Annotations des classes et des relations d'héritage : un mécanisme unifié pour améliorer les facteurs de qualité des bibliothèques de classes

Pierre Crescenzo, Christophe Jalady et Philippe Lahire

Laboratoire I3S UNSA/CNRS - Projet OCL
Les Algorithmes Bât. Euclide B - 2000, route des Lucioles
BP 121 - F-06903 Sophia-Antipolis cedex - France
E-mail: {Prénom.Nom}@unice.fr

Abstract

Nous proposons ici de montrer l'utilité de l'ajout d'annotations dans les classes pour mieux préciser l'usage des relations d'héritage dans des bibliothèques de classes. La valeur ajoutée attendue est d'améliorer la documentation, la réutilisabilité, les capacités d'évolution et la robustesse de ces bibliothèques. Typiquement ces annotations pourront être exploitées par les environnements de programmation. Nous nous appuyons sur des taxonomies d'héritage existantes pour montrer l'apport de ces annotations et nous définissons un mécanisme unifié et flexible capable de prendre en compte une quelconque autre taxonomie. Nous proposons une intégration de ce mécanisme dans le langage Eiffel à travers une extension de la clause `inherit` *et de la clause* `indexing` *du langage. les grandes lignes de la mise en œuvre basée sur le modèle OFL définie dans notre équipe sont également mises en évidence.*

1. Introduction

Les langages de programmation comme Eiffel, Smalltalk, Java, C# ou C++ offrent un support pour spécifier des classes[1], des relations d'héritage et d'agrégation (ou clientèle) plus ou moins complexes, des caractéristiques (attributs, méthodes, etc.), des règles de visibilité et de protection principalement sur les primitives, mais aussi parfois sur les classifieurs eux-mêmes.

L'expressivité de ces langages permet de prendre en compte de multiples situations sans pour autant que les choix du concepteur de l'application soient toujours spécifiés de manière explicite ou même qu'ils puissent l'être. Même si des tentatives de clarification sont faites en ce sens, c'est par exemple le cas pour l'usage de l'héritage, il reste encore beaucoup à faire dans ce domaine. Ainsi, de nombreux articles discutent des utilisations justifiées (ou non) de la relation d'héritage [10, 11].

Par exemple en Java ou en Eiffel, le mot-clé `abstract` pour le premier et `deferred` pour le second permet de souligner que le classifieur contient des définitions de routines abstraites ou retardées, selon le vocabulaire employé dans chacun des langages.

En Java, plutôt que d'utiliser une seule relation d'héritage on dispose de deux mots clés : `extends` qui décrit une relation d'héritage simple entre classes ou multiple entre interfaces, et `implements` qui décrit une relation d'héritage multiple entre une classe et des interfaces. Par ailleurs, on trouve aussi en C++ la possibilité de faire un héritage *privé* (mot-clé `private`), qui restreint l'application du polymorphisme sur les instances comme c'est normalement le cas lorsque l'on utilise le symbole `::`.

Ces approches constituent à notre sens une avancée car elles permettent au développeur d'application de mieux spécifier l'usage qu'il fait de l'héritage mais elles nous semblent encore insuffisantes. Ainsi il peut être intéressant de donner plus d'explication sur l'utilisation des mot-clés *extends* ou *implements* en Java ou du mot-clé *inherit* en Eiffel, pour ne citer que ces deux langages. De même des travaux comme ceux de [9] ou de [5] montrent qu'il est important de permettre à un classifieur de se protéger des autres entités du programme dans le but de ne pas permettre à ces entités de briser la cohérence de ses instances. Dans notre approche nous désirons mieux prendre en compte cet aspect par rapport à l'utilisation de relations d'héritage entre classifieurs.

Notre proposition consiste donc à permettre au développeur d'application, de manière facultative, d'ajouter un ensemble d'annotations à la description des classifieurs ou des relations d'héritage afin de mieux en préciser l'usage. Il est important de mentionner que ces annotations

[1] Dans la suite on préférera plutôt le terme plus général de *classifieur* utilisé en UML [15].

sont amenées à être utilisées par des compilateurs, des interprètes ou des environnements de programmation plutôt que par l'exécutif du langage. Elles n'ont pas vocation à modifier la sémantique des relations d'héritage ou plus généralement du langage. Nous défendons l'idée que grâce à cette approche, un développeur se voit offrir des moyens supplémentaires en vue d'obtenir des applications plus documentées, réutilisables, maintenables et robustes.

Nous donnons d'abord un aperçu de deux classifications concernant la relation d'héritage que nous avons trouvé dans l'état de l'art et nous proposons une troisième classification (ici dite *classification de référence*). Dans un deuxième temps nous présentons les principaux éléments qui sont utiles pour la définition d'une annotation. Dans un troisième temps nous examinerons à travers un exemple les principaux éléments de notre approche ainsi que son apport dans le processus de développement de bibliothèques. Puis nous proposons une intégration de notre approche dans le langage Eiffel[2]. Nous terminerons par l'évocation de travaux connexes et sur les perspectives à plus long terme de ce travail.

2. Classifier les informations

Dans l'état de l'art nous avons trouvé deux taxonomies de l'héritage. Il s'agit d'une part de la taxonomie de Bertrand Meyer [5] qui a pour principal objectif d'expliciter les différents cas où l'utilisation de l'héritage peut se justifier, et d'autre part, de celle proposée par Xavier Girod [7] dont l'objectif est plutôt de recenser les différentes sortes d'utilisation de l'héritage en prenant le point de vue du concepteur d'application. Nous proposons ensuite une troisième classification qui est orthogonale aux précédentes et dont l'objectif est de recenser les différentes limitations que l'on peut appliquer aux nœuds de ces classifications sans en dénaturer le sens.

2.1. Justifier l'utilisation de l'héritage

Nous reproduisons ici la taxonomie proposée par Bertrand Meyer (voir figure 1)[3] et nous rappelons brièvement le sens des différents types d'héritage en ciblant plus particulièrement les points intéressants, c'est-à-dire, ceux qui nous permettent d'envisager l'application de contrôles exhibés par cette classification. Il est utile de noter que cette classification s'applique plus particulièrement au langage Eiffel et moins à d'autres langages, même si son propos peut être généralisé avec profit.

