Université USTL - Lille I
Laboratoire LIFL - INRIA
Lieu Lille, France
Équipe RMoD/Adam
Maître de stage Stéphane Ducasse
Jury Jean-Luc Dekeyser et Christophe Chaillou

Mémoire de Master Recherche

Vers une traitification des Collections

Guillaume Libersat <glibersat@sigill.org>

Février à Juin 2008

-

Soutenu le 16 Juin 2008

Remerciements

Je remercie Stéphane Ducasse pour m'avoir encadré pendant ce travail et pour m'avoir guidé sur la découverte de nouveaux domaines. Je tiens à remercier particulièrement Mathieu Suen pour son aide et ses conseils tout au long du stage. Merci aussi à Alexandre Bergel pour toujours avoir été volontaire pour les relectures.

J'adresse aussi des remerciements spéciaux aux joyeux lurons du bureau 223 ainsi qu'à A.F. Le Meur qui m'ont aidé à garder le moral dans les moments difficiles, chacun à leur manière.

Je tiens aussi à remercier Pierre Tramo, J2EE lead architect, pour ses remarques pertinentes sur les collections java.

Enfin, je me vois obligé de remercier Alban pour toutes les soirées passées à hanter l'INRIA, rythmées au chant de « léon, si t'es champion, appui sur l'champignon ».

Table des matières

1 Introduction **1**
1.1 Environnement 1
1.2 Contexte 1
1.2.1 Problématique 1
1.2.2 Objectifs et réalisations 2
1.3 Outillage pour la réalisation 3

2 Etat de l'art : Composition de comportement et refactorisations **5**
2.1 Composition de comportement en orienté-objet 5
2.1.1 L'héritage 5
2.1.2 Les mixins 11
2.1.3 Les traits 12
2.1.4 Synthèse 21
2.2 Refactorisations autour des traits et des collections 22
2.2.1 Interfaces and specifications for the Smalltalk-80 Collection Classes 22
2.2.2 Applying Traits to the Smalltalk Collection classes 24
2.2.3 Nile 28
2.2.4 Synthèse 30

3 Analyse des bibliothèques de collections existantes **33**
3.1 Analyse des Collections 33
3.1.1 Les classes de collections 33
3.1.2 Les collections en Java 35
3.1.3 Les collections en Smalltalk 39
3.2 Synthèse et identification des problèmes 43
3.2.1 Généralités conceptuelles 43
3.2.2 Similitudes 44
3.2.3 Divergences 44
3.2.4 Conclusion 45

4 Des traits-protocoles pour les Collections **47**
4.1 Une hiérarchie complexe et hétérogène 47
4.2 Unifier les vues structurelles 50
4.3 Faire face à l'hétérogénéité 50
4.3.1 Vers des implantations spécifiques 50
4.3.2 La nécessité des protocoles 51

4.3.3 Matérialisation des protocoles 52
4.4 Améliorer la consistance par les contrats 53
4.4.1 Définition des contrats 54
4.4.2 Supports d'exécution 55
4.5 Garder un système ouvert 61
4.6 Améliorer l'expressivité des protocoles 64

5 Évaluation et méthodologie 67
5.1 Evaluation 67
5.1.1 Granularité des protocoles 67
5.1.2 Factorisation de code 69
5.1.3 Comparaison avec la refactorisation précédente 70
5.1.4 Modèles de traits 70
5.2 Perspectives 71

6 Conclusion 73

A La syntaxe Smalltalk 77

B Trait Users pour OBBrowser 81

1 Introduction

1.1 Environnement

Ces travaux ont été réalisés dans le cadre de mon Master 2 recherche, spécialité « Systèmes Embarqués et distribués, Image et Génie Logiciel ». Il s'agit d'un stage de fin d'études d'une durée de cinq mois qui a pour but d'initier les étudiants à la recherche en informatique.

Cadre. La formation doctorale est rattachée au LIFL, ou « Laboratoire d'Informatique Fondamentale de Lille ». Il s'agit d'un laboratoire universitaire dépendant de l'Université de Lille1.

Le stage s'est cependant déroulé à l'INRIA (« Institut National de Recherche en Informatique et Automatique ») Lille - Nord Europe. Cet organisme a pour mission la recherche fondamentale et appliquée dans le domaine des sciences et technologies de l'information et de la communication.

Équipe. L'équipe « RMoD », dans laquelle ce stage a été réalisé, est une équipe INRIA en cours de création qui est actuellement rattachée à l'équipe-projet « Adam ».

« RMoD » se pose comme objectif général la question suivante : « Comment aider les équipes de développement à maintenir et faire évoluer leurs logiciels ? ». Cet objectif est réalisé à travers deux objectifs complémentaires : la remodularisation et les constructions modulaires.

L'équipe est menée par Stéphane Ducasse.

1.2 Contexte

1.2.1 Problématique

Récemment, les traits [SDNB03, BDNW08, DWBN07] ont proposé une solution aux problèmes liés à l'héritage multiple [Mey88, Kee89, Str86].

Les traits sont des entités à granularité fine qui permettent de factoriser du comportement pour les classes. Dans ce modèle, l'avantage indéniable est que l'entité composante dispose de tout le contrôle de la composition. Nous les présentons à la section 2.1.3.

Des recherches précédentes ont permis de valider l'utilité des traits en refactorisant des bibliothèques existantes de collections et de streams. Ces refactorisations

ont montré une maturation du modèle des traits avec l'expérience acquise dans le domaine.

En effet, la première refactorisation des collections a consisté à une traitification des classes sans aller plus loin. Les dernières refactorisations réalisées à l'aide des traits, elles, s'intéressent à la réécriture complète afin de donner un aspect plus modulaire et de tirer vraiment parti des possibilités offertes par les traits.

Aucune hiérarchie importante n'a vraiment, à ce jour, était refactorisée avec comme objectif de tirer au maximum parti des traits.

Refactoriser avec les traits des hiérarchies où de nombreuses classes hétérogènes sont présentes reste donc, à ce jour, un terrain encore non exploré.

1.2.2 Objectifs et réalisations

Buts. Le but de ce travail est de mettre les traits à l'épreuve et de prouver expérimentalement qu'ils peuvent s'adapter ainsi qu'être un atout pour la refactorisation des grandes bibliothèques.

Plus spécifiquement, nous souhaitons répondre aux questions suivantes :

- Comment est-il possible de proposer une hiérarchie de traits pouvant s'adapter à la diversité d'éléments ?
- Par quels processus peut on s'assurer que la hiérarchie pourra s'adapter aux besoins futurs ?
- Comment s'assurer que cette hiérarchie restera consistante ?
- Quelle est la granularité idéale des traits qui favorise la réutilisation ?
- Jusqu'à quel point pouvons nous corriger les problèmes que nous identifierons dans la bibliothèque des collections existante ?
- Quelles sont les limites et les contraintes que nous rencontrerons lors de l'utilisation des traits ?

Solution. Nous avons décidé de prendre la bibliothèque de collections de Smalltalk comme terrain d'expérimentation. Le choix des collections s'est fait pour plusieurs raisons : i/ elles posent des problèmes lors de leur implémentation avec héritage simple ; ii/ il s'agit d'une bibliothèque très conséquente ; iii/ elles proposent une diversité importante de structures de données, ce qui est représentatif des grandes hiérarchies ; iv/ d'autres travaux impliquant les collections et les traits ont déjà été réalisés ce qui nous permettra une comparaison ; v/ les collections sont une bibliothèque importante des langages de programmation ; vi/ des contraintes sont imposées par les spécifications ANSI Smalltalk et par les implémentations existantes.

Contributions. Les contributions de ce travail sont donc :

1. L'analyse des collections existantes de Smalltalk et Java ainsi que la définition par le standard Smalltalk ANSI ;
2. L'identification des problèmes des bibliothèques de collections existantes ;

3. La proposition d'un design pour une nouvelle hiérarchie de collections, composée d'éléments réutilisables. Il faut bien noter qu'il s'agit d'une proposition et non d'une implantation ;
4. La proposition d'une extension aux traits afin de permettre l'utilisation de contrats ;
5. L'évaluation des traits en tant qu'éléments réutilisables pour définir des bibliothèques et la vérification de l'obtention d'un design plus clair et plus modulaire ;
6. Une identification des problèmes qui surviennent lors de l'utilisation des traits (par exemple, comment choisir la granularité ?).

1.3 Outillage pour la réalisation

Héritage et traits. Afin de réaliser notre proposition, nous disposons d'un large éventail de solutions permettant de composer du comportement. Nous les présentons dans le chapitre 2.

Nous basons cependant notre travail sur le modèle de composition le plus abouti, c'est à dire les traits. En effet, ce modèle est beaucoup moins limité que l'héritage, évite la copie ou le wrapping ; ce qui est partie intégrante de notre objectif.

Trois modèles principaux des traits sont disponibles ; c'est pourquoi nous les analyserons afin de déterminer quel est le modèle qui semble le plus approprié pour nous aider à réaliser notre proposition. En effet, les traits sont parties intégrantes de notre travail car il s'agit d'un outil crucial pour la réussite de celui-ci.

Les collections. Une bibliothèque de collections permet de stocker des ensembles de données sous différentes formes. Cette diversité de formes permet de s'adapter aux besoins des programmes et de les rendre plus efficaces en proposant des version optimisées.

Les collections sont essentielles pour un langage de programmation. Il s'agit d'ailleurs d'un très bon cas concret pour notre expérimentation. En effet, celles-ci exposent une hiérarchie très complexe et disposent très souvent de problèmes de conceptions. Nous les décrivons dans la section 3.1.

Nous nous baserons donc sur les collections de Smalltalk-80 pour réaliser nos travaux, mais étudierons tout de même d'autres collections comme celles de Java afin d'obtenir un panel plus représentatif.

Smalltalk. Smalltalk [ANS98] est un langage de programmation dynamique typé principalement inspiré de Lisp et Simula, où le seul moyen d'activer un objet est d'envoyer un message.

Il a été créé dans les années 80 par un groupe de chercheurs du Xerox PARC (Palo Alto Research Center). Parmi ceux-ci, on compte notamment Alan Kay (Turing Award 2002) qui s'est chargé de la majeure partie du design.

Une des particularités du design ce langage est que tout est objet et qu'un objet est toujours une instance de classe. Il n'y a pas de types primitifs comme on peut le rencontrer en Java par exemple. Ainsi, les classes elles-mêmes sont des objets, instances de leurs méta-classes.

Smalltalk a aussi l'originalité d'être le plus réflexif (structurellement et comportementalement) et d'être implanté en lui-même. On peut donc le voir comme un petit système d'exploitation.

Smalltalk peut être considéré comme un langage modèle dans le sens où il essaye de minimiser sa syntaxe[1] tout en essayant d'être le plus consistent possible. Il n'en reste pas moins puissant et a été remarqué dès ses premières versions par le milieu académique.

C'est ainsi que via ce langage, les premiers outils comme les navigateurs de classes ou de code sont nés. Il a d'ailleurs été, depuis sa création, terrain d'expérimentation pour les chercheurs et industriels. Il a ainsi inspiré de nombreux concepts comme la programmation par passage de messages (pour l'informatique répartie par exemple), les interfaces graphiques modernes ou encore les méta-modèles que l'on retrouve dans les langages modernes (Perl, Java, Python, ...).

La dernière version de Smalltalk a été normalisée AINSI en 1998 et est actuellement utilisable sous plusieurs incarnations. On retrouve notamment VisualWorks, Squeak ou encore GNU Smalltalk.

Squeak. Squeak [Squ] est une implantation de Smalltalk, dérivée de standard 80. Elle a été initiée par une équipe d'Apple Computer incluant des membres de la communauté originelle de Smalltalk-80.

Elle a ensuite été reprise et améliorée par les studios Walt Disney Imagineering qui l'ont utilisée à des fins internes.

Elle est maintenant activement maintenue et développée par une communauté internationale.

Cette implémentation fournit une machine virtuelle et une image. Cette première est écrite en elle-même, ce qui la rend entre autres très facilement portable. L'image quant à elle consiste en un cache de bytecode. Ceci permet d'avoir un ensemble de données indépendantes de la machine cible et utilisable sur n'importe quelle architecture du moment qu'une machine virtuelle Squeak est disponible.

Squeak a été récemment sélectionné pour le projet OLPC (One Laptop Per Child [Ben]) afin de fournir l'environnement Etoys, résultat des travaux de recherche de l'équipe d'Alan Kay.

[1] Une synthèse de sa syntaxe est disponible en annexe A

2 Etat de l'art : Composition de comportement et refactorisations

Dans cette section, nous présentons les principales méthodes de composition de comportement proposées jusqu'à ce jour pour la programmation orientée-objet

Nous commencerons par étudier les types d'héritages encore actuellement très largement utilisés, c'est à dire l'héritage simple et l'héritage multiple.

Nous continuerons ensuite sur des propositions plus récentes tels que les Mixins ou les Traits.

Enfin, nous nous introduirons dans un premier temps le concept des Traits, puis nous présenterons quelques refactorisations significatives qui ont pu être faites autour des Collections.

2.1 Composition de comportement en orienté-objet

2.1.1 L'héritage

L'héritage en programmation orienté-objet est un moyen bien connu permettant de créer de nouvelles classes en composant d'autres classes déjà présentes dans le système.

Un des avantages de l'héritage est qu'il permet de réutiliser du code efficacement lorsque les classes partagent suffisamment d'éléments communs. Ceci permet entre autres de réduire la complexité des programmes ainsi que de faciliter la compréhension du programmeur.

On associe aussi à l'héritage la notion de polymorphisme qui indique qu'un élément de comportement (par exemple, une méthode) peut disposer de plusieurs implantations sous-jacentes.

Ce mécanisme peut être utilisé dans le but d'obtenir plusieurs résultats. On peut regrouper ces utilisations sous trois catégories.

Spécialisation. La spécialisation est une des raisons essentielles de l'utilisation de l'héritage. On créé ainsi des classes spécialisées à partir de classes plus abstraites. Le mécanisme est théoriquement réitérable à volonté ; on appelle cette utilisation de l'héritage le sous-typage.

Avec la classe spécialisée, les instances disposeront des données et de comportement qui n'étaient pas prévues dans la classe mère, tout en profitant de ceux de cette dernière.

Prenons comme exemple la classe *Point* qui dispose d'un champ x et d'un champ y. Supposons maintenant que nous souhaitons obtenir un point coloré : *ColoredPoint*. Celui-ci va alors être un sous-type de *Point* et donc disposer des attributs de celui-ci tout en ajoutant la notion de couleur (via un autre champ et/ou une méthode).

Il existe aussi un autre type de spécialisation qui fait appel à la notion de classes abstraites. Ces dernières sont des classes où certaines méthodes ne sont pas implantées. Lorsqu'une classe concrète souhaite spécialiser une classe abstraite, elle doit alors donner un comportement aux méthodes requises. Le fait de donner du comportement par un sous-classage est connu sous le nom d'implantation.

Redéfinition. La plupart des langages orientés-objet permet aux classes de redéfinir l'implantation d'un aspect qui est hérité. Ce processus est appelé redéfinition.

La redéfinition introduit alors un problème : quelle version du comportement une instance doit-elle utiliser ? L'objet pourrait en effet appeler le comportement de sa propre classe ou celui de la classe héritée.

Il n'y a pas de réponse unanime à ce problème. En effet, les langages de programmation y répondent de manières différentes et certains langages permettent même de spécifier qu'un comportement particulier ne pourra pas être redéfini.

Réutilisation. Une des motivations initiales de l'héritage était de permettre qu'une nouvelle classe puisse réutiliser le code déjà existant d'une autre classe. Cette pratique est appelée sous-classage ou héritage d'implantation.

Cependant, dans la plupart des communautées, le sous-classage ayant pour unique but de réutiliser le code est considéré comme une mauvaise pratique. La première raison est que cette utilisation ne garantie en aucun cas un polymorphisme fiable, ce qui implique qu'on ne peut pas substituer fiablement une implantation par une de ses sous-classes.

Il est pourtant possible d'arriver au même but par une autre technique qu'on appelle délégation. Elle a pour avantage d'éviter de tromper l'utilisateur en évitant d'offrir une impression de polymorphisme. On notera cependant qu'elle nécessite plus d'efforts de la part du programmeur.

Dans la suite de cette section, nous présenterons les deux principales formes d'héritage : l'héritage simple et l'héritage multiple.

Héritage simple

L'héritage simple est actuellement utilisé dans de nombreux langages de programmation. En effet, depuis sa création avec Simula 67, cette idée a eu énormément de succès de part sa simplicité et de sa révolution.

Sa puissance en a fait un concept très prisé et est en autre à la base du succès de Smalltalk et Java.

L'héritage simple force la hiérarchie de classes à être linéaire en imposant un seul parent à chaque héritier. Ceci permet donc de réduire le nombre de problèmes qui pourraient surgir avec des hiérachies plus complexes, mais limite aussi certaines possibilités.

Duplication de code. La première conséquence de ces limitations est la duplication de code. Ce problème est le plus fréquemment rencontré avec ce type d'héritage. En effet, il n'est pas rare que dans les programmes l'un veuille composer deux comportements orthogonaux. Dans ce cas, à moins de réorganiser totalement la hiérarchie quand celà s'avère possible, le problème se résoudra souvent par une duplication de code.

L'exemple de la figure 2.1 illustre ce problème. Prenons le cas où nous avons une hiérarchie de véhicules. Comme on peut le voir, nous avons *WheeledVehicule* en tant que racine de la hiérarchie ; cette dernière étant spéciliasée vers des véhicules motorisés ou non. Nous aimerions donc partager le code de motorisation (*MotorisedVehicule*) pour tous les véhicules motorisés.

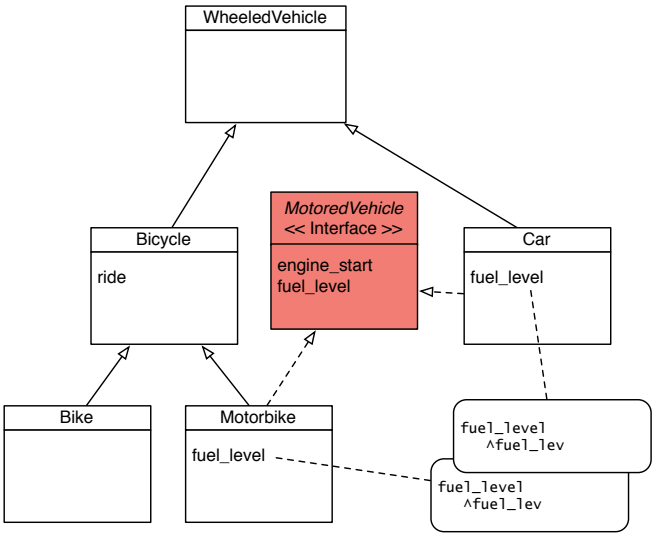

Fig. 2.1: Duplication de code avec le mécanisme d'héritage simple

L'héritage simple ici ne nous permet pas de réaliser cette opération. En effet, le seul moyen de partager le comportement serait de l'inclure dans un des ancêtres communs des classes intéressées. Cependant, ceci impliquerait que les autres sous-classes seraient aussi motorisées, ce qui est faux.

Voici donc une limitation de l'héritage simple où la seule alternative est la duplication de code : il sera nécessaire de doter *Car* et *MotorBike* du même code.

Le concept d'interfaces peut alors être utilisé afin de garantir que les implanta-

tions seront compatibles, mais ceci ne résoud en rien le problème initial car elles ne peuvent pas contenir de code.

Mauvaise conception par défaut. Un autre exemple que l'on peut rapprocher au précédent problème est la difficulté d'obtenir des objets parfaitement uniformes lorsqu'un système grossit. Prenons l'exemple de la classe *RectangleMorph* dans Squeak.

Cette classe représente un bloc de couleur à l'écran que l'on peut manipuler. Celle-ci est une sous-classe des différents composants de Morph, le système de fenêtrage de Squeak, et profite donc de tous les comportements généralistes que la hiérarchie de Morph offre.

Cependant, Squeak propose aussi dans son système une classe appelée *Rectangle*, sous-classe d'*Object*. Or, *RectangleMorph* n'est pas une sous-classe de *Rectangle*, alors qu'il est évident que ce premier est intraséquement un Rectangle. *RectangleMorph* n'implante donc pas tous les messages de *Rectangle* et ne peut donc être utilisé comme tel.

Ce qui est encore plus remarquable est que la classe *RectangleMorph* dispose d'un champ interne qui pointe sur un *Rectangle*.

Un programmeur qui souhaiterait résoudre ce problème est alors contraint à choisir parmi quelques alternatives non satisfaisantes d'un point de vue génie logiciel.

Sa première option est de copier toutes les méthodes de *Rectangle* vers *RectangleMorph*, mais ceci violerait le principe de DRY (Don't Repeat Yourself).

Une autre alternative est d'offrir une méthode de conversion, comme « *RectangleMorph*>>asRectangle » et de supposer que le développeur utilisera cette méthode quand il nécessitera d'envoyer des messages au rectangle.

Enfin, une troisième option est de déléguer les opératinons de *Rectangle*. Une manière simple de faire ceci est d'implanter toutes les méthodes de *Rectangle* sous forme d'une simple méthode au niveau de *RectangleMorph*. En voici un exemple :

```
RectangleMorph>>area
   ^ self asRectangle area
```

Il s'agit certainement de la manière la plus intuitive de faire pour l'utilisateur, mais ceci augmente fortement la taille de l'objet et ajoute un nombre non négligeable de méthodes inutiles, rendant la classe plus difficile à comprendre.

Conclusion. Il est donc clair ici que l'héritage simple, bien que proposant un mécanisme simple et fonctionnel, ne couvre pas tous les cas d'utilisations et peut parfois conduire à des concessions non désirées en terme de génie logiciel.

Héritage multiple

L'héritage multiple propose le même mécanisme que l'héritage simple mais ajoute la possibilité aux héritiers de disposer de plusieurs parents.

Avec l'héritage multiple, il est donc possible d'exprimer beaucoup plus efficacement les compositions de comportements voulues. Cependant, comme on peut l'intuiter, ceci apporte aussi son lot d'ambiguïtés et donc de problèmes liés à ce mécanisme. Nous y reviendrons dans la suite de l'analyse.

Depuis les années 80, il y a eu un nombre significatif de travaux autour de l'héritage multiple mais aucun d'eux n'a jamais été réellement satisfaisant. Certaines personnes en sont même venues à la conclusion suivante :

> *"Multiple inheritance is good, but there is no good way to do it"* – Steve Cook [Coo87]

En effet, les propositions d'héritages multiples ont souvent amenées plus de problèmes qu'elles n'en résolvaient. On peut d'ailleurs ressentir que la tendance revient actuellement à la simplicité avec les langages récents (Java, C#, ...). En effet, il semblerait que les concepteurs aient jugé que la complexité introduite par l'héritage multiple était trop importante comparée à l'apport réel de ce mécanisme.

Dans la suite de cette section, nous proposons d'étudier les principaux problèmes de l'héritage multiple.

Conflits de composition. Reprenons l'exemple proposé dans la section précédente : le *Rectangle* et le *RectangleMorph*. Le problème était qu'un *RectangleMorph* est en soit un rectangle mais qu'il ne sous-type pas *Rectangle*, résultat des contraintes de l'héritage simple.

Avec l'outillage de l'héritage multiple, nous pouvons désormais faire qu'un *RectangleMorph* hérite aussi de *Rectangle*. Nous composerions ainsi cette dernière et *BorderedMorph*, la super-classe de *RectangleMorph* qui est un élément graphique rectangulaire dans l'environnement Squeak.

Cette classe hériterait alors de deux ensembles distincts d'état et de méthodes ayant le même but : la gestion d'un rectangle (sa position par exemple). On a donc clairement ici un problème de composition que l'on peut difficilement résoudre sans outillage de la part du langage.

Le problème du Diamant. Un autre problème bien connu de l'héritage multiple est le fait qu'il introduit des hiérarchies d'héritage non linéaires. Ceci implique plusieurs choses : d'abord une difficulté d'implantation de la part des développeurs du langage, ensuite des problèmes au niveau conceptuel.

