Héritage non-conforme en Eiffel

Implantation sur le compilateur SmartEiffel et retour d'expérience

Frederic Merizen— Dominique Colnet— Philippe Ribet — Cyril Adrian

LORIA
Campus Scientifique
BP 239
54 506 Vandœuvre-lès-Nancy Cedex
{frederic.merizen, dominique.colnet, philippe.ribet}@loria.fr
cyril.adrian@laposte.net

RÉSUMÉ. *L'héritage non-conforme est un mécanisme récemment introduit dans la nouvelle définition du langage Eiffel. Nous rendons compte de la première implantation de ce mécanisme dans un compilateur de production. Ce mécanisme est similaire à l'héritage traditionnel à la différence près qu'il n'induit pas de relation de sous-typage entre l'héritier et le parent, et qu'il n'autorise pas de polymorphisme. Il permet au concepteur de capturer davantage de décisions de conception dans le code source lui-même. De plus, ce mécanisme aide les compilateurs à réduire la quantité de liaison dynamique dans le programme final. L'héritage non-conforme permet de redéfinir des arguments de méthode de manière covariante, et de réduire la visibilité d'une méthode sans risque d'erreurs de typage à l'exécution.*

ABSTRACT. *Non-conforming inheritance is a mechanism recently introduced in the new definition of the Eiffel language. We provide feedback from the first implementation of this mechanism in a production compiler. This mechanism is similar to traditional inheritance but it doesn't induce a subtyping relationship between the child and the parent, and disallows polymorphism. It allows the designer to capture more design decisions in the source code itself. Furthermore this mechanism helps compilers to statically remove dynamic dispatch code. Non-conforming inheritance incurs no type-system soundness problems even when arguments are redefined covariantly or when the exportation status of a feature is restricted in the heir type.*

MOTS-CLÉS : *Héritage, sous-typage, conformance, réutilisation d'implantation*

KEYWORDS: *Inheritance, subtyping, conformance, implementation reuse*

1. Introduction

Dans le cadre de notre travail au sein du groupe ECMA de normalisation du langage Eiffel (ecma TC39-TG4, n.d.), nous avons implanté au sein de notre compilateur SmartEiffel un mécanisme d'héritage dit *non-conforme*. Alors que l'héritage ordinaire sert à la fois comme expression d'une relation de conformance entre types et comme outil de réutilisation de modules, l'héritage non-conforme met en avant la seule réutilisation de modules logiciels. En tant que premiers utilisateurs de ce nouveau mécanisme, nous avons été surpris de constater à quel point celui-ci influençait notre style de programmation. L'héritage non-conforme offre des options de conception intéressantes sans entraîner de sur-coût à l'exécution ou à la compilation. Bien que nous soyons à notre connaissance les premiers à implanter l'héritage non-conforme dans un compilateur de production, ce mécanisme n'est pas lié de façon inhérente au langage Eiffel ou au compilateur SmartEiffel. En particulier, sa mise en œuvre ne nécessite pas une analyse globale du système à compiler. En conséquence, nous chercherons dans cet article à rendre compte de notre expérience avec l'héritage d'une manière essentiellement indépendante du langage. On ne trouvera donc pas ici de détails syntaxiques ou de matériel de référence spécifique à SmartEiffel.

Le choix de mettre en œuvre l'héritage non-conforme dans un langage à classes dit pur peut étonner. On peut en effet considérer que la notion de classe est fondée sur la réunion des notions de type mathématique et de module logiciel, et sur la réunion correspondante des relations de sous-typage (ou conformance) et de réutilisation de code dans la relation d'héritage. Cette relation d'héritage, quand elle est utilisée de manière légitime, traduit toujours une relation « A *est un* B ». Ainsi, une classe POMME pourrait raisonnablement hériter d'une classe FRUIT puisque effectivement une POMME *est une* sorte de FRUIT. Cette relation « A *est un* B » permet d'utiliser un A partout où un B serait attendu, ce qui impose un certain nombre de contraintes à A. En particulier il n'est pratiquement pas possible de rendre privée dans A une méthode publique de B sans que cela affaiblisse le typage statique. Meyer, motivé par l'existence d'exceptions taxonomiques, propose néanmoins un mécanisme permettant de réduire la visibilité d'une méthode héritée dans une classe fille (Meyer, 1997) sans sacrifier le typage statique, mais au prix d'une analyse globale du système qui n'a à notre connaissance jamais été employée dans un compilateur de production.

Pourtant, dans certains cas deux classes A et B peuvent avoir des méthodes et attribut en commun sans être liées par une relation « A *est un* B » au sens de Liskov (Liskov, 1988). Utiliser l'héritage ordinaire pour factoriser ces méthodes et attributs communs serait une erreur méthodologique. On peut bien sûr choisir de factoriser le code commun au moyen d'une classe utilitaire, à laquelle A et B accèdent au travers d'une relation client-fournisseur plutôt que par une relation d'héritage ; on peut aussi choisir de ne pas factoriser le code en question. L'héritage non-conforme permet une autre solution, à la fois simple et élégante.

