Université de Nice-Sophia Antipolis
Laboratoire I3S - Projet OCL
DEA d'Informatique

Rapport de stage
présenté en juin 2003
par Christophe Jalady

Vers une relation d'héritage générique

Responsables de stage :
Philippe Lahire, Pierre Crescenzo et Michel Gautero

Rapporteur : Didier Parigot

Table des matières

Chapitre 1

Introduction

Ce rapport est centré sur l'étude de la relation d'héritage dans les langages à objets. Nous souhaitons étendre cette relation afin de réduire ses limitations en matière de réutilisation et d'adaptation de hiérarchie de classes.

1.1 Cadre et motivations de l'étude

Cette étude se place donc dans le cadre des langages à objets dits *de classes*, dont une définition est donnée dans [GM91] et pour une grande part de ce document elle est dédiée aux langages compilés à typage statique. Ce dernier point souligne notre volonté de nous tourner plutôt vers des solutions qui prennent en compte la fiabilité et la robustesse des applications.

Les langages à objets proposent tous une relation d'héritage entre classes. Cette relation permet une structuration des classes sous forme de hiérarchie, offrant une factorisation des propriétés des objets. Ces hiérarchies permettent de définir des bibliothèques. Ces dernières peuvent regrouper plusieurs hiérarchies et doivent i) pouvoir être réutilisées dans des *contextes* variés, par exemple dans des applications différentes, ou pour construire de nouvelles bibliothèques ii) pouvoir évoluer : une fois réalisée, une bibliothèque doit être maintenue, mais aussi doit pouvoir être étendue pour permettre de résoudre de nouveaux besoins. Dans ce contexte, la disponibilité des sources est généralement admise, facilitant la résolution de ces problèmes par les concepteurs.

Cependant l'utilisation dans différents contextes d'une bibliothèque réalisée par un tiers nécessite aussi de pouvoir l'adapter à ses propres besoins ([HÖL93, HO92, TL94, SAK02, KO00]). En particulier, il pourra s'agir de l'étendre pour une application spécifique; ou encore de lui ajouter des fonctionnalités, ou même de la modifier en partie. Toutes ces adaptations devrons pouvoir se faire par des utilisateurs sans modifier le code source. C'est principalement dans cette direction que sera menée notre étude.

Or la relation d'héritage ne permet que d'étendre une hiérarchie en lui ajoutant de nouvelles classes *feuilles* dérivant d'une ou de plusieurs classes[1] représentant des *noeuds* de la hiérarchie et ne répond ainsi qu'a une partie seulement des besoins.

Un développeur, utilisateur d'une bibliothèque, doit pouvoir, dans une certaine mesure, manipuler les hiérarchies de classes qui composent la bibliothèque. Il doit pouvoir ajouter des entités implémentant de nouvelles fonctionnalités, modifier une implémentation, ou généraliser deux hiérarchies indépendantes.

Ces possibilités sont nécessaires notamment pour l'ajout de code orthogonal. Ce dernier représente la mise en oeuvre de propriétés non spécifiques à une classe particulière (ou à un type particulier) et qui en général concerne un ensemble de classes qui ne sont pas reliées par des relations d'héritage ou d'agregation. Une première approche consiste à développer ces propriétés de manière externe à une

[1] Suivant le type d'héritage possible, simple ou multiple.

application pour ensuite les intégrer[2]. Ce qui demandera un mécanisme fléxible de manipulation des entités de l'application.

D'autres travaux ont été réalisés pour accroître l'expressivité des langages ou, du moins, améliorer leurs capacités à reutiliser l'existant ou à factoriser les comportements. Il s'agit principalement des relations suivantes qui seront décrites plus en détails dans la section 2.

En permettant de factoriser au sein de classe générique un ensemble de propriétés paramétrées par un type, la généricité a permis d'accroître la réutilisabilité du code. Bien qu'elle n'amène aucun pouvoir expressif supplémentaire au langage[3], elle évite une duplication lourde et laborieuse du code.

Les mixins contribuent à un assouplissement de la relation d'héritage. En effet, un mixin ne spécifie pas lors de sa déclaration de cible d'héritage. Il pourra ainsi s'attacher, suivant son utilisation, à différentes super-classes permettant ainsi de spécialiser plusieurs entités sans avoir à impliquer des duplication de code.

Les Meta-Object Protocol, ou MOP, réifient un ensemble de relation et d'entité du langage. Ceci permet d'avoir un plus grand contrôle sur le langage lui-même mais aussi de factoriser des fonctionnalités au sein de métaclasses. Nous montrerons que leur utilisation n'est cependant pas aisée.

La séparation des préoccupations est un paradigme de programmation qui promouvoit des applications modulaires. Celles-ci sont vues comme ayant différentes fonctionnalités qui sont développées en parallèle et ensuite assemblées entres elles. Les différentes *préoccupations* peuvent alors être réutilisées dans d'autres applications. Leur indépendance facilite aussi leur maintenance. L'intégration est cependant peu controlée car elle se base sur une réification du langage d'implémentation, ce qui peut alterer la robustesse d'une application.

Enfin les langages de configuration permettent de modifier le comportement d'une application de manière externe au langage. Ils permettent, dans une certaine mesure, de paramétrer le comportement d'une bibliothèque pour l'adapter à des besoins spécifiques.

1.2 Problématique et plan du rapport

Cette étude portera donc sur la pertinence de l'ajout d'une nouvelle relation, dite relation d'héritage générique, en vue d'améliorer la réutilisabilité, la maintenance et la lisibilité du code d'une application. Nous resterons dans le cadre des langages à objets classiques auxquels nous souhaitons simplement ajouter un type de relation d'héritage.

Nous souhaitons pouvoir modifier la cible[4] d'héritage d'une classe en fonction de son contexte d'utilisation, c'est-à-dire suivant les différentes applications qui l'utilisent ou même suivant les différentes parties d'une même application.

Le chapitre 2 présentera une étude critique des extensions faites aux concepts classiques des langages objets pour leur permettre de résoudre plus efficacement les problèmes mentionnés ci-avant.

Le chapitre 3 présentera d'une part le concept d'*héritage générique* et son influence sur les concepts objets déja existant dans le langage. D'autre part il exhibera une piste pour prévenir certains problèmes qui auront été soulevés dans la première partie du chapitre.

Enfin le chapitre 4 résumera la démarche générale qui a été suivie tout au long de cette étude et donnera quelques perspectives de notre travail.

[2] C'est le principe du paradigme de la séparation des préoccupations (voir section 2.4).

[3] La généricité est modélisable par la seule relation d'héritage, voir [MEY86]

[4] La cible (respectivement source) d'une relation d'héritage représente la super-classe (respectivement la sous-classe). Nous emploierons ces deux termes tout au long de ce rapport.

Chapitre 2

Étude de l'existant

2.1 Généricité

La généricité est, en accord avec [MEY97], une approche permettant une paramétrisation des classes par des types. Cela définit une généralisation *horizontale* à opposer à la généralisation *verticale* offerte par l'héritage. La généricité permet de concilier deux objectifs du génie logiciel : la fiabilité, car nous conservons un typage statique par la déclaration explicite d'un type formel, et la réutilisabilité en permettant d'exprimer une entité qui recouvre des variantes d'une même notion.

Nous présenterons la généricité dans le langage EIFFEL (l'un des premier langages à objets ayant offert cette relation) puis une variante de celle-ci avec les templates de C++.

2.1.1 Généricité et Eiffel

En Eiffel la généricité est multiple, bornée[1] ou non, mais non F-bornée.

Ainsi une *classe générique* peut posséder un ou plusieurs paramètres formels représentant un type. Le paramètre peut alors être utilisé partout dans la classe, en remplacement d'un type quelconque. Ce paramètre sera spécifié par l'*instanciation générique*, c'est-à-dire lors de l'utilisation de la classe générique comme type particulier (dépendant du paramètre spécifié).

Ces paramètres peuvent être bornés par un type, c'est-à-dire que l'instanciation générique devra respecter la conformance entre le paramètre effectif et le type du paramètre formel. Si ce dernier n'est pas borné, tout type est alors valide[2].

Enfin la relation F-bornée représente le fait que la borne elle-même puisse dépendre d'un paramètre formel précédent (par exemple $ORDEREDSET[C \prec COMPARABLE[C]]$, qui exprime que le type C doit être comparable avec lui-même).

Nous pouvons noter que le langage Eiffel accepte une relation de conformance entre deux types provenant de deux dérivations génériques dont les paramètres effectifs sont en conformance. Par exemple, *Étudiant* hérite de *Personne* donc *Étudiant* se conformera à *Personne* et ainsi *Liste d'Étudiant* serra conforme à *Liste de Personne*.

Cette conformance entraîne des erreurs potentielles à l'exécution. En effet nous nous retrouvons dans le cas de l'utilisation en même temps de la covariance et du polymorphisme, situation pouvant entrainer des appels de méthodes invalides. [MEY97] discute de ces problèmes en proposant les appels *CAT*[3] et [COO89] propose une révision de cette conformance pour éviter ces éventuels problèmes.

[1] La litterature parle aussi de généricité *contrainte*. Nous emploierons les deux termes dans la suite.