[2]Il est à noter que notre approche pourrait bien sur s'adapter à d'autres langages mais l'intégration serait sûrement différente.

[3]Les figures sont en anglais dans ce document.

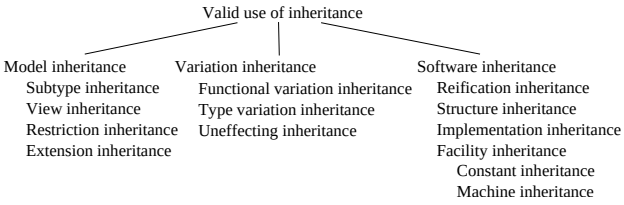

FIG. **1. Taxonomie proposée par B. Meyer**

- **l'héritage de sous-type** permet de partitionner l'ensemble des instances de la classe parente par des ensembles disjoints représentés par les sous-classes. La super-classe doit être abstraite et ne possède donc pas d'instance directe.
- **l'héritage de vue** est utilisé pour opérer une classification multicritères entre les instances. Elle produit des partitions d'instances non disjointes et s'appuie intensivement sur l'héritage multiple pour représenter les différentes *vues* d'un objet. Les classes concernées doivent êtres abstraites.
- **l'héritage de restriction** signifie que la sous-classe doit verifier une contrainte supplémentaire par rapport à la classe parente, comme par exemple un invariant. Les classes doivent être ou toutes deux abstraites ou toutes deux concrètes.
- **l'héritage d'extension** permet de spécifier que la sous-classes ajoute des caractéristiques non applicables à la super-classe. La sous-classe qui peut être abstraite ou concrète étend l'interface (c'est-à-dire l'ensemble des caractéristiques exportées) de la classe dont elle hérite et qui doit être concrète.
- **l'héritage de variation** concerne les situations où la classe héritière veut redéfinir une ou plusieurs caractéristiques de la classe parente. Il n'y a aucun ajout de caractéristiques sauf éventuellement pour le seul besoin de caractéristiques redéfinies. On distingue la variation fonctionnelle qui permet la redéfinition du corps d'une routine, de la variation de type qui ne permet que de redéfinir sa signature (type et nombre des paramètres ou type du résultat). La sous-classe doit être comme sa super-classe abstraite ou concrète.
- **l'héritage d'inactivation** revient à rendre abstraite une ou plusieurs caractéristiques de la classe parente. Il devrait être utilisé uniquement pour fusionner deux routines concrètes ayant le même nom dans le cas d'héritage multiple ou pour réutiliser une classe qui est trop concrète et dans ce cas, cela représente une certaine forme de généralisation. La classe source[4] devient naturellement abstraite.

[4]La classe source est celle qui définit la relation d'héritage et la classe cible est la classe héritée.

– **l'héritage de concrétisation** est utilisé lorsque l'on veut proposer une implémentation à des caractéristiques initialement abstraites. Ainsi la classe parente est forcément retardée tandis que la classe héritière peut devenir concrète. Dans ce type d'héritage on n'étend pas l'interface (l'ensemble des caractéristiques exportées) de la classe parente.
– **l'héritage de structure** est utilisé lorsque la classe parente représente une propriété structurelle qui peut être associée à la classe héritière, sachant que la propriété structurelle n'est qu'un aspect des fonctionnalités de la classe héritière. La classe parente est forcément abstraite car les fonctionalitées qu'elle propose sont spécifiques à la sous-classe.
– l'**héritage d'implémentation** est utilisé pour fournir à la classe héritière des fonctionnalités qui permettent de mettre en œuvre l'abstraction qui lui est associée. Ce type d'héritage nécessite que les classes parente et héritière soient toutes deux concrètes. Cette dernière peut, de plus, posseder une deuxième relation d'héritage lui permettant d'hériter du modéle (c'est la relation *est un*) de cette nouvelle super-classe.
– **l'héritage de service** est principalement utilisé permettre aux classes descendantes de bénéficier de fonctionalitées ; par exemple, on l'utilisera pour hériter d'une classe qui contient des constantes ou des fonctions à exécution unique ou pour hériter des opérations d'une machine abstraite.

2.2. Exprimer différents cas d'utilisation de l'héritage

Dans sa classification, Xavier Girod [7] propose sept utilisations de l'héritage qui se rapprochent beaucoup de la classification de Bertrand Meyer. Il définit cepandant un type d'héritage *multiple*, proposant une encapsulation des différents types d'héritage mis en évidence. Dans la figure 2, ces liens sont matérialisés par un vecteur <h_1, ..., h_n> ou h_1, .., h_n représentent des types différents (respectivement le même type), dans un **héritage combiné** (respectivement un **héritage de fusion**).

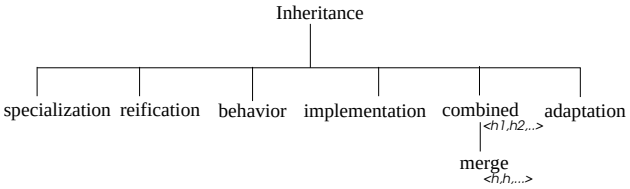

FIG. **2. Taxonomie proposée par Xavier Girod**

Ainsi l'**héritage combiné** nécessitant l'héritage multiple, permet de spécifier une dépendance entre les relations d'héritage mises en évidence. Par exemple une classe BOUNDED_STACK qui hérite de STACK par un héritage de **réalisation** (proche de l'héritage de **concrétisation** - voir figure 1) et de ARRAY par un héritage d'implantation (proche de l'héritage d'implémentation, voir figure 1) met effectivement en œuvre un **héritage combiné** : dans le cas présent l'**héritage de réalisation** est associé à l'**héritage d'implémentation**.

Lorsque les types d'héritage sont identiques, et que la sous-classe représente une sorte d'*agrégat* des classes parentes dans lequel aucune super-classe n'a une place prépondérante, cela correspond à un **héritage de fusion** (qui peut être vu comme une spécialisation de l'**héritage combiné**). Un exemple d'héritage de fusion est le cas de la classe SEAPLANE héritant par spécialisation de BOAT et de PLANE.

