Gestion des objets persistants grâce aux liens entre classes

Adeline Capouillez, Robert Chignoli, Pierre Crescenzo, Philippe Lahire

Gestion des objets persistants grâce aux liens entre classes

Adeline Capouillez, Robert Chignoli, Pierre Crescenzo et Philippe Lahire[1]

1. Introduction

Le but de notre étude est d'améliorer le partage d'objets persistants entre différentes applications. Pour cela, nous souhaitons diminuer la dépendance de ces objets à la structure du schéma de classes. Le moyen que nous utilisons pour atteindre cet objectif est de profiter des informations sémantiques données par les liens d'importation entre classes. Ces informations nous permettent, en effet, de relâcher sans pour autant les briser, le temps de l'exécution d'une application, les liens d'instanciation.

Le fait que plusieurs applications accèdent aux mêmes objets persistants laisse apparaître deux cas de figure :

- **Schéma de classes partiel** Certaines applications peuvent n'avoir qu'une connaissance partielle du schéma de classes persistant. Les instances des classes connues sont évidemment directement accessibles. Cependant, il peut être également souhaitable de charger d'autres objets persistants qui *peuvent être vus comme* des instances de classes connues.
- **Évolution de classes** Les classes d'une application peuvent évoluer. Les instances persistantes sauvegardées par les anciennes versions de ces classes doivent alors pouvoir être chargées, utilisées voire traduites pour s'adapter aux nouvelles versions.

Le second cas de figure, lié à l'évolution de classe, sera traité par la gestion de liens de version spécialisés tels ceux du système `Presage` [T94]. Cette partie de notre travail fera donc l'objet d'une étude future. Pour nous rapprocher de cet objectif, nous avons étudié les liens dérivés de l'héritage tels ceux de réutilisation de code [CCL99a], de spécialisation et de généralisation [CCCL2000] en présentant pour chacun d'eux, l'ensemble des conditions nécessaires à l'établissement du lien. Nous avons ensuite étudié les phases de chargement et de mise à jour en détaillant les différentes situations qui en découlent. Pour chacune d'elles, nous avons donné les contraintes qui se posent et les opérations à réaliser.

Avant d'étendre cette étude aux liens de version, nous souhaitons d'abord présenter ici l'intégration des liens d'utilisation à notre système et montrer quels apports nous pouvons en tirer. Pour cela, nous avons choisi, ici, de présenter seulement le lien d'utilisation le plus couramment utilisé dans les langages à objets : la relation qui lie une classe avec les classes de ses attributs. Au niveau programmation, ce lien ressemble à celui de clientèle au sens d'`Eiffel` [M92]. Au niveau modélisation, ce lien peut, par exemple, être représenté en `UML` [BJR98, RJB98, JBR99] par une agrégation, une dépendance ou encore un simple attribut. Nous avons d'ailleurs choisi d'appeler cette relation *lien d'agrégation*. Ce document présente tout d'abord le contexte dans lequel nous plaçons notre étude, puis l'exemple d'un lien d'agrégation en donnant quelques détails sur les phases de chargement et de mise à jour d'objets persistants en présence de ce lien. Nous effectuons ensuite une généralisation de notre approche pour finir par expliciter nos perspectives concernant ce travail.

[1] Pour tous les auteurs : Laboratoire I3S (UNSA/CNRS), Projet OCL, Sophia Antipolis, France. *E-Mails* : {Adeline.Capouillez | Robert.Chignoli | Pierre.Crescenzo | Philippe.Lahire}@unice.fr

2. Contexte de ce travail

Le modèle OFL [CCL99b, CCL99c], qui sert de base à notre étude, est défini dans le but de mettre en exergue la notion de *lien entre classes* dans les langages à objets (tels Java [GJS96, AG98, F99], Eiffel et C++ [S97]). OFL est conçu dans le cadre du génie logiciel [O99]. Il décrit, pour chaque langage, une entité *concept-langage* assurant la gestion d'un ou plusieurs *concepts-descriptions* qui représentent les différentes *sortes* de classe (en Java, on trouve par exemple les classes, les interfaces, les tableaux, …). Chaque concept-description peut être considéré comme la *source* ou la *cible* d'un lien (décrit par un *concept-lien*) tel l'héritage ou l'agrégation.

Nous devons également préciser que nous nous plaçons dans le cadre d'un langage de programmation (et non d'un SGBD) pour lequel le chargement de classe du schéma persistant ne se fait pas implicitement[2]. Cela signifie que le chargement d'une instance n'entraîne pas le chargement de sa classe. Notre démarche est en effet de charger l'instance en l'*adaptant* au schéma volatile, c'est-à-dire au schéma de l'application. Nous souhaitons ainsi prendre en compte certaines configurations, liées aux problèmes de l'évolution des logiciels, dans lesquelles toutes les applications n'ont pas forcément suivi pas à pas l'évolution d'un schéma de classes persistant.