La difficulté principale liée à la non-linéarité est qu'il faut réussir à ordonnancer cette hiérarchie afin de la rendre exécutable. Dans la plupart des cas, c'est le compilateur qui va réaliser cette tâche et qui livrera une hiérarchie difficilement prévisible lors de l'écriture du code.

On voit ici que nous avons un problème de prédictabilité. En effet, si on introduit des changements dans la hiérarchie, il est possible que l'algorithme produise un autre ordre d'exécution qui pourrait modifier le comportement final.

Un des cas flagrant de problème de linérisation est appelée le *Problème du Diamant* ou *Fork-Join Problem*. Nous l'illustrons par l'exemple qui suit en nous appuyant sur la figure 2.2.

Prenons une classe mère appelée *Vehicle* qui possède une méthode *engine_start()*. Cette classe possède deux héritiers, à savoir *Boat* et *Car* qui surchargent la méthode précédemment citée.

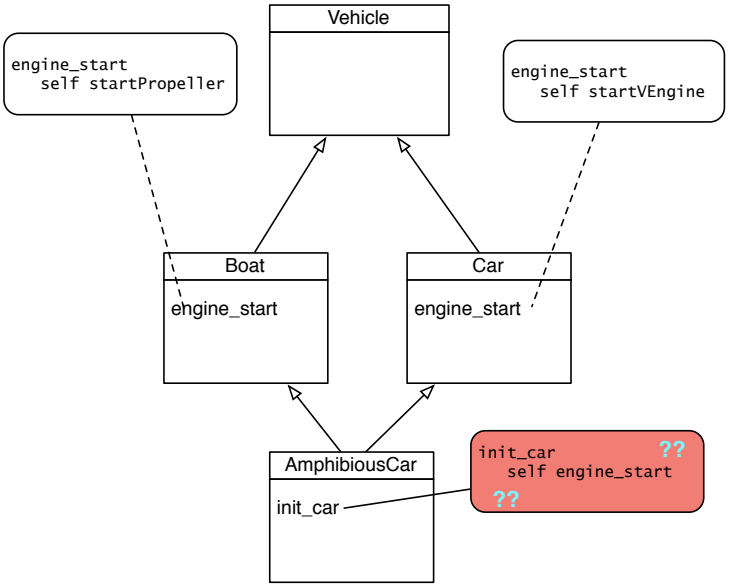

FIG. 2.2: Problème comportementale lié à la composition via le mécanisme d'héritage multiple

On souhaite alors composer ces deux classes vers un nouveau véhicule : *AmphibiousCar*. Nous nous trouvons alors face à un problème : quelle méthode *engine_start()* doit-on appeler ? Celle de *Boat* ou celle de *Car* ?

De nombreux travaux ont à ce propos été effectués et plusieurs langages ont alors proposé des solutions plus ou moins partielles à ce défaut :

- Perl et Python utilisent une liste de classes ordonnée depuis lesquelles hériter. Le compilateur utilise alors la première méthode qu'il trouve en effectuant une recherche en profondeur dans la liste des super-classes.
- C++ quant-à lui, demande au programmeur d'expliciter de quel parent l'élément attendu provient. On aura donc des références aux éléments telles que "Car : :Vehicle.Name" ;
- Eiffel permet aux programmeurs d'expliciter les éléments qui seront fusionnés ou non lors de la composition. Il fusionnera aussi automatiquement les éléments ayant le même nom et la même implantation. Eiffel permet aussi de renommer les éléments hérités afin de les distinguer ;
- CLOS donne au programmeur la possibilité de contrôler la combinaison des

méthodes et même de changer la couche réflexive via son Protocole Méta-Objets (MOP) dans le cas où il souhaiterait redéfinir la notion d'héritage, de dispatch ou encore d'instanciation.

Cependant, on notera qu'aucune des solutions citées ci-dessus n'est réellement satisfaisante car elles ne couvrent jamais tous les cas d'utilisations d'une manière générique.

Hiérarchie fragiles. Comme des méthodes ayant le même nom peuvent être héritées depuis la même classe, le fait de n'avoir qu'un mot clé « super » pour exprimer l'accès aux méthodes de classes héritées n'est pas suffisant.

C'est pourquoi, en C++, par exemple, le langage impose d'explicitement référencer la méthode redéfinie avec son nom de super-classe.

Ceci induit donc des références aux classes dans le code source et rend ainsi le code vulnérable aux changements de structures dans la hiérarchie de classes.

D'autres langages imposent une hiérarchie linéaire pour les super-classes. Cependant, une telle linéarisation mène souvent à des comportements innatendus.

2.1.2 Les mixins

A la suite des problèmes qu'apportait l'héritage multiple, des travaux connexes ont commencé à émerger afin de résoudre les problèmes de hiérarchie d'une autre manière.

On a d'abord vu apparaître les » Flavors« , introduites comme une extension de Lisp. On peut les considérer comme la première forme de Mixins.

Les Mixins sont des classes utilitaires qui regroupent des fonctionnalités

Hériter d'un Mixin n'a pas un but de spécialisation, mais il s'agit est un moyen efficace de collecter des fonctionnalités (groupes de méthodes). Les Mixins qui vont être utilisés par une classe vont alors être intégrés à la hiérarchie.

La figure 2.3 illustre ce mécanisme. Nous avons ici une classe nommée *AmphibiousCar* qui va composer les deux mixins *Car* et *Boat*. Les mixins seront donc intégrés à la hiérarchie

Le but des Mixins est donc qu'ils soient hérités pour la réutilisation de code et non pas instanciés en tant que classes indépendantes.

Le mécanisme de Mixins permet de résoudre la plupart des problèmes engendrés par l'héritage multiple. Cependant, il introduit à son tour des problèmes d'implantations tout aussi complexes ce qui, en fait, ne fait que déplacer la difficulté. Parmi ces problèmes, on remarquera particulièrement les problèmes d'ordonnancements qui se rapprochent des problèmes liés à l'héritage multiple. On peut d'ailleurs le voir sur la figure précédente : comment connaître le mixin qui précédera directement *AmphibiousCar* dans la hiérarchie une fois la linéarisation effectuée ?

De nombreux langages utilisent cependant les Mixins. On peut citer Perl, Python, D, Ruby ou encore StrongTalk.

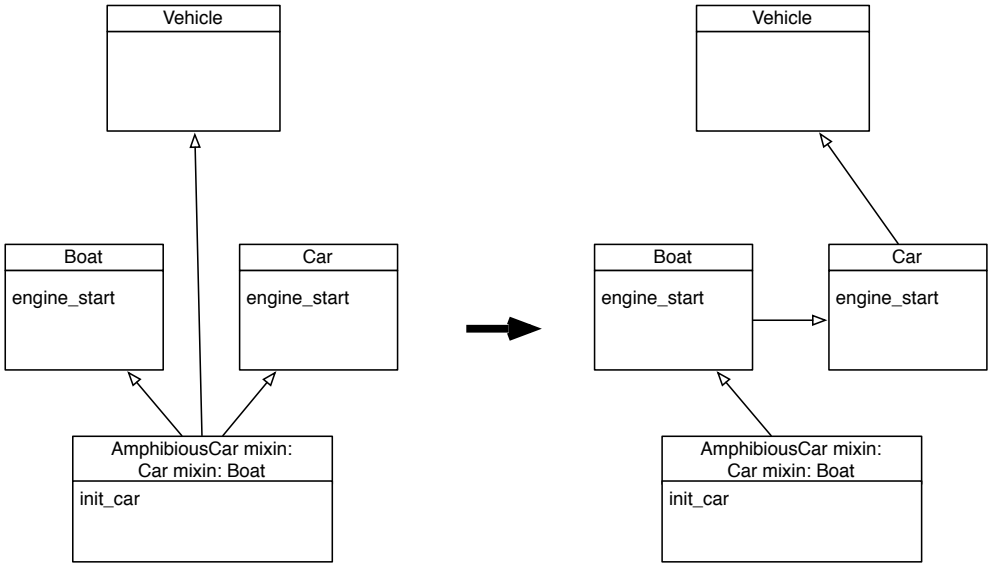

FIG. 2.3: Intégration des mixins dans la hiérarchie de classes.

D'autres langages comme Java ou C# n'implémentent pas les Mixins, mais offrent un mécanisme qui dispose d'une partie de leurs fonctionnalités : les interfaces. Ces dernières permettent non pas la réutilisation de code (car, contrairement aux Mixins, elles ne peuvent contenir du comportement), mais autorisent ainsi du polymorphisme.

2.1.3 Les traits

Dans cette section, nous proposons d'introduire les traits en commençant par présenter les motivations à travers les défauts des techniques actuelles, puis nous introduirons l'idée générale des traits et finirons sur une présentation plus détaillée des modèles proposés jusqu'alors.

Le mécanisme des traits est un moyen assez récent permettant de composer du comportement de manière flexible dans les langages à héritage simple.

Les traits partent du constat que les classes disposent de trop de responsabilités. En effet, on peut voir que ces dernières sont utilisées à la fois pour i/ la cration d'objets, ii/ la réutilisation de code.

Ces deux rôles sont en fait assez conflictuels. Tout d'abord, dans le premier cas on aimerait avoir des classes fournies tandis que dans le second, on aimerait avoir une granularité assez fine. Ensuite, la génération d'instances nécessite que chaque classe ait une unique place dans la hiérarchie tandis que les unités de réutilisation devraient être utilisables à n'importe quel endroit. On voit bien ici que ces deux rôles se composent assez difficilement.

Les traits utilisent donc un modèle qui permet de séparer les préoccupations et ainsi permettent de résoudre une bonne partie des problèmes jusqu'alors rencontrés avec l'héritage multiple ou les Mixins.

Ainsi, les traits souhaitent décomposer les classes en briques de bases destinées à être réutilisées, en proposant une représentation de première classe des aspects comportementaux d'une classe.

En partie grâce à ceci, les traits disposent de bonnes propriétés comme la réutilisation de code efficace, la suppression des indirections, une meilleure modularité ou encore une amélioration de la compréhension des hiérarchies de classes.

Les traits permettent aussi de doter le programmeur d'outils permettant de régler la plupart des problèmes récurrents liés aux méthodes usuelles. Ainsi, les compositions conflictuelles ou encore le problème du diamant peuvent être évités.

Plusieurs modèles de traits ont été proposés depuis 2002. Dans les paragraphes suivants, nous introduisons donc ces différents modèles, en parcourant les concepts en détail dans leur première incarnation, puis nous nous focaliserons sur les différences vis à vis de la proposition initiale lors de la description des modèles qui ont suivis.

Traits sans état

Les traits sans état [SDNB03] prennent l'approche de ne s'occuper que du comportement et non de l'état en supposant que celui-ci sera ajouté lors de la composition vers les classes.

Cette version impose donc qu'aucun n'état ne soit déclaré dans les traits et que tous les accès à l'état soient indirects. On devra donc écrire des accesseurs si l'ont souhaite référer à l'état.

Aussi, on notera qu'un trait ne dispose pas de super-classe et que si le mot-clé "super" est utilisé, il restera indéfini jusqu'à la composition vers une classe.

Structure. Les traits sont une collection de méthodes de première classe. Ces méthodes doivent être du comportement pur et ne peuvent référencer une variable d'instance directement.

On observe deux types de méthodes dans un trait comme illustré sur la figure 2.4 :

- Les *méthodes fournies* : Ce sont les méthodes qui se trouvent dans le trait et qui pourront être utilisées par les compositeurs[1] ;
- Les *méthodes requires* : Ces méthodes sont des méthodes qui sont nécessaires au fonctionnement des méthodes fournies. Elles pourront être résolues lors de la composition vers un autre trait ou vers une classe.

[1] On définit compositeur par une classe ou un trait qui compose d'autres traits. Le terme *composant* n'est pas utilisé pour éviter l'amalgame avec la programmation logiciel orientée composants.

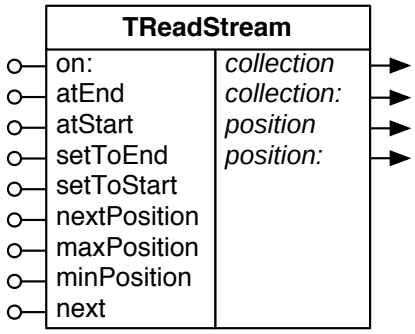

FIG. 2.4: Un trait avec ses méthodes fournies (gauche) et ses méthodes requises (droite). Le formalisme utilisé est une extension d'UML.

Propriétés fondamentales. Une propriété à la base des traits est l'applatissement. En effet, celle-ci énonce que la sémantique d'une classe définie avec les traits est exactement la même que celle d'une classe construite sans.

Ainsi, si une classe A est définie en utilisant le trait T, et que T définit les méthodes a et b ; alors la sémantique de A sera la même que si A définissait directement a et b.

De plus, cette propriété implique que le mot clé **super** n'a pas de signification particulière pour les traits et qu'il indique simplement que la résolution des méthodes commencera à la super-classe de la classe qui utilise le trait.

Une autre propriété, elle aussi très importante, est que l'ordre de composition n'est pas significatif. Ainsi, les conflits de méthodes des traits doivent être résolus au premier niveau de composition. On utilise alors les règles suivantes pour définir la précédence :

1. Les méthodes de la classe ont précédence sur les méthodes des traits ;
2. Les méthodes des traits ont précédence sur les méthodes de la super-classe (suivant la propriété d'applatissement).

Composition de classes. Bien que les traits semblent être très proches des Mixins, ils se différencient par le fait qu'ils n'utilisent pas l'opérateur d'héritage pour réaliser la composition.

Ils restent cependant totalement rétro-compatibles avec l'héritage simple en préférant lui être complémentaire que substitutif. L'héritage est alors utilisé pour dériver une classe vers une autre tandis que les traits sont utilisés dans un but structurel.

On peut résumer ces relations par l'équation suivante :

$$\textit{Classe} = \textit{Super-classe} + \textit{Etat} + \textit{Traits} + \textit{Ciment}$$

On va ainsi créer une classe à partir d'une super-classe, de nouvelles variables d'instances (si nécessaire), d'un ensemble de traits et de méthodes ciment. Ces dernières sont utilisées pour connecter les différents traits et accéder aux variables d'instances.

On dit alors qu'une classe est *complète* lorsque toutes les méthodes requises sont satisfaites.

On observe aussi plusieurs cas récurrents de composition des traits. On notera particulièrement la somme, la redéfinition ou encore l'héritage comme décrit sur la figure 2.5

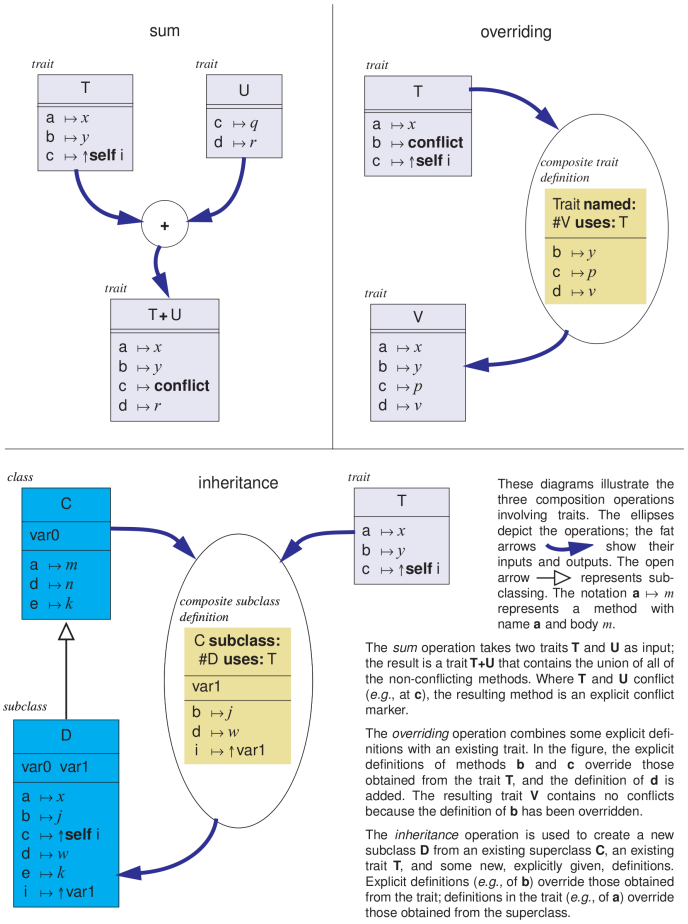

FIG. 2.5: Les opérations principales des traits : somme, redéfinition, héritage.

On notera qu'il est donc possible aussi de composer les traits entre eux et de réaliser ce qu'on appelle des traits composites. Ces derniers, contrairement aux classes sont souvent incomplets (i.e. des méthodes sont requises) mais gardent exactement les mêmes propriétés qu'un trait grâce à la propriété d'applatissement.

Résolution de conflits. Les traits se démarquent de l'existant dans le sens où la méthode veut que le programmeur ait le maximum de contrôle, et ce, le plus finement possible, lors de la composition.

Ceci est extrêmement important car ces cas de conflits arrivent fréquemment et il est nécessaire d'expliciter comment les résoudre pour éviter les suppositions automatiques comme nous l'avons expliqué dans les sections précédentes.

La composition des traits garantit que la résolution de conflits se fera au niveau du composite et non dans un sous-trait ; rendant ainsi la composition associative comme commutative.

Afin de permettre ces résolutions, on dispose des opérateurs suivants :

- **Aliasing** : L'Aliasing permet de spécifier un nom différent pour une méthode lors de la composition du trait auquel elle appartient. Ce n'est pas du renommage dans le sens où les autres méthodes restent inchangées (les méthodes utilisant l'ancien nom continue de le faire). Ceci se révèle très pratique lorsque, par exemple, on souhaite utiliser deux méthodes ayant le même noms, mais fournies par deux traits différents ;
- **Exclusion** : Avec cet opérateur, il est possible d'exclure une ou plusieurs méthodes des traits composés. Ainsi, il est possible de choisir quelle méthode sera utilisée lorsque plusieurs traits proposent deux méthodes de même nom ;

Ces deux opérations sont illustrées sur la figure 2.6

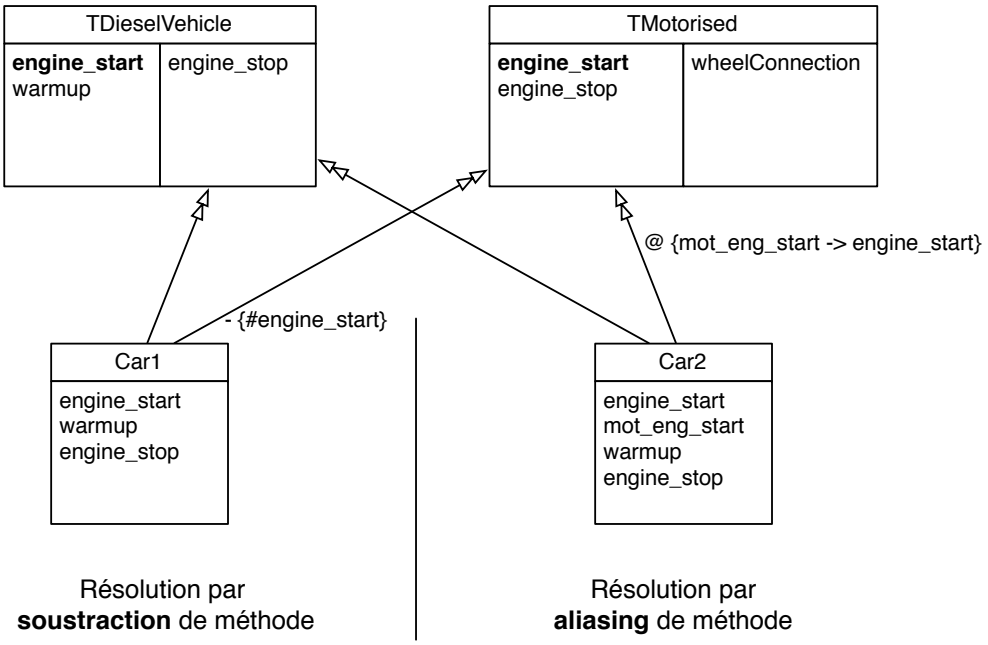

Fig. 2.6: Aliasing et Exclusion avec les traits.

Traits avec état

Bien que les traits sans état soient tout à fait utilisables en pratique, il s'avère que dans certains cas l'introduction d'état semble réduire l'effort de programmation.

Nous caractériserons tout d'abord les défauts qui ont menés à l'élaboration de ce nouveau modèle, puis nous introduirons les nouveaux concepts de ces traits et en viendront pour finir aux problèmes qu'ils induisent.

Défauts des traits sans état. Tout d'abord, les classes utilisant des traits sans état peuvent contenir beaucoup de code ciment lors de la composition de multiples traits (comme expliqué dans la section 2.1.3).

Deuxièmement, il très fréquent de voir des traits artificiellement incomplets à cause des nombreux accesseurs non définis. Ceci a pour désavantage majeur de polluer les interfaces avec des informations inutiles et donc de dégrader la compréhension.

Ensuite, l'introduction d'un nouvel état (et donc d'accesseurs) dans un trait propage automatiquement la nécessité d'implémenter cet accesseur dans toutes les classes utilisatrices. La figure 2.7 illustre ce problème : le trait *TSyncReadWrite* a besoin d'accéder à un verrou (« lock ») et on voit bien ici que les accesseurs ont besoin d'être dupliqués dans SyncFile, SyncStream et SyncSocket.

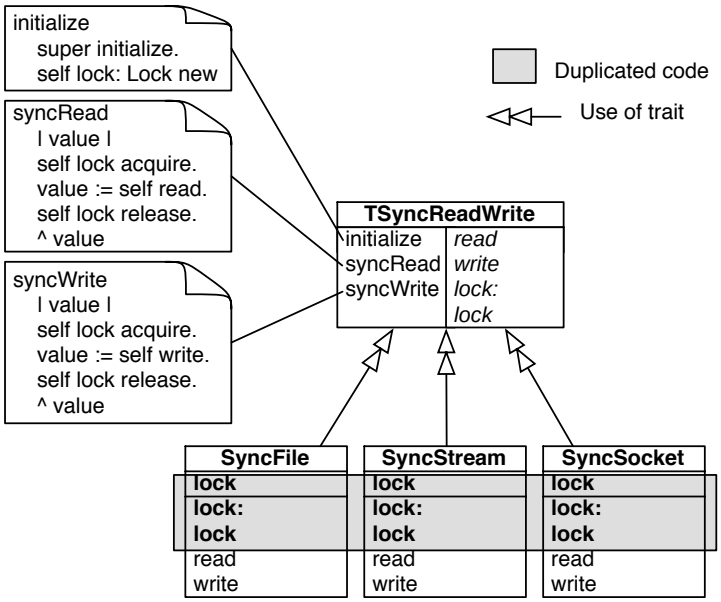

FIG. 2.7: La variable « lock » et les méthodes « lock » et « lock : » sont dupliquées au sein des utilisateurs du trait *TSyncReadWrite*

Pour finir, le fait de devoir ajouter des accesseurs pour chaque variable d'ins-

tance, à la base privée, viole le principe d'encapsulation de la classe.

Les traits avec état (« Stateful Traits » [BDNW08]) proposent donc une évolution des traits sans état, plus satisfaisante d'un point de vue génie logiciel, en introduisant la possibilité d'avoir des variables d'instance.

Cette évolution se veut la plus minimaliste possible et la plus proche de la philosophie d'origine, à savoir que le compositeur garde le maximum de contrôle sur la composition.

Nouveaux mécanismes. Les variables d'instance ainsi introduites ont alors une portée uniquement locale au trait auquel elles appartiennent. Il est cependant possible d'étendre la visibilité et d'exposer ces variables, mais uniquement sur demande explicite du compositeur.

Pour ce faire, l'utilisateur du trait peut donner l'accès à une variable du trait en la nommant. Ceci fonctionne comme l'aliasing des traits sans état : il ne s'agit pas de renommage et les méthodes du trait continuent d'utiliser l'ancien nom de variable.

