avec une classe abstraite et des méthodes implémentées par les sous-classes.

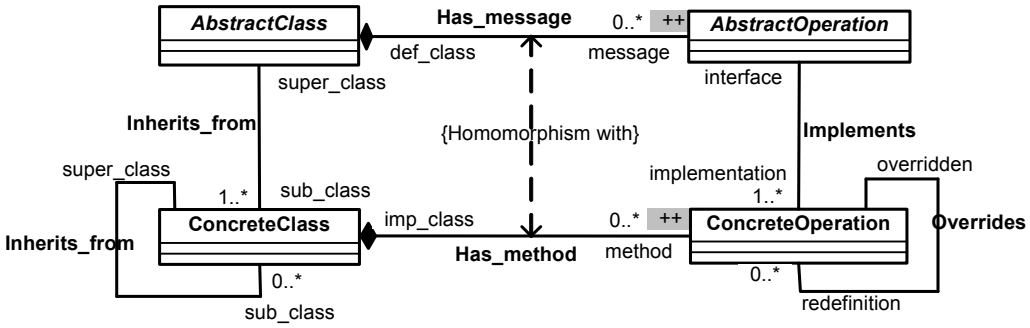


Figure 6.1. Modèle du problème résolu par le patron Visiteur.

Les tests nous ont permis de constater que pour faire partie d'un problème de conception, une méthode abstraite définie par la racine `AbstractClass` n'est pas obligatoirement implémentée par toutes les sous-classes composant une branche (i.e. un chemin) de la hiérarchie. Rien n'empêche des hiérarchies de classes telles que celles montrées à la Figure 6.2 d'être des instances du modèle statique de problème associé au patron Visiteur. Dans chacune de ces hiérarchies une méthode (`g()`) est implémentée par une seule classe dans chaque chemin distinct de la hiérarchie. Ainsi, même si l'illustration donnée dans Gamma et al.(1995) montre une classe dont l'ensemble des méthodes abstraites sont définies par toutes les sous-classes, nous pensons qu'il n'est pas nécessaire qu'une méthode soit implémentée par toutes les sous-classes pour qu'elle fasse partie d'un problème. En fait, il suffit que la méthode soit implémentée par *au moins une classe de chaque chemin de la hiérarchie*.

Ainsi, nous avons décidé de modifier notre représentation du problème résolu par le Visiteur (c'est le même modèle structurel que celui du Pont et de Stratégie) en remplaçant l'exigence ''*une méthode couverte par le patron doit être implémentée par toutes les classes présentes dans un chemin*'' par ''*une méthode couverte par le patron doit être implémentée par au moins une classe de chaque chemin vers la*

racine''. Cela signifie, d'une part, le relâchement de la contrainte `Homomorphism_With`, et d'autre part, la modification de la cardinalité « 1..* » associée à la relation `Implements` qui relie `AbstractOperation` et `ConcreteOperation` en une cardinalité « 0..* ». En pratique, rien n'empêche que la racine de la hiérarchie soit une classe concrète et que les méthodes des sous-classes redéfinissent des méthodes concrètes de la racine. De la même façon, des classes intermédiaires qui ne sont ni des racines, ni des feuilles, peuvent être aussi bien abstraites que concrètes, par exemple la classe C2 de la Figure 6.2(b) est une classe intérmédiaire abstraite. Par contre, une classe feuille est obligatoirement concrète.

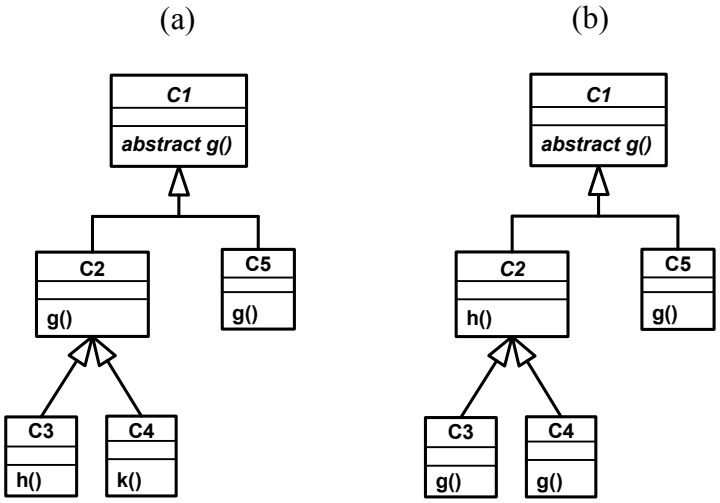


Figure 6.2. Des exemples d'hiérarchies où une méthode abstraitre est implémentée par une seule classe dans chaque chemin de la hiérarchie.

Le modèle de problème obtenu après modification est montré à la Figure 6.3. La partie du modèle qui est représentée avec des tirets (---) regroupe l'ensemble des entités qui n'ont pas nécessairement tous des entités correspondantes dans une instance du problème dans un modèle donné en entrée : la cardinalité de ces entités est « 0..* ».

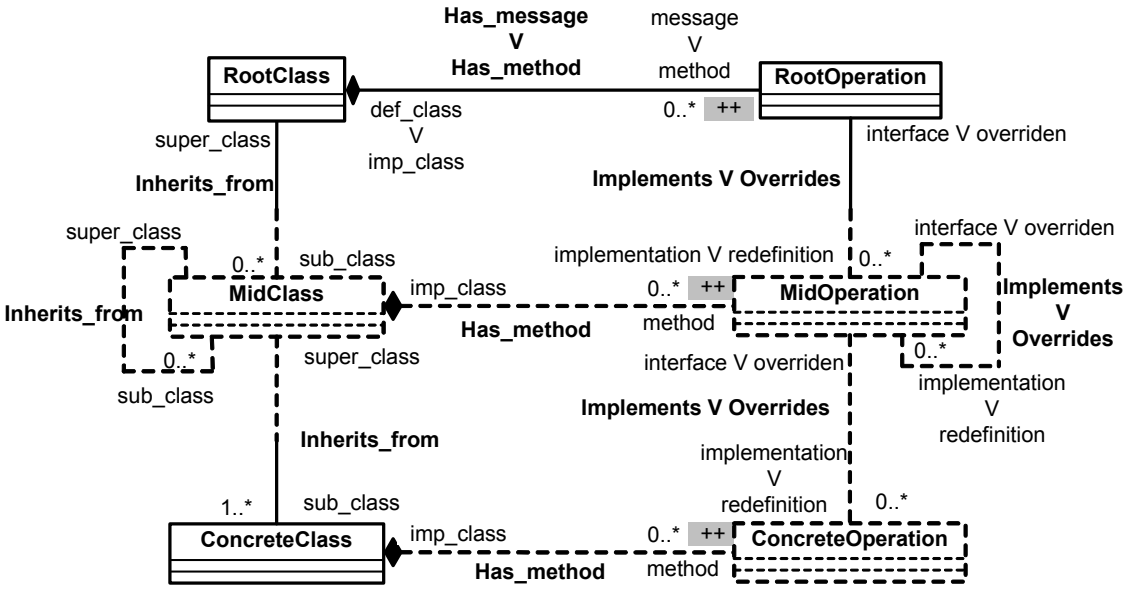


Figure 6.3. Le modèle modifié du problème résolu par le patron Visiteur.

La modification du modèle de problème a nécessité une modification du problème de satisfaction de contraintes associé au patron Visiteur. En fait, l'extraction des variables, des domaines et des contraintes se fait de la même façon que celle décrite au chapitre 4, excepté que nous permettons à certaines variables d'avoir une valeur nulle (e.g. les variables `var_mid_class`, `var_mid_operation` et `var_concrete_operation`). Toutefois, nous avons ajouté des contraintes pour spécifier que pour une branche donnée de la solution et une opération `RootOperation` donnée, au moins une des variables `var_mid_operation` ou une variable `var_concrete_operation` est non nulle, i.e. au moins, une classe du chemin implémente ou redéfinit l'opération `RootOperation` pour qu'elle puisse faire partie du problème. Considérons, par exemple, la hiérarchie de classes de la Figure 6.4. Les contraintes que nous avons spécifiées permettent d'éliminer les chemins (C1, C8, C9) et (C1, C8, C10) car aucune sous-classe de ces deux chemins ne redéfinit la méthode racine (`g()`).

Nous avons aussi modifié l'extracteur du domaine pour générer des relations d'implémentation ou redéfinition qu'elles soient directes ou indirectes.

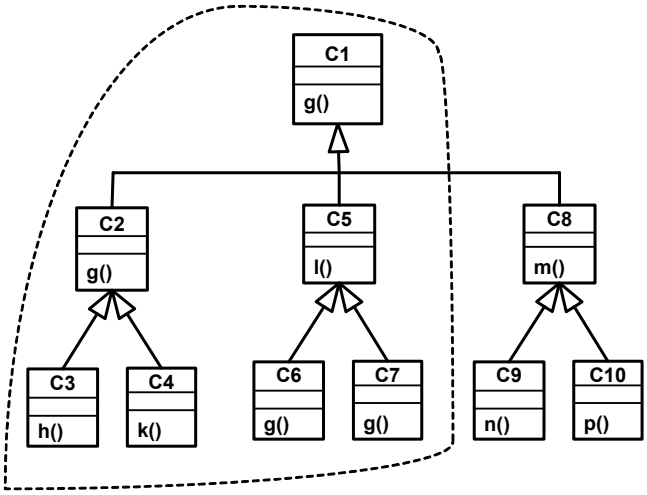


Figure 6.4. Exemple d'hiérarchie où une méthode n'est pas redéfinie dans tous les chemins.

Ainsi, si nous considérons l'exemple de la hiérarchie de classes montrée à la Figure 6.4, notre nouvelle codification du problème de conception retournerait une instance du modèle statique du problème associé au Visiteur: la partie de la hiérarchie qui est entourée de pointillés. Par contre, dans le cas de la hiérarchie de classes de la Figure 6.5, nous détectons deux instances chevauchantes du modèle statique du problème: une instance contenant l'ensemble des classes/opérations {C1(g), C2, C3(g), C4(g), C5(g)}, et une instance contenant l'ensemble des classes/opérations {C2(h), C3(h), C4(h)}.

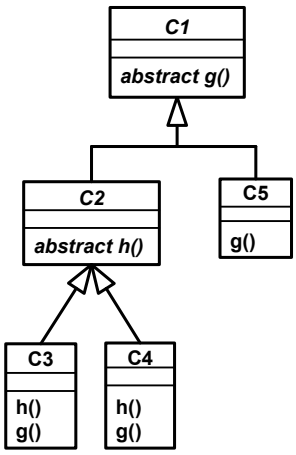


Figure 6.5. Hiérarchie pour illustration d'instances chevauchantes.

6.2.3 Problème de regroupement des instances détectées

La modification de la représentation du problème telle que présentée à la section précédente a nécessité que nous modifions aussi la façon dont nous regroupons les solutions (i.e. instances détectées de problèmes). Nous rappelons que nous récupérons les solutions au fur et à mesure qu'elles sont retournées par le solveur et nous les organisons en une forêt d'arbres. À la fin du processus, chaque arbre de la forêt correspond à une instance différente du problème. Toutefois, suite au relâchement que nous avons fait dans les contraintes du problème, les arbres de la forêt ne correspondent plus nécessairement à des instances indépendantes surtout quand différentes instances se chevauchent.

Pour illustrer ce problème, considérons l'exemple de la Figure 6.6. Selon notre nouvelle codification du problème, et sachant que le regroupement en forêt d'arbres se fait sur la base des classes mais aussi de la signature (i.e. l'ensemble des opérations faisant partie de l'instance détectée), le résultat de notre regroupement des solutions donne les arbres de la Figure 6.7. En fait, ce regroupement consiste à construire pour chaque signature (i.e. ensemble d'opérations) l'arbre maximal.

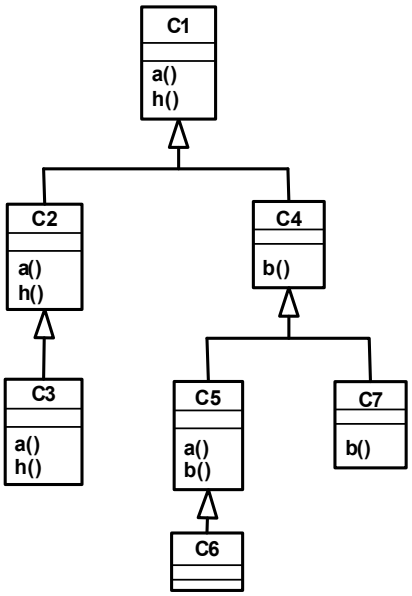


Figure 6.6. Exemple pour illustration du problème de regroupement

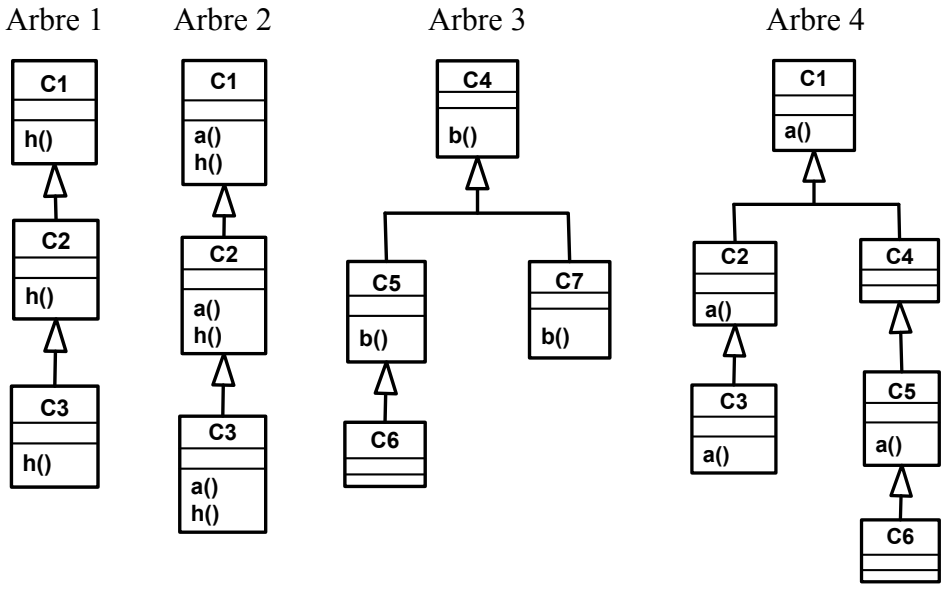


Figure 6.7. Résultats du regroupement par classes et signatures

Pour regrouper correctement les arbres ainsi obtenus, nous avons ajouté de nouvelles règles de regroupement :

1. Lorsque deux arbres contiennent le même sous-ensemble de classes, nous gardons celui ayant la signature maximale, i.e. l'ensemble maximal des méthodes.
2. Lorsque les signatures de deux arbres se chevauchent, i.e. la classe racine est identique et l'intersection des deux ensembles de méthodes n'est pas vide, nous éliminons l'intersection de l'arbre le plus petit et l'attribuons exclusivement à l'arbre le plus grand.

Dans notre exemple de la Figure 6.7, les arbres 1 et 2 sont identiques. En appliquant la première règle, l'arbre 1 est éliminé puisque sa signature est incluse dans celle de l'arbre 2. Ensuite, les signatures des arbres 2 et 4 se chevauchent. Nous appliquons notre deuxième règle qui élimine l'intersection {a} du plus petit arbre qui est l'arbre 2. Le résultat final est montré à la Figure 6.8 : ce sont les différentes instances du modèle statique du problème résolu par le patron Visiteur.