[2] Comme Eiffel possède une racine unique dans sa hiérarchie d'héritage, ANY, la généricité non bornée peut être assimilée, en première aproximation, à une généricité bornée à ANY.

[3] On parle d'appel CAT (Changer l'Accés ou le Type) si une redéfinition de la routine rendrait l'appel invalide du fait du changement du statut d'exportation ou du type d'un argument.

2.1.2 Templates et C++

Les *templates* représentent la généricité dans le langage C++ ([STR92]). Ce sont des patrons de classe, au sens syntaxique du terme. Ces patrons permettent de paramétrer une classe par un ensemble d'entités (en fait par un ensemble de noms désignant ces entités) comme par exemple un paramètre générique, une fonction voire même une super-classe.

Supposons qu'une classe de type *Personne* possède une méthode *afficherNom()* comme dans l'exemple ci-dessous[4]. Il est possible de créer une classe *template* définissant la propriété d'être *Docteur* et qui pourra se dériver sur la classe de type *Personne* :

```
1  class Personne{
2          string  nom;
3
4          virtual void afficherNom(){
5                  cout << nom << endl;
6          }
7  }
8
9  template<class super>
10 class Docteur : public super {
11
12         virtual void afficherNom(){
13                 cout << "Dr ";
14                 super::afficherNom();
15         }
16 }
```

– Exemple de template en C++

Nous avons ici la déclaration de deux classes dont l'une (la classe *Docteur*) est une classe paramétrée. Nous pouvons voir que la classe *Docteur* va hériter de son paramètre *super*. L'appel de la méthode *afficherNom()* sur une instance de la classe *Docteur* que nous aurons préalablement dérivée génériquement de la classe *Personne* préfixera le nom de la personne par "Dr". L'intérêt est que la classe *Docteur* pourra être dérivée à partir de n'importe quelle sous-classe de la classe *Personne*[5]. Notons qu'aucun contrôle n'est demandé à l'utilisateur et que la cohérence de la solution ne sera testée qu'à l'éxécution ou au mieux à la compilation.

Retour à la généricité "classique" : Nous pouvons noter que les templates ne permettent qu'une généricité non contrainte. De plus, C++ adopte une règle de conformance différente d'Eiffel : une classe dérivée génériquement sur une classe B, n'est pas en conformance avec une classe générique dérivée sur A, qui est elle-même une super-classe de B (voir par exemple la fin de 2.1.1). L'explication donnée dans [STR92], motive ce choix par la différence d'implémentation des objets issues des deux classes dérivées.

2.1.3 Discussion

Nous avons vu ici deux approches totalement différentes de la généricité. Celle du langage Eiffel, qui introduit une nouvelle entité dans le langage : la *classe générique*, et celle de C++ qui s'appuie sur des notions syntaxiques ou de réécriture, et définit de fait des patrons de classes.

[4] Cet exemple ne présente pas l'utilisation des templates pour la généricité "classique" horizontale mais plutôt un cas d'utilisation spécifique des templates.

[5] Ici nous avons décrit une utilisation particulière des templates. En fait cet exemple est l'implémentation d'entité appelé *mixins* que nous verrons un peu plus tard. Cet exemple montre juste la possibilité des templates.

Comme cela a déja été dit, la généricité n'apporte en elle même aucun pouvoir expressif supplémentaire aux langages à objets. Elle est modélisable par la seule relation d'héritage (voir [MEY86]). Cependant son absence rend certains concepts comme les *conteneurs* (tableau, liste, ...) laborieux à implémenter ou à utiliser. Le langage JAVA, dans sa version 1.5 devrait proposer à son tour une implémentation de la généricité.

Il existe une implémentation de la relation de généricité F-borné dans l'extension PIZZA du langage JAVA ([MO97]).

Il est à noter que certains cas d'utilisation de la généricité contrainte sont jugés criticables par [EVE97].

2.2 Programmation par Mixins

Un mixin, en accord avec [GB90], est une sous-classe abstraite que nous pourrons "plugger" à différentes super-classes. Les mixins sont utilisés en CLOS, mais de manière implicite : aucune construction syntaxique ne permet de déclarer un mixin, seul l'appel à *(call-next-methode)* (un équivalent à *super* de Java ou à *precursor* d'Eiffel), alors qu'aucune super-classe n'était spécifiée, permettait de qualifier cette entité de mixin.

2.2.1 Description par l'exemple

L'exemple précédent, les templates de C++, est une implémentation des mixins. Nous allons présenter ici les caractéristiques des mixins, et les aspects qu'il sera intéressant de trouver dans la relation que nous désirons mettre en œuvre.

Un mixin est une nouvelle entité ajoutée aux langages à objets. Ce sont des *modules* (encapsulation de propriété) pouvant contenir des attributs et des méthodes. Les mixins n'ont pas de relation d'héritage entre eux (du moins, dans l'approche décrite dans [GB90]). Les mixins sont attachés individuellement à une classe au fur et à mesure des besoins. Cette dérivation crée automatiquement une nouvelle classe, sous-classe de la première, possédant les nouvelles propriétés[6] apportées par le mixin. Enfin les mixins représentent une type, au sens d'une interface commune entre toutes les classes issus de ce mixin.

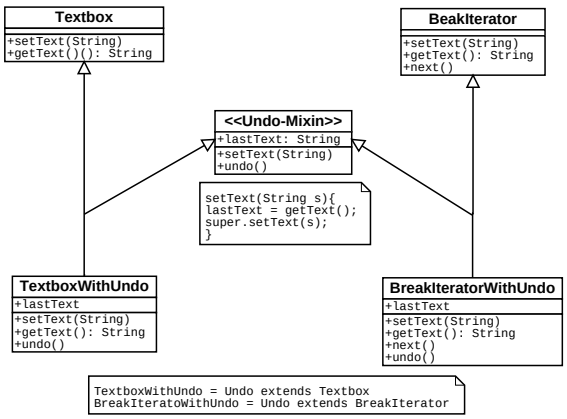

FIG. 2.1 – Exemple d'utilisation des mixins.

L'exemple[7] présenté en figure 2.1 montre la factorisation de fonctionnalités par l'intermediaire d'un mixin (ici le *undo-mixin*). Nous pouvons remarquer l'appel à *super* dans le corps de la méthode *setText(...)* encapsulée dans le mixin. Enfin les deux classes dérivant du mixin, *TextboxWithUndo* et *BreakIteratorWithUndo* possèdent bien la méthode *undo()* apportée par le mixin.

[6] En fait elles peuvent aussi représenter de simples redéfinitions (c'est le cas de *setText()* dans l'exemple de la figure 2.1).

[7] L'exemple et écrit en JAM, une extension du langage Java avec des mixins (voir [DA99]).

2.2.2 Discussion

Les mixins sont, au premier abord, très proches de la relation qui sera présentée dans le chapitre 3. Mais contrairement à cette approche, notre étude ne tente pas d'ajouter une nouvelle entité au langage. Nous souhaitons conserver la classe comme entité de construction principale.

Les mixins sont vus comme un moyens de spécialiser chaque classe appartenant à un ensemble de super-classes. Notre démarche se veut différente : nous souhaitons plutôt pouvoir modifier une super-classe, en d'autres termes, pouvoir spécialiser "par le haut" notre hiérarchie.

Les mixins ne préviennent pas une explosion du nombre des classes générées. Les mixins ne permettent qu'une extension d'une hiérarchie, par la dérivation d'une super-classe pour donner une nouvelle classe. Ceci n'est pas satisfaisant et ne correspond pas à notre objectif qui est de pouvoir adapter une hiérarchie sans forcément devoir l'étendre.

La popularité des mixins dans le langage CLOS a incité à les introduire dans d'autre langages comme C++ ([SKA93]) ou Java ([DA99]).

2.3 Classes virtuelles

Le langage BETA[8] généralise les notions de procédure, classe, etc, sous la forme de *pattern*. La redéfinition des méthodes est modélisée par les *procédures virtuelles*, et la généralisation de la redéfinition aux classes est faite par les *classes virtuelles*.

BETA à un comportement spécifique relativement à ces procédures virtuelles : c'est la *super-méthode* qui définit, via le mot clef *inner*, où les futures spécialisations de l'algorithme seront intégrées. BETA ne permet donc une redéfinition que par l'enrichissement de la méthode, par opposition à une redéfinition libre. Ce choix se justifie par la conservation d'une *compatibilité structurelle* de la méthode par opposition à une simple *compatibilité nominative* dans les autres langages.

Les *classes virtuelles* généralisent cette notion aux classes internes : Une classe interne à une autre classe pourra être étendue lors la spécialisation de cette dernière.

Enfin la notion de *super-classe virtuelle* est la plus intéressante pour notre étude. Dans l'exemple suivant, issu de [OLM89], nous souhaitons ajouter un attribut à toute une hiérarchie. Pour cela nous allons créer une *super-classe virtuelle* qui va servir de générateur d'instance.