En plus de ces quatres relations d'héritage, X. Girod mentionne aussi **l'héritage de spécialisation**, **l'héritage de comportement** et **l'héritage d'adaptation**, respectivement proches de l'héritage d'extension, de l'héritage de structure et de l'héritage de variation proposés dans la figure 1.

2.3. Une classification dite de référence

Nous avons rappelé les principaux aspects de deux taxonomies sur l'usage de l'héritage. Nous proposons maintenant une classification plus technique dans laquelle nous recensons les adaptations élémentaires du comportement du classifieur parent qu'il est concrètement possible [5] de faire dans le classifieur source lorsqu'il hérite. Cette classification est orthogonale aux deux précédentes dans le sens qu'elle ne se base pas sur les mêmes préocupations et nous examinerons dans la section 3 comment elle peut contribuer à améliorer la qualité des bibliothèques. Elle sera nommée dans la suite *classification de référence*.

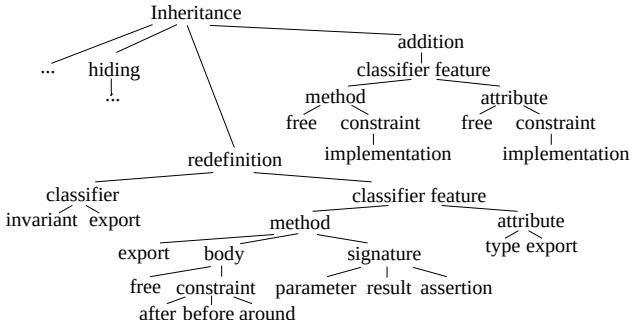

FIG. **3. Extrait de la classification de référence**

La figure 3 représente seulement un extrait de cette clas-

[5] Nous nous plaçons ici dans le contexte d'un développeur qui décrit le code de l'application.

sification, dans lequel nous nous intéressons principalement aux adaptations relatives à l'ajout et à la redéfinition de caractéristiques. Parmi les catégories d'adaptation non citées dans le schéma on peut recenser, par exemple, le renommage de caractéristiques ou l'inactivation d'un corps de méthode. Dans cette classification, chaque nœud correspond à une catégorie d'adaptation à laquelle est associée une propriété (notée *constraint*) qui peut prendre l'une des trois valeurs suivantes :

- **mandatory** : l'adaptation associée au nœud doit être appliquée pour chaque occurrence de la métaentité représentée (en général un classifieur, un attribut ou une méthode). Par exemple *toutes les méthodes de la classe doivent être redéfinies*.
- **forbidden** : aucune occurrence de la métaentité représentée ne peut effectuer cette adaptation.
- **allowed** : l'adaptation associée au nœud est possible, c'est-à-dire qu'elle peut ou pas, être appliquée sur une occurrence donnée de la métaentité.

Par exemple il pourra être intéressant de d'autoriser (*allowed*) toute les possibilités de redéfinition concernant une méthode (*inheritance/redefinition/classifier_feature/method*) ou bien d'interdire (*forbidden*) la redéfinition du corps de n'importe quelle méthode du classifieur cible (*inheritance/...method/body/free*) ou encore enfin de rendre obligatoire (*mandatory*), pour toute méthode du classifieur cible la redéfinition de son corps avec la contrainte que celle-ci soit telle qu'elle n'ajoute du code qu'après l'exécution du code d'origine de la méthode (*inheritance/.../body/constraint/after*).

De même en ce qui concerne l'ajout de caractéristiques on pourra permettre (*allowed*) seulement l'ajout de méthode (*inheritance/addition/classifier_feature/method/free*) ou bien encore ne permettre (toujours *allowed*), que l'ajout de méthode d'implémentation, c'est à dire de méthodes non exportées et utilisées seulement par le corps des méthodes redéfinies (*inheritance/.../method/constraint/implementation*).

De plus à chaque nœud correspond une ou plusieurs règles qui définissent la sémantique de l'adaptation et qui dépendent naturellement de la valeur associée à la propriété *constraint*.

Exemple de règle :

- nœud considéré : *inheritance/...method/body/free*
- pour toute relation d'héritage *R*, soit *R.S* son classifieur source et *R.T* son classifieur cible,
- soit $R.M_{ST}$ l'ensemble des méthodes de *R.S* qui sont redéfinies dans *R.T*,
- *constraint = forbidden* **implies** $R.M_{ST}$ *= ensemble vide*

La classification proposée veut être indépendante des langages. Il est donc probable que certaines adaptations n'ont pas de sens dans un langage donnée. Par exemple la redéfinition de signature n'existe pas en Java alors qu'elle fait partie des facilités offertes par le langage Eiffel.

3. Vers la définition d'une annotation

Notre objectif dans cette section est de montrer de manière intuitive comment à partir d'une part des classifications de l'état de l'art concernant l'usage de l'héritage vues plus haut (figures 1 et 2), et d'autre part de la classification dite de référence (figure 3), on pourra *i)* donner notre définition d'un nœud de classification, *ii)* mettre en évidence les besoins de personnalisation du contenu de ces nœuds et *iii)* définir ce que représente une annotation.

3.1. Définir un noeud de taxonomie

Nous proposons une manière de présenter et d'enrichir les informations associées à une classification existante (principalement celle de B. Meyer ou de X. Girod). La plupart du temps les informations qui décrivent les types d'héritage (nœuds de la classification) sont données sous forme d'un texte ou d'exemples explicatifs sujets à interprétation. Nous proposons à la fois, d'en extraire, quand c'est possible, des règles plus formelles qui définissent leur sémantique mais aussi d'organiser toutes les informations que l'on pourra recueillir en trois catégories[6] :

- **les informations d'adaptation** : elles décrivent les capacités d'adaptation du classifieur source et elle peuvent donc être décrites par un ensemble de nœuds de la classification de référence (on l'appelle la **définition initiale**). Naturellement cette description est d'autant plus subjective que les explications associées à la classification sont floues ou ambiguës et elle dépend de l'expressivité du mécanisme d'héritage du langage. Au pire on associera la racine de la classification de référence à un nœud.
- **les informations sur la structure de la hiérarchie** : Elles dépendent de la hiérarchie globale. Par exemple, dans le cas d'une relation de sous-typage, si B est un sous-type de A et C est aussi un sous-type de A, alors l'ensemble des instances de A et B sont disjointes. Ces informations correspondent à des contraintes et elles pourront être définies à l'aide de métainvariants.
- **les informations intuitives** : Elles correspondent à une description textuelle ou à des exemples de la sémantique de l'héritage. C'est dans cette catégorie que nous placerons les informations qui ne pourront pas être formalisées et qui seront donc les plus difficiles à vérifier.

