FACULTÉ DES SCIENCES
DÉPARTEMENT D'INFORMATIQUE

UNIVERSITÉ LIBRE DE BRUXELLES,
UNIVERSITÉ D'EUROPE

ULB

Implémentation des traits à états en Java

Pierre ALEXIS

Mémoire présenté sous la direction des **Professeurs Raymond DEVILLERS et Roel WUYTS**
en vue de l'obtention du grade de Licencié en Informatique
Année académique 2006–2007

Remerciements

Je tiens à remercier vivement Roel Wuyts, promoteur de ce mémoire, à la fois pour son suivi soutenu, ses conseils, ses nombreuses relectures et son éternel optimisme.

Je tiens également à remercier Raymond Devillers sans qui ce mémoire n'aurait jamais pu voir le jour.

Je remercie également les membres de mon Jury pour le temps qu'ils passeront à lire ce travail : Raymond Devillers, Thierry Massart et Yves Roggeman.

Mes remerciements vont également à Yann-Gaël Guéhéneuc pour ses conseils.

Je tiens également à exprimer toute ma gratitude envers mes parents pour leurs nombreuses relectures de ce document et pour leur soutien durant toutes mes études.

Aboutissement de mes études, ce mémoire est l'occasion de remercier également toutes celles et ceux qui m'ont accompagné durant ces années en offrant soutien et amitié, que ce soit au niveau des études où dans mes tâches de délégué étudiant. Alessandro, Andy, Anne, Jérémy, Julien, Laurent, Markus, Renaud, Sébastien, Yannick, soyez remerciés. Enfin, je tiens à adresser des remerciements tout particuliers à Xavier pour son amitié et son soutien.

Table des matières

Table des figures

Table des listings

Chapitre 1

Introduction

Comment concevoir rapidement des logiciels de qualité ? Telle est la préoccupation de nombreux informaticiens. Répondre à cette question n'est pas évident. De nos jours, la conception de logiciels est devenue un vaste domaine dans lequel interviennent de nombreux métiers différents, allant, par exemple, de l'analyste au développeur en passant par le testeur. Chaque métier associe ses propres savoir-faire et méthodologies afin de répondre à cette question. Savoir-faire et méthodologies qui font l'objet d'évolutions et d'innovations constantes afin, d'une part, d'accroître sans cesse cette productivité et cette qualité tant recherchée, d'autre part, de répondre à la complexité croissante des logiciels et de leur architecture.

Au niveau du programmeur, un concept fondamental permet de contribuer à ces objectifs de qualité et de rapidité de développement : la *réutilisation de code*, que les anglophones désignent parfois par la devise « Don't repeat yourself ». La réutilisation de code consiste à regrouper des morceaux de code identiques d'un programme en un emplacement commun afin d'éviter au maximum que des mêmes lignes de code se retrouvent dupliquées en différents endroits.

La réutilisation de code est fondamentale en programmation car elle apporte de nombreux avantages [BMN03]. Tout d'abord, le code devient beaucoup plus léger, et donc, par conséquent, beaucoup plus facilement compréhensible. Ce principe permet également une correction plus rapide des erreurs. En effet, lors de la détection d'une erreur dans un code dupliqué à de multiples endroits, la correction doit être apportée à chacun des emplacements où se situe le code fautif. Par contre, lorsque le code commun est regroupé en un endroit unique, la correction ne doit plus être effectuée qu'une seule fois. On évite ainsi par la même occasion une source d'erreurs courante qui consiste à oublier une correction dans un des duplicatas de code. De même, la maintenance du code s'en voit fortement simplifiée. Un ajout ou une modification ne doit plus être répercuté partout où le code commun apparaît, un seul emplacement est à modifier.

Afin que le programmeur puisse respecter au mieux ce principe clé de réutilisation de code, il importe que les langages de programmation qu'il utilise lui offrent des outils et des mécanismes permettant une réutilisation optimale du code. Par exemple, la programmation procédurale[1] traditionnelle offre les concepts de *procédures* ou *fonctions* qui permettent de regrouper un ensemble d'instructions communes. Une instruction particulière permet alors ensuite de faire *appel* à la

[1] Également nommée programmation structurée.

procédure ou à la fonction dont on veut réutiliser le code.

Dans ce mémoire, nous allons nous intéresser à un paradigme de programmation très populaire à l'heure actuelle [CDM+05] : la programmation orientée objet, imaginée dans les années septantes et qui propose de centrer la conception des logiciels autour de la notion d'*objets*. Au contraire, par exemple, de la programmation structurée qui centre la découpe des programmes autour des concepts de procédures et de fonctions.

Ce paradigme de programmation propose de nombreux mécanismes de réutilisation de code, imaginés et améliorés au fil des années. Nous nous intéresserons à deux mécanismes bien établis : l'*héritage* et les *mixins*, dont nous ferons la présentation, et qui feront ensuite l'objet d'une évaluation afin d'en dégager les éventuels défauts et limitations.

Il y a quelques années, un nouveau mécanisme de réutilisation de code a été imaginé par des chercheurs du *Software Composition Group* de l'Université de Berne : les *traits* dont l'objectif est de tenter de répondre aux limitations et problèmes posés par les méthodes de réutilisation de code existantes de la programmation orientée objet. Ce nouveau mécanisme fera l'objet d'une large présentation pour être ensuite évalué.

Assez rapidement, une amélioration aux traits d'origine a été proposée par les mêmes auteurs, lesquels ont développé la notion de traits *à états*, par opposition aux traits d'origine qui sont *sans état*. Nous ferons également la présentation de cette évolution des traits.

S'il est important de poser les bases théoriques d'un mécanisme de réutilisation de code, il est également important que le mécanisme soit proposé et implémenté par les langages de programmation qu'il vise, afin que le programmeur puisse bénéficier de ses apports. C'est pourquoi, nous étudierons l'ajout des traits à états à un langage de programmation orienté objet, à savoir *Java*, langage de programmation très populaire à l'heure actuelle [CDM+05] et qui ne propose pas encore ce mécanisme de réutilisation de code.

Pour ce faire, nous étudierons dans un premier temps les différentes manières d'intégrer les traits à états au langage Java. Nous proposerons ensuite d'étendre la syntaxe originelle de Java afin d'y intégrer cette notion de traits à états. Ces nouveaux éléments de syntaxe formeront les bases d'un nouveau langage, qui se veut être une extension de Java, que nous dénommerons *tJava*. Nous analyserons ensuite comment et avec quels outils un compilateur peut être implémenté pour ce nouveau langage. Parmi les outils analysés, nous n'en retiendrons qu'un, celui que nous jugerons convenir le mieux pour la réalisation d'un compilateur tJava. Nous validerons enfin notre choix en tentant une implémentation du compilateur.

1.1 Objectifs

Les objectifs visés par ce mémoire sont multiples. Nous tenterons premièrement d'introduire le plus largement possible la notion de traits. Pour ce faire, nous commencerons par présenter les raisons qui ont amené au développement de cette nouvelle méthode de réutilisation de code. Nous présenterons ensuite les traits dans leurs différentes variantes. Tout au long de notre présentation, de multiples aspects des traits seront analysés afin d'offrir un large résumé de la littérature publiée

à ce jour sur ce sujet. Nous tenterons également d'objectiver notre présentation en présentant diverses évaluations des traits.

Lorsqu'on désire sortir une technologie de son cadre formel, dans le but d'une éventuelle adoption, il est important de la vulgariser afin qu'un maximum de personnes puissent, d'une part, l'appréhender, d'autre part, en saisir les divers apports. C'est pourquoi, le deuxième objectif de ce mémoire sera *pédagogique* afin que soit ciblé un lectorat le plus large possible. En conséquence, les prérequis nécessaires à la bonne compréhension de ce mémoire se limitent à quelques notions triviales de programmation. Une pédagogie par l'exemple a été également largement développée afin d'aider à la compréhension des concepts plus complexes. À cet effet, de nombreux exemples ont été imaginés, et ceux repris de la littérature scientifique ont été largement retravaillés. De même, tous les exemples imaginés sont issus d'un même domaine afin d'éviter d'avoir à recontextualiser chaque nouvel exemple.

Enfin, le dernier objectif de ce mémoire sera de poser les bases d'une intégration des traits à états dans le langage de programmation Java, par le biais de la définition d'un nouveau langage, tJava. Diverses méthodes pour effectuer cette intégration seront étudiées, de même que les outils nécessaires à la réalisation de tJava. Finalement, nous tenterons une implémentation de tJava.

1.2 Organisation du document

Les traits se proposant d'améliorer les méthodes de réutilisation de code de la programmation orientée objet, le chapitre 2 sera dans un premier temps consacré à une présentation de ce paradigme de programmation afin de familiariser le lecteur avec certains de ses concepts clés (section 2.1). Nous introduirons ensuite deux méthodes de réutilisation de code de la programmation orientée objet : l'héritage (section 2.2) et les mixins (section 2.3). La section suivante (section 2.4) étudiera les problèmes et limitations des méthodes introduites.

Au chapitre 3 nous allons introduire les traits, dans leurs deux variantes, sans état (section 3.1) et à états (section 3.2). À la section suivante (section 3.3), nous étudierons quelles incidences les traits peuvent avoir sur un système de typage dans un contexte de langage typé statiquement. Enfin, nous terminerons par une brève présentation des travaux actuels et futurs sur les traits (section 3.4).

Le chapitre 4 sera consacré à l'étude de l'ajout des traits à états à Java, langage de programmation orienté objet que nous introduirons à la section 4.1. Deux approches seront étudiées pour ajouter les traits à Java. L'une (section 4.2) proposant l'émulation de traits à l'aide de la programmation orientée aspect (paradigme de programmation qui fera l'objet d'une introduction préalable), l'autre favorisant une intégration native des traits au langage Java. Pour cette deuxième approche, la définition d'un nouveau langage sera proposée à la section 4.3. Des nouveaux éléments de syntaxe seront introduits (section 4.3.1) et une extension de la grammaire originelle de Java sera proposée (section 4.3.2). Nous en profiterons également pour introduire préalablement le concept de grammaire. La section suivante (section 4.4) étudiera deux méthodes possibles pour implémenter un compilateur tJava : soit une modification des mécanismes du langage Java (section 4.4.1), soit l'utilisation d'une propriété importante des traits (section 4.4.2),

la propriété d'aplatissement.

Dans l'optique d'implémenter un compilateur pour tJava, nous étudierons au chapitre suivant (chapitre 5) les Rewritable Reference Attributed Grammars. Ce nouveau type de grammaire introduit de nouveaux concepts permettant de simplifier la conception de compilateurs. Pour présenter les Rewritable Reference Attributed Grammars, nous allons procéder par étapes en rappelant d'abord ce qu'est une canonical attributed grammar (section 5.1), ensuite en introduisant les Reference Attributed Grammars (section 5.2) et leur extension orienté objet pour finalement terminer par les Rewritable Reference Attributed Grammars (section 5.3). Nous étudierons enfin, à la section 5.4, dans quelle mesure les Rewritable Reference Attributed Grammars peuvent être utilisées pour l'implémentation d'un compilateur tJava.

Le chapitre suivant (chapitre 6) sera consacré au choix des outils pour implémenter le compilateur tJava. Nous établirons dans un premier temps (à la section 6.1) une liste de critères qui nous serviront à choisir l'outil le plus adapté. Nous présenterons ensuite à la section 6.2 une série d'outils éligibles. Nous parcourrons pour ce faire leurs avantages et leurs inconvénients et évaluerons leur respect des critères définis à la première section. Enfin, nous terminerons ce chapitre par un résumé, à la section 6.3, des qualités de chacun des outils pour finalement n'en retenir qu'un.

Le dernier chapitre (chapitre 7) sera consacré à la validation. Dans un premier temps nous allons présenter l'outil retenu, à savoir JastAdd II, en parcourant ses possibilités et en présentant son utilisation (section 7.1). À la section 7.2 nous présenterons notre tentative de modifier le JastAdd Extensible Java Compiler (un compilateur Java écrit à l'aide de l'outil précité, JastAdd II) afin d'y introduire la notion de traits et de réaliser ainsi l'implémentation de tJava.

Chapitre 2

Programmation orientée objet et réutilisation de code

Dans ce chapitre, nous allons nous intéresser à l'un des paradigmes de programmation les plus répandus à ce jour [CDM+05] et que les traits se proposent d'enrichir : la programmation orientée objet. Nous en rappellerons tout d'abord les principes fondateurs. Nous étudierons ensuite deux grandes méthodes de réutilisation de code issues de la programmation orientée objet : l'*héritage* et les *mixins*. Nous terminerons ce chapitre par une critique des méthodes de réutilisation de code étudiées. Nous relèverons ainsi une série de limitations et de défauts que les traits se voudront de pallier.

2.1 Concepts de la programmation orientée objet

Dans les années septante, une nouvelle approche d'analyse et de conception des programmes est apparue : la programmation orientée objet qui a permis l'introduction de nouvelles méthodes de réutilisation de code. L'orienté objet doit son origine à Ole-Johan Dahl et Kristen Nygaard qui introduisirent, au travers de leur langage de programmation Simula 67, les premières bases de ce nouveau paradigme de programmation [Weg90]. Néanmoins, ce n'est qu'avec Smalltalk 72, puis Smalltalk 80 que fut réellement défini et popularisé l'orienté objet [Weg90].

2.1.1 Les objets

En programmation orientée objet, *a contrario* de la programmation procédurale, le code n'est plus organisé autour de fonctions ou procédures[1], mais autour d'objets. Un objet est un regroupement de données et de fonctions (ou procédures) liées et est destiné à représenter des concepts ou des entités du monde physique. Imaginons par exemple un jeu vidéo de stratégie militaire qui fasse intervenir des soldats. En programmation orientée objet, le soldat sera représenté sous

[1] La différence entre une fonction et une procédure tient au fait qu'une fonction peut renvoyer à l'appelant une valeur, tandis qu'une procédure ne renvoie rien.

forme d'un objet possédant, d'une part, diverses données propres, telles que ses points de vie restant, son nom ou son acuité visuelle, d'autre part, une série de fonctions, telles que « attaquer », « mourir » ou « se soigner » qui permettent de définir le *comportement* du soldat.

Les fonctions et procédures d'un objet sont généralement appelées des *méthodes*, afin de les distinguer des fonctions ou procédures qui ne sont pas attachées à un objet. De même, les variables d'un objet sont parfois appelées des attributs. La figure 2.1 illustre l'exemple d'un objet « Soldat ».

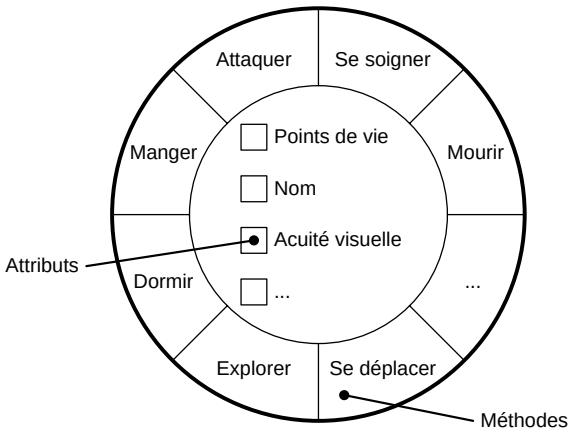

FIG. 2.1 – Exemple d'un objet

À un niveau plus conceptuel, il est souvent d'usage de penser un objet en termes de « responsabilités » qu'il est important de bien définir et limiter pour chacun des objets. Et ce afin que soient regroupés des attributs et des méthodes qui, mis ensemble, forment un objet cohérent et sensé. Par exemple, l'objet « Soldat » n'est certainement pas responsable de la gestion des points acquis par le joueur, les méthodes relatives à cet aspect du jeu vidéo verraient donc plutôt leur place au sein d'un autre objet (par exemple un objet « Jeu »).

2.1.2 Messages et interfaces

Afin de pouvoir interagir avec des objets, des *messages* peuvent être envoyés à ces derniers. Ils permettent d'enclencher l'exécution d'une des méthodes de l'objet destinataire du message. Un message est composé d'un nom et d'un ensemble de paramètres. Un même message peut éventuellement être envoyé à plusieurs objets différents. Chacun des objets peut alors interpréter différemment ce même message en lui associant une méthode propre ou, en d'autres mots, une implémentation propre. Cette possibilité d'interprétation différente d'un message en fonction de l'objet récepteur du message est appelée *polymorphisme* (mot venant du grec et signifiant « plusieurs formes »). Le processus d'association par un objet d'un message reçu à une méthode est appelé *method lookup*.

Un exemple est donné à la figure 2.2 où l'objet représentant le soldat Nelson envoie, à partir de sa méthode « Attaquer », le message « Tuer » à l'objet représentant le soldat Villeneuve. L'envoi de ce message a pour effet de lancer l'exécution de la méthode « Mourir » de l'objet représentant le soldat Villeneuve. La méthode « Mourir » est alors responsable d'effectuer tous

les traitements relatifs à la mort du soldat Villeneuve, par exemple mettre à zéro le nombre de points de vie restant.

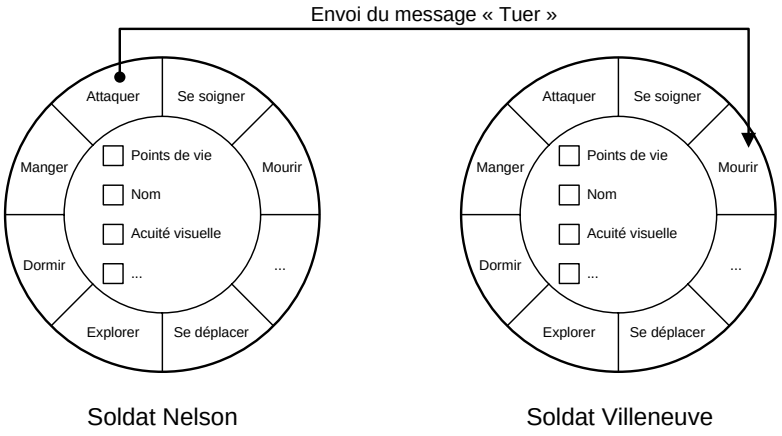

FIG. 2.2 – Exemple d'envoi d'un message d'un objet à un autre

Une méthode pouvant être déclenchée directement par l'envoi d'un message sera dite *publique* ou *accessible*. *A contrario*, une méthode destinée à un usage interne au sein d'une classe et qui ne peut être déclenchée directement par l'envoi d'un message sera dite *privée* ou *inaccessible*.

L'ensemble des méthodes accessibles d'un objet forme l'*interface* de l'objet [Weg90]. L'interface détermine également un ensemble de messages qui peuvent être envoyés à l'objet. Plusieurs objets peuvent avoir une interface identique, mais des implémentations complètement différentes. C'est le principe de *boîte noire* : l'implémentation et l'état d'un objet sont cachés des autres objets, seuls les messages permettent une interaction entre objets. Ce principe permet de changer facilement et de manière transparente l'implémentation associée à un message, sans impacter les interactions.

2.1.3 Les classes

Souvent, dans un programme, plusieurs objets de même type doivent être définis. Par exemple, dans l'exemple précédent (illustré à la figure 2.2), deux objets « Soldat » sont définis. Il serait redondant et inutile de devoir redéfinir pour chacun des soldats leurs méthodes, celles-ci étant probablement équivalentes. De même, la définition de leurs attributs est identique, seule la *valeur* des attributs diffère.

Une solution élégante a été imaginée : les *classes*, que l'on retrouve dans bon nombre de langages orientés objet. Une classe est un canevas définissant les méthodes et attributs pour un type particulier d'objets. Ce canevas est alors utilisé pour créer des objets, appelés *instances* de la classe, qui partagent un même comportement mais qui contiennent leurs propres valeurs pour les attributs. Une classe peut dès lors être vue comme un *générateur d'instances*. La figure 2.3 illustre le concept de classe. La classe « Soldat » définit les méthodes et attributs d'un soldat. Les trois instances de la classe, « Nelson », « Villeneuve » et « Napoléon » retiennent quant à elles les valeurs des attributs.

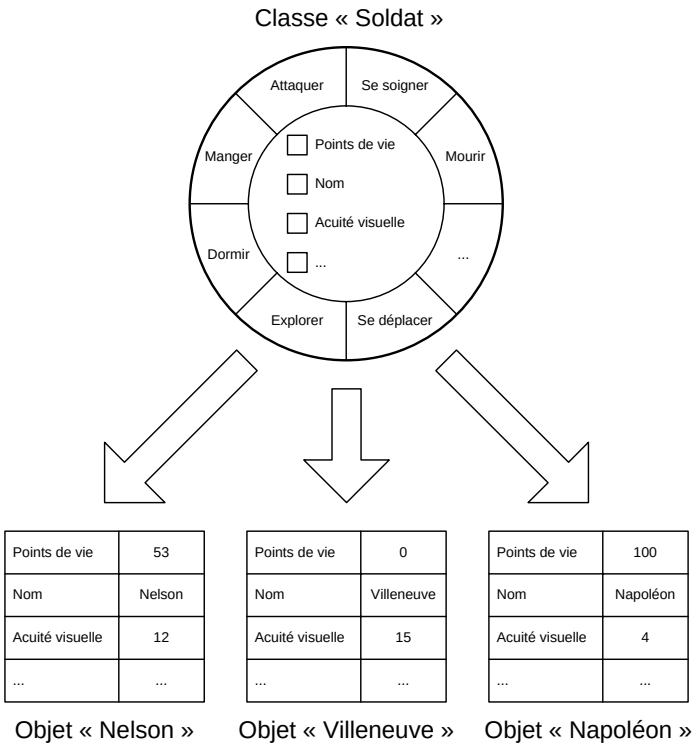

FIG. 2.3 – Exemple d'une classe

Le *method lookup* devient légèrement plus complexe, puisque l'objet doit retenir de quelle classe il est issu afin de pouvoir retrouver ses méthodes.

2.2 L'héritage

L'héritage est la première méthode de réutilisation de code introduite dans le paradigme orienté objet. Il doit son origine au constat suivant : souvent, plusieurs objets, bien que de type différent (en d'autres mots générés à partir de classes différentes), partagent un certain comportement commun qu'il serait avantageux de ne pas devoir implémenter plusieurs fois. Une première solution apportée par l'orienté objet à ce besoin de réutilisation de code est l'*héritage*. Ce mécanisme permet de dériver des classes existantes pour en créer de nouvelles, plus spécialisées. Cette spécialisation peut s'opérer de deux manières différentes :

- Soit en ajoutant à la nouvelle classe des méthodes additionnelles et en offrant une nouvelle interface, plus riche.
- Soit en proposant dans la nouvelle classe une implémentation différente de l'une ou l'autre méthode. On parle dans ce cas de *redéfinition de méthode* (ou en anglais de *method overriding*).

Imaginons que dans notre jeu vidéo de stratégie militaire nous désirions introduire plusieurs sortes de soldats, par exemple des archers et des cavaliers. Il est évident que ces deux types de soldats partagent une série d'attributs communs, ainsi qu'un comportement commun. Tous deux possèdent un nom, un nombre de points de vie restant, la possibilité de se déplacer, de mourir, etc.

Cependant, les archers et les cavaliers se distinguent des autres soldats par des caractéristiques propres. L'archer est, par exemple, capable de tirer des flèches tandis que le cavalier est, lui, capable de charger avec sa monture. Une modélisation basée sur l'héritage permettra d'exprimer typiquement ces spécificités, tout en regroupant les caractéristiques communes dans une classe de base.

Cet exemple est illustré à la figure 2.4 et nous permet par la même occasion d'introduire l'*UML*[2], une forme standardisée [Gro05] pour représenter des classes et leurs relations (*a contrario* de la forme utilisée dans les figures précédentes).

Sur cette figure, sont représentées trois classes. Tout d'abord, une classe de base « Soldat » regroupant les attributs et méthodes communs à tous les soldats. Cette classe de base est ensuite dérivée en deux autres classes plus spécialisées, « Archer » et « Cavalier », qui ajoutent toutes deux des attributs et des méthodes qui leur sont propres. La relation d'héritage entre la classe de base et les classes dérivées est symbolisée par une flèche terminée par un triangle creux.

Comme expliqué ci-dessus, la spécialisation ne s'arrête pas à l'ajout d'éléments dans la nouvelle classe, mais peut également s'opérer par la redéfinition de méthodes. Dans cet exemple, on a imaginé que la classe « Cavalier » propose une implémentation différente de la méthode « Se déplacer » puisqu'en plus du soldat à déplacer, il faut également déplacer la monture.

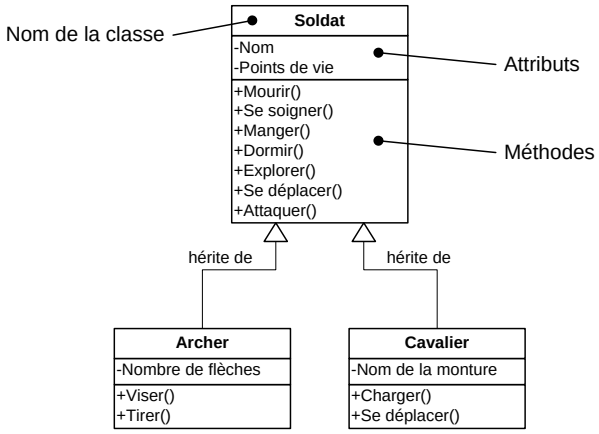

FIG. 2.4 – Exemple d'héritage

2.2.1 Une hiérarchie de classes

Une classe de base est également appelée classe *parente* ou *super-classe*. De même, une classe dérivée est également nommée classe *enfant* ou *sous-classe*. Une relation importante liant une classe enfant à sa classe parente est la relation « est ». Un archer « est » un soldat. L'inverse n'est pas vrai. Un soldat « n'est pas » spécialement un archer.