2.3.1 Description par l'exemple

Nous souhaitons ajouter un champ *ArcNo* représentant un numéro à l'ensemble des classes d'une hiérarchie de type : *Publication*, *Article*, *Book*.

```
1  Publication: class Record
2              (# Author: ...; Title: ...; Date: ... #) ;
3
4  Book: class Publication
5      (# ... #) ;
6
7  Article: class Publication
8          (# ... #) ;
9
10
11 PubGen: class
12         (# GenType: virtual class Publication;
13            ArcPub: class GenType;
```

[8] Voir [FAQ] qui est un très bon point de départ vers ce langage.

```
14                      (# ArcNo: @integer #) ;
15              New: proc
16                      (# RN: @integer; R: ^ArcPub
17                         enter RN
18                         do new ArcPub[] -> R[];
19                            RN-> R.ArcNo;
20                         exit R[]
21              #) ;
22  #) ;
23
24  BookGen: class PubGen
25          (# Gentype: extended class Book #) ;
26
27  ArticleGen: class PubGen
28              (# Gentype: extended class Article #) ;
```

– Exemple en BETA

Nous pouvons voir, lignes 1 à 8, un ensemble de définition de classes : *Publication* qui hérite de *Record*, *Book* et *Article* qui hérite de *Publication*. Puis la ligne 11 définit une classe *PubGen* qui va être la classe responsable de la génération des instances de la classe *Publication*.

Les lignes 12 et 13 définissent des classes internes dont l'une *GenType* est *virtuelle*, c'est à dire qu'elle pourra être spécialisée dans une sous-classe de la classe *PubGen*. La classe interne *ArcPub* étend la classe *GenType* en lui ajoutant le champ désiré (ligne 14). Le but est de pouvoir modifier automatiquement la super-classe de *ArcPub*. Une classe héritière de *PubGen* pourra redéfinir la classe *GenType*, et ainsi modifier la cible d'héritage de la classe interne *ArcPub*.

La fonction *New*, redéfinie à la ligne 15 permet de créer de nouvelles instances de la classe *Publication*, via les classes internes *ArcPub* et *GenType*. Les instances générées seront des instances de la classe *ArcPub*, c'est-à-dire une extension, par l'ajout d'un champ *ArcNo*, de la classe *GenType*.

Enfin les lignes 24 et 27 définissent les classes génératrices d'instances de *Book* et *Article* en ne redéfinissant que *Gentype*.

La création d'instance des classes *Publication*, *Book* et *article* se fait respectivement en instanciant respectivement les classes *PubGen*, *BookGen* et *ArticleGen*.

2.3.2 Discussion

Ainsi, cette approche a permis de simuler l'ajout d'un attribut à la hiérarchie de la classe *Personne*. Cette solution n'est cependant pas vraiment idéale. Si l'encapsulation de l'ajout de la fonctionnalité est faite, il faut cependant encore créer les générateurs d'instances pour chaque classe de la hiérarchie. De plus il est nécessaire d'utiliser les générateurs d'instances. Nous n'avons pas adapté la bibliothèque elle-même, mais seulement adapté son utilisation pour prendre en compte l'extension de la bibliothèque.

Enfin nous pouvons rapprocher cette utilisation des classes virtuelles au patron de conception *Abstract Factory* définis dans [EG96].

2.4 Séparation des préoccupations

La séparation des préoccupations est un nouveau paradigme de programmation introduit par [WLH95]. Lors du développement d'une application ou d'une bibliothèque, ce paradigme propose de séparer les différents *aspects* d'un problème ainsi que de séparer leur implémentation. Il est proposé de se concentrer sur l'algorithme principal du développement, et de concevoir séparément les fonctionnalités connexes. Ces différentes préoccupations sont ensuite assemblées grâce à des *points de jointure* qui spécifient où interviendront les différentes préoccupations au sein de l'application.

Nous allons vous présenter un exemple d'utilisation reposant sur le langages AspectJ.

2.4.1 Un exemple : AspectJ

AspectJ ([GK]) est un langage permettant d'intégrer le paradigme de la séparation des préocupations dans le langage Java. Il se base sur une transformation du code de manière statique (au moment de la compilation).

Dans l'exemple suivant, nous souhaitons avoir la fonctionnalité de *trace* des appels de méthode qui imprime tous les appels de méthodes effectués sur les objets. Voici une implémentation en AspectJ.

```
1  abstract aspect Trace {
2
3      protected static void traceEntry(String str, Object o) {
4        stream.println("Entering␣" + str + ":␣" + o.toString());
5      }
6
7
8      abstract pointcut myClass(Object obj);
9
10     pointcut myConstructor(Object obj): myClass(obj) && execution(new(..));
11     pointcut myMethod(Object obj): myClass(obj) &&
12         execution(* *(..)) && !execution(String toString());
13
14     before(Object obj): myConstructor(obj) {
15         traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
16     }
17     before(Object obj): myMethod(obj) {
18         traceEntry("" + thisJoinPointStaticPart.getSignature(), obj);
19     }
20 }
```

– Exemple en AspectJ

Nous pouvons voir, à la ligne 1, la définition d'une nouvelle entité : un aspect. Il va encapsuler plusieurs propriétés dont une méthode (ligne 3) qui nous permettra d'afficher les appels de méthodes sur un objet.

Les lignes 8, 10 et 11 définissent des points de jointure qui localiseront dans notre programme les endroits où il faudra appliquer la préoccupation de trace de l'appel de méthode.

Enfin à partir de la ligne 14 est spécifié comment seront combinés les points de jointure et l'aspect pour réaliser l'application complète. Ainsi, avant chaque exécution d'une méthode (ou d'un constructeur) sur un objet, la méthode *traceEntry(...)* encapsulée par l'aspect *Trace* sera appelée (lignes 15 et 18).

2.4.2 Discussion

La séparation des préoccupations permet ainsi d'adapter de manière externe une classe ou un ensemble de classes à travers la manipulation de certaines entités réifiées, mais aussi, en modifiant ces entités par l'intermediaire de l'ajout d'un comportement supplémentaire.

Un problème reste cependant ouvert, celui de la combinaison des préoccupations. En effet, permettre un développement des fonctionnalités de manière indépendante est utile mais permettre leur intégration de manière séparée en conservant une certaine cohérence est plus difficile. [KO00] discute de ce point.

La séparation des préoccupations a été appliquée au paradigme objet mais aussi fonctionnel [GK97].

2.5 MOP et langages réfléxifs

Le MOP, ou Méta-Object Protocol, d'un langage, est une réification des entités de ce langage. Grâce à ce métaniveau, il devient possible de contrôler précisément certains aspects de la sémantique du langage, et ainsi de factoriser un comportement pour un ensemble d'entités.

Les MOP réifient un ensemble d'entités qui deviennent alors des objets de première classe. La manipulation de ces métaentités, mais surtout leur modification (par exemple par spécialisation) permettent de modifier la sémantique du langage sous-jacent.

Par exemple la fonction *lookup*, qui modélise la sélection dynamique par la recherche de la méthode associée à un message et un receveur, est potentiellement redéfinissable par le programmeur.

Si les relations, ou composants-relations[9] sont réifiés, des métaentités représentant des composants-descriptions sont aussi présentes.

Les métaclasses par exemple permettent, de factoriser le comportement d'un ensemble de classes, elles ont ainsi le même comportement vis-à-vis des classes que celui que possèdent ces dernières vis-à-vis des objets. Mais les métaclasses permettent aussi d'encapsuler un service pour le rendre disponible à un ensemble de classes non liées par une relation d'héritage. L'absence de hiérarchie commune empéche de factoriser ce service au sein d'une même racine.

Nous allons donner, dans la section suivante, un exemple simple du réel intérêt des MOP. Le problème de la *trace*, qui est de pouvoir afficher tous les appels de méthodes d'une certaine classe peut être aisément résolu grâce au MOP. Nous allons en donner une implémentation en OpenJava.

2.5.1 Un exemple : OpenJava

OpenJava se place à un niveau méta par rapport au langage Java.

Une particularité d'OpenJava est que le MOP n'est accessible qu'à la compilation et non à l'exécution. Son utilisation se fait en compilant une première fois l'application utilisant le MOP, ce qui produit alors du code Java standard qui sera ensuite compilé par un compilateur Java traditionnel.

Voici le code d'une implémentation de la préoccupation de trace des méthodes.

```
1  public class TraceClass instantiates MetaClass extends OJClass{
2      public void translateDefinition() throws MOPException {
3          OJMethod[] methods = getDeclaredMethods();
4          for(int i=0;i<methods.length;i++){
5              Statement printer = makeStatement(System.out.println(methods[i] + "is called."));
6              methods[i].getBody().insertElementAt(printer,0);
7          }
8      }
9  }
10
11 public class Personne instantiates TraceClass{
12     public String getName(){...};
13     public int getAge(){...};
14 }
```

– Exemple en OpenJava

Nous pouvons voir, ligne 1, que notre classe *TraceClass* est une instance de la métaclasse *MetaClass*, et hérite de la métaclasse *OJClass*. *MetaClass* représente la métaclasse de toutes les métaclasses tandis que *OJClass* est la racine de l'arbre d'héritage des métaclasses.

[9] Nous reprenons la terminologie de OFL décrite dans [CRE01].

La ligne 2 redéfinit la méthode *translatedefinition()* définie dans la métaclasse *OJClass* qui va nous permettre d'incorporer le comportement de trace.

Ici l'implémentation est faite par l'ajout du code qui assure l'affichage du nom de la méthode à la définition du code de la méthode elle-même. Cette fusion sera faite à la compilation. Ainsi dans les lignes 3 à 6, les méthodes définies dans la classe sont une à une modifiées en ajoutant un *Statement*, c'est-à-dire du code Java, représentant l'algorithme d'affichage du nom de la méthode.