Nous admettons qu'il est également possible de réaliser le chargement d'une vision aplatie[3] de classe. Nous ne faisons aucune hypothèse sur le fait que le chargement est plus ou moins statique ou dynamique.

Nous envisageons d'utiliser le service ROOPS [C99] qui offre une modélisation persistante des entités OFL. ROOPS est conçu pour permettre le stockage, outre des instances et des classes, de toutes les informations concernant les liens entre classes[4] dans un objectif de contrôle et de précision de l'information persistante. La représentation persistante des classes et des liens est donc réalisée ici pour améliorer l'utilisation des instances persistantes[5] et non pour permettre le chargement dynamique des classes et des liens même si cela est tout à fait envisageable dans un autre contexte.

3. Présentation de l'approche

Notre approche s'appuie sur la notion d'*hyper-généricité* [D94]. La première forme de paramétrage apparue en informatique concernait les valeurs, il s'agissait simplement de permettre la modification de la valeur d'une entité. Vint ensuite le paramétrage sur les types, qui a donné comme principal résultat la notion de *généricité* que l'on retrouve par exemple dans les classes génériques d'Eiffel et, d'une manière plus rustique, dans les *templates* de C++. Nous proposons, pour notre part, d'offrir des paramètres sur les liens entre classes : nous parlons donc d'hyper-généricité.

Les liens d'importation ont pour objectif la structuration de l'application. De ce fait, leur sémantique est plutôt indépendante du contexte du logiciel (système d'information, CAO, calcul scientifique, …) et des classes de problèmes tels la persistance, la mobilité, le parallélisme ou la distribution. Les liens d'utilisation, pour leur part, reflètent, comme leur nom l'indique, l'utilisation de composants par d'autres. Leur sémantique peut donc dépendre assez fortement du contexte et des classes de problèmes pris en compte.

[2] Le chargement explicite de classe est évidemment possible.
[3] Par *aplatie*, nous entendons qu'une fermeture transitive est réalisée sur la classe et que toutes ses primitives sont alors vues comme locales.
[4] Les liens entre classes et objets, comme l'instanciation, ou entre objets sont également modélisés par OFL et ROOPS mais n'entrent pas dans le cadre de ce document.
[5] Une amélioration peut en effet être envisagée puisque l'information structurelle présente est plus précise.

Pour généraliser l'usage de liens bien adaptés au contexte et bénéficier des informations méta pour améliorer des services comme la persistance, nous proposons dans `OFL`, un recensement des *paramètres généraux* qui vont influer sur la définition du lien (hyper-généricité). L'ensemble des valeurs données à ces paramètres définit donc la sémantique du lien. Créer un nouveau lien avec une nouvelle sémantique consiste ainsi à valuer ses paramètres.

Nous répertorions également les opérateurs sémantiques, que nous nommons *actions*, dont l'exécution peut être influencée par ces paramètres. Ces actions sont des contrôles ou des algorithmes qui doivent être mis en œuvre à la compilation ou lors de l'exécution du code. Par exemple, on trouve parmi les actions le *`lookup`* (recherche de la primitive adéquate dans le graphe de typage), l'*`assign`* (affectation d'une valeur à une variable), etc.

Le méta-programmeur doit donc pouvoir intervenir à au moins deux niveaux : ajustement des paramètres (si la sémantique de ceux-ci lui convient) ou, en dernier recours, modification du code des actions (si le comportement qu'il veut décrire n'est pas paramétré ou s'il souhaite modifier la sémantique d'un paramètre existant).

Deux approches sont possibles pour définir la sémantique des liens d'utilisation en accord avec les contraintes associées au contexte et aux classes de problèmes :

1. S'en tenir aux paramètres généraux (ceux qui sont communs à presque toutes sortes de lien d'utilisation) mais, à chaque fois que c'est utile, intégrer la sémantique propre au lien en modifiant le code des actions `OFL`.
2. Enrichir les paramètres généraux par des paramètres spécifiques à chaque concept-lien en définissant, pour chaque contexte d'application et pour chaque classe de problèmes, un ensemble de paramètres pertinents qui, comme les paramètres généraux, vont piloter l'exécution des actions `OFL`. Cela revient à ne modifier les actions que lorsque l'on intègre un nouveau type d'application ou une nouvelle classe de problèmes, au prix d'un effort de modélisation (la mise en évidence des paramètres) plus important.