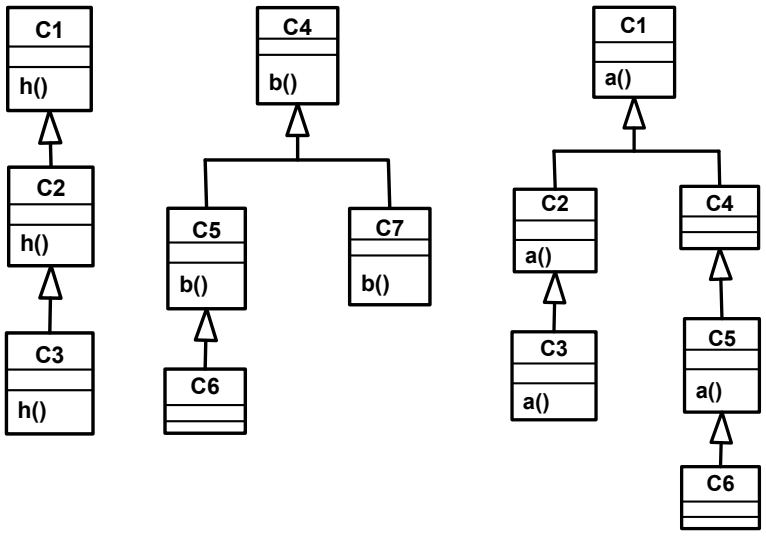


Figure 6.8. Les différentes instances du problème présentes dans l'exemple de la Figure 6.6.

6.3 Validation des aspects détection et transformation

Dans cette section, nous présenterons d'abord notre démarche pour appliquer nos processus de détection et transformation sur de plus gros modèles. Nous discuterons ensuite des problèmes que nous avons rencontrés et qui sont reliés à l'aspect transformation. Finalement, nous présenterons un résumé des résultats de l'ensemble des tests que nous avons réalisés.

6.3.1 Démarche

Pour faire des tests sur de plus gros modèles, nous avons décidé d'utiliser un outil de rétro-ingénierie pour générer des modèles à partir de code source. En effet, actuellement, il existe une panoplie de projets du domaine public dont le code source est disponible. Les modèles ainsi générés sont alors donnés en entrée à notre outil pour l'application des patrons de conception. Pour ce faire, nous avons donc besoin de choisir :

- Un outil de rétro-ingénierie adéquat. Notre choix se base sur trois critères : (i) la disponibilité de l'outil, (ii) le type de modèle généré et, (iii) le degré de conformité du modèle généré au code source.

- Des applications (i.e. du code source) intéressantes par rapport à notre objectif de trouver des problèmes de conception.

6.3.1.1 Choix de l'outil de rétro-ingénierie

Nous avons essayé de trouver un outil qui nous permet, d'une part de générer des modèles XMI pour pouvoir les utiliser facilement avec EMF et d'autre part de faire la rétro-ingénierie sans avoir à apporter beaucoup de corrections aux modèles générés. Il y a plusieurs outils qui permettent de faire de la rétro-ingénierie: ArgoUML, MagicDraw, eclipseUML (anciennement omondo), Objecteering, Together et Rational Rose. La majorité de ces outils offrent une version d'essai. Mais ils ne génèrent pas tous des modèles XMI, et pour ceux qui les génèrent, ils ne sont pas toujours compatibles avec EMF. Nous avons opté pour l'utilisation de Rational Rose car il est entièrement compatible avec EMF.

6.3.1.2 Modifications apportées aux modèles générés

Nous avons été confrontés à quelques problèmes dans la manipulation des modèles générés à partir du code. Nous énumérons ci-après quelques modifications que nous avons apporteés aux modèles générés pour pallier à ces difficultés :

- Problème des agrégations ou compositions: l'outil de réingénierie récupère cette information sous forme d'un attribut de type *List*, *Array*, *Vector* ou autre type de collections. C'est juste en analysant le code que nous pouvons compléter l'information et conclure que c'est une composition ou agrégation et modifier le modèle en créant l'association correspondante et en supprimant l'attribut de type collection.
- Problème de récupération de la relation d'implémentation existant entre une classe et une interface: Cela se produit quand on crée un modèle EMF à partir du modèle XMI généré à partir du code. En fait, cela est dû au fait que dans EMF, une interface est une classe dont l'attribut interface est égal à *true*. Ainsi, une interface implémentée par une classe est ajoutée à la liste `eSuperTypes` de cette classe.

Bien sûr, la taille et la complexité du code influencent les résultats retournés par l'outil de rétro-ingénierie. Ils influencent aussi la complexité du diagramme de classes généré. Pour cela, dans certains cas, nous nous sommes limités à l'analyse d'un noyau significatif de classes par rapport à la fonctionnalité et aux objectifs de l'application considérée. Ainsi, les modèles que nous utilisons pour les tests sont des modèles que nous avons construits en nettoyant ceux générés par l'outil de rétro-ingénierie et en les complétant par l'analyse du code.

6.3.1.3 Choix des applications à analyser

Il y a une multitude de projets libres (*open-source*) en Java sur le web. Nous avons choisi de faire des tests sur des projets plutôt indépendants de EMF/Eclipse. En effet, dans la majorité des plugs-in et frameworks basés sur Eclipse, les patrons de conception sont déjà utilisés (e.g. EMF lui-même, GEF, Draw2d, JhotDraw). Ainsi, nous avons effectué des tests sur des outils tels que BeautyJ, JreversePro, HtmlCleaner et NIL. Nous avons essayé d'analyser différentes versions de ces outils, quand elles sont disponibles. Les versions analysées ainsi qu'une description sommaire de ces outils sont données au Tableau 6-2.

Toutefois, nous nous sommes généralement limités à l'analyse de deux versions par outil car c'est une tâche fastidieuse qui nécessite une analyse manuelle du code associé à chaque version. En effet, les modèles générés à partir du code en utilisant Rational sont incomplets: certaines relations ne sont pas récupérées. En fait, c'est un problème reconnu dans la rétro-ingénierie, certains types de relations ne sont pas facilement récupérables (e.g. composition versus agrégation, relation entre paquetages). Nous avons donc décidé de vérifier et compléter les modèles générés manuellement.

Avant de présenter un résumé des résultats des tests, nous discuterons d'abord des problèmes rencontrés lorsque nous appliquons les transformations inhérentes aux patrons détectés dans les applications analysées. En particulier, nous utiliserons BeautyJ pour illustrer les problèmes résultant du chevauchement des instances de problèmes.

Outil	**Version/ révision**	**Description**	**Taille** (Nombre de classes)
BeautyJ	1.0	BeautyJ analyse et transforme du code Java en un code plus structuré et propre ou en XML. Il permet aussi de générer du code Java à partir d'un fichier Xjava.	53
	1.1		
NIL	20	Un langage de programmation testable, supportant *l'injection de dépendance*, et d'autres bonnes pratiques qui ne sont pas évidentes à coder dans Java.	38
	21		68
JReversePro	1.4.1	Décompilateur/Désassembleur Java	78
	1.4.2		
HtmlCleaner	0.8	Il permet d'analyser et de nettoyer des pages html. Il transforme du HTML en un fichier XML bien formé.	18
	2.1		28

Tableau 6-2. Outils analysés.

6.3.2 Problème de chevauchement des instances détectées de problèmes

Lorsque deux instances de problèmes de conception se chevauchent, il est possible que certaines règles de transformation reliées aux patrons détectés soient en conflit. Pour illustrer cela, nous allons utiliser un extrait du modèle généré à partir de BeautyJ. La Figure 6.9 montre un extrait de la hiérarchie de classes faisant partie du problème résolu par le patron Visiteur. Notre processus de détection a marqué les classes `SourceObject`, `Package`, `SourceObjectDeclared`, `SourceObjectDeclaredVisible` et `Class` et les méthodes `initFromXML` et `initFromAST` comme faisant partie du problème détecté. Les noms de ces entités sont en caractère gras dans la Figure 6.9.

Notre framework a aussi détecté une instance du modèle de problème associé au patron Composite. En effet, en vérifiant le code, nous avons constaté que dans la classe `Package`, il y a des méthodes dans lesquelles il y a différentes boucles pour

traiter les classes (représentées par `Class`) du package courant ou les packages contenus dans le package courant (Figure 6.9). Les objets de type `Package` ou `Class` exhibent en partie un comportement similaire, i.e. les deux classes définissent un ensemble de méthodes ayant la même signature (e.g. `addToPackage`, `initFromXML` et `initFromAST`). Les classes `Package` et `Class` font donc partie de deux instances de problèmes qui se chevauchent: une instance du patron Visiteur et une instance du patron Composite.

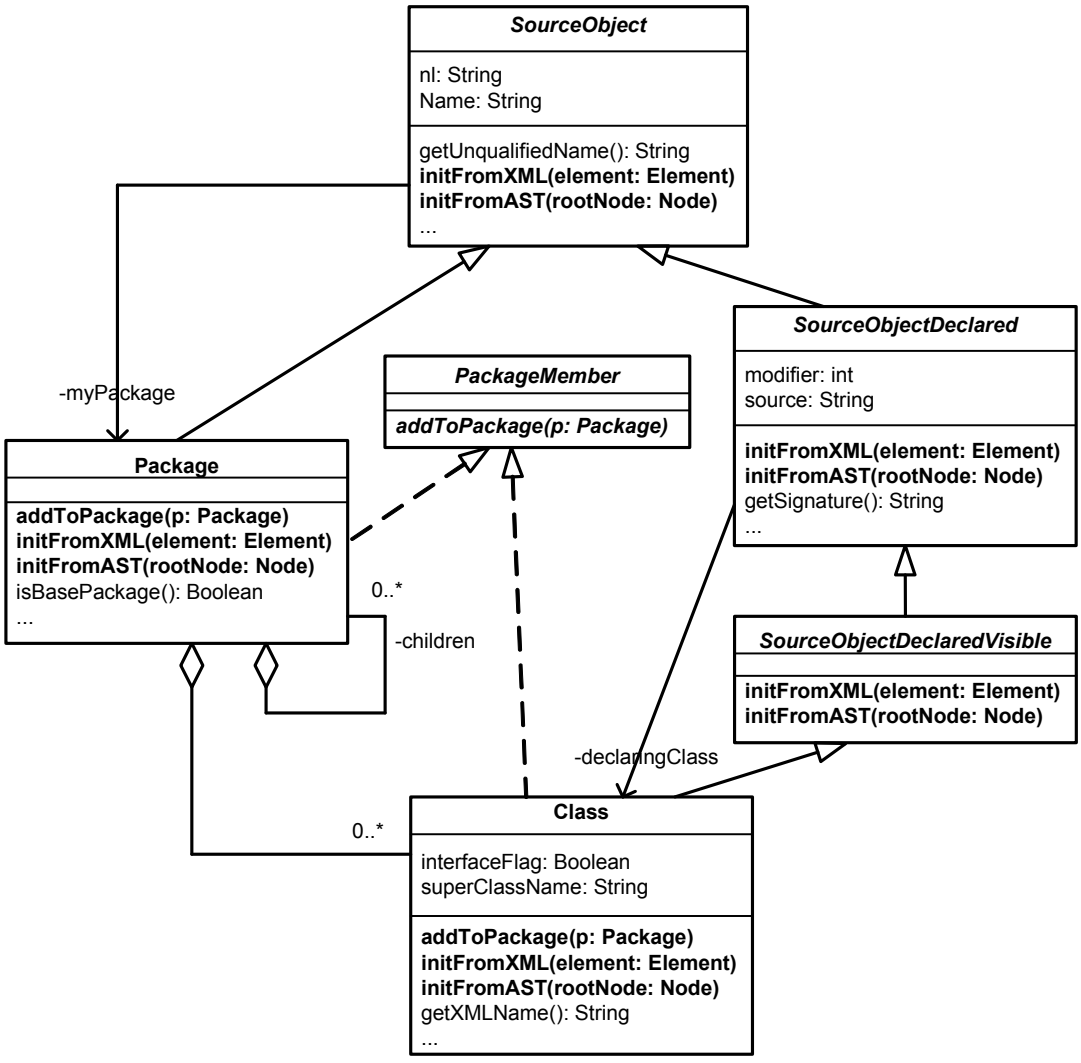


Figure 6.9. Extrait de BeautyJ

Lorsque nous essayons d'appliquer automatiquement les patrons détectés (i.e. Visiteur et Composite) au modèle de la Figure 6.9, notre outil les applique de façon

parallèle et le modèle résultant n'est pas correct. En fait, les règles qui transforment les classes `Package` et `Class` et leurs composantes ne sont pas *parallèlement indépendantes*. Deux transformations peuvent être considérées parallèlement indépendantes si les éléments communs sur lesquels elles vont s'appliquer, sont préservés par les deux transformations (Corradini et al., 1996). Dans le cas où deux transformations ne sont pas parallèlement indépendantes (i.e. elles sont mutuellement exclusives), l'application des deux transformations doit se faire de façon séquentielle.

Nous avons été confrontés à ce problème à chaque fois qu'il y a chevauchement des instances de problèmes de conception. Nous avons décidé d'appliquer les patrons associés aux problèmes qui se chevauchent, un à la fois. La question qui se pose alors est: ''*Dans quel ordre ces patrons doivent-ils être exécutés ?*''. Cette question peut être traduite autrement par: ''*Est-ce que l'application des patrons est symétrique dans le cas de chevauchement* ?''. Dans le cas de BeautyJ, le modèle résultant de l'application du patron Composite suivie de l'application du patron Visiteur est différent du modèle obtenu par l'application du patron Visiteur suivie de l'application du patron Composite.

En effet, l'application automatique dans notre framework du patron Composite au modèle de la Figure 6.9 donne le modèle de la Figure 6.10. Le patron a été correctement appliqué, toutefois, il aurait été plus intéressant d'exploiter l'interface (`PackageMember`) commune aux deux classes concernées par le problème (`Package` et `Class`) au lieu d'introduire une classe supplémentaire. Cependant, nous pensons qu'il est possible de régler ce problème facilement en interagissant avec le concepteur pour pouvoir mieux guider l'application automatique du patron. Par exemple, dans ce cas, le concepteur pourrait spécifier que l'interface `Component` existe déjà et qu'elle correspond à l'interface `PackageMember`. Notre framework, dans ce cas, se limiterait à mettre à jour cette interface de façon à ce qu'elle définisse toutes les méthodes communes aux classes `Package` et `Class` et à remplacer les relations de composition entre `Package` et `Class`, et `Package` et `Package` par une relation de composition entre `Package` et `PackageMember`.

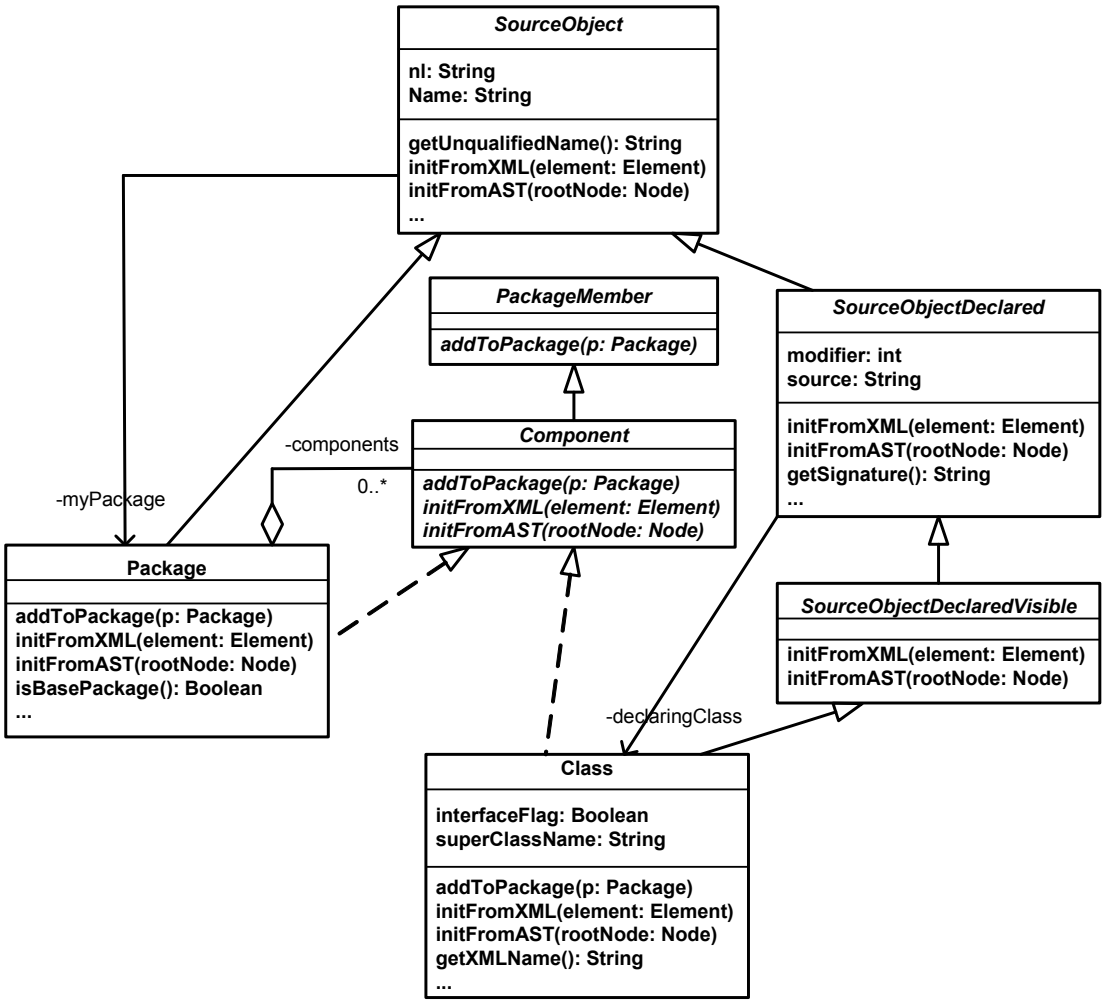


Figure 6.10. Extrait de BeautyJ après application automatique du patron Composite.