L'utilisation de la métaclasse *TraceClass* peut être faite comme dans l'exemple de la ligne 11 : Nous définissons une nouvelle classe *Personne* ayant pour métaclasse la classe *TraceClass*. Toutes les méthodes de la classe *Personne* afficheront alors leur nom à chaque appel.

2.5.2 Discussion

Nous avons vu dans la section précédente une façon d'encapsuler une préoccupation pour la rendre réutilisable. Toute classe voulant "tracer" ses méthodes doit simplement avoir comme métaclasse *TraceClass*. Le MOP répond donc bien à un des problèmes cités dans notre introduction, à savoir créer des composants réutilisables inter-bibliothèques.

Cependant l'adaptation de ces nouvelles entités n'est pas résolue. En quelque sorte nous n'avons en fait que déplacé le problème. Le MOP permet de factoriser des comportements pour un ensemble d'entités, mais la modification de ces concepts par l'utilisateur n'est toujours pas facilité.

Du point de vue de la *conception*, la présence et l'utilisation du MOP est intéressante. La factorisation de comportement est bénéfique à la fois pour la maintenance et le développement en séparant les algorithmes qui ne sont pas associés aux mêmes besoins[10].

Du point de vue de l'utilisation, la complexité des entités ajoutées par un MOP, mais surtout la modification possible de la sémantique du langage en font un outil d'abord difficile. De plus la question de l'adaptation de ces nouvelles fonctionnalités proposées à un niveau méta pour une application sparticulière n'est pas proposée en dehors de la modification directe du code source.

D'autres MOP restent présents à l'exécution, en particulier le MOP de CLOS, qui permet même de modifier la hiérarchie de classe à l'exécution, ou encore celui de SmallTalk.

2.6 Bilan

Dans les sections précedentes, nous avons examiné plusieurs techniques permettant d'améliorer les capacités d'adaptation, d'évolution et de réutilisation des hiérarchies de classes.

Un premier aspect qui semble interessant est la capacité à pouvoir changer la cible de l'héritage, un autre est de placer le code orthogonal ou représentant une adaptation à l'exterieur de la classe.

Cependant, nous souhaitons rester dans le contexte des langages objets et donc éviter d'ajouter de nouvelles entités au langage, ou de réifier une trop importante partie de celui-ci.

D'autres approches que celle présentées ici amènent d'autres éléments de solutions, comme le concept de *Traits* ([NS02]), la combinaison de hiérarchie ([HO92]), l'héritage inverse ([TL94, SAK02]), ou encore les multiméthodes ([DGB86, CHA92, STE90]).

[10] Dans un sens, nous retrouvons la séparation des préoccupations présentée dans la section précédente. Cependant nous pouvons remarquer que les MOP restent dans un cadre purement Objet en généralisant seulement le concept de Classe à elles-même.

Chapitre 3

Vers une relation d'héritage générique

Le point de départ de cette étude est notre volonté à ne pas fixer une fois pour toute la classe héritée, nous souhaitons pouvoir modifier la cible de la relation d'héritage d'une classe.

Ainsi une classe G pourrait hériter dans un contexte particulier (dans une application spécifique ou dans une partie d'application) d'une classe A, mais dans un autre contexte hériter d'une classe B. Il semble raisonnable d'obliger que les différentes cibles appartiennent à une même hiérarchie et qu'ainsi elles ne représentent pas des concepts complètement différents. En effet, la relation d'héritage est couramment admise[1] comme étant la relation *est un* entre une sous-classe et sa super-classe. G doit donc aussi satisfaire cette relation entres ses différentes cibles possibles. Les cibles (A ou B) doivent donc appartenir à la même hiérarchie (représenter un même concept) pour conserver un minimum de cohérence.

3.1 Approches possibles

Nous pouvons voir notre relation sous deux points de vue très différents. En effet nous avons, dans l'état de l'art, abordé plusieurs techniques pour faire évoluer les langages à objets. Deux de ces techniques nous ont permis d'identifier deux aspects de notre nouvelle relation d'héritage.

Tout d'abord, à la manière de la généricité bornée, nous pouvons spécifier comme cible d'héritage pour G non pas une classe mais un paramètre qui devra être valué par une classe pour utiliser G. Ce point de vue sera nommé *héritage générique*.

Une autre solution est de permettre de modifier de manière externe au langage la cible d'héritage de la classe G. A la façon du langage AspectJ, nous pourrions modifier une partie de notre application de manière externe. Ce point de vue sera nommé *redéfinition de la cible d'héritage*.

3.1.1 Vers une nouvelle relation : l'héritage générique

Nous souhaitons donc spécifier, comme cible d'héritage de la classe G, une borne (représentant un type minimum dans la hiérarchie) et non une classe précise.

Dans un certains sens, nous allons étudier une variante de la relation d'héritage où la cible ne sera pas spécifiée en extension mais plutôt en intention. Nous allons spécifier de quel service ou concept la classe G souhaite hériter et non de quelle implémentation précise de ces concepts elle va hériter.

La relation d'héritage actuelle, c'est-à-dire proposée par la plupart des langages objets, semble aller à l'encontre de principes objets établis dans [MEY97].

Ce dernier exprime l'héritage comme permettant de définir de nouvelles classes par extension, spécialisation et combinaison de classes définies précédemment. Cette même référence définit aussi

[1] Ceci provient surtout du fait que la relation *est-un* semble être le plus petit point commun entre une majorité des utilisations possibles de l'héritage.

le principe d'indépendance de représentation, ou un client d'un module doit pouvoir spécifier une opération sans savoir comment elle est représentée.

Cependant, la relation d'héritage offerte par les langages ne permet, pour une classe cliente de cette relation, de ne spécifier comme cible qu'une implémentation d'un ensemble de services (par la spécification de super-classes particulières) et non un ensemble de ces services de manière abstraite. Ainsi l'extension, la spécialisation et la combinaison de super-classes existantes ne peut se faire de manière totalement abstraite.

Nous ne souhaitons pas, dans certains cas[2], être obligés de spécifier, lors de l'héritage, une implémentation particulière de services associés à un type, mais seulement leur description abstraite. L'implémentation de ces derniers pourra être choisie ultérieurement par l'utilisateur, en accord avec son contexte d'application.

Cette nouvelle relation d'héritage a pour but d'étendre les possibilités de conception de hiérarchie de classe (en augmentant l'abstraction de la description), mais aussi ses capacités de réutilisation ou d'adaptation. Ces dernières possibilités pourront être améliorées en permettant à l'utilisateur final (typiquement, le développeur utilisant la hiérarchie) de spécifier l'implémentation du module hérité.

Quand une classe G va hériter génériquement d'un type A[3], elle sera potentiellement héritière de n'importe quel type de la hiérarchie de A. L'utilisateur pourra spécifier G comme étant une sous-classe de B ou C (voir figure 3.1). La spécification de la super-classe de G sera nommée *instanciation générique d'héritage*.

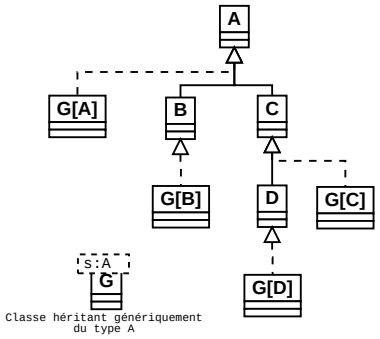

FIG. 3.1 – Instanciation générique d'héritage possible d'une classe d'héritage générique G.

3.1.2 Permettre la redéfinition de la cible d'héritage

Nous souhaitons pouvoir redéfinir la cible de la relation d'héritage d'une classe. Cela permettrait d'adapter le composant à une application particulière.

Pour pouvoir redéfinir la cible il nous faudra un métalangage nous permettant de manipuler la relation d'héritage. Nous pourrons nous inspirer des langages du paradigme de la séparation des préoccupations ou de langages de configuration qui permettent de manipuler de manière externe certaines entités du langage.

Notons qu'ici le concepteur du composant n'a pas à intervenir pour préciser le cadre d'adaptation.

[2]Nous ne souhaitons pas supprimer le comportement classique de la relation d'héritage, mais plutôt de lui ajouter un deuxième, alternative plus pertinente pour certains cas d'utilisations.

[3]Nous noterons tout au long de cette étude la classe d'héritage générique G par G[s :A] ou simplement G.

3.2 Problèmes recensés

L'introduction de la possibilité de modifier la cible de la relation d'héritage d'une classe va poser de nombreux problèmes. Nous allons les détailler dans cette section. Nous prendrons pour exemple la relation d'héritage générique.

3.2.1 Conflits

Le fait de ne pas donner, dans la relation d'héritage générique, l'interface exacte héritée par la sous-classe, entraîne des conflits potentiels.

Une instanciation d'héritage générique de notre classe source G sur la classe A représentant le type de la borne n'est pas problématique. Toute méthode définie dans G introduira une nouvelle méthode ou redéfinira une méthode de A.