Il nous semble que la seconde approche est plus novatrice et intéressante car le lourd travail de méta-programmation que l'on retrouve généralement dans les systèmes qui disposent d'une couche méta est ici localisé et mis en œuvre uniquement lorsque cela s'avère nécessaire. Tout le reste du travail consiste à ajuster des paramètres.

Pour faire le parallèle avec les liens d'importation (dont l'héritage est le fer de lance), nous pouvons dire que, pour eux, le problème est moins crucial. En effet, de part leurs sémantiques moins diversifiées (la notion d'importation est plus restrictive que celle d'utilisation), les paramètres généraux devraient suffire dans la quasi-totalité des cas.

Par la valuation des paramètres généraux, nous souhaitons permettre la génération de nouveaux liens entre classes. Ainsi, nous espérons pouvoir améliorer la maintenabilité des logiciels autant par l'apport de liens de version dont nous parlions dans l'introduction que par tous les bénéfices que l'on peut tirer d'une spécification plus précise : documentation plus explicite, volontés du programmeur mieux exprimées, contrôles mieux ciblés et plus pertinents à la compilation/interprétation et à l'exécution, … Nous pouvons également, par la valuation de paramètres plus spécifiques, adapter nos liens à un contexte (exemple : systèmes d'information) ou à une classe de problèmes (exemple : persistance).

4. Exemple de lien d'utilisation : l'agrégation

On définit un lien d'agrégation comme une relation entre une classe-source et une classe-cible[6]. On retrouve ce lien dans la quasi-totalité des langages à objets. En effet, il est implicitement posé lorsqu'une classe (dite ici classe-source) définit un attribut, un paramètre de méthode, un retour de fonction ou une variable locale de méthode[7] (dont la classe est donc ici la classe-cible) : on peut dire que la classe-source *utilise les services de* la classe-cible.

La seule condition pour pouvoir établir un tel lien est donc que la classe-source définisse un attribut dont le type est décrit par la classe-cible.

Voici deux exemples d'agrégation :

1. La classe *AUTOMOBILE* (classe-source) établit un lien d'agrégation vers la classe *PERSONNE* (classe-cible) pour référencer le propriétaire. Autrement dit, la classe *AUTOMOBILE* possède un attribut *propriétaire* de type *PERSONNE*.
2. La classe *PERSONNE* établit un lien d'agrégation vers elle-même pour indiquer l'éventuel conjoint.

En OFL, le lien d'agrégation est défini par la valuation des paramètres de concept-lien. Voici, pour ce lien, quelques paramètres avec leur valeur (illustration de l'hyper-généricité) :

- ✓ `name : aggregation` (lien d'agrégation)
- ✓ `kind : inter-type-use` (lien d'utilisation entre types)
- ✓ `cardinality : 1-∞` (lien multiple : reliant *1* source à *n* cibles)
- ✓ `circularity : true` (circularité autorisée pour ce lien)
- ✓ `repetition : true` (le lien étant multiple, une source peut avoir plusieurs fois la même cible)
- ✓ `symmetry : false` (la sémantique du lien n'est pas symétrique : l'agrégation est un lien orienté)
- ✓ `opposite : none` (nous ne disposons pas ici d'un concept-lien inverse)
- ✓ …

5. Agrégation et persistance

Dans un environnement où les classes sont reliées par des liens plus spécifiques que l'héritage, nous avons expliqué [CCCL2000] qu'il était possible de mieux exploiter les objets persistants dans des applications écrites à différents moments du cycle de vie d'un schéma de classes. En particulier, nous avons montré que, grâce à la présence d'un lien spécifique, tel la généralisation ou la spécialisation, entre une classe *AUTOMOBILE* et une classe *AUTOMOBILE_DIESEL*, il est possible, à une application qui possède seulement l'une ou l'autre de ces classes, de charger et, dans certains cas, de mettre à jour des objets persistants qu'elle n'aurait pas pu utiliser sinon.