Nous présenterons en section 2 un exemple de conception typique tirant parti de l'expressivité de l'héritage non-conforme. La section 3 est consacrée à la sémantique

de l'héritage non-conforme, et à ses interactions avec d'autres aspects du langage hôte. En section 5, nous examinerons des cas pratiques d'utilisation de l'héritage non-conforme dans le compilateur SmartEiffel et sa bibliothèque. Nous verrons en section 6 comment l'héritage non-conforme peut être mis à profit pour exprimer certains design-patterns connus. La section 7 présente des mécanismes similaires à l'héritage non-conforme dans d'autres langages qu'Eiffel, et est suivie par une conclusion en section 10 et une bibliographie.

2. Exemple motivant

Examinons l'implantation d'une structure de données classique, la pile, pour motiver l'introduction de l'héritage non-conforme. L'implantation Java (Arnold *et al.*, 1996) de la classe `Stack` s'appuie sur l'héritage traditionnel : la classe `java.util.Stack` hérite de la classe `java.util.Vector`, la classe `Vector` représentant la notion de collection ordonnée de taille variable. On peut imaginer que ce choix d'héritage a rendu très simple l'implantation de la classe `Stack`. Pourtant, cet héritage est suspect d'un point de vue méthodologique : peut-on vraiment considérer qu'une pile *est une* collection ? De fait, en choisissant d'implanter la classe `Stack` comme héritière de la classe `Vector`, les auteurs de la bibliothèque standard Java ont aussi donnée à `Stack` des méthodes publiques violant la sémantique de pile – par exemple, des méthodes `add` et `get` permettant d'accéder à n'importe quel élément de la pile, plutôt qu'à son seul sommet.

En Eiffel (Meyer, 1994), la réponse traditionnelle à ce problème a été de permettre à une classe fille de diminuer la visibilité de routines héritées. En imaginant que cette approche soit possible en Java, la classe `Stack` pourrait rendre privées des méthodes telles que `add` et `get`. Pourtant, en l'absence de dispositifs complémentaires, la protection offerte est faible : la relation de sous-typage permet en effet d'affecter un objet de type `Stack` à une variable de type `Vector`. Or les méthodes `add` et `get` de `Vector` sont publiques. Un appel polymorphe à `Vector.get` est donc possible et peut résulter en l'exécution de `Stack.get`. Techniquement, on peut assez facilement faire en sorte qu'un tel appel déclenche une exception à l'exécution, mais cette solution n'est guère satisfaisante dans un langage statiquement typé. Meyer a proposé un mécanisme permettant au compilateur de vérifier statiquement que le programme n'effectue aucun appel polymorphe à une méthode dont la visibilité diminue au cours de l'héritage. En pratique, ce mécanisme, qui nécessite une analyse globale du système à compiler, est délicat à mettre en œuvre et n'est pas utilisé dans les compilateurs Eiffel de production. Sa mise en œuvre serait encore plus problématique dans un langage qui permet comme Java de charger dynamiquement des classes en cours d'exécution.

Comme nous venons de le voir, dans le cas des classes `Stack` et `Vector`, la relation de sous-typage induite par l'héritage est source de problèmes, parce-qu'une pile n'*est pas* une collection. La relation d'héritage non-conforme exprime justement la relation d'extension de module logiciel de l'héritage classique sans lui associer de relation de sous-typage, ni la possibilité d'affectations polymorphes. Aussi, nous avons

implanté dans SmartEiffel (Colnet *et al.*, n.d.) une classe STACK héritant de manière non-conforme de la classe ARRAY (l'équivalent de la classe `Vector`). Nous avons par ailleurs rendu privées dans STACK les méthodes de ARRAY qui ne font pas partie du modèle classique de la pile. Dans le cas de la bibliothèque de SmartEiffel, STACK a un ancêtre non-conforme, ARRAY, et aucun ancêtre conforme. STACK n'est donc le sous-type d'aucun autre type, pas même de l'ancêtre universel ANY (équivalent de la classe Java `Object`). Dès la compilation, d'éventuelles tentatives d'affectation d'une variable source vers une variable cible d'un type ancêtre non-conforme sont détectées par le système de typage et provoquent une erreur de compilation. En particulier, il n'est pas possible de stocker un objet de type STACK dans une variable de type ARRAY accéder à la méthode `get` privée. Le mécanisme d'héritage non-conforme nous a permis d'exprimer d'une manière simple une classe pile pure, tout en tirant parti des facilités offertes par un classe de collection existante. Nous avons employé le même mécanisme d'héritage non-conforme pour implanter une classe QUEUE, représentant le concept de file, au moyen de RING_ARRAY, collection généraliste optimisée pour les ajouts et suppressions en queue et en tête.