Cependant, si le paramètre spécifié est une sous classe de A, G n'a aucune connaissance exacte de l'interface de cette sous-classe, en particulier si elle étend celle de A. Une méthode introduite dans G peut alors entrer en conflit avec une méthode définie dans la sous-classe de A comme montré dans la figure 3.2.

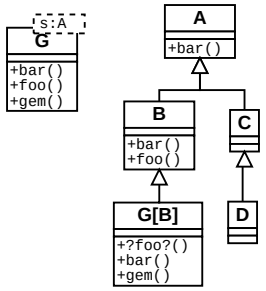

FIG. 3.2 – Exemple de conflits.

Dans la figure 3.2 la fonction *bar()* est bien redéfinie par G tandis que *gem()* est introduite. Cependant la méthode *foo()* ne peut être considérée comme une redéfinition.

Ces conflits sont uniquement des conflits de nom, les méthodes n'ont *a priori* aucune raison d'avoir la même sémantique. Laisser l'instanciation d'héritage générique redéfinir[4] la méthode en conflit (*G[B]* redéfinirait la méthode *foo()* de B) n'est pas acceptable sans accord explicite de l'utilisateur. Il faut donc donner à l'utilisateur final (l'utilisateur instanciant génériquement G sur B) la possibilité de résoudre ces conflits.

Permettre un renommage ou une suppression d'une des méthodes, comme cela est permis en Eiffel pour résoudre les problèmes de l'héritage multiple, peut être envisagé. Mais dans ce cas, il faut définir la localisation de cette adaptation

La spécialisation de notre classe d'héritage générique en fonction des paramètres possibles (afin de résoudre les conflits) semble nécessaire, mais amène plusieurs questions. Où effectuer cette spécialisation ? De manière globale et externe au langage, comme dans les langages de configuration, ou directement dans la classe G ? Peut-on accepter de spécialiser deux instanciations sur le même paramètre formel différemment ?

L'alternative d'éviter ce problème ne serait-elle pas plutôt à envisager ?

Nous pouvons noter que ce problème n'apparaîtrait pas si l'on contraignait l'héritage de la hiérarchie de A à ne pas étendre l'interface de A. Une autre solution serait de contraindre la hiérarchie de G à ne pas étendre l'interface de A.

[4] Un terme plus approprié serait celui de "masquer".

3.2.2 Redéfinition covariante des méthodes

La redéfinition covariante des méthodes n'est pas sûre du point de vue des types mais est nécessaire pour une bonne modélisation [DUC02].

La redéfinition covariante des méthodes entraîne des problèmes graves avec l'ajout de l'héritage générique. En effet nous avons borné le type du paramètre générique, ce qui nous permet d'utiliser le type de la borne ou un quelconque sous-type. Ce fait nous permettrait d'avoir une redéfinition contravariante alors même qu'elle n'est pas prévue dans le langage.

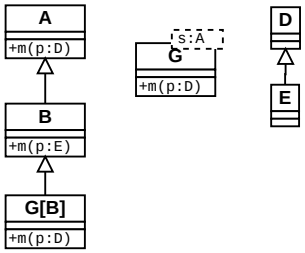

FIG. 3.3 – Exemple avec redéfinition contravariante.

L'exemple de la figure 3.3 montre que l'instanciation générique de la classe G avec le paramètre formel spécifié à B entrainera une redéfinition contravariante de la méthode m.

Notons que les langages actuels possèdent tous un comportement de variance différent : Java est non variant, C++ possède un comportement variant et Eiffel est covariant.

3.2.3 Assertions

Les assertions représentent une technique du génie logiciel permettant de préciser la sémantique de différentes entités. La programmation par contrat, développée par le langage Eiffel, s'appuie sur ces techniques.

En s'appuyant sur les propriétés de ces concepts définies dans [MEY97], nous allons étudier leur influence sur notre relation.

Invariants de classe

> Les invariants des parents d'une classe s'appliquent à cette classe.

G doit respecter les invariants définis dans A, mais devra vérifier aussi les invariants définis dans le paramètre effectif lors de son instanciation d'héritage générique. Si G est instancié génériquement sur B, alors G doit respecter les invariants de B. En admettant que G ne respecte pas ces invariants, il ne peut être instancié génériquement sur B mais aussi sur toute la sous-hiérarchie de B.

Notons que cette vérification ne peut être effectuée que par le développeur final, utilisateur de G, et qu'elle n'est pas automatisable (le compilateur ne pourra pas détecter certaines incompatibilités).

Préconditions et postconditions

> Une redéclaration de routine ne peut remplacer la précondition d'origine que par une précondition plus faible ou égale, et la postcondition d'origine par une postcondition plus forte ou égale.

Si les méthodes définies par G doivent respecter ces règles d'assertion pour la classe A, elles doivent aussi les respecter pour tout paramètre d'instanciation d'héritage générique effectif.

Exemple L'exemple présenté en figure 3.4 montre la difficulté de conserver les propriétés définies plus haut (la sous-classe devant respecter les assertions de sa super-classe) si nous tentons de modifier la cible de la relation d'héritage de la classe G.

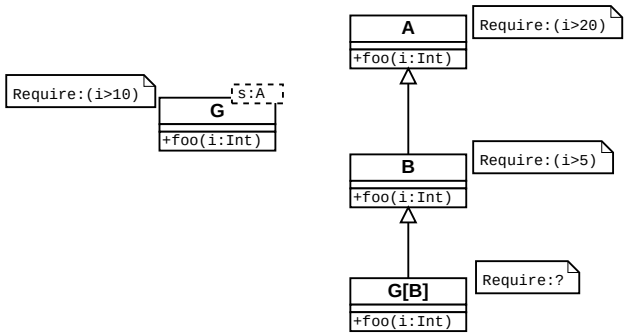

FIG. 3.4 – Exemple avec non respect des assertions.

Discussion Si ces contraintes semblent très fortes, elles ne sont en fait là que pour une sécurité supplémentaire de l'application, et ne sont pas une limitation réelle. Notre relation d'héritage générique permet de spécifier un ensemble de propriétés (abstraites) dont nous voulons hériter pour permettre un choix ultérieur de l'implémentation de ces dernières. G, en spécifiant de nouvelles contraintes (assertions ou invariants), ne fait que "sélectionner" les implémentations potentielles.

3.2.4 Discussion générale

Notre approche doit donc pouvoir prendre en compte les problèmes décrits plus haut.

Ces problèmes viennent du fait que nous n'avons aucun contrôle sur les adaptations faites la définition de relations d'héritage. En effet, la relation d'héritage permet de nombreuses adaptations (ajouts de nouvelles propriétés, redéfinitions, ajouts d'assertions, etc ...) et est utilisée dans de nombreux cas (sous-typage, extension, implémentation ...). De nombreux articles discutent des utilisations justifiées (ou non) de la relation d'héritage [TAI96, DCH87, MEY97].

Dans la section suivante, nous allons étudier l'influence sur les problèmes ci-avant d'une meillleure spécification de l'utilisation dans le cadre de la modification de la cible d'héritage.

3.3 Vers une spécification de l'utilisation de l'héritage

Pouvons nous résoudre les problèmes rencontrés ci-dessus en posant des contraintes sur les relations d'héritage? C'est-à-dire, pouvons-nous prévenir les problèmes montrés dans la section précédente en limitant les types des relations d'héritage qui touchent l'utilisation de l'héritage générique?

Plusieurs taxinomies de la relation d'héritage ont été faites ([MEY97, GIR91]), spécialisant son utilisation. Leur utilisation lors de la définition d'une relation d'héritage pourrait permettre de certifier l'abscence de problèmes lors de la modification de la cible d'héritage. Nous souhaitons donc que le programmeur puisse exprimer son usage de la relation d'héritage qu'il décrit.

Les deux taxinomies présentées dans les références ci-dessus se rapprochent sur beaucoup de points, aussi nous n'étudierons que celle de [MEY97]. Une particularité de la taxinomie de [GIR91] sera toutefois présentée à la fin.

Mais avant cela nous allons appuyer notre démarche d'expliciter la sémantique attendue de la relation d'héritage sur quelques travaux connexes.

3.3.1 Quelques travaux connexes

Nous pouvons déjà noter que cette capacité à pouvoir spécifier l'héritage utilisé est déjà présente, explicitement ou implicitement dans certains langages.

En Java par exemple, hériter d'une classe abstraite oblige à concrétiser la sous-classe, c'est-à-dire, concrétiser toutes les propriétés, à moins que l'on ne précise explicitement que la classe source est elle aussi abstraite.

De même, [TL95] propose grâce à l'utilisation d'un MOP, de spécifier la nature des classes, c'est-à-dire de spécifier certaines propriétés particulières comme le fait d'être abstraite ou de ne pouvoir posséder de sous-classes. L'article mentionné se base sur le langage SmallTalk, auquel il a été ajouté un protocole meta-objet : ClassTalk.

D'autres caractéristiques sont introduites comme la possibilité de donner un ensemble de méthodes à redéfinir par les sous-classes ou empêcher la modification de l'interface.

Ces propriétés particulières sont encapsulées au sein de métaclasses dont les classes doivent préciser qu'elles en sont des instances.

Ces caractéristiques influent sur les futures relations d'héritage entre une classe, instance d'une métaclasse précise et ses (futures) sous-classes. D'une certaine manière, l'article propose à une classe de pouvoir contraindre les futurs héritages dont elle sera la cible.