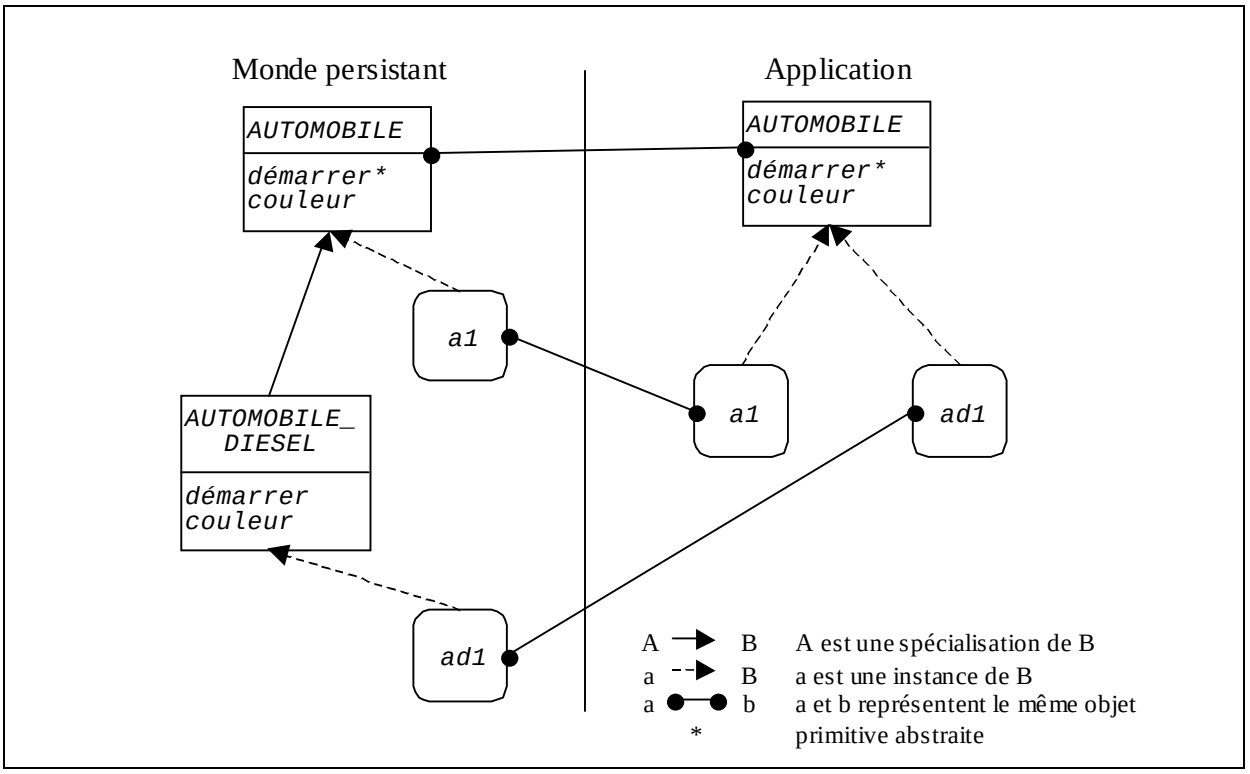
Nous présentons les informations utiles pour décrire un lien d'agrégation. Elles ne sont pas spécifiques à un tel lien mais généralisable à tout lien d'utilisation. De plus, nous montrons qu'elles sont notamment utiles lors des phases de chargement et de mise à jour de données persistantes. Enfin, nous mettons au jour l'intérêt d'un tel lien lors de l'exécution des méthodes.

L'exemple qui nous sert de support est présenté dans la figure suivante. Nous souhaitons charger, dans une application qui a connaissance de la classe *AUTOMOBILE* (mais pas des classes *AUTOMOBILE_DIESEL* et *AUTOMOBILE_ESSENCE* qui sont des spécialisations d'*AUTOMOBILE*

[6] Il est bien évidemment possible de considérer un lien d'agrégation multiple mais en rester à un lien simple ne nuit pas à la généralité.
[7] Désormais, nous simplifierons en ne parlant plus que d'attribut.

dans le monde persistant), toutes les automobiles du monde persistant. Nous souhaitons donc charger les objets persistants *a1* (instance directe d'*AUTOMOBILE*, *a1* est une automobile abstraite) et *ad1* (instance indirecte d'*AUTOMOBILE*).

La classe *AUTOMOBILE* est semi-concrète puisqu'elle possède une méthode abstraite *démarrer* ainsi qu'une méthode concrète *couleur* (qui permet de spécifier la couleur de l'automobile).



Une fois les objets persistants *a1* et *ad1* chargés au sein de l'application, nous souhaitons les traiter comme des automobiles, c'est-à-dire leur appliquer des méthodes d'*AUTOMOBILE*. À la fin de l'utilisation de ces objets, nous souhaitons les mettre à jour en mémoire persistante. Nous présentons donc, dans la suite, la phase de chargement de ces objets, un exemple d'exécution de méthode appliquée aux copies volatiles de ces objets, et enfin, la phase de mise à jour vers le monde persistant.