Il est possible aussi de réunir deux variables qui ont la même sémantique. Ainsi, grâce à une nouvelle opération, l' « Union », le compositeur peut spécifier ces cas. Un exemple d'union est disponible sur la figure 2.8 où l'utilisateur, C, unifie x et y respectivement de T1 et T2 vers une variable d'instance w.

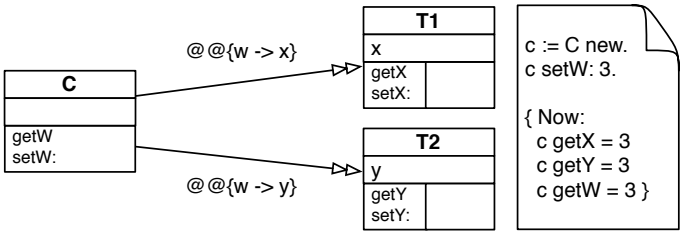

Fig. 2.8: Un exemple d'union de variables. Ici, x et y deviennent w pour C.

Renaissance de problèmes. L'introduction d'état dans les traits a été réalisée de manière à respecter au maximum les propriétés des traits sans état.

Cependant, un problème récurent des langages utilisant l'héritage multiple réapparaît avec les traits à état. En effet, les traits étant partagés entre plusieurs classes, il n'est donc plus possible d'accéder aux variables d'instance de manière linéaire (c'est à dire par position). On en revient donc à des problèmes de linéarisation comme décrits dans la section 2.1.1 qui ne sont pas triviaux à résoudre.

Afin de palier à ce problème, les traits a état utilisent la méthode de *copy down*. Plus de détails sont disponibles dans le papier original.

Bénéfices et observations. Il a été observé par expérimentations que les traits avec état permettent de i/ préserver l'encapsulation, ii/ réduire le nombre de

méthodes définies et ii/ réduire le nombre de méthodes requises (à corréler avec les accesseurs).

On notera cependant qu'ils introduisent une complexité non négligeable qui n'était pas présente dans les traits sans état.

Utiliser les traits à état à la place des traits sans état n'est donc pas systématiquement au vu de la complexité introduite.

Freezable Traits

Le modèle des traits gelables (« Freezable Traits » [DWBN07]) est le dernier modèle de la famille des traits a avoir été proposé.

Il part du constat que tous les conflits de compositions ne peuvent pas être réglés à l'aide des modèles précédants sans une réécriture partielle des méthodes conflictuelles.

Celui-ci propose donc de modifier légèrement le modèle original des traits en ajoutant quelques concepts.

Tout d'abord, il considère que le langage dispose de deux modificateurs d'accès : *private* et *public*, ce qui n'était pas le cas dans les versions précédentes.

Ensuite, les traits gelables ajoutent deux opérateurs de composition qui permettent de modifier cette visiblité lors de l'étape de composition.

Visibilité par défaut. Avec les traits gelables, le développeur doit alors spécifier une visibilité par défaut pour chaque méthode. Ces opérateurs de visibilité sont équivalents à ceux que l'on trouve avec Java ou C++ par exemple.

La sémantique de ces opérateurs de visibilité est la suivante :

- *Private* : une méthode dite privée ne sera accessible qu'à l'intérieur du trait auquel elle appartient. C'est pourquoi, la méthode est liée au trait dès sa définition et n'est pas visible en dehors. Les autres méthodes qui réfèrent cette méthode appelleront toujours cette dernière, même s'il y a composition avec d'autres classes ou traits.
- *Public* : une méthode dite publique est visible au niveau du compositeur et est liée tardivement, comme dans le modèle original des traits. La méthode peut être redéfinie dans l'élément compositeur et les appels à cette méthode depuis d'autres méthodes résulteront à l'appel de la méthode redéfinie.

Visibilité dynamique. Là où les traits gelables se démarquent des fonctionnalités des langages proposant du contrôle de visibilité est qu'il est possible de modifier cette visiblité lors de la composition grâce à deux nouveaux opérateurs : *freeze* et *defrost*. Le premier change la visibilité d'une méthode publique à privée tandis que le second effectue l'opération inverse.

La figure 2.9 illustre la sémantique de ces deux opérateurs :

- Sur (a), les deux méthodes publiques x entrent en conflit. Pour éviter cette situation, le compositeur décide de geler le x de *T1*. Ainsi, *T1 foo* devient

précocément liée à *T1 x*, ce qui signifie que c'est cette dernière qui sera toujours appelée.
- Avec (b), les deux méthodes publiques x entrent aussi en conflit et le compositeur décide de geler les deux. Ainsi, l'invocation de x depuis *T1 foo* est liée à *T1 x* et celui dans *T2 bar* à *T1 x*.
- Sur (c), les deux méthodes x sont privées. Le compositeur décide de dégeler les deux méthodes x. Ceci génère un conflit de composition qui est résolu en définissant une nouvelle méthode x dans la classe *Composer*. Ainsi, les invocations sont tardivement liées et appelleront donc la nouvelle méthode.
- Enfin, sur (d), la méthode *T1 x* est privée tandis que *T2 x* ne l'est pas. Le compositeur décide alors de geler *T2 x* et de dégeler *T1 x*. Ainsi, on se retrouve exactement dans le schéma inverse de (a).

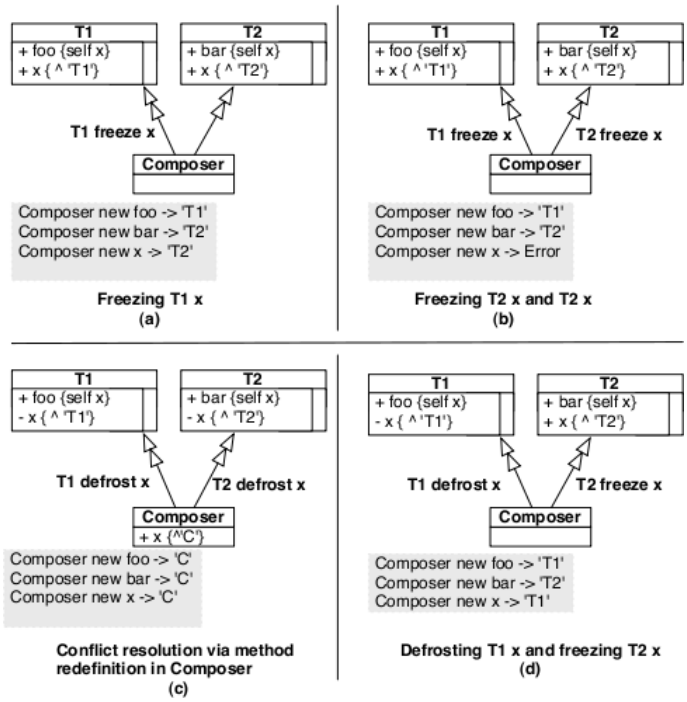

FIG. 2.9: La sémantique des opérateurs freeze et defrost

Différentes vues. Le fait que le compositeur décide de la visibilité finale des méthodes lors de composition implique qu'un même trait peut être vu de manière différente. Par exemple, la figure 2.10 illustre comment le trait *TSyncReadWrite* peut être utilisé par deux classes, *SyncStream* et *SequenceableStream*. Dans les deux cas, les classes n'ont pas les mêmes éléments gelés et donc pas la même vue sur le trait.

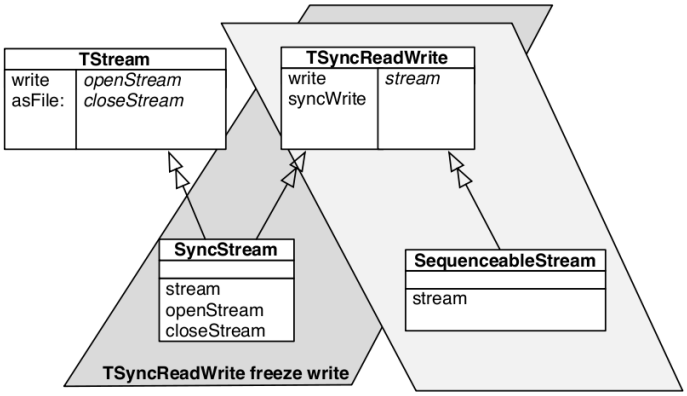

Fig. 2.10: Différentes vues d'un même trait

2.1.4 Synthèse

Nous avons ici exposés les principales catégories de mécanismes de composition de comportement en programmation orientée-objet.

Les types d'héritages basiques remontent aux débuts de l'orienté-objet mais perdurent toujours, même dans les langages actuels. En effet, ces mécanismes sont extrêmement éprouvés et les développeurs y sont grandement habitués. Cependant, ils ne sont pas exempts de défauts, comme nous avons pu le démontrer. Des deux modèles principaux (simple et multiple), les concepteurs de langages ont encore un choix cornélien à faire entre simplicité et expressivité.

Des initiatives ayant pour but de résoudre ces concessions ont alors vu le jour, comme les Mixins. Ces derniers permettent de résoudre certains problèmes liés à l'héritage multiple, en autorisant des hiérarchies parallèles dans les langages à héritage simple, mais sont loin d'être réellement efficaces en pratiques. De plus, ils introduisent leur ensemble de défauts, ce qui implique une complexité non négligeable.

D'autres propositions, comme les traits ont plus récemment fait leur apparition. Ils diffèrent de leurs prédécesseurs dans le sens où les compositions peuvent être réellement contrôlées, ce qui permet d'éviter un nombre important de défauts lors de l'utilisation de l'héritage multiple. En effet, les traits prennent l'initiative de non pas remplacer l'opérateur d'héritage mais de venir le compléter avec d'autres opérations.

Pour résumer, nous avons donc à notre disposition un panel d'outils intéressants nous permettant d'avoir plusieurs approches vis à vis de la composition de comportement.

Cependant, les traits se démarquent des autres solutions dans le sens où nous atteignons un niveau d'expressivité idéal en échange d'une faible contrepartie. Ils se combinent de plus très élégamment avec les langages à héritage simple, ce qui en fait un élément idéal pour la composition de comportement.

2.2 Refactorisations autour des traits et des collections

2.2.1 Interfaces and specifications for the Smalltalk-80 Collection Classes

Cette méthode de refactorisation [Coo92] part du constat que les hiérarchies de classes doivent être utilisées dans un but structurel et dans un but d'amélioration de la compréhension de la bibliothèque. Ainsi, l'héritage est réellement un mécanisme de production qui n'a pas de rapport avec l'utilisation de ces classes par les clients.

Le but est alors ici d'extraire des protocoles (= un ensemble de messages) plus intelligents à l'aide d'un programme qui prendra en compte tous les défauts de l'implantation existante. Ces protocoles seront compilés dans une hiérarchie.

Les défauts pris en compte sont les suivants : méthodes annulées dans les sous-classes et méthodes supprimées dans les sous-classes. En effet, il est possible de trouver des méthodes comme la classe *Interval*, qui ne permet pas d'ajouter un élément, mais qui pourtant dispose d'une méthode *add* :

```
add: newObject
  "Provide an error notification that
 adding to an Interval is not allowed"
 self shouldNotImplement
```

Hiérarchie de protocoles. La méthode propose un outil qui permet d'extraire une hiérarchie d'interfaces. Cette dernière est une représentation partiellement ordonnée qui permet d'organiser les interfaces partagées entre les classes d'une bibliothèque.

L'extraction est réalisée grâce à l'ordre partiel des protocoles en prenant en compte les relations de conformance. La conformance est définie par la règle suivante :

> *Une interface B se conforme à une interface A si chaque objet satisfiant B satisfait aussi A.*

Par exemple, si nous avons :

A = { isEmpty, at : }
B = { isEmpty, at :, add :, remove : }

Alors, **B** se conforme à **A** car les méthodes de **B** incluent les méthodes de **A**. Ceci signifie que les objets supportant les méthodes de **B** peuvent être substitués lorsqu'un objet de type **A** est requis.

L'algorithme introduit aussi la notion de *protocole abstrait* qui sont une représentation des messages partagés par deux protocoles. On va donc obtenir des représentations comme illustré sur la figure 2.11.

Ces protocoles sont appelés abstraits car ils ne sont pas implantés explicitement par des classes. Cependant, ils s'avèrent très utiles pour faire ressortir les similitudes.

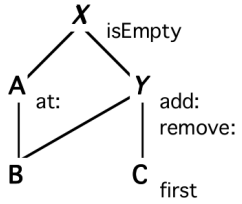

FIG. 2.11: Un exemple de hiérarchie de protocoles. Ici **X** et **Y** sont deux *protocoles abstraits.*

Spécification des classes. Une analyse plus poussée est alors utilisée pour réussir à appréhender plus finement les problèmes de conception. Il s'agit d'une analyse basée sur la spécification de comportement des classes et méthodes.

Chaque méthode est alors aussi annotée avec des pre et post conditions comme l'ont fait les concepteurs des bibliothèques d'Eiffel. Ceci met en lumière les problèmes suivants :

- Des méthodes sont héritées alors qu'elles devraient être annulées car elles violent l'invariant des sous-classes ;
- Des classes utilisent des méthodes ayant le même nom à des fins différentes : des noms différents devraient être utilisés ;
- Deux méthodes ayant des noms différents peuvent en fait réaliser la même chose : elles devraient donc être fusionnées.

Résultats. Le résultat de cette expérience est une hiérarchie d'interfaces très complexe (cf figure 2.12), même si Smalltalk est pourtant un langage à héritage simple. Il est donc clair que la hiérarchie actuelle est très restrictive comparée à la granularité attendue des différentes classes. De plus, les concepts partagés sont souvent orthogonaux, d'où la difficulté de créer une hiérarchie avec de l'héritage simple.

Aussi, un nombre d'erreurs ont pu être mises en évidences par la visualisation du graphe. Des opérations basiques ont ainsi été remontées dans la hiérarchie, la granularité s'est affinée pour améliorer la modularité (certains protocoles n'ont qu'un ou deux messages) et on voit l'apparition de protocoles abstraits (dus à la suppression des méthodes lors de l'héritage).

Il est à noter aussi que la hiérarchie s'est rapprochée sur certains points de la structure de la bibliothèque d'Eiffel.

Cependant, cette méthode a aussi montré que de nombreuses questions restent encore en suspend, même après la refactorisation. En effet, des incohérences conceptuelles se trouvent toujours facilement et un travail encore plus poussé serait nécessaire pour les effacer.

De plus, la comparaison de l'arbre d'héritage avec la hiérarchie de protocoles met bien en évidence le fait qu'il y a une grande disparité entre les deux modèles et qu'il est techniquement impossible, avec l'héritage simple, de réussir à calquer l'un sur l'autre.

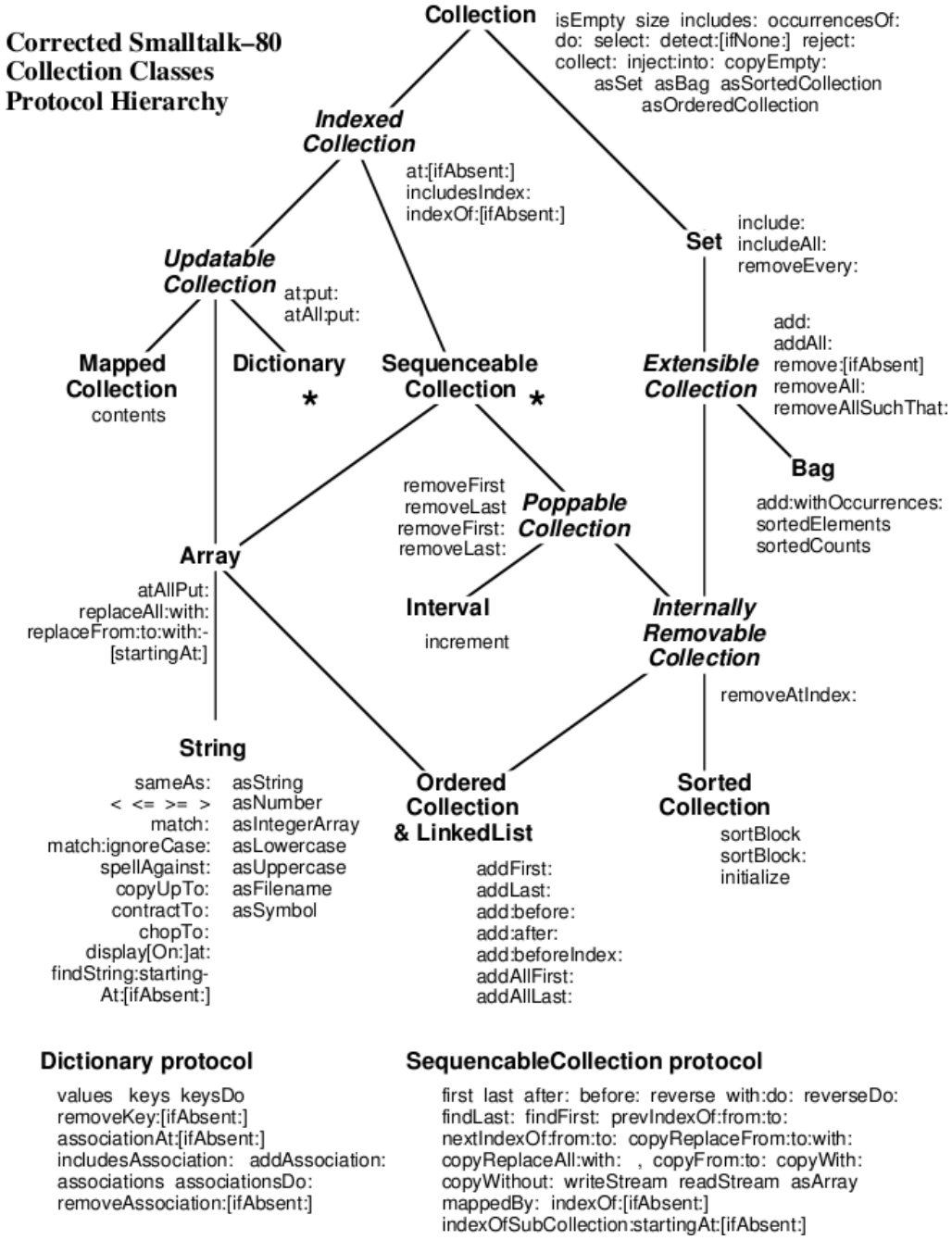

Fig. 2.12: Résultat de refactorisation par l'extraction d'interfaces

2.2.2 Applying Traits to the Smalltalk Collection classes

Ces travaux [BSD03] ont été réalisés dans le cadre de la création du modèle des traits, dans un but de preuve d'efficacité de ces derniers.

Ils proposent d'évaluer la hiérarchie de Collections de Smalltalk-80 à travers l'implantation Squeak et de la refactoriser à l'aider des traits sans état. Une description de cette hiérarchie est disponible dans la section 3.1.3.

Les travaux caractérisent tout d'abord les problèmes de la hiérarchie pour ensuite proposer un modèle qui permet de les résoudre.

Héritage inutile. Tout d'abord, le nombre de chaînes d'héritages dans lesquelles une méthode est redéfinie au moins trois fois a été comptabilisé. Bien que le fait de redéfinir une méthode n'est pas un mal (le mécanisme d'héritage ayant été en parti créé pour ceci à la base), la plupart des utilisations de ce mécanismes ne font en fait pas appel à la super-classe.

Ceci indique que ces classes ne paramétrisent pas une méthode plus générique mais corrigent en fait l'implantation, au lieu d'enrichir le comportement comme elles devraient le faire.

Ainsi, 164 méthodes ont été identifiées et donc autant de cas où un héritage n'était pas nécessaire. Ceci n'est pas problématique en terme d'exécution mais indique une perte de temps lors du développement. En effet, l'héritage est considéré comme étant un mécanisme d'aide à la compréhension dans le sens où le programmeur n'a qu'à comprendre les différences entre un parent et sa classe fille pour saisir l'objectif de cette dernière.

Si l'héritage est utilisé de manière incorrecte, il va donc remplir son rôle inverse : gêner la compréhension.

Duplication de code. Les travaux mettent aussi en avant des problèmes liés à la rigidité de l'héritage simple. En effet, dans certains cas, des classes aimeraient réutiliser des méthodes de classes existantes mais, à défaut, sont contraintes de dupliquer le code.

C'est le cas par exemple de *PluggableDictionary* et *PluggableSet* qui partagent plusieurs méthodes ayant le même but, mais qui pourtant ne peuvent avoir un ancêtre commun. *PluggableDictionary* est une sous-classe de *Dictionary* et *PluggableSet* est une sous-classe de *Set*. Nous observons donc qu'il n'y a alors aucune possibilité d'ancêtre commun car *Set* et *Dictionary* n'ont pas de points communs.

Défauts conceptuels. L'auteur de cette publication met aussi en avant cinq défauts conceptuels majeurs :

1. Beaucoup de collections sont présentes dans la bibliothèque car les concepteurs ont essayé de compenser le fait que les classes sont difficiles à réutiliser en fournissant toutes les combinaisons possibles de fonctionnalités ;
2. L'immutabilité n'est pas présente dans la hiérarchie des collections de Smalltalk-80 à l'exception de deux cas : *Symbols* et *Intervals*. Celle-ci pourrait pourtant être très utile dans de nombreux cas ;

3. Des classes présentent des préoccupations très orthogonales (read/write, binary/text) qu'il n'est donc pas forcément évident de composer. Les combinaisons sont alors exprimées en dupliquant du code. De plus, toutes les combinaisons ne sont pas disponibles ;
4. Certaines classes ne sont pas celles qui sont attendues (comme l'avait montré Cook [Coo92]) car elles n'implantent pas toutes les méthodes ;
5. Enfin, l'auteur souligne le fait qu'il existe des classes auquelles on souhaiterait donner un comportement de type collection, bien qu'elles n'en soient pas à première vue. Ca pourrait être le cas par exemple d'un Chemin qui représente une suite de points ; ce qu'on peut facilement associer à un type de collection.

Résultats. La refactorisation s'est concrétisée via 29 classes (dont 6 abstraites) et 52 traits. En moyenne, le nombre de traits utilisés pour construire une classe est de 5 ; le maximum étant 22. La hiérarchie résultante de ces travaux peut être consultée sur la figure 2.13.

La méthode utilisée pour arriver à cette hiérarchie a consisté par tout d'abord refactoriser 13 classes concrètes. Cette phase a permis de mettre en place la structure basique des traits fonctionnels et des traits d'implantation ; totalisant 46 traits.

Deux types de traits. Des traits fonctionnels et des traits d'implantations ont été utilisés pour réaliser la hiérarchie.

Ces premiers sont la matérialisation des caractéristiques des collections de la hiérarchie originale (ex : être ordonné). Ainsi, il est possible de combiner ces traits pour obtenir la combinaison voulue. Ces combinaisons sont disponibles sous deux formes : des traits composites ou des super-classes destinées à être héritées. La granularité a été aussi fortement affinée, même si cela n'était pas forcément nécessaire, de manière à assurer une ouverture du système et rendre ces éléments aisément modifiables. Aussi, les auteurs insistent sur le fait que ceci rend la compréhensible de la hiérarchie plus simple.

Afin d'identifier la nature de ces traits fonctionnels, ces derniers utilisent une convention de nommage où sont suffixées les lettres *S*, *U*, *M* ou *I*. *S* ou *U* indique que l'implantation doit être respectivement séquencielle ou non ; *M* indique la mutabilité, *I* l'inverse.

Les traits d'implantation quant à eux combinent les différentes fonctionnalités des traits fonctionnels avec les implantations compatibles (par exemple, une version implanté avec des vecteurs). Ainsi, par exemple, le comportement pour créer des nouvelles instances sera collecté dans un trait nommé *TInstanceCreationImpl* et sera utilisé par *TOrderedImpl* et d'autres traits d'implantation.

