Une approche pour améliorer la réutilisabilité des modèles métiers

Pierre Crescenzo, Philippe Lahire

Une approche pour améliorer la réutilisabilité des modèles métiers

Pierre Crescenzo et Philippe Lahire

Laboratoire I3S (CNRS - UNS)
Projet OCL
2000 route des Lucioles
Les Algorithmes, Bâtiment Euclide
BP 121
F-06903 Sophia-Antipolis cedex
France

Pierre.Crescenzo@unice.fr et Philippe.Lahire@unice.fr

RÉSUMÉ. *Dans de précédents travaux, nous avons proposé une approche basée sur la programmation par aspects et la programmation par sujets pour séparer et composer les préoccupations d'une application. L'objectif était, à travers la définition d'un protocole de composition, d'augmenter la réutilisabilité des classes qui forment une application. Dans le présent article nous proposons de faire évoluer cette approche afin de l'appliquer à des modèles et non plus à des programmes. Notre proposition constitue une première tentative dont l'objectif est de présenter un éventail d'adaptations intéressantes et de les valider sur un exemple.*

ABSTRACT. *In previous works, we proposed an approach based on aspect-oriented programming and on subject-oriented programming in order to separate and compose the concerns of an application. The objective was, through the specification of a composition protocol, to increase the reusability of classes which build up an application. In this paper we propose to make changes to this approach in order to apply it to models instead of programs. Our proposal is a first attempt whose objective is to present a first set of interesting adaptations and to validate them on an example.*

MOTS-CLÉS : *séparation des préoccupations, composition de modèles, programmation par sujets, MDA, ingénierie des modèles*

KEYWORDS: *Separation of Concerns, Model Composition, Subject-Oriented Programming, MDA approach, Model Engineering*

1. Introduction

La réutilisation du code informatique écrit pour une application donnée dans le but d'en développer de nouvelles est essentielle pour pouvoir réagir rapidement aux besoins des utilisateurs et du marché. Cependant, elle nécessite souvent un travail difficile et long parce qu'adapter et réutiliser du code existant n'est pas une tâche aisée.

La réutilisabilité représente ainsi un des plus importants défis de la programmation en général et de la programmation à objets en particulier. Mais il faut bien admettre que, malgré les progrès importants des langages de programmation dans ce domaine, le but est assez loin d'être atteint.

Dans de précédents travaux de notre équipe (Quintian, 2004, Lahire *et al.*, 2004), nous avons émis des propositions qui améliorent la réutilisabilité des entités logicielles au niveau de l'activité de programmation. Nous avons pour cela pris appui sur les technologies de séparation des préoccupations et, plus spécifiquement, sur la programmation par aspects et par sujets.

Dans le présent article, nous proposons de transposer nos idées vers le modèle de l'application. Plus précisément, nos solutions précédentes s'appliquent à du code de programme et nous voulons défendre l'idée que la réutilisation doit s'opérer, autant que possible, dès l'étape de conception (UML, EMF d'Eclipse).

Nous souhaitons donc proposer un système d'*adaptateurs* capables d'*exprimer* et de *réaliser* des adaptations, des modifications, des évolutions, voire des fusions de modèles d'application. Nous reprenons dans ce but les techniques utilisées pour l'adaptation de code et proposons et discutons leur transposition au niveau de la modélisation : lorsque le concepteur réalise son graphe de classificateurs UML en reprenant des graphes existant, par exemple.

Nous gérons donc nos adaptations en tirant parti de la technologie de séparation des préoccupations. Trois sortes de préoccupations peuvent être décrites : *i)* fonctionnelles : il s'agit de préoccupations qui s'adressent directement à la tâche principale de l'application et qui lui ajoutent des fonctionnalités ou en modifient ; *ii)* non-fonctionnelles : il s'agit de préoccupations qui concernent des tâches importantes mais non forcément essentielles à l'application (performance, persistance, distribution…) ; *iii)* hybrides : comme ce terme le laisse présager, elles sont en partie fonctionnelles et en partie non-fonctionnelles (exemple : intégration d'un patron de conception, cf. (Lahire *et al.*, 2004)).

Ce découpage en *préoccupations* des applications, autant au niveau conceptuel que durant la programmation, apporte évidemment une plus grande modularité et une meilleure structuration des données et comportements. Il induit également de nouveaux problèmes bien connus en programmation par aspects ou par sujets, et difficiles à résoudre : les préoccupations décrites séparément doivent être ensuite réunies et composées en *respectant leur fonctionnement et leur sémantique*, en *résolvant les inévitables conflits* et en *accordant les éléments définis indépendamment*.

Le système d'adaptateurs décrit dans (Quintian, 2004) et rapidement résumé dans la section 2 donne des éléments de réponse à ces problèmes. Il offre également, grâce à la réification de la notion d'adaptation et à un langage dédié, la possibilité de *réutiliser les adaptations elles-mêmes* pour les appliquer dans différents contextes proches.

Ce sont ces idées que nous souhaitons maintenant transcrire au niveau de la conception, en montrant sur un premier exemple fort simple la possibilité d'adapter — c'est à dire de faire évoluer, de composer, en deux mots de rendre plus réutilisable — la conception d'une application.

Pour atteindre cet objectif, nous nous plaçons dans le cadre de la conception et de la programmation à objets. Le paradigme *objets* est celui qui nous sert de cible, c'est à lui que s'adressent nos propositions, mais aussi de support, c'est en l'utilisant que nous modélisons nos adaptateurs et adaptations.

Dans cet article, nous présentons tout d'abord, dans la section 2, un résumé des principaux concepts et des idées essentielles défendus dans (Quintian, 2004, Lahire *et al.*, 2004) qui s'appliquent aux langages à objets, ainsi qu'une généralisation aux modèles. Cette base nous sert ensuite, dans la section 3, à proposer un nouveau modèle d'adaptation qui s'adresse aux modèles d'application et non plus à leur code. Nous développons ensuite, dans la section 4, un exemple de composition de modèles. Nous donnons un aperçu des travaux connexes dans la section 5, puis nous concluons et énonçons les perspectives dans la section finale 6, y compris concernant l'implémentation.

2. Adaptation dans les langages à objets

Nous présentons ici les principaux aspects développés dans (Quintian, 2004, Lahire *et al.*, 2004). Rappelons que le thème de base est l'amélioration de la réutilisabilité dans les langages à objets. La proposition faite par les auteurs de cet article repose sur une idée simple : réifier la notion d'adaptation dans le but de mieux capturer, définir et modéliser sa sémantique et son comportement. La volonté de permettre la réutilisation de l'adaptation elle-même est également un point intéressant à garder à l'esprit.

2.1. *Différents types d'adaptation*

Les différentes adaptations qu'il est possible de réaliser sur le code d'une application sont nombreuses. Pour les réifier, il est nécessaire de les organiser, de les structurer et de les typer. Voici, dans la figure 1, la hiérarchie de types d'adaptation retenue.

La thèse (Quintian, 2004) donne une description assez fine de cette hiérarchie, que nous vous invitons à consulter. Nous présentons ici une vision globale accompagnée de deux zooms, choisis parmi les cas qui peuvent apporter un éclairage particulier ou un débat.

De manière générale, il a été décrit quatre grandes sortes d'adaptation :

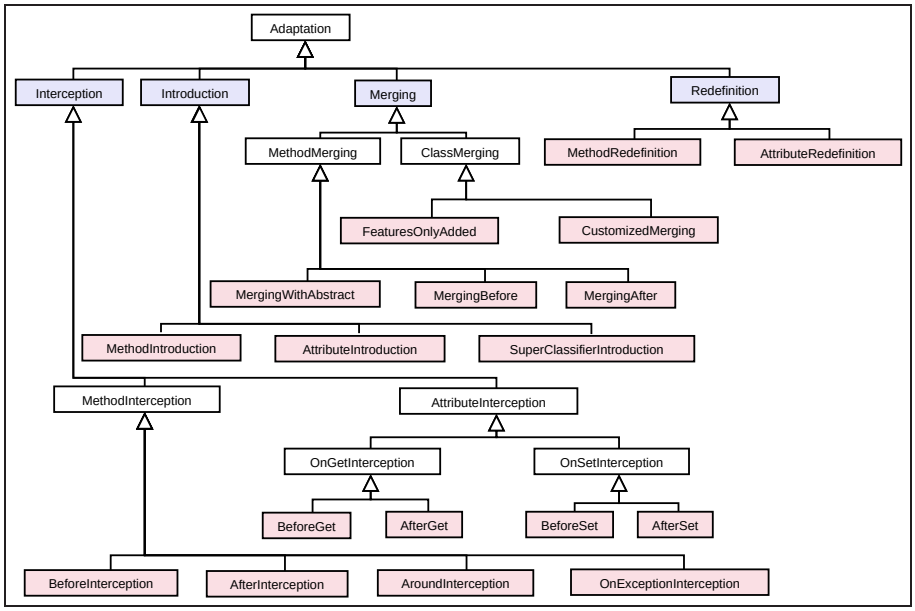

Figure 1. *la hiérarchie des adaptations sur les langages*

1) les *interceptions* (*Interception*) : il s'agit d'intervenir lors de l'appel ou de l'accès aux membres (attributs, méthodes. . .) de classificateur. Ces adaptations permettent d'ajouter du code à une méthode au début, à la fin ou autour du code existant, ou encore d'ajouter un comportement dans le cas d'une levée d'exception. Pour les attributs, les interceptions peuvent se faire lors de l'accès en lecture ou lors de l'accès en écriture.