[6] Il est à noter qu'un nœud de taxonomie peut éventuellement faire référence à d'autres nœuds comme c'est le cas dans la figure 2.

Dans la suite nous ne considérons que les deux premières catégories, la troisième sort du cadre de cet article. L'ensemble des règles qui seront recensées au niveau d'un nœud N de la taxonomie est égal à l'union des règles associées aux nœuds ancêtres. Ces règles peuvent être comprises comme étant des assertions de niveau méta pour l'application. On rappelle que l'ensemble des règles qui rentrent dans la définition des informations d'adaptation ont été définies dans chacun des nœuds de la classification dite de référence (voir section 2.3).

3.2. Pouvoir personnaliser une taxonomie

Savoir décrire un nœud de taxonomie comme celles des figures 1 et 2 n'est pas suffisant, il faut pouvoir le personnaliser de manière ponctuelle. Ceci provient du fait que l'on peut être capable, pour certaines relations d'héritage faisant partie de la description d'une bibliothèque de classes, de spécifier, plus finement que cela n'est fait dans la définition initiale du nœud considéré, l'usage de la relation d'héritage. Notre approche consiste à permettre au développeur de modifier lorsqu'il en ressent le besoin, le contenu des informations d'adaptation (c'est-à-dire l'ensemble des nœuds qui correspondent à la définition initiale). Nous croyons que le fait de pouvoir mettre des contraintes plus fortes sur le contenu d'un classifieur source permet d'augmenter à la fois les capacités de contrôle et d'automatisation de certaines tâches concernant la maintenance ou l'évolution de l'application. Cet apport sera rediscuté plus en détail dans la section 4.3, mais considérons dès à présent quelques personnalisations possibles de la définition initiale.

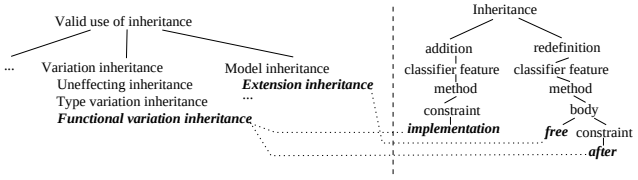

FIG. **4. Exemple de personnalisation**

Dans la figure 4 qui contient un extrait de la taxonomie de B. Meyer et de la classification dite de référence nous proposons deux exemples de personnalisation. Le premier concerne l'*héritage d'extension* qui permet au classifieur source de redéfinir librement de nouvelles méthodes, ce qui n'était pas autorisé dans la définition initiale de ce nœud.

Le second concerne l'héritage de variation fonctionnelle. Ici il est ajouté deux contraintes au classifieur source qui concernent l'ajout et la redéfinition de caractéristiques ; désormais elle ne peut :

- redéfinir une méthode qu'en l'enrichissant par du code supplémentaire placé après celui qui se trouve dans la méthode du classifieur cible à redéfinir ;
- ajouter des méthodes seulement si elles sont appelées par le corps des méthodes redéfinies dans le classifieur source et non exportées vers des classifieurs autre que lui-même ;
- ni redéfinir, ni ajouter d'autres caractéristiques au classifieur source.

Bien sûr ce ne sont que des exemples et de multiples autres personnalisations intéressantes sont possibles pour ces deux types d'héritage. Ainsi une variante pourrait être d'autoriser la redéfinition libre des méthodes au lieu de la contraindre.

Un autre exemple non décrit dans la figure 4, serait de considérer l'*héritage de concrétisation*. Il sera par exemple intéressant de le décliner soit sous une forme assez libre où la principale contrainte serait de rendre obligatoire la concrétisation de l'ensemble des primitives, soit sous des formes plus strictes qui nécessiteraient en plus

- la même interface pour le classifieur source et le classifieur cible (c'est à dire l'impossibilité d'ajouter des caractéristiques) ou,
- l'interdiction de redéfinir les caractéristiques déja concrètes dans le classifieur cible ou encore,
- l'interdiction de modifier les assertions ou d'ajouter une clause à l'invariant du classifieur ou enfin,
- d'autres types de contrainte (compatibles) que l'on pourrait exraire de la classification de référence.

3.3. Définition d'une annotation

De manière intuitive, une annotation va nous permettre d'une part d'associer un usage *U* (par exemple *variation inheritance*) à une relation d'héritage *R* qui pourrait exister entre deux classifieurs d'une bibliothèque, et d'autre part de mémoriser une redéfinition éventuelle de la définition initiale de *U*. Ainsi une annotation va contenir les informations suivantes :

- le nom associé à un nœud d'une taxonomie (par exemple *Variation Inheritance* si on considère la taxonomie de B. Meyer). Il est à noter que celui-ci contient en particulier un ensemble (sa définition initiale), constitué d'éléments de la forme *:*
  - un *nom de nœud* de la classification de référence et
  - une *valeur* parmi *allowed* / *mandatory* / *forbidden*
- un ensemble de nœuds de la classification dite de référence qui représente une redéfinition de la définition initiale de U (elle est appelée *définition redéfinie*).

