Mise en oeuvre des patrons de conception par représentation explicite du problème

Hafedh Mili, Ghizlane El Boussaidi, et Aziz Salah

Laboratoire de Recherche en Technologies du Commerce Électronique (LATECE)
Faculté des Sciences
Université du Québec à Montréal
B.P 8888, succursale Centre-Ville,
Montréal (Québec) H3C 3P8, Canada
{hafedh.mili, el_boussaidi.ghizlane, salah.aziz}@uqam.ca

RÉSUMÉ. *Les patrons de conception constituent une codification intuitive de solutions éprouvées à des problèmes de conception récurrents. Plusieurs ont cherché à développer des méthodes pour la représentation et la mise en œuvre des patrons. Or, aucune des approches que nous avons étudiées ne représente* ***explicitement*** *le problème de conception que le patron cherche à résoudre. La représentation du problème a plusieurs avantages, dont : 1) une meilleure caractérisation de l'applicabilité des patrons—plus précise que la description textuelle préconisée par [Gamma et al., 1995], 2) une représentation plus naturelle des transformations inhérentes aux patrons, et 3) la possibilité d'identifier automatiquement les opportunités d'application de patrons dans un modèle d'analyse. Dans cet article, nous décrivons les principes de notre représentation, et son implantation dans le cadre EMF d'Éclipse™.*

ABSTRACT. *Design patterns embody proven solutions to recurring design problems. Ever since the gang of four popularized the concept, researchers have been trying to develop methods for representing design patterns, and applying them to modeling problems. To the best of our knowledge, none of the approaches proposed so far* ***represents the design problem*** *that the pattern is meant to solve,* ***explicitly****. An explicit representation of the problem has several advantages, 1) a better characterization of the problem space addressed by the pattern—better than the textual description embodied in pattern templates, 2) a more natural representation of the transformations embodied in the application of the pattern, and 3) it is one step towards the automatic detection and application of patterns. In this paper, we describe the principles underlying our approach, and the current implementation in the Eclipse Modeling Framework™.*

MOTS-CLÉS : *Patrons de conception, Métamodèle, Développement par transformation.*

KEYWORDS: *Design patterns, Metamodels, Transformational development.*

1. Introduction

Le développement de logiciels a souvent été modélisé comme une suite de transformations appliquées aux exigences usager pour obtenir un logiciel fonctionnel et qui réponde aux exigences de qualité. Les chercheurs ont longtemps tenté de caractériser ces transformations de façon suffisamment précise pour pouvoir les encoder dans des procédures ou processeurs systématiques que l'on peut appliquer à de nouvelles exigences (Mili et al., 2001). Les recherches menées durant les années '80 nous ont appris deux choses : 1) un système d'aide au développement de logiciels qui se veut générique, doit capturer explicitement tant les connaissances de développement, que des connaissances du domaine d'application, et 2) les connaissances de développement demeurent difficiles à capturer. En effet, bien que nous ayons développé des métriques fiables de qualité *a-posteriori* (e. g. couplage et cohésion) sur le produit final, nous ne savions toujours pas comment garantir ces qualités par construction. Les patrons de conception ~~— jugés par certains comme une abdication de la communauté de chercheurs [Trial gang of four] —~~ ont pallié à ce problème en capturant, pour des problèmes récurrents en conception, les solutions qui se sont avérées efficaces à l'usage. Une importante contribution de Gamma et al. était de cataloguer un certain nombre de patrons et de les documenter de façon à faciliter leur compréhension et leur usage *par des concepteurs* (Gamma et al., 1995).

Plusieurs travaux se sont intéressés depuis à offrir aux concepteurs de l'aide pour l'application de patrons de conception, dont (Budinsky et al., 1996), (Eden et al., 1997), (Florijin et al., 1997), ~~(Borne et Revault, 1999),~~ (Sunyé et al., 2000), (Albin-Amyot et Guéhénec, 2001), et bien d'autres. Comme tout artéfact réutilisable, l'utilisation d'un patron de conception passe par trois étapes :

1) reconnaître le patron comme étant une solution potentielle au problème de conception en main,

2) comprendre le patron, sa structure, et les principes qui le sous-tendent, et

3) appliquer le patron au problème en main

Chacune de ~~ses~~ ces étapes nécessite une représentation particulière du patron. Pour la première, nous devons disposer d'une représentation du problème de conception que le patron est supposé résoudre, et comparer cette représentation au problème en main. L'étape de compréhension nécessite une représentation digestible par un être humain, i.e. essentiellement des (modèles) graphiques et du texte. La troisième étape nécessite la représentation de la mise en œuvre du patron par une transformation formelle ayant comme paramètres les données du problème en main.

Les approches que nous avons étudiées mettent l'emphase sur la compréhension ou l'application du patron (e. g. (Alencar et al. 1997), (Sunyé et al., 2000), (Sanada & Adams, 2002)), des fois les deux (Fontoura & Lucena,

2001),(Maplesden et al., 2002), mais jamais sur les trois. Or, nous croyons que la représentation du problème est cruciale pour les trois tâches :

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. Les approches que nous avons étudiées se contentent de représenter l' « après »,

3) l'application du patron à un fragment de modèle ~~d'analyse~~ 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.

Dans cet article, nous décrivons notre approche à la représentation et la mise en œuvre des patrons qui prend en compte la représentation du problème pour les trois tâches.

La section 2 introduit notre approche pour la modélisation du problème en utilisant l'exemple du patron « Pont »~~et cela par le biais d'un exemple~~. Dans ~~La~~ la section 3 nous ~~décrit~~décrivons les modèles de solution et de transformation reliés à un patron. ~~et n~~Nous donnons aussi une vue de tout le processus d'application d'un patron selon notre approche. La section 4 décrit l'implantation de notre représentation ~~faite e~~en utilisant le cadre d'application EMF (Eclipse Modeling Framework). ~~que nous présentons brièvement, le cadre d'application EMF (Eclipse Modeling Framework). La section 4 décrit l'implantation en EMF.~~ Dans la section 5, nous comparons ~~Nous comparons~~ notre approche aux travaux reliés, et nous examinons la détermination automatique de l'applicabilité d'un patron à un modèle donné. ~~à la section 5.~~ Nous concluons dans la section ~~5~~6.

2. Modélisation du problème

2.1. Exemple : le patron « Pont »

Prenons l'exemple du patron pont (*bridge*). La ~~f~~Figure 1 illustre une situation ou le patron « pont » serait approprié. Ici nous reprenons les arguments des auteurs du patron, tels quels (Gamma et al., 1995). Supposons que nous voulions développer un programme qui manipule des fenêtres graphiques, et qui soit portable de l'environnement Microsoft (Windows) à l'environnement UNIX (XW~~w~~indows). La façon traditionnelle de procéder consiste à chapeauter les diverses implantations de fenêtre graphique par une même classe *abstraite* que nous appellerons **Fenetre** ; cette classe définira, sous la forme de méthodes abstraites (ou *virtuelles*, en C++), les comportements que ses implantations devront

supporter. Cette solution est décrite par le coté gauche de la Figure 1. Supposons maintenant que l'on ait besoin de définir un autre *type* de fenêtres, par exemple des fenêtres carrées, qui peuvent définir des comportements supplémentaires ou optimiser les comportements existants. Il faudra créer une nouvelle classe abstraite sous **Fenêtre**, que nous appellerons **FenetreCarree**, et définir une implantation de **FenetreCarree** par plate-forme. Cette situation est illustrée par le coté droit de la Figure 1. Similairement, si nous devions supporter une troisième plate-forme, il nous faudra ajouter une implantation de chaque *type* de fenêtre.