2) les *ajouts* (*Introduction*) concernent aussi bien les membres de classificateur (ajout d'attribut, de méthode, de constructeur. . .) que la possibilité d'indiquer un nouveau superclassificateur (possibilité qui pose des problèmes différents si le langage-cible est à héritage simple ou multiple).

3) les *fusions* (*Merging*) pour les méthodes et les classificateurs : Les fusions de méthodes sont possible en choisissant lequel des deux corps doit être exécuté avant l'autre dans la méthode résultante. Les fusions de classificateurs (prendre deux classes pour n'en faire qu'une, par exemple) sont soit simples (aucun conflit n'apparaît) soit doivent être assistées par le programmeur. Les fusions d'attributs ne sont, *a priori*, pas prévues.

4) les *redéfinitions* (*Redefinition*) d'attribut ou de méthode doivent respecter un certain nombre de règles qui limitent les erreurs possibles.

Remarquez que les suppressions ne sont pas prévues. Nous nous reposons en effet sur le fait que l'application résultat de la composition doit savoir *faire* au moins tout ce que faisaient ses composants.

Les deux types d'adaptation que nous avons choisis de décrire sont l'*interception de méthode* (*MethodInterception*) et la *fusion complexe de classificateurs* (*CustomizedMerging*). L'interception de méthode est une adaptation d'une très grande utilité. Elle permet notamment d'*ajouter du comportement* dans une méthode. Cependant, elle ne peut pas modifier sa signature (nom de la méthode ; nombre, type et ordre des paramètres ; type éventuel de résultat ; assertions). Il s'agit donc typiquement d'une intégration de code au sein d'une méthode déjà définie.

Cette intégration peut être utile pour plusieurs raisons. L'une des plus évidentes est la correction d'un oubli ou d'une erreur. Ce n'est cependant pas forcément la plus courante. Nous pensons en effet, dans le cadre de la séparation des préoccupations, beaucoup plus à l'expression d'une préoccupation ajoutée (exemple : la trace ou la persistance). Mais en quoi une telle préoccupation nécessite-t-elle l'usage d'une interception de méthode ? Imaginons que nous voulions *sans modifier le code d'une classe*, tracer les appels à une de ses méthodes. Nous allons alors réaliser un adaptateur contenant une adaptation d'*interception de méthode* (en choisissant celle *avant exécution* par exemple). Dans cette adaptation, nous définissons l'ajout d'une ligne de code d'affichage d'une trace. Si nous nous trouvons dans le cas de l'intégration de la persistance, nous pouvons, par exemple, demander le chargement en mémoire ou au contraire la sauvegarde sur disque d'un objet, au début ou à la fin de l'exécution d'une méthode.

La fusion complexe de classificateurs est une des adaptations qui peut requérir l'intervention d'un programmeur. Cette fusion concerne deux classificateurs ou plus, généralement deux, qui vont être regroupés au sein d'un seul. Typiquement, nous prenons deux classes proches pour n'en faire qu'une. Dans la *fusion de classificateurs avec ajouts seulement* (*FeaturesOnlyAdded*), nous ne considérons que les fusions qui ne génèrent aucun conflit : tous les membres ont des signatures et des usages séparés. Ainsi, la fusion se fait par ajouts seulement, les membres de chacun des classificateurs fusionnés étant comme copiés, dans n'importe quel ordre dans le classificateur résultat. Lorsque des conflits apparaissent, la fusion complexe de classificateurs est utilisée. Les conflits peuvent être principalement de deux sortes : de signature ou de sémantique. Pour ces deux sortes, il est parfois possible d'agir automatiquement mais cela reste anecdotique et il est généralement indispensable de demander à un programmeur de lever ces conflits. Cela peut se faire de plusieurs manières : renommage ou adaptation de signatures de membres, fusions de membres dont la sémantique est similaire, suppression de doublons…

Précisons enfin que la hiérarchie n'est pas figée. Il s'agit plus d'un exemple de hiérarchie simple, classique et raisonnable que d'une version complète et très détaillée de celle-ci. Nous pourrions par exemple aisément ajouter l'interception de constructeur ou la fusion d'attribut, à condition de définir précisément leur sémantique et leur comportement.

Vous trouverez, dans la suite du présent article, une nouvelle hiérarchie de type d'adaptation qui s'applique, quant à elle, aux modèles.

2.2. *Modélisation d'une adaptation*

Le second point important que nous souhaitons présenter dans cette partie 2 est constitué par les choix qui ont permis de modéliser une adaptation et une composition d'adaptation. Comme vous pourrez le constater, un souci constant de réutilisabilité a présidé à cette tâche.

Nous proposons, sur la figure 2, les principales entités réifiées.

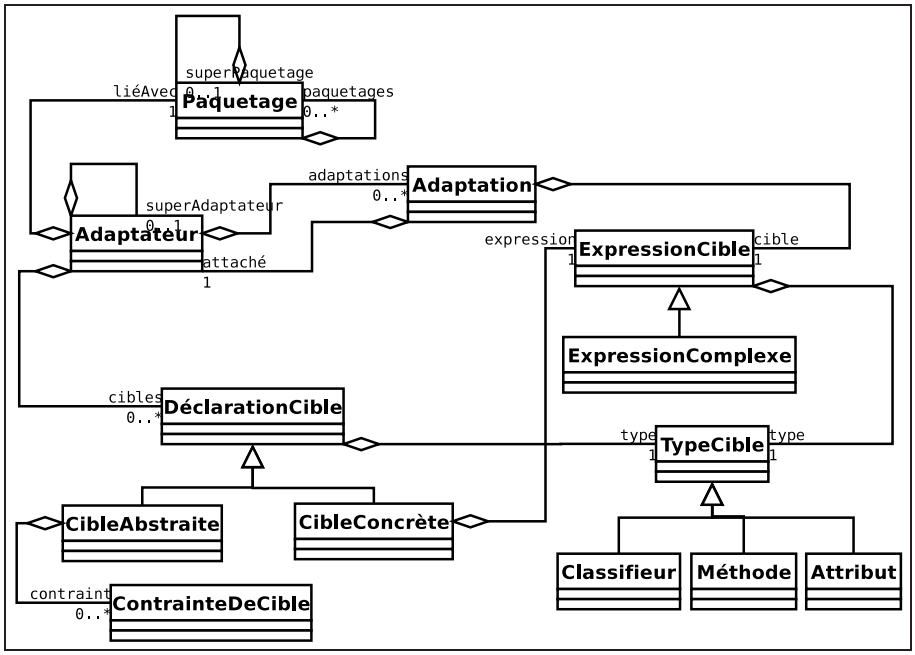

Figure 2. *la modélisation des adaptations sur les langages*

Sur cette figure, vous pouvez remarquer un fait important : les adaptations, et donc les préoccupations, sont encapsulées dans un conteneur que nous nommons Paquetage (au sens général d'*ensemble de classes*) qui regroupe la totalité des classificateurs développés pour cette préoccupation. Il s'agit d'un choix permettant d'offrir une structure contrainte mais large. Nous nous reposons sur une réification traditionnelle des classificateurs : chacun possède un nom, d'éventuels modificateurs (`public`, `deferred`, `final`, `frozen`...), des membres (attributs, attributs de classe, méthodes, méthodes de classe, constructeurs, destructeurs, initialiseurs...) et des assertions (invariants de classe). Les méthodes ont une signature (nom, modificateurs, paramètres, éventuel type de retour), des assertions (préconditions et postconditions) et un corps (structure ordonnée d'instructions).