Une classe parente peut très bien elle-même hériter d'une autre classe parente et ainsi de suite. L'héritage ne se limite donc pas à deux niveaux. On voit dès lors apparaître une *hiérarchie* de classes dont la taille peut être arbitrairement grande. Il est souvent coutume d'avoir au sommet

[2]Unified Modeling Language

de cette hiérarchie une classe « Objet » dont descendent toutes les autres classes puisqu'on peut dire que toute classe descendante « est » un « Objet ». C'est dans cette classe que sont regroupés tous les attributs et méthodes communs à toutes les classes. Un exemple de hiérarchie est illustré à la figure 2.5

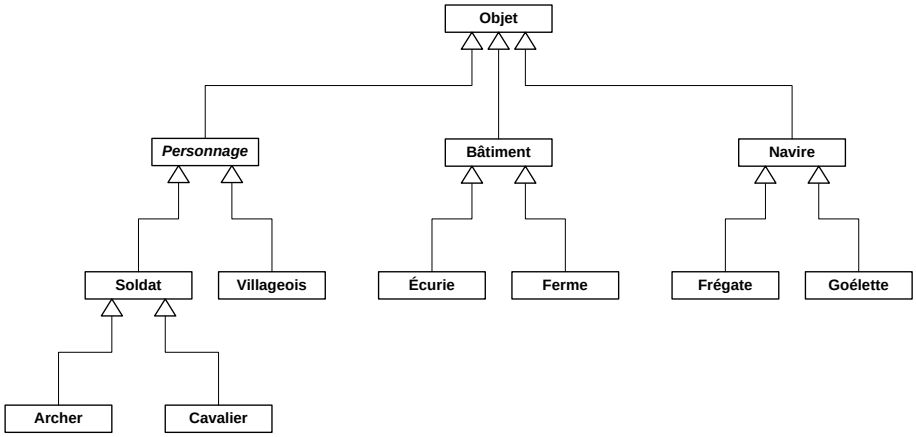

FIG. 2.5 – Exemple de hiérarchie de classes

2.2.2 Classes abstraites

Certaines classes parentes ne sont parfois utilisées que pour regrouper du comportement commun à toutes leurs classes enfants et ne permettent pas la génération d'instances. Ces classes sont appelées des classes *abstraites*, elles ne peuvent servir de canevas pour la création d'objets concrets. Ces classes contiennent généralement des *méthodes abstraites*, c'est-à-dire des méthodes déclarées au sein de la classe abstraite mais dont l'implémentation est laissée aux classes dérivées.

En reprenant notre exemple, on pourrait imaginer que la classe « Soldat » soit abstraite, celle-ci étant, par exemple, incapable de fournir une implémentation générique à tous les soldats pour la méthode « Attaquer » dont le comportement serait entièrement dépendant du type précis du soldat.

Une classe héritant d'une classe abstraite peut à son tour laisser l'implémentation des méthodes abstraites héritées à charge de ses classes dérivées. Elle est alors également abstraite.

2.2.3 Method lookup

L'utilisation de l'héritage complexifie légèrement le *method lookup*. En effet, lorsqu'un message est envoyé à un objet, la méthode correspondant au message ne se trouve peut-être plus dans la classe ayant servi à instancier l'objet, il faudra éventuellement aller la chercher dans une classe parente.

La figure 2.6 illustre un *method lookup* non trivial. Le message « Tuer » est envoyé à l'objet « Murat » qui est un « Cavalier ». Malheureusement, la méthode « Mourir » associée au message

« Tuer » n'est pas implémentée par la classe « Cavalier », il faut aller la chercher dans la classe parente « Soldat ».

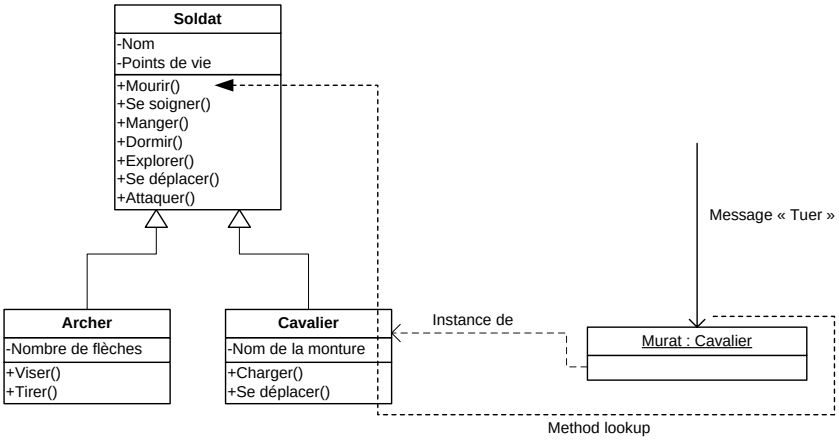

FIG. 2.6 – Exemple de *method lookup* lors d'un héritage de méthodes

2.2.4 Destinations particulières pour l'envoi de messages

Lorsqu'un objet, au sein d'une de ses méthodes, désire s'envoyer un message à lui-même, il peut utiliser un mot-clé spécial (souvent *this* ou *self*) pour se désigner en tant que destinataire du message.

La figure 2.7 illustre l'usage du mot-clé *self* au sein d'une méthode et le *lookup* initié en conséquence. Le message « Explorer » est envoyé à « Murat », objet de type « Cavalier ». Le *lookup* (1) de la méthode « Explorer » correspondante démarre donc au niveau de la classe « Cavalier ». Celle-ci ne s'y trouve pas. On continue donc le *lookup* au niveau de la classe parente, à savoir la classe « Soldat ». La méthode « Explorer » y est déclarée. Son implémentation a comme première instruction l'envoi du message « Se déplacer » à l'objet lui-même à l'aide du mot-clé *self*. Le *lookup* de la méthode « Se déplacer » (2) démarre alors au niveau de la classe « Cavalier » puisque l'objet « Murat » en est une instance.

Dans la même optique, un autre mot-clé (souvent *super)* permet à un objet, au sein d'une de ses méthodes, de s'envoyer un message en commençant le *method lookup* dans la classe parente de la classe où est implémentée la méthode à l'origine de l'envoi du message.

La figure 2.8 illustre l'usage du mot-clé *super* au sein d'une méthode et le *lookup* initié en conséquence. L'exemple est identique au précédent, à la différence que, cette fois-ci, le message « Se déplacer » est envoyé à *super*. En conséquence, le *method lookup* initié (2) commence au niveau de la classe « Personnage », classe parente de la classe « Soldat » où est implémentée la méthode « Explorer ». Il est important de remarquer que le *method lookup* ne commence pas au niveau de la classe parente de la classe « Cavalier » qui a servi à l'instanciation de l'objet « Murat ».

Une différence fondamentale distingue le *lookup* effectué suite à l'envoi d'un message à *self* ou à *super*. Dans le cas d'un message envoyé à *super*, le *lookup* peut être effectué à la compilation. On

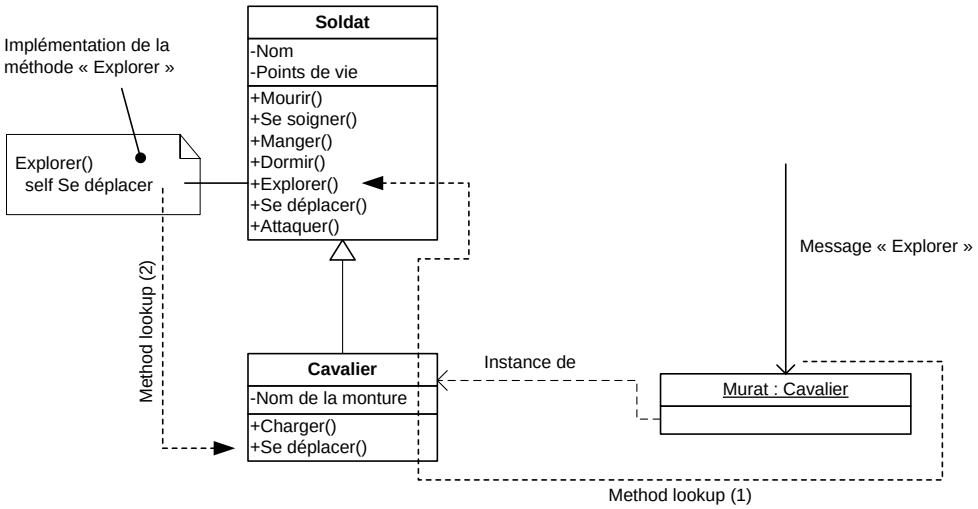

FIG. 2.7 – Exemple de *method lookup* lors de l'usage de *self*

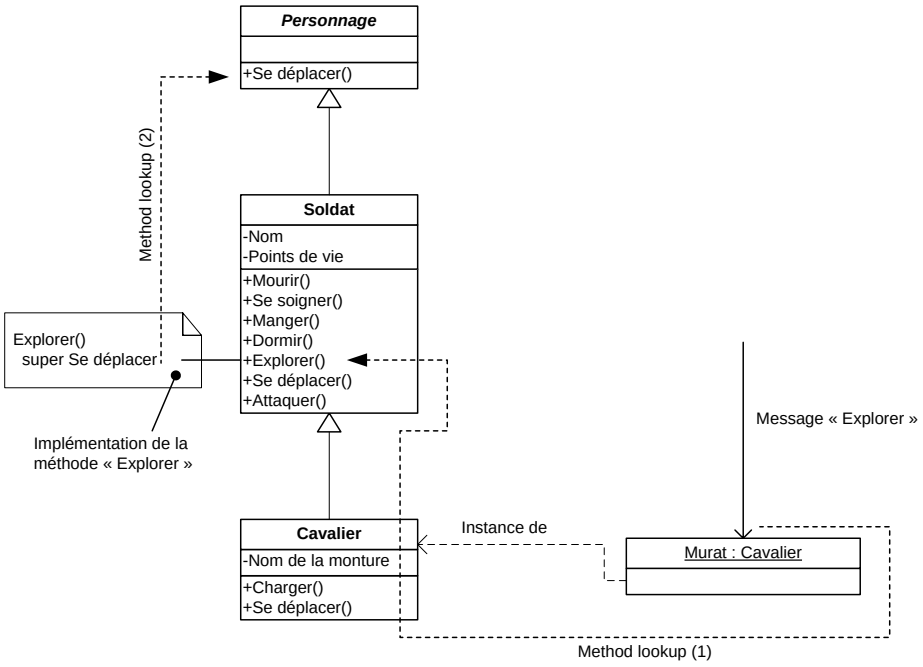

FIG. 2.8 – Exemple de *method lookup* lors de l'usage de *super*

sait en effet à ce stade déterminer quelle est la classe parente de la classe où se situe l'instruction d'envoi de message. Peu importe le type réel de l'objet à l'origine de l'envoi du message. Par contre, lors de l'envoi d'un message à *self*, le *lookup* dépend du type réel de l'objet. Dans notre exemple, le *lookup* est initié à partir de la classe « Cavalier ». Mais si Murat avait été de type « Archer », il aurait fallu démarrer le *lookup* à partir de la classe « Archer ». En conséquence, la méthode correspondant à l'instruction « self Se déplacer » ne peut être déterminée à la compilation ; ce travail doit être effectué lors de l'exécution du programme.

2.2.5 Multiplicité de l'héritage

On distingue deux types d'héritage, selon leur multiplicité :

– L'héritage *simple* où une classe ne peut hériter que d'une seule autre classe.
– L'héritage *multiple* où une classe peut hériter d'une ou plusieurs autres classes.

Il est souvent utile pour une classe de pouvoir être construite à partir de plusieurs autres classes afin de reprendre le comportement de chacune. Si l'on considère à nouveau l'exemple du jeu vidéo de stratégie militaire, on peut imaginer un nouveau type de soldat « Archer de cavalerie » (soit un archer se déplaçant à cheval) qui « est » à la fois un « Archer » et un « Cavalier ». Il est dans ce cas opportun d'utiliser l'héritage multiple afin de construire notre nouvelle classe « Archer de cavalerie », en illustre la figure 2.9.

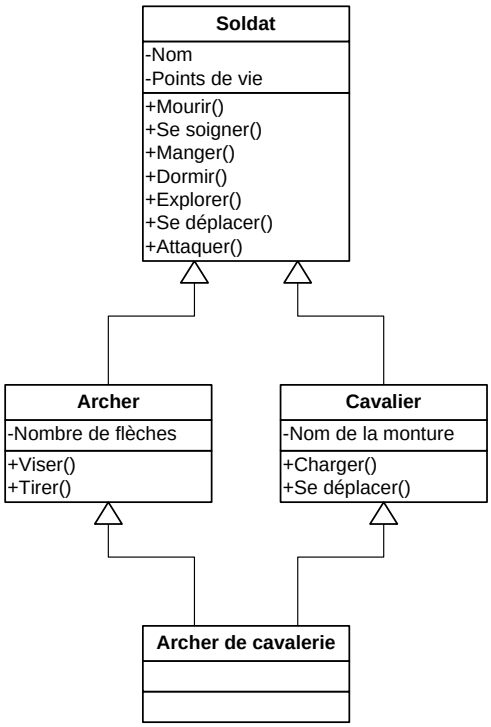

FIG. 2.9 – Exemple d'héritage multiple

2.3 Les mixins

Basés sur *Flavors* [Moo86, Can82], une extension au langage de programmation Lisp, la notion de *mixins* fut développée en 1990 par Gilad Bracha et William Cook [BC90]. Les mixins sont des pures unités de réutilisation de code qui permettent d'ajouter un même comportement à plusieurs classes. Plus précisément, un mixin est une *sous-classe abstraite* (donc non-instanciable) utilisée pour spécialiser le comportement d'une classe parente. Le lien entre un mixin et sa classe parente est obtenu à l'aide de l'héritage simple.

Reprenons à titre d'exemple la hiérarchie de classes présentée à la figure 2.5. Intéressons-nous cette fois aux navires et bâtiments. Imaginons que l'on veuille les rendre inflammables. Plutôt que de regrouper les méthodes et attributs caractéristiques d'un objet inflammable dans une classe parente aux classes « Navire » et « Bâtiment », utilisons un mixin que l'on appliquera à ces deux classes pour obtenir à la fois des « Navire inflammable » et des « Bâtiment inflammable ». La figure 2.10 illustre cet exemple. La flèche pointillée représente l'application d'un mixin.

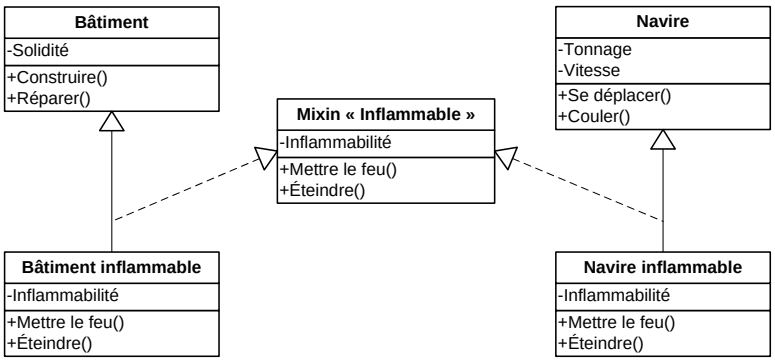

FIG. 2.10 – Exemple d'un mixin

Afin de mieux comprendre la philosophie sous-jacente aux mixins, voyons comment ceux-ci peuvent être simulés en utilisant les concepts de l'orienté objet présentés à la section précédente. Pour obtenir la classe « Navire inflammable » à l'aide de la classe « Navire » et du mixin « Inflammable », nous aurions pu créer explicitement la classe « Navire inflammable », sous-classe de « Navire », et y recopier l'ensemble des méthodes et attributs du mixin « Inflammable ».

Cette simulation illustre bien l'usage intrinsèque de l'héritage simple par les mixins. Par conséquent, il n'est pas possible d'appliquer simultanément plusieurs mixins à une classe parente. Si l'on désire ajouter à une classe du comportement provenant, par exemple, de deux mixins, il faut passer par une classe intermédiaire sur laquelle on aura appliqué le premier mixin pour ensuite appliquer le second mixin à cette classe intermédiaire. Exemple en est donné à la figure 2.11.

Autre conséquence de l'utilisation intrinsèque de l'héritage simple par les mixins : si un mixin est appliqué sur une classe parente possédant une méthode portant le même nom qu'une méthode du mixin, la méthode provenant du mixin aura précédence sur celle de la classe parente. Il y a donc redéfinition de la méthode de la classe parente par le mixin.

Un mixin étant destiné à être greffé à une classe parente, celui-ci peut faire usage du mot-clé *super* afin d'accéder aux méthodes de la classe parente. Raison pour laquelle un mixin est une

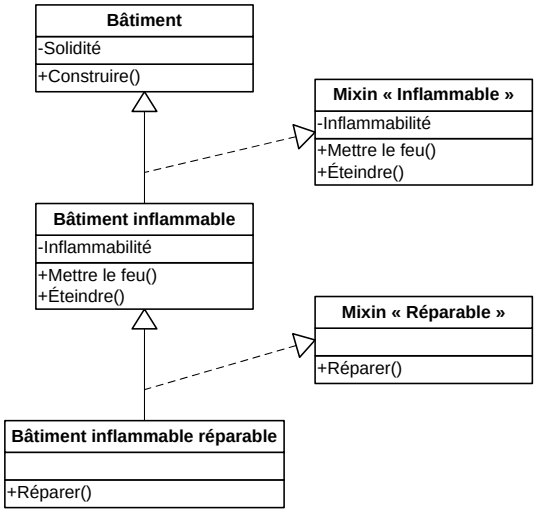

FIG. 2.11 – Exemple d'application de plusieurs mixins à une classe

classe abstraite et ne peut générer des instances, puisque l'implémentation de certaines méthodes est laissée à la classe parente.

2.4 Problèmes et limitations des méthodes existantes

Les méthodes de réutilisation de code orientées objet introduites dans les sections précédentes présentent un certain nombre d'inconvénients et de limitations, tant sur le plan de la *décomposition* que sur le plan de la *composition*. La décomposition désigne le processus de découpage du code en petites unités réutilisables, tandis que la composition se rapporte au processus inverse, c'est-à-dire l'assemblage et l'utilisation de ces petites unités. Dans cette section nous allons analyser, pour chacune des méthodes de réutilisation de code introduite précédemment, les problèmes conceptuels qu'elles engendrent.

2.4.1 Critique de l'héritage simple

L'héritage simple ne permet pas toujours de regrouper et de réutiliser toutes les caractéristiques communes à plusieurs classes. Reprenons l'exemple de hiérarchie de classes introduite dans les sections précédentes, et en particulier la classe « Archer de cavalerie » composée à partir de deux autres classes dont on aimerait réutiliser le comportement. L'héritage simple ne permet malheureusement pas d'exprimer cette double réutilisation. En conséquence, seul le comportement d'une des deux classes peut être réutilisé tandis que le comportement de l'autre classe devra être dupliqué. Une illustration est donnée à la figure 2.12.

Une solution souvent avancée afin d'éviter cette duplication de code est l'implémentation des fonctionnalités communes à plusieurs classes « *trop* haut » dans la hiérarchie. En reprenant l'exemple du jeu vidéo de stratégie militaire, imaginons que l'on rajoute à la hiérarchie de classes un nouveau type de navire permettant le transport de troupes et de chevaux. Une nouvelle

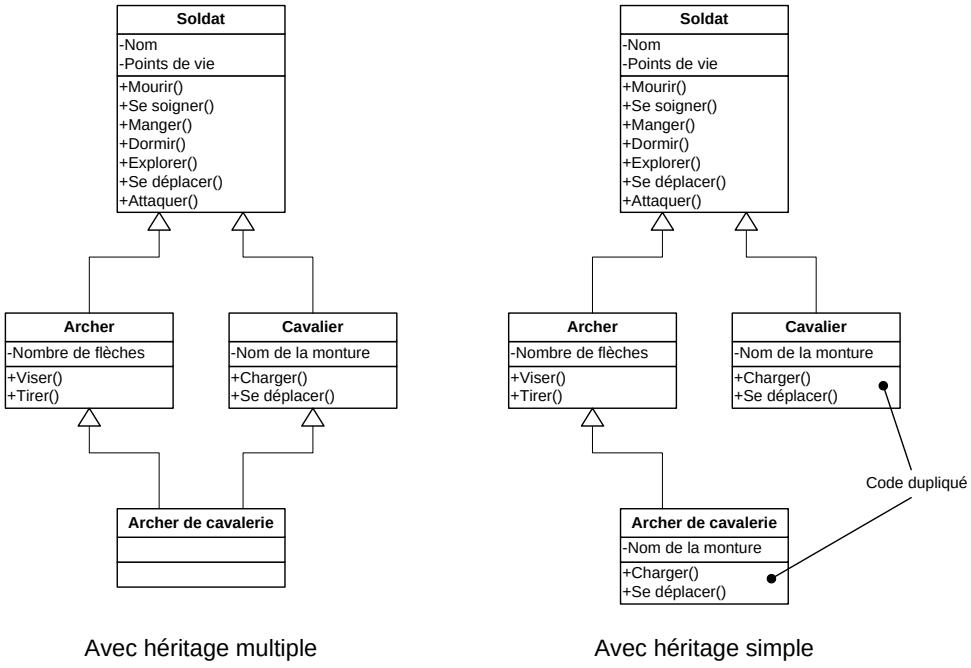

FIG. 2.12 – Exemple de duplication de code induit par les limitations de l'héritage simple

classe « Navire de transport » est ajoutée comme sous-classe de « Navire ». Deux classes dans cette hiérarchie se distinguent désormais par un comportement commun qu'il serait bon de regrouper. À la fois les navires de transport et les écuries peuvent contenir des chevaux. Ces deux classes possèdent donc des attributs communs, telles que le nombre de chevaux contenus, et des méthodes communes afin de pouvoir, par exemple, entrer ou sortir un cheval de l'écurie ou du navire. Malheureusement, si l'on désire implémenter ce comportement commun dans une classe supérieure commune, il n'est d'autre possibilité que de le faire dans la classe « Objet », seul ancêtre commun aux classes « Écurie » et « Navire de transport ». Cet exemple est illustré à la figure 2.13.

Si sur le plan fonctionnel, cette solution apporte une réponse au problème de duplication de code, le prix à payer n'en est pas moins assez lourd. La classe « Objet » se retrouve polluée avec du code qui ne lui est pas destiné et, pire, toutes ses sous-classes le sont également.

2.4.2 Critique de l'héritage multiple

Un des problèmes les plus souvent évoqués à charge de l'héritage multiple est *le problème du diamant* qui survient lorsqu'un même attribut ou une même méthode est hérité plusieurs fois par le biais de chemins différents. La figure 2.14 illustre cette situation.

Dans cet exemple, trois attributs de la classe « Archer de cavalerie » sont hérités de la classe « Soldat » via deux chemins différents. La question qui se pose alors est de savoir si, au sein de la classe « Archer de cavalerie », ces attributs doivent être fusionnés ou si deux instances distinctes de ces attributs doivent coexister. La réponse n'est malheureusement pas universelle puisque propre à chaque attribut, selon le sens qui lui est donné. S'il apparaît assez naturel que

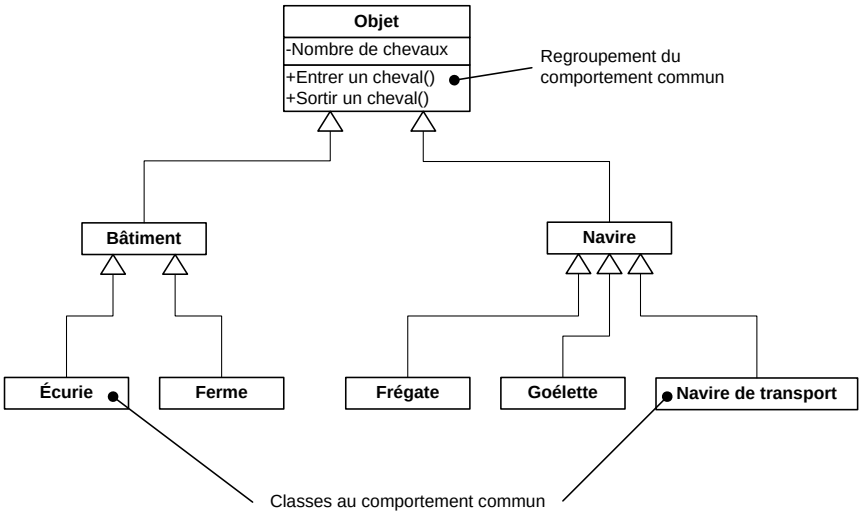

FIG. 2.13 – Exemple d'implémentation de comportement commun « trop haut » dans la hiérarchie

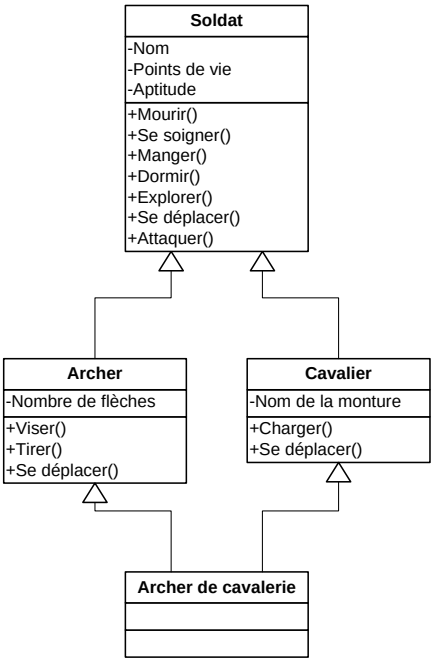

FIG. 2.14 – Exemple de problème du diamant

l'attribut « Nom » soit fusionné, il n'en est pas de même pour l'attribut « Aptitude ». En effet, l'archer de cavalerie, étant à la fois un archer et un cavalier, possède deux aptitudes différentes. D'une part son aptitude d'archer, d'autre part son aptitude de cavalier. Cet attribut ne doit donc pas être fusionné. Conséquence de cette différence de traitement entre attributs : les langages de programmation proposant de l'héritage multiple doivent enrichir et complexifier leur syntaxe afin que toutes les nuances de l'héritage puissent être exprimées.

Un autre problème concerne l'initialisation des attributs. Par exemple, quelle valeur initiale va recevoir l'attribut « Points de vie » de l'archer de cavalerie si l'on imagine qu'il est initialisé différemment par ses classes parentes « Cavalier » et « Archer » (un cavalier étant bien plus résistant qu'un archer, il recevra plus de points de vie initiaux). Afin d'éviter toute ambiguïté, une nouvelle valeur initiale doit donc être explicitement fournie par la classe « Archer de cavalerie ».

Si, au sein de la classe « Archer de cavalerie », on désire faire appel à la méthode « Se déplacer » héritée des classes parents, on doit être capable de distinguer la version de la méthode à appeler, soit celle définie dans la classe « Archer », soit celle définie dans la classe « Cavalier ». Le mot clé *super* ne suffit donc plus, le nom de la classe parente d'où provient la méthode à appeler doit être explicitement fourni lors d'un appel ambigu. Ce qui a pour effet malheureux de fortement renforcer les *dépendances* entre classes enfants et classes parents ; si l'on renomme la classe parente, le changement de nom doit être répercuté partout où est effectué un appel à une méthode du parent renommé.

Linéarisation du graphe des ancêtres

Afin de pallier les conflits engendrés par l'héritage multiple, certains langages de programmation, tel CLOS[3], procèdent à une *linéarisation* du graphe des ancêtres afin d'obtenir une *chaîne* d'héritage simple. Par exemple, si l'on reprend l'exemple de la classe « Archer de cavalerie », on pourrait obtenir la linéarisation suivante : « Archer de cavalerie », « Archer », « Cavalier », « Soldat », etc. comme illustré à la figure 2.15. Les avantages de cette linéarisation sont multiples. En plus d'empêcher l'héritage de mêmes attributs ou méthodes par le biais de chemins différents, la linéarisation permet de rétablir l'usage du mot-clé *super* pour identifier de manière univoque la classe parente.