L'utilisation conjointe de ces deux types de traits est donnée en exemple sur la figure 2.14

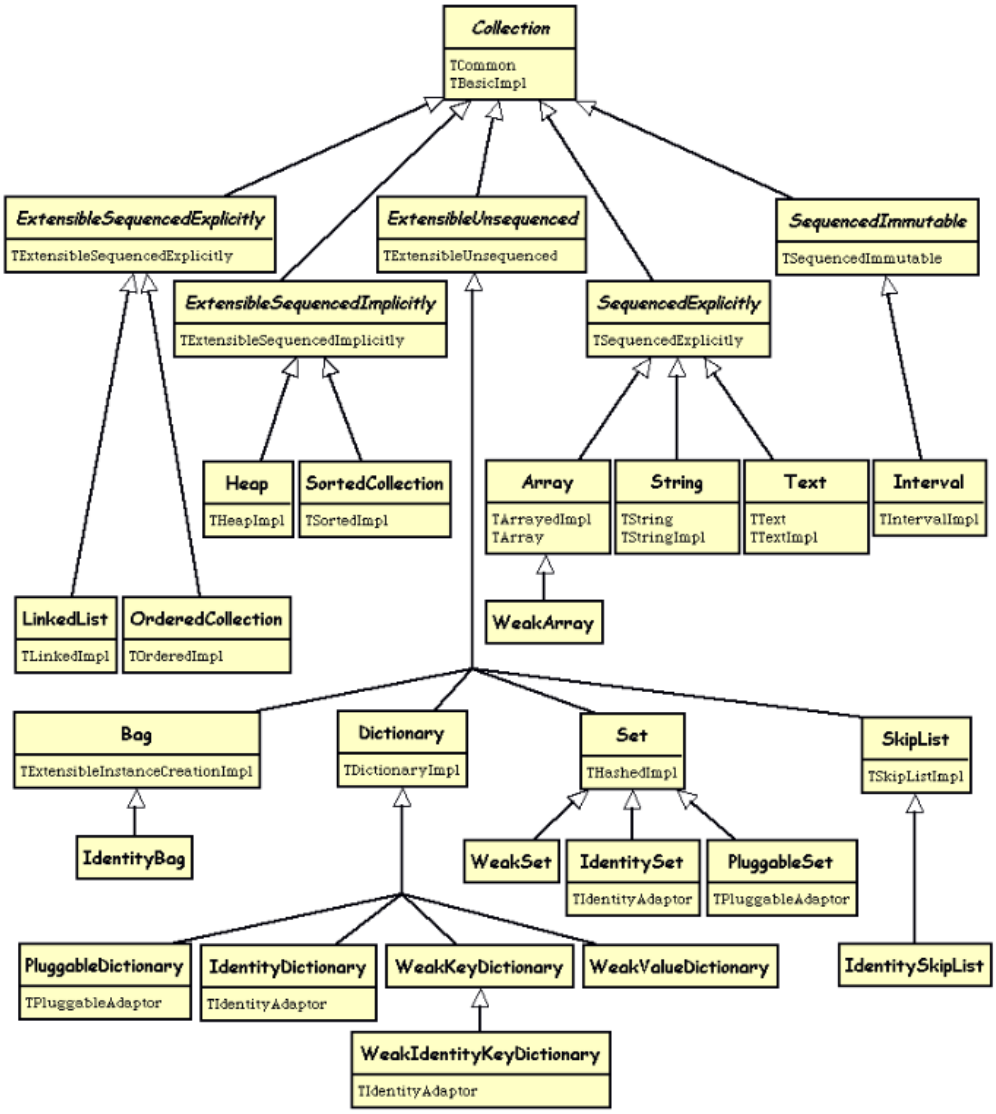

FIG. 2.13: Résultat de la refactorisation de la hiérarchie des collections de Smalltalk-80 avec les traits.

Enfin, les auteurs en concluent que les traits simplifient grandement la refactorisation et que les outils sont importants. En effet, ils ont remarqué que les outils de Smalltalk les ont aidé dans la démarche de refactorisation et que le fait d'avoir développé une extension de ces outils étant capables de comprendre les traits leur a aussi été bénéfique.

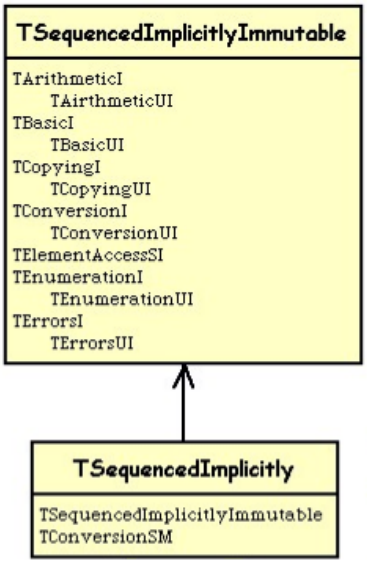

Fig. 2.14: Un exemple d'utilisation conjointe des traits fonctionnels et d'implantation.

2.2.3 Nile

Bien que Nile [CDW07] ne soit pas un travail autour des collections, il est intéressant d'étudier la réalisation de cette bibliothèque. Nile est le résultat du travail d'une reconception de la bibliothèque des streams à l'aide des traits sans état.

Le but de ces travaux était d'évaluer à quel niveau les traits permettent la réutilisation de code mais surtout d'étudier les problèmes émergents lors de la conception d'une nouvelle bibliothèque avec les traits.

Identification des problèmes. A travers l'analyse de l'ancienne hiérarchie de streams, l'auteur met en avant plusieurs défauts de conception liés à l'utilisation de l'héritage simple. En voici les principaux :

- **Méthodes implantées trop hautes** : une technique répandue permettant d'éviter la duplication de code est d'implanter des méthodes dans la super-classe commune aux classes qui nécessitent cet élément. Cependant, cette technique pollue les interfaces car certaines classes qui héritent de cette super-classe (souvent indirectement) n'ont pas besoin de ces méthodes. Nous aurons alors des méthodes annulées ;
- **État de super-classe inutilisé** : Dans certains cas, il arrive que des classes qui héritent d'une super-classe ne fassent jamais référence à l'état de cette dernière. Ceci n'est pas en soit un problème mais peut indiquer que la sous-classe n'utilise pas le code fournit par sa classe mère ;
- **Simulation d'héritage multiple** : Même avec les langages à héritage

simple, il est fréquent d'avoir besoin de mélanger plusieurs concepts orthogonaux afin de composer une nouvelle classe. Dans ce cas de figure, la seule solution est la duplication de code pour simuler cet héritage multiple car les classes ne partagent pas la même hiérarchie.

Bien que ces problèmes soient mis en avant par l'analyse des streams, on peut tout à fait les généraliser à la plupart des bibliothèques existantes, dont les collections.

Nile, le résultat. Nile propose un ensemble de traits formant le noyau de la bibliothèque. Les auteurs se sont basés sur le standard Smalltalk-ANSI pour ce noyau.

Il consiste en six traits, comme l'illustre la figure 2.15.

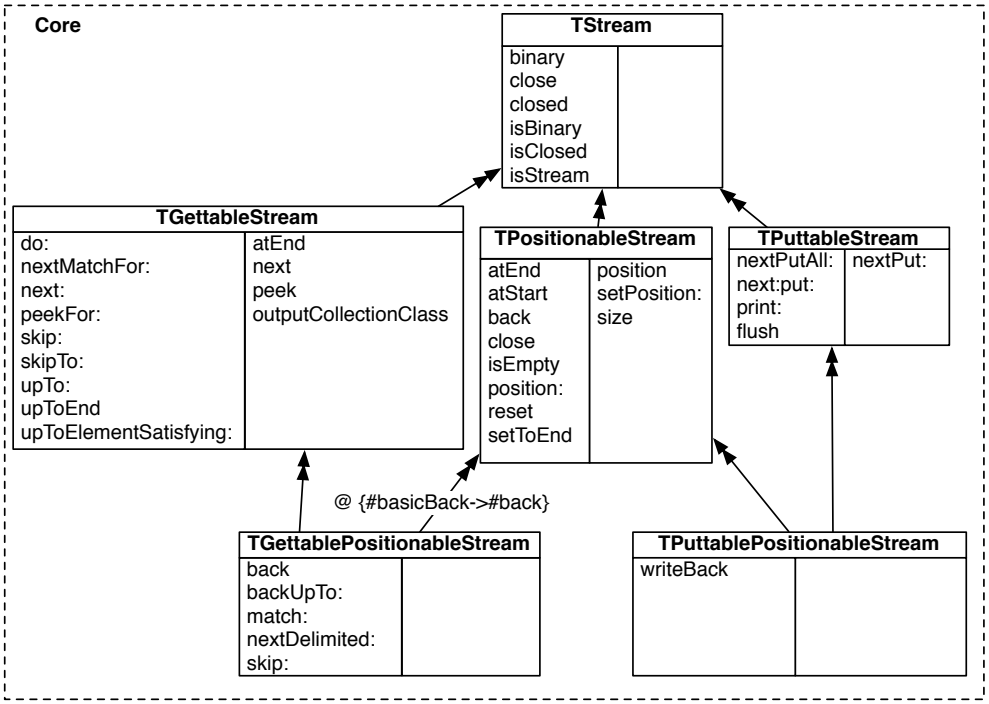

Fig. 2.15: Les six traits composant le noyau de Nile.

Un premier trait, *TStream* permet de regrouper toutes les fonctionnalités communes aux streams. La hiérarchie est ensuite divisée selon les éléments nécessitant une lecture (*TGettableStream*) ou une écriture (*TPuttableStream*). On notera aussi que les auteurs ont différencié le cas des streams nécessitant un positionnement absolu (*TPositionableStream*).

Les traits les plus bas de la hiérarchie permettent principalement de combiner les fonctionnalités précédemment citées.

Nile construit ensuite son implantation des streams autour de ce noyau, mais nous estimons que leur analyse n'est pas pertinente dans le cadre de notre étude car spécifique aux streams.

Preuve de modularité. Afin de démontrer la modularité introduite grâce aux traits, les auteurs ont implanté deux bibliothèques supplémentaires à l'aide du noyau de Nile.

La première permet de gérer un historique (telle que celui trouvé dans les navigateurs web), tandis que la seconde réimplante la bibliothèque originale de Smalltalk afin de rendre Nile rétro-compatible.

Évaluation des traits. Le choix de la granularité des traits s'est faite de manière intuitive en gardant en tête que les traits ne doivent pas être de trop fine granularité et que leur but n'est pas seulement de grouper des méthodes. Ainsi, des abstractions cohérentes, regroupées par trait, sont proposées.

Le fait d'avoir identifié les concepts orthogonaux et de grouper les autres permet d'obtenir de bonnes combinaisons et donc de maximiser la réutilisabilité.

Une discussion intéressante menée au cours de ces travaux est la suivante : quelle décision doit-on prendre entre la création d'une classe et l'ajout d'un ensemble de traits ? L'auteur en déduit qu'un bonne manière de décider est de se poser la question suivante : « est-ce que les futurs clients auront besoin de bénéficier de ces fonctionnalités ? ». Si oui, il faudra alors privilégier la création de traits.

Les traits ont permis une réduction de 40% de code ainsi que la suppression des problèmes d'annulations de méthodes.

Il faut aussi noter que la nouvelle hiérarchie de streams a été évaluée en terme de performance et qu'aucune perte de vitesse n'a été constatée.

Ceci démontre donc bien que les traits permettent une décomposition idéale et naturelle d'une hiérarchie de classes, tout en assurant un maintient des performances originales.

2.2.4 Synthèse

Les deux refactorisations effectuées avec les traits ont permis de montrer qu'ils sont des éléments idéaux pour modulariser une hiérarchie. En effet, la duplication de code a été supprimée, l'annulation de méthodes a été écartée et les possibilités de composition ont été étendues.

Le gain d'expressivité de composition est beaucoup plus important pour Nile que pour la première refactorisation des collections. En effet, avec Nile, la refactorisation s'est faite de manière moins systématique, ce qui a résulté d'une hiérarchie beaucoup plus extensible. On peut voir aussi que la manière d'utiliser les traits a évolué au cours du temps : les refactorisations récentes ont plus de recul sur la manière d'utiliser les traits pour refactoriser une hiérarchie.

La première refactorisation des collections a permis de mettre en avant les problèmes de conceptions (méthodes annulées, implantées trop hautes, ...) des hiérarchies. Nile confirme ces problèmes avec son analyse des streams bien qu'il s'intéresse plutôt à trouver un design intéressant et évolutif plutôt de recréer l'existant avec les traits.

A ce propos, Nile a commencé à s'intéresser à la cohérence avec la hiérarchie de protocoles de Smalltalk-ANSI. Quant à Cook, il tente par ses travaux d'améliorer ces protocoles afin de les rendre plus proches de la réalité d'implantation. Il regrette d'ailleurs de ne pas disposer de plus d'expressivité au niveau de l'héritage afin de pouvoir encore mieux rapprocher l'implantation.

Dans ces analyses, il a donc été très intéressant de voir comment les traits peuvent être habillement utilisés pour résoudre les problèmes habituels. De plus, il est clair que les traits peuvent apporter un complément très intéressant aux propositions qui, au moment de leur écriture, ne pouvaient pas tirer parti d'un tel mécanisme.

3 Analyse des bibliothèques de collections existantes

Cette chapitre propose tout d'abord de présenter les Collections de manière générale puis d'analyser deux cas concrets d'implantation des collections : Java puis Smalltalk. Ensuite, nous rapprochons ces deux implantations afin d'en faire ressortir les éléments communs et divergents. Enfin, nous conclurons en synthétisant les bonnes propriétés et nous caractériserons les problèmes posés par les collections afin de pouvoir mieux appréhender la difficulté.

3.1 Analyse des Collections

On peut définir les collections comme un ensemble de structures de données permettant le groupement et la manipulation d'éléments.

Avoir à disposition une bibliothèque de Collections dans un langage est essentiel et procure de nombreux avantages pour le développement d'applications.

Tout d'abord, ceci permet d'éviter la duplication de code dans le sens où les Collections sont très souvent utilisées. Nous évitons donc que chaque développeur réécrive sa propre implantation.

Ensuite, les Collections proposent un ensemble cohérent de classes, ce qui permet :

- Une réduction du temps d'apprentissage de la part du programmeur ;
- Une réduction du temps d'implantation de nouvelles structures, par réutilisation des briques de base existantes ;
- Une flexibilité des programmes, dans le sens où le programmeur peut aisément changer les structures de données utilisées sans effets de bords (et ainsi tester et améliorer les performances du programme).

Enfin, chaque implantation permet de capitaliser les connaissances du domaine et donc de fournir une version hautement optimisée et exempte de bogues.

Aussi, on trouve une variété importante de structures dans les Collections. En effet, chaque collection est propre à son cas d'utilisation et n'offre pas les mêmes propriétés à la fois sur le plan ensembliste que sur les aspects de performance.

On peut par ailleurs catégoriser les principaux types de collections comme suit.

3.1.1 Les classes de collections

Dans cette section, nous présenterons un panorama des classes de Collections existantes. Cette présentation ne se veut pas exhaustive, mais représentative de

la diversité des Collections usuelles.

Sets. Les *Sets* sont des ensembles non ordonnés où chaque élément est unique. Ils sont une implantation directe des ensembles mathématiques. Nous pouvons le définir comme suit :

Soit S un Set, a un élément
Invariant : $\forall a \in S,\ \nexists b \in S \setminus a$ t.q. $b = a$

Ils sont idéaux pour donner un accès rapide aux données et fournissent des opérations utiles telles que l'union, l'intersection, etc...

Les *Sets* peuvent être implantés de diverses manières :

1. Un arbre binaire équilibré, conférant une complexité $O(n)$ pour la plupart des opérations ;
2. Une table de Hachage, travaillant dans le pire des cas en $O(n)$;
3. Un tableau, utilisant une implantation simple et travaillant en $O(n)$.

Bags. Les *Bags*, ou *Multisets* se comportent comme les *Sets* à l'exception qu'ils peuvent accueillir plusieurs fois le même élément.

On les définit comme les Sets, à l'exception de la condition d'unicité.

On obtient de ces *Bags* la notion de multiplicité (le nombre d'occurrences d'un élément).

Lists. Les listes sont des collections de données où l'ordre des éléments est significatif

Avec cette structure, il est possible d'accéder aux éléments par index, de les parcourir dans l'ordre croissant/décroissant d'index, d'insérer ou de supprimer des éléments ou encore de mettre à jour un de ces derniers.

Les deux implantations courantes sont sous forme de listes chaînées ou sous forme de tableaux à taille variable.

Dans le premier cas, les éléments sont généralement accessible en temps $O(n)$ tandis que dans le second, nous obtenons une complexité en $O(1)$.

Arrays. Les tableaux sont des structures de données de taille fixe. Leurs éléments sont indexables par des entiers et sont donc ordonnés. Ceci n'implique pas nécessairement qu'ils sont triés.

Les tableaux sont très répandus dans les langages et sont donc extrêmement utilisés par les programmeurs.

Pour la plupart des implémentations, on accède les éléments en temps constant, c'est à dire en $O(1)$.

Graphs. Le graphe est une structure de données non-hiérarchique, directement associée à la notion de graphe mathématique. Cette structure, bien que peu courante dans la plupart des langages généraux, reste très intéressante. Un graphe est composé d'un ensemble de noeuds reliés par des arêtes qui établissent des relations entre ces premiers. On les définit comme suit :

Soit G un graphe, V l'ensemble des sommets, E l'ensemble des arêtes.
$G = (V, E, \varphi)$ où φ est une fonction $\phi : E \rightarrow V \times V$

Ces structures sont parfaitement adaptées pour représenter des données qui sont interconnectées de manière complexe entre elles.

On réalise souvent des opérations comme trouver un lien entre deux noeuds ou trouver la distance la plus courte d'un point à un autre à l'aide d'algorithmes comme celui de Dijkstra.

Encore une fois, cette structure peut être implantée de différentes manières : une liste d'adjacence ou encore une matrice d'adjacence.

Trees. Les arbres sont une spécialisation des graphes dans le sens où ce sont des graphes acycliques connexes. Il est aussi communément admis que leurs arêtes ne doivent pas être dirigées ni se voir affecter des poids.

On considère qu'il y a un noeud primaire, appelé "racine" et des noeuds terminaux, appelés "feuilles", en analogie avec les végétaux.

On effectue des opérations comme le parcours pre ou post fixé, la recherche d'éléments ou encore la section d'une branche.

Cependant, bien qu'ils soient directement une spécialisation des graphes, on notera que les structures de données utilisées pour représenter ces premiers ne sont pas du tout les mêmes que celles utilisées pour les graphes. La plupart des implantations utilisent des structures comme un "*Binary Space partitionning*", une pile ou encore des tableaux.

3.1.2 Les collections en Java

Cette section propose de présenter rapidement les Collections en Java. Bien que les travaux de ce rapport ne portent pas sur le langage Java, nous avons jugés intéressant d'analyser d'autres bibliothèques afin de pouvoir évaluer différentes implantations de Collections.

Il existe par ailleurs plusieurs implantations de Collections en Java. Nous nous focaliserons sur l'implantation officielle, c'est-à-dire celle inclue avec Java 5.0 : le *Java Collections Framework*[1] (JCF).

Le JCF a été conçu avec deux objectifs principaux : une taille assez réduite ainsi qu'une prise en main rapide pour le programmeur.

Ainsi, les Collections du JCF essaient de n'implanter que les méthodes les plus nécessaires, soit parce qu'elles sont essentielles, soit parce qu'elles permettent d'améliorer considérablement les performances.

[1] http ://java.sun.com/j2se/1.5.0/docs/guide/collections/

Cependant, la plupart des structures de données citées dans la section précédente sont proposées par le JCF ; à l'exception des structures très spécialisées comme les Graphes.

Enfin, le JCF permet d'assurer que les différentes collections puissent travailler ensemble efficacement.

Organisation et composition. La figure 3.1 présente la partie essentielle de la hiérarchie du *Java Collections Framework*.

Seules 9 interfaces composent le noyau des Collections, parmi lesquelles on retrouve *Set*, *List* ou encore *Queue*.

On peut remarquer que les *Map*s ne sont pas considérées comme des *Collection*s. En effet, les auteurs l'expliquent par le fait qu'ils considèrent qu'il s'agit de deux concepts orthogonaux :

> *"This was by design. We feel that mappings are not collections and collections are not mappings. Thus, it makes little sense for Map to extend the Collection interface (or vice versa)."* – Java Collections API Design FAQ[2].

On dispose ainsi de deux racines, *Collection* et *Map*, qui forment les Collections à elles deux.

Le *JCF* permet alors la conversion de l'un à l'autre sous forme de vues. Il est donc possible d'utiliser les *Collection*s en tant que *Map*s et inversement.

Enfin, le framework est principalement composé des éléments suivants :

- **Interfaces** : Le noyau du framework, représentant les types usuels ;
- **General Purpose implementations** : les implémentations génériques, réutilisables ;
- **Legacy implementations** : Les implantations de compatibilité avec les anciennes versions de Java ;
- **Concurrent implementations** : Des implantations spécifiques, dédiées à être accédées de manière hautement concurrentes ;
- **Wrapper implementations** : des implantations permettant d'ajouter des fonctionnalités, comme la synchronisation ou d'autres responsabilités ;
- **Algorithms** : Un ensemble de méthodes statiques permettant de réaliser des opérations tel que le tri ;

Ces implantations sont réalisées en fonctions des interfaces précédemment décrites. La figure 3.2 indique comment les interfaces usuelles sont utilisées.

Opérations non supportées. Le nombre d'interfaces du noyau étant volontairement assez réduit (dans le sens où il n'essaye pas de capturer tous les besoins), les développeurs ont donc choisi de proposer des méthodes optionnelles.

Avec ces méthodes optionnelles, il est possible de générer une *UnsupportedOperationException* permettant d'indiquer que l'implémentation ne supporte pas

[2] http ://java.sun.com/j2se/1.5.0/docs/guide/collections/designfaq.html

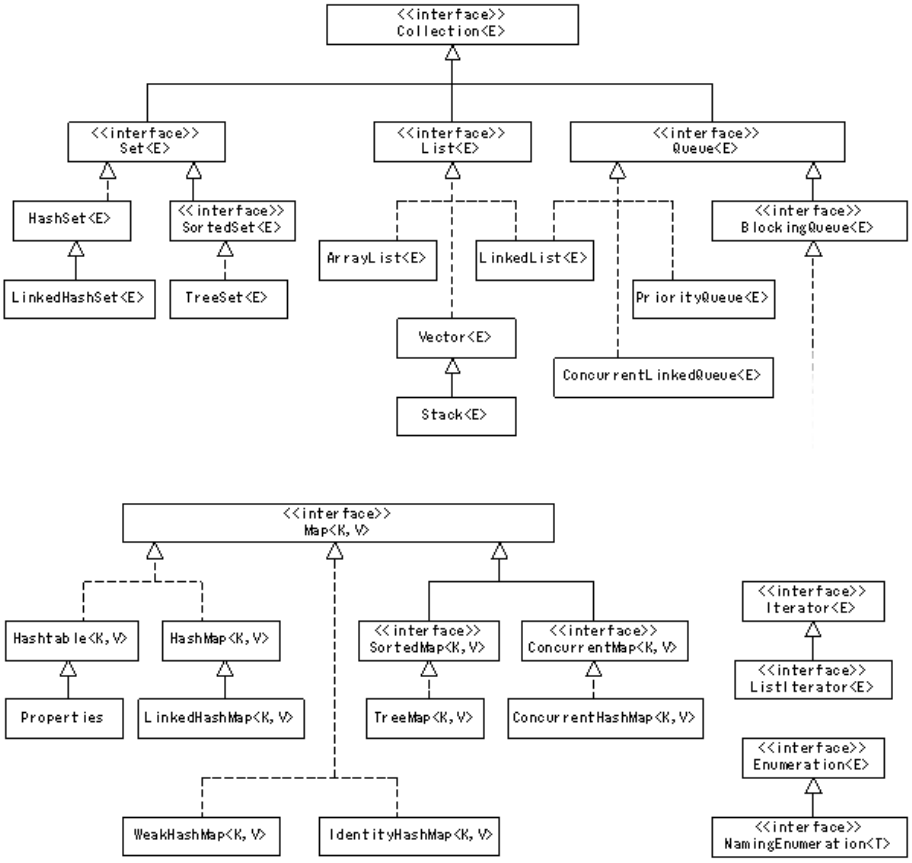

FIG. 3.1: Hiérarchie des Collections en Java 5

cette opération. La figure 3.3 décrit en UML les opérations possibles sur l'interface « Collection », racine de la hiérarchie. On peut clairement remarquer que toutes les opérations ne pourront pas être supportées dans tous les cas.