Lorsque nous appliquons le patron Visiteur au modèle de la Figure 6.10, nous obtenons le modèle de la Figure 6.11. En fait, la hiérarchie initiale des classes a été modifiée: les méthodes concernées par le problème (`initFromXML`, `initFromAST`) ont été remplacées par la méthode `accept(visitor:SourceObjectVisitor)`. Une hiérarchie de visiteurs a été créée. Un visiteur concret a été généré pour chaque méthode faisant partie du problème. Toutefois, comme l'interface `Component` ne fait pas partie du fragment qui a été marqué comme instance du modèle de problème résolu par le Visiteur, elle n'a pas été modifiée. Ainsi, l'interface définit encore les méthodes `initFromXML` et `initFromAST` alors que les classes implémentant cette interface (`Package` et `Class`) n'implémentent plus cette méthode.

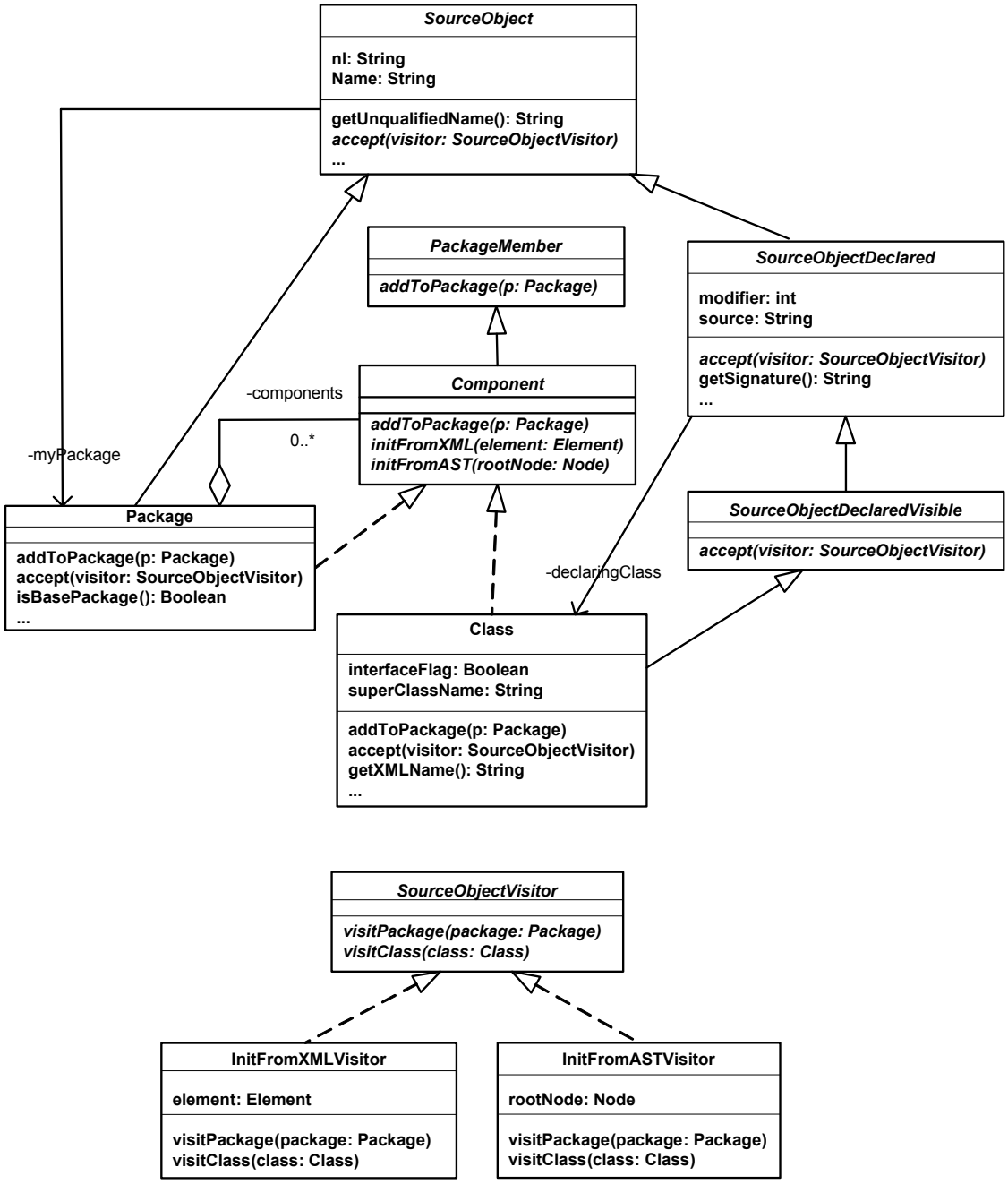


Figure 6.11. Extrait de BeautyJ après application automatique du patron Composite suivi du patron Visiteur.

Ce problème est dû au fait que les transformations consécutives (i.e. celle du Composite suivi de celle du Visiteur) faites sur les classes `Package` et `Class` et leurs méthodes ne sont pas concurrentes. Deux transformations consécutives sont dites

concurrentes si on peut les appliquer dans n'importe quel ordre et que le résultat final reste le même (Corradini et al., 1996). Cela signifie que l'application de l'une ne dépend pas de l'autre. De façon plus explicite, pour que deux transformations consécutives soient considérées concurrentes ou *séquentiellement indépendantes*, une transformation ne doit pas supprimer un élément utilisé par l'autre transformation, et une transformation n'utilise pas un élément généré par l'autre.

Ainsi, en utilisant notre framework, nous avons forcé l'application du patron Visiteur en premier au modèle de la Figure 6.9 suivi du patron Composite, tout en spécifiant que l'interface `Component` existait déjà et qu'elle correspondait à l'interface `PackageMember` du modèle source. Le modèle obtenu est montré à la Figure 6.12. Ce modèle est syntaxiquement correct et il préserve la sémantique du modèle source. Le problème est que, de façon générale, nous ne pouvons pas prévoir *à priori* l'ordre dans lequel les patrons chevauchants doivent être appliqués. Dans notre implémentation actuelle de l'approche, lorsque des instances de problèmes de conception se chevauchent, nous laissons le soin au concepteur de décider de la séquence selon laquelle les patrons détectés doivent être appliqués. D'autres alternatives sont discutées dans les travaux futurs au chapitre 7.

Notons que dans notre framework, les paramètres d'une opération faisant partie du problème résolu par le visiteur (e.g. le paramètre `element` de la méthode `initFromXML`), sont transformés en des attributs du visiteur concret qui a été généré à partir de cette opération (e.g. l'attribut `element` de `InitFromXMLVisitor`). En outre, dans le modèle transformé, nous avons généré dans chaque classe de visiteur concret une méthode pour chaque classe concrète de la hiérarchie à visiter, i.e. chaque visiteur concret implémente deux méthodes `visitPackage` et `visitClass` (Figure 6.12).

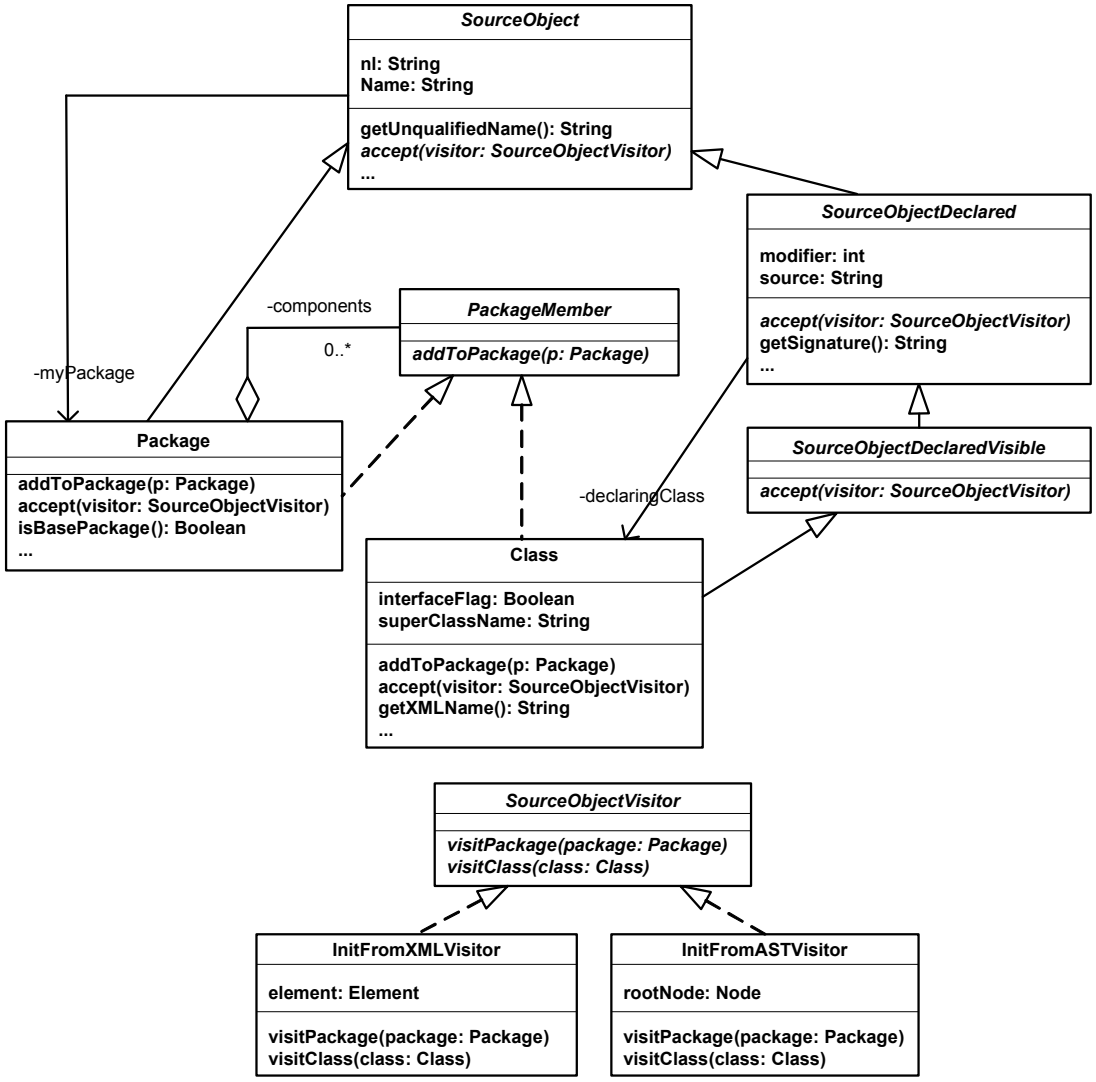


Figure 6.12. Extrait de BeautyJ après application automatique du patron Visiteur suivi du patron Composite.

Cependant, nous pensons que, pour une classe donnée de visiteur concret, il faut plutôt générer une méthode pour chaque classe source dont la méthode associée à ce visiteur est concrète[4]. Par exemple, dans BeautyJ, les méthodes `initFromXML` et

[4] En fait, comme nous l'avons souligné dans la section 6.2, l'exemple utilisé par Gamma et al.(95) pour illustrer le patron Visiteur est trop simple ce qui créé une certaine ambiguité quant à l'application du patron dans des contextes plus complexes.

`initFromAST` étaient des méthodes concrètes des classes abstraites `SourceObject`, `SourceObjectDeclared` et `SourceObjectDeclaredVisible` (Figure 6.9). Ainsi, même si ces classes sont abstraites, nous sommes obligés d'implémenter dans chacune de ces classes une méthode concrète `accept(visitor:SourceObjectVisitor)` et dans chacun des visiteurs des méthodes pour visiter ces classes (Figure 6.13).

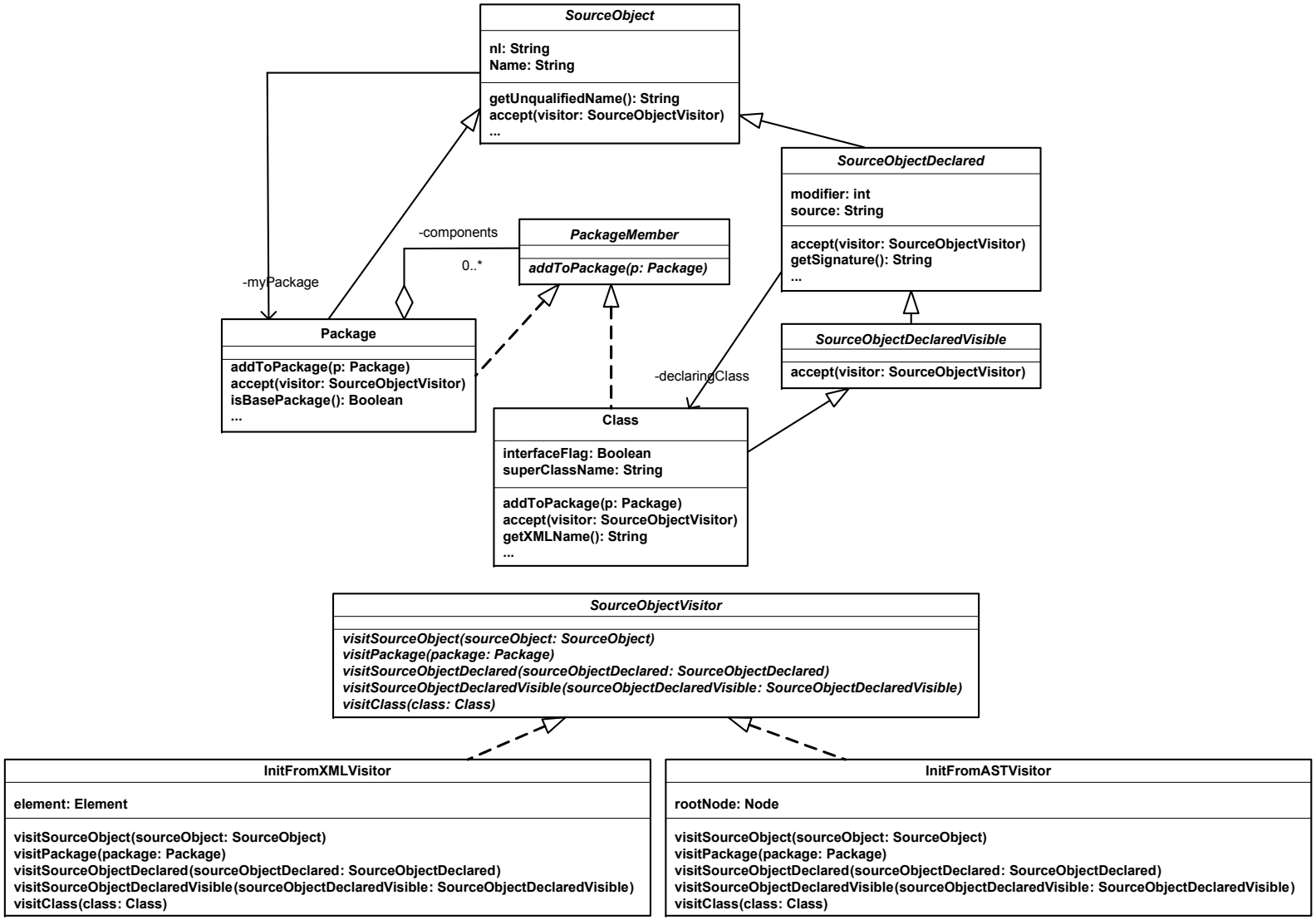


Figure 6.13. Application du patron Visiteur en tenant compte de toutes les méthodes concrètes faisant partie du problème.