Dans cet exemple, deux linéarisations sont possibles, l'une plaçant « Archer » avant « Cavalier », l'autre plaçant « Cavalier » avant « Archer ». Afin que l'algorithme de linéarisation puisse choisir de manière déterministe une solution parmi toutes les solutions possibles, un ordre de préférence doit être spécifié par le programmeur. Ce dernier devra donc se demander si un archer de cavalerie est d'abord un archer puis un cavalier ou l'inverse. Au niveau du code, cet ordre de préférence est souvent exprimé par l'ordre dans lequel les classes parentes sont déclarées. Par exemple, le pseudo-code suivant conduirait à la linéarisation « Archer de cavalerie », « Archer », « Cavalier » :

```
class ArcherDeCavalerie extends Archer, Cavalier
```

[3]Common Lisp Object System

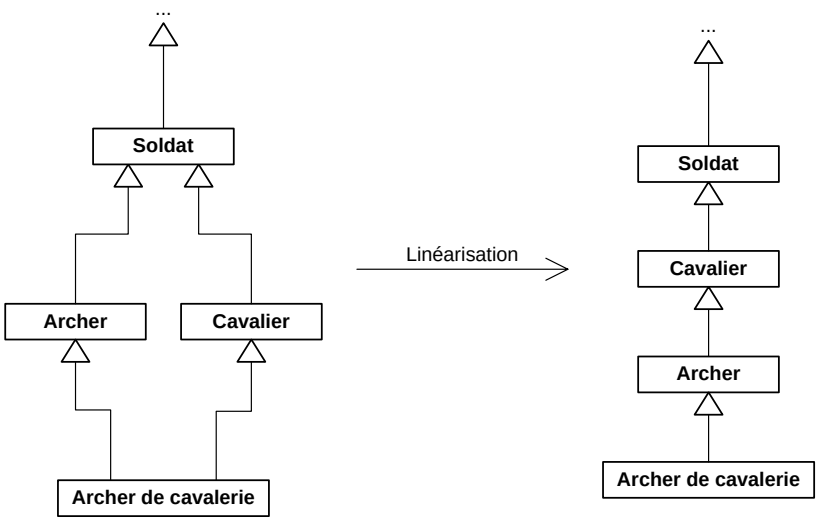

FIG. 2.15 – Exemple de linéarisation d'un héritage multiple

Malheureusement, la linéarisation souffre d'une série de désavantages. Tout d'abord, il n'est pas toujours possible de linéariser un graphe d'héritages en raison de contraintes trop fortes [BCH+96]. Un exemple est donné à la figure 2.16. Les classes « Cuirassé de transport » et « Navire de transport cuirassé » héritent toutes deux des mêmes classes de base « Cuirassé » et « Navire de transport ». Néanmoins, l'une représentera plutôt un cuirassé doté de facilités de transport, tandis que l'autre représentera plutôt un navire de transport équipé avec les fonctionnalités d'un cuirassé, l'ordre de préférence pour effectuer la linéarisation étant inversé. L'algorithme de linéarisation est donc soumis à deux contraintes d'ordre. D'une part la classe « Cuirassé de transport » impose que la classe « Cuirassé » hérite de la classe « Navire de transport », d'autre part la classe « Navire de transport cuirassé » impose que la classe « Navire de transport » hérite de la classe « Cuirassé ». Ces deux contraintes étant opposées, il est impossible de les satisfaire. La linéarisation est donc impossible ; on parle dans ce cas de graphe d'héritage *inconsistant*.

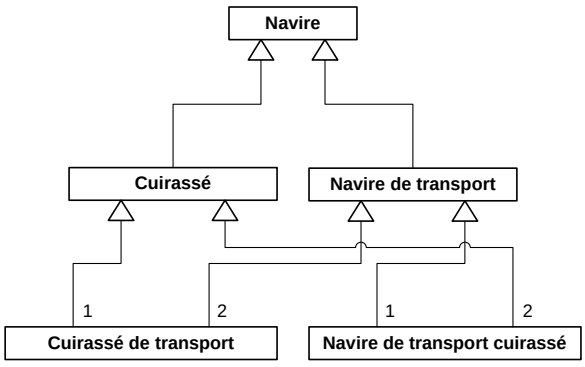

FIG. 2.16 – Exemple de graphe d'héritage *inconsistant*

Une autre conséquence problématique de la linéarisation est la fragilisation de la hiérarchie. En effet, l'ajout d'une classe ou la modification d'une relation d'héritage dans la hiérarchie peut résulter en une linéarisation totalement différente. Si l'on reprend la hiérarchie de la figure 2.15, alors que l'auteur de la classe « Archer » suppose un héritage direct entre sa classe et la classe parente « Soldat », l'ajout de la classe « Archer de cavalerie » dans la hiérarchie modifie com-

plètement ce comportement : la classe « Cavalier » vient s'insérer dans la relation directe qui liait « Archer » à « Soldat ». Le programmeur doit donc avoir une connaissance complète de la hiérarchie de classes afin de pouvoir anticiper ce genre d'effets de bord, ce qui n'est pas forcément souhaitable.

Enfin, tout comme les mixins, la linéarisation est soumise à des difficultés liées, d'une part, à la redéfinition implicite de méthodes, d'autre part, à la recherche et à l'existence d'un ordre total pour les fonctionnalités offertes par les différentes classes de la hiérarchie. Ces difficultés seront étudiées dans la section suivante dont l'objet est d'identifier et de décrire les problèmes liés à l'usage de mixins.

Conclusion

Bien que l'héritage multiple propose une approche *a priori* naturelle et intuitive de réutilisation de code, les divers problèmes évoqués précédemment dévoilent en fait un mécanisme qui s'avère à la fois complexe à comprendre pour le programmeur et complexe à implémenter pour les concepteurs de compilateur. Raison pour laquelle de nombreux langages orientés objet actuels ne proposent pas d'héritage multiple (c'est le cas par exemple de Java ou des langages .Net), leurs concepteurs ayant jugé que cette forme d'héritage apporte plus de problèmes qu'elle n'en résout. En guise de conclusion, nous retiendrons donc de l'héritage multiple cette citation d'Alan Snyder :

> « Multiple inheritance is good, but there is no good way to do it. » [Coo87]

2.4.3 Critique des mixins

Les problèmes inhérents aux mixins proviennent essentiellement de leur composition linaire basée sur l'héritage simple. En d'autres mots, si on désire appliquer plusieurs mixins à une classe, ceux-ci ne peuvent l'être en même temps, des classes intermédiaires devront être utilisées afin d'appliquer les mixins l'un après l'autre.

Un premier problème peut survenir lorsqu'un mixin possède une méthode dont le nom est identique à une méthode de la classe parente sur laquelle le mixin est appliqué. L'usage intrinsèque de l'héritage simple par le mixin aura pour conséquence de *redéfinir* la méthode de la classe parente. Parfois cette redéfinition n'est pas désirée. Dans ce cas, afin de conserver l'accès à la méthode originale, l'ajout d'une classe supplémentaire dans la hiérarchie sera alors nécessaire afin de simuler un renommage de la méthode au nom conflictuel. On parle dans ce cas de *code de colle* (*glue code* en anglais), en référence à du code qui ne sert pas l'objet principal du programme, mais qui est destiné à faire la jonction entre plusieurs de ses composantes.

Un exemple de redéfinition de méthode par un mixin est illustré à la figure 2.17. Sur cet exemple, deux mixins sont introduits, afin d'ajouter des fonctionnalités de commerce au jeu vidéo. Le premier mixin permet de rendre un objet achetable et comporte, entre autres, un attribut « prix » et une méthode « ObtenirLePrix » offrant la possibilité de récupérer ce prix. Le deuxième mixin permet de rendre taxable un objet auparavant défini comme achetable. Pour ce faire, un

attribut « Taux », représentant le taux de taxation, est ajouté et la méthode « ObtenirLePrix » est redéfinie afin d'ajouter une taxe au prix défini dans la classe parente. Ces deux mixins sont appliqués successivement à la classe « Navire » afin de permettre au joueur d'acheter des navires pour compléter sa flotte (ou du moins d'en connaître le prix, les méthodes d'achat n'étant pas reprises).

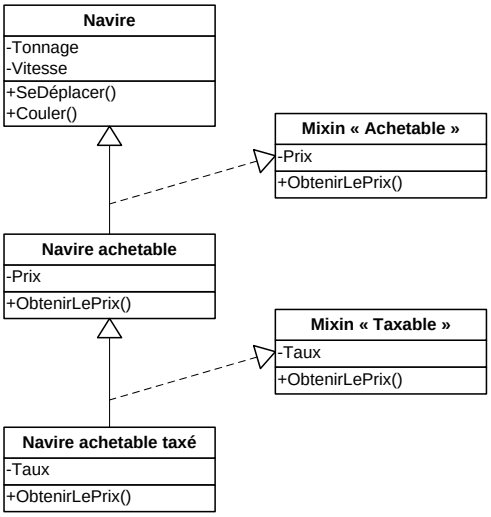

Fig. 2.17 – Exemple de redéfinition de méthode par un mixin

La redéfinition de la méthode « ObtenirLePrix » opérée par le mixin « Taxable » a pour effet de rendre indisponible le prix d'origine pour la classe finale résultante. Afin de rendre ce prix accessible, un *mixin de colle* doit être appliqué avant que la redéfinition ne soit opérée. Illustré à la figure 2.18, ce mixin comporte une nouvelle méthode « ObtenirLePrixHorsTaxe » faisant appel à la version originale de la méthode « ObtenirLePrix » se trouvant dans la classe parente. Le prix original est alors disponible via cette nouvelle méthode.

Les redéfinitions opérées par les mixins fragilisent fortement la hiérarchie de classes. En effet, un mixin ajouté en haut d'une hiérarchie peut résulter en la nécessité de renommer une méthode, changement qui doit être répercuté dans toutes les classes descendantes partout où apparaissent des appels à la méthode renommée. De plus, l'ajout de code de colle un peu partout dans la hiérarchie a tendance à la complexifier, nuisant ainsi à sa compréhension.

Un autre problème inhérent aux mixins est lié à leur linéarisation. Il est en effet parfois difficile, voire impossible, de trouver un ordre total approprié dans lequel appliquer les mixins. Un exemple est donné à la figure 2.19. Par rapport à l'exemple précédent, un nouveau mixin est introduit, « Réductionnable », qui permet d'appliquer une réduction de prix à un objet auparavant défini comme achetable. Pour ce faire, un attribut « Réduction », représentant la ristourne, est ajouté et la méthode « ObtenirLePrix » est redéfinie afin de soustraire le montant de la réduction du prix défini dans la classe parente. Les trois mixins appliqués l'un après l'autre permettent dès lors de définir un prix pour l'objet, de majorer ce prix d'une taxe et de lui soustraire une réduction. Une nouvelle méthode est également ajoutée aux trois mixins, « PrixEnTexte », permettant de retourner le prix sous forme textuelle. Au niveau du mixin « Achetable » cette méthode retourne simplement le prix suivi du nom de la devise « euros ». Le mixin « Taxable » retourne quant à lui le montant majoré de la taxe suivi du nom de la devise « euros ». Le montant d'origine, à

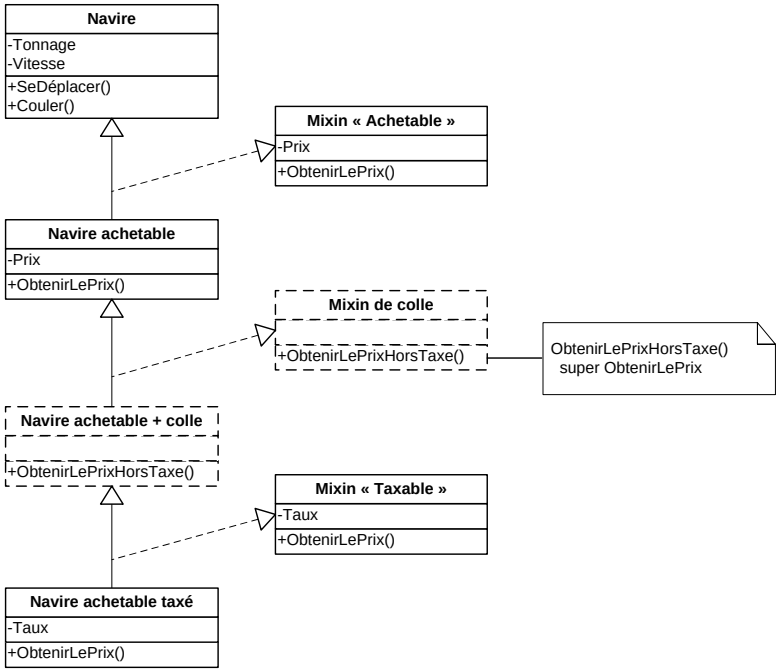

FIG. 2.18 – Exemple de *code de colle* pour simuler le renommage d'une méthode

partir duquel est calculé la taxe, n'est pas affiché. Enfin, au niveau du mixin « Réductionnable », il est d'abord demandé à la classe père de retourner la version textuelle du prix pour ensuite lui adjoindre le montant de la réduction. En imaginant un montant de 100 euros, une taxe de 20 % et une ristourne de 4 euros, le montant textuel produit serait : « 120 euros TTC moins 4 euros de réduction ».

L'ordre dans lequel sont appliqués les trois mixins a une importance cruciale dans le calcul du prix final. Tel qu'ordonné dans l'exemple, le calcul de la taxe est effectué *avant* l'application de la réduction. Si maintenant on désire changer l'ordre des calculs et calculer la taxe *après* avoir déduit la réduction, il suffit d'inverser l'ordre dans lequel sont appliqués les mixins « Taxable » et « Réductionnable ». Malheureusement, cette inversion a également pour effet de modifier le comportement final de la méthode « PrixEnTexte » : la réduction n'est plus affichée (puisque la méthode « PrixEnTexte » du mixin « Taxable » n'affiche pas le montant d'origine, à partir duquel est calculée la taxe). Reprenant les valeurs numériques de l'exemple précédent, le montant textuel produit serait : « 115,2 euros TTC ».

Il n'existe donc pas d'ordre total approprié dans lequel appliquer les mixins permettant à la fois de calculer la taxe *après* avoir déduit la réduction et d'afficher le montant de la réduction. La seule solution est de modifier le code des mixins, changement qui n'est peut-être pas désiré par d'autres classes sur lesquelles sont appliqués ces mixins.

2.4.4 Les classes comme unités de réutilisation

À la fois l'héritage simple et l'héritage multiple attribuent deux rôles bien distincts aux classes : d'une part, elles servent de canevas à la génération d'objets, d'autre part, elles sont

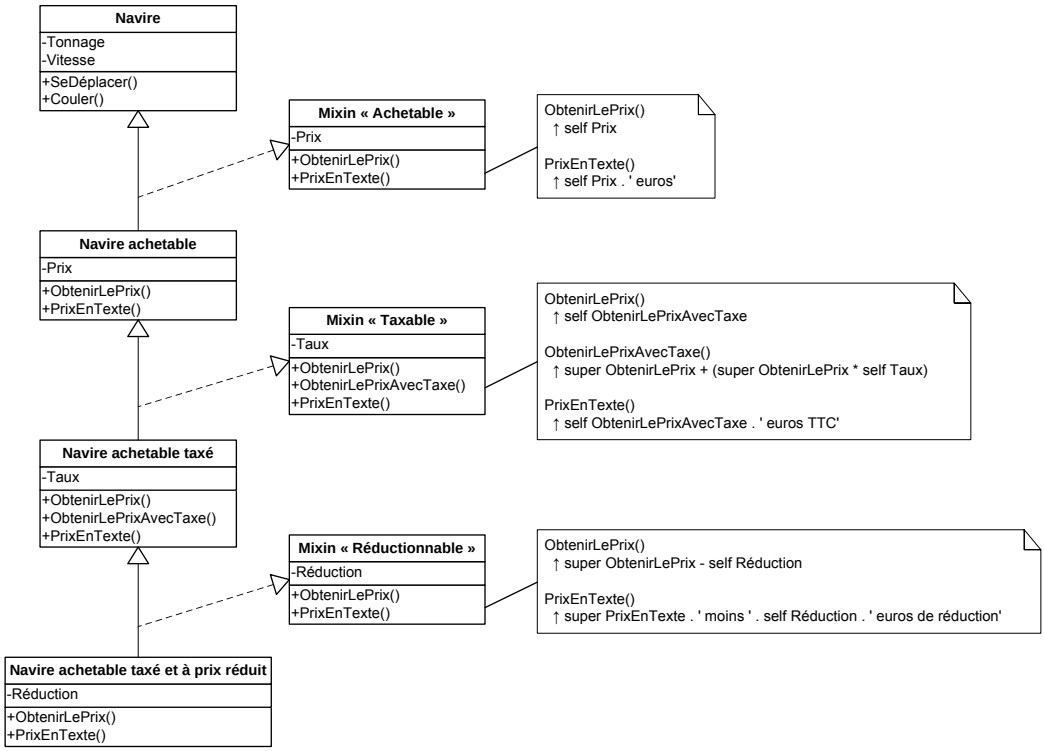

FIG. 2.19 – Exemple d'importance de l'ordre d'application de mixins

utilisées comme unités de réutilisation de code. Servir de générateur d'instances suppose que l'objet créé sera *complet*, c'est-à-dire qu'il contiendra *toutes* les méthodes et *tous* les attributs qui lui correspondent. Par contre, une unité de réutilisation de code ne sera optimale que si elle regroupe un *minimum* de fonctionnalités. Elle sera dès lors plus facilement réutilisable dans d'autres contextes. On le voit donc, les deux rôles attribués aux classes sont en opposition : d'une part on désire maximiser du comportement, d'autre part on désire le minimiser.

C'est pourquoi un autre type d'approche de réutilisation de code est souvent préféré, i.e. les approches qui limitent les classes à leur premier rôle de générateur d'instances et qui se tournent vers d'autres structures pour effectuer la réutilisation de code. C'est le cas des mixins. Certes ceux-ci sont définis comme étant des sous-classes, mais c'est sans compter qu'elles sont abstraites. Les mixins ne peuvent donc servir de générateur d'instances et ne sont utilisés qu'à des fins de réalisation de code reléguant aux classes leur rôle principal de générateur d'instances.

Chapitre 3

Les traits

Les traits offrent une solution légère et élégante aux problèmes de composition et décomposition évoqués dans le chapitre précédent. Imaginés par Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz et Andrew P. Black [SDNB03], les traits sont des unités de réutilisation de code permettant d'enrichir les classes avec du comportement. *A contrario* des classes, les traits ne peuvent servir à la génération d'objets. Leur rôle est donc uniquement de servir d'unité de réutilisation de code, recentrant par la même occasion les classes sur leur rôle premier de génératrice d'instances. Les traits se veulent être un complément au mécanisme bien établi d'héritage simple avec lequel la cohabitation parfaite est assurée.

Dans leur première version, les traits se limitaient à regrouper un ensemble de méthodes et ne pouvaient pas contenir d'attributs (i.e. des variables d'instance). On parle alors de *traits sans état*. Par la suite furent introduits les *traits à états* [BDNW07], en d'autres mots des traits pouvant contenir des attributs.

La première section de ce chapitre aura pour but d'introduire les traits sans état. Seront analysés les aspects de composition et décomposition ainsi que les mécanismes de résolution de conflits. Une évaluation des traits sans état clôturera cette section et analysera les éventuelles réponses apportées par les traits aux problèmes de composition et décomposition largement évoqués au chapitre précédent.

À la section suivante, nous étudierons les traits à états en commençant par dégager une série de limitations dues à l'absence d'état dans les traits. Nous définirons ensuite les traits à états et analyserons les mécanismes supplémentaires de composition qu'ils proposent. Une évaluation des traits à états terminera cette section. Cette évaluation revisitera notamment le problème du diamant évoqué au chapitre précédent.

Enfin, nous terminerons ce chapitre en présentant certains travaux actuels et futurs concernant les traits, et notamment quelques implémentations réalisées afin d'ajouter des traits à des langages existants.

3.1 Les traits sans état

Les traits sans état, que nous dénommerons simplement traits tout au long de cette section, sont essentiellement un ensemble de méthodes permettant de participer à la construction à la fois de classes et d'autres traits. Ces derniers seront appelés des traits *composites* car composés à partir d'autres traits.

Deux types de méthodes sont à distinguer au sein d'un trait :

- Les méthodes *fournies*, i.e. les méthodes dont l'implémentation est proposée par le trait ;
- Les méthodes *requises*, i.e. les méthodes dont le trait fait usage mais dont l'implémentation est laissée à l'utilisateur du trait.

Les méthodes requises seront, entre autres, utilisées par un trait afin de leur offrir une possibilité d'accéder indirectement à des variables d'état. En effet, lorsqu'un trait désire utiliser une variable d'état, puisqu'il ne peut la définir en son sein, il lui suffira de définir des méthodes *requises* permettant d'accéder et de modifier[1] cette variable. Charge alors à la classe utilisatrice du trait de fournir une implémentation à ces méthodes, et donc un accès à la variable d'état que la classe aura pu définir en son sein.

Un exemple de trait est donné à la figure 3.1. Le langage UML est légèrement enrichi afin de permettre la représentation des traits. Les traits seront représentés de la même manière qu'une classe, à l'exception de l'absence de la liste d'attributs et de la division de la liste des méthodes en deux colonnes. La colonne de droite reprend les méthodes *fournies*, celle de gauche les méthodes *requises*. Afin de mieux distinguer un trait d'une classe dans un diagramme UML, les noms des traits seront préfixés par convention de la lettre « T ». Les noms de classes ne seront, eux, pas préfixés par une lettre particulière.

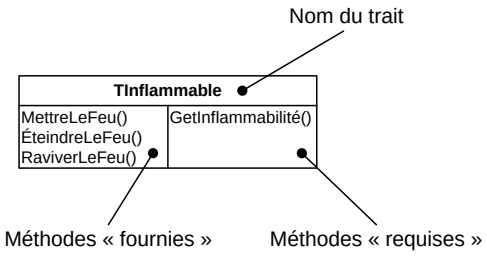

FIG. 3.1 – Un premier exemple de trait sans état

3.1.1 Composition de classes à partir de traits

En programmation orientée objet traditionnelle, les classes sont composées selon l'équation suivante :

$$classe = classe\ parente + attributs + méthodes\ de\ spécialisation$$

Une classe est construite à partir d'une classe parente, un ensemble d'attributs et un ensemble

[1] Ces méthodes sont appelées respectivement *accesseurs* (*getter* en anglais) et *mutateurs* (*setter* en anglais).

de méthodes permettant la spécialisation (exprimée soit par la redéfinition, soit par l'ajout de méthodes).

La construction de classes sur base éventuelle de traits sera effectuée selon l'équation suivante :

$$classe = classe\ parente + attributs + méthodes\ de\ spécialisation + traits + code\ de\ colle$$

Tout comme en programmation orientée objet traditionnelle, une classe est construite à partir d'une classe parente, un ensemble d'attributs et un ensemble de méthodes de spécialisation, mais elle est en plus construite à partir d'un ensemble de traits. Du code de colle est également ajouté afin de connecter les traits ensemble, i.e. implémenter les méthodes requises, adapter les méthodes fournies par le trait et résoudre les conflits de méthodes (cf. section 3.1.3).

Dans leurs présentations des traits, contrairement à l'équation définie ci-dessus, les auteurs omettent de préciser que des méthodes de spécialisation peuvent être ajoutées à une classe, sans pour autant qu'elles ne proviennent de traits ou qu'elles ne servent de colle. Sans doute dans l'espoir qu'une classe ne serait plus construite *que* sur base de traits. Néanmoins, les traits se voulant un complément au modèle orienté objet d'origine, avec lequel ils partagent une compatibilité ascendante totale[2], il est tout à fait possible d'ajouter à une classe des méthodes de spécialisation sans qu'elles ne proviennent d'un trait. Cette possibilité est dès lors reprise dans l'équation.

Une classe concrète est dite *complète* ou *bien fondée* si elle propose une implémentation pour toutes les méthodes requises des traits qui composent la classe. Les méthodes requises peuvent être implémentées dans la classe elle-même, dans une classe parente directe ou indirecte ou encore peuvent provenir d'un autre trait utilisé par la classe. Si une classe n'est pas en mesure d'apporter une implémentation à une méthode requise par un de ses traits, elle pourra définir cette méthode comme étant abstraite et déléguer ainsi l'implémentation de la méthode à ses classes dérivées. Devenue abstraite, cette classe sera également *complète*.

La figure 3.2 illustre l'application d'un trait à une classe. À nouveau, le langage UML est enrichi afin de représenter l'utilisation d'un trait par une classe. Cette relation est symbolisée à l'aide d'une double flèche triangulaire. Sur cet exemple, la classe « BâtimentInflammable » est construite à partir du trait « TInflammable ». Ce trait requiert l'implémentation d'une méthode « GetInflammabilité » de la part de ses classes utilisatrices afin de pouvoir accéder à une variable représentant l'inflammabilité. La classe « BâtimentInflammable » est *complète* puisqu'elle propose une implémentation pour la méthode « GetInflammabilité ».

Propriété d'aplatissement

Une propriété fondamentale des traits est la *propriété d'aplatissement* qui stipule qu'une méthode issue d'un trait a *exactement* la même sémantique que si elle avait été directement

[2]En informatique, la *compatibilité ascendante* ou *rétro-compatibilité* est la faculté pour une nouvelle technologie d'être compatible avec la technologie qu'elle supplante sans qu'aucune modification ne soit apportée à l'ancienne technologie. En l'occurrence, les traits se veulent un complément à l'orienté objet traditionnel et ne nécessitent aucune modification ou abandon des principes orientés objet de base.

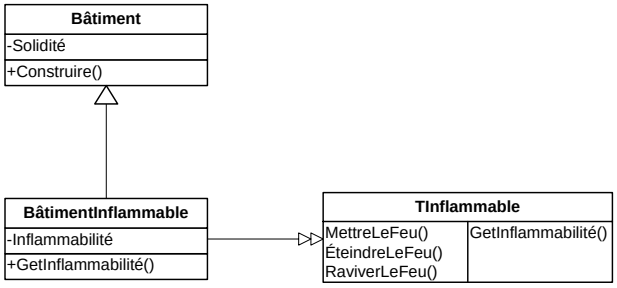

FIG. 3.2 – Exemple d'application d'un trait à une classe

implémentée dans la classe utilisatrice du trait, pour autant que cette méthode n'ait pas été redéfinie dans la classe.

Conséquence de cette propriété, bien que la notion de parent n'existe pas pour un trait, l'usage du mot-clé *super* est autorisé au sein d'une méthode d'un trait. Celui-ci fera naturellement référence à la classe parente de la classe qui *utilise* le trait.