Commentons plus précisément la figure 2. Chaque adaptation est localisée dans un ou plusieurs adaptateurs (généralement un, mais il peut être pratique, lorsque l'adaptation est complexe et difficile, de la décrire en plusieurs étapes). Elles se trouvent donc hors des classificateurs qu'elles adaptent, il s'agit bien d'une démarche non intrusive. Une composition est constituée d'un ou plusieurs adaptateurs, en général plusieurs. Dans (Quintian, 2004), les règles de composition sont très simples, elles ne peuvent spécifier que l'ordre des adaptations.

Chaque adaptateur est identifié par un nom unique, peut être abstrait ou concret et peut hériter (au sens de la spécialisation) d'un autre adaptateur. Chaque adaptateur référence des cibles d'adaptation (*i. e.* les entités sur lesquelles les adaptations vont s'appliquer) qui peuvent être concrètes ou abstraites : les abstraites permettent une spécification *a posteriori* des cibles, au moment de l'application de l'adaptation, quand la cible réelle est connue. Lorsqu'une cible est abstraite, il est possible de lui associer des contraintes qui devront être vérifiées lors de sa concrétisation. Pour concrétiser une cible d'adaptation, trois options s'offre aux programmeurs : citer une cible, citer une liste de cibles ou encore donner une expression régulière qui identifie un ensemble de cibles. Dans les deux derniers cas, cela permet d'appliquer l'adaptation à plus d'une cible.

Chaque adaptation est typée en fonction du type de l'entité sur laquelle elle doit s'appliquer. Il y a donc des adaptations pour les classificateurs, pour les méthodes, pour les attributs, etc.

Enfin, précisons que chaque adaptation peut être déclarée *in situ* (dans ce cas, elle est réalisée par modification des classificateurs cibles) ou *ex situ* (de nouveaux classificateurs sont alors créés pour intégrer les éléments de l'adaptation dont les cibles ne sont pas modifiées directement).

Les éléments présentés jusqu'ici sont issus de la thèse de doctorat de Laurent Quintian (Quintian, 2004) et d'un article de revue de Philippe Lahire et Laurent Quintian (Lahire *et al.*, 2004). Nous vous invitons à vous référer à ces deux documents pour une description plus détaillée qui n'avait pas sa place ici. Passons maintenant à la transposition de ces idées au monde des modèles d'application.

2.3. *Adaptation de l'approche pour les modèles*

Par rapport au modèle proposé dans (Quintian, 2004) pour l'adaptation de programmes à objets, les adaptations proposées dans la figure 3 ont, comme nous le verrons dans la suite, évolué. Elles ne concernent pour l'instant que la partie relative au diagramme de classes (modèle MOF, EMF ou diagramme des classes UML) à l'exclusion de toute information liée de près ou de loin à la description du corps d'une méthode. Nous retrouvons ces informations dans les autres diagrammes d'UML (diagrammes de séquence, de collaboration, d'activité ou de communication) (Rumbaugh *et al.*, 2004).

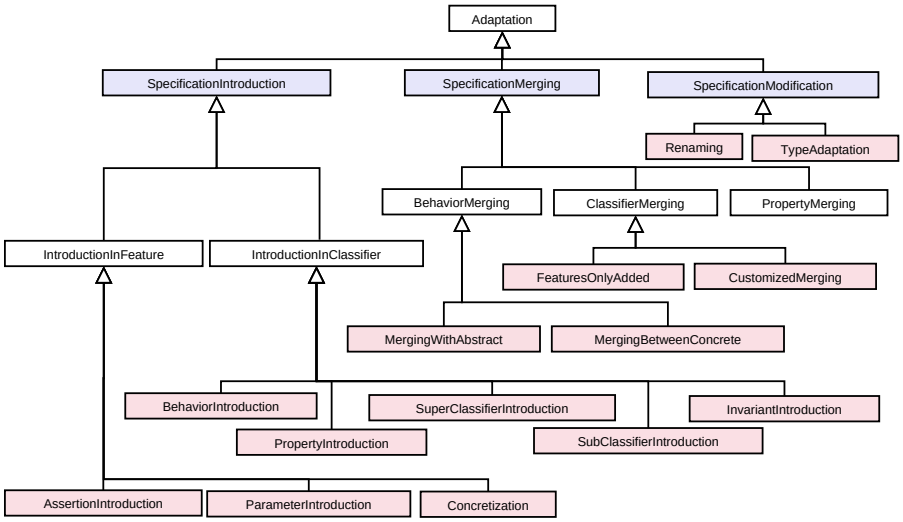

Figure 3. *la hiérarchie des adaptations sur les modèles*

La figure 3 propose trois grandes catégories d'adaptation [1] : l'introduction, la fusion et la modification de spécifications. Par rapport aux adaptations mentionnées dans la section 2, nous remarquons bien évidemment qu'il n'est citée que la spécification de propriétés, de comportement ou de classes et non plus leur programmation. C'est pourquoi l'insertion de primitive (*BehaviorIntroduction* ou *PropertyIntroduction*) concerne la signature d'une propriété (attribut, association, etc.) ou d'une méthode (procédure, fonction, constructeur, etc.). Toujours au niveau d'un classificateur, nous pouvons insérer un nouvel invariant (*InvariantIntroduction*) et un nouveau classificateur (ce classificateur peut être par exemple une interface, une classe abstraite ou concrète) que ce soit comme ancêtre ou descendant (*SuperClassifierIntroduction* ou *SubClassifierIntroduction*). Dans un métamodèle comme UML, par exemple, nous bénéficions de l'héritage multiple et donc le nombre de classes parentes n'est pas limité *a priori*. Lors de la composition de deux modèles M et M', il est particulièrement intéressant de pouvoir spécifier l'ajout dans M d'une ou plusieurs classes descendantes C'1,. . . ,C'n de M' pour une classe C [2] : cela permet en particulier de prendre en compte des modèles non isomorphes. Les informations permettant de décrire une association sont plus nombreuses (rôle(s), type des extrémités, cardinalité, etc.) ; le mécanisme d'adaptation mis en œuvre doit le permettre. Pour être cohérent par rapport aux adap-

1. Pour simplifier la hiérarchie, dans un premier temps, nous avons volontairement réduit son niveau de détail mais, pour une meilleure flexibilité, il pourra être intéressant de l'augmenter.
2. Il est à noter que pour favoriser la flexibilité et la réutilisation des modèles, un adaptateur abstrait peut indiquer qu'il faut penser à donner la liste des descendants alors que l'adaptateur spécifique mentionne une liste vide.

tations de classificateurs, il devra être possible d'introduire de nouvelles assertions (*AssertionIntroduction*) et de nouveaux paramètres (*ParameterIntroduction*) pour une méthode existante. Il est aussi intéressant de pouvoir spécifier dans une adaptation qu'une méthode originellement abstraite doit être concrétisée dans le modèle (*Concretization*).

Un autre aspect important concerne la fusion de spécifications de classe ou de primitive. C'est un type d'adaptation qui est essentiel lorsque nous voulons composer des fonctionnalités décrites sur des modèles dont le niveau de généralité est différent (par exemple l'un est très général et comporte des entités abstraites et l'autre est très concret), ou qui sont plusieurs vues d'un même modèle. Les adaptations que nous proposons permettent de fusionner les spécifications de deux méthodes, constructeurs, attributs ou associations suivant des règles qui seront précisées plus bas. De même, il est possible, en s'appuyant sur la fusion des primitives mentionnées ci-dessus, de fusionner des classificateurs entre eux.

La sous-hiérarchie qui décrit la fusion des classes, des propriétés et des méthodes est proche de celle proposée dans la section 2 mais elle concerne la signature des propriétés et des méthodes et devra prendre en compte notamment les assertions des méthodes, les invariants de classe et les particularités d'une association. Entre les adaptations de type *MergingWithAbstract* et *MergingWithConcrete* la différence essentielle est que, dans un cas, les deux méthodes sont marquées concrètes alors que, dans l'autre cas, l'une au moins est abstraite. En prenant en compte d'autres diagrammes que le diagramme des classes ou en proposant un mini-langage de description du comportement, cela augmenterait sensiblement l'intérêt de différencier les deux types de fusion.

D'autres adaptations sont proposées pour permettre le renommage de méthodes ou de classes (*Renaming*) ou bien pour adapter le type de retour d'une méthode ou le type d'un attribut ou du rôle d'une association (*TypeAdaptation*).