6.3.3 Résumé des résultats des tests

Nous avons effectué des tests visant à détecter et à transformer des instances des modèles de problèmes associés aux patrons Pont, Visiteur, Stratégie, Composite et Chaîne de responsabilité. Le Tableau 6-3 résume les résultats des processus de détection et de transformation appliqués aux modèles extraits des applications que nous avons choisi d'analyser.

6.3.3.1 Résultats du processus de détection

Dans le cas de BeautyJ , nous avons détecté deux instances du modèle statique du problème associé aux patrons Visiteur, Pont et Stratégie. Nous avons éliminé une instance car elle correspond à une implémentation du patron *template method*. Pour l'autre instance (Figure 6.9 de la sous-section précédente), notre connaissance du domaine nous suggère qu'il aurait été plus adéquat d'utiliser le patron Visiteur pour éviter la prolifération des méthodes à travers la hiérarchie de classes modélisant les différents composants du code. Notre framework a aussi détecté une instance du modèle de problème associé au patron Composite. Nous avons été en mesure de confirmer le résultat en analysant le code. Notons que pour la partie analyse du code, BeautyJ utilise JJTree qui est un préprocesseur de JavaCC; nous n'avons analysé que la partie réalisée par les développeurs de BeautyJ.

Nous avons aussi détecté dans BeautyJ une structure (Figure 6.14) qui correspond à celle du modèle de problème résolu par le patron Chaîne de responsabilité. Le modèle de problème associé à ce patron décrit l'exigence pour laquelle le patron fournit une solution, i.e. une classe qui définit une méthode pouvant être réécrite par les éléments qui composent cette classe. En fait, une chaîne de classes correspondant à ce problème contient une classe agrégat d'une autre classe qui elle-même est un agrégat d'une autre classe etc., et l'ensemble de ces classes ont au moins une méthode ayant la même signature. Cependant, pour pouvoir confirmer ou non si ce sont des instances du patron Chaîne de responsabilité, il faut analyser le code: si la méthode d'un parent boucle sur ses enfants alors c'est une simple composition, sinon, si la méthode d'un enfant renvoie à la méthode du parent, c'est

une chaîne de responsabilité. Dans le contexte de BeautyJ, c'est la méthode `initFromAST` du parent qui fait appel à celle de ses enfants: c'est une simple chaîne de composition.

Outil	**Problèmes détectés automatiquement**	**Résultat de l'analyse du code**	**Résultats des transformations**
BeautyJ	2 instances du modèle statique de problème des patrons Visiteur, Pont et Stratégie.	1 instance confirmée du problème résolu par le Visiteur.	Les instances se chevauchent : elles sont transformées correctement lorsque le patron Visiteur est appliqué en premier.
	1 instance du Composite.	Oui	
	1 instance de Chaîne de responsabilité.	Non, c'est une simple chaîne de composition.	
NIL	4 instances du modèle statique de problème des patrons Visiteur, Pont et Stratégie.	Le concepteur a utilisé le patron Visiteur dans la révision 21 pour deux de ces instances.	Instances transformées correctement mais ne coïncident pas avec la façon dont le concepteur a appliqué le patron.
JReversePro	1 instance du Composite	Oui	Instance transformée correctement
	3 instances du modèle statique de problème des patrons Visiteur, Pont et Stratégie.	Écartées à partir des versions analysées.	
HtmlCleaner	4 instances du modèle statique de problème des patrons Visiteur, Pont et Stratégie.	2 paires composent la solution du patron Visiteur et 2 paires composant celle du patron Pont.	

Tableau 6-3. Résultats des processus de détection et de transformation

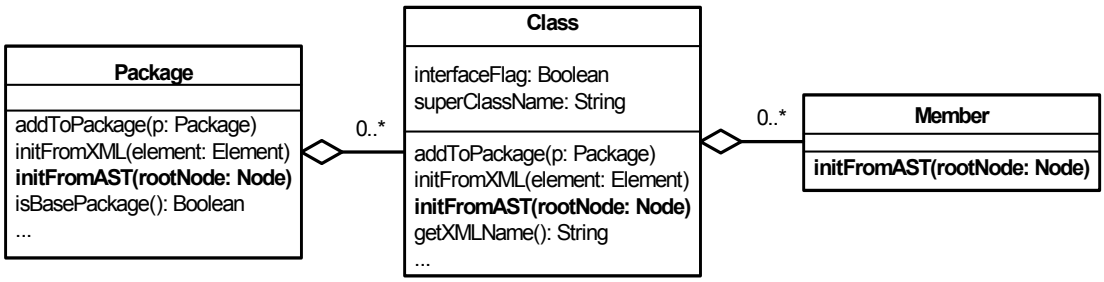


Figure 6.14. Extrait de BeautyJ correspondant à la structure du modèle de problème résolu par le patron Chaîne de responsabilité.

L'analyse de la révision 20 du langage Nil nous a permis de détecter quatre instances du modèle statique du problème associé aux patrons Pont, Visiteur et Stratégie. Deux de ces instances sont effectivement des instances du problème résolu par le patron Visiteur, puisque l'analyse de la révision 21 nous a permis de constater que le concepteur a bien utilisé le patron Visiteur pour ces deux instances.

Dans le cas de JReversePro, nous avons analysé trois paquetages. Nous avons détecté une instance du modèle de problème associé au patron Composite. Nous avons aussi trouvé trois instances du modèle statique du problème résolu par les patrons Visiteur, Pont et Stratégie mais l'analyse du code de différentes versions ne nous a pas permis de conclure à propos de l'évolution de ces instances.

Dans HTMLCleaner, Nous avons détecté quatre instances du modèle statique du problème associé aux patrons Pont, Visiteur et Stratégie. Nous avons analysé le code et nous avons écarté les quatre instances car deux parmi ces instances composaient une instance de la solution proposée par le patron Visiteur alors que les deux autres instances correspondaient à une implémentation du patron Pont. En fait, la structure du modèle statique des problèmes associés aux patrons Pont, Visiteur et Stratégie, i.e. une hiérarchie de classes avec des méthodes redéfinies, est une structure que nous retrouvons aussi dans les instances des solutions proposées par ces patrons et aussi par d'autres patrons tels que le patron de méthode (*template method*).

6.3.3.2 Résultats du processus de transformation

Suite à l'étape de détection, pour les instances de problèmes que nous avons retenues, nous avons utilisé notre framework pour appliquer les transformations inhérentes aux patrons détectés. Les transformations se font de façon correcte quand les instances détectées de problèmes ne se chevauchent pas. C'est le cas des instances détectées dans JReverePro et Nil. En effet, dans ces cas là, les modèles obtenus après application de nos transformations restent conformes au méta-modèle UML. Toutefois, la question qui se pose est: « *Est-ce que le modèle obtenu correspond aux attentes du concepteur* ? ». Cette question est reliée aussi à la question de la correction sémantique du modèle obtenu. Pour répondre à cette question nous avons évalué à chaque fois manuellement le modèle résultant des transformations. Dans le cas de JReversePro, la sémantique a été préservée. Toutefois, dans le cas de Nil, nous avons comparé le modèle résultant de l'application automatique du patron Visiteur par notre framework au modèle généré à partir de la version 21 où le concepteur a appliqué lui-même le patron, et les résultats sont différents. En effet, en se basant sur sa connaissance de l'application, le concepteur a décidé de fusionner les hiérarchies de visiteurs des deux instances.

Dans le cas de BeautyJ, les instances détectées du Visiteur et du Composite se chevauchent. Comme nous l'avons expliqué à la section 6.3.2, il faut imposer un ordre particulier d'exécution aux patrons chevauchants pour obtenir un modèle qui soit syntaxiquement correct et qui préserve la sémantique du modèle source.

6.4 Discussion

Notre étude des patrons de conception nous a permis de représenter un certain nombre de ces patrons et d'utiliser cette représentation pour les appliquer automatiquement. Toutefois, comme nous l'avons souligné à la section 6.2, notre approche pour la représentation des problèmes ne s'applique pas à certains types de patrons tels que les patrons de codage (e.g. le patron de méthode), et les méta-patrons (e.g. le patron Interpréteur). De plus, les illustrations données dans Gamma et al.(1995) des problèmes résolus par les différents patrons permettent généralement

une compréhension limitée de ces problèmes. En effet, nous nous sommes posé plusieurs questions auxquelles nous n'avons pas trouvé de réponse dans le catalogue des patrons de conception. Certaines de ces questions ont été soulevées dans la section 6.2.

Pour les aspects détection et marquage, nous avons pu détecter et marquer des instances de problèmes qui sont exprimés en termes de mauvaises solutions. Quand le modèle de problème est une combinaison de mauvaise solution et de points de variation, nous ne sommes pas en mesure de confirmer que les structures détectées, dans ces cas là, sont de vraies instances du problème à moins de disposer de différentes versions du système analysé, ou d'avoir recours à l'avis d'un spécialiste du domaine analysé. Nous n'avons pas pu détecter certains problèmes lorsque le concepteur a donné des noms différents à des méthodes appartenant à différentes classes mais qui sont *sémantiquement équivalentes*. Nous pensons que si nous complétions notre approche par l'implémentation d'une représentation utilisant les modèles comportementaux des systèmes, nous pourrions améliorer les résultats de notre processus de détection. Ce point est discuté dans les travaux futurs au chapitre 7.

Nonobstant le problème des points de variation, certaines structures (e.g. hiérarchie de classes avec méthodes redéfinies) se retrouvent autant dans des modèles de problèmes que dans des modèles de solutions. Pour éléminer les instances détectées qui font partie des solutions, nous pouvons utiliser le marquage que nous faisons dans les conclusions des règles. En effet, toute entité générée dans la partie RHS d'une règle est marquée par le rôle qu'elle joue dans la solution. Le marquage des solutions est fait de façon analogue à celui des problèmes, i.e. une marque est composée du nom du patron appliqué, du rôle joué par l'entité marquée dans la solution et du numéro de l'instance qui correspond en fait au numéro de l'instance du problème associé.

Pour l'aspect transformation, comme les résultats des tests l'ont démontré, les transformations sont correctes lorsque les instances de patrons ne se chevauchent pas. Dans l'implémentation actuelle de notre approche, nous guidons le transformateur—

lorsque les instances se chevauchent—en lui indiquant l'ordre dans lequel les patrons doivent être appliqués. Ce problème est discuté plus en détails au chapitre suivant dans les travaux futurs.

Dans le cadre de la complétude des transformations (i.e. on transforme tout ce qui doit être transformé), d'une part, nos règles de transformation couvrent toutes les entités définies par le méta-modèle UML et d'autre part, elles couvrent toutes les marques (ou rôles) définies par nos modèles de problèmes. Pour ce qui est de l'incomplétude des modèles sources, dans notre implémentation actuelle de l'approche, nous ne marquons un fragment de modèle que lorsqu'il satisfait toutes les contraintes définies par un modèle de problème. Par conséquent, l'exécution des règles de transformation d'un patron n'est enclenchée que lorsqu'une instance complète du modèle de problème relié au patron est détectée. Toutefois, comme nous visons à offrir un outil qui supporte autant la refactorisation des modèles que l'aide à la conception, nous pensons qu'il serait utile de permettre à un concepteur de marquer manuellement un fragment de modèle qui correspondrait à une instance incomplète d'un modèle de problème (i.e. correspondance partielle entre un fragment de modèle et un modèle de problème), et de modifier les règles de transformation associées aux patrons de façon à ce qu'elles transforment les entités marquées (i.e. la partie présente du problème) et génèrent par défaut la partie manquante de la solution (correspondant à la partie manquante du problème).

Chapitre 7

Conclusion et perspectives

Il y a un consensus dans la communauté du génie logiciel sur l'importance d'utiliser les patrons de conception dans le processus de développement. Cependant, comme nous l'avons souligné au chapitre 2, le support offert par les outils pour utiliser ces patrons est limité. En effet, les approches que nous avons étudiées ne permettent pas d'appliquer un patron à un modèle existant, elles permettent plutôt de générer une structure correspondant à la solution proposée par un patron, que le concepteur peut personnaliser en donnant les noms appropriés aux entités de la structure. Au meilleur de notre connaissance, aucune approche ne s'est penchée sur la représentation des problèmes résolus par ces patrons et ne s'est attaquée au problème de détection de l'opportunité ou du contexte d'application de ces patrons. C'est cela qui distingue notre approche pour la représentation et la mise en œuvre des patrons de conception.

7.1 Contributions

Les travaux présentés dans cette thèse proposent et implémentent une approche pour la représentation et la mise en œuvre des patrons. Cette approche vise à fournir aux développeurs un répertoire de modèles réutilisables spécifiant les patrons et un environnement à base de règles permettant d'automatiser l'application de ces patrons. L'étude des patrons de conception nous a amené à considérer la représentation des problèmes qu'ils traitent comme un élément fondamental pour la compréhension des patrons et l'automatisation de leur application. Ainsi, en

remplacement aux rubriques textuelles que l'on trouve dans la documentation de Gamma et al.(1995), ***nous avons proposé des modèles pour représenter les problèmes résolus par les patrons de conception***.

Sur le plan théorique, notre représentation explicite du problème de conception résolu par un patron sous forme de modèle permet de :

- ***Caractériser d'une façon plus précise et complète un patron et son contexte d'utilisation*** : En proposant de représenter un patron par une paire de modèles (problème, solution), nous décrivons aussi bien l'état d'un modèle *avant* l'application du patron que l'état *après* son application. Cela permet de mieux comprendre les patrons. De plus, notre représentation des problèmes et des solutions met l'accent sur les points de variation. En effet, les patrons de conception visent essentiellement à améliorer la résilience du logiciel face à des scénarios d'évolution. De ce fait, les points de variation représentent un aspect fondamental dans la définition des patrons.
- ***Évaluer la pertinence du patron à un modèle donné en entrée*** : Ayant un modèle explicite du problème résolu par un patron, il devient possible d'en détecter les instances dans un modèle donné en entrée. Pour ce faire, nous avons utilisé les techniques de *pattern matching*.
- ***Mécaniser l'application d'un patron*** : Nous avons pu spécifier de façon déclarative l'application d'un patron en la représentant par une transformation d'une instance du modèle de problème en une instance du modèle correspondant de la solution. Les modèles étant des graphes, l'approche que nous avons proposée pour la spécification des transformations peut être considérée comme une représentation textuelle d'une grammaire de graphes.

Sur le plan pratique, nous avons développé un cadre d'application basé sur les règles et dirigé par les modèles qui permet la représentation et l'application des patrons de conception. Ce cadre inclut :

- ***Un méta-modèle pour représenter les patrons*** : L'examen détaillé des patrons de conception nous a permis de construire un langage pour décrire les modèles de problèmes résolus par ces patrons et les modèles de solutions que ces

patrons prosposent. Ce langage est basé sur le méta-modèle UML auquel nous avons ajouté quelques constructions pour représenter les problèmes de conception. Nous avons implémenté ce langage sous la forme d'un méta-modèle en utilisant EMF™.

- ***Un outil de détection des instances des problèmes de conception*** : Nous avons implémenté une approche semi-automatique pour reconnaître et marquer, dans un modèle donné, les instances des problèmes de conception résolus par les patrons. Nous avons utilisé les techniques de satisfaction de contraintes pour implémenter cet outil. Dans le cas des patrons dont les problèmes incluent des points de variation, la détection se fait de façon semi-automatique puisque le concepteur doit lui-même spécifier les variations potentielles futures, cette information n'étant pas toujours disponible.
- ***Une description basée sur les règles des transformations sous-jacentes aux patrons de Gamma et al.(1995)*** : Nous avons représenté la transformation inhérente à l'application d'un patron par un ensemble de règles de production que nous avons implémentées en utilisant la technologie des moteurs d'inférence.