Règles de précédence

Lorsqu'un trait est appliqué à une classe, il se peut qu'à la fois le trait et la classe partagent une méthode de même définition. Les règles de précédence suivantes sont dès lors appliquées afin d'éviter toute ambiguïté :

- Les méthodes définies dans la classe ont précédence sur les méthodes définies dans le trait. Cela permet à une classe de proposer sa propre implémentation à une méthode provenant d'un trait et dont l'implémentation ne lui sied pas.
- Les méthodes définies dans un trait ont précédence sur les méthodes héritées de la classe parente. C'est une conséquence de la propriété d'aplatissement. La méthode du trait a la même sémantique que si elle avait directement implémentée dans la classe. Elle vient donc redéfinir la méthode héritée du parent.

A contrario des mixins, les traits ne doivent pas être appliqués linéairement à une classe, l'ordre dans lequel ils sont appliqués n'a pas d'importance. En conséquence, des conflits de méthodes peuvent apparaître. Par exemple lorsqu'une classe fait usage de deux traits différents définissant une même méthode. Dans ce cas, le conflit doit être explicitement résolu à l'aide de code de colle. La résolution de conflits sera abordée plus en détail à la section 3.1.3.

Un exemple plus complet d'utilisation de trait est donné à la figure 3.3 et permet d'illustrer les règles de précédence. Sur cet exemple, un même trait, « TInflammable », est appliqué à deux classes, « BâtimentInflammable » et « NavireInflammable ». Il est important de remarquer que la méthode « ToString »[3] a été définie à plusieurs endroits. D'une part au niveau des deux classes parentes « Bâtiment » et « Navire », d'autre part au sein du trait. Alors que la classe « NavireInflammable » se satisfait de l'implémentation apportée par le trait à la méthode « ToS-

[3]Le but d'une méthode « ToString » est généralement de retourner un texte décrivant l'objet, ce qui est la cas dans notre exemple.

tring », il n'en est pas de même pour la classe « BâtimentInflammable » qui a préféré redéfinir la méthode et lui apporter une implémentation différente. Au niveau du trait, l'implémentation de la méthode « ToString » illustre l'usage du mot clé *super*. En effet, il est d'abord demandé à la classe parente de la classe utilisatrice du trait de retourner son texte décrivant l'objet pour ensuite lui accoler le texte « inflammable ».

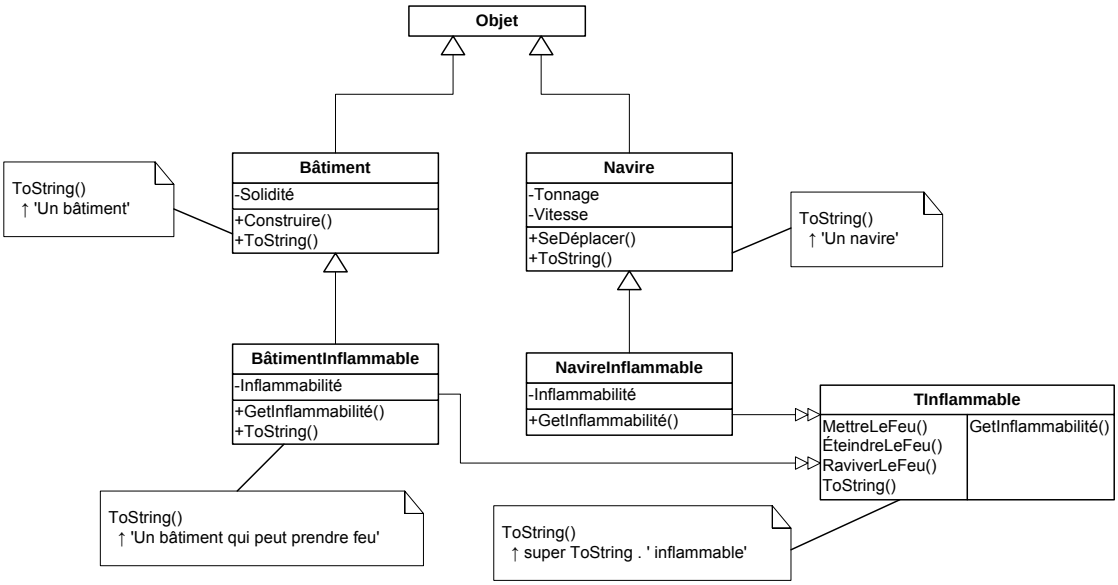

FIG. 3.3 – Exemple plus complet d'application d'un trait à des classes

En pratique, si le message « ToString » est envoyé à un objet de type « BâtimentInflammable », c'est la méthode « ToString » de la classe « BâtimentInflammable » qui va être exécutée, cette méthode ayant précédence sur la même méthode définie dans le trait. Le texte retourné sera donc « Un bâtiment qui peut prendre feu ». Par contre, si le message « ToString » est envoyé à un objet de type « NavireInflammable », c'est la méthode « ToString » du trait « TInflammable » qui va être exécutée, cette méthode ayant précédence sur la méthode définie dans la classe parente « Navire ». L'usage du mot-clé *super* dans le trait aura pour conséquence un appel à la méthode « ToString » de la classe « Navire », parente de la classe « NavireInflammable ». Le texte retourné par la classe « Navire » sera « Un navire ». À ce texte, sera accolé l'adjectif « inflammable ». Au final donc, le texte retourné sera donc « Un navire inflammable ».

3.1.2 Composition de traits composites à partir de traits

De la même manière que des classes peuvent être construites à partir de traits, des traits peuvent être construis à partir d'autres traits. On parle dans ce cas de traits composites.

La construction de traits sur base éventuelle d'autres traits sera effectuée selon l'équation suivante :

$$trait = méthodes\ fournies + méthodes\ requises + sous\text{-}traits + code\ de\ colle$$

À la différence des classes, qui doivent être *complètes*, les traits composites ne doivent pas

nécessairement fournir une implémentation pour toutes les méthodes requises des sous-traits utilisés. Les méthodes requises non satisfaites des sous-traits deviennent alors des méthodes requises du trait composite.

Tout comme les classes, les traits composites bénéficient de la propriété d'aplatissement. Une méthode définie dans un sous-trait a la même sémantique que si elle avait été implémentée directement dans le trait composite, pour autant que cette méthode n'ait pas été redéfinie dans le trait composite.

La composition de traits à partir d'autres traits partage d'autres similitudes avec la composition de classes à partir de traits. Les méthodes définies dans le trait composite ont précédence sur les méthodes acquises à partir des sous-traits et l'ordre d'application des traits n'a pas d'importance, les éventuels conflits doivent également être résolus explicitement.

La figure 3.4 illustre un exemple de trait composite, « TAchetable », qui permet de rendre un objet achetable. Ce trait composite fait usage de deux autres traits, « TTaxable » et « TRéductionnable » afin d'être secondé dans le calcul du prix final, chacun des sous-trait s'occupant d'une composante particulière du prix. Les deux traits « TTaxable » et « TRéductionnable » requièrent chacun deux méthodes de la part de leurs utilisateurs. Pour chacun des deux sous-traits, seule une des deux méthodes requises est implémentée par le trait composite, l'autre méthode étant laissée comme requise pour les utilisateurs du trait composite. Le trait « TAchetable » définissant également une méthode requise propre, la classe « Navire achetable » se retrouve donc avec trois méthodes requises à implémenter.

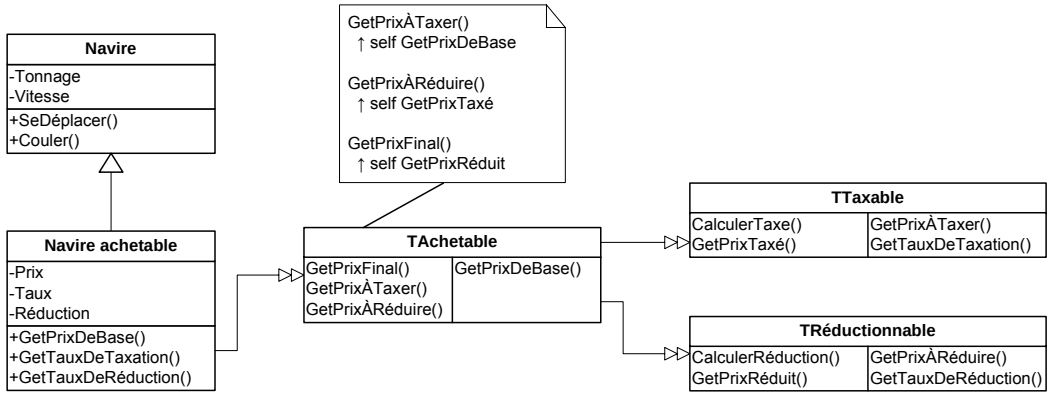

FIG. 3.4 – Exemple de trait composite

3.1.3 Résolution de conflits

Un conflit se produit lorsque l'on compose deux traits proposant des méthodes différentes portant un même nom. Idéalement, par méthodes « différentes » on entend des méthodes dont les *corps*[4] sont différents. Comparer les corps des méthodes afin de déterminer si elles sont différentes pouvant se révéler fastidieux, une autre différentiation est souvent retenue : deux méthodes de même nom seront considérées comme différentes si elles ne proviennent pas du même trait (i.e. si

[4]Le *corps* d'une méthode est l'ensemble de ses instructions.

leur implémentation ne se trouve pas dans un même trait). Avec cette approche, deux méthodes aux corps identiques, mais issues de traits différents seront donc considérées comme différentes et risquent dès lors d'entrer en conflit. Néanmoins, cette situation est plutôt rare et le conflit peut être facilement résolu.

La figure 3.5 illustre un conflit. Deux traits sont appliqués à la classe « NavireInflammableAchetable » afin de lui ajouter du comportement. Malheureusement, ces deux traits possèdent chacun une méthode « ToString » dont les corps sont différents.

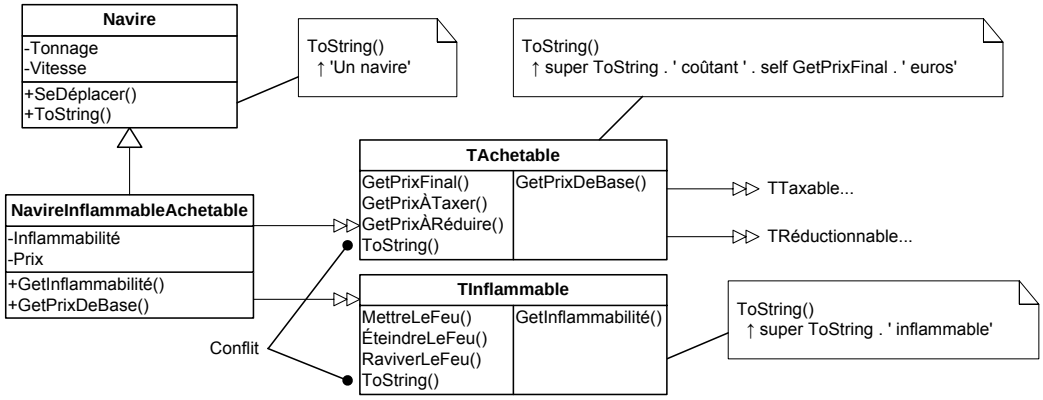

FIG. 3.5 – Exemple d'un conflit lors de la composition de traits

Lorsqu'un conflit se présente, deux solutions sont possibles afin de le résoudre :

- Ne garder qu'une seule méthode en *excluant* toutes les autres. Les clauses de composition des traits permettent en effet au programmeur d'exclure une méthode provenant d'un trait.
- S'il n'est pas possible de départager les méthodes conflictuelles pour n'en garder qu'une, on peut ajouter au niveau de la classe utilisatrice ou du trait composite une méthode de même nom que les méthodes conflictuelles afin de les supplanter. La classe utilisatrice ou le trait composite devra alors proposer sa propre implémentation. Cette méthode de résolution de conflit est basée sur les règles de précédence qui stipulent qu'une méthode définie dans une classe ou un trait composite a précédence sur les méthodes définies dans les traits utilisés.

Les figures 3.6 et 3.7 illustrent les deux méthodes de résolution de conflit en s'attaquant au conflit illustré à la figure 3.5. À la figure 3.6, le conflit est résolu en excluant une des deux méthodes « ToString », seule celle provenant du trait « TAchetable » est conservée. L'opérateur d'exclusion est représenté par le signe −. Par contre, à la figure 3.7, le conflit est résolu en redéfinissant la méthode « ToString » au niveau de la classe « NavireInflammableAchetable », les deux méthodes conflictuelles sont dès lors ignorées et inexploitées.

Surnommage

Lorsqu'un conflit est résolu, il est souvent utile de pouvoir accéder aux méthodes exclues ou redéfinies afin de, par exemple, ne pas avoir à dupliquer le code qu'elles renferment. C'est pourquoi les traits supportent une opération de *surnommage* qui permet de donner un autre nom – *en plus* du nom originel – à une méthode provenant d'un trait.

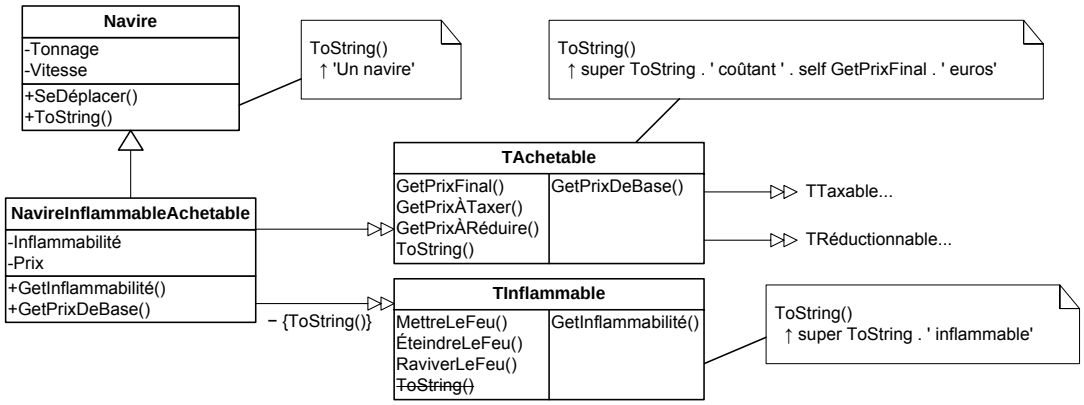

FIG. 3.6 – Exemple de résolution de conflit par exclusion

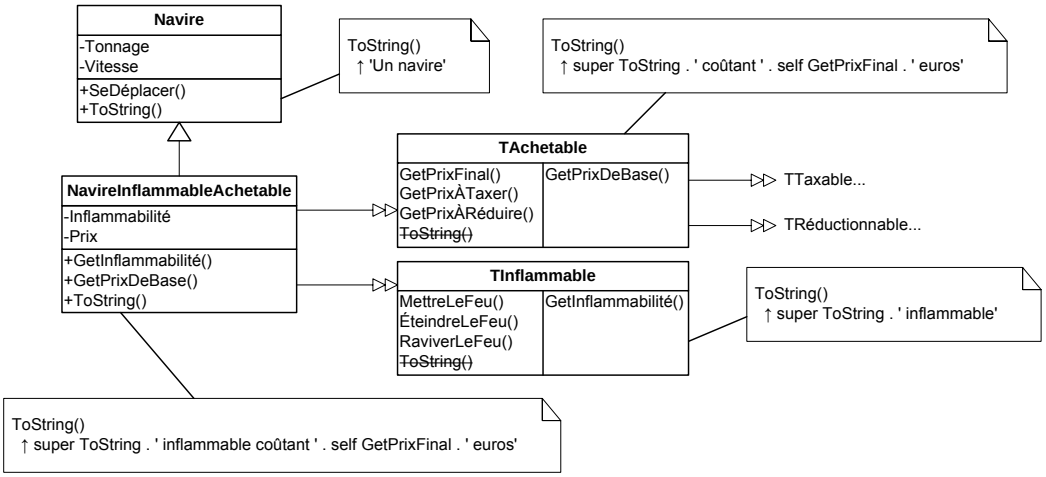

FIG. 3.7 – Exemple de résolution de conflit par redéfinition

Un exemple est donné à la figure 3.8. L'opérateur utilisé pour représenter la définition d'un surnom est l'arrobase (@). Le nom originel et le nouveau nom sont séparés par le symbole → qui doit être lu comme « est un surnom de » ou « se réfère à ». En conséquence, à gauche de la flèche se trouve le nom additionnel de la méthode définie à droite de la flèche. L'exemple suppose que les méthodes « ToString » définies dans les traits sont réutilisables, c'est pourquoi leur implémentation est légèrement différente des exemples précédents. Une fois *surnommées*, ces deux méthodes deviennent accessibles via leur nouveau nom, et ce uniquement au sein de la classe « NavireInflammableAchetable ». Les deux traits n'ont pas connaissance de ce nouveau nom. L'implémentation de la méthode « ToString » de la classe « NavireInflammableAchetable » fait alors usage des deux méthodes surnommées afin d'en récupérer le comportement.

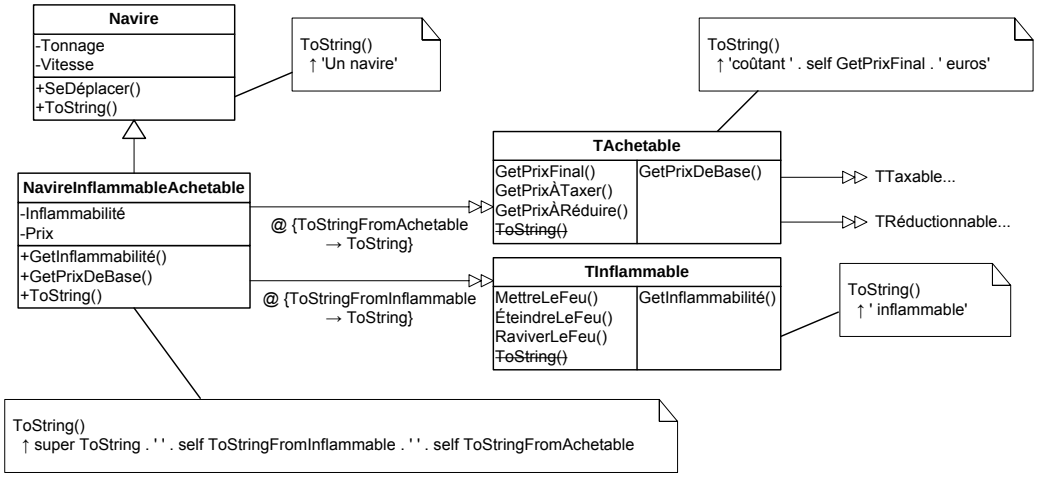

FIG. 3.8 – Exemple de surnommage de méthodes

3.1.4 Autres utilisations des opérateurs de composition

Bien que les opérateurs d'*exclusion* et de *surnommage* de méthodes ont comme fonction première d'aider à la résolution de conflits lors de la composition de plusieurs traits, ils peuvent être utilisés à d'autres fins.

L'opérateur d'exclusion permet d'exclure une méthode d'un trait, peu importe si la méthode exclue est à l'origine d'un conflit. Dès lors, cet opérateur peut être utilisé pour alléger un trait qui contiendrait « trop » de méthodes dont l'utilisateur n'a pas d'utilité.

Lors de l'exclusion d'une méthode, il se peut que celle-ci rejoigne la liste des méthodes « requises » par le trait. En effet, il n'est pas impossible que la méthode exclue soit utilisée par une autre méthode du trait, elle est donc requise. Charge alors à l'utilisateur du trait de, soit réhabiliter la méthode exclue en ne l'excluant plus, soit proposer une implémentation propre en remplacement de la méthode exclue.

La possibilité d'exclure une méthode qui n'est pas à l'origine d'un conflit pourra poser certains problèmes si on désire considérer un trait comme étant un *type*. Cette question sera évoquée par la suite, à la section 3.3.2.

Qu'une méthode soit conflictuelle ou pas, le surnommage permet de la rendre accessible à partir d'un nom additionnel. Cet opérateur pourra donc être utilisé à plusieurs fins :

- Rendre accessible une méthode conflictuelle qui a été écartée.
- Passer outre une redéfinition lorsqu'une classe possède une méthode portant le même nom qu'une méthode d'un de ses traits. Si la redéfinition n'est pas voulue, il suffira d'utiliser le surnommage afin de rendre la méthode redéfinie accessible sous un autre nom.
- Rencontrer les desiderata d'un autre trait. Imaginons une classe utilisant deux traits dont l'un requiert une méthode m. Il se peut que l'autre trait puisse répondre à ce besoin à l'aide d'une de ses méthodes. Si celle-ci ne porte pas le nom requis, en l'occurrence m, l'opérateur de surnommage pourra être utilisé afin de lui donner le nom *ad hoc*.

Il est important de noter que *surnommage* et *renommage* sont deux opérations bien distinctes et que seul le surnommage est proposé par les traits. Le surnommage permet de donner un nouveau nom à une méthode, sans affecter l'ancien nom, tandis que le renommage *invalide* l'ancien nom. Par conséquent, lorsqu'une méthode est renommée, le changement de nom doit être répercuté partout où l'ancien nom est utilisé. En particulier, tous les appels vers la méthode renommée doivent être modifiés.

Cette différence fondamentale est due à la propriété d'aplatissement qui stipule qu'une méthode issue d'un trait a exactement la même sémantique que si elle avait été directement implémentée dans la classe utilisatrice. En l'occurrence, le corps d'une méthode issue d'un trait ne peut différer selon la classe sur laquelle le trait est appliqué. Or, si renommer une méthode était possible, les corps des méthodes (qui peuvent contenir des appels vers une méthode renommée) seraient différents selon les renommages effectués par les classes utilisatrices, ce qui contredit la propriété d'aplatissement.

Autre conséquence de la propriété d'aplatissement, lorsqu'une méthode est surnommée, l'ancien et le nouveau nom de la méthode font tous deux référence exactement au *même* corps, celui-ci n'est pas modifié par l'opération de surnommage. Par conséquent, si le corps de la méthode surnommée effectue un appel récursif[5], c'est l'ancien nom de la méthode qui sera appelé et non le nouveau. Ce qui peut poser un certain nombre de problèmes, puisqu'un surnommage est souvent effectué pour rendre accessible une méthode qui ne l'est plus avec son nom d'origine.

De même, le surnommage ne permet *pas* de résoudre les conflits de méthodes dont la sémantique est différente, i.e. des méthodes dont le nom est identique mais qui remplissent des fonctions différentes qu'il est impossible de synthétiser. Un exemple est donné à la figure 3.9. Deux traits, « TAchetable » et « TVendable », sont appliqués à la classe « NavireAchetableVendable » afin de rendre un navire à la fois achetable et vendable. Malheureusement, la méthode « CalculerPrix » pose conflit car présente dans les deux traits. Cette méthode effectue des calculs complètement différents suivant qu'il s'agit de calculer le prix d'achat ou de vente.

Pour tenter de résoudre ce conflit, plusieurs solutions sont possibles :

- Soit on évince l'une des deux méthodes pour n'en retenir qu'une. Par exemple, on conserve uniquement la méthode « CalculerPrix » du trait « TAchetable », mais, dans ce cas, la

[5] On parle d'appel récursif lorsqu'une méthode s'appelle elle-même.

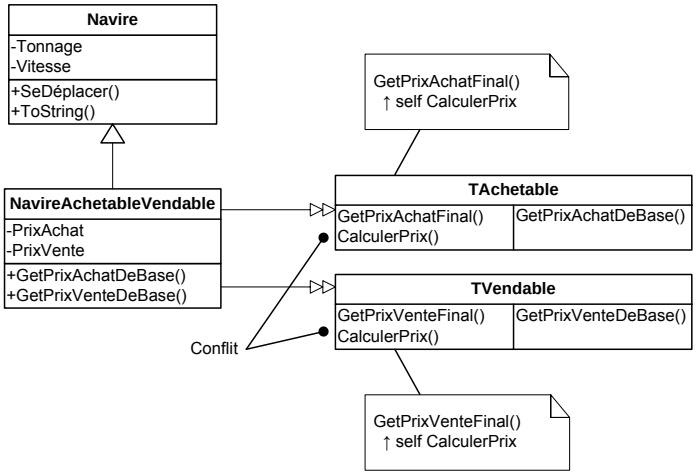

FIG. 3.9 – Exemple de conflit *sémantique*

méthode « GetPrixVenteFinal » du trait « TVendable » fera appel à une version inadaptée de la méthode « CalculerPrix ».

– Soit on redéfini la méthode « CalculerPrix » dans la classe « NavireAchetableVendable », mais il est malheureusement impossible d'offrir une implémentation qui satisfasse les deux traits.
– Soit on évince l'une des deux méthodes pour n'en retenir qu'une *et* on surnomme la méthode évincée avec un nouveau nom. Par exemple, on conserve uniquement la méthode « CalculerPrix » du trait « TAchetable » et on surnomme la méthode « CalculerPrix » du trait « TVendable » avec le surnom « CalculerPrixVente ». Malheureusement, puisqu'il s'agit d'un surnom et non d'un renommage, le corps de la méthode « GetPrixVenteFinal » fera toujours appel à la méthode « CalculerPrix » dont l'unique version restante est inadaptée.

Cette exemple illustre bien un conflit sémantique qu'il est impossible de résoudre. Cette limitation peut être vue comme assez contraignante, des conflits sémantiques n'étant pas rares. Le programmeur veillera donc à bien nommer ses méthodes.

En résumé, seule la classe à l'origine du surnommage peut utiliser le nouveau nom. Les traits qui composent la classe n'ont, eux, pas connaissance de ce surnommage.

3.1.5 Évaluation

Au chapitre précédent, une série de problèmes relatifs aux méthodes existantes de réutilisation de code ont été mis en évidence. Dans cette section, nous allons analyser et évaluer la réponse apportée par les traits à ces problèmes, car, si les traits ont été imaginés, c'est avant tout pour tenter de les pallier.

Réutilisation multiple

L'héritage simple, comme son nom l'indique, trouve sa principale limitation dans l'impossibilité de réutiliser du code provenant de plusieurs sources. Les traits, dont la composition multiple est permise, lèvent cette limitation.

Le problème du diamant

Un des problèmes majeurs posé par l'héritage multiple est le problème du diamant qui survient lorsqu'un même attribut ou une même méthode est héritée plusieurs fois par le biais de chemins différents. Ce même problème peut également survenir avec les traits. Non pas au niveau des attributs, puisque les traits sans état ne possèdent pas d'attribut, mais bien au niveau des méthodes.

Un exemple est illustré à la figure 3.10. La classe « A » utilise deux traits, « TB » et « TC », lesquels, à leur tour, utilisent chacun un même trait « TD ». La méthode « Foo », du trait « TD », est donc acquise deux fois par la classe « A » via deux chemins différents, d'une part via le trait « TB » ; d'autre part via le trait « TC ». Y a-t-il pour autant un conflit ? Non, car il s'agit d'une *même* méthode acquise à partir d'un *même* trait d'origine. Lors du processus d'aplatissement, cette même méthode fera partie intégrante des traits « TB » et « TC », peu importe ensuite quelle version sera utilisée pour être incluse dans la classe « A ».