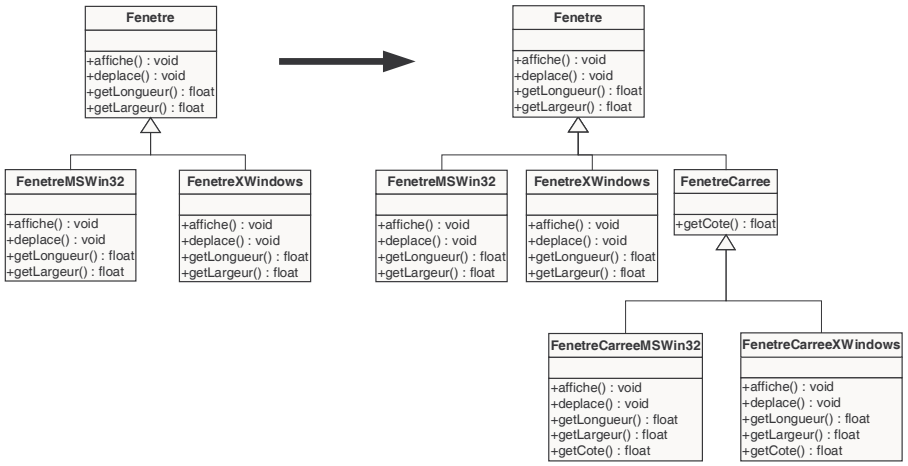


Figure 1. Un exemple de problème résolu par le patron « Pont »

La solution qu'apporte ce patron est de découpler l'implantation de l'abstraction, et de créer deux hiérarchies parallèles qui peuvent évoluer indépendamment l'une de l'autre ; en particulier, nous n'aurons à définir de nouvelles classes d'implantation que lorsqu'elles fournissent une implantation inédite d'un comportement défini dans l'une des abstractions (voir Figure 2).

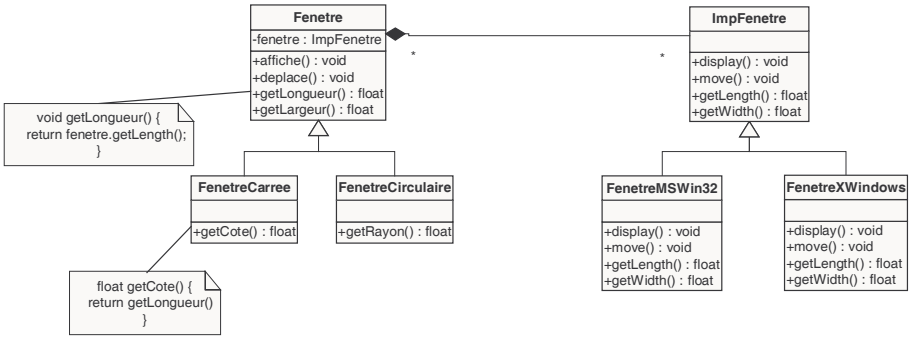
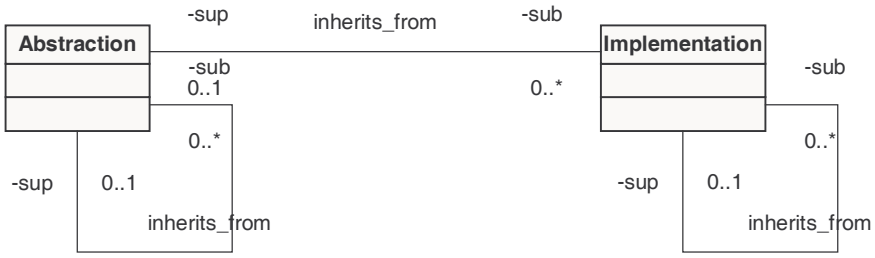


Figure 2. La solution apportée par le patron « Pont »

Le défi maintenant est de traduire cette information en, 1) une description formelle du problème qui soit à la fois intuitive et manipulable par la machine, 2) une description formelle de la solution qui soit aussi intuitive et manipulable, et 3) une description formelle de la transformation de l'un vers l'autre. Les diagrammes des Figure 1 et 2Ces diagrammesdescriptions, dont le but est d'expliquer le problème de conception, décrivent une *instance* du problème. Nous devons en extraire la *classe* de problèmes. Dans les sections qui suivent, nous présentons les principes de notre représentation.

2.2. Un Méta-modèle du problème

Les instances du problème de conception sont des *modèles* de niveau analyse ou conception. Pour décrire la *classe* de problèmes, nous allons définir un méta-modèle du problème, i.e. un modèle dont les instances seront des modèles tels que l'exemple de la Figure 1. Nous représenterons une première esquisse dans la Figure



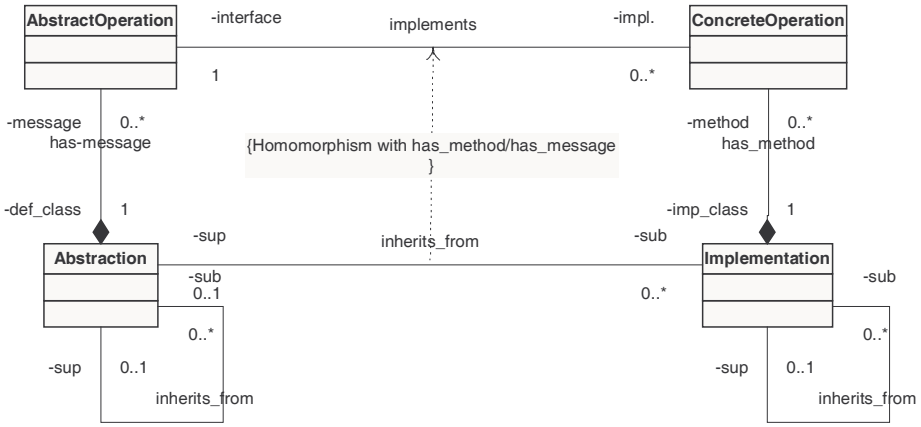
3.

Figure 3. Un premier métamodèle du problème.

Les classes « Abstraction » et « Implementation » sont des méta-classes dans le sens que leur instances sont des classes telles que **Fenetre** ou **FenetreMSWin32**, respectivement. Les associations « inherits_from » représentent *les relations d'héritage qui doivent relier les instances des classes correspondantes*. Par exemple, l' « abstraction » **FenetreCarree** (voire. f. Figure. 12) « inherits_from » l' « abstraction » **Fenetre**. De même, l' « implementation » **FenetreCarreeMSWin32** « inherits_from » **FenetreCarree**, qui est une « abstraction » (voir Figure 1e. f. Fig. 2). Dans l'exemple de la Figure 12, nous n'avions pas de situation de plusieurs niveaux– d'implantation, mais ceci est envisageable. Notez que pour le moment, nous ne nous préoccupons pas de ce que

cela veut dire d'être une « abstraction » ou une « implementation ». Nous pouvons les interpréter, pour le moment, comme des simples *marquages*. Nous reviendrons plus tard sur la sémantique possible de ces méta-classes.