5.1. Phase de chargement

Voyons ici ce à quoi il faut prendre garde durant la phase de chargement des objets et ce dont nous pouvons tirer parti si nous avons connaissance d'un lien d'agrégation. Nous ne tenons compte ici que de la nature du lien (concept-lien) présent – ici agrégation – mais rappelons que cette nature est définie par la valuation des paramètres de concept-lien. Imaginons donc que nous souhaitons charger l'objet *ad1* de la classe *AUTOMOBILE_DIESEL*. Le déroulement des opérations pourrait être le suivant :

1. Si *ad1* est déjà chargé en tant qu'instance d'*AUTOMOBILE*, alors utiliser la copie volatile actuelle.
2. Si *ad1* est déjà chargé en tant qu'instance d'une classe *C* différente d'*AUTOMOBILE*, alors :
   a. Si *C* est une spécialisation d'*AUTOMOBILE*, alors utiliser la copie volatile actuelle. En effet, cette copie *est une* *AUTOMOBILE* grâce au polymorphisme.
   b. Sinon, si *C* est une généralisation d'*AUTOMOBILE*, alors faire migrer la copie volatile vers *AUTOMOBILE* et l'utiliser (cette migration est très simple à réaliser car la version

persistante d'*ad1* est une *AUTOMOBILE_DIESEL* et donc, *a fortiori*, une *AUTOMOBILE*).

c. Sinon, notifier à l'application qu'*ad1* est déjà chargé sous une forme potentiellement incompatible. La procédure de chargement de cet objet est abandonnée.

3. Si *ad1* n'est pas déjà chargé, le charger en tant qu'instance d'*AUTOMOBILE*.
4. Pour chaque classe-cible *Cible* d'une agrégation dont *AUTOMOBILE* est classe-source, choisir une stratégie de chargement [C99]. Exemples : « on charge d'ores et déjà toutes les instances de *Cible* référencées par *ad1* » ou « on ne charge *a priori* aucune instance de *Cible* référencée par *ad1* » ou « on charge seulement les instances de *Cible* référencée par *ad1* qui sont utiles pour l'évaluation des invariants[8] de *ad1* » ou …[9]

De cet exemple, nous pouvons déduire un certain nombre de paramètres spécifiques à la gestion des objets persistants (classe de problème *persistance*) :

- ✓ moment du chargement de l'objet (à l'utilisation de l'objet / au début de l'exécution d'une primitive / au début de l'application / …),
- ✓ contrôles au chargement (la classe existe-t-elle ? / une classe *compatible* existe-t-elle ? / l'objet dispose-t-il d'un *adaptateur* [CCCL2000] ? / …),
- ✓ comportement en cas d'échec aux contrôles de chargement (annulation de la transaction / déclenchement d'une exception logicielle / arrêt de l'application / …),
- ✓ autorisation de migration d'objets persistants (peut-on briser le lien d'instanciation pour en créer un autre dans le but de changer la classe d'un objet [CCCL2000] ? : oui / non / dans quelles conditions ?),
- ✓ …

Vu sous l'angle du lien d'importation (spécialisation) entre *AUTOMOBILE* et *AUTOMOBILE_DIESEL*, le chargement doit aussi respecter les étapes suivantes (rappelons que nous chargeons, vers une classe *AUTOMOBILE* connue, une automobile diesel *ad1*, instance directe de la classe *AUTOMOBILE_DIESEL* inconnue de l'application) :

1. récupérer les informations persistantes relatives au lien, entre autres : son concept-lien (ici, nous désirons savoir que nous avons affaire à un lien de spécialisation) et ses paramètres (par exemple sa non-circularité),
2. éventuellement, vérifier si la classe *AUTOMOBILE* est concrète (toutes les méthodes sont implémentées), abstraite (aucune méthode n'est implémentée) ou semi-concrète (il existe au moins une méthode non implémentée et une méthode implémentée). Cela peut être par exemple utile si l'on ne souhaite charger que des objets concrets.
3. masquer dans *ad1* les valeurs des attributs propres à *AUTOMOBILE_DIESEL*.

5.2. Phase d'exécution d'une méthode

Désormais nous avons, dans l'application, l'ensemble des automobiles présentes dans le monde persistant. Nous voulons effectuer un travail sur ces objets. Nous voyons que tant que nous utilisons des méthodes concrètes, nous pouvons travailler sur ces instances, même si leur type est semi-concret. Nous pouvons donc changer la couleur de tous ces objets, mais nous ne pouvons pas les faire démarrer. Ceci est une originalité par rapport aux langages à objets à classes actuels. En effet, pour ces derniers, une classe est abstraite et ne peut exécuter aucune primitive dès lors qu'une seule de ces primitives est abstraite.