L'article se concentre sur l'amélioration de l'organisation structurelle des classes et non sur des propriétés permettant d'obtenir de nouveaux contrôles sur les hiérarchies. Cependant, il propose une utilisation pour l'amélioration des environnements de programmation en augmentant la compréhension d'un composant.

3.3.2 Taxinomie de [MEY97]

Cette classification propose un arbre d'utilisation justifié de la relation d'héritage entre deux classes. Cette hiérarchie est décomposée en trois sous-branches principales représentant trois points de vue différents de la relation d'héritage.

Nous la présentons ici succintement. Les définitions données sont en grande partie reprise de [MEY97].

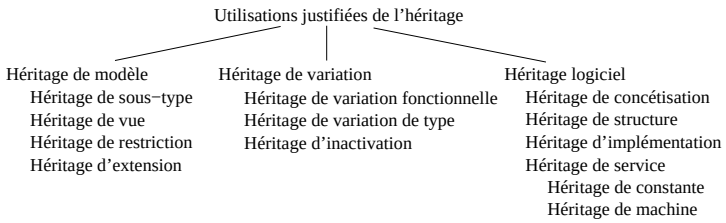

FIG. 3.5 – Taxinomie des utilisations justifiées de l'héritage.

Héritage de Variation : Ce type d'héritage permet de décrire une classe par ses différences avec une autre classe. Il possède trois variantes :

- **Héritage de variation fonctionnelle :** Cet héritage correspond au changement du corps de certaines caractéristiques, c'est à dire de leur implémentation. Les assertions peuvent êtres modifiées. L'ajout de caractéristiques est possible mais seulement dans le cadre de la nouvelle implémentation des caractéristiques redéfinies (celles-ci ne sont accessibles uniquement par les caractéristiques redéfinies).
- **Héritage de variation de type :** Ne permet que de modifier le type et la signature des caractéristiques (type et nombre des arguments et du résultat).
- **Héritage d'inactivation :** Il permet de redéfinir des caractéristiques effectives en les rendant abstraites.

Héritage de modèle : Ce type d'héritage reflète les relations "est-un" entre abstractions du modèle et se decline sous quatre formes :

- **Héritage de sous-type :** Il correspond à l'héritage fait entre une super-classe A abstraite et sa sous-classe B. A et B représentent un ensemble d'objets externes A' et B' tel que

B' et un sous-ensemble de *A'*. L'ensemble modélisé par tout autre sous-type de *A* est disjoint de *B'*.

Héritage de vue : Cet héritage permet de classer les instances de *A*. Les ensembles des instances de *B* et de *C* ne sont pas disjoints. *A*, *B* et *C* sont abstraites.

Héritage de restriction : Il définit le fait que les instances de *B* vérifient une contrainte supplémentaire. Cette contrainte devrait être ajoutée à l'invariant de *B*. *A* et *B* peuvent être toutes deux abstraites ou concrètes.

Héritage d'extension : Il définit le fait que *B* ajoute de nouvelles caractéristiques non présentes et non applicables à *A*.

Héritage Logiciel : Ce type d'héritage exprime des relations au sein du logiciel, sans contrepartie évidente dans le modèle. On en distingue quatre sortes :

Héritage de concrétisation : Ce type d'héritage définit le fait que *B* concrétise certaines caractéristiques de *A*.

Héritage de structure : *A* factorise de manière abstraite une propriété particulière que *B* doit posséder (par exemple *COMPARABLE, NUMERIC*).

Héritage d'implémentation : Il définit le fait que les caractéristiques héritées de *A* par *B* permettent d'implémenter l'abstraction associée à *B*. *A* et *B* doivent être concrètes et l'ensemble de leurs caractéristiques ne contient ni des attributs constants, ni des fonctions à exécution unique.

Héritage de service : *A* fournit à *B* un ensemble de caractéristiques liées logiquement entre elles.

Héritage de constante : Les caractéristiques de *A* ne sont que des constantes ou des fonctions à exécution unique décrivant des objets partagés.

Héritage de machine : Les caractéristiques de *A* sont des routines.

3.3.3 Taxinomie de Xavier Girod ([GIR91])

Xavier Girod présente dans sa thèse une taxinomie proche (mais aussi plus simple) de celle de Bertrand Meyer. Une différence intéressante apparaît cependant. Il définit le concept d'*héritage combiné* qui représente l'utilisation de plusieurs relations d'héritage spécifiques.

En effet, en cas d'héritage multiple, il peut être intéressant de noter que certaines relations n'ont de sens que dans leur globalité. L'exemple mis en avant est celui d'une classe *PileFixe* possédant les liens d'héritage de sous-type et d'héritage d'implémentation[5] respectivement aux classes *Pile* et *Tableau*.

3.3.4 Application à notre sujet

Nous allons maintenant proposer un tableau représentant les problèmes associés à chacun des types d'héritage utilisé.

Définitions préliminaires Nous noterons *G* la classe dont nous souhaitons modifier la cible d'héritage. Cette cible sera soit valuée à la classe *A* (cas de la redéfinition de la cible d'héritage), soit contrainte à être au moins de type *A* (cas de la paramétrisation de la cible d'héritage).

Nous noterons *D* la nouvelle cible d'héritage que nous souhaitons spécifier. Notons que *D* est une sous-classe de *A*.

H sera l'ensemble de tous les héritages utilisés (suivant la taxinomie définie précédemment) entre les classes *D* et *A*.

h représente l'héritage spécifié entre *G* et *A*. Pour certains exemples, nous ajouterons aussi une classe *B*, sous-classe de *A*, et super-classe de *D*.

Vous pourrez vous reporter à la figure 3.6 qui synthétise toutes ces définitions.

[5] Nous utilisons ici les termes de la taxinomie de Bertrand Meyer.

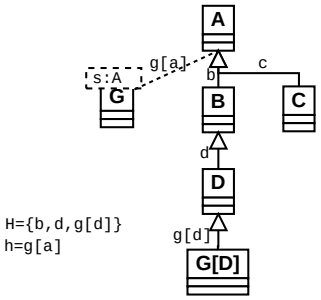

FIG. 3.6 – Définitions préliminaires.

3.3.5 Tableau des problèmes

Le tableau 3.1 représente les problèmes rencontrés en fonction de l'utilisation des différents types d'héritage. Nous nous plaçons dans le cas de l'utilisation des relations d'héritage définies par Bertrand Meyer.

I1 : Une incohérence peut apparaître, amenant une modification du type d'héritage. En effet, G possède un lien d'héritage de variation fonctionnelle avec sa super-classe, alors que D peut inactiver (c'est-à-dire rendre abstraite) les caractéristiques. Si nous spécifions D comme super-classe de G, il y a un risque que G n'ait pas la même relation d'héritage que celle qu'il avait avec A : G ne peut redéfinir le corps d'une caractéristique abstraite sans la concrétiser.

I2 : Ici se pose la question de savoir si l'on autorise l'inactivation d'une méthode déjà inactivée. En effet, G, dans sa relation d'héritage avec A, inactive une caractéristique de A. La spécification de D, qui elle aussi inactive une caractéristique de A, comme nouvelle super-classe à G peut amener cette dernière à inactiver une caractéristique déjà inactivée.

I3 : G possède un lien d'héritage dont la cible doit être concrète. D met en jeu des relations qui peuvent la rendre abstraite donc définir cette dernière classe comme super-classe de G pourrait être incohérent.

I4 : G possède un lien d'héritage de concrétisation vers la classe cible A ; mais D aussi. Ainsi G, lors de la modification de sa cible d'héritage vers D, peut tenter de concrétiser une caractéristique déjà effective.

I5 : G possède un lien d'héritage dont la cible doit être abstraite. D met en jeux des relations qui peuvent la rendre concrète; donc définir cette dernière classe comme super-classe de G pourrait être incohérent.

I6 : A doit être concrète pour G, mais abstraite pour D. Cependant cette configuration est tout de même possible : une classe B, super-classe de D et sous-classe de A, peut rendre abstraite des caractéristiques de A, cela montre ainsi la difficulté de spécifier des contraintes sur l'utilisation de différents types d'héritage.

I7 : Cela semble incohérent. Le rôle des classes en présence semble incompatible avec les différentes relations que l'on souhaite leur attacher.

I8 : Les ensembles des instances des classes G et D devraient être disjoints.

I9 : A doit être abstraite pour G, mais concrète pour D.

3.3.6 Discussion

Le tableaux 3.1 montre un ensemble d'incohérences ou de problèmes pouvant apparaître lors de la modification de la cible d'héritage de la classe G.