Ce méta-modèle ~~représente t il fidèlement tous les cas d'utilisation du patron « Pont » ? Indépendamment de la réponse qui est non il nous faut plus d'information pour supporter l'application du patron à une instance de modèle : la description (et la transformation) des opérations~~n'est pas encore complet : il nous manque la représentation des opérations qui vont être affectées par l'application du patron. ~~Autant les « Abstraction »s que les « Implementation »s vont supporter des opérations.~~ Les opérations des « Abstractions » vont être abstraites, et celles des



« Implementation »s vont être concrètes. De plus, chaque « Implementation » doit implanter toutes les opérations des « Abstraction »s qu'elle implante. Pour fins de présentation, ceci est représenté dans le modèle (Figure 4) comme une contrainte entre les associations « implements » et « inherits_from ».

Figure 4. Méta-modèle du problème résolu par le patron « Pont ». Prise 2.

En principe, ~~I~~il faudrait ~~, en principe,~~ représenter les paramètres et les types de retour des opérations pour pouvoir les transformer pour obtenir la solution. Nous nous passerons de ces détails pour le moment. Il faudra aussi prévoir le cas où les abstractions ne sont pas des classes abstraites pures, mais implantent certaines opérations génériques en termes d'opérations abstraites—comme c'est ~~(était)~~ le cas de la librairie Collection de Smalltalk, par exemple. Les méthodes concrètes qui

font partie des abstractions seront traitées différemment des méthodes abstraites[1]. Pour les fins de cet article, nous nous contenterons de cette représentation.

2.3. Chaînon manquant : la dérivée par rapport au temps !

L'objectif premier des patrons de conception est de protéger le programme qui les utilise contre les effets des changements possibles ou probables dans leur fonctionnalité, leur environnement, ou leur implantation. Prenons l'exemple suivant :

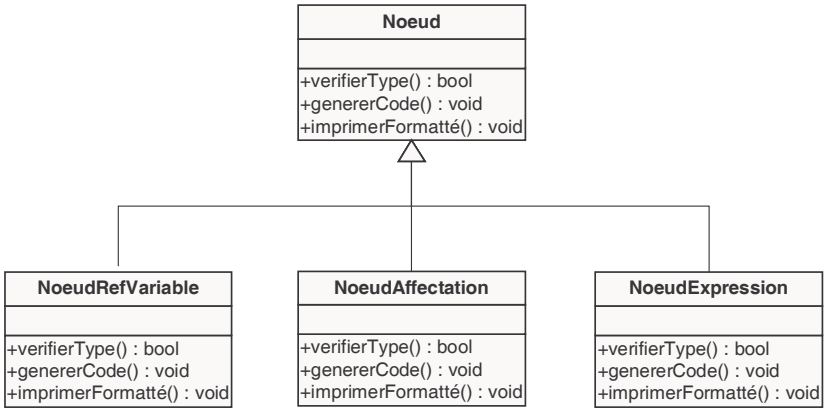


Figure 5. *Problème ambigu.*

Le lecteur averti reconnaîtra l'exemple que l'on utilise souvent pour le patron « visiteur ». Le patron « visiteur » est applicable lorsque la hiérarchie de classes est stable, alors que les traitements (l'ensemble des méthodes) ne le sont pas. En effet, pour chaque nouveau traitement, la conception classique exigerait la modification de toute la hiérarchie. La solution que propose le patron visiteur regroupe, dans une même classe, toutes les implantations d'un comportement donné. Notons que dans le cas ou l'ensemble de traitement est stable, mais que la liste des types est variable, alors l'approche classique (par héritage) est la plus appropriée.

Ainsi, nous avons décidé de représenter dans le modèle du problème les aspects qui changent. En étudiant les différents types d'évolution, nous avons pu les ramener tous à des changements de cardinalités d'associations au niveau du méta-modèle : le nombre de sous-classes d'une classe donnée (« pont », « *Abstract Factory* »), le nombre d'opérations d'une classe donnée (« visiteur »,

[1] Les méthodes abstraites des abstractions dans le modèle source deviennent des méthodes concrètes qui délèguent à l'objet implantation ~~(??????« fenetre » dans la Figure 2)~~. Les méthodes concrètes sont plus complexes à traiter et peuvent mener à des instances du « self-problem » (ElBoussaidi & Mili, 2004).

« décorateur », etc.), le nombre d'implantations d'une opération donnée (« stratégie », « *Template Method* », etc.). Nous représentons donc les points de variation en ajoutant le symbole « ++ » aux cardinalités des associations concernées. La Figure 6 montre le nouveau modèle du problème résolu par « pont». Ici, nous disons que le nombre d'abstractions ~~va~~ est susceptible d'évoluer, ~~et~~ ainsi que le nombre d'implantations par abstraction (e.g. le nombre de plate-formes) ~~va aussi évoluer~~.

2.4. Le langage de description des modèles de problèmes

L'exemple précédent nous a donné une idée sur les constructions supportées par notre langage. Notons que dans notre langage, les notions de *Abstraction* ou d'*Implementation* telles que vues précédemment ne font pas partie des *primitives* du langage : le concepteur (qui décrit les patrons de conception) peut définir les concepts qu'il veut et leur attribuer la sémantique qu'il veut. Par contre, ces concepts doivent hériter, quelque part, du noyau d'UML qui est compatible avec MOF. Donc, Abstraction et Implementation, même s'ils sont spécifiques au patron « Pont »[2], sont des classifieurs. De même, la référence à *AbstractOperation* et *ConcreteOperation* n'est pas universelle, mais il existe un type « Operation » duquel ces catégories héritent.

Pour finir, notons que outre la variabilité dans la cardinalité des associations, nous avons introduit la notion de *famille* qui est un ensemble d'entités du même type (de niveau méta-métamodèle) qui partagent certaines propriétés, et qu'il est avantageux de traiter comme une seule entité. On parle de *familles de classes* (e.g. l'ensemble de sous-classes d'une classe donnée, ou ce que Odell appelle *powertype* (Odell, 1995)) ou *familles de méthodes*. Mises à part ces deux modifications, le méta-langage pour décrire les modèles de problèmes reste sensiblement UML. Nous reviendrons sur certaines caractéristiques du langage quand nous parlerons d'implantation en EMF™.

[2] En fait ces deux notions sont utilisées dans plusieurs patrons.