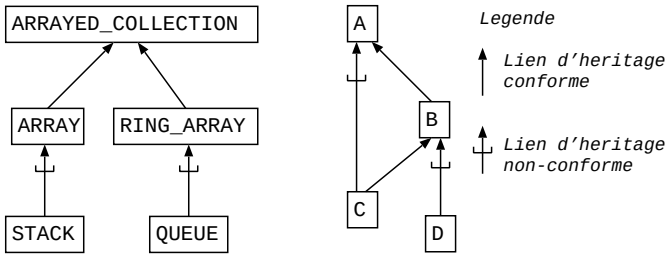
3. Le mécanisme d'héritage non-conforme

3.1. *Objectifs*

Le mécanisme d'héritage non-conforme a été introduit dans le langage Eiffel pour permettre l'héritage d'implantation et l'héritage utilitaire sans entraîner la possibilité d'appels de méthodes polymorphes. Nous avons déjà vu un exemple d'*héritage d'implantation* dans l'exemple de la classe STACK héritant de ARRAY. Plus généralement, une d'héritage est dit d'implantation quand une classe hérite d'une autre classe pour réaliser une abstraction différente. L'*héritage utilitaire* ressemble à l'héritage d'implantation, mais diffère de lui en ce sens que la classe ancêtre représente moins un type abstrait qu'une bibliothèque de fonctions et/ou de constantes.

Pour réaliser l'objectif d'un héritage entre une classe `A` et une classe `B` n'entraînant pas la possibilité d'*appels* polymorphes de méthodes de `A` à travers des variables de type `B` (par exemple des appels à `Stack.get` au travers d'une variable de type `Vector`), nous avons introduit l'héritage non-conforme qui n'introduit pas de relation de sous-typage (ou *conformance*) entre les types en jeu, et interdisant par conséquent toute *affectation* polymorphe entre ces types.

L'héritage classique garde bien sûr son utilité et est conservé. Pour éviter toute confusion entre les deux types d'héritage, nous appellerons l'héritage classique *héritage conforme*. La figure 1 introduit une notation graphique pour ces deux types d'héritage.



(a) Une pile et une file

(b) La classe A est un ancêtre conforme de B et C, et un ancêtre non-conforme de D.

Figure 1. *Héritage conforme et héritage non-conforme*

4. Typage de l'héritage non-conforme

Comme nous l'avons déjà vu, l'héritage non-conforme se distingue de l'héritage classique par le fait qu'il n'entraîne pas de relation de sous-typage, c'est-à-dire de conformance, entre les types parent et héritier. Cette non-conformance interdit d'affecter un objet de la classe fille à une variable de la classe-mère, ou de passer en paramètre effectif d'une méthode un objet de la classe fille quand le paramètre formel est de la classe mère. Cette interdiction de polymorphisme n'admet aucune exception, ce qui distingue ce mécanisme de l'héritage privé de C++ . En dehors de cette absence de sous-typage, l'héritage non-conforme présente exactement les mêmes caractéristiques que l'héritage ordinaire du langage hôte. Ainsi, en Eiffel l'héritage multiple et répété, la redéfinition, le renommage et le changement de visibilité de méthodes héritées sont supportés sans difficulté particulière.

Dans un contexte où une classe peut hériter plusieurs fois, certaines de ces relations d'héritage étant conformes et d'autres non, il convient toutefois de préciser le vocabulaire pour décrire les relations d'héritage et de sous-typage qui lient les classes du système.

Pour deux classes A et B, nous dirons que A est un *parent* de B si B hérite de A, indépendemment de la nature conforme ou non de cet héritage. La relation « A est *ancêtre* de B » est la clôture réflexive transitive de la relation « A est parent de B ».

La classe A est un *parent conforme* de B si A est parent de B et qu'une au moins des relations d'héritage qui lie B à A est conforme. La relation « A est *ancêtre conforme* de B » est la clôture réflexive transitive de la relation « A est parent conforme de B ». Pour les besoins de notre discussion, nous dirons de manière synonyme que A est un *sous-type* de B ou que B est un ancêtre conforme de A.

Enfin, nous dirons que A est un *parent non-conforme* de B si A est un parent de B mais sans être un parent conforme de B. De même, A est un *ancêtre non-conforme* de B si A est un ancêtre de B sans être un ancêtre conforme (attention, ce n'est pas la clôture réflexive transitive de la relation « A est un parent non-conforme de B»). Dans la hiérarchie de classes illustrée en figure 1(b), la classe D a deux ancêtres non-conformes, à savoir A et B. La classe C n'a pas d'ancêtre non-conforme.