3. Réutilisation de préoccupations exprimées par des modèles

Dans cette partie, nous allons vous présenter trois préoccupations, décrites au niveau du modèle d'application (UML), et leur composition.

Nous reprenons l'exemple proposé dans (Tarr *et al.,* 1999, Janzen *et al.,* 2004) et modifié dans (Quintian, 2004). Nous souhaitons montrer la capacité de composition de plusieurs préoccupations relatives à la notion d'arbre et, plus précisément pour certaines, à des arbres d'expressions arithmétiques (à valeurs entières, pour simplifier). Les modèles manipulés sont autosuffisants à ceci près qu'ils nécessitent la création de descendant pour certaines classes afin d'être opérationnel.

Vous remarquerez qu'il serait peut-être bon de pouvoir dire, dans le modèle, que des descendants sont attendus à certains endroits comme par exemple `unaryExpression` et `binaryExpression` (voir figure 8). En quelque sorte, nous intégrerions la notion de *classe requise*. Nous ne le faisons pas dans nos exemples

pour ne pas compliquer inutilement les choses mais, pour spécifier cette information, plusieurs solutions seraient possibles :

– définir des interfaces et non des classes. Cette approche présente plusieurs inconvénients : une interface ne pouvant hériter d'une classe, la majeure partie de la hiérarchie devrait être composé d'interfaces ce qui ne permettrait pas la spécification de méthodes pouvant être concrètes et surtout ce qui interférerait avec l'information que nous voulons faire passer quand nous spécifions réellement une interface.

– définir des classes abstraites lorsque des descendants sont attendus. Mais nous voulons pouvoir faire passer deux informations différentes avec le même mot-clé : l'entité peut se trouver au milieu de la hiérarchie, ne peut avoir d'instance et participer à la structuration de l'information (c'est le sens usuel) ou le modèle n'est pas utilisable si cette entité n'a pas de descendant.

Il semble intéressant d'introduire un moyen de définir de manière uniforme le fait qu'un descendant ou qu'une méthode est requise.

3.1. *Préoccupation* Évaluation des expressions arithmétiques

La figure 4 décrit la mise en œuvre d'un mécanisme d'évaluation pour une expression arithmétique. Les différents opérateurs binaires (plus, moins, etc.) ou unaire (moins, etc.) sont décrits avec leur méthode d'évaluation. Par exemple, la méthode `eval()` de l'opérateur binaire `Plus` aurait pour code `return left.eval()+right.eval()`. La méthode `value()` est abstraite. Cela signifie que la préoccupation n'a pas assez d'information pour l'implanter et que cette méthode devra être présente de manière concrète dans une des préoccupations à composer. Il en va de même pour les méthodes `getLeft()` et `getRight()` et pour les opérateurs binaires en général (Muller *et al.*, 2003).

Le protocole de composition qui permettra de réutiliser ce modèle pour le composer avec d'autres modèles est donné dans la figure 5. Pour le décrire nous utilisons la syntaxe du langage dédié proposé (la grammaire précise de ce langage n'est pas donnée ici, les exemples étant relativement explicites).

Remarquons un certain nombre de points :

– Dans le protocole de composition nous posons une contrainte sur certaines cibles d'adaptation pour être certains que les méthodes décrites dans la classe correspondant à *expressionClass* (lignes *06*, *08*, *12*) seront présentes.

– Une autre contrainte est placée sur les cibles d'adaptation *unaryOpClass* et *binaryOpClass* (lignes *09*, *10*, *13*) : comme le modèle de la figure 4 l'indique, il est nécessaire de mettre en évidence la nécessité de retrouver dans les classes qui leur correspondent une version concrète des méthodes abstraites se trouvant dans ces classes et permettant d'accéder à la réification. Il est à noter que c'est un choix fait lors de l'écriture de ce protocole et qu'il aurait été possible de proposer un protocole n'intégrant pas cette contrainte mais proposant en plus une adaptation de "concrétisation"

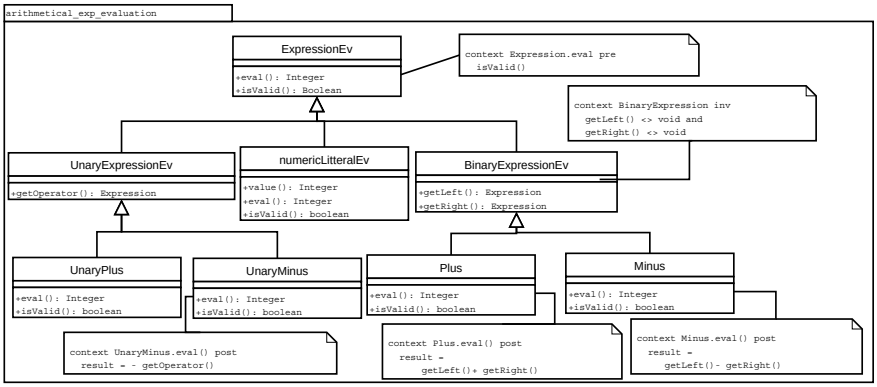

Figure 4. *le modèle décrivant l'évaluation des expressions arithmétiques*

```
01 concern example.arithmetical_exp_evaluation
02 abstract adapter EvaluationProtocolAdapter {
03
04 abstract Class target ''class(es) being used as an expression'' : expressionClass
05 abstract Class target ''class(es) being used as a numeric litteral'' : numericClass
06 require numericClass inherit from expressionClass
07 abstract Class target '' class(es) used as a binary operator'' : binaryOpClass
08 require binaryOpClass inherit from expressionClass
09 require binaryOpClass exists concrete expressionClass getLeft()
10 require binaryOpClass exists concrete expressionClass getRight()
11 abstract Class target '' class(es) used as a unary operator'' : unaryOpClass
12 require unaryOpClass inherit from expressionClass
13 require unaryOpClass exists concrete expressionClass getOperator()
14
15 adaptation becomeNumeric ''Modify class to make it an expression'' :
16 extend class expressionClass with ExpressionEv
17 adaptation becomeNumeric ''Modify class to make it a numeric'' :
18 extend class numericClass with NumericLitteralEv
19 adaptation becomeBinaryOp ''Modify class to make it a binary operator'' :
20 extend class binaryOpClass with BinaryExpressionEv
21 adaptation becomeUnaryOp ''Modify class to make it an unary operator'' :
22 extend class unaryOpClass with UnaryExpressionEv
23
24 abstract adaptation numericUpdate ''Concretize method value in numeric class'' :
25 concretize method public Integer value() in numericClass
26 abstract adaptation binaryDescendant ''Suggest to add descendant(s) to BinaryOp class'' :
27       add descendant to binaryOpClass
28 abstract adaptation unaryDescendant ''Suggest to add descendant(s) to UnaryOp class'' :
29       add descendant to unaryOpClass
30  }
```

Figure 5. *le protocole de composition associé à l'évaluation d'une expression arithmétique*

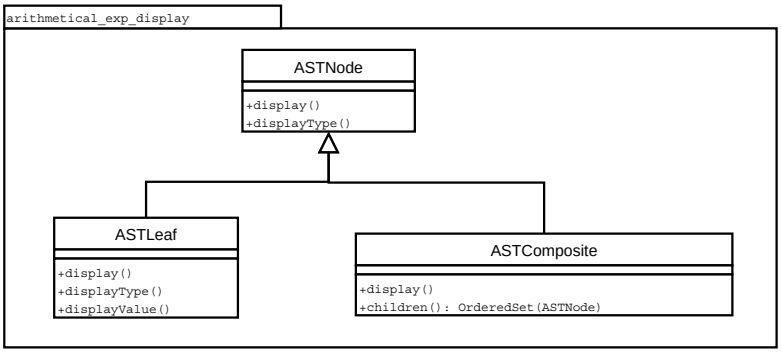

Figure 6. *le modèle décrivant l'affichage d'un arbre*

comme cela est le cas pour la méthode *value()* (lignes *24* et *25*).

– Il est suggéré aussi (lignes *26* à *29*) d'introduire dans le modèle cible une ou plusieurs classes comme descendantes de *unaryOpClass* et *binaryOpClass*. Cette adaptation a ici un caractère de suggestion plutôt que d'obligation pour augmenter la flexibilité[3].