Les cases comportant une annotation d'incohérence ne spécifient pas que les problèmes apparaîtrons obligatoirement, mais plutôt montrent des points où la démarche, sans même penser à l'implémentation, introduit un risque d'incohérence. Par exemple, pour le cas d'une utilisation d'un héritage d'extension dans les deux héritages en présence (héritage de G à A et de D à A, cf. la case

h \ ∃x∈H | x est	HV[a] fonctionnelle	HV type	HV inactivation	HL concrétisation	HL structure	HL implémentation	HLS constante	HLS machine	HM sous-type	HM vue	HM restriction	HM extension
HV fonctionnelle	assertion	covariance	I1								assertion	
HV type	assertion	covariance									assertion	
HV inactivation	assertion	covariance	I2								assertion	
HL concrétisation	assertion	covariance		I4	I3	I3 I9					assertion	
HL structure	assertion	covariance		I5	I7 I3	I3 I9	I7				assertion	
HL implémentation	assertion	covariance	I3	I6	I7	I7	I7	I6	I6		assertion	
HLS constante	assertion	covariance	I3	I6	I7	I7					assertion	
HLS machine	assertion	covariance			I7	I7					assertion	
HM sous-type	assertion	covariance				I7	I7			I8	assertion	
HM vue	assertion	covariance				I7	I7				assertion	
HM restriction	assertion	covariance									assertion	
HM extension	assertion	covariance									assertion	conflits de nom

TAB. 3.1 – Tableaux des problèmes potentiels suivant les relations d'héritages mises en jeux.

[a] HV, HL, HLS et HM représentent respectivement l'Héritage de Variation, l'Héritage Logiciel, l'Héritage Logiciel de Service et l'Héritage de Modèle.

en bas à gauche du tableaux 3.1), le problème du conflit de nom indiqué n'est pas obligatoire (G et D peuvent très bien ajouter des propriétés avec des noms différents). Cependant, la démarche même d'étendre les interfaces présente un risque potentiel.

De même les cases ne comportant pas d'annotation d'incohérence ne préviennent pas de problèmes possibles. En effet, l'héritage de variation fonctionnelle n'empêche pas l'extension de l'interface[6] et donc pourrait aboutir à des conflits de nom. Cependant, la démarche de modifier l'implémentation des propriétés n'implique pas de problèmes de conflits de nom.

Il s'avère aussi que dans le cas bibliothèques industrielles, plusieurs types d'héritage sont utilisés en même temps lors d'un héritage entre classe, multipliant ainsi les problèmes possibles.

Toutes ces incertitudes proviennent d'un manque d'une sémantique claire des différents types d'héritage; mais surtout d'un manque de contraintes fortes sur ces derniers. Les contraintes ne doivent pas être au niveau du modèle (ce que veut dire l'héritage, c'est à dire héritage de sous-type, de restrictions ...) mais du niveau de l'implémentation (ce que permet de faire cet héritage concrètement (ajout de méthode et avec quelle visibilité, puis-je modifier les assertions ...)

3.4 Pertinence et aperçu d'une autre classification

Il semble donc nécessaire de contraindre l'utilisation faite au travers de la relation d'héritage. Il nous faut donc un moyen de désigner ce qu'il est possible de faire pour une classe source vis-à-vis de sa super-classe.

Nous avons dressé une autre classification de ce qu'il est possible de faire au travers de la relation d'héritage.

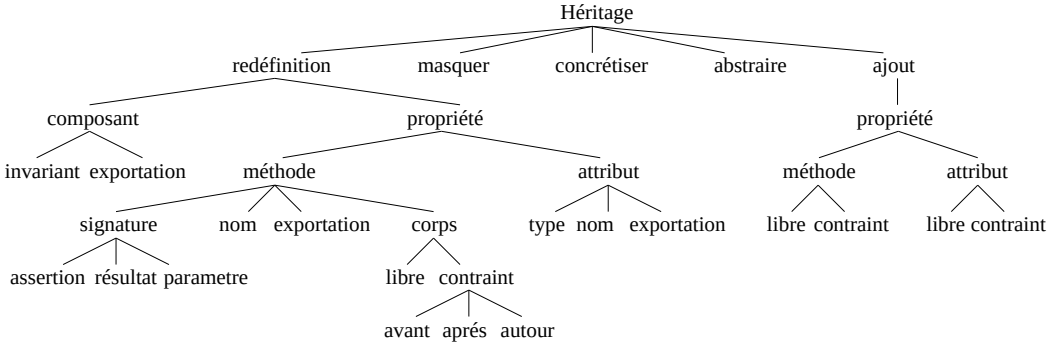

FIG. 3.7 – Taxinomie des possibilités offertes au travers de la relation d'héritage.

Nous pouvons voir sur la figure 3.7 un certain nombre d'actions possibles offertes lors de l'utilisation de la relation d'héritage. Par exemple, la capacité de redéfinir le nom d'une méthode ou encore le corps de celle-ci. De même il peut être possible, suivant le langage, de redéfinir la signature des propriétés, ou encore à renommer les méthodes.

Ce genre d'information est beaucoup plus pertinent pour contraindre notre composant de façon à ce qu'il garantisse une certaine intégrité lors de son utilisation au sein de notre relation d'héritage générique.

En effet, avec cette classification nous pouvons demander à un utilisateur de spécifier ce qu'il fait lors de son utilisation de la relation d'héritage. Ceci est inspiré du langage Eiffel, qui, pour pouvoir redéfinir une propriété, c'est-à-dire pour changer son implémentation ou son algorithme, demande l'introduction explicite du mot clef *"redefine"* dans les clauses d'héritage[7].

Mais surtout, nous pouvons maintenant donner la possibilité à une classe de limiter les actions possibles de ses futures sous-classes. Cette première pourrait interdire la redéfinition de propriétés

[6] Il est à préciser que si l'héritage de variation fonctionnelle peut interdire l'extension de l'interface offerte par la relation d'agrégation (ou de composition), il n'interdit pas l'extension de l'interface offerte pas la relation d'héritage.

[7] En fait ce n'est pas réellement aussi strict et de nombreuse exceptions apparaissent permettant tout autant de redéfinir une caractéristique.

(cas du *final* de Java ou de *frozen* de Eiffel) mais aussi interdire (ou autoriser) la redéfinition des assertions ou des signatures en spécifiant un ensemble d'action interdites (ou autorisées) pour ses sous-classes.

3.5 Synthèse

Nous avons proposé les grandes lignes d'une approche pour modifier la cible d'une relation d'héritage, soit par la paramétrisation de cette cible (héritage générique), soit par l'utilisation d'un méta-langage permettant de manipuler les entités du langage sous-jacent (redéfinition de la cible). L'ajout de cette possibilité à engendré des problèmes qu'il nous faut résoudre, à savoir des conflits de nom, des incohérences concernant la redéfinition de méthodes ou l'emploi d'assertions.

Notre étude à montré que l'utilisation de classifications comme celle de Bertrand Meyer ou Xavier Girod, pour exprimer *pourquoi* une classe hérite d'une autre, n'est pas suffisante pour prévenir ces problèmes. Il est important de présenter en supplément un moyen de spécifier plus précisément l'ensemble des adaptations qu'une classe source peut opérer sur les caractéristiques ou sur le contenu de la classe cible. C'est pourquoi nous proposons une classification précise (mais extensible) des adaptations qui peuvent être envisagées[8]. Par exemple comme l'indique la figure 3.7, il est possible de limiter les redéfinitions de méthode de telle manière à ce que la classe source ne puisse que rajouter (après l'appel au code original), un traitement particulier[9].

Grâce à l'utilisation de cette classification il sera donc possible de limiter les adaptations que la classe source pourra réaliser, par rapport aux possibilités offertes par le langage utilisé. Ces limitations pourront être imposées lors de l'écriture de la classe cible ou pourront être librement spécifiées par le concepteur de la classe source. Nous défendons l'idée que l'expression de ces limitations offrent un support intéressant pour résoudre les problèmes mentionnés dans la section 3.2. Nous proposons d'encapsuler la spécification de ces limitations dans le concept d'*annotation de l'héritage*.

Nous avons proposé, dans [PC03], une définition plus précise du concept d'*annotation* présenté rapidement dans ce rapport. Nous y avons aussi mentionné d'autres apports potentiels ainsi que des exemples pratiques d'utilisation.

[8] Naturellement suivant le langage utilisé, certaines adaptations ne pourront être mises en oeuvre.

[9] Cela correspond au noeud *Héritage/redéfinition/propriétés/méthodes/corps/contraint/après* de la classification.

Chapitre 4

Conclusion et perspective

L'objectif de ce travail est d'étudier la possibilité d'introduire un mécanisme permettant de rendre une hiérarchie de classes plus réutilisable, c'est à dire d'améliorer ses capacités d'adaption et d'évolution mais en préservant sa robustesse et sa fiabilité. Nous avons choisi de nous intéresser à cette problématique dans le cadre d'une extension du mécanisme d'héritage des langages à objets.

Notre étude de l'état de l'art a montré que des travaux significatifs étaient menés dans ce domaine. En particulier il nous a semblé intéressant de nous inspirer des approches basées sur

- la généricité horizontale : Elle doit s'intégrer de manière homogène avec notre proposition qui concerne la généricité verticale.
- les méta-informations : la définition de notre mécanisme d'annotations repose sur cette technologie.
- les *mixins* et les *templates* C++ : l'idée de paramétrer la cible de l'héritage est fortement présente dans ces deux approches et la notion de patron de relation par analogie avec la notion de patron de classe (*template*) nous semble intéressante.
- la programmation par aspects : le fait de pouvoir adapter une hiérarchie de classes de manière à ce qu'elle intègre désormais une nouvelle fonctionnalité orthogonale est particulièrement important. Notre solution doit proposer cette facilité.
- les *classes virtuelles* (Beta) et les *before/after* de CLOS : ces concepts permettent d'assurer que la redéfinition du corps d'une méthode pourra être contraint. Nous avons intégré cette facilité dans la classification associée à notre mécanisme d'annotations. Par ailleurs les idées developpées semblent un bon support pour décrire des patrons de classes et de méthodes.