[8] Dans OFL, les classes sont composées d'attributs, de méthodes et d'un invariant (formule logique qui doit être vérifiée à tout moment) [M92b].

[9] Le choix de la stratégie de chargement dépend en partie de la *sorte* de lien d'utilisation. On peut imaginer ce fait en explorant par ailleurs un lien de *composition* qui indique une sémantique d'utilisation plus forte que l'agrégation (exemple : une brique fait partie intégrante d'un mur et n'est pas identifiable en dehors de celui-ci ; si le mur est détruit, la brique aussi).

Voici les étapes possibles de l'exécution d'une méthode de la classe *AUTOMOBILE* :

1. déclenchement (éventuel) d'une transaction ; ce déclenchement dépend de la granularité de la transaction : on peut, par exemple, déclencher une nouvelle transaction à chaque fois qu'un objet volatile demande à une copie volatile d'un objet persistant l'exécution d'une méthode [L92],
2. vérification que la méthode est concrète (si ce n'est pas le cas alors l'exécution est refusée, la transaction en cours est annulée et une exception *méthode_abstraite* est générée),
3. vérification des préconditions[10] de la méthode,
4. exécution du corps de la méthode,
5. vérification des postconditions de la méthode, et
6. fermeture de la transaction (si elle a été ouverte).

Il est évident que durant les phases 3 à 5 il peut être nécessaire d'effectuer le chargement d'objets. Soit leur classe est connue de l'application et le chargement est direct, soit elle est inconnue et nous utilisons alors le protocole présenté dans la section 5.1.

Nous proposons de nouveaux paramètres généraux associés à cette phase :

- ✓ attribut lisible directement ? (ou passage par un accesseur)
- ✓ attribut modifiable directement ?
- ✓ politique de gestion mémoire (allocation et libération : manuelles / automatiques / mixtes / …),
- ✓ …

puis des paramètres plus spécifiques au traitement de la persistance :

- ✓ moment de début de transaction (au début de l'exécution d'une primitive / au début de l'application / …),
- ✓ moment de fin de transaction (à la fin de l'exécution d'une primitive / à la fin de l'application / …),
- ✓ …

5.3. Phase de mise à jour

À la fin de l'application ou lorsque la copie volatile d'*ad1* n'est plus utile ou encore lorsque l'application le décide, il faut effectuer l'opération qui consiste à mettre la représentation persistante d'*ad1* en adéquation avec cette copie volatile : c'est l'opération de mise à jour. Cette opération est évidemment aussi effectuée sur *a1*, mais cela ne présente alors aucune difficulté. Voici un scénario possible pour réaliser cette étape sur *ad1* :

1. Si aucun attribut de la copie volatile d'*ad1* n'a été modifié depuis son chargement, libérer simplement cette copie.
2. Sinon, vérifier que la copie volatile d'*ad1* respecte l'invariant d'*AUTOMOBILE_DIESEL*.
3. Si c'est le cas, pour chaque classe-source *Source* d'une agrégation dont *AUTOMOBILE_DIESEL* est classe-cible, vérifier que les invariants des instances persistantes (les volatiles ayant été contrôlées au fur et à mesure de l'exécution) de *Source* qui référencent *ad1* sont toujours valides (par rapport à la copie volatile d'*ad1*).
4. Si c'est le cas, mettre la représentation persistante d'*ad1* à jour par rapport à sa copie volatile puis libérer cette dernière.
5. Sinon, choisir une stratégie de mise à jour. Exemples : « interdire la mise à jour d'*ad1* » ou « notifier le problème à l'application qui tente la mise à jour d'*ad1* pour demander un

[10] Dans OFL, les méthodes sont équipées de préconditions (formule logique qui doit être vérifiée avant l'exécution de la méthode) et de postconditions (formule logique qui doit être vérifiée après l'exécution de la méthode) [M92b].

maintien de la cohérence » ou « permettre la mise à jour et notifier l'incohérence aux instances de *Source* qui référencent *ad1* » ou …

Nous présentons ci-dessous des paramètres qui traitent spécifiquement du problème de la persistance :

- ✓ génération d'adaptateur (peut-on *relâcher* le lien d'instanciation pour pouvoir utiliser l'objet avec une classe *proche* de la sienne [CCCL2000] ?[11] : oui / non / dans quelles conditions ?),
- ✓ contrôles à la mise à jour,
- ✓ comportement en cas d'échec aux contrôles de mise à jour,
- ✓ …

6. Travaux Connexes

Cette section a pour objectif de situer notre approche par rapport aux travaux relatifs à la description de liens, dans les domaines suivants : méta-modèles, méthodes de conception, systèmes ouverts méta-programmables et patrons de conception. Notons que les méta-modèles et les patrons de conception sont des travaux connexes à notre démarche méta bien qu'ils ne traitent pas explicitement de la persistance.