4.1. *Impact sur la compilation séparée*

L'héritage non-conforme oblige le compilateur à procéder à des vérifications qui ne sont pas nécessaires dans le cas de l'héritage conforme. En effet, si une classe A a pour parent non-conforme une classe B, l'objet courant (`this` en Java) dans A ne sera pas conforme à B. Si `myB` est une variable de type B, l'affectation `myB = this ;`, correcte si elle est écrite dans B, devient incorrecte si elle est héritée par A. Dans le cas d'Eiffel, l'existence de types ancrés permet à ce problème de se produire pour d'autres variables que `this` .

Pour garantir statiquement le typage d'une application, le compilateur doit donc revérifier dans le contexte de la classe fille tout code venant d'un parent non-conforme. Si une méthode héritée utilise une affectation problématique, le programmeur devra redéfinir cette méthode dans la classe fille. Pour faciliter au programmeur la redéfinition de la méthode, et aussi pour faciliter au compilateur la détection des affectations problématiques, il est préférable que le code source des parents non-conformes soit disponible au moment de la compilation de la classe fille. Nous pensons cependant que le bytecode d'une classe java contient suffisamment d'information de typage pour permettre au compilateur de procéder à la recherche d'affectations non-sûres, mais nous n'avons pas mis en pratique cette théorie. De même, nous pensons qu'il suffit d'annoter du code objet binaire avec des informations de typage relativement restreintes pour permettre au compilateur de vérifier si une méthode héritée reste valide dans un héritier non-conforme donné. Héritage non-conforme et compilation séparée ne semblent donc pas fondamentalement incompatibles.

4.2. *Solidité* soundness *du système de typage*

Depuis la création du langage, il a toujours été possible en Eiffel de redéfinir des méthodes héritées de manière covariante et restreindre leur visibilité. Nous pensons que ces possibilités sont utiles pour concevoir des systèmes flexibles. Il est cependant connu que la présence de ces options rend le système de typage d'Eiffel non-solide (*unsound*) (Castagna, 1995; Bruce *et al.*, 1996; Castagna, 1996). Meyer a montré (Meyer, n.d.) que la redéfinition covariante et la restriction de visibilité d'une méthode ne pouvait causer des erreurs de typage à l'exécution que si cette méthode était appelée de manière polymorphe. Ainsi, une classe peut redéfinir de manière covariante ou restreindre la visibilité d'une méthode héritée d'un parent non-conforme sans que cela entraîne de risque d'erreur de typage à l'exécution, puisqu'il n'est jamais possible

qu'un objet de la classe fille soit attaché à une variable de la classe mère non-conforme, et qu'aucun appel polymorphe dangereux ne peut donc avoir lieu.

4.3. *La classe ancêtre universelle*

Beaucoup de langages à classe ont une classe qui joue le rôle d'ancêtre universel : `Object` dans le cas de Java, ANY dans le cas d'Eiffel... En introduisant l'héritage non-conforme dans SmartEiffel, nous avons choisi de permettre à une classe d'avoir ANY comme ancêtre non-conforme : ANY est donc un ancêtre universel, mais pas un ancêtre *conforme* universel. Dans le standard ECMA, des dispositions particulières ont été prises pour interdire qu'une classe ait ANY comme parent non-conforme ; ANY est alors un ancêtre conforme universel. Le principal avantage du choix du standard ECMA est de permettre d'utiliser ANY comme option par défaut pour le type de l'argument de certaines fonctions utilitaires telles que la copie ou la comparaison d'objets.

En pratique, nous avons constaté que les types génériques et les types ancrés offraient suffisamment de flexibilité pour que cette option par défaut ne sont pas nécessaire. Nous avons donc préféré laisser aux développeurs la possibilité de rendre certaines classes non-conformes à l'ancêtre universel. Dans SmartEiffel, le graphe d'héritage est connexe, mais le graphe d'héritage conforme peut ne pas ne l'être. Nous espérons que cela encouragera les développeurs à tirer parti au maximum de la sécurité du système de typage, et à limiter les transtypages. Nous verrons en section 5.3 une autre raison pour laquelle nous préférons autoriser une classe à être un descendant non-conforme de la classe ANY.

5. L'héritage non-conforme en action

5.1. *Héritage utilitaire*

Une application immédiate de l'héritage non conforme est la possibilité d'hériter de classes qui ne servent en fait que de bibliothèques de fonctions ou de constantes. Meyer appelle ce type d'héritage de l'héritage utilitaire (*commodity inheritance*). La classe `java.lang.Math` constitue un bon exemple d'une classe-bibliothèque de fonctions. En Java, les méthodes et attributs de telles classes sont normalement tous statiques ; depuis la version 1.5 du langage, il est possible d'accéder de manière très commode à ces méthodes et attributs en utilisant une directive `import` . Bien que le mécanisme sous-jacent soit très différent, on obtient en Eiffel la même élégance syntaxique en faisant de la classe utilisatrice un héritier non-conforme de la classe bibliothèque. On trouve plusieurs exemples d'utilisation de l'héritage utilitaire dans notre compilateur et notre bibliothèque standard. Par exemple, la bibliothèque graphique vision comprend une classe COLOR_LIST qui liste des constantes de couleur.