3.2. *Préoccupation* Affichage d'un arbre

Le traitement de l'affichage d'un arbre est décrit dans la figure 6. Il est à noter que contrairement aux deux autres préoccupations qui ont une racine commune, cette préoccupation s'applique plus généralement à un arbre (l'arbre des expressions arithmétiques est uniquement un des arbres possibles). Il est donc normal que les noms diffèrent beaucoup. Notons en particulier que la préoccupation ne sait pas comment implémenter l'affichage d'une valeur puis qu'elle ne connaît pas la forme de la valeur, et il en va de même pour le type de nœud de l'arbre. Pour illustrer cette non-connaissance, les primitives `displayType()` et `displayValue()` sont abstraites. À ce niveau de généralité, nous ne savons pas non plus comment mettre en œuvre la gestion des nœuds fils, c'est pourquoi la méthode `children` est également abstraite.

Si nous considérons l'aspect de la réutilisation, parmi les trois préoccupations citées, l'affichage de structures arborescentes est sans aucun doute celle qui est potentiellement la plus réutilisable et il semble donc intéressant d'équiper le modèle associé d'un manuel de réutilisation (ou protocole de composition). Ce protocole de composition permettra de guider le concepteur dans le processus d'élaboration du modèle résultat.

3. Nous pourrions envisager de mettre une contrainte minimale sur le nombre de descendants (par exemple : au moins 1).

```
01 concern example.arithmetical_expr_display
02 abstract adapter DisplayProtocolAdapter {
03
04 abstract Class target ``class(es) being used as a tree node'' : nodeClass
05 abstract Class target `` class(es) used as a leaf'' : leafClass
06 require leafClass inherit from nodeClass
07 abstract Class target `` class(es) used as a composite'' : compositeClass
08 require compositeClass inherit from nodeClass
09
10 adaptation becomeNode ``Modify class to make it a node'' :
11       extend class nodeClass with ASTNode
12 adaptation becomeLeaf ``Modify class to make it a leaf'' :
13       extend class leafClass with ASTLeaf
14 adaptation becomeComposite ``Modify class to make it a composite'' :
15       extend class compositeClass with ASTComposite
16
18 abstract adaptation NodeUpdate ``Concretize method displayType in Node class'' :
19 concretize method public void displayType() in nodeClass
20 abstract adaptation LeafUpdate ``Concretize method displayValue in Leaf class'' :
21 concretize method public void displayValue() in leafClass
22 abstract adaptation CompositeUpdate ``Concretize method children in Composite class'' :
23 concretize method public ASTNode[] children() in compositeClass
24  }
```

Figure 7. *le protocole de composition associé à l'affichage d'un arbre*

Le protocole de composition contient les adaptations suivantes :

– identification des classes jouant des rôles spécifiés par la préoccupation : *ASTNode*, *ASTLeaf* et *ASTComposite* : chaque classe qui jouera un de ces rôles devra être adaptée.

– abstraction de l'implémentation des trois méthodes suivantes : *ASTNode.displayType()*, *ASTLeaf.displayValue()* et *ASTComposite.children()*.

La description de ce protocole de composition utilisant lui aussi la syntaxe du langage dédié proposé est donné dans la figure 7. Retrouvons dans l'adaptateur qui y est décrit tous les éléments nécessaires au protocole de composition de la préoccupation :

– les trois variables de type *Class* représentent les classes clientes et jouent chacune l'un des trois rôles nécessaires à la préoccupation (voir lignes 04 à 08) ;

– la fusion (adaptation *CustomizedMerging*) des classes de la préoccupation dans les classes clientes (voir lignes 10 à 15) ;

– la nécessité pour les classes clientes de concrétiser les trois méthodes abstraites introduites lors de la fusion (voir lignes 18 à 23).

Il est intéressant de noter les contraintes qui sont introduites lors de la définition des cibles d'adaptations abstraites (lignes 6 et 8). Elles permettent de spécifier que les classes (définies dans le "second" modèle) qui joueront successivement le rôle de nœud intermédiaire (*compositeClass*) et terminal (*leafClass*) de l'arbre à afficher devront hériter d'une de celles identifiées par la cible d'adaptation *nodeClass*. Elles permettent de garantir que toutes les méthodes nécessaires à leur mise en œuvre sont présentes. Cette contrainte peut apparaître forte et donc réduire sensiblement le nombre

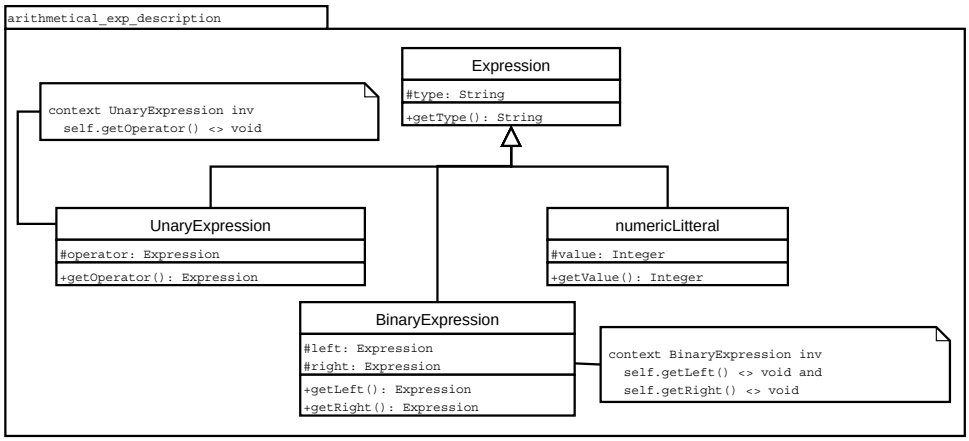

Figure 8. *le modèle décrivant l'arbre des expressions arithmétiques*

de modèles dans lequel la préoccupation d'affichage pourrait s'intégrer. Aussi, nous aurions très bien pu choisir de décrire autrement le protocole afin de relâcher cette contrainte (par exemple en introduisant dans chaque classe identifiée par *compositeClass* et *leafClass* une nouvelle classe parente (adaptation de type *superClassifierIntroduction*) identifiée par *nodeClass*.

Notre approche permet donc d'encapsuler complètement le protocole de composition de la préoccupation et de réaliser l'affichage des structures arborescentes. Ce protocole de composition va pouvoir être réutilisé par héritage dans un nouvel adaptateur qui va composer directement les trois préoccupations (voir section 4).

3.3. *Préoccupation* Description des expressions arithmétiques

La figure 8 décrit la syntaxe abstraite d'une expression arithmétique qui permet de prendre en compte des expressions binaires, unaires et des littéraux. Compte tenu des informations nécessaires il n'est pas utile ici de décrire plus finement les expressions binaires ou unaires.

Ce modèle ne comporte aucune classe terminale qui ne soit complètement implémentée ; il contient donc des choix d'implémentation de la réification. Conformément au niveau assez général de la description il serait possible de produire un protocole de réutilisation même si celui-ci aurait une porté limitée. Compte-tenu de ces remarques et de la place réduite dont nous disposons, nous avons choisi ici de ne pas le décrire.

4. Composition des trois modèles

Plaçons-nous maintenant dans le cadre d'un environnement dédié à l'élaboration d'expressions numériques dans lequel nous voudrions pouvoir disposer d'une réification, d'un mécanisme d'évaluation et d'un autre pour l'affichage [4]. Il est donc nécessaire de composer les trois préoccupations mentionnées ci-dessus. Chaque préoccupation est associée à un paquetage qui permet de l'identifier. Suivant une approche similaire à celle proposée dans (Quintian, 2004), nous considérons deux étapes bien distinctes : *i)* la définition des préoccupations dans le but d'améliorer leur degré de réutilisabilité et de favoriser le guidage et le contrôle de sa composition avec d'autres le moment voulu (section 3) et *ii)* la composition elle-même décrite ci-après.

La composition peut être réalisée *in situ* ou *ex situ*. La composition *in situ* est une composition asymétrique qui s'adresse plutôt aux préoccupations non-fonctionnelles (contenues dans un modèle réutilisable - notons le R) que l'on veut ajouter à des modèles clients. Ce mode de composition modifie directement les modèles clients pour y intégrer le contenu du modèle R. La composition *ex situ* est une composition symétrique qui placera au même niveau les modèles à composer. Ce mode de composition permet de garder inchangés les modèles à composer car leur composition génère un nouveau modèle qui peut alors être utilisé comme n'importe quel autre. La composition *ex situ* est particulièrement utile lorsque les préoccupations sont fonctionnelles [5]. Dans l'exemple proposé ici, nous utiliserons tour à tour ces deux modes de composition.