Comme la spécification est alors faite au niveau de la documentation, vu qu'il faut indiquer quelles sont les opérations implantées, on classifie alors les types de Collections selon les termes suivants :

- **Unmodifiable** : Les Collections qui ne peuvent pas changer ;
- **Immutable** : Ceci indique qu'en plus d'être non modifiable, l'état des objets de la Collection ne changera pas ;
- **Fixed-size** : Les Collections qui ont une taille fixe ;
- **Random Access** : On regroupe sous ce terme les Collections qui ont un temps d'accès rapide (i.e. en temps constant). Ceci permet aux algorithmes de pouvoir s'adapter en fonction du type de Collection à traiter.

Le cas des tableaux. Une des spécificité de Java est de ne pas proposer de type *Array* dans sa bibliothèque de Collections. Cela s'explique par le fait que

Itf \ Impl	HashTable	ResizableArray	BalancedTree	LinkedList	HashT.+LL.
Set	HashSet	∅	TreeSet	∅	LinkedHashSet
List	∅	ArrayList	∅	LinkedList	∅
Map	HashMap	∅	TreeMap	∅	LinkedHashMap

FIG. 3.2: Utilisation des interfaces dans les implantations usuelles. A la verticale, les interfaces ; à l'horizontale, les implantations.

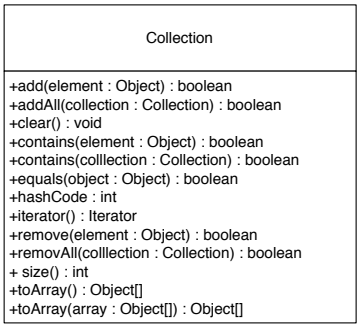

FIG. 3.3: L'interface Collection de Java 5. Des méthodes comme "add" seront annulées dans certains cas.

les tableaux se trouvent dans le langage (sous forme d'un type primitif) depuis le début et qu'il est donc impossible de les déplacer dans le *JCF*. Ceci casse bien entendu le contrat de compatibilité entre les différentes classes du *JCF* ; c'est pourquoi ce framework propose une classe utilitaire, appelée "Arrays" qui permet de transformer les tableaux natifs Java en Listes.

Découplage. Afin de contourner les problèmes de couplage fort à cause du typage statique, le *JCF* propose la notion d'*Iterator*. Ceci permet d'obtenir un couplage faible et donc de pouvoir changer rapidement d'implantation de Collection.

Voici un exemple d'utilisation :

```
Collection list;
Iterator elements = list.iterator();
while (elements.hasNext() )
   System.out.println( elements.next() );
```

Ainsi, dans ce code, nous pourrions avoir n'importe quel type de Collection au niveau de « list » : nous ne sommes donc plus couplés à un type particulier.

Un tour d'horizon plus détaillé du *Java Colletions Framework* peut être obtenu sur le site officiel de Java[3].

[3]http ://java.sun.com/j2se/1.5.0/docs/guide/collections/overview.html

3.1.3 Les collections en Smalltalk

En Smalltalk-80, les classes de collections désignent à la fois un ensemble de « Collections » et de « Streams ». Dans le cadre de ce mémoire, nous ne nous intéresserons qu'aux « Collections ». On se référera à l'analyse de Damien Cassou dans Nile [CDW07] en ce qui concerne les « Streams ».

Bien que le standard Smalltalk-80 définisse 17 classes pour Smalltalk, Squeak a été enrichi au cours des années et compte actuellement plus de 98 classes. Ces 28 classes originelles ont été maintes fois refactorisées jusqu'à obtenir le standard cité. On les considère d'ailleurs aujourd'hui comme un exemple de design orienté-objet.

Cependant, parmi les 98 classes de Squeak, beaucoup sont en fait des spécialisations (Bitmap par exemple). Il est possible de réduire le coeur de ces dernières à un total de 37 classes qui ne répondent à pas moins de 800 messages et définissent plus de 1000 méthodes.

Dans ces classes essentielles, on compte un certain nombre de classes abstraites, avec notamment *Collection*, *SequenceableCollection*, *ArrayedCollection* et *Integer Array*.

Ainsi, les Collections de Squeak sont donc actuellement très riches comme le décrit la figure 3.5. Il est possible d'extraire les classes les plus fondamentales pour obtenir l'arbre représenté sur la figure 3.4.

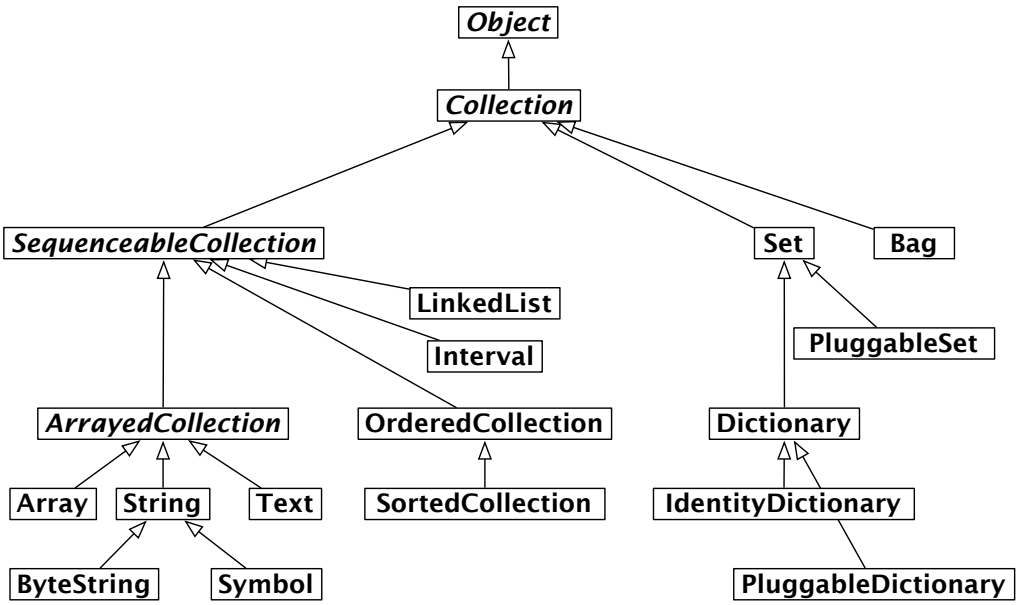

Fig. 3.4: Organisation des collections clés dans Squeak

Les opérations itératives. Suivant son héritage, Smalltalk a donc été inspiré par des langages comme Lisp. C'est pourquoi, les Collections ont été pensées

Collection
- **Bag**
  - **IdentityBag**
- CharacterSet
- *SequenceableCollection*
  - *ArrayedCollection*
    - **Array**
      - **WeakArray**
    - Array2D
    - ByteArray
    - ColorArray
    - FloatArray
    - IntegerArray
    - RunArray
    - **String**
      - Symbol
    - **Text**
    - WordArray
      - WordArrayForSegment
  - **Heap**
  - **Interval**
  - **LinkedList**
  - MappedCollection
  - **OrderedCollection**
    - **SortedCollection**
- **Set**
  - **Dictionary**
    - **IdentityDictionary**
    - **PluggableDictionary**
    - **WeakKeyDictionary**
      - **WeakIdentityKeyDictionary**
    - **WeakValueDictionary**
  - **IdentitySet**
  - **PluggableSet**
  - **WeakSet**
- **SkipList**
  - **IdentitySkipList**
- WeakRegistry

Stream
- AttributedTextStream
- *PositionableStream*
  - **ReadStream**
  - **WriteStream**
    - **LimitedWriteStream**
    - **ReadWriteStream**
      - RWBinaryOrTextStream
      - **Transcripter**
    - TextStream
    - TranscriptStream

FIG. 3.5: Listing des collections de Squeak. L'indentation indique un sous-classage tandis que l'italique signifie qu'il s'agit d'une classe abstraite.

pour être utilisées dans cet esprit. On ne va donc par exemple pas parcourir ces collections sous forme de boucles ; l'utilisation d'opérations comme « map » ou « apply » seront privilégiées.

Suivant cette idée, nous allons ainsi trouver des messages comme « select » ou « collect » qui permettent d'exprimer efficacement des opérations sur l'ensemble d'une Collection.

Voici un exemple qui sélectionne tous les éléments ayant un prix inférieur à 10 dans la Collection *aCollection* :

```
aCollection select: [ :anItem | anItem price < 10 ]
```

Au delà de la concision, ceci permet aussi d'exprimer une requête uniforme. En effet, il n'est pas nécessaire de connaître l'implantation sous-jacente : ceci fonctionnerait aussi bien sur une *LinkedList* qu'un *Array* dû au fait que toutes les sous-classes de *Collection* comprennent le message « select ».

Les protocoles. Afin de tirer parti de la puissance de cette uniformité, il est donc nécessaire que toutes les Collections définissent les mêmes messages. C'est ainsi que l'utilisation des protocoles permet d'assurer que si nous manipulons une Collection, alors nous disposons d'un ensemble de fonctionnalités de-facto.

On sait donc qu'un objet d'une collection va forcément être testable (« isEmpty » par exemple) ou encore donner la possibilité d'accéder à leurs propriétés (méthode « size »).

La plupart des protocoles des Collections de Smalltalk est regroupé dans la figure 3.6. Les méthodes définies dans ces protocoles sont alors définies, redéfinies ou encore optimisées selon les implantations nécessaires. Il est à noter qu'elles sont parfois aussi annulées lors de sous-classages.

Cohérence. Smalltalk propose une grande cohérence que ce soit au niveau de son langage ou de ses bibliothèques. Les Collections n'échappent pas à la règle.

Tout d'abord, les méthodes de créations restent les mêmes quelle que soit la nature de la Collection. Nous en disposons de quatre :

1. utilisation d'une collection constante. Ce sont des collections disponibles dans tout le système et accessibles depuis n'importe quel endroit ;
2. création par copie ;
3. conversion depuis une autre collection ;
4. création explicite.

Uniformité apparente. Bien que les Collections suivent les mêmes protocoles, l'uniformité n'est qu'apparente. En effet, les classes de Collections peuvent n'implanter que certains protocoles ou, comme indiqué sur la figure 3.7, être implantées à l'aide de diverses structures et donc exposer des codes métiers différents.

Voici une liste des critères sur lesquels nous pouvons différencier les sous-classes de *Collection* :

Protocole	**Méthodes**
accessing	size, capacity, at : *anIndex*, at : *anIndex* put : *anElement*
testing	isEmpty, includes : *anElement*, contains : *aBlock*, occurrencesOf : *anElement*
adding	add : *anElement*, addAll : *aCollection*
removing	remove : *anElement*, remove : *anElement* ifAbsent : *aBlock*, removeAll : *aCollection*
enumerating	do : *aBlock*, collect : *aBlock*, select : *aBlock*, reject : *aBlock*, detect : *aBlock*, detect : *aBlock* ifNone : *aNoneBlock*, inject : *aValue* into : *aBinaryBlock*
converting	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection : *aBlock*
creation	with : *anElement*, with :with :, with :with :with :, with :with :with :with :, withAll : *aCollection*

FIG. 3.6: Quelques protocoles standards des Collections

Arrayed Implementation	**Ordered Implementation**	**Hashed Implementation**	**Linked Implementation**	**Interval Implementation**
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

FIG. 3.7: Des exemples de classes réparties selon leur implantation

- **Séquencialité** : Bien que toutes les sous-classes de *SequenceableCollection* traitent leurs éléments dans un ordre défini, les classes **Set**, **Bag** et **Dictionary** n'ont pas de notion de suite. Ces deux types de classes sont synthétisées dans la figure 3.8 ;
- **Triable** : Seules certaines Collections maintiennent un ordre particulier, défini par l'utilisateur, au sein de leurs éléments ;
- **Indexable** : Certaines Collections, comme les *Array*s sont indexables, c'est à dire qu'ils est possible de référencer leurs éléments via un index (i.e., un entier). Des structures telles que *LinkedList* ne permettent pas ce genre d'accès ;
- **Accesible par clé** : Seuls *Dictionary* et ses sous-classes sont accessibles via une clé ;
- **Modifiable** : Toutes les Collections ne sont pas modifiables, bien que la plupart le soit. *Symbol* ou *Interval* ne le sont pas par exemple.
- **Taille variable** : Les collections telles que *Array* sont de taille fixe, il n'est

Sequenceable		Not Sequenceable	
Accessible by Index	**Not Indexable**	**Accessible by Key**	**Not Keyed**
Interval SortedCollection Array ByteArray OrderedCollection String Symbol	LinkedList SkipList	Dictionary IdentityDictionary PluggableDictionary	Set IdentitySet PluggableSet Bag IdentityBag

FIG. 3.8: Critère de séquencialité des classes de collections de Smalltalk.

donc pas possible de changer le nombre d'éléments au cours du temps sans recréer une nouvelle instance. D'autres sont capables de s'adapter à la demande ;

- **Multiplicité** : Il est possible que certaines collections acceptent des éléments dupliqués alors que d'autres non (respectivement *Bag* et *Set*) ;
- **Hétérogénéité** : La plupart des Collections vont pouvoir comporter des éléments de nature différentes. D'autres, telles que *String*, ne pourront quant à elles comporter que des *Character*s.

Pour plus de précisions, on se référera aux chapitres « Collections » des livres « Squeak by Example » [BDN+07] et « Inside Smalltalk : Volume 1 » [LP90].

3.2 Synthèse et identification des problèmes

Le but de cette section est de synthétiser les informations autour des collections afin i/ d'extraire les problèmes auxquels nous devrons faire face et ii/ d'extraire des modèles existants les bonnes et mauvaises pratiques.

Nous commençons, dans un premier temps, par énoncer les généralités conceptuelles des collections. Dans un second temps, nous rapprochons nos deux cas d'étude, à savoir Java et Smalltalk puis nous terminons par exposer les différences fondamentales de leurs deux hiérarchies de collections.

3.2.1 Généralités conceptuelles

Hétérogénéité de classes. Une des propriétés remarquables des collections est que les classes partagent un rôle semblable, c'est à dire maintenir un ensemble d'éléments, mais qu'elles sont très diversifiées quant à leur manière de le faire.

Nous avons donc très souvent affaire à de nombreuses classes différentes. Ces classes combinent et partagent cependant de nombreuses caractéristiques, comme la séquencialité ou encore l'ordre.

Il est aussi à noter que ces classes de collections peuvent très souvent être implantées à l'aide de plusieurs structures de données. L'hétérogénéité est donc elle aussi présente au niveau de l'implantation.

Utilisation cohérente. Bien que nous ayons un grand nombre de disparités dans les classes, une bibliothèque de collections essaye toujours de masquer au maximum ces différences aux yeux de l'utilisateur.

Tout d'abord, ceci apporte une manière cohérente de manipuler ces classes, réduisant ainsi le temps d'apprentissage. Ensuite, disposer d'interfaces consistantes permet d'assurer un bon polymorphisme au sein du système. Il est donc alors possible d'écrire plus facilement du code générique et d'améliorer l'évolutivité des programmes.

3.2.2 Similitudes

Bien que Java et Smalltalk soient deux langages assez éloignés, il est intéressant d'analyser les similitudes qu'ils présentent au niveau de leurs bibliothèques de collections.

Caractéristiques similaires. Les collections, au delà de leurs implantations, exposent très souvent des caractéristiques similaires qui peuvent être catégorisées. A l'exception de quelques catégories, nous avons pu assimiler les notions de Smalltalk à celles de Java assez facilement.

Il en résulte donc que les caractéristiques attendues de la part d'une bibliothèque de collections, bien que nombreuses, semblent faire consensus.

Annulation de méthodes. Un défaut commun des deux bibliothèques que nous pouvons relever est l'annulation de méthodes (le terme « méthodes optionnelles » est utilisé en Java) . Ceci est dû au fait que les deux langages utilisent l'héritage simple. A cause de cette contrainte, les concepteurs ont du remonter des méthodes dans la hiérarchie de classes afin d'éviter la duplication de code et rendre l'ensemble plus polymorphique.

Cependant, annuler une méthode n'est pas anodin : l'interface affichée par la classe n'est plus représentative des fonctionnalités qu'elle fournit.

3.2.3 Divergences

De part leur nature différente, il est évident aussi que les bibliothèques de Java et Smalltalk proposent des hiérarchies ayant des différences notoires. Nous exposons ici ces différences et évaluons les deux implantions le cas échéant.

Unicité de hiérarchie. Java a pris la décision de diviser la hiérarchie des collections en deux : nous avons alors une racine pour les « maps » et une pour les autres collections. La justification apportée par les concepteurs reste assez floue, même si on peut la comprendre en partie.

Il est possible de faire le parallèle avec l'opération d'itération de Smalltalk, #do :. Comment celle-ci doit-il réagir lorsqu'il s'agit d'une map ? Doit-on parcourir les clés, les valeurs ou le couple des deux ?

Java propose alors des conversions à l'aide de vues afin de résoudre le problème de polymorphisme.

Avoir une seule hiérarchie permet cependant de garder un système homogène.

Caractéristiques spécifiques. Java propose des caractéristiques qui ne sont pas disponibles dans Smalltalk comme par exemple, « Immutable ». Cette propriété est intéressante, mais très difficile à fournir, surtout dans un système hautement réflexif comme Smalltalk. Java résout ce problème en proposant une vue.

Java propose aussi des implantations spécifiées, comme des versions des collections hautement concurrentes.

Types primitifs. Java propose des types primitifs pour certaines de ses collections. Ainsi que le type « Array » n'entre pas dans la hiérarchie des collections.

Ceci a le double désavantage de ne pas pouvoir profiter des propriétés polymorphiques ni de pouvoir le manipuler en tant qu'objet. Pour apporter une solution à ce dernier point, les concepteurs ont introduit une classe utilitaire permettant de manipuler les tableaux à travers des méthodes statiques.

Cependant, ces classes utilitaires ont du mal à trouver leur place au sein de la hiérarchie des collections et il s'agit ici clairement d'un défaut de conception à éviter.

Protocoles. Smalltalk, contrairement à Java, définit une hiérarchie de protocoles en plus de sa hiérarchie de classes. Ceci permet de standardiser le comportement des méthodes et de clairement définir les caractéristiques. Ces protocoles sont définis dans la norme ANSI.

Iterators. Une grande différence au niveau des classes de collections entre Java et Smalltalk est que le polymorphisme en Java est assuré au travers des itérateurs.

Smalltalk, lui, utilise les messages de type #do : comme structures de contrôle. C'est pourquoi, l'approche est ici radicalement différente.

Cependant, les itérateurs ont le défaut de violer l'encapsulation car ils accèdent directement aux objets de la collection.

3.2.4 Conclusion

De part cette analyse, nous pouvons déduire qu'une bonne hiérarchie de collections assure un polymorphisme généralisé, une bonne capacité d'expression vis à vis de la combinaison des caractéristiques et s'assure de minimiser le nombre de concepts.

Ceci doit se réaliser en évitant tout d'abord les problèmes singuliers à l'héritage simple (annulation, duplication) ; en prenant en compte l'orthogonalité des caractéristiques et enfin ; en assurant une prise en compte de l'hétérogénéité des structures de données.

4 Des traits-protocoles pour les Collections

4.1 Une hiérarchie complexe et hétérogène

Notre travail consiste donc à proposer une hiérarchie de collections idéale à base de traits en s'appuyant sur les collections de Smalltalk-80 ainsi que d'évaluer les traits lors de ce processus de refactorisation.

Protocoles. Comme nous l'avons précédemment expliqué, les collections sont des structures qui nécessitent très souvent d'utiliser et de composer des concepts orthogonaux. Elles ont donc très souvent tendance à inciter la composition de plusieurs éléments, ce qui n'est pas réalisable dans les langages à héritage simple, comme nous l'avons montré dans la section 2.1.1. C'est d'ailleurs pour cette raison que les spécifications proposent ce qu'on appelle des « protocoles », où il est possible d'exprimer cette composition n-aire.

Ceci permet alors de spécifier une bibliothèque plus finement, mais introduit deux vues dans la hiérarchie : la vue « hiérarchie de protocoles » et la vue « hiérarchie de classes » ; comme la figure 4.1 l'illustre.

Nous proposons donc tout d'abord de supprimer cette dualité.

Structures de données hétérogènes. L'approche de Nile vis à vis des traits démontre que ces derniers sont capables de générer des hiérarchies de classes où la réutilisation et la structuration sont optimales. En effet, la hiérarchie proposée identifie clairement des traits pour chaque caractéristique des Streams et permet une composition de ceux-ci afin d'exprimer tous les besoins nécessaires.

Cependant, la refactorisation des Collections se démarque vis à vis des Streams. En effet, dans ces derniers, la plupart des objets manipulés disposent de la même implantation sous-jacente, ce qui facilite la composition des traits.

Ce problème est en fait une des difficultés majeures lors de la refactorisation de bibliothèques disposant de nombreux éléments hétérogènes, comme les collections. Même si le développeur réussi à identifier les traits idéaux d'un point de vue modularité, il n'est en rien assuré que ceux-ci seront fonctionnels une fois composés.

Nous illustrons ce problème sur la figure 4.2. Supposons que nous avons deux types de collections : une liste chaînée (*LinkedList*) et un tableau triable (*SortableArray*). *LinkedList* fait usage d'un trait nommé *TGrowable* qui indique qu'il

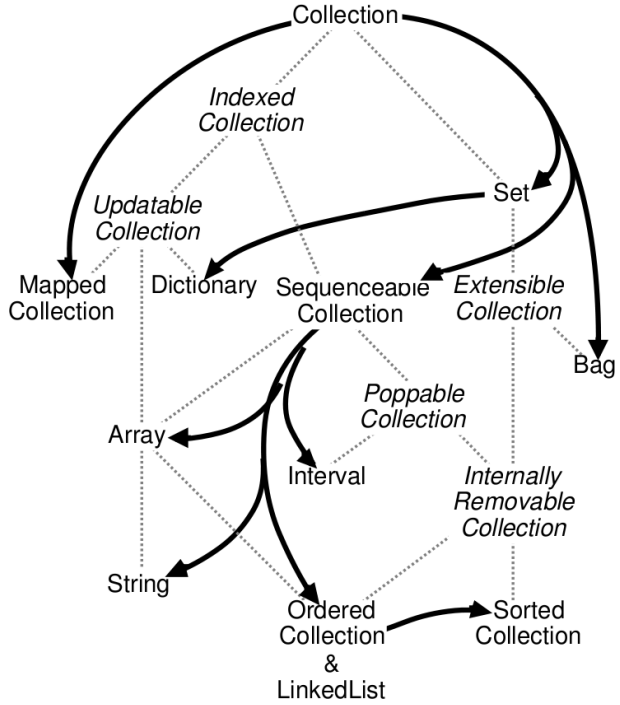

FIG. 4.1: Différences notoires entre protocoles (traits en pointillés) et implantations (traits pleins) [Coo92].

est possible de faire grossir cette collection ; *SortableArray* d'un trait nommé *TSortable* qui se charge de trier les éléments.

L'implantation de « *TGrowable*>>add » sera donc la suivante :

```
TGrowable>>add: anElement
  self elements last append: (LLNode with: anElement)
```

Et l'implantation de « *TSortable*>>sort » ressemblera à quelque chose comme :

```
TSortable>>sort
  Quiksort onArray: self elements
```

On remarque donc que ces deux implantions sont propres aux structures de données sous-jacentes. Par exemple, dans le premier cas, afin de faire grossir la structure de données, nous ajoutons explicitement un *LLNode* en fin de liste. Si nous devions travailler sur un tableau, cette fonction échouerait.