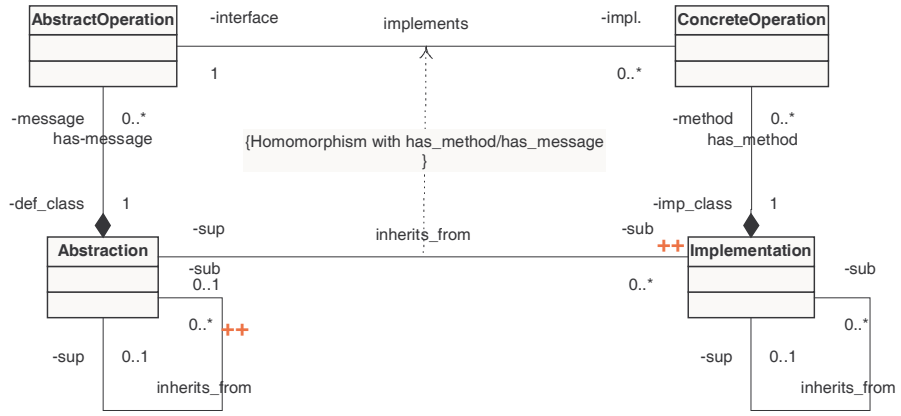


Figure 6. *Méta-modèle du problème résolu par «-Pont-», avec les points d'évolution.*

3. Représentation de la solution et de la transformation

3.1. Représentation de la solution

Nous avons utilisé les mêmes principes pour la représentation de la solution. ~~En fait, s~~Sur ce plan, notre approche est similaire aux approches de représentation de patrons de conception par méta-modèle, dont (Pagel & Winter, 1996), (Albin-Amyot & Guéhéneuc, 2001), (Sanada & Adams, 2002), et (France et al., 2004). La Figure 7 montre le (méta)~~méta~~ modèle de la solution apportée par le patron « Pont ».

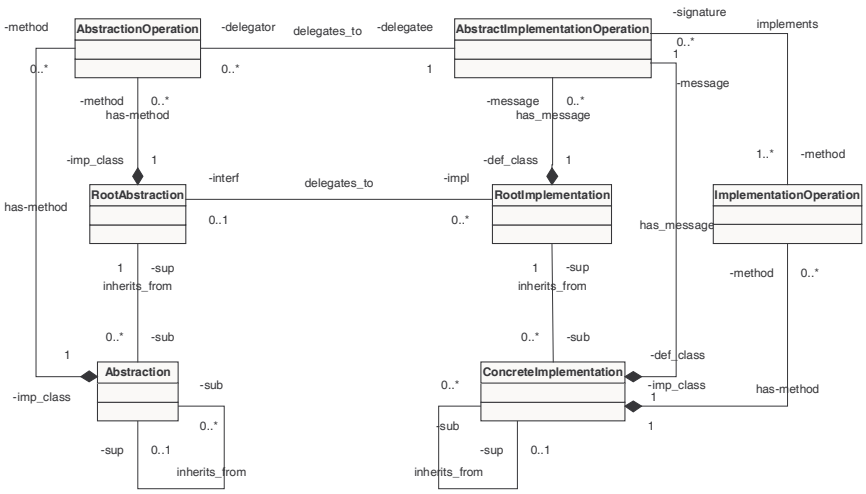


Figure 7. *Méta-modèle de la solution apportée par le patron « Pont ».*

Nous lisons ce modèle de la façon suivante : nous avons une hiérarchie de classes (**RootAbstraction** et **Abstraction**), qui délègue ses traitements à une autre hiérarchie de classes (**RootImplementation** et **ConcreteImplementation**) ~~à une hiérarchies~~. Notez que nous devons faire une distinction entre les classes racines et les classes courantes, tant pour les abstractions que pour les implémentations[3], puisque la racine des implémentations est une classe abstraite, alors que ses sous-classes sont des classes concrètes qui implantent ses méthodes abstraites. Par contre, toutes les classes de la hiérarchie des abstractions sont des classes concrètes 😊: leurs opérations délèguent l'appel aux opérations correspondantes de l'objet implantation.

Rappelons que, comme pour le cas des modèles de problèmes, la signification des méta~~méta~~ classes « **Abstraction** » ou « **RootAbstraction** » est spécifique au patron « Pont », et aura le sens que nous voudrons lui accorder. Notons aussi que nous ne sommes pas obligés de reprendre les termes du problème pour décrire la solution: nous prévoyons établir une correspondance explicite—décrite dans la prochaine section—qui établira le lien entre éléments du problème et éléments de la solution.

Finalement, notons que la modélisation de solutions nécessite~~nt~~ de nouvelles constructions dans notre langage, dont la représentation de constantes. Avec le patron « Pont », toutes les opérations qui apparaissent dans la solution proviennent directement des données du problème, et c'est souvent le cas. Par contre, il y a des patrons qui exigent l'ajout de nouvelles classes ou méthodes. Par exemple, le patron « Observateur/Observé » exige que les objets observés implantent des méthodes spécifiques, propres au patron, pour notifier les observateurs (e.g. « notify(…) », avec ou sans paramètres), et il nous faut des façons d'exprimer cela. Notre langage de représentation supporte la représentation de constantes.

3.2. Représentation de la correspondance entre problème et solution

L'application d'un patron de conception consiste à transformer une instance de la classe de problèmes en une instance de la classe de solutions. On peut donc représenter la transformation par une correspondance entre les éléments du méta-modèle de problème et ceux du méta-modèle de solution. Ainsi, partant d'un modèle à l'entrée auquel on veut appliquer un patron, on commence d'abord par identifier les éléments du modèle qui jouent les rôles décrits par le méta-modèle du problème. Pour le cas du patron « Pont », on cherche, par exemple, à identifier les classes qui jouent les rôles de **Abstraction** et **Implementation** dans le modèle à

[3] Cette distinction n'était pas strictement nécessaire pour le cas du problème, à condition de ~~pourvu que l'on spécifie~~spécifier, quelque part, que l'ensemble~~s~~ des abstractions forme un arbre (El Boussaidi & Mili, 2004).

l'entrée. Ceci nous amène à un *modèle marqué*. Le modèle ainsi marqué est transformé en une instance du méta-modèle de solution grâce à la représentation de la transformation. La Figure 8 montre notre processus pour l'application de patrons.

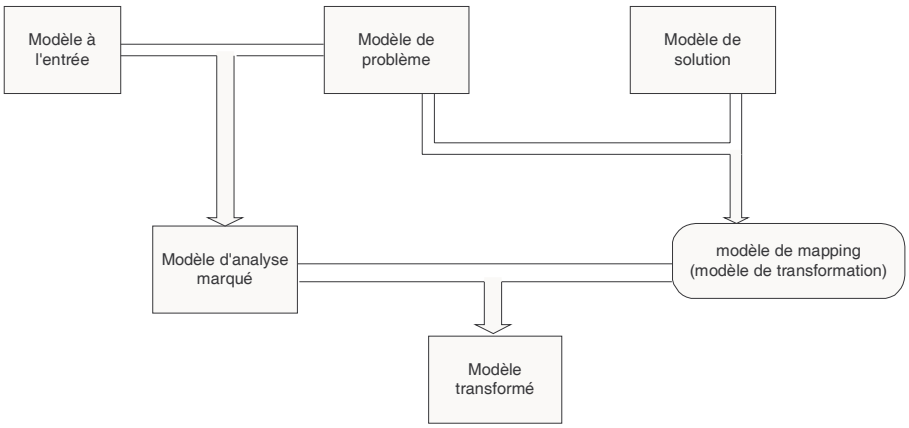


Figure 8. *Processus d'application de patron de conception.*

La Figure 9 montre le modèle de représentation des correspondances entre méta-modèle de problème et méta-modèle de solution. Toutes les correspondances spécifient (au plus) un élément source, (au plus) un élément destination, et une description de la transformation (attribut « transformation » de la classe **ElementMapping**). Une correspondance qui n'a pas d'élément source indique un élément fourni par la solution. Une correspondance qui n'indique pas un élément destination indique un élément qui disparaît dans la transformation. La correspondance entre méta-modèle du problème et méta-modèle de la solution est représentée par une instance de **ModelMapping**, qui une agrégation de **AssociationMapping**'s et de **–ClassMapping**'s. Et ainsi de suite. La classe **Element** sur fond gris représente les éléments du modèle à l'entrée (une classe, une association, un attribut, etc.). Elle correspond au type ***ModelElement*** du MOF.