La manipulation des définitions initiale et redéfinie nécessitera de disposer d'opérateurs ensemblistes mais aussi d'opérateurs permettant d'ajouter ou d'enlever des éléments à ces ensembles, de modifier la valeur (*allowed*, *mandatory* ou *forbidden*) associée à un de leurs éléments. On détaillera un peu plus la mise en œuvre dans la section 5.5, mais on peut déjà préciser que l'utilisation de ces annotations don-

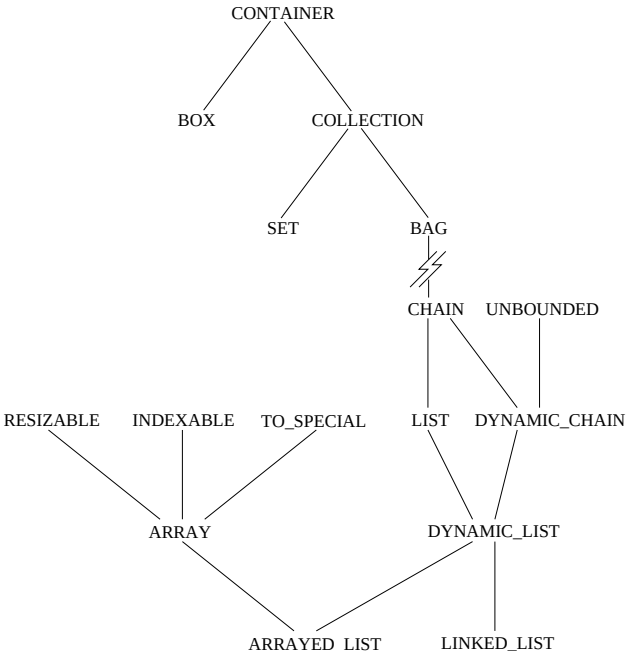

FIG. **5. Extrait d'une bibliotheque d'Eiffel**

nera lieu à la définition d'actions à réaliser en cas de succès ou d'échec de la vérification des règles (ces actions dépendent bien sur des fonctionnalités que l'on veut introduire, par exemple, dans un environnement de programmation), à travers l'utilisation des annotations. Des exemples de fonctionnalités sont données dans la section 4.

4. Apports de l'approche à travers un exemple

Nous proposons d'étudier l'apport de l'introduction d'annotations dans une bibliothèque de structures de données. Nous avons montré ci-dessus l'existence de deux taxonomies, mais d'autres peuvent évidemment sur être considérées ; leurs applications sont variables et concernent plus particulièrement l'amélioration de la documentation, de la réutilisabilité, de la robustesse ou bien encore de l'adaptabilité d'une bibliothèque de classes ou sa conception. Nous nous attachons ci-après à aborder ces différents points à travers l'exemple choisi.

La figure 5 représente un extrait de la bibliothèque des structures de données du langage Eiffel[7]. Nous remarquons deux classes en bas de la hiérarchie, *ARRAYED_LIST* et *LINKED_LIST*, qui implémentent différemment le concept de liste classique modélisé par la classe *DYNAMIC_LIST*. On notera que cette dernière se différencie de la classe *LIST*

[7]C'est plus précisément la bibliothèque de classes livrée avec Eiffel-Studio 5.3.

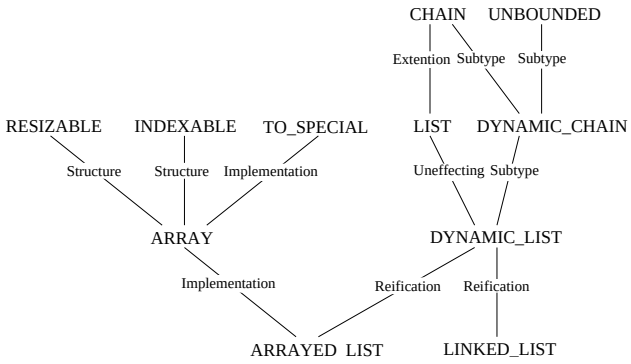

FIG. **6. Justifification de l'utilisation de l'héritage**

par la présence de caractéristiques permettant notamment d'ajouter des éléments en début et en fin de liste.

La super-hiérarchie de droite sur la figure représente la classification permettant de modéliser le concept de *CONTAINER* (structure de donnée abstraite représentant un ensemble d'éléments) et ses divers raffinements comme *COLLECTION* (qui inclut la possibilité d'ajouter ou d'enlever des éléments), *SET* (qui garantie l'unicité des éléments), *BAG* (qui au contraire laisse libre le nombre d'occurrence d'un element), *CHAIN* (séquence, circulaire ou non, d'éléments), etc.

Toujours sur la figure, la super-hérarchie de gauche modélise le concept de tableau. Nous l'avons incluse principalement pour montrer qu'elle participe à l'implémentation de la classe *ARRAYED_LIST*.

4.1. Documentation et Cohérence

La figure 6 propose une description de l'usage de l'héritage selon la taxonomie proposée dans la section 2.1. L'extraction de l'usage de l'héritage `a posteriori`, en s'appuyant sur le contenu de la bibliothèque d'origine est un travail difficile dont le résultat est sujet à polémique. Il est donc susceptible d'être remis en question comme nous le montrerons plus loin. Notre approche offre au développeur un formalisme lui permettant d'annoter une relation d'héritage pour décrire son usage en utilisant cette classification. Ce sera pour lui un moyen de justifier l'emploi de l'héritage chaque fois qu'il en a besoin. Ce travail de réflexion supplémentaire est selon nous essentiel dans le processus de réalisation d'un logiciel. Il sera d'autant plus productif qu'il est complété par la possibilité de contrôler que ces annotations sont cohérentes par rapport au code de l'application.

En effet, pouvoir justifier auprès de son chef de projet de l'emploi de l'héritage par rapport à d'autres solutions privilégiant l'emploi de relations d'agrégation ou de clientèle est