Ainsi, lorsque nous allons, par exemple, vouloir créer une liste chaînée triable (*SortableLinkedList*), nous allons naturellement vouloir composer *TGrowable* et *TSortable*, ceci s'inscrivant parfaitement dans la logique d'utilisation traits.

Il est cependant évident que ces deux traits ne s'accordent pas car les structures de données qu'ils manipulent sont différentes. Ils sont donc incompatibles alors que leurs rôles, eux, ne le sont pas.

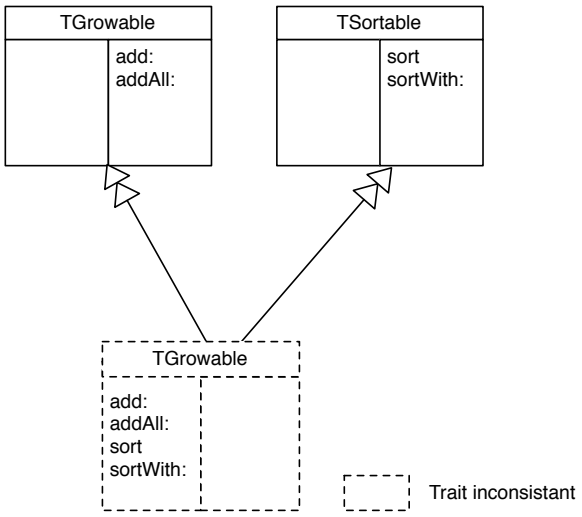

FIG. 4.2: La composition repose sur l'hypothèse que les implantations sont compatibles. Ainsi, le trait TGrowable utilisant un structure de tableau et le trait TSortable, utilisant une structure de liste chaînée ne peuvent produire une composition consistante.

Il est donc clair que dans de grandes hiérachies où les éléments reposent sur des implantations différentes, un trait par caractéristique n'est pas suffisant.

Nous proposons donc un ensemble de procédés permettant de faire face à ce problème d'hétérogénéité en nous attachant à la consistance polymorphique et comportementale.

Pollution d'interfaces. Au cours de l'analyse des classes des collections, nous avons fait le constat d'une pollution d'interfaces de ces classes. En effet, les opérations de conversions sont en bonne partie responsable du bruit introduit.

Nous proposons donc un moyen d'éviter cette pollution en réutilisant le concept de protocoles afin d'introduire des méthodes génériques.

Affinage des protocoles. La hiérarchie de protocoles des collections de Smalltalk-ANSI, bien qu'utilisée actuellement dans la plupart des implantations de Smalltalk, ne permet de fournir toutes les classes de collections dont nous aimerions bénéficier.

Ceci est principalement dû au fait que la granularité des protocoles n'est pas assez fine.

Avec les propositions que nous avons fait précédemment, nous introduisons donc un exemple d'une nouvelle hiérarchie maintenant réalisable qu'il sera intéressant de prendre comme référence pour une étude de cas complet.

4.2 Unifier les vues structurelles

William Cook, dans sa publication « Interfaces and Specifications for the Smalltalk-80 Collection Classes » (cf. section 2.2.1) regrette le fait de ne pas avoir accès à l'héritage multiple pour pouvoir exprimer la hiérarchie de protocoles des collections plus fidèlement :

> ... *« to express this structure more directly, Smalltalk would need multiple inheritance or mixins. »* – [Coo92], Section 6 : Interfaces Versus Inheritance.

Bien que la plupart des langages actuels utilisent l'héritage simple (pour les raisons évoquées en section 2.1.1), nous disposons maintenant des traits qui nous permettent d'exprimer des compositions venant de plusieurs éléments ; ceci en évitant la plupart des problèmes de l'héritage multiple.

Il est donc maintenant possible de combler le fossé entre protocoles et hiérarchie de classes grâce aux outils dont nous disposons.

Nous nous basons donc sur la hiérarchie de protocoles originale de Smalltalk-ANSI [ANS98], qui exprime des compositions n-aires de protocoles, comme point de départ pour une nouvelle hiérarchie de traits.

Chaque protocole devient donc un trait (par exemple, « Updatable Collection » deviendra *TUpdatable*) et il est donc possible d'obtenir une vue unique.

Ceci a de multiples avantages ; en voici trois significatifs :

1. Nous pouvons désormais implanter les protocoles de manière fidèle et nous y conformer ;
2. Nous évitons les problèmes exprimés précédemment comme la duplication de code ou l'annulation de méthodes ;
3. Le développeur peut obtenir une compréhension du système par simple lecture de la spécification, sans devoir s'adapter aux compromis d'implantation.

4.3 Faire face à l'hétérogénéité

4.3.1 Vers des implantations spécifiques

Le fait de devoir manipuler le même concept sous différentes formes pour une même caractéristique nous amène alors à déduire qu'il est nécessaire de disposer de plusieurs implantations pour chaque caractéristique.

Ces implantations différentes nécessitent donc d'être spécifiques à chaque structure de données sous-jacente. Si nous avons alors une implantation pour chaque caractéristique, nous pourrons donc aisément composer les différents traits compatibles afin de créer l'ensemble des classes de collections.

Bien que ceci semble idéal, l'introduction de plusieurs implantations pour une même caractéristique soulève aussi plusieurs questions. En voici une liste non exhaustive :

- Comment assurer que ces implantations seront conformes au même protocole ?
- Comment assurer que ces implantations offriront une même sémantique, bien que manipulant des structures de données différentes ?
- Comment permettre aux développeurs souhaitant étendre ces hiérarchies de facilement connaître les traits compatibles ?
- Comment proposer une vue facile à appréhender à l'utilisateur des collections. En effet, y a t-il un intérêt à ce que ce denier connaisse l'existence des traits spécifiques aux structures de données ?

4.3.2 La nécessité des protocoles

Permettre d'avoir plusieurs implantations partageant un même rôle dans un système induit qu'il est nécessaire que ces premières respectent le même ensemble de messages. En effet, ce n'est pas parce qu'une implantation va utiliser telle ou telle structure de données qu'elle doit réagir à des messages différents.

Si les implantations étaient différentes, il ne serait pas possible de demander un tri sur un tableau triable de la même manière qu'on le ferait sur une liste chaînée triable. Or, lorsque nous souhaitons réaliser cette opération de tri, nous n'avons pas envie de nous occuper de l'implantation sous-jacente.

La suite de cette section décrit les bénéfices de cette abstraction.

Assurer un polymorphisme. En premier lieu, connaître le message correspondant à une opération donnée permet de créer des méthodes génériques et donc d'assurer un bon polymorphisme entre les différentes classes de collections.

Ainsi, nous pouvons écrire l'expression suivante sans nous soucier de la nature sous-jacente de aSortableCollection :

```
aSortableCollection sort
```

Nous ne nous devons de connaître qu'une seule information : la collection qui est traitée implante effectivement le protocole associé à la notion de tri.

Améliorer la compréhension. En deuxième lieu, définir un protocole unifié pour toutes les classes remplissant le même rôle rend le système consistant. En effet, comme nous allons devoir implanter une classe pour chaque type de structure de données, le nombre de classes présentes dans le système peut vite croître comme le montre le calcul suivante :

Nombre Classes = Nombre de structures de données $*$ Nombre de caractéristiques

Définir un protocole donne donc de bonnes propriétés d'utilisation : une courbe d'apprentissage plus douce pour l'utilisateur de la bibliothèque ainsi qu'une maintenance plus aisée.

Se préparer aux nouvelles collections. Troisième point non négligeable, connaître à l'avance les messages qui doivent être supportés permet de créer des extensions plus efficacement et surtout plus fiablement.

Dans le cas où nous n'avons pas de consensus défini sur un protocole, si nous devons ajouter une implantation de la caractéristique « Growable » pour les structures de type "Arbre", comment allons nous procéder ?

Avec plusieurs implantations déjà existantes dans le système, la seule solution est d'aller inspecter ces classes pour en déduire les messages qu'il faut implanter.

Ceci suppose deux choses de la classe que nous prenons comme exemple :

1. Qu'elle ait effectivement bien implanté tous les messages nécessaires ;
2. Qu'elle n'ait pas ajouté des méthodes propres à son utilisation, auquel cas nous pourrions en venir à considérer trop de messages.

Avec un protocole bien défini, nous pouvons donc combler le manque de spécification. En effet, pour connaître les messages à implanter afin de développer notre trait *TGrowable* dédié aux arbres, il nous suffit alors de nous référer au protocole « Growable ».

4.3.3 Matérialisation des protocoles

Comme nous l'avons soutenu, les protocoles sont réellement nécessaires pour faire cohabiter plusieurs implantations partageant un même rôle.

Ceci combiné avec le fait que nous avons unifié la hiérarchie des protocoles avec celle d'implantation (grâce aux traits), il nous est alors possible de donner corps aux protocoles.

Nous les matérialisons donc à l'aide de traits définissant pour chaque message du protocole associé, une méthode portant le nom de ce message.

Nous appelons ces traits des « traits-protocole » et nous décidons par convention de préfixer leurs noms par « TP ». La figure 4.3 présente un exemple de trait-protocole.

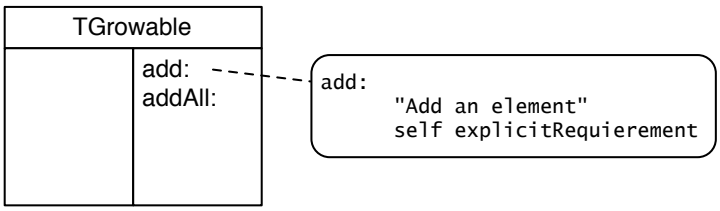

Fig. 4.3: Un exemple de trait-protocole pour le protocole Growable.

Détails d'un trait-protocole. Un trait-protocole, dans sa forme la plus simple, est donc composé de deux éléments : des méthodes et de la documentation.

Tout d'abord, les méthodes sont nommées suivants les noms des messages du protocole. Ces méthodes ne définissent pas de comportement lié au message qu'elles représentent. Elles ont cependant la contrainte de devoir lever une exception de type « Nécessité d'implantation » (*explicitRequirement*).

Lever cette exception permet de faciliter la détection d'erreurs d'implantation. En effet, dans le cas où un trait souhaite implanter un trait-protocole et qu'il oublie de redéfinir une méthode, une exception sera levée, indiquant clairement qu'un oubli d'implantation a été commis. Ce comportement est essentiel afin de ne pas provoquer des exécutions silencieuses, mais faussées. Nous pourrons alors traquer facilement l'erreur afin de la corriger.

Ensuite, la classe, ainsi que chacune de ses méthodes, dispose d'une documentation définissant respectivement l'objet du protocole et la fonctionnalité réalisée par la méthode. Posséder la documentation à ce niveau est particulièrement intéressant car elle sera ainsi automatiquement propagée (via la composition) aux implanteurs du trait-protocole. Ceci réduit donc le nombre de classes non documentées.

Remplacement des conventions. Lors de la première refactorisation des Collections avec les traits, une convention de nommage été proposée (cf. section 2.2.2) afin de classifier les traits (par exemple, S pour Séquenciel, ...).

Bien qu'il s'agisse en effet d'une méthode fonctionnelle, elle n'est pas idéale car elle implique que le programmeur connaisse et respecte cette convention.

Avec la méthode que nous proposons, nous n'utilisons que les mécanismes du langage pour caractériser la nature de nos traits, ce qui ne demande pas une connaissance préalable des symboles.

Extensions particulières. Bien entendu, une implantation particulière est libre d'étendre la liste des messages compréhensibles pour un protocole. Il est en effet probable que des méthodes à utilisation principalement interne soient définies lors de l'établissement de traits d'implantation.

Cependant, nous n'encourageons pas cette pratique dans le sens où l'utilisateur pourrait supposer que ces messages font partis du protocole, ce qui dégraderait les possibilités polymorphiques.

4.4 Améliorer la consistance par les contrats

Le fait de donner corps aux protocoles nous permet donc de rendre la diversité d'implantations consistante. Nous pouvons donc maintenant affirmer qu'un bon polymorphisme est garanti. Cependant, même si les méthodes sont effectivement uniformisées, nous n'avons pas de garanties au niveau comportementale.

Lorsqu'un trait va implanter un protocole, celui-ci va composer le trait-protocole et donc redéfinir les méthodes attendues. La classe qui utilisera ce trait pourra

donc répondre aux messages attendus, mais rien n'empêche le développeur d'implanter incorrectement ces deniers.

En effet, même si nous souhaitons que ces méthodes soient documentées, le langage naturel n'est pas toujours le mieux adapté pour décrire un comportement. Ce dernier n'est en rien formel, il est donc soumis à interprétation et un développeur n'ayant pas une parfaite maîtrise de la langue dans laquelle la documentation a été rédigée peut ne pas en saisir toute la subtilité.

Ainsi, nous souhaitons enrichir les traits-protocoles avec des contrats afin de i/ améliorer la consistance comportementale de la bibliothèque et ii/ rendre la tâche de développement plus balisée.

4.4.1 Définition des contrats

Afin de mettre en place ce mécanisme, deux types de contrats doivent être mis en place pour assurer une bonne spécification.

Comportement des méthodes. Tout d'abord, nous devons disposer de contrats au niveau des messages du protocole, afin de spécifier le comportement même de ceux-ci. Lorsqu'un trait va implanter les messages du trait-protocole en question, ce premier va donc redéfinir les méthodes.

Ce que nous voulons, c'est que lorsqu'un trait implante un trait-protocol, il soit soumis aux exigences du protocole. Les contrats définis pour les méthodes du trait-protocole seront donc appliqués aux méthodes redéfinies du trait implanteur.

Invariants de traits et classes. Ensuite, appliquer un invariant de classe pour les traits se révèle être d'une grande aide pour la consistance et le bon assemblage des traits.

En effet, à travers des invariants, il est possible d'exprimer des propriétés de compositions intéressantes. Dans les sections précédentes, nous avons présenté un moyen de régler les problèmes de compositions de traits ayant des implantations sous-jacentes différentes, mais il est toujours possible de composer des traits non compatibles.

Supposons que nous voulons créer une collection qui se trie automatiquement. Nous allons alors utiliser le protocole « Sorting », qui permet le tri automatique.

Ce trait disposera d'un invariant qui vérifie que les éléments sont bien triés à tout instant de l'existence de la collection.

En voulant bien faire, nous aimerions aussi ajouter le tri explicite. Le protocole « Sortable » sera alors lui aussi utilisé. Cependant, ce dernier permet d'utiliser un critère de comparaison personnalisé pour trier les éléments.

On se rend donc compte que ces deux traits sont en fait incompatibles car ils entrent en conflit en agissant sur les mêmes propriétés de la collection. L'état « trié » assuré par le protocole « Sorting » ne sera donc plus maintenant, car « Sortable » fait qu'il est possible d'altérer cette propriété de tri.

Ceci peut être révélé très rapidement (à la première utilisation de notre classe) grâce à la prise en compte de tous les contrats des traits composés dans la classe. Cette prise en compte sera réalisée pendant l'étape de composition des traits.

Il nous suffira donc d'ajouter au composeur tous les invariants des traits, comme l'illustre la figure 4.4.

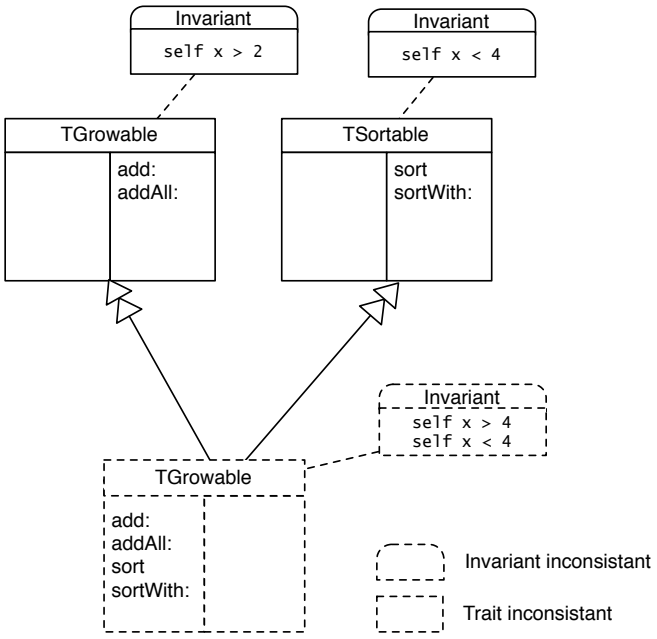

FIG. 4.4: La probabilité de composer des traits incompatibles est reduite à l'aide des invariants.

Fusionner les contrats lors de la composition des traits permet donc d'assurer que des traits incompatibles ne sont pas composés entre eux.

4.4.2 Supports d'exécution

Maintenant que nous avons décrit comment enrichir les traits-protocoles avec des contrats pour améliorer la consistance du système, il est intéressant d'étudier comment cette solution s'intègre aux outils existants.

Smalltalk

Smalltalk, au moins dans sa version Squeak, ne propose pas de support des contrats.

Seule une publication « Design by Contract in Smalltalk » (DbCS) [CMPR96] propose une extension de l'environnement Smalltalk pour y inclure les contrats. Nous présentons les éléments essentiels de cette méthode.

Invariants. En ce qui concerne les invariants de classes, le modèle DbCS propose d'ajouter un champ « classInvariant » lors de la définition d'une classe. Ce champ contient une chaîne de caractères représentant un bloc exprimant les conditions de l'invariant, comme le présente l'exemple suivant :

```
Object subclass: #BankAccount
   instanceVariableNames: 'balance owner'
   ...
   classInvariant: '[(balance > 0) and:
                     [owner nationality = 'spanish']]'
```

Cet invariant sera injecté au début et à la fin de chaque méthode.

Pre et post-conditions. Pour la gestion des conditions de pre et post-conditions, cette méthode implique de faire des appels aux méthodes « require » et « ensure » comme suit :

```
withdrawal:anAmount personalCode:code
   "Withdraws money from the account if the code is right"
   self require:
      [balance>=0].
   self ensure:
      [(personalCode <> code) & self noChange) or:
       [(self old) balance = (balance + anAmount)]].
```

L'appel à « require » sera appelé au début de la méthode, tandis que « ensure » sera inséré avant chaque retour de méthode.

La méthode finale aura donc cette structure :

```
methodName: args
   "comment"
   |temporaries|
   checkInvClass
   self require: ...
   < statements >
   self ensure: ...
   checkInvClass
   ^ return_value
```

Nous disposons donc d'un moyen complet, efficace et dans l'esprit de Smalltalk (*« every action in the system is a consequence of sending a message to an object »*) capable d'exprimer des invariants de classes et des conditions de méthodes.

Les traits

Étant donné que les traits ne gèrent pas les contrats, nous proposons une extension à ceux-ci afin de supporter cette fonctionnalité. Cette extension se base sur le modèle des traits sans état et nous nous appuyons sur le modèle d'implantation des contrats cité dans le paragraphe précédent.

Invariants. Tout d'abord, il est nécessaire de s'intéresser aux invariants de classes/traits. Lorsqu'une classe A hérite d'une classe B, elle hérite de l'invariant

de la classe A et peut soit garder le même invariant ou le raffiner (mais jamais l'étendre).

Bien que la composition de trait ne soit pas la même opération qu'un héritage, nous pouvons voir que la composition de trait doit respecter la même contrainte : c'est à dire garder ou renforcer le contrat mais ne jamais l'affaiblir.

Lorsqu'un trait ou une classe compose des traits, nous allons donc aussi composer les invariants des traits utilisés. Selon la propriété que nous venons d'énoncer, nous les composerons donc à l'aide de l'opérateur logique « AND ». Nous obtenons alors la formule suivante :

$$invariant_{final} = (invariant_{trait1} \wedge ... \wedge invariant_{traitn}) \wedge invariant_{composeur}$$

Par exemple, nous avons sur la figure 4.5 la classe *C* qui vient composer *T1* et *T2*. *T1* a comme invariant "[self total > 2]" et *T2* "[(self total > 10) and : [self total < 100]]".

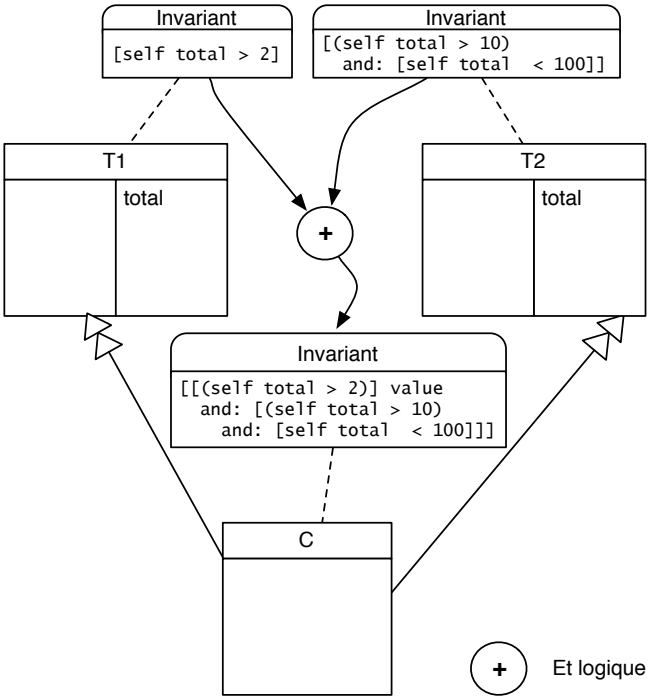

FIG. 4.5: Les invariants lors de la composition de traits sont fusionnés à l'aide de l'opérateur « AND ».

Quand nous composerons ces deux traits nous obtiendrons donc comme invariant :

```
[[self total > 2] value and: [(self total > 10)
                              and: [self total < 100]]]
```

ce qui pourrait se réduire en :

```
[(self total > 10) and: [self total < 100]]
```

Pre et post-conditions. Ensuite, nous devons traiter le cas des pre et post-conditions. On peut distinguer deux cas de composition de méthodes avec les traits : i/ l'utilisation d'une méthode de trait dans une classe ou ii/ la redéfinition d'une méthode d'un trait dans l'entité composante.

Sans redéfinition. Dans le premier cas (figure 4.6), notre composition va faire pointer les méthodes directement vers les méthodes des traits d'origine. Nous n'avons donc rien à faire car les conditions seront les mêmes que lorsque la méthode était définie dans le trait.

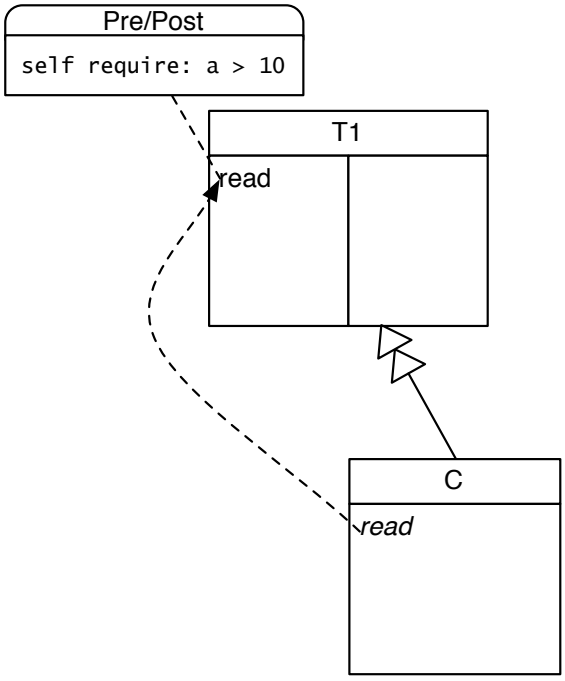

Fig. 4.6: Composition d'un trait sans redéfinition : aucun mécanisme n'est à rajouter.

Avec redéfinition. Le second cas, où de la redéfinition est introduite, est plus problématique. Nous l'étudions en deux sous-cas, le premier étant plus simple.

Le premier se destine à caractériser le comportement à adopter lorsque nous allons redéfinir une méthode venant d'un seul trait. En effet, dans la spécification des traits, il est indiqué qu'une méthode définie dans l'élément compositeur prend

précédence sur celle provenant du trait. Nous ne remettons bien entendu pas en cause cette propriété, mais nous proposons cependant de dupliquer le contrat de la méthode redéfinie et de l'intégrer à la méthode de l'élément compositeur. Ceci est illustré sur la figure 4.7.