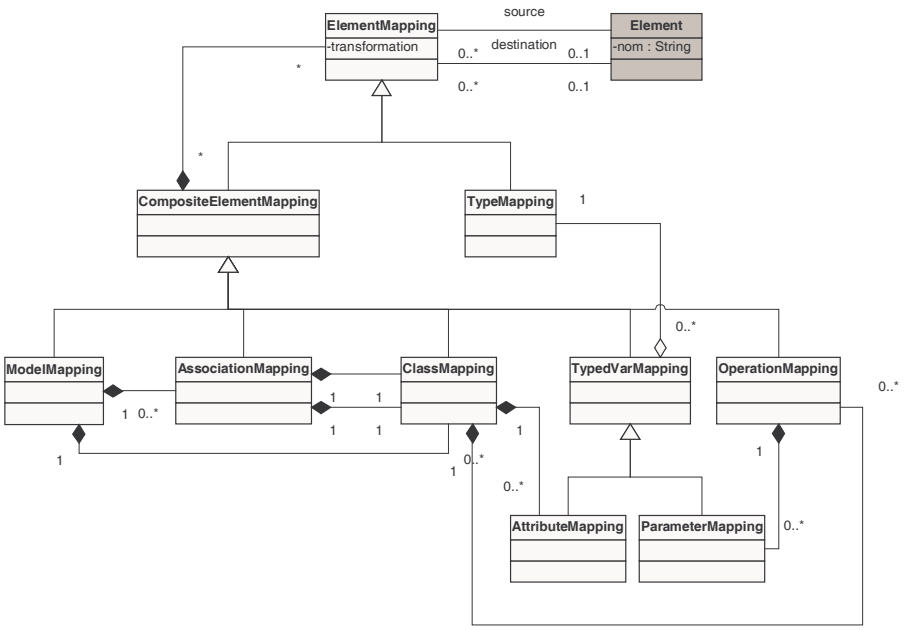


Figure 9. *Modèle des correspondances entre modèles du problème et modèles de solution traitées par un patron de conception.*

4. Implémentation

Nous avons implémenté notre représentation des patrons et la procédure d'application de patrons au sein de l'environnement Eclipse™, et plus

particulièrement, en utilisant le cadre d'application Eclipse Modeling Framework (EMF). Trois facteurs ont influencé notre choix :

1) le désir de supporter nos modèles et procédures dans un environnement ouvert du domaine public,

2) supporter facilement la visualisation graphique des modèles de problèmes et de solutions, et

3) supporter la sérialisation des modèles de patrons sous la forme du standard d'échange XMI.

Nous commençons d'abord par un survol du cadre EMF. Nous présenterons ensuite des éléments de notre représentation. Nous finirons la section par une description de l'algorithme de transformation de modèles.

4.1. Le cadre EMF

EMF est un cadre d'application développé en Java destiné à l'environnement de développement Eclipse. De nos jours, un développeur va typiquement manipuler des fichiers Java, des modèles UML (diagrammes de classes) exportés de divers outils, et des fichiers XML (format XSD ou autre). Le cadre EMF a été développé dans le but d'offrir un cadre unifié pour la manipulation de ces trois types de fichiers. En particulier, l'API de EMF permet de générer du code Java à partir de modèles UML et vice-versa. EMF implante donc le méta-modèle commun au code Java (aspects statiques), aux modèles objets UML, et aux dialectes XML. On y retrouve donc une implantation quasi-partielle du MOF. L'implantation est *partielle* parce qu'on n'y retrouve pas les aspects dynamiques de UML tels les diagrammes de séquences ou les diagrammes d'activités. L'implantation est *quasi*-partielle parce que même les concepts de MOF qui sont traités par EMF ne le sont pas de façon fidèle. Nous reviendrons la dessus plus tard. La Figure 10 montre l'essentiel des classes du paquetage Ecore, qui est le sous-ensemble de EMF qui implante le sous-ensemble de MOF. Les classes ombrées sont abstraites (**EModelElement**, **ENamedElement**, **ETypedElement**, **EClassifier**, et **EstructuralFeature**).

4.2. Méta-modèle commun aux problèmes et solutions

La Figure 11 montre le méta-modèle commun aux problèmes et solutions des patrons de conception. Notre idée initiale était d'utiliser ECore pour décrire le méta-modèle des patrons, de la même façon que MOF nous permet de décrire des méta-modèles de divers langages (dont UML, CWM ou d'autres) ; ce métamodèle de patrons nous permettrait alors de définir des modèles de patron. Or, EMF ne supporte pas les trois niveaux de modélisation du MOF. Donc, plutôt que de définir le métamodèle des patrons de conception comme une instance de ECore, nous

l'avons défini comme une *extension* des classes de ECore. La Figure 11 montre le méta-modèle que nous commenterons brièvement.

Tout d'abord, 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 métaclasses. Cette sous-classe s'appelle **ModelPackage**. Les classes de ECore sont en couleur (gris), alors que les nouvelles classes sont en blanc (Figure 11).