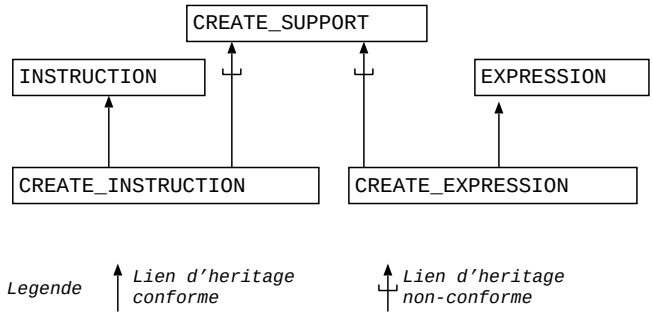


Figure 2. *Un exemple d'héritage d'implantation*

5.2. *Héritage d'implantation*

La possibilité de factoriser des comportements faisait partie de notre cahier des charges pour l'héritage non-conforme. Nous voulions pouvoir écrire une classe qui hérite d'une autre pour utiliser son comportement, mais en exportant une interface différente de celle de la classe mère. Nous avons déjà rencontré un exemple de cette utilisation de l'héritage, qui porte le nom d'héritage d'implantation, sous la forme des classes STACK et ARRAY. Nous avons employé à plusieurs reprises l'héritage d'implantation dans le compilateur et la bibliothèque standard. Par exemple, les classes AVL_DICTIONARY (dictionnaire associatif) et AVL_SET (ensemble) utilisent toutes les deux une classe AVL_TREE (arbre AVL) pour leur implantation. Un autre exemple, tiré du compilateur, est présenté en figure 2. Les classes abstraites INSTRUCTION et EXPRESSION représentent respectivement une instruction et une expression du programme Eiffel source. Chacune de ces classes a de nombreux descendants, dont respectivement CREATE_INSTRUCTION et CREATE_EXPRESSION, qui représentent respectivement une instruction et une expression de création d'un nouvel objet. Ces classes présentent des interfaces différentes, mais la majeure partie du code a pu être mise en commun dans une classe CREATE_SUPPORT. En faisant hériter CREATE_INSTRUCTION et CREATE_EXPRESSION de CREATE_SUPPORT de manière non-conforme, nous avons pu modéliser le fait que, bien qu'elles emploient les mêmes mécanismes sous-jacents, les instructions et les expressions de création jouent dans le compilateur des rôles qui ne sont jamais interchangeables.

On notera encore que la classe INSTRUCTION a un grand nombre de descendants, et que la plupart de ses routines sont abstraites et polymorphes. Certaines de ces méthodes sont directement implantées dans CREATE_INSTRUCTION par des méthodes concrètes héritées de CREATE_SUPPORT. Bien que ces routines soient héritées de manière non-conforme, une liaison dynamique sur l'interface INSTRUCTION peut aboutir à leur exécution. En revanche, aucune routine de CREATE_INSTRUCTION ne sera jamais exécutée suite à une liaison dynamique sur l'interface CREATE_SUPPORT. Il est

donc faux de dire dans l'absolu que l'héritage non-conforme empêche la liaison dynamique.

5.3. *Cacher des propriétés universelles*

La classe ancêtre universelle possède un certain nombre de méthodes utilitaires généralistes, dont IS_EQUAL qui permet de comparer deux objets champ par champ, et TWIN qui permet de cloner un objet. Ces méthodes sont généralement utiles, mais on souhaite parfois les cacher pour promouvoir l'utilisation de l'aliasing (Zendra *et al.*, 2001) et de l'opérateur de comparaison rapide = (c.a.d. l'opérateur == de Java). Nous avons vu en section 4.2 qu'on pouvait réduire la visibilité d'une méthode sans sacrifier le typage statique à condition de ne cacher que des méthodes venant d'un parent non-conforme. En particulier, dans une classe n'ayant que des parents non-conformes, il estpossible de réduire la visibilité de méthodes venant de la classe ANY.

5.4. *Gain de performance par réduction du polymorphisme*

En plus d'ouvrir le champ à de nouvelles possibilités de conception orientées objet, l'héritage non-conforme peut contribuer à améliorer légèrement les performances d'un programme. En effet, dans un graphe d'héritage comportant des liens d'héritage non-conforme, le degré de polymorphisme des méthodes sera inférieur ou égal au degré de polymorphisme de ces mêmes méthodes dans le graphe obtenu en remplaçant les liens d'héritage non-conforme par des liens d'héritage conforme. Ce gain de précision du typage peut avoir pour effet des tables de méthodes virtuelles plus petites ou des fonctions de dispatch dynamique plus compactes.