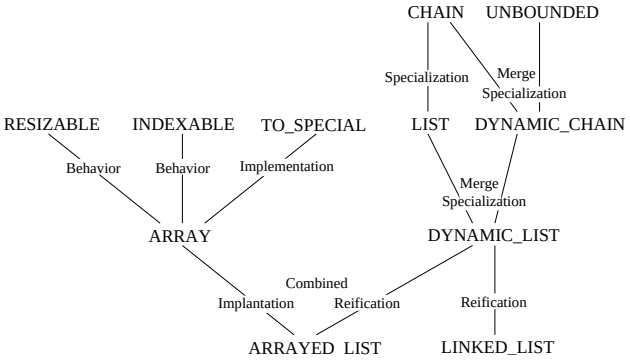

FIG. **7. Utilisation des annotations justifier les choix de conception**

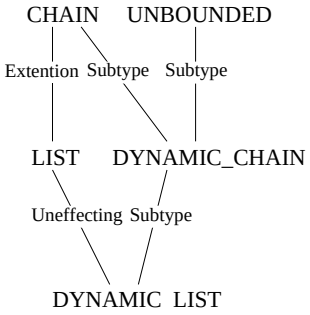

FIG. **8. Points de vues sur la bibliothŁque de classes**

bien sûr positif. mais offrir un premier niveau de contrôle, ne serait-ce que pour se persuader que les entités définies respectent bien quelques règles minimales associés aux différents types d'héritage, est aussi important. Pour cela, on s'appuie sur la vérification des règles décrivant la définition initiale de chacun des usages de l'héritage utilisés. Il sera instructif d'essayer de reconnaître les types d'héritage dans les principales bibliothèques et de vérifier qu'ils sont cohérents par rapport au code source. On pourra par exemple générer des rapports de vérification mettant en évidence les incohérences éventuelles.
La figure 7 permet d'examiner l'extrait de la bibliothèque sous un angle différent : celui de la taxonomie de X. Girod (section 2.2). Ainsi son utilisation permet de mieux adresser les problèmes liés à la conception en particulier en prenant en compte les dépendances entre plusieurs relations d'héritages. Elle favorise ainsi la mise en œuvre de processus de *reverse-engineering*. Dans la figure, on mentionne en particulier une utilisation de l'héritage combinée et deux de l'héritage de fusion. Ainsi la classe *ARRAYED_LIST* implémente les méthodes retardées de la classe *DYNAMIC_LIST* avec celles de la classe *ARRAY*. De même la classe *DYNAMIC_LIST* hérite de manière égale des fonctionnalités des classes *LIST* et *DYNAMIC_CHAIN* et elle les personnalise quand c'est nécessaire.

4.2. Points de vue selon la nature de l'héritage

La figure 8 montre un autre intérêt d'annoter l'héritage. Les annotations fournissent des informations qui pourront être exploitées pour visualiser différents points de vue d'une bibliothèque de classes et présenter une vision moins complexe de la hiérarchie. L'intérêt est évident : permettre au progammeur de se focaliser sur les relations d'héritage qui le préoccupe afin de pouvoir vérifier que l'emploi de l'héritage de sous-typage est cohérent dans le cas des classes visualisées dans la figure. Une autre utilisation de ces informations par les environnements de programmation peut être d'offrir à un développeur, qui n'a pas participé à la conception de la bibliothèque, la vision adéquate pour la comprendre progressivement et envisager plus sereinement sa réutilisation voire son extension.

4.3. Aide à l'écriture et à la mise au point

Cette section s'appuie fortement sur la composition de classifications présentée en détail dans la section 2.3. Dans la phase d'écriture et de mise au point d'une application il est important, pour le développeur, de bénéficier du maximum d'aide de la part de l'environnement de programmation. Par exemple, en fonction du type d'héritage employé, un éditeur intelligent peut assister le programmeur dans la description de sa classe. De même toujours en fonction du type d'héritage, des contrôles pourront être réalisés dans le but d'informer le développeur de possibles erreurs. Ces contrôles seront d'autant plus pertinents que ce que l'on veut faire dans le descendant sera précisément décrit par le type d'héritage employé. C'est pourquoi nous proposons de redéfinir la définition initiale associée aux nœuds des taxonomie (figures 1 et 2).

Pouvoir adapter la sémantique d'un usage de l'héritage, vraisemblablement en la contraignant un peu plus et dans tous les cas en s'assurant qu'elle garde sa pertinence, permet d'envisager l'automatisation de certaines activités de maintenance, en plus des aides à l'édition et à la mise au point. Pour ce faire, on pourra ajouter, échanger ou supprimer des éléments qui participe à sa définition initiale.

4.4. Garantir une plus grande robustesse

Un autre aspect important qu'il est possible de prendre en compte avec des annotations concerne la protection des classes vis à vis de celles qui peuvent devenir des descendantes. Bien que cet aspect de la description d'un pro-

gramme soit la plupart du temps associé à l'emploi de *modificateurs*, il peut être intéressant d'annoter une classe afin qu'elle spécifie un peu mieux ce qui risque de briser sa cohérence si d'aventure les classes descendantes l'utilisaient mal. Ces annotations seraient considérées comme des informations qui pourraient être utilisées par exemple pour générer un rapport qui synthétiserait les erreurs de conception/programmation potentielles et les risques qui en résultent pour la robustesse de l'application considérée.

Parmi les contraintes qu'une classe peut souhaiter voir appliquer par des descendants on peut citer :

- l'utilisation par la classe héritère de seulement certains types d'héritage pour l'atteindre,
- l'acceptation seulement de certains types de classifieur comme héritier,
- etc.

5. Intégration des annotations pour Eiffel

Le choix de proposer ici une intégration dans le le langage Eiffel plutôt qu'un autre comme Java par exemple, est dicté par plusieurs constatations :

- Java propose plusieurs relations d'héritage identifiées explicitement par des mots clés différents (*extends*, *implements*) ou implicitement par le type de classifieur de la classe cible (*interface*, *class*). Eiffel propose un mécanisme unique. En proposant une intégration pour Java nous risquons donc de rendre confuse une première explication de l'approche en donnant l'impression de faire partiellement double emploi.
- Les deux classification trouvées dans l'état de l'art reposent partiellement sur des facilités offertes par Eiffel (héritage multiple, covariance, mécanisme évolué d'assertions, etc.).

5.1. Annotation de l'héritage : mécanisme de base

Le langage Eiffel propose déjà une clause d'indexation décrite principalement avant la description d'une classe ; elle permet de donner des informations complémentaires sur la classe qui sont destinées à être utilisées par des outils externes ou par l'environnement de programmation. Ces informations sont décrites à l'aide de plusieurs lignes indépendantes constituées par un nom indiquant le type de l'information, le caractère " :" et une chaine caractères qui contient soit une phrase de commentaire ou d'explication, soit une séquence de mots séparés par des virgules.

exemple :

```
description : "cette classe ...."
keywords : "mot1, mot2, ..."
...
```

La liste des types d'information (ici *description* et *keywords*) est libre et peut être étendue à volonté par le développeur de la classe. Naturellement il est particulièrement important qu'il y ait une uniformisation de leur description au moins à l'intérieur d'une même bibliothèque, car elle permet d'envisager la mise en œuvre de recherches systématique.