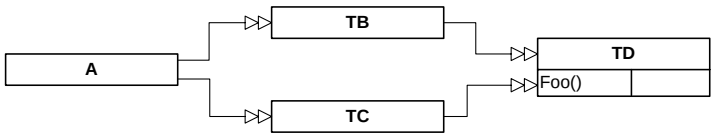

Fig. 3.10 – Exemple d'architecture en diamant

Les auteurs à l'origine des traits [SDNB03] soutiennent même qu'il serait *risqué* de considérer cette situation comme un conflit et de demander au programmeur de la classe « A » de le résoudre en choisissant l'une ou l'autre méthode « Foo ». En effet, plus tard, un des deux traits « TB » ou « TC » pourrait se voir implémenter sa propre version de la méthode « Foo », différente de celle acquise par le trait « TD ». Dans ce cas, un nouveau conflit apparaît, réel cette fois, car deux versions différentes de la méthode « Foo » sont acquises par la classe « A ». Malheureusement, ce conflit ne sera plus signalé au programmeur, celui-ci ayant déjà été résolu précédemment.

À la section 2.4.2, nous avons vu qu'une des difficultés majeures d'une architecture en diamant était l'héritage multiple d'un attribut de même origine. En effet, la question se pose alors de savoir si les multiples instances de cet attribut héritées doivent être fusionnées ou coexister parallèlement. L'exemple illustré à la figure 2.14 nous montrait que la réponse dépendait de chaque attribut : dans certains cas une fusion était appropriée, dans d'autres pas.

Bien que les traits sans état ne possèdent pas d'attribut, ceux-ci sont néanmoins simulés à l'aide d'*accesseurs requis* (i.e. des méthodes permettant d'accéder à des attributs et qui devront être implémentées par la classe utilisatrice du trait). Dès lors, revisitons l'exemple présenté à

la figure 2.14 afin d'opérer la même construction, à l'aide de traits cette fois-ci. Le résultat est présenté à la figure 3.11.

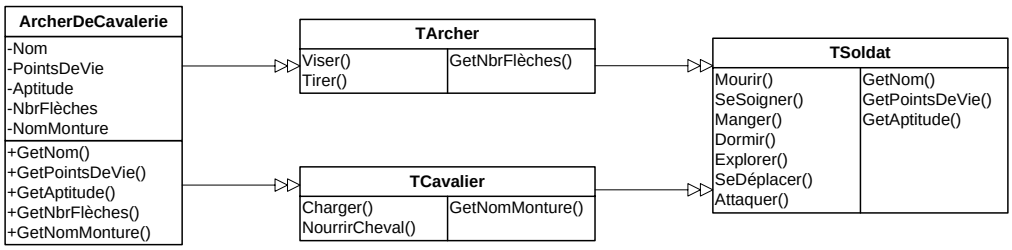

FIG. 3.11 – L'exemple de la figure 2.14 construit à l'aide de traits

Le problème cette fois-ci est différent. Le trait « TSoldat » définit une méthode requise « GetAptitude » afin de représenter l'attribut « Aptitude ». Les deux traits « TArcher » et « TCavalier », utilisateurs du trait « TSoldat », ne peuvent offrir d'implémentation pour cette méthode requise. Ils transmettent donc cette responsabilité à leurs utilisateurs. En conséquence, la classe « ArcherDeCavalerie » se voit imposer *deux fois* l'implémentation de la méthode « GetAptitude ». Malheureusement, elle ne peut pas offrir une implémentation différente pour le trait « TArcher » et pour le trait « TCavalier », les deux traits seront donc complétés à l'aide de la *même* méthode et feront dès lors référence au *même* attribut. En conséquence, il n'est pas possible de faire coexister deux instances différentes de l'attribut « Aptitude », les traits sans état imposant la fusion.

Accès aux méthodes conflictuelles et redéfinies

Héritage multiple et traits partagent tous deux une difficulté commune : des conflits peuvent facilement apparaître. Afin de les résoudre, des approches différentes ont été retenues par chacun.

L'héritage multiple impose que les méthodes ou attributs conflictuels soient qualifiés sans ambiguïté, au besoin en rajoutant le nom du parent d'où provient la méthode ou l'attribut d'origine ambiguë. Cette approche a pour effet malheureux de fortement renforcer le couplage entre classes parentes et enfants. En effet, lorsqu'une classe parente est renommée, le changement doit être répercuté partout où une méthode ou un attribut a été qualifié à l'aide du nom de cette classe parente renommée.

Les traits, quand à eux, préconisent la définition de surnoms pour accéder aux méthodes conflictuelles. Le surnom n'est défini qu'une seule fois, *en dehors* de l'implémentation des méthodes, et lui seul est ensuite utilisé afin d'accéder aux méthodes conflictuelles. Par conséquent, lors du renommage d'un trait, seule la définition du surnom doit être modifiée, les méthodes utilisant ce surnom ne sont pas impactées.

Les traits se voulant être une extension de l'héritage simple, lorsqu'une classe désire communiquer avec sa classe parente, elle ne doit pas explicitement la nommer et peut utiliser le mot-clé générique *super*. Conséquence de cette généricité, le mot-clé *super* peut être utilisé au sein d'une méthode d'un trait, il fera simplement référence au parent de la classe *utilisatrice* du trait.

Redéfinitions involontaires

Les mixins, bien qu'avantageux, connaissent également leur lot de problèmes et limitations, dont la présentation a été faite à la section 2.4.3. Parmi ceux-ci, la redéfinition parfois involontaire par le mixin de méthodes de la classe parente sur laquelle il est appliqué.

Les traits rencontrent également ce problème. En effet, lorsqu'une classe utilise un trait, les méthodes héritées du parent peuvent se voir redéfinir par les méthodes provenant du trait. C'est une conséquence des règles de précédence. Néanmoins, à la différence des mixins, qui nécessitaient l'insertion d'une classe intermédiaire dans la hiérarchie, le contournement de cette redéfinition involontaire pourra, avec les traits, se faire de manière beaucoup plus élégante, en utilisant un minimum de code de colle. En effet, il suffit, lors de l'application du trait, d'exclure la méthode à l'origine de la redéfinition et, au besoin, lui donner un surnom si un accès à cette méthode est nécessaire.

Existence d'un ordre total

Les mixins étant appliqués linéairement, il n'est pas toujours possible de trouver un ordre d'application satisfaisant (cf. section 2.4.3). Ce problème n'est plus d'actualité avec les traits, puisque l'ordre dans lequel sont appliqués plusieurs traits n'a aucune importance. Le sort de chacun des conflits est réglé individuellement, méthode par méthode.

Dispersion du code de colle

La résolution des conflits induits par l'application de mixins implique une grande dispersion de code de colle dans la hiérarchie de classe, au détriment de sa clarté (cf. section 2.4.3). Les traits ne connaissent pas ce problème. En effet, lorsque plusieurs traits sont combinés, le code de colle est entièrement centralisé au niveau de l'entité combinant les traits. Et ce, dans l'optique qu'une entité combinant plusieurs traits possède le contrôle total sur la manière dont ils doivent être combinés. Par conséquent, la hiérarchie de classe n'est plus polluée par une dispersion du code de colle, au profit d'une meilleure compréhension.

Fragilisation de la hiérarchie

Une préoccupation importante, lorsqu'il s'agit d'évaluer une méthode de réutilisation de code, est sa résistance par rapport à divers changements effectués ultérieurement. L'usage de mixins, par exemple, mène à une fragilisation de la hiérarchie : des changements, en apparence anodins, peuvent être lourds de conséquences et impacter, par exemple, tous les clients directs et indirects bénéficiaires des fonctionnalités apportées par le mixin (cf. section 2.4.3).

Les traits permettent, eux, de limiter généralement l'impact d'un changement au client direct du trait. Par exemple :

- Si une méthode ajoutée pose un nouveau conflit, le client direct pourra généralement résoudre le conflit sans impacter les autres clients.
- Si une méthode est retirée d'un trait, le client direct pourra pallier ce nouveau manquement en proposant sa propre implémentation, sans que les autres clients ne soient importunés.
- Si une méthode d'un trait est renommée, le client direct pourra utiliser le mécanisme de surnommage afin que l'ancien nom soit conservé et reste utilisable par les autres clients indirects.
- Si une nouvelle méthode est requise par un trait, seul le client direct devra apporter une réponse à ce nouveau requis.

Ces exemples montrent donc que les effets en cascade, contrairement aux mixins, peuvent être, dans de nombreux cas, facilement évités. On reconnaîtra donc aux traits leur relative robustesse.

Les classes comme unités de réutilisation

Au chapitre précédent a été mis en évidence le paradoxe entre les deux rôles attribués aux classes par les diverses méthodes d'héritage. En effet, en plus de laisser aux classes leur responsabilité première de génératrices d'instances, les méthodes d'héritage leur attribuent un rôle supplémentaire, celui d'unité de réutilisation de code. Ces deux rôles sont antinomiques, le premier imposant aux classes d'être les plus complètes possible, l'autre préconisant une concision maximale.

Les traits permettent de décharger les classes de leur second rôle d'unité de réutilisation de code, leur laissant ainsi comme unique responsabilité la génération d'objets. De plus, contrairement aux mixins, les traits sont totalement indépendants de la hiérarchie d'héritage. Dès lors, ils ne sont pas « pollués » par une éventuelle série de méthodes acquises d'un quelconque parent et peuvent pleinement jouer leur rôle de paquetage de méthodes réutilisables afin de maximiser de la réutilisabilité.

Lorsqu'une classe est composée à partir de traits, il n'est pas toujours évident pour le programmeur d'avoir une vue claire et complète de la résultante de la composition, les méthodes intégrées à la classe pouvant être dispersées dans de nombreux traits. C'est pourquoi, il est conseillé aux outils de programmation de proposer deux vues :

- Une vue *aplatie* permettant de voir la résultante d'une composition de traits, suite au processus d'applatissement ;
- Une vue *structurelle* permettant de mettre en évidence les relations de composition et décomposition entre les classes et traits.

3.2 Les traits à états

Dans leur forme originelle, les traits sont sans état, i.e. ils ne peuvent contenir d'attributs et ont comme seule vocation de regrouper un ensemble de méthodes. Dès lors, lorsqu'un trait sans état désire faire usage d'attributs, il doit recourir à la définition d'*accesseurs requis*, lesquels,

une fois implémentés par la classe utilisatrice du trait, lui permettront d'accéder à des attributs définis au sein de la classe. Par définition, les traits sans état sont *incomplets*, une partie de leur comportement étant déléguée à ses utilisateurs qui devront déclarer eux-mêmes les attributs et accesseurs manquants.

Afin de pallier cette carence, une évolution naturelle des traits a été proposée par les auteurs à l'origine des traits : les *traits à états* [BDNW07]. Les traits à états autorisent la définition et l'usage d'attributs au sein d'un trait. Un nouvel opérateur est également introduit afin de donner au client un certain contrôle sur ces attributs.

Dans cette section, nous étudierons tout d'abord les limitations dues à l'absence d'état dans les traits. Nous introduirons ensuite les traits à états et présenterons leur nouvel opérateur de composition. Nous évaluerons ensuite les traits à états. Premièrement, nous analyserons dans quelle mesure ils permettent de lever les limitations dues à l'absence d'état dans les traits. Nous discuterons ensuite du respect par les traits à états de la propriété d'aplatissement. Nous finirons cette évaluation par revisiter le problème du diamant en l'adaptant aux traits à états.

3.2.1 Limitations des traits sans état

L'absence d'état au sein des traits pose une série de limitations que les traits à états s'efforceront de lever :

- L'ensemble des méthodes requises d'un trait est pollué par une série d'accesseurs requis, dont l'implémentation triviale et redondante décourage et limite l'utilisation de traits.
- Les clients se voient forcés d'ajouter du code de colle identique d'un client à l'autre.
- L'introduction de nouveaux attributs dans le trait impose une modification au niveau des clients.
- L'usage de méthodes requises pour définir des attributs au sein d'un trait brise les principes d'encapsulation et de boîte noire : les attributs internes sont accessibles à la fois par le client du trait, mais également depuis l'extérieur du client.

Réutilisation limitée

Dans une grand majorité des cas, les traits sans état imposent à leurs clients l'implémentation d'une série d'accesseurs. Ceux-ci ne sont pas « prêt à porter » : à chaque utilisation d'un trait des retouches sont nécessaires, limitant ainsi la réutilisabilité. De plus, d'implémentation triviale et redondante, ces accesseurs viennent noyer l'ensemble des méthodes requises d'un trait alors que l'attention devrait plutôt être attirée sur les méthodes requises non-triviales dont la responsabilité échoit réellement à la classe.

Redondance du code de colle

Lorsqu'un trait est utilisé par plusieurs classes clientes, un même code de colle est ajouté à chacune de ces classes. Conséquence, une duplication de code inutile que, paradoxalement, les

traits avaient pour vocation d'éviter. Toute duplication de code étant à éviter, il serait également intéressant de remédier à cette duplication inutile.

L'obligation pour les classes de fournir les attributs nécessaires aux traits qu'elles utilisent entraîne également un autre effet de bord indésirable : l'émergence de classes composées uniquement de ce code de colle trivial. Ces classes pouvant même se révéler nombreuses[6], il serait avantageux d'essayer de les éviter.

Introduction de nouveaux attributs

Lorsqu'un trait désire changer son implémentation et utiliser de nouveaux attributs, même lorsque ceux-ci sont internes (en d'autres mots, qu'ils n'ajoutent pas de comportement), des accesseurs requis supplémentaires apparaissent. Cette modification est lourde de conséquences car *toutes* les classes qui utilisent ce trait sont impactées ; chacune doit fournir une implémentation pour le nouvel accesseur requis.

Violation de l'encapsulation

Les traits sans état violent les principes d'encapsulation et de boîte noire de deux manières. Premièrement, en déléguant à leurs clients la gestion des attributs, les traits sans état exposent inutilement des informations relatives à leur représentation interne. Les traits sans état ne peuvent en effet pas utiliser des attributs qui leur sont internes sans impliquer le client qui doit fournir les variables d'instances et accesseurs nécessaires. Il serait donc opportun que les attributs, qui ne présentent aucun intérêt pour le client, soient gérés en interne, par le trait, sans que le client n'en ait connaissance.

La seconde violation concerne la visibilité des accesseurs. Les méthodes acquises par un client à partir d'un trait viennent enrichir l'interface du client, en d'autres mots, elles sont publiques et accessibles de l'extérieur par d'autres objets. Dès lors, lorsqu'un trait désire faire usage d'un attribut qui lui est interne, non seulement cet attribut est visible et accessible par le client, mais il l'est également depuis l'extérieur du client.

Certes certains langages permettent de restreindre la visibilité d'une méthode à l'aide d'opérateurs. C'est le cas par exemple de Java et C#, avec lesquels il serait possible de rendre *privées* certaines méthodes requises (dont les accesseurs). Néanmoins, cette approche dépend fortement du langage de programmation utilisé. Une approche plus générique et transversale à tous les langages serait donc bienvenue.

[6]Lors d'un exercice de refactorisation d'un ensemble de classes à l'aide de traits effectué par les auteurs des traits, il a été constaté que 24 % des classes ne contenaient d'autre code que le code de colle nécessaire pour pallier le manque d'attributs [BDNW07].

3.2.2 Définition des traits à états

Les traits à états *étendent* les traits sans état en ajoutant la possibilité de définir des attributs au sein du trait. L'équation définissant les traits devient donc :

$$trait = méthodes\ fournies + méthodes\ requises + attributs + sous\text{-}traits + code\ de\ colle$$

Un exemple de trait à états est donné à la figure 3.12 et revisite le premier exemple de trait sans état donné à la figure 3.1. L'attribut « Inflammabilité » est désormais intégré au trait, un accesseur requis n'est plus nécessaire. Afin que la liste des méthodes requises ne soit pas vide, une nouvelle méthode a été ajoutée au trait : « AppelerPompiers ». Cette méthode est requise car on l'imagine sous l'entière responsabilité des clients du trait. On suppose que seuls ceux-ci sont capables, en fonction de leur contexte, de connaître l'emplacement des pompiers les plus proches.

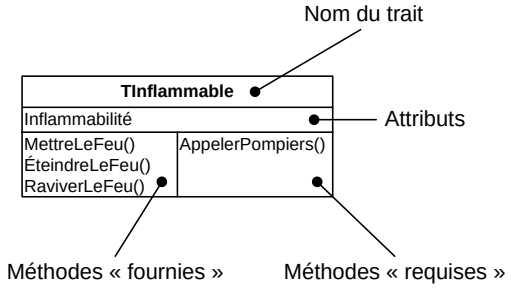

FIG. 3.12 – Un premier exemple de trait à états

Visibilité des attributs

Par défaut, les attributs sont *privés* au trait qui les définit. Dès lors, lorsque deux traits définissant un même attribut sont composés, aucun conflit n'est généré. Un exemple est donné à la figure 3.13. Les deux traits « TVendable » et « TAchetable » définissent tous deux un même attribut « PrixDeBase ». Ces deux attributs n'occasionnent pas de conflit, chacun étant visible uniquement par leur trait d'origine.

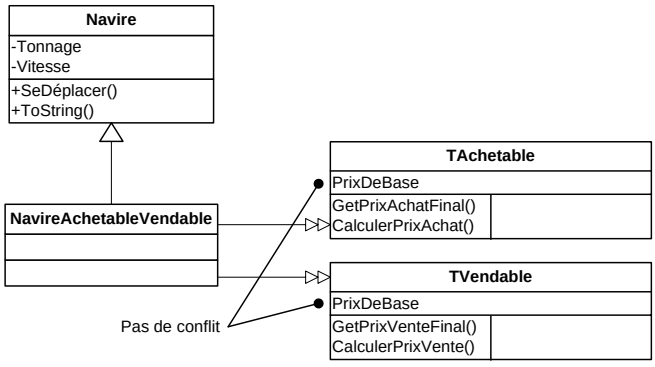

FIG. 3.13 – Exemple de composition de traits n'occasionnant pas de conflit

Si sémantiquement les attributs sont définis comme privés, lors d'un éventuel processus d'aplatissement, la présence d'un attribut de même nom dans plusieurs traits peut poser problème. En effet, l'aplatissement consiste à « recopier » l'ensemble des méthodes et attributs des traits au sein du client. Dès lors, deux variables[7] d'instance de même nom seront définies dans le client aplati. Afin de respecter le caractère privé des variables d'un trait, lors de l'aplatissement, on veillera à les *renommer* à l'aide d'un nom dont on est sûr qu'il n'occasionnera pas de conflits. Par exemple en préfixant le nom de chaque variable avec le nom du trait. Le choix du nouveau nom est laissé entièrement à la discrétion du processeur d'applatissement. Celui-ci peut en effet organiser comme il le souhaite tout élément privé d'un trait, ces derniers n'étant pas accessibles au client. Un exemple d'aplatissement est donné à la figure 3.14. Tous les corps de toutes les méthodes ne sont pas repris sur le schéma, seule l'implémentation de la méthode « CalculerPrixAchat » est présentée.

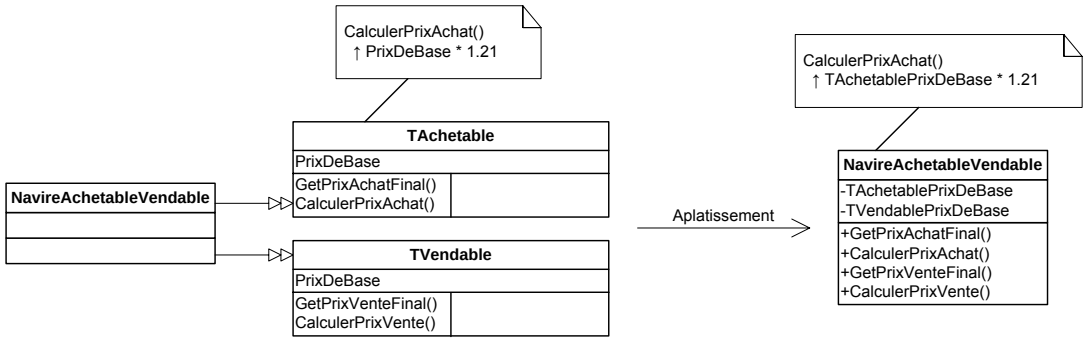

FIG. 3.14 – Exemple d'aplatissement d'une classe utilisant des traits à états

Accès aux attributs

Afin qu'un client puisse accéder aux variables définies au sein d'un trait, un nouvel opérateur d'*accès* a été défini (@@). Cet opérateur permet à un client d'accéder à une variable d'un trait en lui donnant un éventuel nouveau nom. Il est important de noter que l'ancien nom privé reste d'usage au sein du trait et que le nom utilisé pour l'accès n'est disponible que pour le client.

Un exemple d'accès aux attributs est donné à la figure 3.15. Tout comme avec le surnommage de méthode, le nom originel et le nom d'accès à l'attribut sont séparés par le symbole → qui doit être lu comme « permet d'accéder à ». En conséquence, à gauche de la flèche se trouve le nom permettant l'accès à l'attribut défini à droite de la flèche. Dans cet exemple, d'une part l'attribut « PrixDeBase » du trait « TAchetable » est rendu accessible par la classe via le nom « PrixAchatDeBase », d'autre part l'attribut « PrixDeBase » du trait « TVendable » est rendu accessible via le nom « PrixVenteDeBase ». Les traits continuent à utiliser les anciens noms privés dans leurs méthodes, tandis que la classe utilise les nouveaux noms.

Lors du processus d'aplatissement, il faudra s'assurer que les deux noms fassent référence à la même variable. Pour y arriver, une solution est d'éventuellement remplacer le nom privé par le nom d'accès. Cette opération n'est cependant pas toujours possible, le nom d'accès attribué par le client étant peut-être déjà utilisé pour nommer des variables locales dans des méthodes du

[7] Variables et attributs sont synonymes.

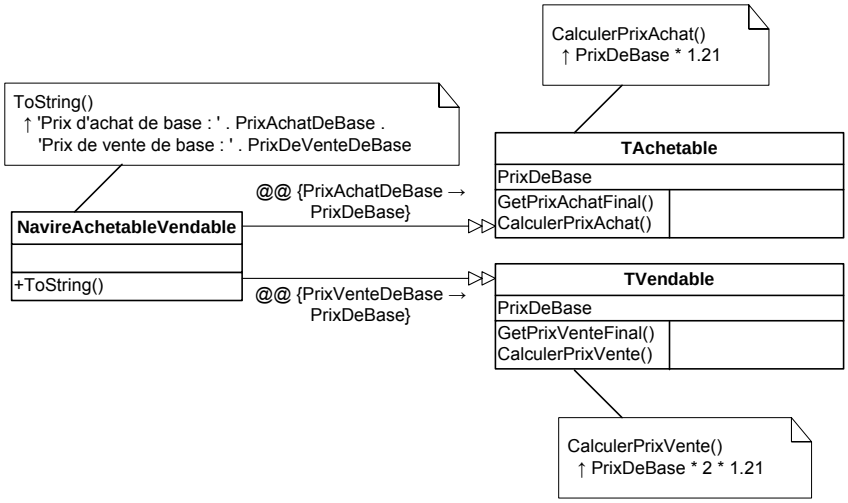

FIG. 3.15 – Exemple d'accès aux attributs d'un trait

trait. Dès lors, si on remplace dans le trait le nom privé de l'attribut par son nom d'accès, celui-ci pourra entrer en conflit avec certaines variables locales. Une solution pourrait être alors de renommer ces variables locales. Un exemple d'un tel aplatissement est illustré à la figure 3.16 et correspond à l'aplatissement de l'exemple précédent présenté à la figure 3.15. L'approche retenue a été de renommer les attributs privés en utilisant le nouveau nom d'accès fourni par la classe cliente.

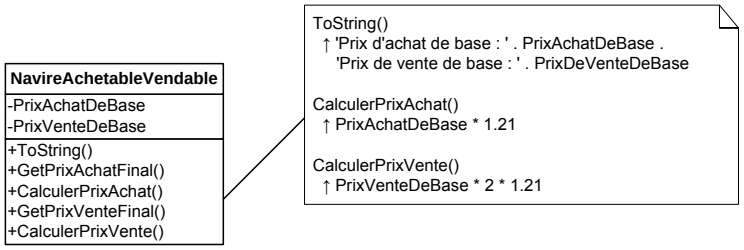

FIG. 3.16 – Exemple d'aplatissement d'une classe accédant aux attributs de ses traits

Fusion des attributs

Plusieurs attributs provenant de traits différents peuvent être *fusionnés* lors de la composition de traits afin qu'ils puissent partager la même valeur. Il suffit pour ce faire de rendre ces attributs disponibles au client via un *même* nom d'accès.

Un exemple de fusion d'attributs est donné à la figure 3.17. Les exemples précédents supposaient différents les deux prix de base (d'achat et de vente) acquis à partir des deux traits. Dans ce nouvel exemple, on présume qu'un même prix de base est utilisé à la fois pour le calcul des prix d'achat et de vente finaux. Une fusion des deux attributs « PrixDeBase » de chacun des traits est alors opérée en leur conférant un même nom d'accès. Il est à remarquer que le nom d'accès est identique aux noms privés, ce qui est tout à fait autorisé.

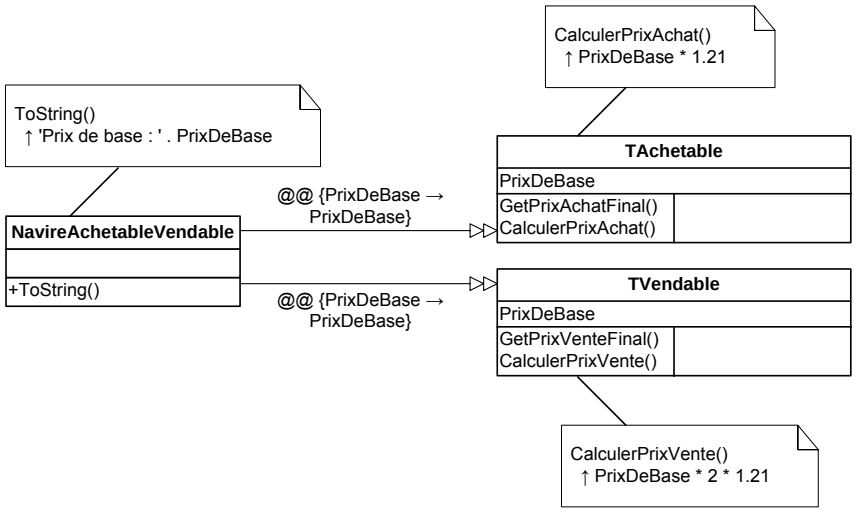

FIG. 3.17 – Exemple de fusion d'attributs

Un aplatissement possible de cet exemple est donné à la figure 3.18. Aucun renommage des attributs privés n'a été nécessaire, puisque le nom d'accès fourni par la classe cliente est identique au nom privé.