6.1. Méta-modèles

MOF dont les initiales correspondent à *Meta-Object Facility* a été spécifié par l'OMG pour permettre la gestion des informations méta qui sont nécessaires à un système CORBA [OMG97]. La généralité de l'ensemble de concepts fourni permet d'utiliser MOF pour la description d'autres systèmes de types (exemples : UML, C++, etc.). Parmi ces concepts, on trouve entre autres celui de *class* qui correspond à la description d'un concept dans un système de types (par exemple, une interface ou un constructeur Java), et celui d'*association* qui permet de modéliser les liens qui existent entre les concepts d'un système de types. Chaque concept se voit associer un certain nombre de caractéristiques (exemples : « est-ce qu'un concept est abstrait » ou bien « quelle est la cardinalité d'une association »). Décrire un système de types revient donc à créer des instances de concepts MOF et à utiliser leurs primitives associées, au moyen du langage de définition proposé (*Meta-Object Definition Language*). Un de nos objectifs à court terme est de décrire notre méta-modèle à l'aide de MOF et de bénéficier ainsi des différents outils associés.

6.2. Systèmes ouverts centrés sur les liens

Le système FLO [DDM95, DDM96] est basé sur le langage Lisp, il permet de définir des dépendances et des contraintes sémantiques entre les instances de classe qui sont définies par des liens décrits en dehors de la classe. La description de ces liens s'appuie sur des opérateurs comme *permitted-if* (proche des notions de préconditions ou d'invariants) et *implies* (actions réalisées par le lien en s'appuyant sur les primitives des classes intervenant dans le lien). Un autre aspect intéressant de ce travail est la présence d'un opérateur supplémentaire *corresponds* qui permet de différencier le traitement réalisé sur les différentes instances d'une même classe. FLO a été plus particulièrement utilisé dans des domaines comme la représentation de connaissances ou la construction d'interfaces graphiques et une intégration des problèmes liés à la persistance a été réalisée

[11] Il se peut que la copie volatile représentant l'objet persistant ait une forme différente de ce dernier (par exemple, elle est d'un type plus spécialisé). Dans ce cas, lors de la mise à jour, nous envisageons, dans [CCCL2000], d'associer à l'objet persistant un adaptateur, pour ne pas perdre les données supplémentaires.

[BFDJ97]. Notre approche est un peu différente puisqu'elle s'appuie sur le fait que « toute relation entre objets est décrite par un lien entre classes ».

6.3. Méthodes de conception

Les méthodes de conception doivent permettre la définition de liens sémantiques entre les entités. L'approche proposée par `MECANO` [GIR91] propose à l'utilisateur non pas de définir un lien d'héritage entre deux classes mais plutôt de spécifier les usages qu'il compte faire de l'héritage (spécialisation, implémentation, fusion, adaptation, etc.) ou de la délégation (communication, utilisation, etc.) ; cependant la liste des types de relation entre entités, comme celle des types d'entités sont figées dans la méthode.

6.4. Patrons de conception

Les patrons de conception [GHJV95] peuvent aussi être considérés comme une approche pour décrire les liens entre classes. En effet, un des effets bénéfiques de cette approche serait d'utiliser des combinaisons d'utilisation des mécanismes d'héritage et d'agrégation associées à la réalisation de classes *ad hoc* pour décrire des liens plus spécifiques ; l'introduction d'un niveau méta [RF2000] pour le contrôle et la gestion des patrons de conception pourrait fournir des facilités supplémentaires pour une mise en œuvre des liens.

7. Perspectives et conclusion

Ce document résume nos travaux en matière de paramétrage du concept de lien entre classes en s'appuyant sur le lien d'agrégation. Cette démarche s'inscrit dans un projet de plus grande envergure dont le but principal est l'augmentation de la qualité et de la maintenabilité des logiciels par une meilleure spécification des liens entre classes.

Nous nous intéressons désormais principalement aux axes de développement suivants :

- comme nous l'avons présenté au début, la généralisation de cette approche aux liens de version, permettant de prendre en compte l'évolution des applications,
- la composition des concepts-liens (dans les concepts-descriptions) et de liens (dans les concepts-liens),
- la description de notre modèle avec `MOF`, `XML` ou `MODL`, ce travail ayant déjà été entamé en `XML` [CCCL2000b], et
- la réalisation d'un prototype basé sur l'idée d'un pré-processeur de `Java-étendu` (avec quelques liens spécifiques) vers du `Java` standard.