En particulier, il est possible d'étendre un langage à héritage simple avec un mécanisme d'héritage multiple non-conforme sans modifier l'algorithme de lookup ou des structures de *lookup* des méthodes virtuelles. Par ailleurs, le compilateur SmartEiffel dispose d'un mécanisme de gestion de la liaison dynamique (Collin *et al.*, 1997) capable d'exploiter la réduction du polymorphisme entrainée par l'emploi de l'héritage non-conforme.

6. Design patterns et héritage non-conforme

Nous avons constaté que l'héritage non-conforme pouvait aider à implanter élégamment certains design patterns (Gamma *et al.*, 1995) en Eiffel (Jézéquel *et al.*, 1999).

6.1. *Flyweight et singleton*

Le pattern *flyweight* mutualise des objets pour supporter de manière efficace un grand nombre d'objets à grain très fin. Dans ce contexte, il est important que les objets à grain fin soient aliasés au maximum. Une classe *singleton* se caractérise par le fait de n'avoir qu'une seule instance à l'exécution. Dans le cas de ces deux patterns, on peut employer la technique décrite en section 5.3 pour cacher la méthode `twin` héritée de ANY et empêcher ainsi que des objets soient dupliqués par inadvertance. Un exemple de classe *flyweight* est présenté en annexe A.1.

6.2. *Class Adapter*

Ce pattern (à ne pas confondre avec son cousin, *Object Adapter*), donne à une classe existante une nouvelle interface, pour permettre à du code client existant d'utiliser la classe en question. On peut réaliser le *class adapter* en lui faisant hériter l'interface à implanter de manière conforme, et la classe à adapter de manière non-conforme. Un exemple de *Class Adapter* implantant un flot d'entrée à l'aide d'un itérateur sur chaîne de caractères est présenté en annexe A.2.

6.3. *Template Method*

Le pattern *template method* définit dans une classe le squelette d'un algorithme. Cet algorithme fait appel à des sous-fonctions abstraites, à définir par les classes héritières. Il s'agit en fait d'un cas particuler d'héritage utilitaire, vu en section 5.1.

7. Mécanismes semblables

7.1. *Le mécanisme INCLUDE de Sather*

Comme Java, le langage Sather(Gomes *et al.*, n.d.) est basé sur une dichotomie classe/interface. Comme en Java, les interfaces Sather peuvent hériter d'interfaces, et les classes Sather peuvent implanter une ou plusieurs interfaces. En revanche, il n'existe pas en Sather d'héritage entre classes : en effet, en Sather la conformance est purement du domaine des interfaces, et non des classes. Les classes Sather peuvent en revanche *inclure* d'autres classes. L'inclusion de classes de Sather est un mécanisme d'extension de modules logiciel qui fonctionne essentiellement de la même manière que l'héritage non-conforme d'Eiffel. L'inclusion multiple (pendant de l'héritage multiple) est supporté. Une classe Sather peut restreindre la visibilité de méthodes obtenues par inclusion d'une autre classe, comme une classe Eiffel peut restreindre la visibilité de méthodes héritées (sachant que cette restriction n'est sûre que si la méthode a été héritée de manière non-conforme).

Le mécanisme de Sather diffère de celui d'Eiffel par deux points importants. Premièrement, Sather s'appuie sur une dichotomie classe/interface alors qu'en Eiffel les deux types d'héritage s'appuient uniformément sur la notion de classe. Deuxièmement, bien que Sather ait un équivalent de l'héritage non-conforme, les redéfinitions covariantes sont interdites en Sather même quand elles sont parfaitement sûres. En fait, en Sather, seules les redéfinitions contravariantes sont permises.

7.2. *Héritage privé/protégé de C++*

Le mécanisme d'héritage `private` ou `protected` de C++ et l'héritage non-conforme se ressemblent mais présentent une différence importante : le mécanisme de C++ ne sert pas à supprimer le polymorphisme, et il ne le fait effectivement pas. En effet, quand une classe `B` hérite d'une classe `A` de manière privée, la classe `B` (et elle seule) est autorisée à affecter un objet depuis une variable de type `B` vers une variable de type `A`. Cette possibilité est illustrée par le code en figure 3. On remarquera qu'à partir du moment où la classe `B` peut effectuer des *affectations* polymorphes de `B` vers `A`, des *appels* polymorphes vers des méthodes de `B` à travers `A` peuvent se produire n'importe où dans le code, ici par exemple dans la fonction `main`. En corollaire, le caractère privé d'une méthode ne peut pas être garanti. Ici, la méthode privée `f` de `B` est appelée par la fonction `main`.

```
#include <iostream>

class A {                          class B: private A {
public:                            public:
  A* as_A(void) {                    using A::as_A;
    return this;
  }                                private:
  virtual void f(void) {             virtual void f(void) {
    std::cout << "Hello";              std::cout << " world!\n";
  }                                  }
};                                 };

int main(void) {
  B* b = new B;
  A* a = new A;
  a->f();             // Cet appel polymorphe affiche "Hello"
  a = b->as_A();      // Affectation polymorphe
  a->f();             // Cet appel polymorphe affiche " world!\n"
}
```

Figure 3. *L'héritage* `private` *de C++ ne supprime pas le polymorphisme*

8. Traits

Schärli *et al.* proposent un mécanisme de composition de traits et son application au langage SmallTalk (Schärli *et al.*, 2003). La composition de traits et l'héritage non-conforme de classes visent un même objectif de réutilisation de code sans polymorphisme. Contrairement aux classes, même utilisées comme ancêtres non-conformes,

les traits ne peuvent pas hériter d'autres entités (traits ou classes) du programme, et les traits ne peuvent pas avoir d'état, c'est-à-dire de variables d'instance.