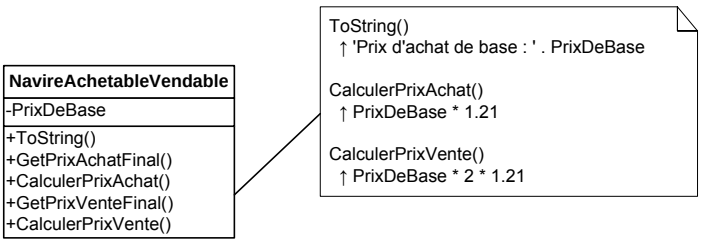

FIG. 3.18 – Exemple d'aplatissement d'une classe ayant fusionné des attributs de ses traits

3.2.3 Conflits d'attributs

L'approche retenue par les concepteurs des traits à états empêche toute apparition involontaire de conflits. D'une part, les attributs étant par défaut privés, lorsque plusieurs traits définissant un même attribut sont composés, aucun conflit n'est généré : chaque trait utilise sa propre instance privée de l'attribut. D'autre part, le programmeur possède un contrôle total sur les attributs qu'il désire rendre accessibles. Il peut à la fois choisir quels attributs doivent être rendus accessibles, mais également choisir les noms utilisés pour accéder à ces attributs.

En conséquence, lorsque l'implémentation d'un trait est modifiée et qu'un nouvel attribut lui est ajouté, aucun conflit ne sera occasionné, celui-ci restant privé par défaut.

Que se passe-t-il si le programmeur définit au sein du client un attribut portant le même nom qu'un des noms d'accès ? Imaginons qu'à l'exemple illustré à la figure 3.17 un attribut portant le nom « PrixDeBase » ait été rajouté à la classe « NavireAchetableVendable ». Un conflit de nom apparaît, certes. Néanmoins, il s'agit là d'une *erreur* de la part du programmeur, ce dernier

ayant le contrôle total à la fois sur les noms d'accès et sur les noms donnés aux attributs du client. C'est donc au programmeur de fournir des noms différents pour éviter ce genre de conflits.

3.2.4 Levée des limitations

Tels qu'introduits, les traits à états permettent de lever les limitations présentées à la section 3.2.1. En particulier :

- L'ensemble des méthodes requises d'un trait n'est plus pollué par une série d'accesseurs requis, leur implémentation étant possible directement au sein du trait.
- Les clients d'un trait ne doivent plus implémenter du code de colle redondant d'un client à l'autre.
- L'introduction de nouveaux attributs au sein d'un trait n'impose pas à ses clients de modifications, les attributs ajoutés étant par défaut privés au trait.
- Des accesseurs publics ne doivent pas être ajoutés aux classes clientes laissant ainsi la possibilité aux attributs de rester privés selon les principes d'encapsulation et de boîte noire.

3.2.5 Propriété d'aplatissement

Dans leur forme originelle les traits sans état respectent la propriété d'aplatissement qui stipule qu'une méthode issue d'un trait a exactement la même sémantique que si elle avait été directement implémentée dans la classe utilisatrice du trait, pour autant que cette méthode n'ait pas été redéfinie dans la classe. Cela implique qu'une classe utilisant des traits peut être convertie en une classe n'utilisant pas de trait et dans laquelle ont été recopiées toutes les méthodes des traits utilisés.

Cette propriété est encore vraie pour les traits à états [BDNW07], bien que l'aplatissement ne se résume plus à une simple copie conforme dans le client des méthodes et attributs du trait. Les attributs dont l'accès n'a pas été demandé par le client, se voient en effet renommés afin de conserver leur caractère privé et d'éviter les éventuels conflits. Néanmoins, ces attributs étant privés, peu importe le nom qui leur est donné lors de l'aplatissement, le client n'en sera nullement impacté puisqu'il ne fait pas usage de ces attributs. En conséquence ce renommage interne n'altère pas la propriété d'aplatissement.

3.2.6 Le problème du diamant revisité

Lors des chapitres précédents, nous avons vu qu'une architecture en diamant n'était pas sans poser une série de problèmes lors de l'utilisation non seulement de l'héritage multiple mais également des traits sans état. Cette architecture un peu particulière est excellente pour évaluer les capacités d'une méthode de réutilisation de code à résoudre les conflits et à appréhender des architectures plus complexes. Dès lors, analysons cette architecture dans un contexte de traits à états.

Pour ce faire, revisitons l'exemple largement repris lors des chapitres précédents dont l'objectif était la construction d'une classe « ArcherDeCavalerie » suivant une architecture en diamant. La figure 3.19 illustre l'usage de traits à états pour réaliser cette même construction. L'exemple a été réduit à sa plus simple expression : seul l'attribut « Aptitude » et son accesseur sont représentés afin d'offrir une clarté maximale.

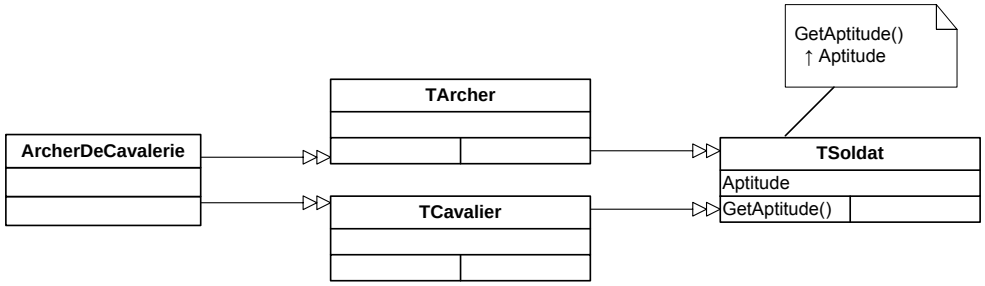

FIG. 3.19 – Construction de la classe « ArcherDeCavalerie » à l'aide de traits à états

Acquisition multiple d'un attribut de même origine

Nous avons vu au chapitre 2.4.2 que lorsqu'un attribut de même origine est acquis plusieurs fois par le biais de chemins différents, le sort à réserver aux différentes instances acquises dépend du sens donné à l'attribut. Certains attributs appellent plutôt à une fusion des instances, d'autres plutôt au maintien des différentes instances. Ces nuances ne sont pas toujours exprimables. C'est le cas, par exemple, des traits sans état qui ne permettent pas la cohabitation de plusieurs instances, la fusion étant imposée. L'ajout d'états aux traits change-t-il ce comportement ? Pour répondre à cette question, analysons l'exemple de la figure 3.19.

Les deux traits « TArcher » et « TCavalier » n'ayant pas manifesté leur volonté d'accéder à l'attribut « Aptitude » en usant de l'opérateur d'accès, cet attribut reste privé au trait « TSoldat ». Par conséquent, lors de l'aplatissement des traits « TArcher » et « TCavalier », un nouveau nom sera donné à l'attribut « Aptitude », afin d'en respecter le caractère privé. Un exemple d'aplatissement est donné à la figure 3.20. L'attribut « Aptitude » étant entièrement privé au trait « TSoldat », celui-ci n'est pas *exposé* par les traits « TArcher » et « TCavalier » [BDNW07]. En d'autres mots, cet attribut est caché et rendu inaccessible à la classe « ArcherDeCavalerie » qui ne peut leur adjoindre un nom d'accès et éventuellement procéder à une fusion. Par conséquent, lors de l'aplatissement de la classe « ArcherDeCavalerie », c'est au processeur de décider ou non de la fusion de ces deux attributs cachés. Deux solutions sont donc possibles (représentées à la figure 3.21) :

- Soit le processeur fusionne les deux attributs masqués « TSoldat_Aptitude » en considérant qu'ils sont identiques car ayant le même nom et provenant du même trait d'origine.
- Soit le processeur ne fusionne pas les deux attributs masqués « TSoldat_Aptitude ». Auquel cas, un conflit de méthode apparaît. En effet, la méthode « GetAptitude », bien que provenant d'un même trait d'origine, se voit décliner désormais en deux versions, chacune utilisant son propre attribut pour représenter l'aptitude. Par conséquent, ce conflit doit être résolu, soit en choisissant l'une ou l'autre version, soit en proposant une nouvelle implémentation commune au sein de la classe « ArcherDeCavalerie ».

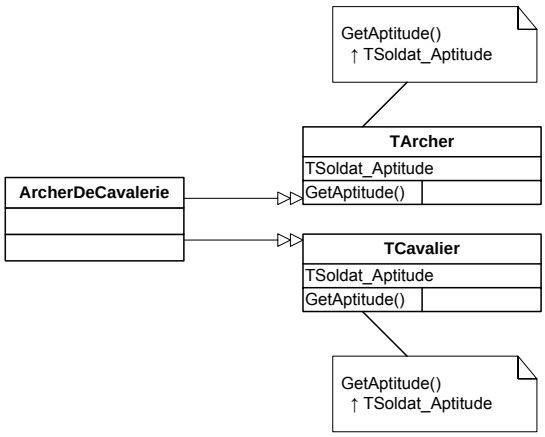

FIG. 3.20 – Aplatissement des traits « TArcher » et « TCavalier »

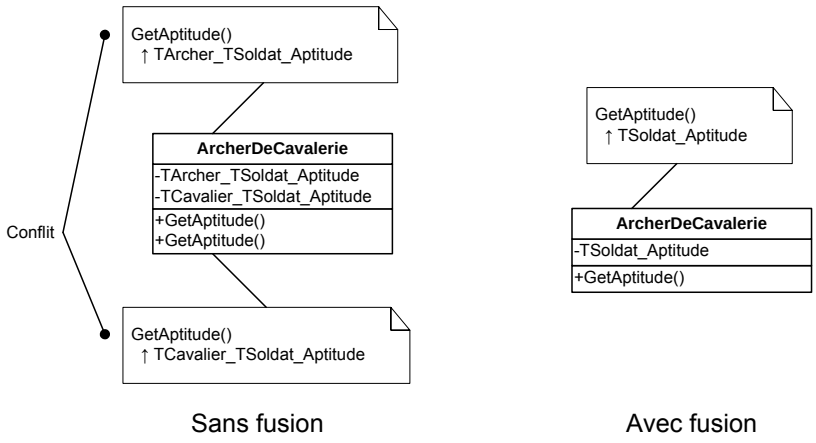

FIG. 3.21 – Aplatissement de la classe « ArcherDeCavalerie »

Si on avait voulu donner la possibilité à la classe « ArcherDeCavalerie » de pouvoir choisir le sort à réserver aux deux instances acquises de l'attribut « Aptitude », il aurait fallu que les deux traits « TArcher » et « TCavalier » usent de l'opérateur d'accès afin de fournir un nom d'accès à l'attribut « Aptitude » acquis à partir du trait « TSoldat ». Les deux traits auraient ainsi exposé l'attribut « Aptitude » à la classe « ArcherDeCavalerie » qui aurait, à son tour, pu utiliser l'opérateur d'accès afin de déterminer le sort à réserver à ces deux instances de l'attribut « Aptitude ». Un exemple de fusion des deux instances de l'attribut « Aptitude » par la classe « ArcherDeCavalerie » est donné à la figure 3.22.

Acquisition multiple d'une méthode de même origine

Alors qu'avec les traits sans état, l'acquisition multiple d'une méthode originaire d'un même trait par le biais de chemins différents n'était pas considérée comme un conflit, avec les traits à états cette situation pourrait être conflictuelle. En effet, un des attributs utilisés par la méthode acquise plusieurs fois pourrait, par exemple, se voir renommé sur l'un des chemins. Au final, la méthode est acquise selon plusieurs déclinaisons, chacune usant d'attributs de noms différents.

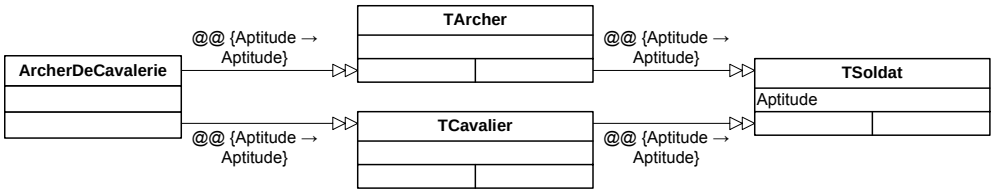

FIG. 3.22 – Exposition de l'attribut « Aptitude » par les traits « TArcher » et « TCavalier » fusionnés ensuite par la classe « ArcherDeCavalerie »

Afin de décréter qu'il n'y a pas de conflit, il faut donc s'assurer que toutes les versions d'une méthode de même origine acquise plusieurs fois soient identiques et qu'elles fasses toutes, dans leur corps, référence exactement aux mêmes attributs.

3.3 Traits et typage

De nombreux langages de programmation orientés objet, dont Java, utilisent un système de typage statique. C'est-à-dire que les informations de type relatives aux variables ou expressions sont connues à la compilation [Weg90] et non déterminées à l'exécution. Le compilateur peut dès lors effectuer lui-même certaines vérifications de consistance (par exemple vérifier qu'un message est bien envoyé à un objet capable de le traiter). Afin que ces informations de type puissent être connues du compilateur, le programmeur doit associer un *type* à chaque variable, constante, méthode ou paramètre.

Dans un contexte de langage typé statiquement, il est intéressant de se demander si les traits, à l'instar des classes, peuvent être considérés comme un type et dans quelle mesure il est possible de les intégrer dans un système de typage statique.

Dans cette section, nous allons d'abord introduire quelques notions de typage. Ensuite, nous verrons dans quelle mesure un trait peut être considéré comme un type et comment des informations de typage pourraient éventuellement être associées aux traits. Nous terminerons cette section par discuter des limitations de typage au sein même des traits et de la manière dont ces limitations peuvent être levées.

3.3.1 Notions de typage

Type et sous-type

En programmation orientée objet, correspond à chaque objet un ou plusieurs *types*. Un type permet de caractériser un objet sur base de contraintes [DT88], par exemple de définir *quelles* opérations peuvent être effectuées sur l'objet et *comment*.

Dans de nombreux langages orientés objet, la définition d'une classe entraîne automatiquement la définition d'un type correspondant (les classes sont alors considérées comme un type). En effet, une classe détermine un ensemble de contraintes ; par exemple elle borne l'ensemble

de messages qui peuvent lui être envoyés. Un type peut dès lors être défini sur base de ces contraintes.

Lorsqu'un objet est compatible avec un type X, en d'autres mots lorsqu'il respecte les contraintes définies par X, on dira que cet objet *est de type* X. La figure 3.23 illustre l'instanciation de l'objet « Nelson » à partir de la classe « Soldat ». On dira que l'objet « Nelson » *est de type* « Soldat », puisque « Nelson » est compatible avec le type « Soldat ». Cette compatibilité de type est trivialement due au fait que l'objet « Nelson », étant instancié à partir de la classe soldat, en reprend toutes les caractéristiques et en respecte donc les contraintes définies.

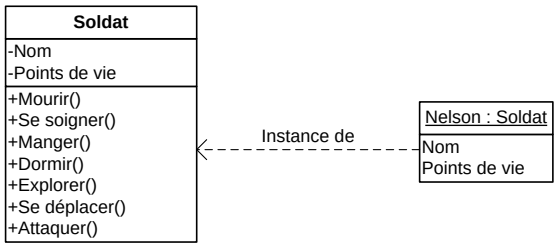

FIG. 3.23 – L'objet « Nelson » *est de type* « Soldat »

Certains langages orientés objet construisent une hiérarchie de types calquée sur la hiérarchie de classes. Apparaissent dès lors des relations de *sous-type* et de *super-type*. Soit un sous-type S dont le super-type est T. Dans de nombreux langages, si un objet est de type S alors il sera également de type T. Cette assertion suppose que les objets instanciés à partir de la sous-classe rencontrent toutes les contraintes de la classe parente. Ce qui est aisément envisageable, puisqu'une sous-classe ne fait que spécialiser ou rajouter du comportement, elle n'enlève rien à la classe parente.

Si on prend l'exemple illustré à la figure 3.24, on peut dire que la classe « Cavalier » est un *sous-type* de la classe « Soldat » puisque la classe « Cavalier » est une sous-classe de la classe « Soldat ».

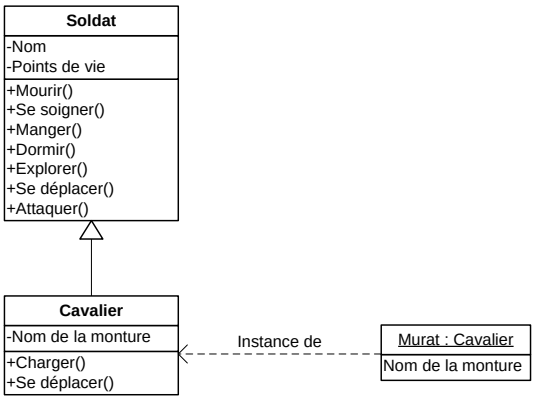

FIG. 3.24 – La classe « Cavalier » est un *sous-type* de la classe « Soldat »

L'objet « Murat », instancié à partir de la classe « Cavalier », est trivialement de type « Cavalier ». Puisque la classe « Cavalier » est un *sous-type* de la classe « Soldat », l'objet « Murat » est également de type « Soldat ». On a donc un premier exemple d'objet possédant

plusieurs types.

Typage statique et dynamique

En orienté objet, on distingue essentiellement deux grandes classes de mécanismes de typage [Car97, CW85] :

- Le typage dynamique : les types sont implicites et associés aux objets à l'exécution, lors de leur création.
- Le typage statique : le programmeur associe explicitement un type aux variables, constantes, paramètres, retours de méthode et opérateurs.

Le typage statique permet d'effectuer des vérifications de contraintes à la compilation. Par exemple, de vérifier la consistance des envois de messages ou, dit autrement, de s'assurer que les messages sont bien envoyés à des objets capables de les traiter.

Un exemple d'application du typage statique est la restriction contrôlée du champ d'application d'une méthode. En d'autres mots, grâce au typage statique, on pourra s'assurer que des objets passés en paramètre d'une méthode, qu'elle suppose d'un certain type, correspondent bien au type attendu et que les objets passés en paramètre seront en mesure de comprendre les messages que la méthode leur enverra. Imaginons par exemple une méthode qui ne s'applique qu'à des objets de type « Cavalier ». On définira alors le paramètre de cette méthode comme étant de type « Cavalier ». Ainsi, si la méthode est appelée en lui passant en paramètre un objet d'un autre type, le compilateur sera en mesure de détecter cette inconsistance.

Parmi les langages typés dynamiquement on retrouve notamment Smalltalk, Perl et Lisp. Des exemples de langages typés statiquement sont Java, C# et C++.

3.3.2 Les traits comme type

Les langages orientés objet typés statiquement reconnaissent généralement les classes comme étant un type. Dès lors, le nom d'une classe peut servir pour typer une variable, un paramètre de méthode, etc. Nous avons déjà évoqué à la section précédente un exemple de méthode dont son paramètre était défini comme étant de type « Cavalier ».

Une question naturelle qui se pose est de savoir si on peut considérer les traits comme étant un type. Et, si oui, comment peut-on leur associer cette notion de type dans le cadre d'un langage de programmation existant.

Les traits renferment un ensemble statique de méthodes permettant d'ajouter du comportement à une classe. Cet ensemble de méthodes définit une interface, laquelle viendra enrichir l'interface existante de la classe sur laquelle le trait est appliqué. On peut dès lors considérer les traits comme pouvant être un type.

Cette conclusion doit néanmoins être nuancée. En effet, lorsqu'un trait T est appliqué à une classe, l'opérateur d'exclusion permet d'exclure un nombre arbitraire de méthodes de ce trait.

Conséquence, c'est une version incomplète de l'interface initiale du trait T qui vient enrichir l'interface de la classe. Dès lors, les objets instanciés à partir de cette classe ne satisfont plus l'intégralité de l'interface définie par le trait. Ils ne peuvent donc pas être considérés comme étant de type T. Face à ce constat, deux alternatives sont possibles :

- Soit on restreint l'utilisation de l'opérateur d'exclusion à la seule résolution de conflits. Auquel cas, l'interface apportée par le trait reste nécessairement complète. En effet, lors de la résolution d'un conflit, une méthode exclue l'est toujours au profit d'une autre qui viendra la substituer.
- Soit on autorise l'exclusion arbitraire de n'importe quelle méthode d'un trait. Auquel cas, on accepte que les objets instanciés à partir d'une classe n'intégrant que partiellement un trait ne soient pas considérés comme étant du type de ce trait.

3.3.3 Ajout du typage aux traits

Si l'on désire associer la notion de type aux traits, une première approche consiste à modifier le système de typage existant du langage afin que les traits puissent être reconnus comme étant des types à part entière, de la même manière que les classes sont reconnues comme étant des types. Cette approche nécessite une persistance de la notion de trait tout au long du processus de typage et une modification profonde du système de typage. En particulier, le compilateur devra être lourdement retravaillé, ce qui n'est pas toujours souhaité, voir même possible.

Un des gros atouts des traits est la possibilité de les ajouter à un langage existant sans avoir à modifier le compilateur d'origine de ce langage. En effet, la propriété d'aplatissement induit la possibilité de transformer un code source usant de traits en un code source de sémantique totalement équivalente n'usant plus de trait et dès lors compilable par le compilateur d'origine du langage. Dans cette optique, une approche alternative permettant d'associer un type à un trait et utilisant des mécanismes existants du langage d'origine est à privilégier.

De nombreux langages orientés objet récents supportent la notion d'*interface* telle qu'introduite par le langage Java et qui permet la définition explicite de types afin de les utiliser comme mécanisme de sous-typage. Si un langage de programmation supporte cette notion d'interface, un type pourra alors être associé aux traits sans qu'aucune modification du compilateur d'origine ne soit nécessaire. Avant d'analyser comment cette association de type peut être effectuée, introduisons d'abord plus en détails cette notion d'interface.

Les interfaces (telles qu'introduites par le langage Java)

Une interface, telle qu'introduite par le langage Java, est tout simplement un ensemble de signatures[8] de méthodes. Une interface pourrait être vue comme étant un trait sans état ne contenant que des méthodes requises.

[8]La signature d'une méthode est ce qui permet de l'identifier. Il peut s'agir du nom de la méthode. Mais parfois cette information seule ne suffit pas pour identifier univoquement une méthode. D'autres éléments peuvent dès lors constituer la signature d'une méthode, tels le nombre de ses paramètres ou leur type.

Une classe peut annoncer implémenter une interface. Elle s'engage alors à fournir une implémentation pour chacune des méthodes définies dans l'interface.

Les interfaces sont reconnues comme étant un type. Par conséquent, tout objet instancié à partir d'une classe C implémentant une interface I sera à la fois de type C et I. Un exemple d'interface, représentée suivant la notation UML, est donné à la figure 3.25. L'interface « IInflammable » définit trois signatures de méthode. La classe « BâtimentInflammable » annonce implémenter l'interface « IInflammable » par le biais d'une ligne terminée d'un cercle. Elle s'engage donc à fournir une implémentation pour les trois méthodes définies dans l'interface « IInflammable ». Conséquence, tout objet de type « BâtimentInflammable » pourra, par exemple, être assigné à une variable de type « IInflammable ».

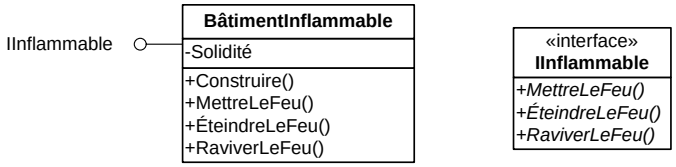

FIG. 3.25 – Exemple d'interface

La notion d'interface est souvent introduite dans un langage afin de compenser l'absence d'héritage multiple. En effet, une classe peut déclarer implémenter *plusieurs* interfaces. À défaut donc de permettre à une classe d'être construite incrémentalement à partir de plusieurs autres classes parentes, cela lui permet au moins de construire incrémentalement son type sur base de plusieurs autres types.

Association d'informations de typage aux traits

À la lecture de cette présentation des interfaces, on comprend facilement comment des informations de typage pourraient être associées aux traits, sans aucune modification des mécanismes de typage du langage. Il suffit en effet d'adjoindre à chacun des traits une interface correspondante. Cette interface reprendrait les signatures de l'ensemble des méthodes définies au sein du trait.

Cette association d'interface peut s'opérer de trois manières différentes :

1. Soit les interfaces correspondant aux traits sont générées automatiquement lors de l'aplatissement. On veillera que soit donné aux interfaces le même nom que leur trait correspondant. Le programmeur pourra ainsi utiliser le nom du trait comme nom de type dans son code. Lors du processus d'aplatissement, on veillera également à changer les déclarations des classes usant de traits afin qu'elles annoncent l'implémentation des interfaces correspondantes. Un exemple d'un tel aplatissement est donné à la figure 3.26.
2. Soit les interfaces correspondant aux traits sont déclarées manuellement par le programmeur et laissées indépendantes du trait. Cette approche a l'avantage de laisser un contrôle total au programmeur du typage de ses traits. Par exemple, le programmeur pourra décider de masquer l'utilisation d'un trait par une classe en n'annonçant pas qu'elle implémente l'interface correspondante. Avec cette approche, tout le travail revient au programmeur.

3. Soit les interfaces correspondant aux traits sont déclarées manuellement par le programmeur et associées à leur trait. Lors du processus d'aplatissement, on veillera alors à changer les déclarations des classes usant de traits afin qu'elles annoncent l'implémentation des interfaces associées aux traits par le programmeur. Cette approche a l'avantage de combiner une certaine flexibilité, tout en maintenant un certain automatisme. Le programmeur peut en effet choisir lui-même le nom des interfaces et décider d'associer ou non une interface à un trait. Un exemple de cette approche est illustré à la figure 3.27.

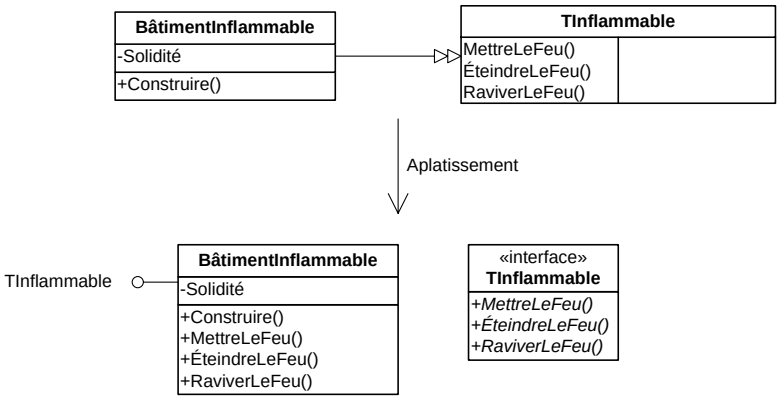

FIG. 3.26 – Exemple de génération automatique d'interfaces lors de l'aplatissement

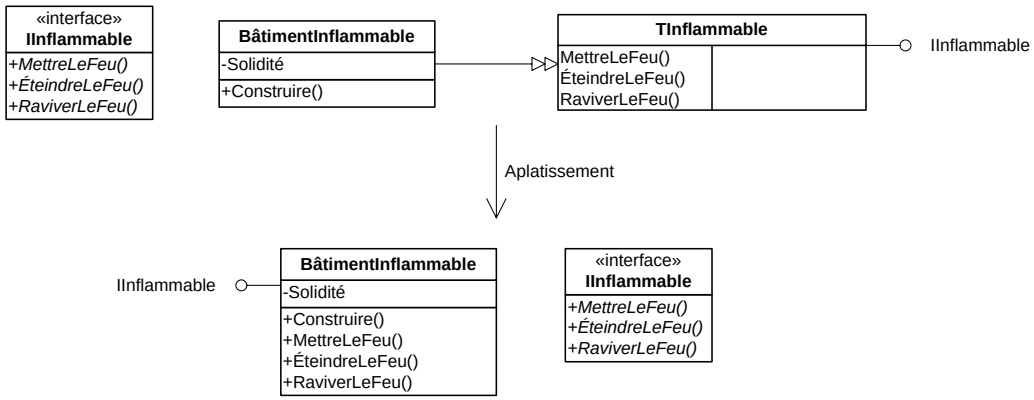

FIG. 3.27 – Exemple d'association automatique d'interfaces lors de l'aplatissement

3.3.4 Typage au sein d'un trait

Un autre problème de typage relatif aux traits, lorsqu'ils sont implémentés dans un langage typé statiquement, concerne le typage des méthodes, attributs, paramètres, etc. au sein des traits.