Notre approche est ***générique*** et ***déclarative*** puisque les règles de transformations sont formulées en termes des éléments des modèles de problèmes et de solutions et non en termes des éléments des modèles auxquels on désire appliquer les patrons.

7.2 Travaux futurs

Nous prévoyons d'explorer, dans le futur, différentes pistes de recherche touchant à différents aspects du processus de conception logicielle. Dans un premier temps, nous aimerions améliorer et compléter certains aspects de notre approche pour la représentation et l'application des patrons. Nous nous intéressons aussi aux problèmes et défis de la conception architecturale avec la perspective d'utiliser les résultats de notre approche pour les patrons de conception pour représenter et appliquer les styles architecturaux. Notre recherche s'inscrit dans une préoccupation

plus globale qui porte sur l'automatisation de la transition des modèles d'analyse vers les modèles de conception.

7.2.1 Représentation et application des patrons

Pour pouvoir améliorer et compléter notre approche, nous prévoyons, d'une part, d'étendre l'aspect représentation et, d'autre part, d'explorer d'autres stratégies pour régler les problèmes que nous avons rencontrés et qui sont reliés aux aspects détection et transformation.

7.2.1.1 Représentation des patrons

Une description plus complète des patrons nécessite la spécification des interactions entre les différentes entités participantes au patron. En effet, l'aspect dynamique est important dans plusieurs patrons, notamment les patrons comportementaux. Nous prévoyons donc d'étendre notre approche pour capturer cet aspect. Pour ce faire, nous pensons qu'une étude approfondie des interactions décrites dans les patrons de conception devrait nous amener à définir un méta-modèle de l'aspect dynamique des patrons. Ce méta-modèle pourrait être implémenté soit en étendant EMF pour incorporer le méta-modèle de la partie dynamique, soit en intégrant une autre notation (e.g. BPEL).

Par ailleurs, nous avons besoin de mieux cerner les catégories de problèmes de conception et de patrons auxquels notre approche peut s'appliquer. Pour cette raison, nous pensons utiliser l'approche pour codifier d'autres types de patrons (e.g. patrons d'analyse) et étendre au besoin notre langage de représentation des patrons. Cela nous permettra, d'une part, de mieux cerner les types de problèmes que nous pouvons représenter et, d'autre part, de voir jusqu'à quelle mesure l'approche peut être utilisée pour d'autres types de patrons. Nous pensons qu'il faudra explorer peut être d'autres paradigmes de représentation pour les patrons auxquels notre approche ne s'applique pas telle quelle.

7.2.1.2 Détection des instances des problèmes de conception

Reconnaître l'opportunité d'utilisation d'un patron de conception demeure un défi comparable au problème de marquage dans le contexte du MDA. L'une des raisons est que les problèmes de conception découlent, en général, d'exigences non-fonctionnelles. Or, ces exigences là ne sont pas exprimées de façon explicite dans les modèles. Nous pensons que la capture de l'aspect dynamique des patrons, de façon manipulable, nous permettra d'étendre notre processus de marquage en exploitant les modèles comportementaux des systèmes pendant la phase de détection.

Dans notre implémentation actuelle du processus de détection, nous détectons des instances complètes d'un modèle de problème donné. Cependant, pour offrir un support d'aide à la conception, nous aimerions pourvoir détecter des instances incomplètes, les transformer et générer les entités manquantes de la solution qui correspondraient aux entités manquantes du problème détecté. Pour ce faire, nous pensons qu'il serait posssible de définir pour un patron donné un modèle minimal de problème et de modifier les règles de transformation associées à ce patron de façon à ce qu'elles transforment les entités marquées correspondant aux entités du modèle minimal de problème et elles génèrent la partie manquante de la solution correspondant aux entités présentes dans le modèle de problème mais pas dans le modèle minimal de problème. Pratiquement, il faudra spécifier quelles entités du modèle de problème sont optionnelles par rapport à la spécification du problème de conception résolu par le patron, i.e. même si ces entités sont supprimées du modèle de problème, ce dernier correspondra encore au problème résolu par le patron considéré. L'ensemble des entités nécessaires et les relations entre ces entités forment le modèle minimal de problème. Ainsi, lorsque nous ne détectons pas d'instance d'un modèle de problème, nous supprimons à partir du système de contraintes correspondant, les variables représentant les entités optionnelles et les contraintes sur ces variables. Nous supprimons les variables, une à la fois, et si nous ne détectons pas toujours d'instance, nous supprimons deux variables à la fois, et ainsi de suite. L'instance qui doit être retenue est celle qui aura la taille maximale.

7.2.1.3 Aspect transformation

Nous avons rencontrés plusieurs problèmes en essayant de transformer des modèles par application automatique des patrons de conception. Ces problèmes sont communs aux approches de transformation de modèles, en général, et à la transformation d'un modèle PIM (*platform independent model*) en un modèle PSM (*platform specific model*) en particulier. Par exemple, la transformation d'un modèle PIM vers un modèle spécifique à la plateforme EJB nécessite l'application de patrons propres à la plateforme.

Dans notre contexte, nous pensons que pour certains problèmes, notamment les problèmes de confluence et de conflits lorsque différentes instances du même patron ou de différents patrons s'appliquent au même fragment d'un modèle, nous pouvons explorer deux pistes pour les régler :

- Élaborer une stratégie de résolution de ces conflits en se basant sur les relations entre les patrons de conception, telles que décrites dans la littérature (Zimmer, 1994).
- Appliquer parallèlement les transformations associées aux patrons et utiliser des règles pour construire correctement le modèle transformé, i.e. utiliser des règles pour fusionner les fragments de modèles résultant des transformations. Marschall et Braun (2003) proposent un certain nombre de règles pour fusionner des fragments de modèles, e.g. deux objets ayant la même identité sont fusionnés en un seul objet, deux associations de même type entre deux objets et ayant les mêmes identités, sont fusionnées pour donner une seule association qui a une cardinalité égale au maximum des deux associations.

7.2.2 Vers une conception architecturale par application de style architectural

La question que nous nous posons est: "*Jusqu'à quel degré notre approche pour les patrons de conception est appropriée pour les styles architecturaux ?*". Nous pensons que certains des résultats obtenus pour les patrons de conception peuvent être réutilisés pour les styles architecturaux. En effet, la représentation des problèmes de

conception résolus par les patrons souligne plusieurs notions qui se retrouvent dans la conception architecturale. Une des notions clés est la notion de point de variation (*time hotspot*) qui est aussi une notion importante dans une architecture où nous visons des qualités reliées à l'évolution du système telles que la maintenabilité et l'adaptabilité.

En pratique, la conception architecturale a ses propres défis et notre démarche pour la mise en œuvre des patrons n'est pas appropriée aux styles architecturaux pour certaines tâches. En effet, dans la conception architecturale nous avons besoin de distinguer entre la sélection d'un style architectural (choix des types de composants et de connecteurs) et la détermination des frontières des composants pour le style choisi. La sélection d'un style architectural est souvent faite en se basant sur une liste de qualités désirées telles que la sécurité, la performance, la maintenabilité, etc. (Bass et al., 2003). L'approche utilisée pour la sélection des patrons n'est donc pas appropriée pour la sélection des styles architecturaux. En outre, notre expérience avec les patrons de conception a montré qu'une distinction est nécessaire entre la description d'un problème de conception qui représente une exigence pour laquelle il faut trouver une solution et que nous ne sommes pas toujours en mesure de détecter et une description d'une mauvaise solution qui peut être reconnue dans le modèle source et traitée. Or un style architectural n'est pas appliqué pour corriger une mauvaise architecture, i.e. il n'y a pas de "point de départ " comme dans le cas de patrons.

Cependant, une fois le style architectural choisi, nous pouvons utiliser les propriétés structurelles du modèle à l'entrée pour déterminer les frontières des composants. Notre infrastructure de transformation peut être utilisée comme base pour permettre la correspondance structurelle entre le modèle en entrée et un style architectural. De plus les connecteurs d'un style architectural sont des patrons d'interaction et peuvent donc être implémentés par des patrons de conception.

Liste des publications

Conférences internationales avec comité de lecture

Ghizlane El boussaidi, Hafedh Mili, "Detecting Patterns of Poor Design Solutions Using Constraint Propagation". Proceedings of the ACM/IEEE 11th Intern. Conference on Model Driven Engineering Languages and Systems (MODELS 2008), Lecture Notes in Computer Science, vol. 5301, pp. 189-203, Toulouse, September 28-October 3, 2008.

Ghizlane El boussaidi, Hafedh Mili, "A model-driven framework for representing and applying design patterns". Proceedings of 31st IEEE International Computer Software and Applications Conference, vol. 1, pp. 97-100, Beijing, July 23-27, 2007.

Ghizlane El boussaidi, Hafedh Mili, "Une approche à base de règles pour la mise en oeuvre des patrons de conception". Proceedings of 8th Intern. Symposium on Programming and Systems, pp. 96-107, Algiers, May 7-9, 2007.

Hafedh Mili, Ghizlane El boussaidi, "Representing and applying design patterns: what is the problem?". Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS 2005), October 2–7, 2005, Half Moon Resort, Montego Bay, Jamaica. Lecture Notes in Computer Science, vol. 3713, pp. 186-200

Hafedh Mili, Ghizlane El boussaidi, Aziz Salah, "Mise en oeuvre des patrons de conception par représentation explicite du problème". 11ième Conference sur les langages et modèles à objets (LMO), 9-11 mars, 2005, Berne, Swisse. Journal de L'objet, vol. 11, pp. 113-126

H. Mili, A. Leshob, E. Lefebvre, G. Lévesque, Ghizlane El Boussaidi, "Towards a methodology for representing and classifying business processes". Proceedings of the 4th International MCETECH Conference on e-

Technologies, May 4-6, 2009, Ottawa, Canada. Lecture Notes in Business Information Processing, vol. 26, pp. 196-211

N. Moha, J. Rezgui, Y-G. Guéhéneuc, P. Valtchev, Ghizlane El Boussaidi, "Using FCA to Suggest Refactorings to Correct Design Defects". Proceedings of the 4th International Conference On Concept Lattices and Their Applications (CLA 2006), October 30 - November 1st, 2006, Hammamet, Tunisia. Lecture Notes in Computer Science, vol. 4923, pp. 269-275

Rapports techniques

Ghizlane El boussaidi, Hafedh Mili, "Pattern Matching within model-driven approaches: strategies and problems", rapport technique du LATECE, novembre 2007, Montréal, Canada.

Ghizlane El boussaidi, Hafedh Mili, "Les langages de description d'architectures", rapport technique du LATECE, octobre 2006, Montréal, Canada.

Ghizlane El boussaidi, "Les approches de transformation de modèles basées sur les transformations de graphe", rapport technique du LATECE, juin 2006, Montréal, Canada.

Ghizlane El Boussaidi, Hafedh Mili, "Les patrons de conception : représentation et mise en œuvre", rapport technique du LATECE, avril 2004, Montréal, Canada.

Annexe 1

Représentation du patron Pont

Exemple

Considérons un programme qui manipule des images. Sachant que l'implémentation d'une image varie, ce programme doit être portable d'un système d'exploitation à un autre. En utilisant l'héritage (Figure A1.1(a)), on peut définir une classe abstraite `Picture` et des sous-classes `WinPicture` et `OS2Picture` qui implémentent l'image pour différentes plateformes. Pour pouvoir définir un type d'image, par exemple « GIF », nous devons créer une nouvelle classe `GIFPicture` (Figure A1.1(b)) qui est une sous-classe de `Picture` et définir une implémentation de `GIFPicture` par plateforme. De la même façon, pour pouvoir supporter une nouvelle plateforme, il faudra ajouter une nouvelle sous-classe pour chaque type d'image. Ainsi, cette approche ne permet pas de modifier, étendre et réutiliser les abstractions et les implémentations d'une façon indépendante.

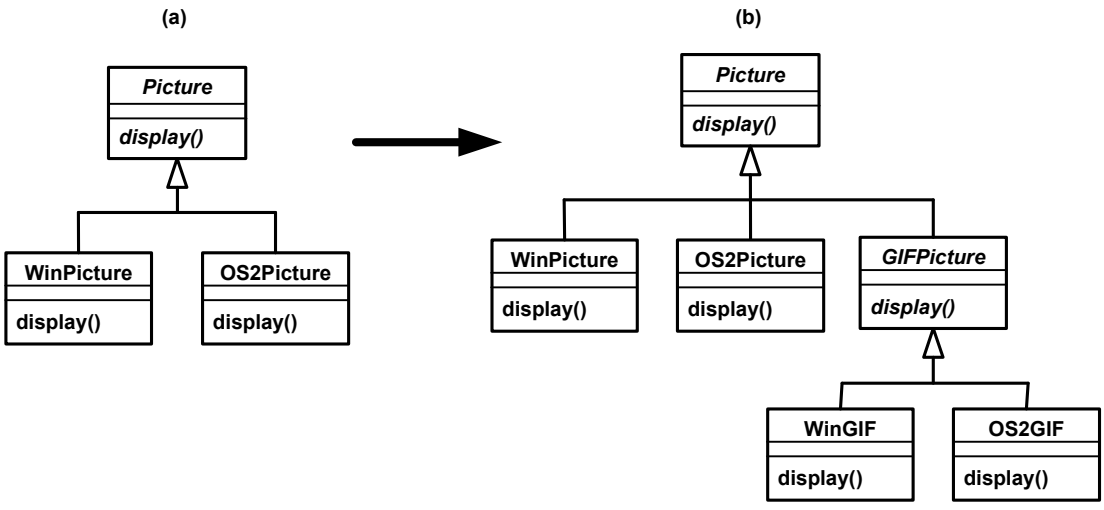


Figure A1.1. Un exemple du problème résolu par le patron Pont.

Le patron Pont propose de mettre les abstractions et les implémentations dans deux hiérarchies de classes séparées. Ces hiérarchies peuvent évoluer indépendamment l'une de l'autre. La structure du patron telle que présentée dans Gamma et al.(1995) est montrée à la Figure A1.2. En appliquant le patron Pont à notre exemple de la Figure A1.1, nous obtenons le modèle de la Figure A1.3. Il y a donc une hiérarchie de classes dont la racine est l'abstraction `Picture` et une autre hiérarchie dont la racine est `PictureImpl` laquelle représente les implémentations de l'image selon les plateformes.

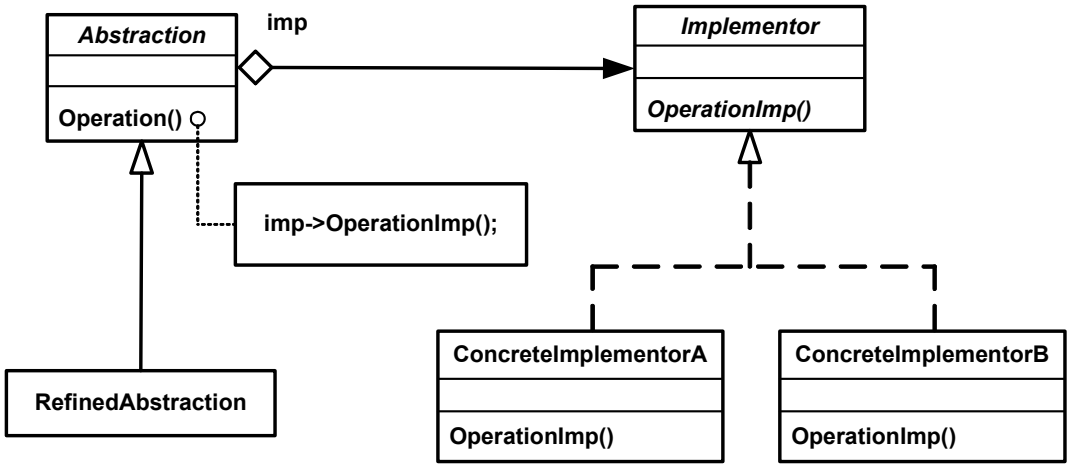


Figure A1.2. La structure du patron Pont.