Le fait que les traits ne puissent pas avoir de variables permet selon les auteurs d'éviter à la composition en diamant d'avoir les problèmes les plus délicats soulevés par l'héritage en diamant. Pourtant, il est possible en Eiffel d'hériter de manière non-conforme d'une classe qui a des attributs sans que cela pose de problème. Le mécanisme de renommage standard de l'héritage Eiffel permet au programmeur de choisir, en cas de diamant, entre la réplication et le partage de chaque attribut.

Comme ils n'entretiennent pas de relation d'héritage, les traits possèdent la *flattening property*, c'est-à-dire que la composition d'un trait avec une classe, ou avec un sous-trait, peut être traitée comme une simple substitution textuelle (avec quelques précautions pour tenir compte des conflits de noms). En particulier, le mot-clé `super` déclenche simplement le *lookup* d'une routine de même nom dans la *classe* ancêtre, indépendamment du fait que ce mot-clé soit écrit directement dans une classe, ou dans un trait que cette classe utilise par composition. Schärli *et al.* montrent comment utiliser cette particularité pour définir un trait qui, par composition, transforme n'importe quel classe de flot d'entrée/sortie en flot synchronisé. On ne peut cependant pas espérer reproduire ce résultat dans un langage à héritage multiple tel qu'Eiffel, car le mot-clé équivalent `Precursor` n'est pas univoque et doit être complété par le nom de la classe parent dans laquelle le *lookup* doit être effectué. Or, le nom de cette classe n'est connu que dans les classes utilisant le trait, et non dans le trait lui-même.

Des travaux ultérieurs (Nierstrasz *et al.*, 2005) montrent que pour que les traits soient utiles comme brique de réutilisation de code dans un langage statiquement typé, il faut que ce langage dispose d'une forme de conformance multiple. L'héritage d'interfaces de Java, par exemple, est suffisamment puissant. Un modèle utilisant uniquement les traits et l'héritage d'interface se rapprocherait du modèle de Sather.

9. Mixins Ruby

Le langage Ruby (Thomas *et al.*, 2001) fournit un bon exemple d'héritage *mixin*. En plus des classes, ruby permet en effet d'écrire des `module`s, qui ressemble aux traits dans le sens où il s'agit d'unites de code composables, mais ne pouvant pas entretenir de relation de sous-typage. Une classe ou un module Ruby peuvent inclure plusieurs sous-modules au moyen du mot-clé `include`. Cette opération, dite de *mixin* de module est traitée en interne par le mécanisme d'héritage simple : l'interpréteur doit donc linéariser la hiérarchie des mixins. Cette linéarisation rend l'opération de *mixin* sensible à l'odre des clauses d'inclusion, ce qui n'est pas le cas de la composition de traits ou de l'héritage Eiffel. L'héritage mixin ne possède pas la *flattening property* : le mot-clé `super` est en effet interprété dans le cadre de la hiérarchie d'héritage linéarisée. Cela rend aussi l'effet d'un appel à `super` plus difficile à prédire dans un mixin Ruby que dans une classe Eiffel, où `Precursor` se réfère simplement à un ancêtre (nommé) de la classe dans laquelle est écrit le mot-clé.

10. Conclusion

Au cours de notre travail sur le langage Eiffel, nous avons expérimenté le mécanisme d'héritage non-conforme sur un projet de dimension moyenne, à savoir le compilateur lui-même et sa bibliothèque. Malgré sa simplicité, ce mécanisme s'est révélé utile pour capturer plus d'informations de design dans le code source même de l'application. Grâce à l'héritage non-conforme, l'emploi de l'héritage d'implantation et de l'héritage utilitaire n'entraînent plus de relations de sous-typage parasites. De plus, l'héritage non-conforme permet des redéfinitions covariantes et des réductions de visibilité de méthodes sans risque d'erreurs de typage à l'exécution.

11. Remerciements

Merci à tous les utilisateurs de SmartEiffel pour le retour d'expérience et les commentaires avisés pendant la mise en place du mécanisme d'héritage non-conforme. Merci à Bertrand Meyer et aux autres membres du groupe de normalisation ECMA TC39-TG4. Merci enfin à un relecteur pour la référence sur les traits.