Considérons l'exemple suivant. Imaginons, toujours dans le cadre du jeu vidéo de stratégie militaire, que l'on désire pouvoir associer deux à deux des personnages de même type afin qu'ils puissent effectuer diverses actions conjointement. Afin d'ajouter ce comportement aux différentes classes représentant les personnages, un nouveau trait « TAssociable » est créé. Deux méthodes sont implémentées au sein de ce trait : « GetAssocié » et « SetAssocié » qui permettent respectivement de récupérer et modifier l'associé d'un personnage. Cet exemple est illustré à la

figure 3.28.

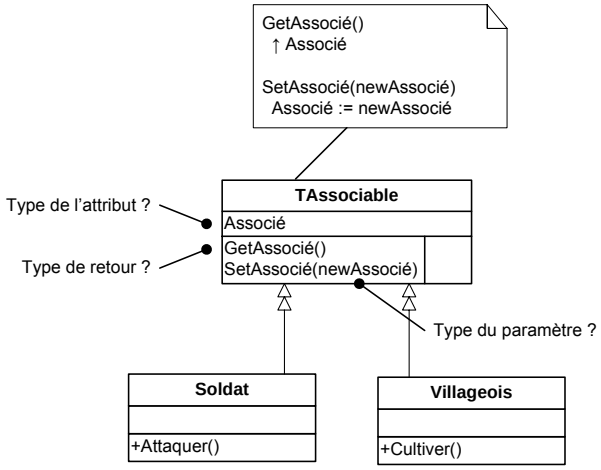

FIG. 3.28 – Exemple de problèmes de typage au sein d'un trait

Au vu de cet exemple, trois questions de typage se posent :

- Quel type associer à l'attribut « Associé » ?
- Quel type de retour associer à la méthode « GetAssocié » ?
- Quel type associer au paramètre « newAssocié » ?

Si l'on désire que seuls des personnages de même type puissent être associés (un soldat avec un soldat, mais pas un soldat avec un villageois), alors la réponse à ces trois questions dépend entièrement des classes sur lesquelles est appliqué le trait. Appliqué à la classe « Soldat », le trait devrait faire usage du type « Soldat ». Par contre, appliqué à la classe « Villageois », le trait devrait faire usage du type « Villageois ». Dans leur forme actuelle, les traits ne permettent pas d'exprimer cette nuance de type et de, par exemple, définir un attribut comme étant du type de la classe sur laquelle le trait est appliqué.

Deux extensions aux traits ont été proposées [NDRS05] afin d'apporter une solution à ce problème :

- Soit l'ajout d'un nouveau mot-clé, utilisable au sein des traits, désignant la classe sur laquelle le trait est appliqué. Lors du processus d'aplatissement, ce mot-clé serait alors remplacé par le nom de la classe dans laquelle sont recopiés les méthodes et attributs du trait.
- Soit l'ajout d'un mécanisme de paramétrisation des traits. Ces derniers pourraient alors se voir associer des paramètres représentant un type, à la manière des *generics*[9] en Java ou en C#. Ces paramètres seraient par la suite utilisé au sein du trait pour représenter le type de certains attributs, valeurs de retour, etc. La classe « Soldat » utiliserait alors, par exemple, le trait « TAssociable » en lui passant comme paramètre le type « Soldat », tandis que la classe « Villageois » passerait au trait comme paramètre le type « Villageois ».

[9]Pour plus de détails sur le concept de *generics*, nous renvoyons le lecteur à d'autres ouvrages.

3.4 Travaux actuels et futurs sur les traits

3.4.1 Langages supportant les traits

De nombreux langages de programmation ont déjà intégré les traits avec succès, soit nativement, soit par le biais d'extensions proposées. Parmi les principaux, on citera :

Squeak Squeak est un dialecte du langage de programmation Smalltalk. Ce langage orienté objet, typé dynamiquement, a été retenu par les auteurs des traits pour leurs premières implémentations [SDNB03, BDNW07]. Squeak supporte donc à la fois les traits sans état et les traits à états.

Scala Scala est un langage de programmation fonctionnel orienté objet adressant, à l'heure actuelle, les plateformes Java et Dotnet. Une notion de trait est supportée par Scala. Dans sa première version [OAC+04], les traits représentaient une unité de réutilisation de code sans état permettant de regrouper des méthodes à la fois fournies ou requises. Leur composition était symétrique (i.e. l'ordre de composition n'a pas d'importance) et les conflits devaient être résolus explicitement. Néanmoins, la résolution de conflits était moins flexible étant donné l'inexistence d'opérateurs de surnommage ou d'exclusion. Depuis la version 2.0 du langage [OAC+06], Scala a changé sa philosophie de résolution de conflits en adoptant une approche similaire à celle des mixins et basée sur une linéarisation. Cette approche ne s'apparente donc désormais plus à la définition de traits présentées aux sections précédentes.

Perl Perl est un langage de programmation qui inclut, depuis sa version 5, certains concepts de l'orienté objet. Une extension à Perl 5 a été développée par Stevan Little afin de porter les traits (dans leur version sans état) pour ce langage [SDNB03]. Il est vraisemblable que dans la version 6 de Perl, les traits soient inclus de base dans le langage (sous le nom de « rôles »).

C# C# est un langage de programmation orienté objet typé statiquement adressant la plateforme Dotnet de Microsoft. Une extension du langage a été développée par Stefan Reichhart afin d'y incorporer les traits sans état [Rei05].

Java Java est également un langage de programmation orienté objet typé statiquement, auquel ce mémoire se propose d'ajouter la notion de trait. Un prototype d'implémentation a déjà été réalisé par Charles Smith [Smi04, 5]. Une nouvelle syntaxe a été introduite au langage Java afin d'y inclure la notion de traits. Afin de réaliser ce prototype, un nouveau compilateur a été écrit, malheureusement basé sur une ancienne spécification du langage Java (la version 1.0.2). De plus, le prototype développé introduit la notion de traits sans état et non la notion de traits à états.

3.4.2 Recherches actuelles ou futures

De nombreuses pistes de recherches relatives aux traits ne sont encore que partiellement explorées, voir totalement inexplorées. Les travaux de recherches concernant les traits sont donc encore loin d'être terminés. Dans cette section, nous allons parcourir quelques-unes des pistes de recherche encore laissées ouvertes.

Traits et encapsulation

Dans leur forme actuelle, les traits ne permettent pas de définir de méthodes qui leur sont *privées*. Toutes les méthodes d'un trait sont publiques et accessibles par leurs clients, ce qui, dans certains cas, peut être considéré comme une violation des principes d'encapsulation et de boîte noire. Dans quelle mesure des opérateurs de visibilité pourraient être introduits dans les traits ?

Remplacement de l'héritage par les traits

La cohabitation des traits et de l'héritage ne rend pas toujours très évidents les choix de décomposition et composition. En particulier, l'analyste-programmeur sera souvent confronté au dilemme suivant : quand doit-on plutôt utiliser l'héritage et quand doit-on plutôt se servir des traits ?

C'est pourquoi, il n'est pas déraisonnable de se demander dans quelle mesure les traits ne pourraient-ils pas complètement remplacer l'héritage de classes et de voir si, en conséquence de ce remplacement, la composition de traits ne devrait pas être étendue avec des nouveaux mécanismes.

Substitution dynamique des traits

Charles Smith et Sophia Drossopoulou ont développé l'idée d'une substitution dynamique des traits sans état [SD05]. En effet, puisque les traits sans état ne définissent *que* du comportement, on pourrait imaginer, qu'à l'exécution, un trait soit substitué par un autre, de même interface, mais avec une implémentation différente pour les méthodes. Cela permettrait de changer dynamiquement le comportement d'un objet, sans changer son interface. Une question qui se pose est de savoir si cette substitution pourrait également s'envisager avec des traits à états.

3.5 Conclusion

Les traits à états offrent un mécanisme puissant et intuitif pour réaliser de la réutilisation de code. Ils permettent en effet de répondre aux nombreuses limitations et problèmes des méthodes existantes. Un léger bémol cependant : les traits ne permettent pas la résolution de conflits sémantiques. Néanmoins, il n'est pas exclu que des mécanismes futurs soient imaginés afin de pallier cette limitation qui se révèle somme toute être peu face aux apports qu'offrent les traits,

ceux-ci pouvant même s'intégrer sans problème aux systèmes de typage des langages orientés objet typés statiquement.

C'est donc avec enthousiasme que nous espérons une large adoption des traits par les langages orientés objet existants. Thème qui fera l'objet du chapitre suivant puisque nous y étudierons l'intégration des traits à Java, langage orienté objet des plus populaire.

Chapitre 4

Ajout des traits à états à Java

Dans ce chapitre nous allons étudier l'ajout des traits à états à Java, un des langages de programmation orienté objet les plus populaires [CDM+05] à l'heure actuelle.

Plusieurs approches sont possibles afin d'intégrer la notion de trait à Java. Une première approche utilisant la programmation orientée aspect a été proposée par Simon Denier [Den04]. Après une rapide présentation de cette approche, nous verrons qu'elle possède une série de limitations que nous évoquerons brièvement.

La deuxième approche que nous étudierons est une intégration native et complète des traits au langage Java. Nous définirons pour ce faire un nouveau langage, nommé tJava, extension du langage Java intégrant la notion de traits à états. Ce nouveau langage introduira des compléments à la syntaxe originelle de Java ; une modification de la grammaire officielle de Java sera alors proposée.

Enfin, nous étudierons comment ce nouveau langage peut être implémenté. Deux solutions seront proposées. Soit la modification des compilateurs et des machines virtuelles existantes, soit l'implémentation d'un translateur basé sur la propriété d'aplatissement.

Avant de commencer à étudier les différentes approches pour ajouter les traits à états à Java, nous allons brièvement introduire le langage Java et quelques-unes de ses notions clés.

4.1 Le langage de programmation Java

Java est un langage de programmation développé originellement par l'entreprise Sun Microsystems dont la première version date de 1995. Java est un langage orienté objet. À ce titre, il en reprend certains concepts fondateurs dont les notions d'objet, de classe, d'héritage, d'encapsulation, d'abstraction et de polymorphisme. Dans les sections suivantes nous allons introduire brièvement le langage Java au travers de certaines de ces notions.

4.1.1 Classes et objets

Étant un langage de programmation orienté objet, Java impose de centrer l'entière conception des applications autour de la notion d'*objets*. Les définitions de méthodes et d'attributs doivent être regroupées au sein de *classes* qui servent de canevas à l'instanciation des objets. Le listing 4.1 illustre la définition d'une classe « Soldat ». Cette classe définit un attribut « pointsDeVie » (initialisé à 100) et deux méthodes. D'une part la méthode « Mourir » qui remet à zéro le nombre de points de vie, d'autre part un accesseur « getPointsDeVie » permettant de récupérer le nombre de points de vie restant.

Listing 4.1 – Exemple de définition d'une classe

```
class Soldat {
  private Integer pointsDeVie = 100;

  public void mourir() {
    pointsDeVie = 0;
  }

  public Integer getPointsDeVie() {
    return pointsDeVie;
  }
}
```

Des modificateurs d'accès permettent de spécifier la portée des attributs et des méthodes. Seuls les attributs et méthodes définis comme étant *public* sont accessibles de l'extérieur.

Le listing 4.2 illustre l'instanciation d'un objet. Un objet « nelson » est généré sur base de la classe « Soldat ». Le message « mourir » est ensuite envoyé à cet objet (en Java l'envoi d'un message à un objet correspond à l'invocation d'une de ses méthodes).

Listing 4.2 – Exemple d'instanciation d'un objet et d'envoi de message

```
Soldat nelson = new Soldat();
nelson.mourir();
```

4.1.2 Héritage

Le mécanisme de réutilisation de code retenu par Java est l'héritage simple. L'héritage multiple n'est dès lors pas possible, chaque classe ne pouvant avoir qu'un seul parent direct. Le listing 4.3 illustre un exemple d'héritage simple. La classe « Archer » étend la classe « Soldat » en lui ajoutant un nouvel attribut et une nouvelle méthode.

Listing 4.3 – Exemple d'héritage

```
class Archer extends Soldat {
  private Integer nombreDeFlèches = 50;

  public void tirer() {
    nombreDeFlèches--;
  }
}
```

4.1.3 Typage

Java est un langage orienté objet typé statiquement (cf. la section 3.3.1 introduisant les notions de typage). En d'autres mots, chaque variable, constante, méthode, paramètre ou opérateur doit être associé à un type. Sont, entre autres, considérés comme des types :

- les types primitifs[1] (nombres entiers, nombres flottants, booléens, etc.) ;
- les classes ;
- les interfaces.

Pour rappel, une interface définit un ensemble de signatures de méthodes qu'une classe peut s'engager à implémenter. Le listing 4.4 illustre la définition et l'usage d'une interface. L'interface « IInflammable » définit deux méthodes : « mettreLeFeu » et « eteindreLeFeu ». La classe « BatimentInflammable » annonce qu'elle implémente l'interface « IInflammable ». Elle fournit donc une implémentation pour les deux méthodes précitées.

Étant typé statiquement, Java autorise la *surcharge* (*overloading* en anglais) de méthodes. En d'autres mots, la possibilité de définir des méthodes de même nom, mais se distinguant par le nombre ou par le type de leurs paramètres. Pour désigner univoquement une méthode on utilisera dès lors sa signature et non son nom seul. La signature d'une méthode est composée de son nom et du type de ses paramètres [GJSB05].

4.1.4 Indépendance de plateforme

Les programmes Java sont indépendants de toute plateforme, en d'autres mots ils peuvent être exécutés sur de nombreuses architectures. Cette indépendance est obtenue par l'introduction d'un langage intermédiaire, le Java Bytecode, composé d'instructions simplifiées proches de celles d'un langage machine. Les programmes écrits en Java sont dès lors d'abord compilés en Java Bytecode et ensuite exécutés dans une machine virtuelle Java capable d'interpréter et d'exécuter le Bytecode en langage machine.

[1] En Java, les types primitifs ne sont pas des objets pour des raisons de performances.

Listing 4.4 – Exemple d'interface

```
interface IInflammable {
  void mettreLeFeu();
  void eteindreLeFeu();
}

class BatimentInflammable implements IInflammable {
  private Boolean estEnFeu;

  public void mettreLeFeu() {
    estEnFeu = true;
  }

  public void eteindreLeFeu() {
    estEnFeu = false;
  }
}
```

4.2 Émulation des traits à l'aide de la programmation orientée aspect

Dans cette section, nous allons voir comment les traits peuvent être émulés à l'aide de la programmation orientée aspect, paradigme de programmation que nous allons tout d'abord brièvement introduire.

4.2.1 Concepts de la programmation orientée aspect

La programmation orientée aspect [EFB01] est un paradigme de programmation proposant de regrouper les aspects transversaux d'un programme au sein de modules séparés, appelés *aspects*. Le code composant chaque aspect est alors ensuite automatiquement greffé aux différentes composantes du programme concernées par un ou plusieurs aspects. Cette greffe est réalisée par un outil appelé *tisseur d'aspects*.

Les méthodes de réutilisation de code étudiées jusqu'à présent proposent essentiellement une découpe fonctionnelle. En d'autres mots, les briques de base (méthodes, objets, etc.) utilisées pour la construction des programmes ne sont généralement pas fragmentées, leur composition est donc relativement linéaire [EFB01]. La programmation orientée aspect permet une composition de granularité plus fine. Il est par exemple possible de regrouper dans un aspect, du code qui sera, par la suite, greffé *au sein même* d'une série de méthodes.

L'exemple souvent repris afin d'illustrer les possibilités de la programmation orientée aspect est le suivant. Imaginons que l'on veuille tracer toute modification d'état apportée aux différents objets d'un programme. On suppose qu'une modification s'exprime par un appel à un mutateur (i.e. une méthode commençant par *set*). En programmation orientée objet traditionnelle, du

code devra être ajouté à chaque mutateur de chaque objet dont l'objectif sera d'enregistrer qu'un changement a été opéré. En programmation orientée aspect, ce code pourra être regroupé au sein d'un aspect et il pourra être demandé au tisseur de greffer ce code à chaque début de méthode dont le nom commence par « set ».

Cette brève présentation de la programmation orientée aspect met en évidence certaines similitudes entre *traits* et *aspects*. Tous deux permettent en effet de regrouper du code qui pourra être ensuite greffé à des classes. Ces similitudes seront d'autant plus mises en évidence à la section suivante, laquelle introduira AspectJ, une extension orientée aspect du langage Java.

4.2.2 AspectJ

AspectJ, extension orientée aspect du langage Java [KHH+01, 2], propose deux méthodes pour greffer à une classe du code situé dans un aspect [KHH+01] :

- L'*entrecroisement dynamique* qui permet l'exécution de code supplémentaire à différents points d'un programme.
- L'*entrecroisement statique* qui permet d'enrichir un type avec, entre autres, de nouvelles méthodes ou attributs. On parle également d'*introduction* d'attributs ou de méthodes.

Intéressons-nous plus particulièrement à l'entrecroisement statique, qui semble se rapprocher le plus de ce à quoi sont destinés les traits.

Afin de réaliser un entrecroisement statique, ou, en d'autres mots, une introduction de méthodes dans une classe, AspectJ introduit un nouveau type de modules, nommés *aspect*, dans lesquels peuvent être regroupées des méthodes qui viendront enrichir un ensemble choisi de classes. Un exemple est illustré au listing 4.5. Un aspect « TInflammable » est défini ; il est composé de deux méthodes « mettreLeFeu » et « eteindreLeFeu » qui viennent enrichir la classe « BatimentInflammable ».

Listing 4.5 – Simulation d'un trait à l'aide d'un aspect

```
aspect TInflammable {
  public void BatimentInflammable.mettreLeFeu() {
    /* du code relatif à la mise en feu */
  }

  public void BatimentInflammable.eteindreLeFeu() {
    /* du code relatif à l'extinction du feu */
  }
}

class BatimentInflammable {
  /* Code de la classe */
}
```

Une question se pose : l'aspect « TInflammable » du listing 4.5 ne pourrait-il finalement pas être considéré comme un trait ?

En effet, cet aspect vient enrichir une classe en lui adjoignant de nouvelles méthodes, tout comme le ferait un trait. Cependant, une différence de taille sépare ce premier exemple d'un vrai trait : l'aspect doit définir lui-même ses utilisateurs. Or, dans une optique de réutilisation de code, il serait plus naturel que ce soit le réutilisateur qui définisse lui-même les traits qu'il désire utiliser, laissant le code du trait indépendant de toute référence à un quelconque utilisateur. Pire, si on désire appliquer ce trait à une autre classe, cette première solution impose une duplication du code, la syntaxe d'AspectJ ne permettant pas d'exprimer l'ajout d'une même méthode à plusieurs classes.

Malgré ce constat, les traits peuvent-ils être émulés à l'aide d'AspectJ ? C'est à cette question que nous allons tenter de répondre à la section suivante.

4.2.3 Émulation des traits à l'aide d'AspectJ

Simon Denier, de l'École des Mines de Nantes, s'est intéressé plus en profondeur aux similitudes qui lient aspects et traits et a finalement réussi à proposer une émulation des traits légèrement plus réaliste [Den04].

Pour ce faire, Simon Denier s'est basé sur le fait qu'AspectJ permet l'introduction de méthodes dans une classe *par le biais d'interfaces*. En d'autres mots, AspectJ peut déclarer une méthode comme faisant partie intégrante d'une interface. Toutes les classes implémentant cette interface se verront alors enrichies des méthodes ajoutées par AspectJ à l'interface. Le listing 4.6 illustre cette construction.

Listing 4.6 – Simulation d'un trait à l'aide d'un aspect et d'une interface

```
interface TInflammable {}

aspect TInflammableImplementation {
  public void TInflammable.mettreLeFeu() {
    /* du code relatif à la mise en feu */
  }

  public void TInflammable.eteindreLeFeu() {
    /* du code relatif à l'extinction du feu */
  }
}

class BatimentInflammable implements TInflammable {
  /* Code de la classe */
}
```

Avec cette solution, les utilisateurs finaux du trait ne doivent plus être référencés au sein de l'aspect. Seule une entité abstraite y est référencée. De plus, ce sont les classes qui manifestent

leur volonté d'utiliser un trait et non plus les traits qui désignent leurs utilisateurs. Cette solution permet également de définir un ensemble de méthodes requises. Il suffit pour ce faire d'enrichir l'interface TInflammable à l'aide de nouvelles signatures de méthodes.

4.2.4 Règles de précédence et résolution de conflits

Les traits imposent le respect strict de règles de précédence afin d'offrir une résolution implicite et naturelle de certains conflits. Malheureusement, l'émulation des traits à l'aide d'AspectJ ne respecte pas toutes les règles de précédence.

Si une classe définit déjà une méthode acquise également à partir d'un aspect, la méthode issue de l'aspect ne sera pas ajoutée à la classe. Il semblerait[2] donc que les méthodes définies au sein des classes ont précédence sur celles définies au sein d'aspects.

Par contre lorsqu'une classe hérite d'une méthode également définie dans un aspect, alors que la méthode définie dans l'aspect devrait avoir précédence, le tisseur d'aspects génère une erreur de conflit. Cette règle de précédence n'est donc pas respectée.

Une autre caractéristique des traits est leur relative habilité à résoudre les conflits engendrés lorsque plusieurs traits proposant une même méthode sont composés. Les traits proposent à cet effet deux opérateurs : un opérateur de surnommage et un opérateur d'exclusion. Malheureusement, AspectJ ne propose pas de tels mécanismes pour résoudre les conflits. Une solution est néanmoins proposée par Simon Denier : le recours à la délégation de classe. Nous n'entrerons pas dans les détails de cette solution, nous invitons donc le lecteur à consulter l'article de référence [Den04]. Il est néanmoins important de noter que cette solution, en plus de briser la propriété d'aplatissement, est assez verbeuse et complexifie la structuration du code, montrant là les premières limites de cette émulation par rapport à une réelle intégration des traits au langage.

Enfin, bien que la définition de traits composites soit possible (i.e. la construction de traits à partir d'autres traits), il est à noter que la résolution de conflits au sein de traits composites est quasiment irréalisable [Den04].

4.2.5 En conclusion

En conclusion, bien qu'une partie du comportement des traits puisse être simulée par l'utilisation d'AspectJ, cette approche montre une série de limites et peut s'avérer très laborieuse à l'usage. Une intégration native des traits au langage Java est dès lors préférable. Notons également que l'émulation proposée ne s'intéresse qu'aux traits sans état. Or les modules d'aspect d'AspectJ permettent la définition en leur sein d'attributs. Il serait donc également intéressant d'étudier comment les traits à états peuvent être émulés à l'aide d'AspectJ et comment les conflits peuvent être résolus.

[2]Il s'agit d'un résultat expérimental de Simon Denier [Den04], aucune confirmation de ce comportement n'a été trouvée dans la documentation d'AspectJ.

4.3 Un nouveau langage : tJava

Dans cette section nous allons introduire un nouveau langage, tJava, qui se veut être une extension au langage Java originel intégrant la notion de traits à états. Dans un premier temps nous allons proposer diverses extensions à la syntaxe originelle par le biais d'exemples. Nous formaliserons ensuite ces extensions de syntaxe en proposant une modification de la grammaire officielle de Java.

4.3.1 Une nouvelle syntaxe

Le langage tJava modifie et étend la syntaxe de Java afin de lui ajouter la notion de trait. Dans cette section, nous allons introduire, par le biais d'exemples, les différentes constructions ajoutées au langage. La nouvelle syntaxe retenue se base sur la syntaxe proposée par Stefan Reichhart dans son implémentation des traits en C# [Rei05], elle-même inspirée et adaptée de l'implémentation faite en Smalltalk. L'avantage de reprendre une syntaxe existante est d'éviter aux programmeurs les efforts liés à l'apprentissage et à la familiarisation d'une syntaxe totalement nouvelle.

Définition de traits

La définition de traits est similaire à la définition de classes. Le mot-clé *trait* définit un nouveau conteneur dans lequel peuvent être regroupés des méthodes et des attributs. Ces méthodes et attributs ne sont pas préfixés par un quelconque opérateur de visibilité (*public*, *private*, etc), car, dans leur définition formelle, les traits, ne définissent pas la personnalisation de la visibilité des méthodes et attributs.

Un trait peut déclarer implémenter une interface à l'aide du mot-clé habituel *implements*, conformément à la troisième stratégie de typage évoquée à la section 3.3.3 et illustrée à la figure 3.27. Cette stratégie de typage a été retenue car elle offre, selon nous, le meilleur compromis entre flexibilité et automatisme.

Le listing 4.7 illustre la définition d'un trait à états. On suppose l'interface « IVendable » définie ailleurs et reprenant les signatures des deux méthodes implémentées dans le trait.

Utilisation de traits par une classe

Un nouveau mot-clé *uses* est ajouté à la syntaxe afin qu'une classe puisse déclarer l'utilisation d'un ou de plusieurs traits. Cette déclaration est située au sein du corps de la classe et non dans la déclaration du type (par exemple après la clause *extends*) afin de ne pas la surcharger. Car, comme on le verra par la suite, la déclaration d'usage de traits peut parfois s'avérer assez longue. De plus, placer cette déclaration d'usage de traits au sein même du corps de la classe a l'avantage de bien symboliser la philosophie sous-jacente aux traits, en particulier leur propriété d'aplatissement qui pourrait être vue comme un remplacement de la clause *uses* par le contenu

Listing 4.7 – Exemple de définition d'un trait

```
import java.text.*;

trait TVendable implements IVendable {
  Double prixDeBase;

  Double calculerPrixVente() {
    return prixDeBase * 1.21;
  }

  String toString(NumberFormat nf) {
    return nf.format(calculerPrixVente());
  }
}
```

des traits utilisés.