#

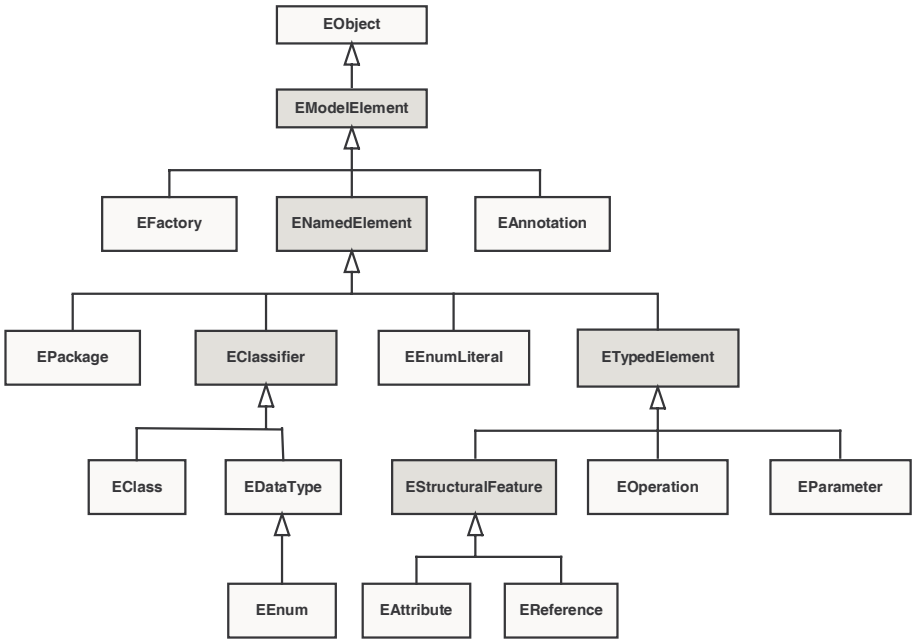
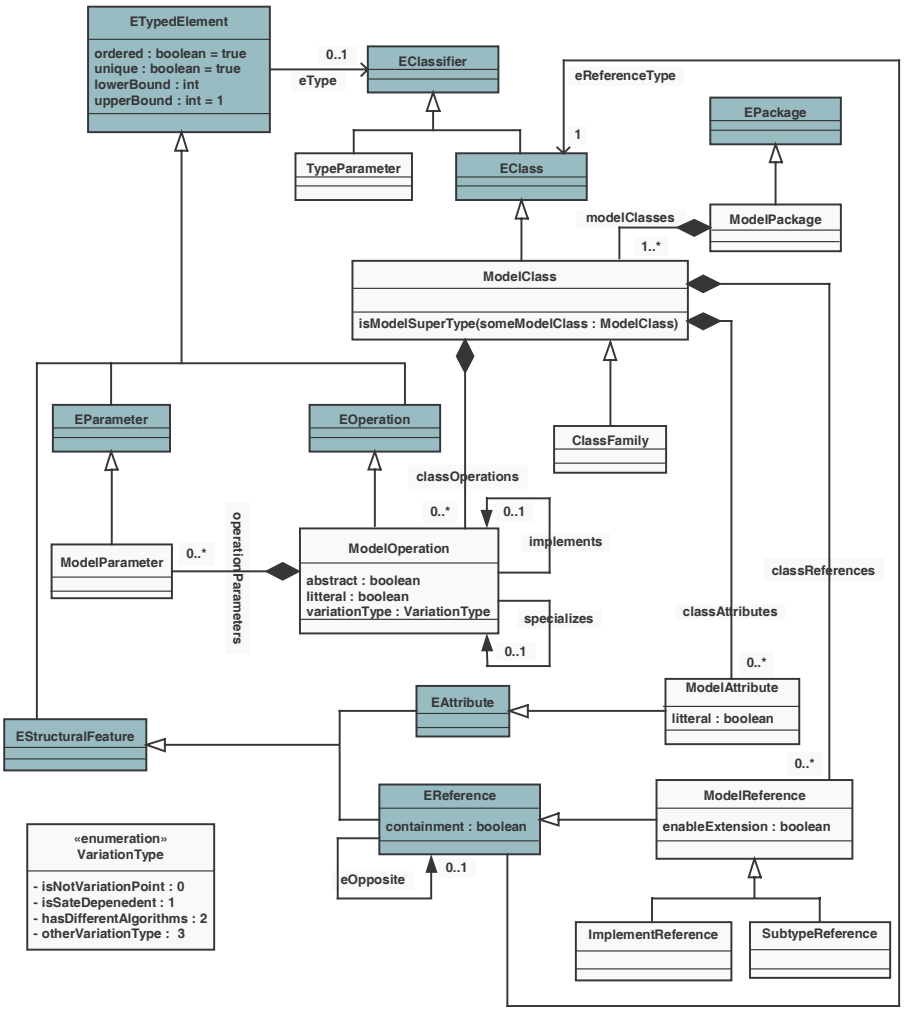


Figure 10. Principales classes du paquetage ECore du cadre EMF.

La Figure 11 montre une classe **ModelClass** comme sous-classe à la classe **EClass** de ECore. **ModelClass** représente toutes les classes apparaissant dans les modèles de problèmes et dans les modèles de solution. Avec l'exemple du patron « Pont », les classes **Abstraction** et **Implementation**, du modèle de problème et **RootAbstraction**, **Abstraction**, **RootImplementation**, et **Concrete-Implementation** du modèle de solution, sont toutes des instances de **ModelClass**. Toutes les opérations aussi sont des instances de **ModelOperation**. Plutôt que d'avoir une méta-classe pour méthodes virtuelles, une pour méthodes concrètes, et une pour méthodes littérales (constantes), la même classe (**ModelOperation**)

accommode les variations grâce aux variables d'instances « abstract » et « literal », toutes les deux booléennes. Notez aussi que nous avons étendu **EReference** par la classe **ModelReference** pour avoir notre propre modèle d'associations pour : 1)



pouvoir représenter la relation d'héritage au niveau métamodèle (**SubtypeReference**), et 2) pour pouvoir représenter l' « ~~évolvabilité~~ évolutivité » de la cardinalité (le symbole ++ utilisé dans la Figure 6).

Figure 11. Métamodèle de modèles de problèmes et de solutions de patrons de conception avec le cadre EMF.

Rappelons que le cadre EMF nous permet de sérialiser une instance du métamodèle de la Figure 11 (i.e. un modèle de problème ou un modèle de solution d'un patron) sous le format XMI. Une telle instance peut être créée soit par invocation de l'API soit via un éditeur graphique basé sur le paquetage EMF.EDIT. Pour le moment, l'aspect graphique nous intéresse peu, et les patrons que nous avons modélisés ont été créés par appels successifs à divers constructeurs et, getters et setters de l'API réflexive de EMF.

4.3. Implémentation de la transformation

Pour diverses raisons, dont l'accès aux fonctionnalités de sérialisation de EMF, nous avons implanté le modèle de correspondances montré dans la Figure 9 *aussi* comme un méta-modèle, avec sa propre extension de la classe **EPackage** (ElBoussaidi & Mili, 2004). Nous ne nous attarderons pas dessus.

Quant à la procédure de transformation elle même, elle prend comme arguments, i) le modèle à l'entrée dûment marqué par les éléments du modèle de problème, ii) la correspondance entre modèle de problème et modèle de solution, et iii) le modèle de solution. Elle produit en sortie le modèle à l'entrée tel que transformé par le patron. Le transformateur procède de façon descendante ou la transformation d'un agrégat source génère d'abord une coquille vide de l'agrégat destination, lui applique les transformations locales, puise 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. Notez qu'une même correspondance (transformation) peut être appliquée à plusieurs éléments du modèle à l'entrée. Par exemple, la transformation **Implementation** (du problème) → **ConcreteImplementation** (de la solution) peut être appliquée plusieurs fois dans le cas du patron « Pont ». L'algorithme est représenté brièvement dans la Figure 12.