12. Bibliographie

Arnold K., Gosling J., *The Java Programming Language*, Addison-Wesley, Reading, Massachusetts, USA, 1996.

Bruce K., Cardelli L., Castagna G., Group T. H. O., Leavens G., Pierce B., « On Binary Methods », *Theory and Practice of Object Systems*, 1996.

Castagna G., « Covariance and Contravariance : Conflict Without a Cause », *Theory and Practice of Object Systems*, vol. 17, n° 3, p. 431-447, 1995.

Castagna G., *Object-Oriented Programming : A Unified Foundation*, Progress in Theoretical Computer Science, Birkäuser, Boston, 1996.

Collin S., Colnet D., Zendra O., « Type Inference for Late Binding : The SmallEiffel Compiler », *JMLC '97 : Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, Springer-Verlag, London, UK, p. 67-81, 1997.

Colnet D., Ribet P., Adrian C., Croizier V., Merizen F., « Web site of SmartEiffel, the GNU Eiffel Compiler Tools and Libraries. », , http://SmartEiffel.loria.fr , n.d.

ecma TC39-TG4, « Standard ECMA-367 : Eiffel Analysis, Design and Programming Language », , http://www.ecma-international.org/publications/standards/Ecma-367.htm , n.d.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addisson-Wesley, Reading, Massachusetts, 1995. ISBN 0201633612.

Gomes B., Stoutamire D., Vaysman B., Klawitter H., « Sather 1.1 : A Language Manual », , http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/%webmaker/DescriptionX2Erem-chapters-1.html , n.d.

Jézéquel J. M., Train M., Mingins C., *Design Patterns and Contracts*, Addisson-Wesley, 1999. ISBN 0-201-30959-9.

Liskov B., « Keynote address - data abstraction and hierarchy », *SIGPLAN Not.*, vol. 23, n° 5, p. 17-34, 1988.

Meyer B., *Eiffel, The Language*, Prentice Hall, 1994.

Meyer B., *Object-oriented Software Construction*, 2 edn, Upper Saddle River, N.J., Prentice Hall, 1997.

Meyer B., « Beware of polymorphic catcalls », , http://archive.eiffel.com/doc/manuals/technology/typing/cat.html , n.d.

Nierstrasz O., Ducasse S., Schärli N., Flattening Traits, Technical Report n° IAM-05-005, Institut für Informatik, Universität Bern, Switzerland, April, 2005.

Schärli N., Ducasse S., Nierstrasz O., Black A., « Traits : Composable Units of Behavior », *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, vol. 2743 of *LNCS*, Springer Verlag, p. 248-274, July, 2003.

Thomas D., Hunt A., *Programming Ruby : The Pragmatic Programmer's Guide*, Addison-Wesley, Reading, MA, USA, 2001.

Zendra O., Colnet D., « Coping with aliasing in the GNU Eiffel Compiler implementation », *Software Pratice and Experience (SP&E)*, vol. 31, n° 6, p. 601-613, 2001. *J. Wiley & Sons*.

A. Utilisation de l'héritage non-conforme dans des design patterns

A.1. *Flyweight*

```
class  FLYWEIGHT
insert  ANY              -- Héritage  non-conforme  en SmartEiffel
   export  {} twin  end  -- Cache  la procédure  twin
end  -- class  FLYWEIGHT
```

A.2. *Class adapter*

```
class  STRING_INPUT_STREAM
inherit   TERMINAL_INPUT_STREAM
insert   ITERATOR_ON_STRING
       rename   item  as  filtered_last_character,
                next  as  filtered_read_character,
                is_off  as  end_of_input
       end
creation   make
feature   {ANY}
   is_connected:    BOOLEAN  is  True
   can_unread_character:      BOOLEAN  is  False
   disconnect   is  do  end
feature   {FILTER}
   filtered_has_stream_pointer,        filtered_has_descriptor:       BOOLEAN  is  False
   filtered_descriptor:      INTEGER
   filtered_stream_pointer:       POINTER
   filtered_unread_character       is
      do
        filter  := Void
      end
end  -- class  STRING_INPUT_STREAM
```

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

1. ARTICLE POUR LES ACTES :
   LMO 2006

2. AUTEURS :
   Frederic Merizen— Dominique Colnet— Philippe Ribet — Cyril Adrian

3. TITRE DE L'ARTICLE :
   Héritage non-conforme en Eiffel

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Héritage non-conforme en Eiffel

5. DATE DE CETTE VERSION :
   23 janvier 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   LORIA
   Campus Scientifique
   BP 239
   54 506 Vandœuvre-lès-Nancy Cedex
   {frederic.merizen, dominique.colnet, philippe.ribet}@loria.fr
   cyril.adrian@laposte.net
   – téléphone : +33(0)3 83 58 17 27
   – télécopie : +33(0)3 83 58 17 01
   – e-mail : Frederic.Merizen@loria.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls` ,
   version 1.22 du 04/10/2005.

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

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