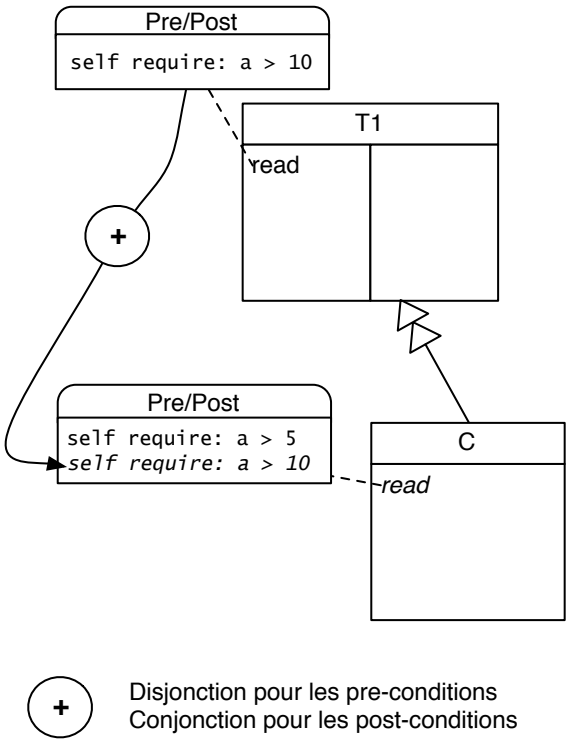

FIG. 4.7: Lors qu'un trait est utilisé et qu'une méthode est redéfinie, son contrat est ajouté à celui de la nouvelle implantation.

Nous faisons ce choix car les méthodes du trait qui utilisent cette méthode redéfinie n'ont pas conscience qu'il s'agit d'une méthode différente. Le comportement qu'elles attendent est donc bien celui de la méthode interne au trait. C'est pourquoi, nous sommes convaincus qu'il est nécessaire d'imposer, dans ce cas, le contrat à la méthode qui redéfinit.

Le deuxième cas s'intéresse à la situation où nous redéfinissons une méthode venant de plusieurs traits. Il s'agit d'une version un peu plus complexe que le cas précédent, en effet, comment devons nous gérer les contrats des différents traits ?

Il est très certainement probable que même si les méthodes portent le même nom dans les différents traits, leur sémantique soit différente car ces traits n'ont pas forcément été conçus pour travailler ensemble.

Cependant, la remarque faite dans le premier cas reste tout à fait valable : les méthodes attendent le même comportement que celui d'origine. C'est pourquoi, nous avançons que les contrats de tous les traits doivent être pris en compte

afin d'assurer le bon fonctionnement des méthodes. Ceci implique la possibilité que le contrat résultant soit insatisfiable. Si la composition implique ce cas de figure, ceci indiquera simplement que les traits ne peuvent fonctionner ensemble. Si nous avions décidé de ne pas agir ainsi, la seule chose résultante aurait été une composition non consistante. Les deux cas de figure sont illustrés sur la figure 4.8.

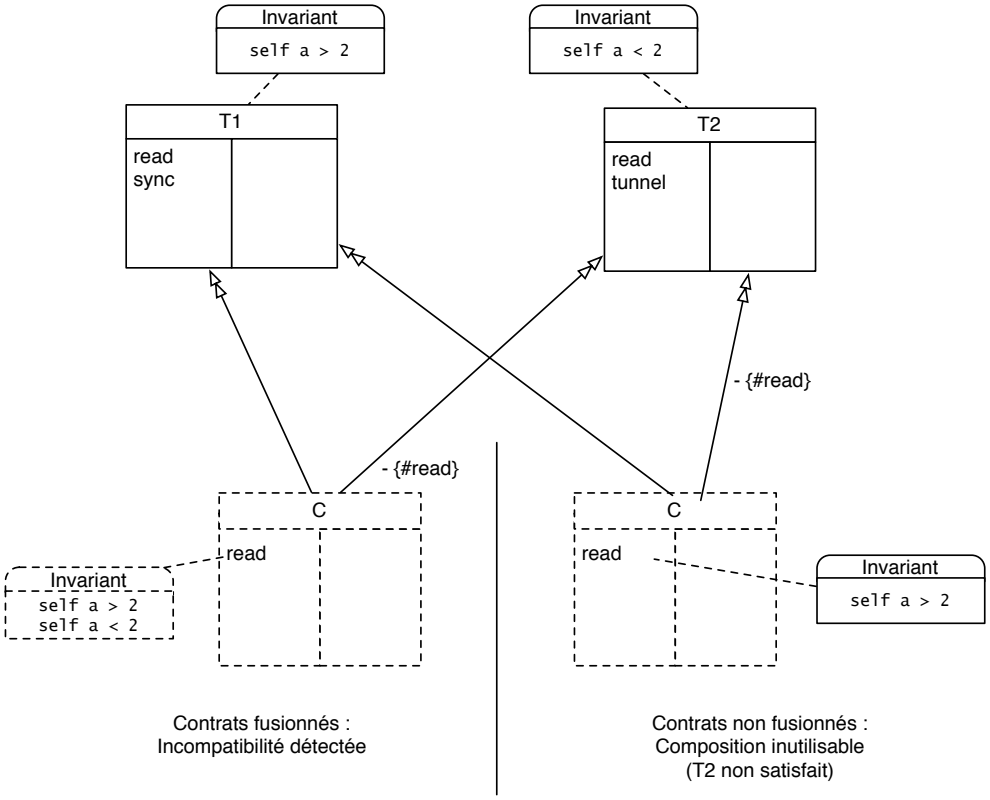

Fig. 4.8: Deux cas de figures lors de la composition et redéfinition de méthodes venant de plusieurs traits. Si nous prenons en compte les contrats des méthodes redéfinies, nous assurons la détection des inconsistances.

Il est intéressant de noter vis à vis de ce problème, que le modèle des traits gelables (cf section 2.1.3) permet de palier à ce problème. En effet, s'il est possible de rendre une méthode précocement liée aux autres méthodes du trait, la question de redéfinition peut être écartée.

Un dernier élément important est de considérer l'opérateur d'exclusion des traits (le « - »). Celui permet de ne pas considérer une méthode d'un trait lors de la composition. Nous considérons que la situation résultante est la même que celle du deuxième sous-cas de figure. Il est donc aussi nécessaire de fusionner le contrat avec l'implantation qui va prendre précédence.

Enfin, deux tous les cas, il aussi nécessaire de pouvoir affiner les pre et post conditions au niveau de la nouvelle implantation. Suivant les règles de la programmation par contrat, nous n'autorisons que l'affaiblissement de la pré-condition et le renforcement de la post-condition.

Nous résumons donc le comportement à adopter vis à vis des pre et post-conditions lors de la composition des traits avec les formules suivantes :

$$precond_{finale} = (precond_{trait1} \vee ... \vee precond_{traitn}) \vee precond_{composeur}$$
$$postcond_{finale} = (postcond_{trait1} \wedge ... \wedge postcond_{traitn}) \wedge postcond_{composeur}$$

4.5 Garder un système ouvert

Dans la hiérarchie des collections, il est fréquent de constater que les classes proposent de convertir leur structure de données vers d'autres. Par exemple, nous pouvons avoir envie de transformer une liste chaînée vers un tableau. Il est aussi intéressant de noter que des classes comme *String* proposent des adaptateurs comme #asPostscript ou encore #utf8toIso.

Ces méthodes, bien que toutes remplissant le rôle de convertisseurs, sont nombreuses et souvent nommées sans réelle convention.

Ceci a plusieurs désavantages :

1. L'interface des classes est polluée dans le sens où nous avons de nombreuses méthodes de conversion alors que nous aimerions plutôt nous focaliser sur les méthodes métier ;
2. Le développeur qui souhaite ajouter un convertisseur doit ajouter une méthode, donc encore ajouter du bruit aux interfaces ;
3. L'utilisation n'est pas facilitée et il est pénible de devoir chercher une méthode parmi une liste importante.

Deux types de convertisseurs. Nous caractérisons les convertisseurs en deux catégories :

1. Les convertisseurs de conteneur : ceux-ci sont destinés aux conversions impliquant un changement de structure de données. On inlcuera par exemple #asArray ou encore #convertToAGivenStructure. Dans tous les cas, les données ne sont pas altérées : seule leur forme de stockage est changée ;
2. Les convertisseurs de contenu, qui permettent de modifier la représentation des données. C'est le cas par exemple d'un encodeur Utf-8 vers Iso, pour les *Strings* ou encore Ogg vers Wav pour des données audio.

Méthodes génériques. Nous allons alors introduire deux méthodes génériques aux collections permettant les deux types de conversions précédemment citées : #convertWith et #encodeWith pour respectivement convertir le conteneur et convertir le contenu.

Ainsi lorsque nous souhaiterons par exemple encoder un texte en UTF-8, nous pourrons effectuer ceci :

```
aString encodeWith: UTF8Encoder
```

Ceci retournera alors un string encodé en UTF-8.

Afin de donner les informations nécessaires aux objets qui doivent réaliser l'encodage, nous proposons alors d'utiliser un double dispatch [DDi].

Ainsi, si *aString* était encodé en « iso8859 », nous obtiendrons cette suite d'appels :

```
1. aString encodeWith: UTF8Encoder
2. UT8Encoder fromIso8859: aString
```

Nous aurons alors donc toutes les informations nécessaires pour réaliser la conversion : « un String encodé en Iso8859 doit être converti vers un String encodé en UTF-8 ».

Cependant, on peut se poser des questions quant aux objets qui peuvent être passés en arguments à ces méthodes de conversion.

Tout d'abord, comment être certain que la conversion est possible ? Ensuite, comment permettre l'écriture de nouveaux convertisseurs conformes ?

Protocoles utilisateurs. Nous répondons à la question en proposant de créer une hiérarchie de protocoles parallèle que nous appelons « Protocoles utilisateurs ». Cet ensemble de protocoles regroupe les actions qui peuvent être utilisées par la hiérarchie de protocoles de la bibliothèque. Un exemple de hiérarchie de protocoles utilisateurs peut être consulté sur la figure 4.9.

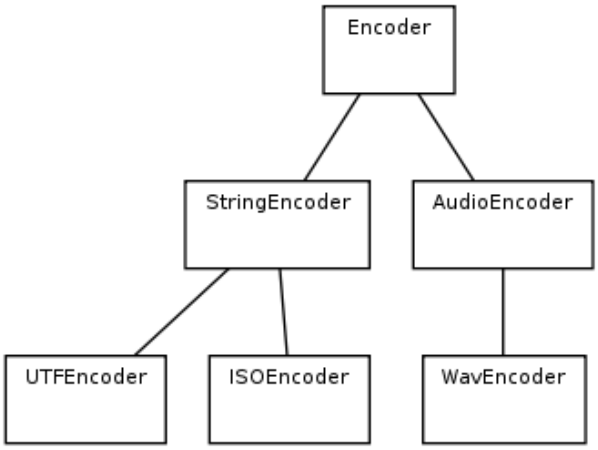

Fig. 4.9: Un exemple de hiérarchie de protocoles utilisateurs.

Ainsi, nous aurons aurons par exemple le protocole « StringEncoder » qui sera capable de répondre aux messages comme #fromIso8859.

Nous matérialisons aussi ce protocole via des traits-protocoles et adoptons le même processus qu'avec les hiérarchies de protocoles présentées dans les sections précédentes. Nous pouvons alors, par exemple, créer deux encodeurs respectant le protocole « StringEncoder » : *TUTF8Encoder* ou *TIso8859Encoder*.

La figure 4.10 donne un exemple d'utilisation de hiérarchie de protocoles utilisateurs par une classe la hiérarchie des protocoles principaux.

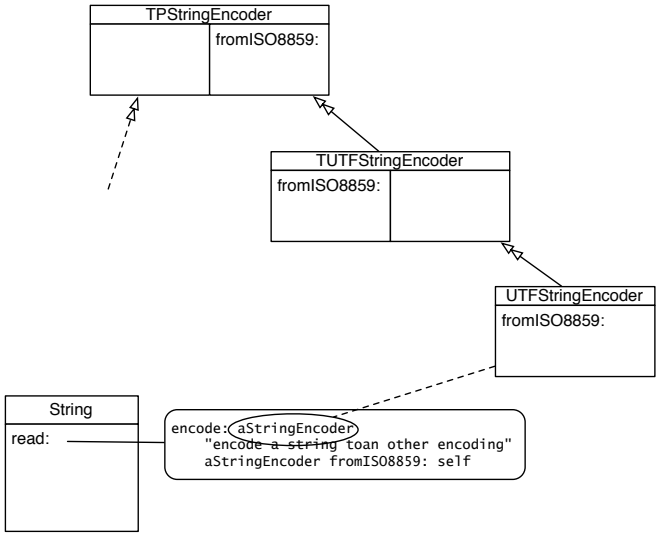

FIG. 4.10: Un exemple d'utilisation des protocoles utilisateurs.

Utilisation des convertisseurs. Afin de s'assurer que la conversion est possible, il nous suffira alors d'utiliser un trait qui implante le protocole requis.

Une question reste cependant en suspend : comment spécifier le protocole qui est requis ? Nous répondons à ceci en appliquant une convention de nommage, comme Smalltalk a l'habitude de le faire.

Ainsi, si nous avons un message d'un protocole qui prend en paramètre un convertisseur de type « WavEncoder », nous le déclarerons de la manière suivante :

```
Wav8File>>convert: aWavEncoder
 ^ aWavEncoder from8bit: self
```

Amélioration de l'utilisation. Bien que nous ayons maintenant les éléments nécessaires pour implanter et utiliser des protocoles utilisateurs, il est intéressant de se poser la question de simplicité d'utilisation d'un point de vue des développeurs.

Lorsqu'un développeur va vouloir utiliser le #convert, il va alors pouvoir regarder la signature et se rendre compte qu'il s'agit d'un *WavEncoder* qui est attendu. Cependant, comment connaître tous les encodeurs disponibles ?

Nous répondons à ceci en ajoutant une extension à l'IDE (Squeak dans notre cas) et permettant d'obtenir facilement une liste d'utilisateurs d'un trait donné.

Ainsi, il suffira au développeur de sélectionner « WavEncoder » et demander à afficher tous les implanteurs de ce protocole afin d'en choisir un dans la liste.

Une implantation de référence pour OBBrowser, l'outil de navigation de classes de Squeak, est disponible en annexe B.

4.6 Améliorer l'expressivité des protocoles

Maintenant que nous disposons d'outils suffisamment adaptés à la gestion des grandes hiérarchies de traits-protocoles, il est intéressant de se poser la question de la flexibilité des protocoles actuels. En effet, la hiérarchie de protocoles de Smalltalk-80, même révisée (cf. figure 2.12), permet difficilement d'exprimer tous les cas.

Comme la composition des traits ne se révèle pas coûteuse en terme de temps d'exécution [BSD03, CDW07], nous pouvons aisément proposer un ensemble de protocoles à plus fin grain, comme celui présenté sur la figure 4.11.

Cette nouvelle hiérarchie ne se veut pas exhaustive, mais représentative des possibilités. Ainsi, nous ne couvrons pas toutes les classes de collection mais exposons une hiérarchie capable d'exprimer la plupart des besoins courants.

Il est intéressant de noter que cette hiérarchie de protocoles permet de proposer des combinaisons au préalable impossible à exprimer. Ainsi, il est par exemple possible de créer un tableau immuable (*ImmutableArray*) en composant simplement *Indexable* et *Ordered*.

Ces protocoles peuvent bien entendu être révisés au niveau de leur granularité mais nous estimons que la proposition est un bon compromis entre modularité et complexité. La méthode employée pour obtenir ce résultat peut être consultée en section 5.1.1.

La suite de cette section donne une description succinte des protocoles.

Collection. Nous avons regroupé dans ce protocole les caractéristiques suivantes :

- Conversion : la capacité à être converti vers une autre forme de collection. Il a été supposé, pour cette version, que toutes les collections peuvent être transformées vers d'autres ;
- Sérializable : la capacité de se représenter sous une forme textuelle. Ceci permet entre autres la possibilité de transiter sur un réseau ou de réaliser de la persistance ;
- Taille : la capacité à garder des informations sur sa capacité et son nombre d'éléments ;
- Copie : la possibilité de copier la structure de données vers une nouvelle instance ;

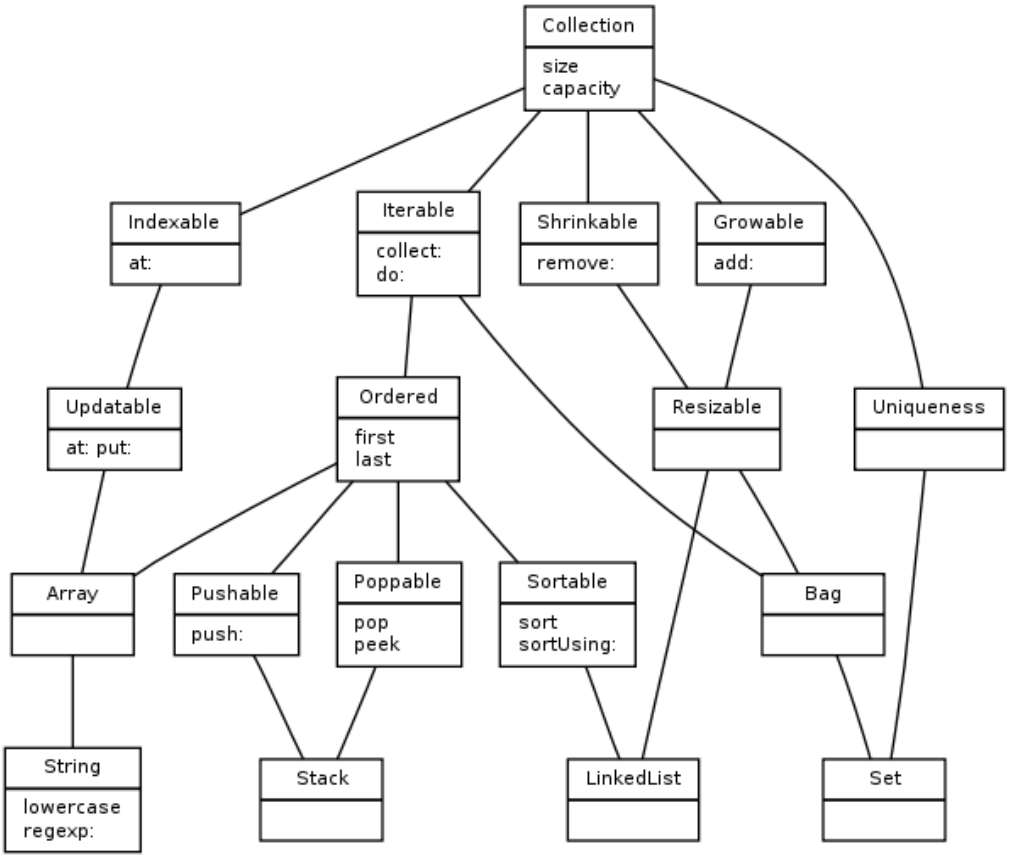

FIG. 4.11: Un exemple de hiérarchie de protocoles à plus fin grain. Les messages les plus essentiels sont détaillés.

Indexable. « Indexable » signifie qu'il est possible d'adresser les éléments selon une clé. Ainsi, un message comme #at : permet de récupérer l'élément à l'indice fourni. Ce protocole ne permet que d'effectuer des opérations de lecture sur la collection. Son pendant permettant l'écriture sur les collection se nomme « Updatable ».

Iterable. Le protocole « Iterable » indique que la collection peut être itérée. Il a été externalisé de « Collection » car nous pouvons imaginer qu'une collection puisse stocker des éléments en aveugle et ne puisse pas itérer sur toutes ses valeurs.

Ordered. « Ordered » indique que les éléments de la collection ont un ordre de stockage défini. Ce protocole ne doit pas être confondu avec le protocole de tri « Sortable ».

Uniqueness. Ce protocole implique qu'un élément ne peut figurer deux fois dans une même collection.

Shrinkable, Growable. Ces deux protocoles permettent respectivement de réduire et d'agrandir la capacité de la collection. On peut les utiliser pour créer des tableaux dynamiques ou encore des listes chaînées. Étant souvent utilisés conjointement, nous les avons regroupés dans une protocole appelé « Reziable » qui combine ces deux caractéristiques.

Poppable, Pushable. Ces caractéristiques expriment respectivement le fait qu'il est possible de retirer et d'ajouter un élément à l'extrémité d'une collection. On les utilise souvent pour réaliser des structures comme les piles (*Stacks*).

Quelques collections. Afin d'illustrer la composabilité des protocoles, nous avons intégré des protocoles terminaux (qui peuvent être utilisés directement par les utilisateurs de la bibliothèque) comme *Stack*, *Bag* ou encore *Array*.

Ces exemples ne sont pas exhaustifs et on pourrait facilement créer d'autres collections moins usuelles comme un *DynamicArray* en composant *Array* et *Resizable*.

5 Évaluation et méthodologie

Un des objectifs de nos travaux était d'évaluer les traits dans un contexte de grandes hiérarchies. Nous effectuons donc ici une revue de notre proposition en mettant en avant les aspects propres aux traits (granularité, type de traits utilisés, ...).

Nous continuons ensuite sur une comparaison avec les travaux connexes puis nous terminons par les perspectives d'évolutions de notre proposition.

5.1 Evaluation

5.1.1 Granularité des protocoles

Nous avons présenté un exemple de hiérarchie de protocoles en section 4.6. Nous estimons que cette hiérarchie est un bon équilibre entre expressivité et concision. Elle permet d'exprimer toutes les structures de données dont nous avons besoin, peut être raisonnablement étendue en combinant ses protocoles et permet une implantation rétro-compatible de la norme Smalltalk-ANSI.

Une des interrogations cruciales lors de l'utilisation des traits est la granularité idéale à adopter. En effet, bien que les traits permettent de décomposer les classes en de nombreux éléments sans impacter les performances, il est tout de même déconseillé de créer trop de traits car la hiérarchie devient alors très complexe à saisir.

Nous présentons ici la méthode qui nous a permis d'obtenir la hiérarchie présentée dans l'exemple. Nous estimons qu'elle peut être efficace pour atteindre un bon équilibre entre ces deux critères.

Celle-ci est en partie automatisée, mais nécessite tout de même que le développeur soit capable d'analyser la bibliothèque. Nous apportons cependant un outil qui permet d'exécuter une phase difficilement calculable par un Homme et qui permet d'élaborer la hiérarchie finale à partir de son résultat.

Cette méthode peut être scindée en trois parties majeures : i/ l'identification des caractéristiques des classes, ii/ la génération d'une hiérarchie minimale grâce à ces informations et iii/ le raffinement manuel de cette hiérarchie.

Identifier les caractéristiques. La première étape consiste à prendre en compte toutes les classes qui vont être implantées. Nous passons alors en revue chacune de ces classes en notant leurs caractéristiques comme « Extensible », « Iterable », etc. Lors de cette étape, nous conseillons au développeur de privilégier une forte décomposition des classes, quitte à générer trop de caractéristiques.

Nous obtenons à la fin de cette revue un ensemble de caractéristiques que nous pouvons alors associer aux classes.

Nous passons alors en revue ces classes une deuxième fois, associons à chacune d'elle les caractéristiques correspondantes et obtenons alors à la fin de cette étape un tableau comme celui proposé en figure 5.1.

Classe / Carac.	Array	String	Stack	Set	...
Iterable	√	√	√	√	...
Growable				√	...
Indexable	√	√	√		...
Poppable			√		...
...	...	...	...	...	...

Fig. 5.1: Un exemple de tableau d'association classes/caractéristiques.

Générer une hiérarchie minimale. Maintenant que nous disposons d'un ensemble d'associations, nous pouvons introduire le calcul de la hiérarchie de protocoles à l'aide de notre outil. Celui-ci prend en entrée un ensemble d'associations et produit la hiérarchie de protocoles la plus minimaliste possible. Chaque caractéristique sera alors représentée par un protocole.