Notre approche intègre la notion d'annotation à travers l'extension de la clause d'héritage (*inherit*) avec une clause d'indexation (*indexing*). La déclaration ci-dessous définit un exemple d'utilisation. On peut noter que le choix d'ajouter cette information dans la clause *indexing* plutôt que par l'intermédiaire d'autres ajouts syntaxiques insiste sur notre volonté de ne pas modifier la sémantique du langage et de faire en sorte que ces informations soient utilisées par des outils externes. De même que pour la clause *indexing* de classe en Eiffel, la clause *indexing* d'héritage est facultative.

Dans cet exemple, la clause *indexing* précise que la classification utilisée est celle décrite dans la figure 1 et que le nœud choisi dans la classification représente un héritage de concrétisation.

```
class LINKED_LIST [G]
inherit
 DYNAMIC_LIST [G]
  rename
    ....
  redefine
    ....
  indexing
   taxonomy : "valid_use_inherit"
   use : "reification_inheritance"
  end
...
end -- class
```

5.2. Contraindre une annotation de l'héritage

On se trouve dans l'exemple ci-dessous dans le même contexte que le précédent mais on désire indiquer dans cette classe plusieurs limitations ponctuelles en se servant du contenu de la classification de référence (figure 3). Pour éviter d'alourdir la classe avec le chemin complet des nœuds comme cela est fait dans la section 2.3 on a préféré donner ici un nom qui se veut le plus significatif possible. Il est à noter que les opérateurs **+**, **-** et **=** signifient respectivement ajouter, enlever et remplacer un élément de la définition initiale. Dans le schéma ci-dessous, le programmeur dit, à travers leur évocation, qu'il permet de redéfinir de manière contrainte le corps des routines (il était libre auparavant, il doit comporter désormais avant le code ajouté, l'appel à la version originale de la méthode), qu'il est possible d'ajouter des attributs ou des méthodes d'implémentation qui ne doivent pas être exportées vers d'autre classes et que désormais il faut rendre effective toutes les méthodes (auparavant c'était facultatif). Les lettres **A**, **F** et **M** corres-

pondent respectivement à *allowed*, *forbidden* et *mandatory* (voir section 2.3).

```
indexing
   ...
  taxonomy_inherit : "valid_use_inherit"
class LINKED_LIST [G]
inherit
 DYNAMIC_LIST [G]
 ....
 indexing
  use : "reification_inheritance"
  redefinition : "- method_redefinition_body
              + method_redefinition_after (A),
              + add_implementation_feature (A),
              = method_body_definition (M)"
 end
...
end -- class
```

On notera que dans le cas où la clause *indexing* associée à chaque relation d'héritage qui est annotée est relative à la même classification, il pourra être intéressant de donner cette information dans la clause *indexing* de la classe afin de ne pas répéter plusieurs fois la même information.

5.3. Autoriser l'usage d'héritages composés

Notre mécanisme s'adapte aussi à l'utilisation d'autres classifications comme celle décrite dans la figure 2 qui veulent exprimer que plusieurs relations d'héritage sont nécessaires pour exprimer un besoin. C'est pourquoi on spécifie un nom différent pour *taxonomy_inherit*.

```
indexing
   ...
 taxonomy_inherit : "use_of_inherit"
class ARRAYED_LIST [G]
inherit
  DYNAMIC_LIST [G]
  ....
   indexing
     use : "realisation"
     super_type_of_inherit : "combined_inherit 1"
  end
  ARRAY [G]
   ....
   indexing
    use : "implementation"
    encompassed_type : "combined_inherit 1"
  end
...
end -- class
```

Dans l'exemple ci-dessus on spécifie le type d'héritage utilisé pour réutiliser les classes DYNAMIC_LIST et ARRAY qui sont des héritages respectivement de réalisation et d'implémentation. Cependant on indique aussi que l'héritage de ces deux classes participe à la mise en œuvre d'un héritage plus complexe appelé *combined_inheritance*. Le nombre ajouté (ici *1*) est facultatif. Il ne sert qu'à éviter toute ambiguité en cas d'utilisation de plusieurs héritages combinés.

5.4. Annotation de classifieurs

Il peut être intéressant de spécifier dans une classe que son éventuel héritage par une autre classe doit vérifier un ensemble de contraintes. Par exemple, comme le propose l'exemple ci-dessous, qu'une classe n'accepte d'être la cible que de relations d'héritage correspondant à des usage précis.

```
indexing
  ...
 inheritance_restrict : "type_variation_inherit,
                         functional_variation_inherit"
class ARRAYED_LIST [G]
inherit
   DYNAMIC_LIST [G]
   ARRAY [G]
...
end -- class
```

La classe ARRAYED_LIST n'autorise que des héritages de variation, de type ou fonctionnelle. On notera que pour contraindre les descendants d'une classe à vérifier certaines propriétés, il n'est pas nécessaire d'annoter l'héritage dans la classe elle-même ; toutefois dans le but de pouvoir toujours mieux contrôler, nous le recommandons.

5.5. Mise en oeuvre du mécanisme d'annotation

Mettre en œuvre notre approche nécessite de pouvoir décrire tout ce qui concerne le mécanisme d'annotation et plus particulièrement les règles et les contraintes qui permettent de caractériser chacun des nœuds de la classification de référence. Ces règles s'appuient sur la réification d'une application comme par exemple les méthodes ou les attributs d'une classe, la liste des méthodes redéfinies, etc. Nous voulons réaliser cette mise en œuvre à partir de la bibliothèque OFL/J qui offre une réification à objets complète[8] de l'ensemble des entités d'une application et qui est équipée de fonctionnalités pour intégrer de nouvelles entités. L'extension du modèle OFL définie dans [13] permet de définir des métaassertions en s'appuyant sur la réification proposée par le modèle. Les réifications d'une annotation et de tous les éléments nécessaires à la description d'une taxonomie seront ajoutées à la bibliothèque OFL/J. Pour plus de simplicité pour celui qui est chargé de décrire les classifications, l'ensemble des métaassertions et des règles pourront être décrites en OCL (Object Constraint Language - UML) [15], et ensuite transformées pour être intégrées à la réification de l'application.