Références

[AG98] K. Arnold and J. Gosling, "*The Java Programming Language*", Sun Microsystems, 1998.

[BFDJ97] L. Berger, M. Fornarino, A.M. Dery and O. Jautzy, "*First Steps to an Interaction and communication Manager between Remote Objects Computing*", in ECOOP'97 (European Conference on Object Oriented Programming), Workshop Models, Formalisms and Methods for Distributed Object-Oriented Computing, June 1997, Finland.

[BJR98] G. Booch, I. Jacobson and J. Rumbaugh, "*The Unified Modeling Language User Guide*", Addison-Wesley Publishing Co., October 1998.

[C99] A. Capouillez, "*ROOPS : un service paramétrable de persistance pour OFL*", Rapport de Recherche 99-15, septembre 1999.

[CCL99a] R. Chignoli, P. Crescenzo et P. Lahire, "*Un modèle de paramétrage des liens entre classes*", Rapport de Recherche 99-13, août 1999.

[CCL99b] R. Chignoli, P. Crescenzo et P. Lahire, "*OFL : une machine virtuelle objet flexible pour la gestion d'objets persistants et mobiles*", Rapport de Recherche 99-09, mars 1999.

[CCL99c] R. Chignoli, P. Crescenzo and P. Lahire, "*Customization of Links between Classes*", Research Report 99-18, November 1999.

[CCCL2000] A. Capouillez, R. Chignoli, P. Crescenzo and P. Lahire, "*How to Improve Persistent-Object Management using Link Information?*", Research Report 2000-01, January 2000.

[CCCL2000b] A. Capouillez, R. Chignoli, P. Crescenzo and P. Lahire, "*Modeling Hypergeneric Relationships between Types in XML*", Research Report 2000-05, March 2000.

[D94] P. Desfray, "*Object Engineering, the Fourth Dimension*", Addison-Wesley Publishing Co., 1994.

[DDM95] A-M. Dery, S. Ducasse and M. Fornarino, "*A reflective model for first class dependencies*", in OOPSLA'95 (10th Object Oriented Programming Systems Languages and Applications), ACM Press, pages 265-280, Austin, October 1995.

[DDM96] A-M. Dery, S. Ducasse and M. Fornarino, "*Object and dependency oriented programming in FLO*", in ISMIS'96 (9th International Symposium on Methodologies for Intelligent Systems), LNCS, June 1996.

[F99] D. Flanagan, "*Java in a Nutshell: a Desktop Quick Reference*", O'Reilly, December 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "*Design patterns, Elements of reusable object oriented software*", Addison-Wesley, 1995.

[GIR91] X. Girod, "*MECANO : une méthode et un Environnement de Construction d'ApplicatioNs par Objets*", thèse de doctorat de l'Université Joseph Fourier – Grenoble 1 – spécialité informatique, juin 1991.

[GJS96] J. Gosling, B. Joy and G. Steele, "*The Java Language Specification*", Sun Microsystems, 1996.

[JBR99] I. Jacobson, G. Booch and J. Rumbaugh, "*Unified Software Development Process*", Addison-Wesley Publishing Co., January 1999.

[L92] P. Lahire, "*Conception et réalisation d'un modèle de persistance pour le langage Eiffel*", Thèse de Doctorat en Informatique, Université de Nice-Sophia Antipolis, mai 1992.

[M92] B. Meyer, "*Eiffel: The Language*", Prentice Hall, 1992.

[M92b] B. Meyer, "*Applying Design by Contract*", IEEE Computer, October 1992.

[O99] C. Oussalah, "*Génie objet : analyse et conception de l'évolution*", Hermes, septembre 1999.

[OMG97] OMG & al. "*Meta Object Facility (MOF) Specification*", OMG document, revised submission, October 1997.

[RF2000] P. Rapicault et M. Fornarino, "*Instanciation et vérification de patterns de conception : un méta-protocole*", dans LMO'00 (Langages et Modèles à Objets 2000), janvier 2000.

[RJB98] J. Rumbaugh, I. Jacobson and G. Booch, "*The Unified Modeling Language Reference Manual*", Addison-Wesley Publishing Co., December 1998.

[S97] B. Stroustrup, "*The C++ Programming Language*", Addison-Wesley Publishing Co., 1997.

[T94] G. Talens, "*Gestion des objets simples et composites*", Thèse de Doctorat en Génie Informatique, Automatique et Traitement du Signal, Université Montpellier II, février 1994.