Les deux protocoles de composition décrits dans les sections 3.1 et 3.2 vont être utilisés pour construire les deux adaptateurs spécifiques qui permettront successivement de composer :

– Les préoccupations *Évaluation des expressions arithmétiques* et *Description des expressions arithmétiques*. Cette composition est faite *ex situ*, ce qui entraîne la création d'une nouvelle préoccupation notée *P* qui correspond à leur composition (*Description et évaluation des expressions arithmétiques*). *P* est associée à un paquetage différent de ceux contenant la définition des deux autres préoccupations.

– Les préoccupations *Affichage d'un arbre* et *P*. Cette composition est faite *in situ* : la préoccupation *P* est modifiée par la première et devient donc *Description, évaluation et affichage des expressions arithmétiques*.

La figure 9 décrit la première composition tandis que la figure 10 donne la seconde. L'ordre d'exécution est important car chaque adaptateur spécifique a été construit de manière à réaliser l'adaptation dans un ordre donné. Il serait possible de réaliser ces compositions dans l'ordre inverse mais il faudrait pour cela modifier le contenu des adaptateurs. Il est à noter que nous avons aussi fait le choix de rendre réutilisable (en

4. Un exemple plus conséquent pourrait concerner un environnement dédié aux expressions OCL.
5. C'est en particulier essentiel lorsque la même préoccupation doit être réutilisée plusieurs fois à différents endroits d'un même modèle client.

```
01 concern example.arithmetical_exp_composition
02 compose example.arithmetical_exp_evaluation with example.arithmetical_exp_description
03 adapter DescriptionAndEvaluation extends EvaluationProtocolAdapter {
04   target expressionClass = example.arithmetical_exp_description.Expression
05   target numericClass = example.arithmetical_exp_description.NumericLitteral
06 target binaryOpClass = example.arithmetical_exp_description.BinaryExpression
07   target unaryOpClass = example.arithmetical_exp_description.UnaryExpression
07   target unaryOpClass = example.arithmetical_exp_description.UnaryExpression
08
09 adaptation numericUpdate :
10 concretize method public Integer value() in numericClass with
11             post result = numericClass.getValue()
12 adaptation binaryDescendant :
13       add descendant Plus, Minus to binaryOpClass
14 adaptation unaryDescendant :
15       add descendant UnaryMinus to unaryOpClass
16  }
```

Figure 9. *la composition de la description et de l'évaluation des expressions arithmétiques*

leur associant un protocole de composition) seulement deux des préoccupations. Nous avons donc naturellement choisi la troisième préoccupation pour cible mais il aurait été possible de faire un adaptateur pour elle ou de choisir n'importe lequel des trois modèles si chacun d'eux était équipé de son protocole de composition.

Notons (figure 9) que le caractère *ex situ* est matérialisé à la ligne *01* par le fait que le paquetage associé à la préoccupation qui résulte de la composition est différent de ceux des deux préoccupations concernées par la composition. Au contraire, dans la composition décrite dans la figure 10, le mode utilisé est *in situ* pour les raisons inverses. L'ordre de réalisation de ces composition est évident puisque celle décrite dans cette même figure nécessite l'existence de la préoccupation *example.arithmetical_exp_composition*.

Chacun des adaptateurs spécifiques (*DescriptionAndEvaluation* et *DescriptionAndEvaluationAndDisplay*) explicite le contenu des cibles d'adaptation qui doit satisfaire les contraintes posées dans les adaptateurs dont ils héritent. De même les adaptations abstraites sont finalisées. Remarquons en particulier que la description du corps des méthodes *value()* (lignes *08* à *10* de la figure 9) et *children()* (lignes *16* à *20* de la figure 10) peut s'exprimer par une postcondition tandis qu'il est plus difficile de le faire pour la méthode comme *displayValue()* de cette dernière figure pour lequel nous avons décrit le comportement par une simple annotation.

4.1. *Résultat de la composition*

Nous décrivons maintenant le résultat des deux compositions définies successivement par les adaptateurs des figures 9 et 10. La première composition est *ex situ* car l'adaptateur correspondant indique par les lignes 01 et 02, que la préoccupation associée à l'adaptateur (*example.arithmetical_exp_composition*) ne fait pas

```
01 concern example.arithmetical_exp_composition
02 compose example.arithmetical_exp_display with example.arithmetical_exp_composition
03 adapter DescriptionAndEvaluationAndDisplay extends DisplayProtocolAdapter {
04   target nodeClass = example.arithmetical_exp_composition.Expression
05   target leafClass = example.arithmetical_exp_composition.NumericLitteral
06 target compositeClass = example.arithmetical_exp_composition.UnaryExpression,
07                         example.arithmetical_exp_composition.BinaryExpression
08
09 adaptation NodeUpdate :
10 concretize method public void displayType() in nodeClass with
11             annotation ‘‘display nodeClass.getType()’’
12   adaptation LeafUpdate :
13 concretize method public void displayValue() in leafClass with
14             annotation ‘‘display leafClass.getValue()’’
15 adaptation CompositeUpdate :
16 concretize method public OrderedSet(nodeClass) children() in compositeClass with
17             post self.OclIsTypeOf(example.arithmetical_exp_composition.UnaryExpression)
18                       implies result = OrderedSet(compositeClass.getOperator())
19             post self.OclIsTypeOf(example.arithmetical_exp_composition.BinaryExpression)
20                       implies result = OrderedSet(compositeClass.getLeft()).union (
21                                        OrderedSet(compositeClass.getRight()))
22  }
```

Figure 10. *la composition de la description, de l'évaluation et de l'affichage des expressions arithmétiques*

partie des deux préoccupations à composer (*example.arithmetical_exp_display* et *example.arithmetical_exp_composition*). Par contre c'est le cas dans le second adaptateur et la composition est alors *in situ*.

4.1.1. *Première composition (ex situ)*

À partir des cibles d'adaptation maintenant connues (figure 9), nous vérifions dans un premier temps que les contraintes du protocole de composition (figure 5) sont satisfaites. Nous constatons que les contraintes d'héritage induites respectivement par les lignes 06, 08 et 12 du protocole sont respectées :

– *NumericLitteral* (cible *numericClass*) hérite de *Expression* (cible *expressionClass*),

– *BinaryExpression* (cible *binaryOpClass*) hérite de *Expression* (cible *expressionClass*), et

– *UnaryExpression* (cible *unaryOpClass*) hérite de *Expression* (cible *expressionClass*).

De même, nous voyons que les contraintes d'existence des méthodes (lignes 09, 10 et 13) sont elles aussi respectées. En effet, *BinaryExpression* (cible *binaryOpClass*) contient les méthodes concrètes *getLeft* et *getRight*, tandis que *UnaryExpression* (cible *unaryOpClass*) contient la méthode concrète *getOperator*.

Dans un deuxième temps, les adaptations décrites dans les adaptateurs concrets et abstraits (figures 9 et 5) sont réalisées et produisent le résultat suivant (synthétisé dans la figure 11) :

– Les méthodes *eval* et *isValid* sont ajoutées au contenu de la classe *Expression* (cible *expressionClass*). Elles proviennent de *ExpressionEv* et sont toutes les deux abstraites.

– Les méthodes *eval* et *isValid* sont ajoutées à la classe *NumericLitteral* (cible *numericClass*). Elles sont toutes les deux concrètes et proviennent de *NumericLitteralEv*. La primitive *value* était déjà présente sous forme d'un attribut ; elle est insérée comme une méthode abstraite[6]. Cette méthode est concrétisée (lignes 09 et 10 de la figure 9) suivant ainsi la contrainte fixée par le protocole de composition. À cette occasion, une autre adaptation permet l'ajout d'une postcondition qui utilise des primitives de *NumericLitteral* (Ligne 11 de la figure 9).

– À propos des classes *BinaryExpression* (cible *binaryOpClass*) et *UnaryExpression* (cible *unaryOpClass*) rien ne change car les méthodes de *BinaryExpressionEv* et *unaryOpClassEv* qui devraient être insérées sont abstraites alors qu'elles le sont déjà dans les classes *BinaryExpression* et *UnaryExpression*. Cependant, ces deux classes initialement sans héritières se voient ajoutés (conformément aux spécifications du protocole de composition), les descendants *Plus* et *Minus* pour la première et *UnaryMinus* pour la seconde (lignes 12 à 15 de la figure 9).

4.1.2. *Deuxième composition (in situ)*

Comme pour la première composition, nous vérifions d'abord que l'adaptateur de la figure 10 permet de satisfaire les contraintes du protocole de composition (figure 7). Les contraintes d'héritage induites respectivement par les lignes 06 et 08 de l'adaptateur sont respectées : la classe *NumericLitteral* (cible *leafClass*) hérite de la classe *Expression* (cible *nodeClass*) et *BinaryExpression* et *UnaryExpression* (cible *compositeClass*) hérite de *Expression* (cible *nodeClass*).