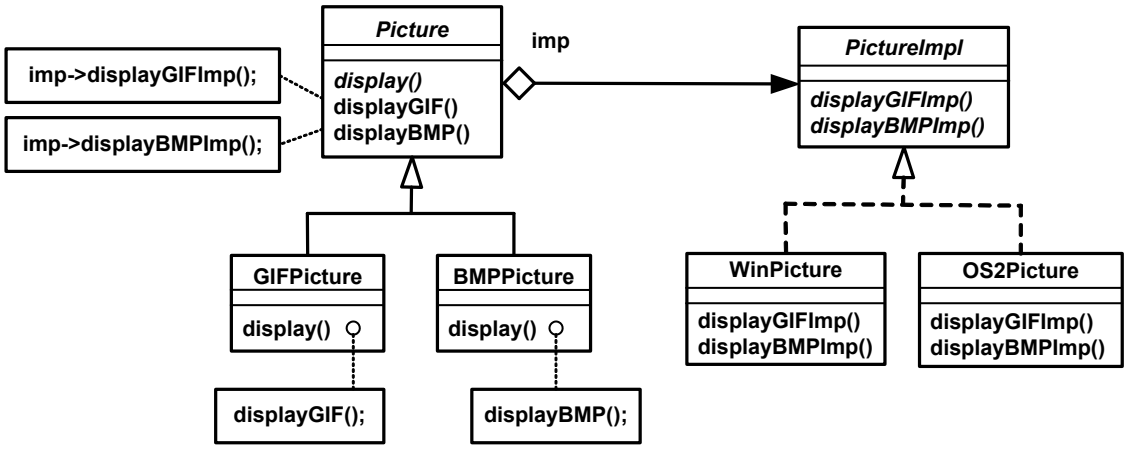


Figure A1.3. Exemple d'application de la solution proposée par le patron Pont

Le modèle du problème

Pour représenter le modèle du problème associé au patron Pont, nous avons utilisé une approche similaire à celle que nous avons utilisée pour représenter les problèmes de conception résolus par les patrons Visiteur et Composite. Ce modèle est présenté à la Figure A1.4.

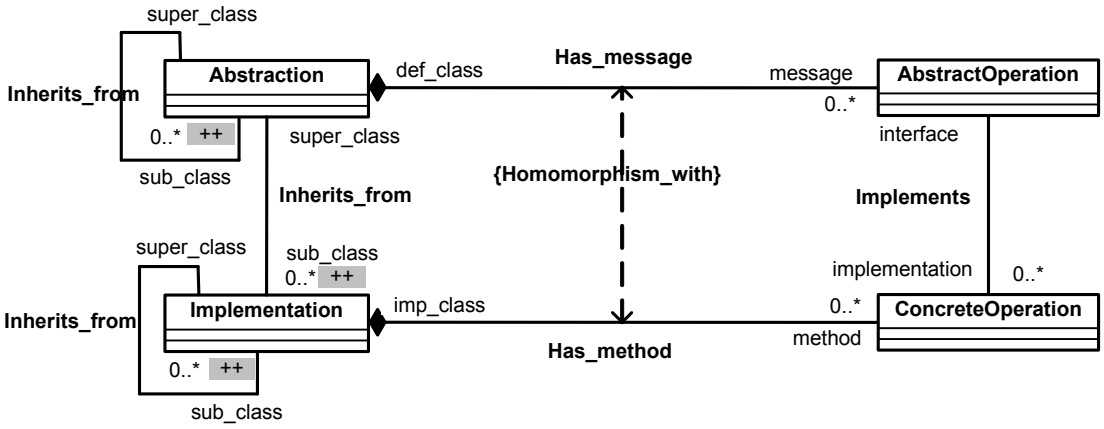


Figure A1.4. Modèle du problème résolu par le patron Pont

Les classes `Abstraction` et `Implementation` sont des méta-classes dans le sens que leurs instances sont des classes telles que `Picture` ou `WinPicture`, respectivement. Les associations `Inherits_from` représentent les relations d'héritage qui doivent relier les instances des classes correspondantes. Par exemple, l'abstraction `GIFPicture` (voir Figure A1.1) hérite de l'abstraction `Picture`. De même, l'implementation `WinGIF` hérite de la classe `GIFPicture`, qui est une abstraction. Dans l'exemple de la Figure A1.1, nous n'avions pas de situation de plusieurs niveaux d'implémentation, mais ceci est envisageable.

Nous avons aussi représenté dans notre modèle du problème les opérations qui sont affectées par l'application du patron. Les opérations des « `Abstractions` » vont être abstraites, et celles des « `Implementations` » vont être concrètes. De plus, chaque classe `Implementation` doit implémenter toutes les opérations abstraites des classes `Abstraction` qu'elle implémente. Ceci est représenté dans le modèle (Figure A1.4) comme une contrainte appelée `Homomorphism_with` entre

les associations `Has_message` et `Has_method`. Les opérations non affectées par l'application du patron ne sont pas représentées dans le modèle.

En fait, comme nous l'avons illustré dans notre exemple (Figure A1.1), le nombre d'abstractions et d'implémentations peut augmenter. C'est cette variation qui présente un problème et que le patron Pont résout. Nous avons représenté ces points de variation par l'ajout du symbole « ++ » aux cardinalités des associations concernées (i.e. les relations `Inherits_from`). Ainsi, le modèle du problème montre que le nombre d'abstractions est susceptible d'évoluer ainsi que le nombre d'implémentations par abstraction (e.g. le nombre de plateformes).

Le modèle de la solution

Le patron « Pont », comme nous l'avons déjà vu, propose une solution qui consiste à découpler une abstraction de son implémentation de façon à ce que les deux puissent varier indépendamment. Le modèle de solution que nous proposons pour le patron Pont est montré à la Figure A1.5.

Nous lisons ce modèle de la façon suivante : nous avons une hiérarchie de classes (`RootAbstraction` et `RefinedAbstraction`), qui délègue ses traitements à une autre hiérarchie de classes (`RootImplementor` et `ConcreteImplementor`). Les symboles « ++ » des cardinalités des associations `Inherits_from` qui relient respectivement `RefinedAbstraction` à `RootAbstraction` et `ConcreteImplementor` à `RootImplementor` signifient que le nombre de classes de type `RefinedAbstraction` et le nombre de classes de type `ConcreteImplementor` sont susceptibles d'augmenter, i.e. d'autres abstractions et implémentations vont probablement s'ajouter ultérieurement.

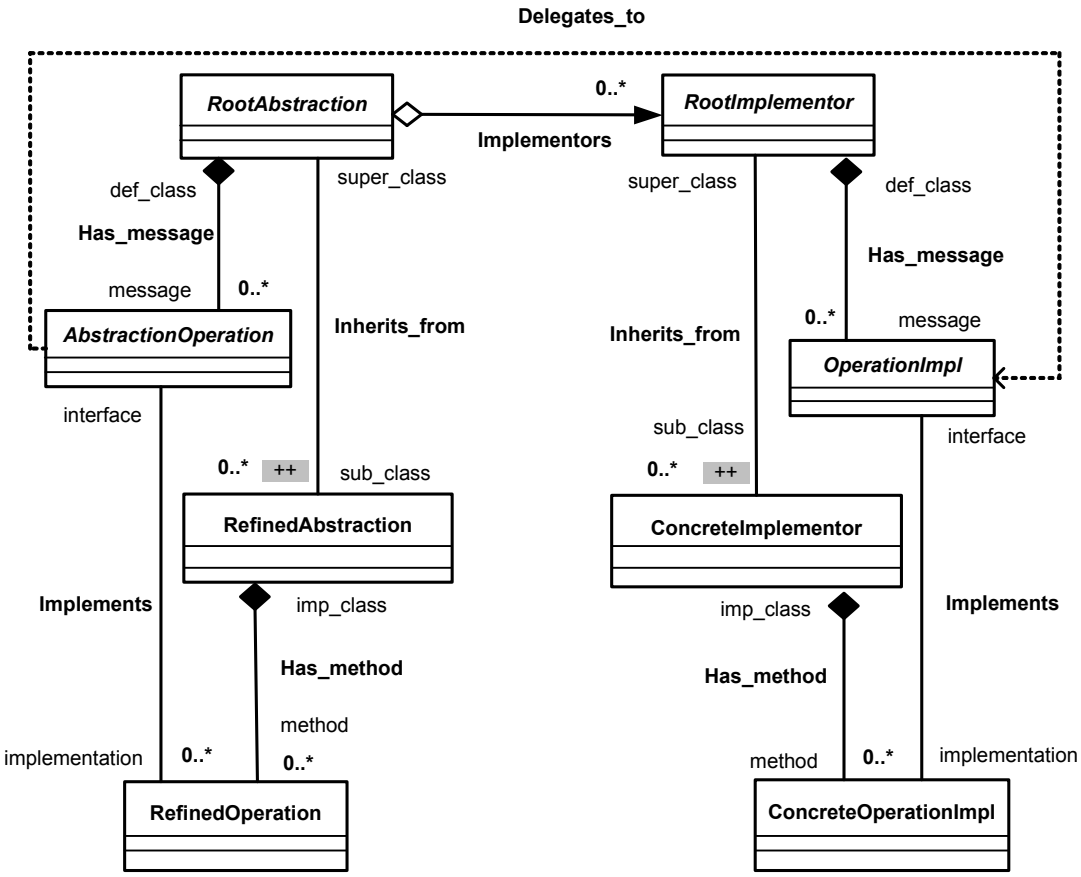
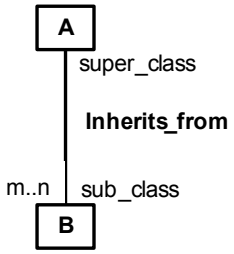
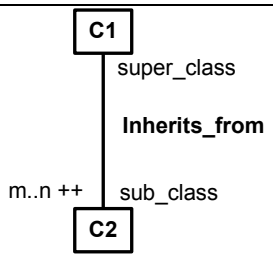
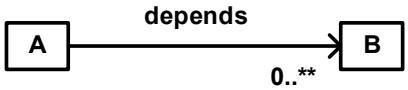
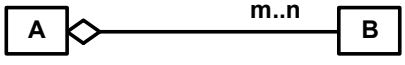
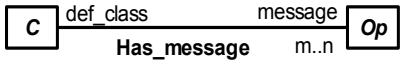
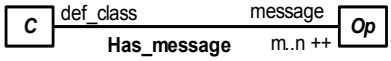
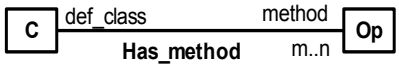
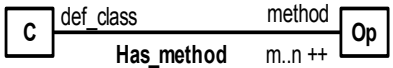
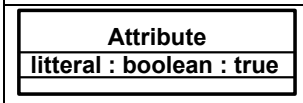
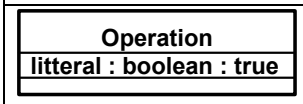
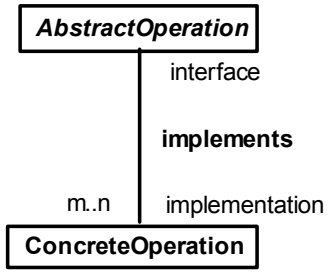


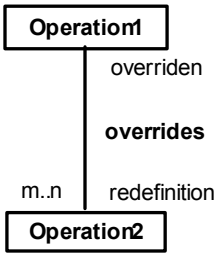
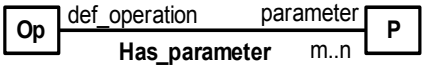
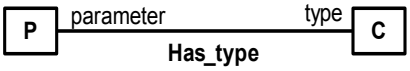
Figure A1.5. Modèle de la solution proposée par le patron Pont

Annexe 2

Résumé de la notation

	Il peut y avoir m à n (n≥m≥0) classes de type B qui héritent d'une classe de type A. Ceci est une configuration que l'on retrouve dans plusieurs patrons dont le patron Pont, Fabrication abstraite, Visiteur, Stratégie, etc.
	Il peut y avoir m à n (n≥m≥0) classes de type B qui héritent d'une classe de type A. Ce nombre est susceptible d'augmenter. C'est le cas dans le problème de conception résolu par le patron Pont.
	Une classe de type A dépend d'un nombre trop élevé de classes de type B. Une classe dépend d'une autre dans le sens qu'elle utilise, communique avec, ou connaît l'autre classe. «0.. **» signifie une cardinalité trop élevée ou une interaction trop complexe, i.e. un objet est couplé à un trop grand nombre d'objets. Ceci est un symptôme que l'on retrouve dans les problèmes traités par les patrons Médiateur et Chaîne de responsabilité.
	Une classe de type A est composée d'un ensemble de classes (m à n avec n≥m≥0) de type B. C'est une configuration que l'on retrouve dans le problème de conception résolu par le patron Composite, i.e. une classe composite est composée de plusieurs classes feuilles.

	Une classe abstraite *C* qui définit un certain nombre (m à n avec n≥m≥0) d'opérations abstraites *Op*. Une telle configuration se présente, par exemple, dans les cas des patrons Pont et Stratégie.
	Une classe abstraite *C* qui définit un certain nombre (m à n avec n≥m≥0) d'opérations abstraites *Op*. Ce nombre est susceptible d'augmenter. C'est le cas de notre exemple du patron Visiteur.
	Une classe concrète C qui implémente un certain nombre d'opérations concrètes Op.
	Une classe concrète C qui implémente un certain nombre d'opérations concrètes Op. Ce nombre est susceptible d'augmenter. C'est le cas de notre exemple du patron Visiteur.
	Un attribut littéral. Cet attribut sera ajouté littéralement (i.e. tel quel) à la classe qui joue le rôle de la classe qui contient cet attribut dans le modèle de solution.
	Une opération littérale. Cette opération sera ajoutée littéralement (i.e. telle quelle) à la classe qui joue le rôle de la classe qui contient cette opération dans le modèle de solution.
	L'opération abstraite `AbstractOperation` est implémentée par m à n (n≥m≥0) opérations concrètes `ConcreteOperation`.

	L'opération concrète `Operation1` est redéfinie par m à n (n≥m≥0) opérations concrètes `Operation2`.
	L'opération `Op` a m à n (n≥m≥0) paramètres P.
	Le paramètre P a le type C.

Annexe 3

Spécification et génération du méta-modèle

Nous avons choisi de définir notre méta-modèle en utilisant des interfaces Java. Nous avons créé pour chaque classe du méta-modèle une interface annotée en Java. EMF peut déduire les attributs et références du méta-modèle en se basant sur les annotations et les méthodes *get()* contenues dans ces interfaces.

Dans la Figure A3.1, nous pouvons voir l'exemple de l'interface ***ModelClass*** qui étend l'interface `EClass` du modèle EMF. Le premier tag `@model` (ligne 5) qui apparaît dans cette interface spécifie que ***ModelClass*** fait partie du modèle EMF correspondant à notre méta-modèle, et que EMF doit générer une classe d'implémentation pour cette interface. En fait, pour faire partie de la définition du méta-modèle, une interface ou une méthode doit être marquée explicitement par l'annotation `@model` dans le commentaire Javadoc. EMF associe au tag `@model` un certain nombre de propriétés qui permettent, au besoin, de spécifier des informations sur les éléments qui ont été annotés.

Le second tag (ligne 9) (`@model type="ModelOperation" containment="true"`) indique que cette classe a une référence sur une autre classe du méta-modèle de type ***ModelOperation***. Cette référence est de type «plusieurs», cela étant spécifié par le fait qu'elle retourne une liste (`EList<ModelOperation> getClassOperations()`). La propriété `containment="true"` indique qu'une instance de ***ModelClass*** contient des instances de ***ModelOperation***. De la même façon, nous avons défini deux autres annotations qui spécifient des références, une sur ***ModelAttribute*** (ligne 13) et une sur ***ModelReference*** (ligne 17). Les deux références ont leur attribut `containment` égal à *true*. Le dernier tag `@model` (ligne 21) spécifie que

l'opération ***isModelSuperTypeOf*** fait partie du méta-modèle. Elle a un paramètre de type ***ModelClass*** et retourne un *booléen* comme résultat.

```
1.  package org.eclipse.PatternMetamodel;
2.  import org.eclipse.emf.common.util.EList;
3.  import org.eclipse.emf.ecore.EClass;
4.  /**
5.   * @model
6.   */
7.  public interface ModelClass extends EClass {
8.     /**
9.      * @model type="ModelOperation" containment="true"
10.     */
11.    EList<ModelOperation> getClassOperations();
12.    /**
13.     * @model type="ModelAttribute" containment="true"
14.     */
15.    EList<ModelAttribute> getClassAttributes();
16.    /**
17.     * @model type="ModelReference" containment="true"
18.     */
19.    EList<ModelReference> getClassReferences();
20.    /**
21.     * @model
22.     */
23.    boolean IsModelSuperTypeOf(ModelClass someModelClass);
24. }
```

Figure A3.1. L'interface ModelClass

Les interfaces annotées que nous avons définies, sont alors données en entrée au générateur EMF pour générer un modèle EMF dont l'extension est «`.ecore`» et un modèle dont l'extension est «`.genmodel`» qui servira, par la suite, à la génération des classes d'implémentation. Nous avons appelé le méta-modèle généré à partir de nos interfaces ***PatternMetamodel***. Deux fichiers ont été donc générés : ***PatternMetamodel.ecore*** qui est une extension à `Ecore` mais aussi une instance de `Ecore` (i.e. `Ecore` se décrivant lui-même), et ***PatternMetamodel.genmodel*** qui est un modèle intermédiaire pour la génération de code. Notre méta-modèle ***PatternMetamodel.ecore*** est sérialisé sous format XMI (voir annexe 4).