nouveauModele ← **new** EPackage

POUR CHAQUE CLASSE classe CLS **DU** modeleSource

SI classe CLS **EST MARQUEE ALORS**

POUR **LES CLASSES DE CORRESPONDANCE**CHAQUE instance de C**lassMapping** **DONT** sdont sourceElement = **EST EGAL A LA MARQUE**(DE classeCLS)

transformedClass ← **new** ClassCréer une nouvelle classe transformedClass;

transformedClass ←+ classMapping.transformation(CLSMettre à jour les propriétés de transformedClass en utilisant les règles de transformation enregistrées dans classMapping et les propriétés

de la classe solutionClass du modèleSolution correspondant à destinationElement de classMapping;)

POUR TOUTES EOperation OP de LES OPERATIONS DE LA CLASSE classeCLS

SI l'opération est marquée **L'OPERATION EST MARQUEE ALORS**

transformedOper ← **new** EOperationCréer une nouvelle opération transformedOper;

transformedOper ←+

operationMapping.transformation(OP)

transformedClass ←+ Mettre à jour les propriétés de transformedOper en utilisant les règles de transformation en registrées dans l'élément operationMapping contenu dans classMapping ;

// cette opération peut avoir des paramètres qu'il // faut transformer aussi,…

Ajouter transformedOper

// idem pour paramètrescette nouvelle opération à transformedClass ;

SINON

transformedClass ←+ OP

Ajouter l'opération sans transformation à transformedClass ;

FIN SI

FIN POUR // fin pour toutes les opérations de la classe

// on boucle aussi sur tous les attributs et on les transforme s'ilsidem pour les attributs

// sont marqués ou on les récupère tels qu'ils sont ;

nouveauModele ←+ Ajouter transformedClass au nouveau modèle;

FIN POUR // fin pour toutes les classes de correspondance

SINON // la classe n'est pas marquée

nouveauModele ←+ CLS;

~~Ajouter la classe au nouveau modèle telle quelle ;~~

FIN SI

FIN POUR // fin pour toutes les classes du modèle

//Une fois toutes les classes du nouveau modèle créées, on met à jour les //associations :

nouveauModele ←+ les associations du modèle de solution

nouveauModele ←+ les associations externes à des classes transformées~~Une association entre deux classes non marquées est récupérée //telle qu'elle ; Une association entre deux classes transformées est remplacée par //les associations (si présentes) dans le modèle de solution du patron ; Une //association d'une classe non transformée et d'une classe transformée est //remplacée par une association de la classe non transformée et de la classe //définie comme point d'accès du modèle de solution (protection contre la //variation).~~

Figure 12. *Algorithme de transformation*

5. Discussion

5.1. Travaux reliés

Plusieurs travaux se sont intéressés à la représentation et~~/ou~~ à la mise en œuvre des patrons de conception. Les chercheurs ont étudié plusieurs modalités d'utilisation de patrons, allant de la génération d'un ensemble d'éléments composant le patron– appelée approche descendante par Flofijin et al. (Flofijin et al., 1997), par exemple (Pagel et Winter, 1996) (Meijler et al., 1997) (Flofijin et al., 1997) (Lauder et Kent, 1998) (Eden et al., 1999) (Albin-Amiot et Guéhéneuc, 2001) (Mens et Tourwé, 2001) (Maplesden et al., 2002) (France et al., 2004) et le cas particulier de la génération de code par instanciation de patrons (Budinsky et al., 1996)—à l'identification d'instances de patrons—appelée approche ascendante par Flofijin et al., par exemple (Eden et al., 1999) (Albin-Amiot et Guéhéneuc, 2001)—à la ré-ingénierie de modèles existants pour les rendre conformes à un patron, par exemple (Alencar et al., 1997) (Eden et al., 1999) (Sunyé et al., 2000). Chacune de ces utilisations a ses propres exigences en terme de représentation.

T~~Les éléments que nous avons identifiés et par lesquels notre approche se distingue sont les suivants :~~

~~L'aspect structurel du patron : t~~ous les travaux de recherche qui ont proposé une représentation explicite des patrons se sont intéressés à ~~cet~~ l'aspect structurel du patron. Certains proposent des méta-modèles pour spécifier les patrons alors que

d'autres proposent un ensemble de modèles ou un reposoir de patrons. Dans les deux cas, la représentation se limite à la structure de la solution proposée par un patron et l'application du patron se fait selon une démarche descendante. En effet, la plupart des approches ont pour objectif de fournir un support d'aide à la conception et non pas la rétro-conception. Le travail de (Budinsky et al., 1996) est le seul qui s'est intéressé à l'ensemble des rubriques employées par (Gamma et al., 1995) pour décrire un patron. Toutefois, l'approche s'est limitée à en fournir une description textuelle. Aucun des travaux ne s'est intéressé à la spécification de la structure du problème résolu par le patron ce qui distingue notre approche pour la représentation des patrons. La représentation formelle et explicite du problème nous permet de caractériser d'une façon plus précise les indications d'utilisation du patron~~ et par la suite la détection automatique de circonstances nécessitant son application à un modèle d'analyse~~. Cette représentation nous permet aussi de spécifier les transformations inhérentes à l'application du patron d'une façon indépendante des modèles d'analyse.

Quant à l'aspect « évolutivité », que nous considérons essentiel à la définition du problème, aucune approche—à notre connaissance—ne l'a considéré dans la définition du problème.

En ce qui concerne les transformations, seules les approches qui se sont intéressées à la rétro-conception de modèles existants ont fourni une représentation explicite des ~~L'aspect général de la représentation : hormis (Lauder et Kent, 1998) (France et al., 2004), dans la majorité des travaux, la structure de la solution proposée par un patron inclut des cardinalités constantes. Notre langage de description des structures (solution, problème) associées aux patrons permet de les représenter d'une façon générale. En plus, il permet de souligner l'évolutivité des différentes structures.~~

~~L'aspect~~ transformations ~~: quelques approches~~ (Alencar et al., 1997) (Sunyé et al., 2000~~) permettent la rétro conception et donc la transformation de modèles existants.~~). Cependant, dans ces cas là, la représentation d'un patron est implicite et indissociable de la transformation associée à son application. Dans notre approche la représentation du patron est explicite et est dissociée de la ~~ainsi que la~~ représentation~~/description~~ de la transformation correspondante~~associée~~. La transformation est spécifiée de~~ une~~ façon déclarative, ce qui nous permet d'automatiser son application. Notre~~ « moteur de~~ transformations ~~eur »~~ est générique et permet d'appliquer une transformation à un modèle ~~d'analyse~~ en se basant sur le modèle de correspondance (problème → solution) et le modèle de solution.~~ Nous étudions actuellement la possibilité de générer le modèle de correspondance (transformation) de façon automatique à partir de la paire de modèles (problème, solution) et cela en utilisant la même terminologie pour décrire le problème et la solution et en introduisant. Cela permettra de traiter d'une façon similaire les variantes des modèles de problème et/ou solution que peut avoir un patron.~~

5.2. Sémantique du méta-modèle et marquage

Dans ce que nous avons réalisé à date, les classes qui font partie des modèles de problèmes ou de solutions n'ont pas de sémantique propre. En effet, le concepteur pourra leur donnera le nom qu'il/elle veut, et les seules contraintes sémantiques sont celles qui sont implicites dans les associations entre ces classes et les autres éléments du modèle auquel elles appartiennent. Dans certains cas, ces contraintes seront suffisantes, mais souvent, ce ne sera pas le cas. Donc, le modèle de problème, tel que décrit dans la section 2, ne sera pas suffisant pour obtenir un marquage automatique du modèle à l'entrée.