Ensuite nous réalisons les adaptations décrites dans les adaptateurs concrets et abstraits (figures 10 et 7). Le résultat de la première composition décrite dans la figure 11 est modifié pour produire le schéma final décrit dans la figure 11. Nous noterons en particulier les effets suivants :

– Les méthodes *display* et *displayType* sont ajoutées au contenu de la classe *Expression* (cible *nodeClass*). Elles proviennent de la classe *ASTNode* où *display* est concrète et *displayType* abstraite. *displayType* est concrétisée (lignes 09 et 10 de la figure 10) dans la classe *Expression*. Une annotation est ajoutée (non visible sur le schéma) car le rôle de la méthode n'est pas facilement exprimable par l'ajout d'une postcondition[7].

– les méthodes *display*, *displayType* et *displayValue* sont ajoutées à la classe *NumericLitteral* (cible *leafClass*). Elles proviennent de la classe *ASTLeaf* ; elles sont toutes

6. Si nous admettons, comme c'est le cas dans UML, que la surcharge est autorisée alors il n'y a rien à faire, sinon il faudra utiliser en complément une adaptation de renommage (*renaming*) prévue dans le modèle.
7. Elle n'apparaît pas dans la hiérarchie d'adaptation (figure 3) car pour l'instant nous considérons que c'est un paramètre de l'adaptation.

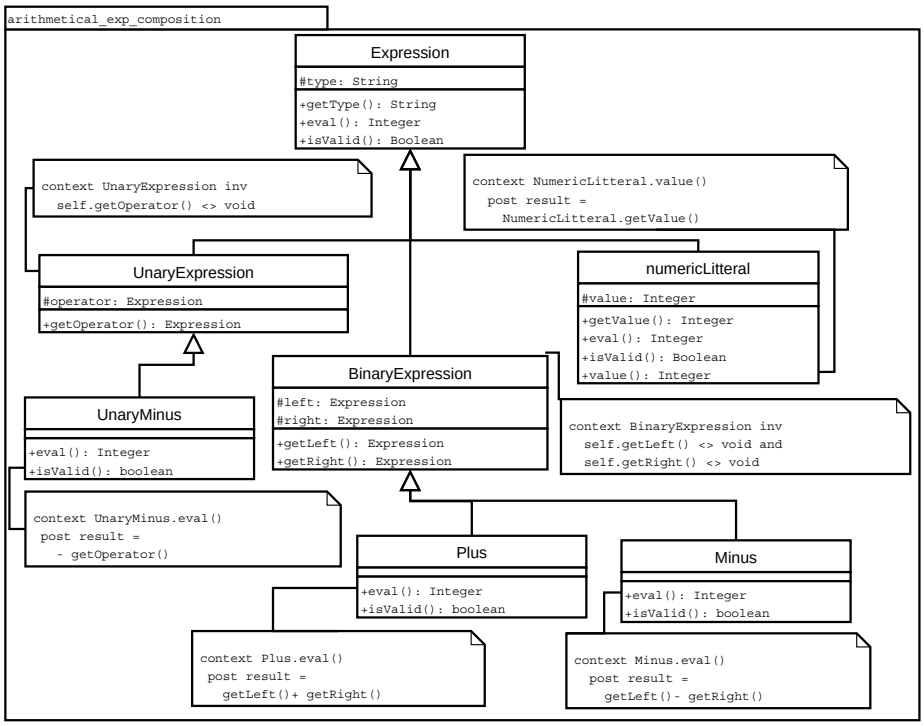

Figure 11. *Résultat de la première composition*

concrètes sauf *displayValue* qui est abstraite. Cette dernière est concrétisée (lignes 12 et 13 de la figure 10) dans la classe *NumericLitteral*. Comme ci-dessus, une annotation est aussi ajoutée.

– Les méthodes *display* et *children* sont ajoutées aux classes *BinaryExpression* et *UnaryExpression* (cible *compositeClass*). Elles proviennent de la classe *ASTComposite* où *children* était abstraite. Elle est concrétisée (lignes 15 et 16 de la figure 10) dans les classes *BinaryExpression* et *UnaryExpression*. Deux autres postconditions sont aussi ajoutées suivant la classe concernée ; elles utilisent selon le cas des primitives de *BinaryExpression* ou *UnaryExpression* (Lignes 17 à 21 de la figure 10)

5. Travaux connexes

Nos travaux se rapprochent de (Estublier *et al.*, 2005) mais notre approche se démarque par le fait qu'elle n'est pas une extension d'UML dédiée à la composition de modèle mais plutôt un métamodèle métier et un langage métier orthogonaux. D'une manière générale il semble intéressant de se positionner à la fois par rapport aux tra-

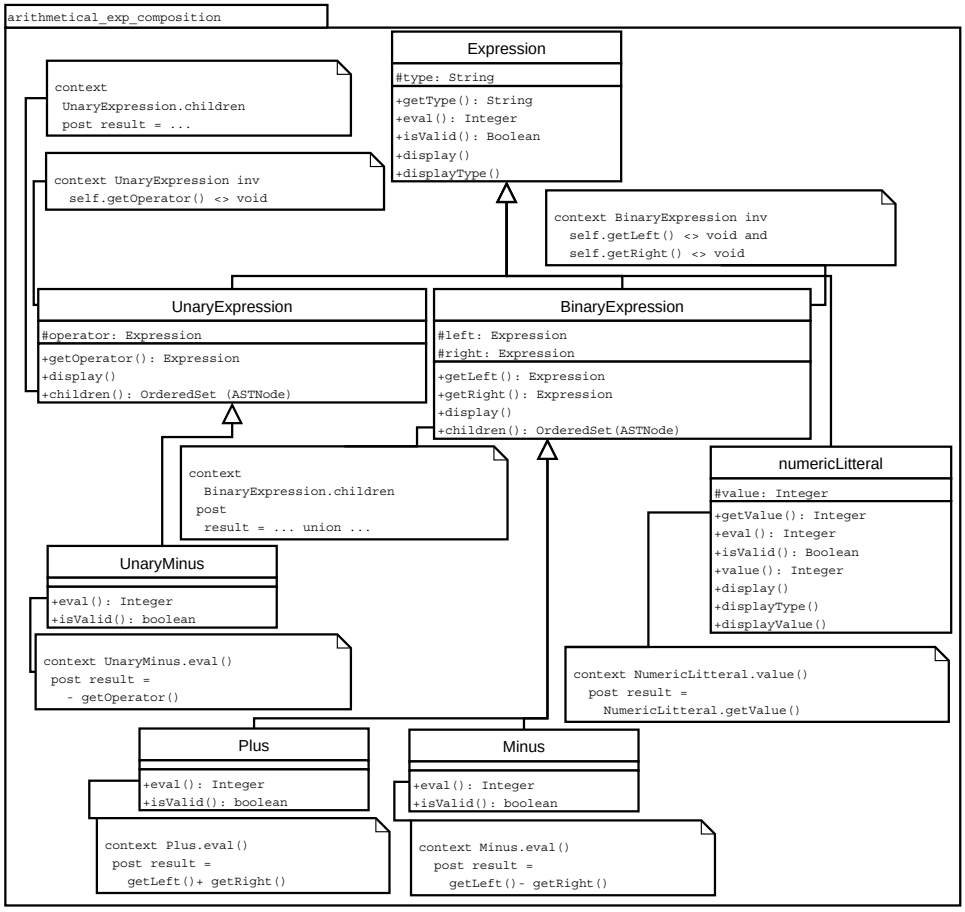

Figure 12. *Résultat de la deuxième composition*

vaux concernant la programmation par séparation des préoccupations et par rapport à l'ingénierie des modèles. Nous avons montré que notre approche s'appuyait sur les travaux relatifs à la programmation par aspects (Kiczales *et al.*, 2001a, Kiczales *et al.*, 2001b) et plus particulièrement par sujets (Ossher *et al.*, 1995, Ossher *et al.*, 2000, Janzen *et al.*, 2004, Tarr *et al.*, 1999) et sur l'approche par adaptateur définie dans (Quintian, 2004, Lahire *et al.*, 2004) qui permet de composer des hiérarchies de classes au sens des langages de programmation. L'originalité de notre approche repose sur la mise en œuvre de mécanismes pour favoriser la description d'un cahier de réutilisation dans le but de guider et contrôler la composition, et sur son ouverture (hiérarchie d'adaptations qui peut être étendue). Il est aussi intéressant de se positionner par rapport à l'ingénierie des modèles car notre approche par composition *ex situ*

ou *in situ* de modèle permet de compléter la problématique visée par les langages de transformation de modèles (Fabro *et al.*, 2005). Il sera en particulier intéressant d'étudier comment augmenter l'expressivité du langage dédié à la description des adaptateurs pour se rapprocher des langages de transformation comme ATL ou MTL (Projet Atlas, INRIA, 2005, Projet Triskell, INRIA, 2005). Par rapport à une approche par composants il est intéressant de noter (et il faudra développer notre approche en ce sens), qu'un modèle peut être vu comme un composant dont les interfaces requises sont exprimées par les primitives abstraites de chacune des feuilles des hiérarchies ce qui se rapproche des idées développées dans (Muller *et al.*, 2003).