En utilisant le générateur EMF et à partir du modèle ***PatternMetamodel.genmodel***, nous avons généré les classes d'implémentation. Le code généré (i.e. implémentation de notre méta-modèle) est organisé en deux paquetages: un paquetage ***org.eclipse.PatternMetamodel*** qui contient l'ensemble des interfaces Java qui représentent le méta-modèle alors que l'autre paquetage ***org.eclipse.PatternMetamodel.impl*** contient les classes d'implémentation. Les interfaces générées sont en fait les interfaces que nous avons créées initialement, augmentées des méthodes « *setter* ». Chaque classe de notre méta-modèle est ainsi représentée par une interface et une classe d'implémentation. Le code généré pour une classe inclut les méthodes d'accès (`get` et `set`) pour chaque attribut et référence qui appartient à cette classe. En ce qui concerne les opérations spécifiées dans les interfaces annotées (e.g. l'opération ***IsModelSuperTypeOf*** de ***ModelClass***), EMF génère les corps vides des méthodes laissant au concepteur le soin de compléter l'implémentation manuellement.

De plus, EMF a généré deux interfaces additionnelles et leurs classes d'implémentation. La première interface ***PatternMetamodelPackage***, implémentée par la classe ***PatternMetamodelPackageImpl***, correspond à un conteneur de tout le méta-modèle. La seconde interface ***PatternMetamodelFactory*** correspond à l'interface d'une classe de fabrication (*Factory*) qui est associée au conteneur ***PatternMetamodelPackage***. Elle est implémentée par la classe ***PatternMetamodelFactoryImpl*** qui sert à créer des instances des classes de notre méta-modèle. L'interface ***PatternMetamodelFactory*** est montrée à la Figure A3.2.

```
public interface PatternMetamodelFactory extends EFactory {
   //L'instance Singleton de la classe de fabrication
   PatternMetamodelFactory eINSTANCE =
       org.eclipse.PatternMetamodel.impl.PatternMetamodelFactoryImpl.init();
   //Retourne une nouvelle instance de la classe ModelPackage
   ModelPackage createModelPackage();
   //Retourne une nouvelle instance de la classe ModelClass
   ModelClass createModelClass();
   //Retourne une nouvelle instance de la classe LiteralClass
   LiteralClass createLiteralClass();
   //Retourne une nouvelle instance de la classe ParameterClass
   ParameterClass createParameterClass();
   //Retourne une nouvelle instance de la classe ModelOperation
   ModelOperation createModelOperation();
   //Retourne une nouvelle instance de la classe ModelAttribute
   ModelAttribute createModelAttribute();
   //Retourne une nouvelle instance de la classe ModelReference
   ModelReference createModelReference();
   //Retourne une nouvelle instance de la classe ImplementsReference
   ImplementsReference createImplementsReference();
   //Retourne une nouvelle instance de la classe InheritsReference
   InheritsFromReference createInheritsFromReference();
   //Retourne le méta-modèle supporté par cette fabrication
   PatternMetamodelPackage getPatternMetamodelPackage();
}
```

Figure A3.2. L'interface PatternMetamodelFactory

PatternMetamodelFactory spécifie un singleton par la variable ***eINSTANCE***. Cette interface déclare aussi une méthode spécifique ***create<nom-de-classe>*** pour chaque classe de notre méta-modèle. Enfin, elle définit une méthode d'accès ***getPatternMetamodelPackage()*** à notre méta-modèle.

Annexe 4

Notre méta-modèle sous format XMI

Le modèle EMF de notre méta-modèle PatternMetamodel.ecore sérialisé sous format XMI :

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ecore=http://www.eclipse.org/emf/2002/Ecore name="PatternMetamodel"
      nsURI="http:///org/eclipse/PatternMetamodel.ecore"
      nsPrefix="org.eclipse.PatternMetamodel">
   <eClassifiers xsi:type="ecore:EClass" name="ModelPackage"
      eSuperTypes="platform:/plugin/org.eclipse.emf.ecore/model/Ecore
      .ecore#//EPackage">
      <eStructuralFeatures xsi:type="ecore:EReference"
         name="modelClasses" lowerBound="1" upperBound="-1"
         eType="#//ModelClass" containment="true"
         resolveProxies="false"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass" name="ModelClass"
      eSuperTypes="platform:/plugin/org.eclipse.emf.ecore/model/Ecore
      .ecore#//EClass">
      <eOperations name="IsModelSuperTypeOf" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EBoolean">
         <eParameters name="someModelClass" eType="#//ModelClass"/>
      </eOperations>
      <eStructuralFeatures xsi:type="ecore:EReference"
         name="classOperations" upperBound="-1"
         eType="#//ModelOperation" containment="true"
         resolveProxies="false"/>
```

```
   <eStructuralFeatures xsi:type="ecore:EReference"
      name="classAttributes" upperBound="-1"
      eType="#//ModelAttribute" containment="true"
      resolveProxies="false"/>
   <eStructuralFeatures xsi:type="ecore:EReference"
      name="classReferences" upperBound="-1"
      eType="#//ModelReference" containment="true"
      resolveProxies="false"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ModelAttribute"
   eSuperTypes="platform:/plugin/org.eclipse.emf.ecore/model/Ecore
   .ecore#//EAttribute">
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="literal"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ModelOperation"
   eSuperTypes="platform:/plugin/org.eclipse.emf.ecore/model/Ecore
   .ecore#//EOperation">
   <eStructuralFeatures xsi:type="ecore:EReference"
      name="implements" eType="#//ModelOperation"/>
   <eStructuralFeatures xsi:type="ecore:EReference"
      name="overrides" eType="#//ModelOperation"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="abstract" eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="literal"
      eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
   <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="enableExtension" eType="ecore:EDataType
      http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
</eClassifiers>
<eClassifiers xsi:type="ecore:EClass" name="ModelReference"
```