Notre positionnement par rapport à l'état de l'art est évident ; nous cherchons à proposer une solution homogène privilégiant les applications industrielles en nous inspirant des travaux existants. Cependant notre étude a montré que l'adjonction d'un mécanisme permettant de modifier la cible de l'héritage induit des problèmes (voir section 3.5). C'est pourquoi notre approche intégre un mécanisme basé sur des annotations qui veut offrir un support pour faire plus de contrôle et ainsi prévenir les cas qui posent problème en vue par exemple de les interdire. Même si cette solution peut ne pas apparaître idéale (elle restreint l'utilisation de la généricité), elle a l'avantage de garantir la fiabilité et la robustesse de l'application.

A l'issu de ce stage un certain nombre de questions restent ouvertes et n'ont été que partiellement abordées ou étudiées.

- Est-il possible d'utiliser à plusieurs endroits d'une même application des instanciations différentes d'une même classe générique ? Dans ce cas comment règle t-on la compatibilité des objets ?
- Est-il préférable d'opter pour l'introduction d'un mécanisme permettant de redéfinir la cible d'une relation d'héritage ou bien est-il opportun d'étendre la généricité aux relations d'héritage ?
- A quel endroit doit être décrite le choix de la classe cible, que ce soit par instanciation ou redéfinition ? à l'intérieur de l'application ou bien à l'extérieur, par exemple à l'aide d'un langage de configuration ?

– Quelles doivent être les capacités d'adaptation mises à disposition de l'architecte de l'application lorsque ce dernier choisit la cible effective d'une relation d'héritage ?
– ...

La réponse à ces questions demande en particulier que l'on réfléchise à la plate-forme de mise en oeuvre mais aussi que l'on étende l'examen de l'état de l'art aux langages de configuration afin de voir comment ces derniers pourraient intégrer des capacités d'adaptation.

Une perspective intéressante pour ce travail serait donc d'étudier une solution complète dans le cadre d'un langage à objets comme Eiffel. En particulier, il pourra être intéressant de proposer i) une solution qui étende Lace[1], ii) une intégration de notre mécanisme d'annotation comme nous avons commencé à le faire dans [PC03], iii) un moyen de définir des patrons de méthodes ce qui permettra d'augmenter l'expressivité de la solution et ainsi de la rapprocher de celles basées sur la séparation des préoccupations.

[1] Lace signifie : Language for Assembling Classes in Eiffel.

Bibliographie

[asp] Aspectj, le site officiel. www.aspectj.org.

[CHA92] Craig CHAMBERS. Object-oriented multi-methods in cecil. In *ECOOP'92*, 1992.

[COO89] William COOK. A proposal for making eiffel type-safe. In *ECOOP 89*, 1989.

[CRE01] Pierre CRESCENZO. *OFL : Un modèle pour paramétrer la sémantique opérationelle des langages à objets. Application aux relation inter-classes.* PhD thesis, Université de Nice-Sophia Antipolis - UFR Sciences, 2001.

[DA99] Elena ZUCCA Davide ANCONA, Giovanni LAGORIO. Jam, theory and practice of a java extension with mixins. Technical report, University of Genova, 1999.

[DCH87] Patrick D. O'BRIEN Daniel C. HALBERT. Using types and inheritance in object-oriented languages, 1987.

[DD01] Ching-Ching TECHAUBOL Dominic DUGGAN. Modular mixin-based inheritance for application frameworks. In *OOPSLA 01*, 2001.

[DGB86] Gregor KICZALES Larry MASINTER Mark STEFIK Frank ZSYBEL Daniel G. BOBROW, Kenneth KHAN. Commonloops : Merging lisp and object-oriented programming. In *OOPSLA'86 ACM SIGPLAN*, 1986.

[DU91a] Bay-Wei CHANG Urs HÖLZLE David UNGAR, Craig CHAMBERS. Self : The power of simplicity. In *LISP and symbolic commputation*, 1991.

[DU91b] Randall B. SMITH David UNGAR. Self : The power of simplicity. In *LISP and symbolic commputation*, 1991.

[DUC02] Roland DUCOURNAU. "real world" as an argument for covariant specialization in programming and modeling. In Springer, editor, *Advances in Object-Oriented INFORMATION Systems, OOIS 2002 Workshops*, pages 3–12, September 2002.

[EG96] Ralph JOHNSON John VLISSIDES Erich GAMMA, Richard HELM. *Design Patterns.* Addison-Wesley, 1996.

[EVE97] Mark EVERED. Unconstraining genericity. Technical report, University of Ulm, 1997.

[FAQ] Faq beta. http ://www.faqs.org/faqs/beta-language-faq/.

[GB90] William COOK Gilad BRACHA. Mixin-based inheritence. In *OOPSLA 90*, 1990.

[GIR91] Xavier GIROD. *MECANO : une méthode et un environement de construction d'applications par objets.* PhD thesis, Université Joseph Fourier, Grenoble, 1991.

[GK] Jim HUGUNIN Mik KERSTEN Jeffrey PALM William G. GRISWOLD Grego KICZALES, Erik HILSDALE. An overview of aspectj.

[GK97] Anurag MENDHEKAR Chris MAEDA Cristina VIDEIRA LOPES Jean Marc LOINGTIER John IRVIN Gregor KICZALES, John LAMPING. Aspect-oriented programming. In *ECCOP'97*, 1997.

[GM91] Dominique COLNET Daniel LEONARD Karl TOMBRE Gerald MASINI, Amedeo NAPOLI. *Les langages à objets.* iia, 1991.

[HÖL93] Urs HÖLZLE. Integrating independently-developed components in object-oriented languages. In *ECOOP'93*, 1993.

[HO92] William HARRISON Harold OSSHER. Combination of inheritance hierarchies. In *OOPSLA'92*, 1992.

[JG] Guy STEELE Gilad BRACHA James GOSLING, Bill JOY. The java language specification 2sd edition. http ://java.sun.com/.

[KA03] Bertrand MEYER Karine ARNOUT. Contrats cachés en .net. In *LMO'03*, 2003.

[KO00] Günter KNIESEL Klaus OSTERMANN. Independent extensibility - an open challege for aspectj and hyperj. Exemple en AspectJ et HyperJ, 2000.

[LQ02] Philippe LAHIRE Laurent QUINTIAN. Vers une mielleure réutilisation des patrons de conception. Technical report, I3S, 2002.

[MEY] Bertrand MEYER. site web. http ://www.inf.ethz.ch/personal/meyer/.

[MEY86] Bertrand MEYER. Genericity versus inheritance. *OOPSLA 86*, 1986.

[MEY92] Bertrand MEYER. *Eiffel : le langage*. InterEditions, 1992.

[MEY97] Bertrand MEYER. *Object-Oriented Software Construction 2 edition*. Prentice Hall, 1997.

[MF98] Matthias FELLEISEN Matthew FLATT, Shriram KRISHNAMURTHI. Classes and mixins. *POPL*, 1998.

[MO97] Philip WADLER Martin ODERSKY. Pizza into java : Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.

[NS02] Oscar NIESTRASZ Nathanael SCHAERLI, Stephane DUCASSE. Classes = traits + states + glue. beyond mixins and multiple inheritance. In *The Inheritance Workshop ECOOP 2002*, 2002.

[OLM89] Birger MOLLER-PEDERSEN Ole Lehrmann MADSEN. Virtual classes : A powerful mechanism in object-oriented programming. 1989.

[PC03] Philippe LAHIRE Pierre CRESCENZO, Christophe JALADY. Annotations des classes et des relations d'héritage : un mécanisme unifié pour améliorer les facteurs de qualité des bibliothèques de classes. Soumission à MASPEGHI 2003, 2003.

[piz] The pizza language, an extension to java. http ://pizzacompiler.sourceforge.net/.

[SAK02] Markku SAKKINEN. Exheritance-class generalisation revived. Technical report, University of Jyväskylä, 2002.

[SKA93] John Max SKALLER. Mixin software technology. Article 20339 of comp.lang.c++, 1993.

[STE90] Guy L. STEELE. *Common Lisp the Language, 2nd Edition*. Thinking Machines, 1990.

[STR92] Bjarne STROUSTRUP. *Le langage c++ 2eme édition*. Addison-Wesley, 1992.

[TAI96] Antero TAIVALSAARI. On the notion of inheritance. *ACM computilng surveys*, 28(3), 1996.

[TL94] Munid QUTAISHAT Ted LAWSON, Christine HOLLINSHEAD. The potential reverse type inheritance in eiffel. *TOOLS Europe 94*, 1994.

[TL95] Pierre COINTE Thierry LEDOUX. Les métaclasses explicites comme outil pour améliorer la conception des bibliothèques de classes, 1995.

[WLH95] Cristina VIDEIRA LOPES Walter L. HURSCH. Separation of concerns. Technical report, College of Computer Scinece, Northeastern University, 1995.

[YS98] Don BATORY Yannis SMARAGDAKIS. Implementing reusable object-oriented components. Technical report, University of Texas at Austin, 1998.