À propos des préoccupations fonctionnelles, il est particulièrement intéressant de pouvoir décrire les comportements associés et de pouvoir les composer. Contrairement aux langages à objets, la description n'est pas réalisée dans la classe elle-même ou parfois tout simplement impossible. Ainsi dans UML, elle est réalisée à l'extérieur du diagramme des classes, dans les diagrammes de collaboration, d'activités, de séquences (Rumbaugh *et al.*, 2004). Dans EMF (Eclipse fundation, 2004), l'aspect "comportement" est inexistant ; il est seulement possible de décrire ce comportement, comme c'est aussi le cas dans UML, par l'intermédiaire d'assertions (Vanwormhoudt, 2005) et d'annotations. Il serait donc utile de proposer une approche pour décrire ces comportements qui puissent s'intégrer avec l'approche que nous proposons pour la composition. Nous pourrons nous appuyer pour cela sur les travaux de l'équipe Triskell relatifs aux lignes de produits (Ziadi, 2004) et à Action Semantics (Sunyé *et al.*, 2001).

6. Conclusion et perspectives

L'approche qui est proposée dans cet article représente une première tentative pour expliquer, contrôler et réaliser la séparation et la composition de modèles. Nous défendons l'idée qu'il est intéressant de pouvoir introduire une séparation des préoccupations dès la phase de conception et d'adopter une approche MDA pour raffiner les modèles à composer et produire un modèle final exécutable.

Pour cela, il est nécessaire d'approfondir la description de chacune des adaptations et des contraintes décrites dans les adaptateurs et le langage dédié associé. En particulier il sera intéressant de voir comment nous pourrions nous appuyer sur (Vanwormhoudt, 2005) pour généraliser le mécanisme de contraintes associé aux adaptateurs (cibles d'adaptation et adaptations abstraites). De même, pour améliorer la capacité de réutilisation il serait intéressant de proposer non pas un mais plusieurs protocoles de composition pour un modèle. Ils représenteraient différentes alternatives (garantissant pour chacune l'intégrité de la composition) ; ce serait à la personne qui compose les modèles de choisir le protocole qui correspond le mieux aux modèles impliqués. Pour diminuer le travail nécessaire à l'introduction de ces alternatives il serait intéressant de proposer un héritage n'impliquant pas le polymorphisme (héritage "privé" en C++ ou "non conforme" en Eiffel).

Du point de vue de l'implémentation, nous sommes en train de réécrire le prototype proposé par (Quintian, 2004) afin de pouvoir offrir une plate-forme commune permettant de considérer à la fois la composition des classes d'un programme (Java dans notre cas) et des classificateurs décrits dans un modèle EMF dont les caractéristiques sont proches de celles du MOF (OMG, 2001). Choisir EMF permet de disposer d'une plate-forme opérationnelle (Eclipse) et de nombreuses extensions intéressantes comme par exemple la mise en œuvre d'OCL (Vanwormhoudt, 2005).

7. Bibliographie

Eclipse fundation, « Eclipse Environment », http ://www.eclipse.org, 2004.

Estublier J., Ionita A. D., « Extending UML for model composition », *Proceedings of the Australian Software Engineering Conference (ASWEC)*, Brisbane, Australia, p. 8, march, 2005.

Fabro M. D. D., Bézivin J., Jouault F., Breton E., Gueltas G., « AMW : A generic Model Weaver », *Actes de la 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, Paris, France, p. 10, Juin, 2005.

Janzen D., Volder K. D., « Programming with Crosscutting Effective Views. », *in* , M. Odersky (ed.), *ECOOP 2004 : proceedings of the Object-Oriented Programming, 18th European Conference*, vol. 3086 of *Lecture Notes in Computer Science*, Springer, p. 195-218, 2004.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., « An Overview of AspectJ », *in* , J. L. Knudsen (ed.), *Proceedings of ECOOP'01*, LNCS(2072), Springer Verlag, Budapest, Hungaria, June, 2001a.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W., « Getting started with ASPECTJ », *Communications of the ACM*, vol. 44, n° 10, p. 59-65, October, 2001b.

Lahire P., Quintian L., New Perspective To Improve Reusability in Object-Oriented Languages, To appear in Journal of Object Technology (JOT) in january 2006, Sophia-Antipolis, France, September, 2004. pages 20.

Muller A., Caron O., Carré B., Vanwormhoudt G., « Réutilisation d'aspects fonctionnels : des vues aux composants », *Actes de la conférence Langages, Modèles et Objets, LMO 2003*, Hermés Sciences, Vannes, France, p. 241-255, janvier, 2003.

OMG, *Meta Object Facility Specification (MOF)*, Object Management Group. November, 2001, Version 1.3.1.

Ossher H., Kaplan M., Harrison W., Katz A., Kruskal V., « Subject-Oriented Composition Rules », *in* , R. J. Wirfs-Brock (ed.), *ACM SIGPLAN Notices*, vol. 30(10), ACM Press, Austin, Texas, USA, October, 1995.

Ossher H., Tarr P., « Hyper/J : Multi-Dimentionnal Separation of Concern for Java », *in* , C. Ghezzy (ed.), *Proceedings of ICSE'00*, ACM Press, Limerick, Ireland, June, 2000.

Projet Atlas, INRIA, « ATL : Atlas Transformation Language », http ://www.sciences.univ-nantes.fr/lina/atl/, 2005.

Projet Triskell, INRIA, « MTL : Model Transformation Language », http ://model-ware.inria.fr/rubrique8.html, 2005.

Quintian L., JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet, Thèse de doctorat, Université de Nice-Sophia Antipolis, France, juillet, 2004.

Rumbaugh J., Jacobson I., Booch G., *UML 2.0 Guide de référence*, CampusPress, Paris, décembre, 2004.

Sunyé G., Pennaneac'h F., Ho W.-M., Le Guennec A., Jézéquel J.-M., « Using UML Action Semantics for Executable Modeling and Beyond », *proceedings of CAISE'01*, LNCS(2068), Springer-Verlag, p. 433-447, june, 2001.

Tarr P., Ossher H., Harrison W., Stanley M. Sutton J., « N degrees of separation : multi-dimensional separation of concerns », *ICSE '99 : Proceedings of the 21st international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, p. 107-119, 1999.

Vanwormhoudt G., « Précision et Validation de Métamodèles avec EMF et OCL », *in* , P. Collet, , P. Lahire (eds), *Actes des journée Objects Composants et Modèles (OCM) du GDR ALP*, Berne, p. 19-28, Mars, 2005.

Ziadi T., Manipulation de lignes de produits en UML, Thèse de doctorat, Université de Rennes I, Rennes, France, décembre, 2004.

ANNEXE POUR LE SERVICE FABRICATION

A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :
   JFDLPA 2005

2. AUTEURS :
   Pierre Crescenzo et Philippe Lahire

3. TITRE DE L'ARTICLE :
   Une approche pour améliorer la réutilisabilité des modèles métiers

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Réutilisabilité des modèles métiers

5. DATE DE CETTE VERSION :
   2 septembre 2005

6. COORDONNÉES DES AUTEURS :

– adresse postale :

Laboratoire I3S (CNRS - UNS)
Projet OCL
2000 route des Lucioles
Les Algorithmes, Bâtiment Euclide
BP 121
F-06903 Sophia-Antipolis cedex
France

Pierre.Crescenzo@unice.fr et Philippe.Lahire@unice.fr

– téléphone : 04 92 94 27 42 et 04 92 94 27 51

– télécopie : 04 92 94 28 96

– e-mail : Pierre.Crescenzo@unice.fr et Philippe.Lahire@unice.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com