La génération de l'ensemble des objets participant à la réification pourra être effectuée par un outil externe inclus dans l'environnement de programmation ou dans le compilateur. Naturellement pour des raisons de performance on

[8] Cette bibliothèque est une première implémentation du métamodèle OFL et est écrite en Java.

ne réifiera que les informations strictement nécessaires à l'exploitation des annotations. Des actions à réaliser ont été mises en évidence dans la section 4 mais la liste n'est pas exhaustive (génération de rapport, affichage suivant différents points de vues, contrôles de cohérence, etc.), pourront être intégrées par l'environnement de programmation en utilisant des approches par séparation des préoccupations ou simplement le patron de conception *visiteur* [14].

6. Travaux connexes

Tout d'abord [8] 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 pas pouvoir posséder de sous-classes. L'article se base sur le langage SmallTalk, auquel on a ajouté un protocole metaobjet : ClassTalk. D'autres caractéristiques sont introduites comme la possibillité de spécifier l'ensemble de méthodes à redéfinir par les sous-classes ou d'empécher la modification de l'interface. Elles sont encapsulées dans des métaclasses. C'est à chaque classe de préciser de quelle métaclasse elle est une instance. Ces caractéristiques influent sur les futures relations d'héritage entre une classe (instance d'une métaclasse donnée) et de ses (futures) sous-classes. D'une certaine manière, l'article propose à une classe de pouvoir contraindre les futures relations d'héritage dont elle sera la cible.

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

Tous les travaux relatifs au *reverse-engineering* se préoccupent des problèmes de maintenance et d'évolution d'une hiérarchie de classes. Les travaux de P. Clarke et B. Malloy [12] présentent une taxonomie de classes pour représenter les modification faites dans différentes versions d'un composant. Ainsi, ils proposent un ensemble de propriétés qui caractérisent les classes, comme le fait d'être générique, abstraite, de représenter un thread/processus, ou bien concernant ses méthodes, de lancer des exceptions, ou de déclarer des protections. Cette taxonomie permettra d'instancier un modèle sur différentes versions des composants afin d'identifer les changements effectués.

Parmi les autres travaux connexes on peut en particulier recenser les recherches autour des méthodes de conception comme UML (Unified Modelling Language de l'OMG) et plus particulièrement les profiles UML [15].

7. Conclusion et perspectives

Nous avons proposé une approche flexible pour annoter les classifieurs et plus précisément l'héritage entre classifieurs. Nous avons montré qu'écrire une classifieur en l'annotant nécessite une plus grande réflexion et une plus grande rigueur. Cependant nous sommes persuadés que le bénéfice qu'il est possible d'en retirer lorsque ces annotations sont prises en compte par les environnements de programmation est l'amélioration de la réutilisation, de l'adaptabilité, de la documentation et de la robustesse des bibliothèques de classes. Le risque est qu'en restreignant l'objectif d'un classifieur comme l'utilisation des annotations semble l'inciter, on constate une prolifération de classifieurs. Pour éviter ce travers, il faudra utiliser les annotations uniquement quand cela apporte quelque chose de significatif. Un certain nombre de points sont encore à améliorer. Cela concerne en particulier la description et la manipulation de classification pour laquelle nous devons mieux prendre en compte l'état de l'art. Il en va de même pour l'étude des problèmes relatifs au *reverse-engineering*. Enfin il pourra être intéressant d'étendre notre approche de telle manière à pouvoir définir aussi des usages pour les classifieurs.

Références

[1] T. Lawson, C. Hollinshead, ans M. Qutaishat, "The Potential For Reverse Type Inheritance in Eiffel", *Technology of Object-Oriented Languages and Systems TOOLS13*, Prentice Hall, 1994, pp. 349-357.

[2] P. Crescenzo, and P. Lahire, "Using both Specialisation and Generalisation in a Programming Language : Why an How ?", *Advances in Object-Oriented Information Systems OOIS 2002 Workshops*, Montpellier, September 2002, pp. 64-73.

[3] M. Sakkinen, "Exheritance, Class Generalization Revived", *Object Oriented Programming ECOOP-2002 The Inheritance Workshop*, June 2002.

[4] P. Crescenzo, OFL : Un Modèle pour Paramétrer la Sémantique Opérationelle Des Langages à Objets - Applicaton aux Relations inter-classes, PhD. Thesis, University of Nice-Sophia Antipolis, December 2001.

[5] B. Meyer, *Object-Oriented Software Construction 2nd edition*, Prentice-Hall, 1997.

[6] G.L. Steele, *Common Lisp the Language 2nd edition*, Digital Press, 1990.

[7] X. Girod, Conception par objets - MECANO : une Méthode et un Environnemnet de Construction d'Application par Objets, Phd. Thesis, University of Joseph Fourier Grenoble I, Grenoble, June 1991.

[8] T. Ledoux, and P. Cointe, "Les Métaclasses Explicites comme Outil pour Améliorer la Conception des Bibliothèques de Classes", GDR'95, Grenoble, 1995.

[9] G. Ardourel, Modélisation des Mécanismes de Protection dans les Langages à Objets, Phd Thesis, University of Montpellier II, Montpellier, December 2002.

[10] A. Taivalsaari, "On the Notion of Inheritance", *ACM Computing Surveys*, ACM press, September 1996, Vol. 28 No. 3 pp. 438-479.

[11] D.C Halbert, and P.D O'Brien, "Using Types and Inheritance in Object-Oriented Languages", *Object Oriented Programming ECOOP-1987*, 1987, pp. 20-31.

[12] P. Clarcke, B. Malloy, and P. Gibson, "Using a Taxonomy Tool to Identify Changes in Object-Oriented Software", to appear in *Conference on Software Maintenance and Reengineering 2003*, 2003.

[13] D. Pescaru, and P. Lahire, "Modifiers in OFL - An Approach For Access Control Customization", I3S Laboratory Research Report, May 2003, pp. 10.

[14] E. Gamma, and all, "Design Patterns - Elements of Reusable Object-Oriented Software, *Addison-Wesley Professional Computing Series*, April 1996.

[15] Object Management Group, "Unified Modeling Language Specification(UML), Version 1.5, Mars 2003.