```
      eSuperTypes="platform:/plugin/org.eclipse.emf.ecore/model/Ecore
      .ecore#//EReference">
      <eStructuralFeatures xsi:type="ecore:EAttribute"
         name="enableExtension" eType="ecore:EDataType
         http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
   </eClassifiers>
   <eClassifiers xsi:type="ecore:EClass"
      name="InheritsFromReference" eSuperTypes="#//ModelReference"/>
   <eClassifiers xsi:type="ecore:EClass" name="ImplementsReference"
      eSuperTypes="#//ModelReference"/>
   <eClassifiers xsi:type="ecore:EClass" name="LiteralClass"
      eSuperTypes="#//ModelClass"/>
   <eClassifiers xsi:type="ecore:EClass" name="ParameterClass"
      eSuperTypes="#//ModelClass"/>
</ecore:EPackage>
```

Annexe 5

L'API de persistance de EMF

EMF permet de sauvegarder les modèles sous format XMI. Il utilise une API de persistance composée d'un ensemble d'interfaces regroupées dans le paquetage «`org.eclipse.emf.ecore.resource`». Nous en abordons ici quelques unes des plus importantes et que nous avons utilisées pour la manipulation de nos modèles (Figure A5.1).

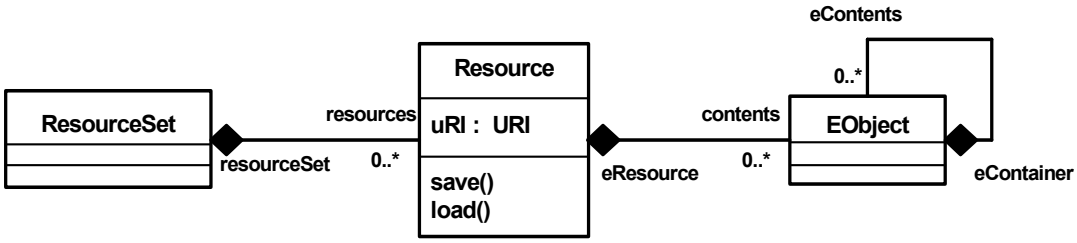


Figure A5.1. Extrait de l'API de persistance de EMF

L'interface `Resource` représente un conteneur (`container`) persistant d'objets. La localisation de ce conteneur est identifiée par son `URI`. Les deux principales méthodes de cette interface sont `save()`, pour enregistrer une ressource, et `load()` pour la charger. Ces méthodes peuvent recevoir des paramètres pour contrôler leur comportement. L'interface `Resource` définit une interface interne `Resource.Factory` qui est utilisée pour créer des ressources. Le type de ressources créées est enregistré en utilisant une interface interne `Resource.Factory.Registry`. Une `Resource.Factory` peut, en effet, être enregistrée pour une catégorie d'`URI`s. Par exemple, nous pouvons enregistrer une `Resource.Factory` pour l'ensemble des ressources ayant une extension donnée (e.g. *.xml). En fait, le paquetage «`org.eclipse.emf.ecore.xmi`» fournit deux implémentations (`XMLResourceImpl` et `XMIResourceImpl`) de

l'interface `Resource` supportant la sérialisation en XML ou XMI. Par défaut, EMF utilise le format XMI pour la sérialisation.

L'interface `ResourceSet` représente une collection de ressources qui ont été créées ou chargées ensemble. Elle fournit les principales méthodes (`createResource()`, `getResource()`) pour gérer une ressource. La méthode `createResource`() est utilisée pour créer une nouvelle ressource vide dans la collection. La méthode `getResource()` crée aussi une ressource mais en faisant son chargement en utilisant son `URI`.

Bibliographie

(Agrawal et al., 2005)

Agrawal A., Karsai G., Kalmar Z., Neema S., Shi F., Vizhanyo A., « The design of a simple language for graph transformations», Journal of Software and System Modeling, 2005.

(Akehurst et Kent, 2002)

Akehurst D.H., Kent S., A Relational Approach to Defining Transformations in a Metamodel. UML 2002, p. 243-258

(Albin-Amiot et Guéhéneuc, 2001)

Albin-Amiot H., Guéhéneuc Y.G., « Meta-modeling Design Patterns: application to pattern detection and code synthesis », Proceedings of ECOOP Workshop on Automating Object Oriented Software Development Methods, Juin 2001.

(Alencar et al., 1997)

Alencar P.S.C., Cowan D.D., Dong J., Lucena C.J.P., « A transformational Process-Based Formal Approach to Object-Oriented Design », Formal Methods Europe FME'97, 1997.

(Alikacem et Sahraoui, 2006)

Alikacem E., Sahraoui H.A., « Détection d'anomalies utilisant un langage de description règle de qualité », In Rousseau R., Urtado C., Vauttier S. (eds.). LMO, 2006, p.185-200

(Allen, 97)

Allen R.J, « A Formal Approach to Software Architecture », PhD Thesis, CMU-CS-97-144, 1997.

(Allen et Garlan, 97)

Allen R.J, Garlan D., « A Formal Basis for Architectural Connection », ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 3, July 1997, p.213–249.

(Appukuttan et al., 2003)

Appukuttan B., Clark T., Reddy S., Tratt L. et Venkatesh R., « A model driven approach to model transformations », Proceedings of the workshop on Model Driven Architecture : Foundations and Application, p. 1-12, University of Twente, 26-27 Juin 2003.

(Arcelli et al., 2005)

Arcelli A., Masiero S., Raibulet C, « Elemental Design Patterns Recognition In Java», In 13th IEEE Int. Work. Software Technology and Engineering Practice, 2005, p. 196-205

(Bacchus et van Beek, 98)

Bacchus F., van Beek P., « On the Conversion between Non-Binary and Binary Constraint Satisfaction Problems », In the 15th Conference on Artificial Intelligence, 1998, p. 311-318, AAAI Press, Menlo Park

(Balogh et Varro, 2006)

Balogh A., Varro D., « Advanced model transformation language constructs in the VIATRA2 framework », Proc. of the ACM symp. on Applied computing, 2006, p. 1280-1287.

(Baresi et Heckel, 2002)

Baresi L., Heckel R., « Tutorial Introduction to Graph Transformation : A Software Engineering Perspective », Proc. of the 1st ICGT, 2002, p. 402–429.

(Baresi et al., 2003)

Baresi L., Heckel R., Thöne S., Varró D., « Modeling and Analysis of Architectural Styles Based on Graph Transformation », the 6th ICSE Workshop on Component Based Software Engineering: Automated Reasoning and Prediction, May 3-4, 2003.

(Barták, 2003)

Barták R., « Foundations of Constraint Programming », a tutorial on ETAPS2003, April 12, 2003, Warshaw, Poland.

(Barták, 99)

Barták R, « Constraint Programming: In Pursuit of the Holy Grail », Proceedings of the Week of Doctoral Students (WDS99), Part IV, MatFyzPress, Prague, June 1999, p. 555-564.

(Bass et al., 2003)

Bass L., Clements, P., Kazman, R., *Software Architecture in Practice*, 2nd Edition, 2003, Addison Wesley.

(Baxter, 92)

Baxter I.D., « Design Maintenance Systems », Communications of the ACM, Vol. 35, No.4, p. 73-89, Avril 1992.

(Bézivin et Blanc, 2002)

Bézivin J., Blanc X., « Promesses et Interrogations de l'approche MDA », la revue Développeur référence, Vol 2.17, 5 septembre 2002.

(Borne et Revault, 99)

Borne I., Revault N., « Comparaison d'outils de mise en oeuvre de Design Patterns », L'Objet, Vol.5, No.2 - numéro spécial sur les Patrons de Conception, p. 243-266, 1999.

(Budinsky et al.,96)

Budinsky F.J., Finnie M.A., Vlissides J.M., Yu P.S., « Automatic Code Generation from Design Patterns », IBM Systems Journal, vol. 35, n° 2, 1996, p. 151-171.

(Budinsky et al., 2003)

Budinsky F.J, Steinberg D., Merks E., Ellersick R., Grose T.J., Eclipse Modeling Framework, Published Aug 11, 2003 by Addison-Wesley Professional. Part of the Eclipse series.

(Buschmann et al., 96)

Buschmann F., Meunier R., Rohnert R., Sommerland P., Stal M., *Pattern-Oriented Software Architecture*, J. Wiley & sons, 1996.

(Chun, 99)

Chun A., « Constraint Programming in Java with JSolver », In the 1st Int. Conf. and Exhibition on the Practical Application of Constraint Technologies and Logic Programming, 1999.

(Ciupke, 99)

Ciupke O., « Automatic Detection of Design Problems in Object-Oriented Reengineering », In TOOLS 30, p. 18-32. IEEE Computer Society Press (1999)

(Clements, 96)

Clements P.C., « A Survey of Architecture Description Languages », Proceedings of the 8th Int'l Workshop Software Specification and Design, Mars 1996.

(Coad, 92)

Coad P., « Object Oriented Patterns », Communications of the ACM, Vol.35, No.9, Septembre 1992.

(Corradini et al., 96)

Corradini A., Montanari U., Rossi F., Ehrig H., Heckel R., Loewe M., « Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach », Université de Pise, Rapport technique TR-96-17, Mars 1996.

(Czarnecki et Helsen, 2003)

Czarnecki K., Helsen S., « Classification of Model Transformation Approaches », Proceedings of the OOPSLA 2003, Workshop on Generative Techniques in the Context of MDA, 2003.

(Debnath et al., 2006)

Debnath N.C., Garis A., Riesco D., Montejano G., « Defining Patterns Using UML Profiles », IEEE International Conference on Computer Systems and Applications , March 8-11, 2006, Sharjah, UAE, p. 1147-1150

(Eden et al., 99)

Eden A.H., Gil J., Hirshfeld Y., Yehudai A., « Towards a mathematical foundation for design patterns », Technical report, dep. of information technology, Uppsala University, 1999.

(Elaasar et al., 2006)

Elaasar M., Briand L., Labiche Y., « A Metamodeling Approach to Pattern Specification and Detection », Proceedings of ACM/IEEE Inter. Conf. on Model Driven Engineering Languages and Systems (MoDELS 2006), October 1-6, Genoa, Italy, 2006

(El-boussaidi et Mili, 2004)

El boussaidi G., Mili H., « Les patrons de conception : représentation et mise en œuvre », rapport technique du LATECE, avril 2004, Montréal, Canada.

(El-boussaidi et Mili, 2006)

El boussaidi G., Mili H., « Les langages de description d'architectures », rapport technique du LATECE, octobre 2006, Montréal, Canada.

(El-boussaidi et Mili, 2008)

El boussaidi G., Mili H., « Detecting Patterns of Poor Design Solutions Using Constraint Propagation », Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), pp. 189-203, Toulouse, September 28-October 3, 2008.

(EMF, 2005)

EMF, Eclipse Modeling Framework Project, EMF developer guide, http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html, 2005.

(Engels et al., 97)

Engels G., Heckel R., Taentzer G., Ehrig H., « A View-Oriented Approach to System Modeling Based on Graph Transformation », the 6th European Software Engineering Conf./5th ACM SIGSOFT Symposium on Foundations of Software Engineering, Switzerland, September 22-25, 1997, Lecture Notes in Computer Science, vol.1301, p.327-343

(Florijn et al., 97)

Florijn G., Meijers M., van-Winsen P., « Tool support for object-oriented patterns », Lecture Notes in Computer Science, vol. 1241, 1997, p. 472-495.

(Fontoura et Lucena, 2001)

Fontoura M., Lucena C., « Extending UML to Improve the Representation of Design Patterns », Journal of OO Programming, vol. 13, n° 11, 2001.

(Fowler, 97)

Fowler M., *Analysis patterns, reusable object models*, Addisson Wesley, 1997.

(France et al., 2004)

France R., Kim D.k., Ghosh S., Song E., « A UML-Based Pattern Specification Technique », IEEE Transactions on Software Engineering, vol. 30, n° 3, 2004, p. 193- 206.

(Freuder, 78)

Freuder, E.C. « Synthesizing Constraint Expressions », In Communications of the ACM, 1978, vol. 21, no. 11, p. 958-966

(Gamma et al., 95)

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

(Gardner et al., 2003)

Gardner T., Griffin C., Koehler J., Hauser R., «Review of OMG MOF 2.0 Q/V/T Submissions & Recommendations towards final Standard», 2003, http://www.omg.org/docs/ad/03-08-02.pdf.

(Garlan et Shaw, 94)

Garlan D., Shaw M., « An Introduction to Software Architecture », Carnegie Mellon University Technical Report CMU-CS-94-166, January 1994.

(Garlan et al., 94)

Garlan D., Allen R., Ockerbloom J., « Exploiting Style in Architectural Design Environments », Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994.

(Garlan et al., 2000)

Garlan D., Monroe R.T., Wile D., « Acme: Architectural Description of Component-Based Systems », Foundations of Component-Based Systems, Leavens G.T. and Sitaraman M. (eds), Cambridge University Press, 2000, pp. 47-68.

(Garlan et al., 2002)

Garlan D., Cheng S.W., Kompanek A., « Reconciling the Needs of Architectural Description with Object-Modeling Notations », Science of Computer Programming Journal, 44 (1), July 2002, p.23-49.

(Gerber et Raymond, 2003)

Gerber A., Raymond K., « MOF to EMOF : There and Back Again », Proceedings of the 2003 OOPSLA workshop on eclipse technology exchange, p. 60-64, Anaheim, California, 2003.

(Gil et Maman, 2005)

Gil, J., Maman, I.: « Micro Patterns in Java Code », In: 20th conference on Object oriented programming systems languages and applications, OOPSLA, 2005, pp. 97-116

(Grunske et al., 2005)

Grunske L., Geiger L., Zündorf A., Van Eetvelde N., Van Gorp P., Varro D., « Using Graph Transformation for Practical Model Driven Software Engineering », in Sami Beydeda, Matthias Book, Volker Gruhn eds, 2005, p. 91-118.

(Guéhéneuc et Jussien, 2001)

Guéhéneuc Y-G., Jussien N, « Using Explanations for Design Patterns Identification», In IJCAI'01 Workshop on Modeling and Solving problems with constraints, 2001, p. 57-64

(Heckel et al., 2002)

Heckel R., Kuster J.M., Taentzer G., « Confluence of Typed Attributed Graph Transformation Systems», In A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, p. 161–176, 2002.

(Hoare, 85)

Hoare C.A.R., *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

(Horvath et al., 2007)

Horvath, A., Varro, G., Varro, D., « Generic search plans for matching advanced graph patterns », In The 6th International GT-VMT Workshop, 2007

(Ivers et al., 2004)

Ivers J., Clements P., Garlan D., Nord R., Schmerl B., Silva J.R.O, «Documenting Component and Connector Views with UML 2.0 », Technical Report, CMU/SEI-2004-TR-008, avril 2004.

(Jamda, 2003)

Jamda project, http://jamda.sourceforge.net/, 2003.

(JMI, 2002)

Java Metadata Interface, JSR-000040 The Java™ Metadata Interface (JMI) Specification (Final Release), juin 2002, http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html

(Johnson, 97)

Johnson R., « Frameworks = (Components + Patterns) », Communications of the ACM, Vol. 40, No.10, Octobre 1997.

(Kalnins et al., 2006)

Kalnins A., Celms E., Sostaks A., « Simple and efficient implementation of pattern matching in MOLA tool », Proceedings of the 7th International Baltic Conference on Databases and Information Systems, Vilnius, Lithuania, July 3-6, 2006, pp. 159-167.

(Kumar, 92)

Kumar V., « Algorithms for Constraint Satisfaction Problems: A Survey », The AI magazine, 1992, vol. 13, n. 1, p. 32-44

(Küster et Abd-El-Razik, 2006)

Küster J. M., Abd-El-Razik M., « Validation of Model Transformations - First Experiences using a White Box Approach », Proceedings of the 3rd Workshop on Model Design and Validation (MoDeV2a), p. 62-77, October 2006

(Lauder et Kent, 98)

Lauder A. et Kent S., « Precise Visual Specification of Design Patterns », Lecture Notes in Computer Science, vol. 1445, p. 114-134, 1998.

(Luckham et al., 95)

Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D., Mann W., « Specification and Analysis of System Architecture Using Rapide », IEEE Transactions on Software Engineering, Vol.21, No.4, April 1995, p.336-355.

(Magee et al., 95)

Magee J., Dulay N., Eisenbach S., Kramer J., « Specifying Distributed Software Architectures », the 5th European Software Engineering Conference (ESEC'95), Spain, 26 September 1995.

(Maplesden et al., 2002)

Maplesden D., Hosking J. et Grundy J., « Design Pattern Modelling and Instantiation using DPML », Proceedings of 14th Intern. Conference on Technology of OO Languages and Systems, 2002.

(Marschall et Braun, 2003)

Marschall F. et Braun P., « Model Transformations for the MDA with BOTL », Proceedings of the workshop on Model Driven Architecture : Foundations and Application, p. 25-36, University of Twente, 26-27 Juin 2003.

(MDA, 2003)

MDA : Model Driven Architecture Guide Version http://www.omg.org/cgi-bin/doc?omg/03-06-01, 12 juin 2003.

(Medvidovic et al., 2002)

Medvidovic N., Rosenblum D.S., Redmiles D.F., Robbins J.E., « Modeling Software Architectures in the Unified Modeling Language », ACM Transactions on Software Engineering and Methodology, Vol. 11, N0 .1, January 2002.

(Medvidovic et Taylor, 2000)

Medvidovic N., Taylor R.N., « A classification and comparison framework for software architecture description languages », IEEE Transactions on Software Engineering, Vol.26, No.1, p.70-93, January 2000.

(Mehlhorn, 84)

Mehlhorn K., « Graph Algorithms and NP-Completeness », In EATCS Monographs On Theoretical Computer Science, vol. 2. Springer verlag, 1984.

(Meijler et al., 97)

Meijler T.D., Demeyer S. et Engel R., « Making Design Patterns Explicit in FACE, A Framework Adaptive Composition Environment », Proceedings of the 6th European Software Engineering Conference (ESEC/FSE'97), p. 94-110, 1997.

(Mens et Tourwé, 2001)

Mens T., Tourwé T., « A Declarative Evolution Framework for Object-Oriented Design Patterns », Proc. IEEE ICSM, Italy, 6-10 Novembre, 2001, p. 570-579.

(Mens et al., 2005)

Mens T., Van Gorp P., Varro D., Karsai G., « Applying a Model Transformation Taxonomy to Graph TRansformation Technology », Proc. of GraMot Workshop, 2005.

(Mili et al., 2002)

Mili H., Mili A., Yacoub S., Addy E., *Reuse-Based Software Engineering: Techniques, Organization, and Control*, John Wiley & Sons, 2002.

(Mili et al., 1995)

Mili H., Mili F. et Mili A., « Reusing software: Issues and research directions», IEEE Transactions on Software Engineering, Vol. 21, No. 6, p. 528-562, 1995.

(MOF, 2003)

MOF : Meta Object Facility, Version 2.0 Core Final Adopted Specification, OMG Document--ptc/03-10-04, Octobre 2003.

(Noble, 98)

Noble J., « Classifying relationships between object-oriented design patterns », Australian Software Engineering Conference (ASWEC), 1998.

(OCL, 2005)

OCL2.0 Specification: Adopted Specification. http://www.omg.org/docs/ptc/05-06-06.pdf, June 2005.

(Pagel et Winter, 96)

Pagel B-U., Winter M., « Towards Pattern-Based Tools », Proceedings of EuropLop, 1996.

(Partsch et Steinbruggen, 83)

Partsch H. et Steinbruggen R., « Program Transformation Systems », Computing Surveys, Vol. 15, No. 3, p. 199-236, Septembre 1983.

(Perez-Martinez, 2003)

Perez-Martinez J.E., « Heavyweight extensions to the UML metamodel to describe the C3 architectural style », ACM SIGSOFT Software Engineering notes, vol.28, No.3, Mai 2003.

(Prechelt et Krämer, 98)

Prechelt, L., Krämer, C.: Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns. J. UCS. 4:12, 866—882, 1998

(QVT, 2005)

QVT: Query View Transformation, Final Adopted Specification, OMG Document--ptc/05-11-01, novembre 2005.

(Rich et Waters, 93)

Rich C., Waters R.D., « Approaches to Automatic Programming », Advances in Computers, Volume 37, pp.1-57, Academic Press, 1993.

(Rossi et al., 90)

Rossi F., Petrie C., Dhar V., On the equivalence of constraint satisfaction problems. In Proceedings of the 9th European Conference on Artificial Intelligence, p- 550-556, Stockholm, Sweden, 1990.

(Rudolf, 98)

Rudolf, M. « Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching », In the 6th International Workshop on Theory and Application of Graph Transformations, LNCS, vol. 1764, pp. 238--251. Springer, London, 1998

(Rudolf et Taentzer, 99)

Rudolf R., Taentzer G., « Introduction to the Language Concepts of AGG », http://tfs.cs.tu-berlin.de/agg, 1999.

(Rumbaugh et al., 96)

Rumbaugh J., Blaha M.R., Lorenson W., Eddy F. et Permerlani W., *Object Oriented Modelling and Design*, Prentice Hall, 1996.

(Sahraoui et al., 2000)

Sahraoui H., Boukadoum M., Lounis H., Ethève F., « Predicting Class Libraries Interface Evolution: an investigation into machine learning approaches », In the 7th APSEC, 2000.

(Sanada et Adams, 2002)

Sanada Y., Adams R., « Representing Design Patterns and Frameworks in UML, Towards a Comprehensive Approach », Journal of Object Technology, vol. 1, n° 2, 2002, p.143-154.

(Shaw et al., 95)

Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M., and Zelesnik G., « Abstractions for Software Architecture and Tools to Support Them », IEEE Trans. Software Eng., vol.21, no.4, p.314-335, April 1995.

(Shaw et Garlan, 94)

Shaw M., Garlan D., « Characteristics of Higher-Level Languages for Software Architecture », Technical Report, CMUCS-94-210, Carnegie Mellon Univ., Dec. 1994

(Smith et Stotts, 2003)

Smith J., Stotts D., « SPQR: Flexible Automated Design Pattern Extraction From Source Code », Technical Report TR03-016, Department of Computer Science, Univ. of North Carolina at Chapel Hill, May 21, 2003.

(Sommerville, 2001)

Sommerville I., *Software Engineering*, Addison Wesley, 2001.

(Sunyé et al., 2000)

Sunyé G., Le Guennec A. et Jézéquel J.M., « Design pattern application in UML », Proceedings of the 14th Object Oriented Programming European Conference, p. 44-62, 2000.

(Taentzer et al., 2005)

Taentzer G., Ehrig K., Guerra E., DeLara J., Lengyel L., Levendovszky T., Prange U., Varro D., Varro-Gyapay S., « Model Transformation by Graph Transformation: A Comparative Study », Model Transformations in Practice Workshop, October 3, 2005, Part of the MoDELS 2005 Conference.

(Taibi et Taibi, 2006)

Taibi T., Taibi F., « Formal Specification of Design Patterns and Their Instances », IEEE International Conference on Computer Systems and Applications , March 8-11, 2006, Sharjah, UAE, p 33-36

(Taylor et al., 96)

Taylor R.N., Medvidovic N., Anderson K.M., Whitehead Jr.E.J., Robbins J.E., Nies K.A., Oreizy P., Dubrow D.L., « A component- and message-based architectural style for GUI software », IEEE Transactions on Software Engineering, Vol.22, NO.6, june 1996.

(Tsang, 93)

Tsang E.P.K., *Foundations of Constraint Satisfaction*, Academic Press, London and San Diego, 1993 , ISBN 0-12-701610-4

(UML, 2005)

UML2.0 Superstructure Specification: Adopted Specification, http://www.omg.org/docs/formal/05-07-04.pdf , August 2005.

(UML, 2006)

UML2.0 Infrastructure Specification: Adopted Specification. http://www.omg.org/docs/formal/05-07-05.pdf, March 2006.

(Varro et al., 2002)

Varro D., Varro G., Pataricza A., « Designing the automatic transformation of visual languages», Special issue on applications of graph transformations (GRATRA 2000), Vol. 44 , N.2, p. 205 – 227, August 2002.

(Varro et al., 2006)

Varro G., Friedl K., Varro D., « Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans », In Karsai G., Taentzer G. (eds.), GraMot 2005. ENTCS, vol. 152, p. 191–205 (2006)

(Varro, 2008)

Varro, G., « Implementing an EJB3-Specific Graph Transformation Plugin by Using Database Independent Queries », Electronic Notes in Theoretical Computer Science, vol. 211, p. 121–132, 2008

(Vestal, 93)

Vestal S., « A Cursory Overview and Comparison of Four Architecture Description Languages », Honeywell technical Report, February 1993.

(Wuyts, 98)

Wuyts, R., « Declarative Reasoning about the Structure of Object-Oriented Systems », In Technology of Object-Oriented Languages and Systems, TOOLS'98, pp. 112--124. IEEE Computer Society Press, 1998

(XMI, 2003)

XMI Specification v2.0, OMG Document--formal/03-05-02, Mai 2003, http://www.omg.org/cgi-bin/doc?formal/2003-05-02.

(Zarras et al., 2001)

Zarras A., Issarny V., Kloukinas C., Nguyen V. K., « Towards a Base UML Profile for Architecture Description », 1st ICSE Workshop on Describing Software Architecture with UML, Canada, p. 22-26, 2001.

(Zimmer, 94)

Zimmer W., « Relationships between design patterns », Pattern Languages of Program Design. Addison-Wesley, 1994.

(Zündorf, 94)

Zündorf A., « Graph Pattern Matching in PROGRES », Lecture Notes in Computer Science, vol. 1073, 1994, p. 454-468.