Par minimaliste, nous entendons qu'il regroupe les caractéristiques utilisées conjointement en une seule, afin de diminuer le nombre de protocoles. L'algorithme va aussi trouver la hiérarchie logique pour ces caractéristiques, selon l'analyse d'utilisation de ces dernières par les classes.

Cet algorithme se déroule en quatre étapes :

1. Chercher les associations qui peuvent être assimilées comme identiques, c'est à dire celles qui sont utilisées exactement par les mêmes classes ;
2. Générer une liste de protocoles en se basant sur la liste d' associations. On obtient alors un dictionnaire de protocoles ayant comme valeur un ensemble vide ;
3. Ordonner les protocoles selon leur relation de couverture. Un protocole couvre un autre quand nous avons la relation suivante :
   Soit $\prec$ cette relation, p un protocole et un ensemble E_n définit par { Classe | Classe utilise n }
   $p \prec p'$ ssi $E_p \subset E_{p'}$
4. Pour chaque protocole, on supprime les liens directs si un lien indirecte existe. Par exemple, si « A pointe vers B », « A pointe vers C » et « B pointe vers C », on supprimera le lien « A pointe vers C ».

Nous obtenons alors une hiérarchie comme décrit sur la figure 5.2.

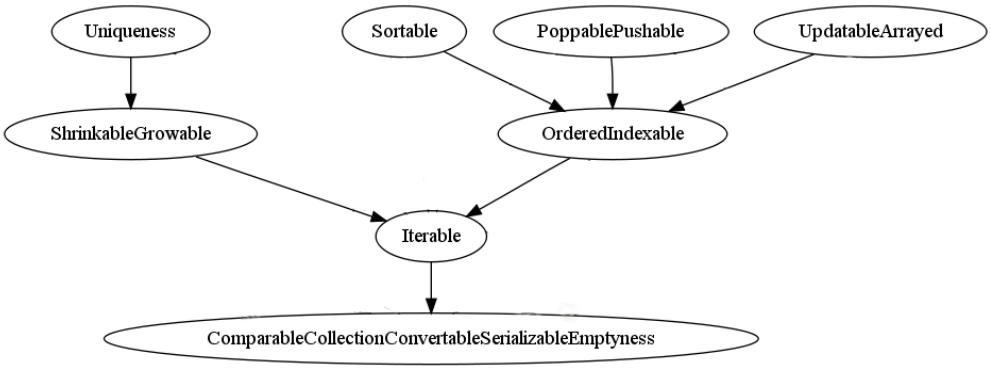

FIG. 5.2: La hiérarchie de protocoles générée à l'aide de l'outil. La flèche symbolise la couverture ($\prec$).

Analyser et améliorer. Nous avons obtenu une hiérarchie minimaliste extraite des associations que nous avons entrées. Cette hiérarchie, bien que fonctionnelle, n'est cependant pas satisfaisante pour obtenir un système ouvert et destiné à recevoir de nouvelles classes.

Nous pouvons voir par exemple que l'algorithme a regroupé les éléments « Growable » et « Shrinkable » en un seul protocole.

C'est à ce moment que l'analyse du concepteur est importante. Nous devons porter un oeil critique sur la hiérarchie de protocoles minimaliste afin de décider si effectivement nous pensons que ces protocoles regroupés ne seront jamais utilisés de manière disjointe.

Dans le cas de « ShrinkableGrowable », il est clair que nous souhaitons diviser ces deux préoccupations. Cependant, l'algorithme nous ayant fait remarqué que ces deux protocoles sont souvent utilisés conjointement, il est pertinent de proposer un sous-protocole regroupant ces deux caractéristiques. Nous avons appelé ce protocole « Resizable » sur l'exemple de hiérarchie (cf. section 4.6).

C'est pourquoi, nous insistons sur le fait que l'utilisation d'outils ne peuvent pas, dans l'état actuel, remplacer l'expérience du concepteur. Cependant, les outils restent cruciaux dans le sens où ils permettent de guider le concepteur dans sa démarche ainsi que de mettre en avant des propriétés qui ne sont pas aisément remarquables à cause d'un grand nombre d'informations.

5.1.2 Factorisation de code

Nous avons présenté le concept de traits-protocoles permettant de présenter une hiérarchie de protocole sans se soucier de l'hétérogénéité des implantations sous-jacentes.

Dans notre proposition, nous indiquons que les méthodes des traits-protocoles doivent nécessairement lever une exception afin de forcer l'implanteur à redéfinir cette méthode.

Un des avantages des traits est qu'ils permettent aussi de contenir du code. C'est pourquoi, même si nous ne l'avons pas mis en avant, nous sommes convaincus qu'il est possible de trouver des cas où une factorisation de code est possible au niveau de ces traits-protocoles.

Ceci permettrait ainsi d'éviter la duplication de code dans les traits d'implantation. Cependant, cette opération nécessite une attention particulière car il n'est pas évident de s'assurer que le code fourni sera compatible avec toutes les implantations. Il serait néanmoins possible d'introduire des comportements qui couvrent la plupart des cas et de les redéfinir dans les cas particuliers.

Ce travail nécessite donc à la fois une réflexion a priori, afin d'en tirer de bonnes abstractions et très certainement une amélioration a posteriori via des retours d'expériences.

5.1.3 Comparaison avec la refactorisation précédente

Notre solution diffère sensiblement du modèle obtenu lors de la première refactorisation. En effet, celle-ci est partie du principe qu'il fallait modulariser les classes existantes et a proposé un modèle constitué de traits permettant d'exprimer la hiérarchie en place.

Notre démarche considère que la hiérarchie actuelle n'est pas une finalité. En effet, même si notre méthode propose d'extraire les caractéristiques existantes, nous mettons l'accent sur l'ouverture du système en le préparant à de futures extensions. Nous pouvons ainsi avoir une bien meilleure expressivité des combinaisons de caractéristiques et donner la possibilité de créer des classes préalablement impossible à composer.

Notre proposition se démarque aussi en insistant sur la consistance du système. En effet, là où une convention de nommage servait à spécifier la nature des traits, nous lui préférons le concept de traits-protocoles. De plus, nous lui ajoutons des contrats, ce qui permet de garantir une certaine garantie vis à vis des comportements implantés.

5.1.4 Modèles de traits

Une des interrogations liée aux traits est d'essayer de déterminer quel modèle est le plus intéressant en pratique. Les refactorisations qui ont été faites jusqu'à présent ont utilisé le modèle sans état.

Dans notre cas, le modèle des traits sans état s'adapte parfaitement car nous ne référençons jamais réellement d'état. Cependant, nous avons tout de même remarqué que le modèle des traits gelables pourrait résoudre le problème lié aux contrats (section 4.4.2).

Nous pensons, vis à vis de l'expérience acquise à travers cet exercice de refactorisation, que le seul réel problème des traits sans état est la violation de l'encapsulation (et donc la pollution d'interface) à travers l'exposition d'accesseurs.

Les autres modèles de traits, bien qu'apportant d'autres fonctionnalités intéressantes, induisent une complexité qui ne justifie pas les avantages qu'ils apportent.

Notre conclusion est donc que l'idéal est certainement un modèle avec état qui conserve la simplicité du modèle sans état, mais qu'à défaut, les traits sans état restent le meilleur compromis.

5.2 Perspectives

Fondations pour une refactorisation. Tout au long de ce document, nous avons proposé des éléments cruciaux pour réaliser une refactorisation avec les traits sur des grandes hiérarchies.

Il serait donc maintenant intéressant de mettre en action ces concepts afin de les valider à travers une mise en pratique sur les collections.

Il s'agit d'un gros travail d'ingénierie qui pourrait être réalisé dans le cadre d'une évaluation poussée des traits afin d'obtenir un retour scientifique important sur ces derniers et d'éventuellement faire évoluer le modèle si le besoin se fait ressentir.

Patrons identifiables. Les outils que nous avons proposés pour refactoriser la hiérarchie des collections, bien qu'appliqués à cette dernière dans notre cas, ne sont très certainement pas limités aux collections.

En effet, nous pensons que les traits-protocoles peuvent être réutilisés dans toutes les hiérarchies assez importantes car ils procurent de bonnes propriétés pour régler les problèmes d'hétérogénéité.

Ainsi, il est très certainement possible de généraliser une grande partie des travaux proposés dans ce document et d'en tirer des bonnes pratiques pour les grandes refactorisations à l'aide des traits.

Traits dynamiques. Une fonctionnalité intéressante que nous aurions aimé disposer pendant la refactorisation est la possibilité d'ajouter ou supprimer des protocoles à la volée.

Ainsi, nous aurions pu ajouter et supprimer des fonctionnalités pendant l'existence de la collection. Par exemple, un cas intéressant est celui où nous souhaitons communiquer une collection à une classe tierce, mais nous aimerions nous assurer que cette classe ne fera pas de modification dessus. Nous pourrions alors lui fournir une version (ou vue) diminuée de cette collection en supprimant le protocole « Updatable ».

Cependant, bien qu'introduire ce genre de fonctionnalités permet d'accéder à de nombreuses possibilités, tout un ensemble de problèmes complexes viennent eux aussi s'ajouter. Nous pouvons citer comme exemple la garantie de consistance lorsque qu'un trait incompatible est ajouté. Le comportement à adopter n'est alors pas évident : le système pourrait tout aussi bien refuser le trait comme supprimer tous les traits incompatibles.

Nous pensons donc que cette voie peut être très enrichissante mais qu'elle nécessite un effort de conception important, dépassant même le cadre des traits et des refactorisations.

6 Conclusion

Notre travail a donc consisté à proposer un modèle utilisant les traits qui permet de refactoriser les collections en leur donnant de bonnes propriétés telles que la consistance, l'extensibilité et la modularité.

Nous avons commencé par proposer une vue hiérarchique à base de traits se calquant sur la hiérarchie de protocoles de Smalltalk-ANSI. Ceci a permis d'éliminer les multiples vues hiérarchiques afin d'éviter les duplications de code, les concessions de conceptions et d'améliorer la compréhension.

Nous avons ensuite introduit la notion de traits-protocoles permettant de faire face à la grande hétérogénéité des structures de données des collections. Grâce à ce concept, nous pouvons désormais implanter les protocoles sous plusieurs formes et grandement améliorer les combinaisons possibles.

Afin d'assurer une meilleure cohérence, nous avons aussi proposé une extension au modèle des traits sans état afin de permettre l'utilisation des contrats avec celui-ci. Utiliser les traits avec des contrats nous permet d'améliorer la consistance au sein de la hiérarchie des collections en i/ accélérant la détection des compositions inconsistantes et ii/ en s'assurant que les implantations spécifiques des traits-protocoles respectent le protocole associé.

Un exemple d'une nouvelle hiérarchie de protocoles a ensuite été proposée afin de prouver que, grâce à nos propositions, nous pouvons améliorer l'expressivité de la hiérarchie. Ceci nous a permis d'étendre les catégories de collections et de proposer des collections qui n'étaient pas réalisables auparavant avec la norme ANSI. Nous avons d'ailleurs introduit une méthode outillée permettant d'obtenir une granularité intéressante pour les hiérarchies de traits/protocoles.

En ce qui concerne les pollutions d'interfaces, nous avons brièvement présenté une solution permettant de les réduire. Nous nous sommes basés sur le fait que la plupart des messages polluants étaient de la même nature : les conversions.

A travers cette démarche, nous avons montré que les traits s'adaptent parfaitement aux grandes hiérarchies de classes et qu'ils permettent de doter ces dernières des propriétés idéalement recherchées (maintenabilité, extensibilité, ...).

Cette proposition reste donc maintenant à être appliquée et éprouvée à travers une refactorisation en profondeur des collections afin d'être améliorée et d'éventuellement en tirer des patrons de conceptions pour les traits.

Bibliographie

[ANS98] ANSI, New York. *American National Standard for Information Systems - Programming Languages - Smalltalk, ANSI/INCITS 319-1998*, 1998. http ://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf.

[BDN+07] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007. http ://SqueakByExample.org/.

[BDNW08] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34(2-3) :83–108, 2008.

[Ben] 100 Dollar Laptop. http ://en.wikipedia.org/wiki/$100_laptop.

[BSD03] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, October 2003.

[CDW07] Damien Cassou, Stéphane Ducasse, and Roel Wuyts. Redesigning with traits : the nile stream trait-based library. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 50–75. ACM Digital Library, 2007.

[CMPR96] M. Carrillo, J. Garcia Molina, E. Pimentel, and I. Repiso. Design by contract in smalltalk. *Journal of Object-Oriented Programming*, 9(7), 1996.

[Coo87] Steve Cook. OOPSLA '87 Panel P2 : Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, pages 35–40. ACM Press, October 1987.

[Coo92] William R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 27, pages 1–15. ACM Press, October 1992.

[DDi] The double dispatch mechanism. http ://en.wikipedia.org/wiki/Double_dispatch.

[DWBN07] Stéphane Ducasse, Roel Wuyts, Alexandre Bergel, and Oscar Nierstrasz. User-changeable visibility : Resolving unanticipated name

clashes in traits. In *Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 171–190, New York, NY, USA, October 2007. ACM Press.

[Kee89] Sonia E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 1989.

[LP90] Wilf LaLonde and John Pugh. *Inside Smalltalk : Volume 1*. Prentice Hall, 1990.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall, 1988.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits : Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Squ] Squeak home page. http ://www.squeak.org/.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., 1986.

A La syntaxe Smalltalk

Le tableau A.1 montre les éléments de base de la syntaxe Smalltalk.

.	le point sépare les instructions.
:=	le symbole **:=** sert à l'affectation.
^	L'accent circonflexe sert à retourner des valeurs à la méthode appelante.
'chaîne de caractères'	les simples quotes délimitent les chaînes de caractères.
[**bloc**]	les crochets délimitent les closures, appelées bloc en Smalltalk.
[**:arg1 :arg2 | bloc**]	les blocs peuvent prendre des arguments.
|tmpVar1 tmpVar2|	les pipes servent à déclarer des variables temporaires. Leur durée de vie et leur portée est limitée à la méthode.
"commentaire"	les commentaires sont entre guillemets.
$a	représente le caractère a, instance de la classe *Character*.
#symb	représente un symbole instance de *ByteSymbol*.
#(a b)	représente un tableau instance de la classe *Array* qui contient deux élément a et b.

Tab. A.1: Éléments de base de la syntaxe

En Smalltalk, il existe six pseudo variables. Elles sont données dans le table A.2.

true et **false**	ces constantes sont des instances des classes *True* et *False*.
nil	**nil** est la valeur par défaut de toute variable.
self	**self** représente l'objet courant. Il est équivalent au **this** de Java.
super	indique que le lookup doit se faire à partir de la super classe.
thisContext	**thisContext** représente la pile d'exécution en cours.

Tab. A.2: Les pseudo variables

Il n'y a pas de fonction en Smalltalk, seulement des méthodes que l'on exécute sur des objets. Il y a trois types de méthodes :

Les méthodes unaires. Les méthodes unaires n'ont pas d'argument.

```
"Appel de la methode removeAll sur l'objet uneCollection :"

uneCollection removeAll.
```

Les méthodes binaires. Les méthodes binaires prennent un argument et leurs noms est composé de symboles ('.' ',' '+' '-'...).

```
"Appel de la methode + sur l'objet 1 avec le parametre 3 :"

1 + 3
```

Les méthodes à mots-clés. Enfin, les méthodes à mots-clés possèdent un nombre illimité de paramètres. Leurs noms sont composés de plusieurs groupes de caractères (autant qu'il y a d'arguments) se terminant chacun par le caractère ':'. Les arguments se placent après chaque caractère ':'.

```
  "Appel de la methode replaceFrom:to:with: qui prend 3 arguments sur
  l'objet uneCollection :"

uneCollection replaceFrom: 1 to: 6 with: uneAutreCollection.

"L'equivalent java serait :"
uneCollection.replaceFromToWith(1, 6, uneAutreCollection);
```

Priorité des messages. Lors de la lecture d'une expression les messages sont évalués dans l'ordre suivant :

1. Méthodes unaires ;
2. Méthodes binaires ;
3. Méthodes à mots-clés.

Par exemple :

```
  5 factorial + 5 gcd: 5
```

Doit être lu de la manière suivante :

```
  ((5 factorial) + 5) gcd: 5
```

Il n'y a donc pas de priorité pour les opérateurs mathématiques. Ainsi **3 + 4 * 3** vaut 21 et non pas 15. Il faut donc écrire **3 + (4 * 3)** si l'on veut que la multiplication prenne précédence sur l'addition.

Les cascades. Il est possible et très fréquent en Smalltalk d'envoyer plusieurs messages au même objet. Cela se fait grâce à l'opérateur ';'.

```
uneCollection
       add: unObjet;
       add: unAutreObject;
       add: unTroisiemeObjet.
```

La définition d'une nouvelle méthode se fait de la façon suivante :

```
String>>lineCount

  "Compte le nombre de lignes dans le receveur (une chaine de
   caracteres). Chaque carriage return (cr) compte pour une nouvelle
   ligne."

   | count |
   count := 1.
   self do:
      [:c | (c == Character cr)
                  ifTrue: [count := count + 1]].
   ^ count
```

Le morceaux de code précédent doit s'interpréter de la façon suivante :

1. une nouvelle méthode appelée **lineCount** est définie dans la classe *String*.
2. un commentaire explique le but de cette méthode.
3. une variable temporaire **count** est déclarée puis initialisée à 1.
4. la méthode *do :* est exécutée sur l'objet en cours (**self**). Cette méthode prend un argument de type bloc, et le bloc doit avoir un argument. Le bloc sera évalué pour chaque caractère de la chaîne. La méthode *do :* est équivalent à une boucle **foreach** ou a une fonction **mapcar** en Common Lisp.
5. Le bloc définit un argument appelé **c**. Cet argument sera remplacé par chaque caractère de la chaîne, l'un après l'autre.
6. Le corps du bloc commence par comparer le caractère en cours avec le caractère de fin de ligne. Le caractère de fin de ligne s'obtient avec **'\n'** en C et en appelant la méthode *cr* sur la classe *Character* en Smalltalk.
7. Le résultat de la comparaison est un booléen : soit **true** soit **false**.
8. Sur ce booléen, la méthode *ifTrue :* est appelée. Cette méthode prend un bloc en paramètre et n'évalue ce bloc que si le receveur est **true**.
9. Si le booléen est **true**, le bloc est exécuté et on ajoute 1 à la variable **count**.
10. Quand la méthode *do :* a fini d'itérer sur tous les caractères, elle se termine puis la valeur de la variable **count** est retournée.

B Trait Users pour OBBrowser

Dans cet annexe, nous présentons une implantation d'un outil d'aide à la recherche des implanteurs d'un trait. Avec cet extension d'OBBrowser, il est donc maintenant possible de parcourir rapidement les implanteurs des protocoles. Ceci est particulièrement utile lors de l'utilisation des traits-protocoles.

Pour réaliser l'opération, il suffit de sélectionner un trait et d'indiquer « browse trait users » B.1.

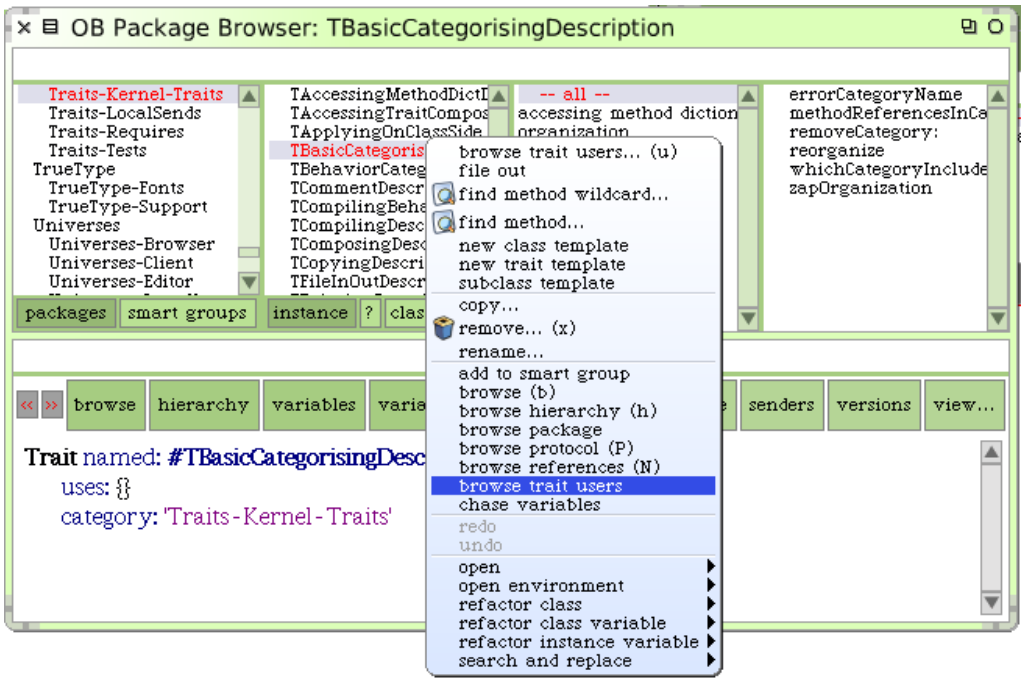

FIG. B.1: Sélection d'un trait avec OBBrowser pour afficher ses implanteurs.

Le résultat de cette commande sera consultable sous la forme d'une navigateur de traits, comme le montre la figure B.2.

Voici le code de référence permettant la collecte des implanteurs :

```
showUsersOf: aTrait
  ClassListBrowser browseClassesSatisfying:    [:class |
    class traitComposition traits includes: aTrait ]
                        title: 'Users of ', aTrait name
```

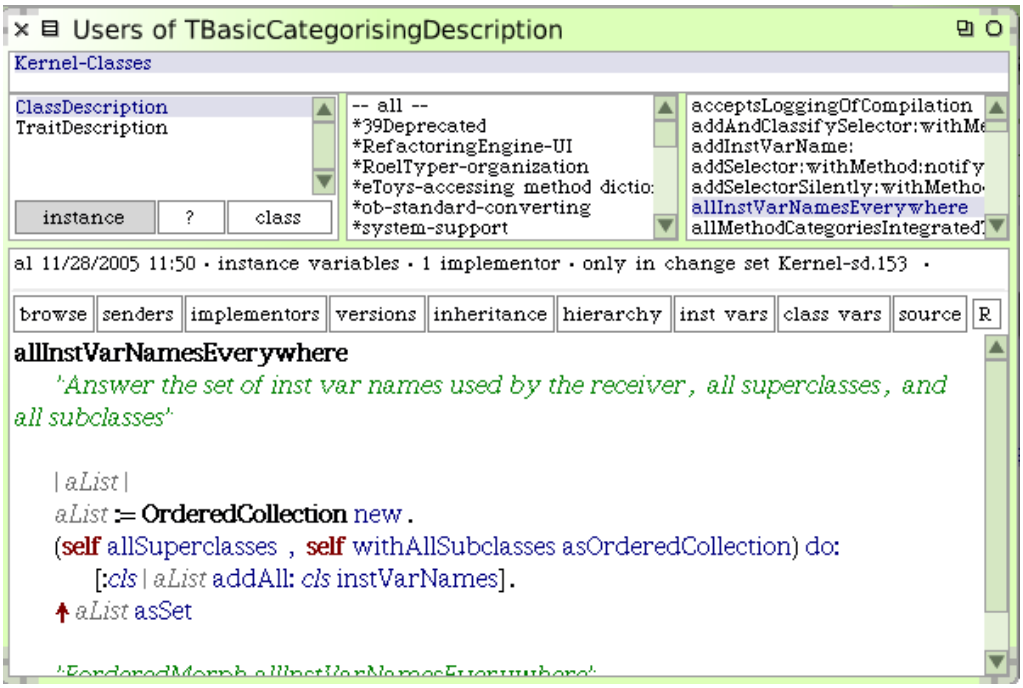

Fig. B.2: Parcours de implanteurs des traits.