La solution que nous avons adoptée consiste à attacher un prédicat d'appartenance à chaque classe de notre métamodèle. Ce prédicat prend comme argument une entité d'un type donné (un classifieur ou une opération ou un attribut) du modèle à l'entrée, et retourne vrai ou faux. Par exemple, pour **Abstraction**, le prédicat peut vérifier, pour une classe donnée, si *toutes* les opérations de la classe sont des opérations abstraites. Une opération est abstraite si elle a été déclarée telle (*abstract*), au cas où le modèle à l'entrée provient de code source, ou la propriété équivalente dans le modèle UML. Ces prédicats d'appartenance vont utiliser l'API réflexive de EMF pour naviguer et évaluer les fragments de modèle à l'entrée, écrits en EMF.

Il est clair que des métaclasses telles que **AbstractOperation** et **AbstractClass** sont d'utilisation générale, et peuvent servir à plusieurs patrons. Le concepteur devrait donc avoir accès à un catalogue de tels concepts pour pouvoir les utiliser tels quels ou les spécialiser. Si une métaclasse A (e.g. **VirtualAbstractOperation**) est une spécialisation d'une métaclasse B (e.g. **AbstractOperation**), la relation suivante doit exister entre leur prédicats d'appartenance :

$$P_A \text{ (<ModelElement>)} \rightarrow P_B \text{ (<ModelElement>)}$$

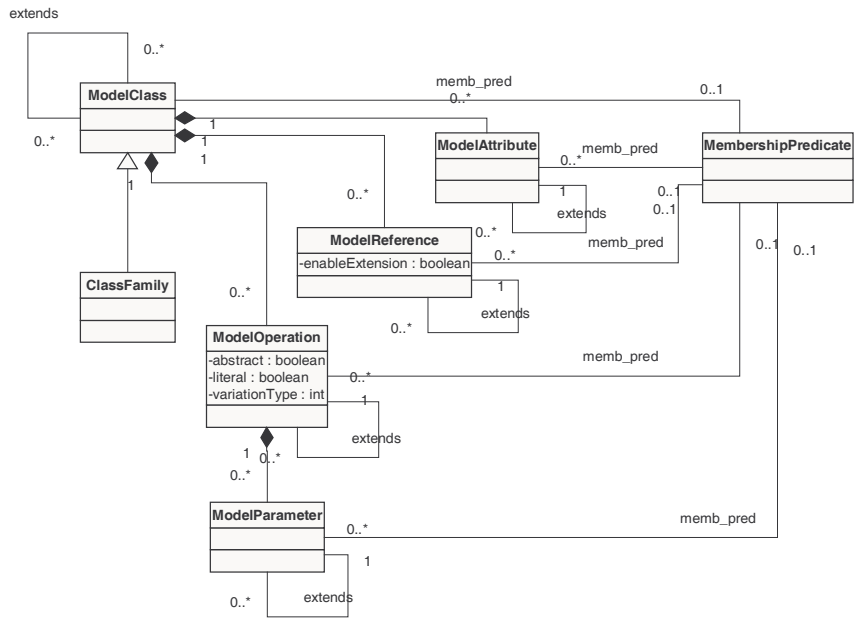
Dans les faits, cela veut dire qu'en spécialisant une métaclasse B par une métaclasse A, le concepteur devra se contenter de préciser les conditions supplémentaires que les instances de A devront satisfaire. Ainsi, pour évaluer l'appartenance d'un élément du modèle à une métaclasse *A*, il faudra prendre la conjonction des prédicats d'appartenance de *A* avec ceux de tous ses ancêtres[4]. ancêtres. La Figure 13 montre des extraits du nouveau méta-modèle montrant les ajouts décrits dans ce paragraphe.

Figure 13. *Extraits du métamodèle de patrons augmenté pour représenter la sémantique des éléments*

[4] Les exceptions sont traitées de façon similaire en Java.

Pour représenter les prédicats d'appartenance, nous avons considéré plusieurs langages dont OCL, les premiers brouillons de QVT (query view transformation) du toujours- « upcoming » standard de l'OMG, et quelques langages hybrides objets- règles (e.g. JESS, ILOG JRules, OPSJ). Nous avons opté pour ces derniers (JRules en particulier), vu leur maturité, et la disponibilité d'outils performants pour l'analyse et l'interprétation de règles. Nous sommes actuellement entrain de développer un moteur de marquage basé sur les principes décrits ci-haut, et de développer un ensemble d'outils dans l'environnement Eclipse pour la manipulation de catalogues de patrons, de méta-classes, et pour la saisie de règles d'appartenance à l'aide de l'éditeur graphique de règles d'ILOG.

Malgré tout, il reste un aspect que nos prédicats d'appartenance ne pourront pas capturer : les scénarios probables d'évolution du modèle à l'entrée, qui feront que tel ou tel patron soit approprié. Cette information est *dynamique* et n'est pas



implicite dans un modèle, qui montre un instantané (*snapshot*) de l'application à l'état actuel. Il y a deux stratégies possibles. Ou bien les concepteurs devront rentrer cette information dans les modèles—par exemple, par le mécanisme des « tag values ». Ou bien on dispose de versions successives d'un logiciel, nous permettant d'identifier les points changeants du logiciel.

Figure 13. *Extraits du métamodèle de patrons augmenté pour représenter la sémantique des éléments*

6. Conclusion

Dans cet article, nous nous sommes intéressés à la représentation et à la mise en œuvre de patrons de conception dans le contexte d'un environnement de développement. Comme pour tout autre artéfact réutilisable, la manipulation de patrons de conception passe par : 1) l'évaluation de l'opportunité d'utilisation de l'artéfact, 2) la compréhension de l'artéfact, et 3) l'application ou intégration de l'artéfact dans le système actuel. Nous avons cherché une caractérisation précise du problème de conception résolu par un patron donné. Cette caractérisation, sous la forme d'un méta-modèle, nous permet de supporter les trois tâches reliées à la (ré)utilisation du patron.

L'approche que nous avons prise est générique et compatible avec le développement transformationnel à la MDA. Comme nous l'avons souligné dans la section 5.2, 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. Capturer ces exigences là de façon structurée et manipulable nous permettra d'offrir un niveau de support accru au niveau de la conception, en général.

7. Bibliographie

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*, 2001.

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.

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.

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

El Boussaidi G., Mili H., Les patrons de conception : représentation et mise en œuvre, rapport de recherche, mars 2004, LATECE.

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

France R.B., 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.

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

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

Maplesden D., Hosking J., Grundy J., « Design Pattern Modelling and Instantiation using DPML », *Proceedings of 14th International Conference on Technology of Object-Oriented Languages and Systems*, 2002.

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

Mens T., Tourwé T., « A Declarative Evolution Framework for Object-Oriented Design Patterns », *Proceedings IEEE International Conference on Software Maintenance*, 2001, p. 570-579.

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

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.

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