Listing 4.8 – Exemple d'utilisation de traits par une classe

```
class NavireAchetableVendable extends Navire {
  uses {
    TVendable; TAchetable;
  }
}
```

Opérateur de surnommage

Lorsqu'une classe déclare utiliser un trait, elle peut décider de surnommer une ou plusieurs méthodes du trait utilisé (i.e. leur ajouter un nouveau nom). Cet opérateur est symbolisé à l'aide d'une flèche, laquelle, contrairement à la présentation faite dans le chapitre précédent, doit être lue comme « est surnommée ». En d'autres mots, à droite de la flèche se trouve le nom additionnel de la méthode définie à gauche de la flèche.

Dans de nombreux langages typés statiquement, dont Java, deux méthodes de même nom peuvent coexister, pour autant que leurs paramètres soient différents (en nombre ou en type). Le nom seul d'une méthode ne permet plus une identification univoque, il faut utiliser les signatures des méthodes. La signature d'une méthode est composée de son nom et des types de ses paramètres [GJSB05]. C'est pourquoi, à la fois le nom et les types des paramètres doivent être spécifiés pour identifier une méthode que l'on désire surnommer. Lorsqu'une clause de surnommage référence une méthode sans paramètre, le parenthésage vide ne peut être omis ; afin que le nom d'une méthode ne soit pas confondu avec un attribut du trait.

Un exemple de surnommage est donné au listing 4.9.

Listing 4.9 – Exemple de surnommage de méthodes

```
class NavireAchetableVendable extends Navire {
  uses {
    TVendable { toString(NumberFormat) -> toStringFromVendable; };
    TAchetable { toString(NumberFormat) -> toStringFromAchetable; };
  }
}
```

Opérateur d'exclusion

L'opérateur d'exclusion est symbolisé par un accent circonflexe. Comme pour l'opérateur de surnommage, lorsqu'on désire exclure une méthode, sa signature complète doit être spécifiée.

Le listing 4.10 illustre un exemple d'exclusion de méthodes. Toutes les méthodes du trait sont exclues. Chaque clause d'exclusion est séparée par un point-virgule.

Listing 4.10 – Exemple d'exclusion de méthodes

```
class NavireAchetableVendable extends Navire {
  uses {
    TVendable { ^toString(NumberFormat); ^calculerPrixVente(); };
  }
}
```

Opérateur d'accès à un attribut

Par défaut, tous les attributs d'un trait à états sont privés. Néanmoins, les utilisateurs d'un trait peuvent associer à ces attributs des noms d'accès afin de pouvoir les manipuler. L'opérateur retenu pour associer un nom d'accès à un attribut est la flèche. Tout comme avec l'opérateur de surnommage de méthode, la flèche doit être lue comme « est accessible via ». En d'autres mots, à droite de la flèche se trouve le nom d'accès de l'attribut défini à gauche de la flèche. Un exemple d'accès est donné au listing 4.11.

Listing 4.11 – Exemple d'accès à des attributs d'un trait

```
class NavireAchetableVendable extends Navire {
  uses {
    TVendable { prixDeBase -> prixVenteDeBase; };
    TAchetable { prixDeBase -> prixAchatDeBase; };
  }
```

Les opérateurs de surnommage et d'accès sont tous deux symbolisés par une flèche. Néanmoins, l'un porte sur des méthodes, l'autre sur des attributs. Raison pour laquelle le parenthésage

vide d'une méthode sans paramètre ne peut être omis, afin qu'il soit possible de distinguer une méthode d'un attribut, et donc de distinguer un surnommage d'un accès à un attribut.

Définition de traits composites

Tout comme une classe peut être composée de traits, un trait peut être composé d'autres traits. La même clause *uses* est utilisée, exactement de la même manière que pour une classe. Un exemple de trait composite est donné au listing 4.12.

Listing 4.12 – Exemple de trait composite

```
trait TAchetable {
  uses {
    // Fusion des attributs prixDeBase
    TTaxable { prixDeBase -> prixDeBase; };
    TReductionnable { prixDeBase -> prixDeBase; };
  }

  Double calculerPrixVente() {
    // Les méthodes calculerTaxe et calculerReduction sont supposées acquises
    // des traits TTaxable et TReductionnable
    return prixDeBase + calculerTaxe() - calculerReduction();
  }
}
```

Définition des méthodes requises

Les traits peuvent non seulement fournir un ensemble de méthodes implémentées, mais également requérir de la part de leurs clients l'implémentation de méthodes dont ils auraient éventuellement usage.

Deux approches sont possibles pour la définition des méthodes requises :

- Soit on considère la définition des méthodes requises comme implicite. La liste des méthodes requises est alors construite sur base des appels non résolus à des méthodes.
- Soit on impose au programmeur de définir explicitement lui-même la liste des méthodes requises.

Les deux approches possèdent chacune leurs avantages et inconvénients. La définition implicite de méthodes demande moins de travail de la part du programmeur. Néanmoins, avec cette approche, on ne peut faire la différence entre un appel à une méthode volontairement non résolu et un appel à une méthode involontairement non résolu dû à une erreur de la part du programmeur. Par exemple, si le programmeur désire faire appel à la méthode « toString », mais qu'il fait par erreur appel à la méthode « toSting », une nouvelle méthode requise est ajoutée à la liste, bien qu'en fait il s'agisse d'un appel erroné à la méthode « toString ». Afin d'éviter ce genre de situation, les outils utilisés par le programmeur devront veiller à ce que ce dernier soit

constamment informé du résultat de la construction de la liste des méthodes requises. Il pourra ainsi se rendre compte rapidement de son erreur.

Certains argueront également que l'approche implicite ne permet pas de définir des méthodes requises non utilisées par le trait, méthodes dont l'unique but serait d'enrichir l'interface du trait (ce que l'approche explicite permet), ce qui peut avoir un sens si l'on désire considérer les traits comme des sous-types. Cette limitation peut néanmoins être levée en associant au trait une interface (au sens Java du terme) à laquelle seront ajoutées les signatures des méthodes requises non utilisées.

Privilégiant la simplicité pour le programmeur, nous avons retenu l'approche implicite. En d'autres mots, notre syntaxe ne prévoit pas la définition de méthodes requises.

Si l'on désire retenir la deuxième approche, à savoir imposer au programmeur la définition explicite des méthodes requises, deux syntaxes peuvent être proposées. Soit l'ajout d'une clause *requires*, soit la possibilité de définir des méthodes *abstraites* au sein du trait. Ces deux propositions de syntaxe sont illustrées aux listing 4.13 et 4.14.

Listing 4.13 – Exemple de définition de méthodes requises

```
trait TVendable {
  requires {
    Double calculerTaxe();
  }
}
```

Listing 4.14 – Exemple de définition de méthodes abstraites

```
trait TVendable {
  abstract Double calculerTaxe();
}
```

Typage au sein d'un trait

À la section 3.3.4, des problèmes de typage au sein des traits ont été évoqués. En particulier, il a été mis en évidence le besoin de pouvoir bénéficier d'un mécanisme permettant de désigner *le type de la classe utilisatrice du trait*. Cet aspect ne sera pas abordé dans notre proposition de syntaxe.

4.3.2 Extension de la grammaire de Java

Afin de formaliser notre proposition de nouvelle syntaxe, nous l'avons intégrée dans la grammaire officielle de Java (version 1.5) publiée dans les spécifications du langage [GJSB05]. Avant de présenter les extensions apportées à la grammaire du langage Java, nous allons brièvement rappeler le concept de grammaire.

Notions de grammaire

Pour décrire formellement un langage, on utilise le concept de *grammaire générative* qui permet de décrire l'ensemble des phrases possibles d'un langage.

Par exemple, pour décrire formellement la langue française on dira qu'une phrase commence par un *sujet*, suivi d'un *verbe* pour se terminer par un *complément*. Un sujet sera par exemple défini comme étant un *nom* précédé d'un *article* et suivi éventuellement d'un *adjectif*. Un nom sera défini comme étant soit « Thierry », soit « Yves », etc. Et ainsi de suite. Ces contraintes peuvent être modélisées par un ensemble de *règles de production* (appelées aussi règles de réécriture) :

phrase → *sujet verbe complément*
sujet → *article nom adjectif*
→ *article nom*
nom → `Thierry`
→ `Yves`
...
article → `le`
→ `la`
...

De cet exemple, on constate que deux types de symboles sont à distinguer :

- Les *symboles terminaux*, en caractères à taille fixe (par exemple `Thierry` ou `le`).
- Les *symboles non-terminaux* en italique (par exemple *phrase* ou *nom*) qui « restent à définir ».

Formellement, une grammaire générative est un quadruplet $\langle T, V, S, P \rangle$ [Mas05] où :

- T est un ensemble fini de *symboles terminaux*. Les symboles terminaux sont des symboles issus de l'alphabet du langage.
- V est un ensemble fini de *symboles non-terminaux*.
- S est un symbole non-terminal ($\in V$) appelé *symbole de départ*. Dans notre exemple il s'agit de *phrase*.
- P est un ensemble fini de *règles de production* de la forme $\alpha \rightarrow \beta$ avec $\alpha \in (V \cup T)^* V (V \cup T)^*$ et $\beta \in (V \cup T)^*$

Une grammaire génératrice est dite *context-free* lorsque les règles de production ont la forme $A \rightarrow \beta$ avec $A \in V$. En d'autres mots, lorsque les règles de production sont indépendantes de tout contexte (A peut en effet toujours être dérivé en β, peu importe ce qui l'entoure).

Extensions apportées à la grammaire de Java

Afin de ne pas reproduire l'entièreté de la grammaire de Java, seules les extensions proposées sont présentées. Elle font évidemment usage de symboles définis ailleurs dans la grammaire originelle. Nous invitons donc le lecteur à se procurer la troisième édition des spécifications du

langage Java [GJSB05, 12] (disponible en libre téléchargement sur le site web de Sun[3]) et à se référer à la grammaire originelle qui y est présentée afin de comprendre au mieux les extensions proposées. La version du langage décrite dans cette troisième édition est la cinquième.

Un certain nombre de conventions sont utilisées dans la grammaire originelle. Afin d'offrir un maximum de clarté, nous avons repris les mêmes conventions, en particulier :

- Les symboles terminaux sont représentés à l'aide de caractères à `taille fixe`.
- Les symboles non-terminaux sont représentés en *italique*.
- [x] dénote zéro ou une occurrence de x.
- {x} dénote un nombre quelconque d'occurences de x (zéro ou plus).
- *CompilationUnit* est le symbole de départ.

Seule la présentation des règles de production diffère légèrement. Notre présentation fait usage de flèches pour dénoter une production. Enfin, on notera que la grammaire originelle de Java est une grammaire context-free, propriété conservée dans nos propositions d'extension.

Les unités de compilation voient l'introduction d'une nouvelle structure, les traits, définis selon les règles de production suivantes :

CompilationUnit	→	*[[Annotations]* `package` *QualifiedIdentifier* `;` *] {ImportDeclaration} {TypeDeclaration} {TraitDeclaration}*
TraitDeclaration	→	`trait` *Identifier TraitBodyDeclaration [*`implements` *TypeList]*
TraitBodyDeclaration	→	`{` *{TraitMemberDecl} TraitUseDecl {TraitMemberDecl}* `}`
TraitMemberDecl	→	`;`
	→	*MethodOrFieldDecl*
	→	`void` *Identifier VoidMethodDeclaratorRest*

L'usage de traits (au sein d'une classe ou d'un trait) est défini selon les règles de production suivantes :

TraitUseDecl	→	`uses {` *{TraitUseEntry}* `}`
TraitUseEntry	→	*QualifiedIdentifier [*`{`*GlueCode*`}`*] ;*
GlueCode	→	*{ GlueEntry ; }*
GlueEntry	→	*Identifier* `(` *TypeList* `) ->` *Identifier* `(` *TypeList* `)`
	→	*Identifier* `->` *Identifier*
	→	`^` *Identifier* `(` *TypeList* `)`

La définition du corps des classes change afin d'intégrer l'usage potentiel de traits :

ClassBody	→	`{` *{ClassBodyDeclaration} TraitUseDecl {ClassBodyDeclaration}* `}`

[3] `http://java.sun.com/docs/books/jls/`

4.4 Implémentation de tJava

Pour implémenter tJava, notre version de Java intégrant les traits dans sa syntaxe, deux solutions sont possibles :

- Modifier les mécanismes de compilation et d'exécution du langage Java afin d'y ajouter les traits comme nouvelle construction reconnue à la fois à la compilation et à l'exécution.
- Se baser sur la propriété d'aplatissement afin de convertir un programme écrit en tJava en un programme équivalent écrit en Java, compilable et exécutable par les outils traditionnels de Java.

Dans les sections suivantes, ces deux solutions seront analysées plus en détails.

4.4.1 En modifiant les mécanismes du langage Java

L'approche idéale pour ajouter les traits à états au langage Java est de leur donner une existence propre à la fois à la compilation et à l'exécution. Les traits seraient alors reconnus à la fois par le compilateur et par la machine virtuelle Java comme des entités à part entière. Illustrée à la figure 4.1, cette approche apporte plusieurs avantages [NDRS05].

FIG. 4.1 – tJava en modifiant le compilateur et le Bytecode de Java

- **Compilation séparée** Reconnaître les traits comme une entité à part entière ayant une existence propre à l'exécution permet de les compiler séparément, avant même qu'ils ne soient appliqués à une quelconque classe. Cette compilation séparée permet de détecter certaines erreurs de programmation à un stade plus précoce de la compilation sans qu'il faille nécessairement appliquer le trait à une classe. Compiler séparément un trait améliore également significativement sa réutilisabilité. En effet, l'entité compilée peut être librement partagée et réutilisée sans qu'aucune transmission (parfois non souhaitée) du code source ne soit nécessaire.

- **Débogage plus clair** Avoir une représentation des traits dans le langage assure un débogage clair et efficace. Il est en effet important que, lors de la compilation d'une classe, si une erreur est détectée, le programmeur puisse savoir si l'erreur est située dans le code de la classe elle-même ou bien dans le code d'un de ses traits qu'elle utilise. Or cette information pourrait ne plus être disponible si la notion de trait est perdue à un stade précoce de la compilation.

- **Réflexion à l'exécution** De nombreux langages de programmation, dont Java, supportent des mécanismes de *réflexion* lors de l'exécution. C'est-à-dire que les programmes qu'ils génèrent sont capables de fournir un certain nombre d'informations relatives à leur

structure. Il est par exemple possible d'énumérer l'ensemble des interfaces implémentées par une classe. Dans une optique d'ajout de traits à Java, il serait avantageux que le système de réflexion soit modifié en conséquence. On pourrait ainsi par exemple énumérer les traits qu'utilise une classe ou les méthodes apportées par un certain trait à une classe. Cette modification du système de réflexion ne peut néanmoins se faire que si les traits sont représentés à l'exécution.

- **Taille du code minimale** Si les traits sont reconnus à l'exécution comme des entités de code, il n'est pas nécessaire que le code qui compose les traits soit dupliqué au sein de toutes les classes utilisatrices du trait, de la même manière que l'héritage est un mécanisme reconnu à l'exécution et qu'il n'est pas nécessaire de dupliquer le code des classes parentes dans les classes enfants. Par conséquent, représenter les traits à l'exécution permet de minimiser la taille des exécutables.

Malheureusement, représenter les traits au sein du langage Java n'est pas sans poser une série d'inconvénients.

- **Adaptation du compilateur et de la machine virtuelle** Si l'on désire représenter les traits au sein du langage Java, un lourd travail d'ingénierie est nécessaire. À la fois le compilateur et les mécanismes d'exécution de la machine virtuelle doivent être modifiés pour tenir compte de cette nouvelle entité de code. Par exemple, le *method lookup* devra être adapté en conséquence afin que la recherche d'une méthode inclue les traits éventuels qui composent une classe.

- **Incompatibilité avec les anciennes machines virtuelles** La machine virtuelle devant être modifiée, les programmes nouvellement compilés et faisant usage de traits ne pourront être exécutés en utilisant les anciennes versions de la machine virtuelle. Dans un premier temps donc, les programmes Java usant de traits risqueraient de n'être exécutables que sur très peu de machines, le temps que les machines virtuelles soient mises à jour.

- **Exécution plus lente** La modification de la machine virtuelle afin qu'elle tienne compte des traits pourrait résulter en une vitesse d'exécution plus lente. En effet, une solution simple pour intégrer les traits est de modifier le *method lookup* [BDNW07]. Dès lors, le code d'un trait, plutôt que d'être directement accessible au sein de ses classes utilisatrices, fait l'objet d'une indirection supplémentaire lorsqu'on désire y accéder. Des efforts supplémentaires d'ingénieries doivent dès lors être déployés afin d'optimiser au maximum l'introduction des traits et de limiter les impacts sur la vitesse d'exécution.

4.4.2 En utilisant la propriété d'aplatissement

Une manière simple et élégante d'ajouter les traits à un langage de programmation existant est de se baser sur leur propriété d'aplatissement évoquée au chapitre précédent (cf. sections 3.1.1 et 3.2.5). Cette propriété stipule qu'une méthode issue d'un trait a *exactement* la même sémantique que si elle avait été directement implémentée dans la classe utilisatrice du trait. Dès lors, toute méthode issue d'un trait peut être *recopiée* au sein de ses utilisateurs, sans que la sémantique de la méthode ne soit changée.

En conséquence, les traits peuvent être facilement ajoutés à n'importe quel langage en suivant cette stratégie [NDRS05] :

1. On crée un nouveau langage L_T qui étend la syntaxe du langage L en lui ajoutant la notion de traits.
2. On écrit un *translateur* pour le langage L_T capable de recopier le contenu des traits dans leurs utilisateurs afin de produire un programme de sémantique équivalente écrit en L dans lequel toute notion de trait a été supprimée.

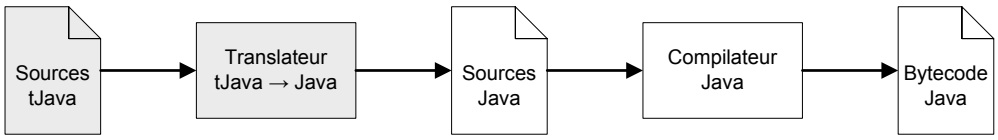

FIG. 4.2 – Translation de sources tJava en sources Java équivalentes

Cette stratégie, illustrée à la figure 4.2, est avantageuse sur plusieurs plans :

- **Simple et générique** Cette stratégie est simple et générique, elle peut être appliquée à n'importe quel langage de programmation orienté objet existant (dont Java).

- **Adaptation du compilateur et de la machine virtuelle** Le compilateur et la machine virtuelle Java ne doivent pas être modifiés, la gestion des traits est sous l'entière responsabilité du translateur.

- **Compatibilité avec les anciennes machines virtuelles** Cette stratégie permet de produire des programmes compatibles avec les anciennes machines virtuelles qui seront en mesure de les exécuter.

- **Exécution rapide** Les méthodes des traits étant recopiées au sein des classes, leur *lookup* est aussi rapide que pour les méthodes implémentées directement dans les classes. La vitesse d'exécution du programme s'en voit dès lors maximisée.

Malheureusement, cette approche pour ajouter les traits à un langage souffre d'une série d'inconvénients, en particulier tous les avantages liés à une représentation native des traits dans le langage (approche évoquée à la section précédente) tombent.

- **Compilation séparée** Les traits ne peuvent plus être compilés séparément, l'accès à leur code source est nécessaire lors du recopiage du contenu des traits dans leurs utilisateurs.

- **Débogage moins clair** Le débogage peut s'avérer déroutant pour le programmeur. Les traits étant recopiés dans leurs utilisateurs, c'est une version aplatie de la classe qui sera affichée lors du débogage, version qui peut s'avérer fort différente de ce que le programmeur a en tête.

- **Réflexion à l'exécution** Cette approche empêche toute réflexion, puisque la notion de trait disparaît totalement du programme final résultant.

- **Code dupliqué** Dernier désavantage, mais non des moindres, le code d'un trait est

dupliqué dans tous ses utilisateurs. Cette duplication peut avoir un impact non négligeable sur la taille de l'exécutable généré.

Réalisation de l'aplatissement

Lorsqu'un client (classe ou trait) utilise un trait, l'aplatissement devra être réalisé comme suit :

- On *alpha-renomme* chaque attribut du trait, pour autant que le client n'ait pas désiré y accéder. Alpha-renommer un attribut consiste à le renommer à l'aide d'un nom qui n'entre pas en conflit avec un attribut du client ou un attribut d'un autre trait utilisé par le client. On peut, pour ce faire, par exemple le préfixer avec le nom du trait.
- On renomme chaque attribut du trait dont l'accès a été désiré par le client à l'aide du nom d'accès spécifié par le client. On s'assure ensuite que ce nom n'entre pas en conflit avec des variables locales des méthodes du trait. Auquel cas on précise qu'il s'agit bien d'un attribut en le préfixant avec `this.`.
- On recopie toutes les méthodes du trait dans le client sauf celles qui :
  - ont été redéfinies au sein du client.
  - ont été exclues.
- Pour tout surnommage de méthode, on crée dans le client une méthode portant le surnom spécifié et reprenant à l'identique les arguments et le corps de la méthode surnommée.
- On ajoute à la liste des interfaces implémentées par le client les interfaces implémentées par le trait.
- Si le trait a été défini dans un autre fichier que le client, on ajoute à la liste des *imports* du client les *imports* du trait.

Lorsqu'un trait est aplati dans une classe, la question se pose de savoir quel modificateur d'accès associer aux méthodes et attributs. La stratégie suivante a été retenue :

- Les méthodes sont marquées comme *publiques*, car elles participent à l'interface de la classe.
- Les attributs privés aux traits sont marqués comme *privés* au sein de la classe, ils ne peuvent être accessibles de l'extérieur.
- Les attributs du trait auxquels la classe accède sont marqués comme *protégés*. En d'autres mots, il ne sont pas accessibles de l'extérieur mais restent accessibles des classes enfants (*a contrario* d'un attribut privé). Une autre approche aurait pu être envisagée, à savoir de marquer ces attributs comme publics. La définition formelle des traits ne précise en effet pas quelle visibilité donner aux attributs aplatis. L'approche la plus respectueuse des bonnes pratiques de programmation orientée objet a été dès lors retenue, ces dernières préconisant de ne pas exposer les attributs.

Les listings 4.15 et 4.16 illustrent un exemple d'aplatissement opéré selon les règles évoquées ci-dessus.

Listing 4.15 – Large exemple de code en tJava

```
import java.text.*;

class NavireVendable extends Navire {
  uses {
    TVendable {
      toString(NumberFormat) -> toStringFromVendable;
      ^toString(NumberFormat);
      tauxDeTaxation -> tauxDeTaxation;
    };
  }
}

trait TVendable implements IVendable {
  Double prixDeBase;
  Double tauxDeTaxation = 1.21;

  Double calculerPrixVente() {
    return prixDeBase * tauxDeTaxation;
  }

  String toString(NumberFormat nf) {
    return nf.format(calculerPrixVente());
  }

  Double getPrixDeBase() {
    return prixDeBase;
  }

  void setPrixDeBase(Double prixDeBase) {
    this.prixDeBase = prixDeBase;
  }
}

interface IVendable {
  Double calculerPrixVente();
  Double getPrixDeBase();
  void setPrixDeBase(Double prixDeBase);
}
```

Listing 4.16 – Code du listing 4.15 aplati

```
import java.text.*;

class NavireVendable extends Navire implements IVendable {
  private Double TVendable_prixDeBase;
  protected Double tauxDeTaxation = 1.21;

  public Double calculerPrixVente() {
    return TVendable_prixDeBase * tauxDeTaxation;
  }

  public String toStringFromVendable(NumberFormat nf) {
    return nf.format(calculerPrixVente());
  }

  public Double getPrixDeBase() {
    return TVendable_prixDeBase;
  }

  public void setPrixDeBase(Double prixDeBase) {
    this.TVendable_prixDeBase = prixDeBase;
  }
}

interface IVendable {
  Double calculerPrixVente();
  Double getPrixDeBase();
  void setPrixDeBase(Double prixDeBase);
}
```

Détections et gestion des erreurs

Lors de la composition de traits par le programmeur, il n'est pas impossible que celui-ci commette un certain nombre d'erreurs. Par exemple, il pourrait subsister certains conflits ou, certaines méthodes requises pourraient ne pas être implémentées. Deux approches sont envisageables pour la détection et la gestion des erreurs :

- Soit le translateur (dont la tâche est, pour rappel, de convertir le code tJava en code Java) se charge lui-même de détecter les erreurs et de les signaler au programmeur ;
- Soit le translateur ne se soucie pas d'éventuelles erreurs possibles et délègue leur détection au compilateur Java.

La deuxième approche suppose beaucoup moins de travail et de calculs de la part du translateur, néanmoins elle peut s'avérer être assez déroutante pour le programmeur. Il est en effet beaucoup plus clair pour le programmeur de se voir annoncer par le translateur le non respect d'un requis d'un trait, plutôt que de laisser le compilateur Java annoncer un appel vers une méthode non résolu. Dès lors, si possible, il est préférable qu'un maximum d'erreurs relatives à l'utilisation de traits soient détectées par le translateur.

Implémentation du translateur

L'implémentation du translateur peut s'avérer assez ardue, même lorsqu'on décide de déléguer la gestion des erreurs au compilateur. En effet, certaines analyses complexes doivent quand même être effectuées. Par exemple, lors de l'aplatissement, les attributs des traits doivent être renommés. Ce travail de refactorisation n'est pas si simple, il faut en effet détecter toutes les utilisations des attributs afin d'y répercuter le renommage. De même, Java étant un langage typé statiquement, la désignation d'une méthode (à exclure ou à surnommer) ne se fait pas sur base de son nom, mais bien sur base de sa signature. Les identificateurs de type doivent donc être résolus afin que le translateur puisse distinguer une méthode d'une autre.

On constate donc qu'une partie des analyses effectuées par le translateur sont équivalentes aux analyses effectuées par le compilateur. Afin d'éviter la duplication d'algorithmes, parfois complexes, d'analyse et de résolution, il serait dès lors opportun de voir dans quelle mesure il serait possible de modifier le compilateur originel afin d'y adjoindre la phase d'aplatissement.

Un compilateur découpe généralement la compilation en deux étapes bien distinctes [Mas05] :

- L'*analyse* qui décompose et analyse les éléments du programme source pour en construire une représentation imagée.
- La *synthèse* qui construit à partir de l'image le programme dans le langage cible.

La représentation imagée construite à la première étape est généralement un arbre, que l'on dénomme *arbre syntaxique abstrait* (en anglais *abstract syntax tree* – AST). Un exemple d'arbre syntaxique abstrait est donné à la figure 4.3. Il représente la définition d'une classe « Test » contenant un attribut entier « i » et une méthode « toString ».

Intégré dans le compilateur, le translateur pourrait utiliser avantageusement cette représen-

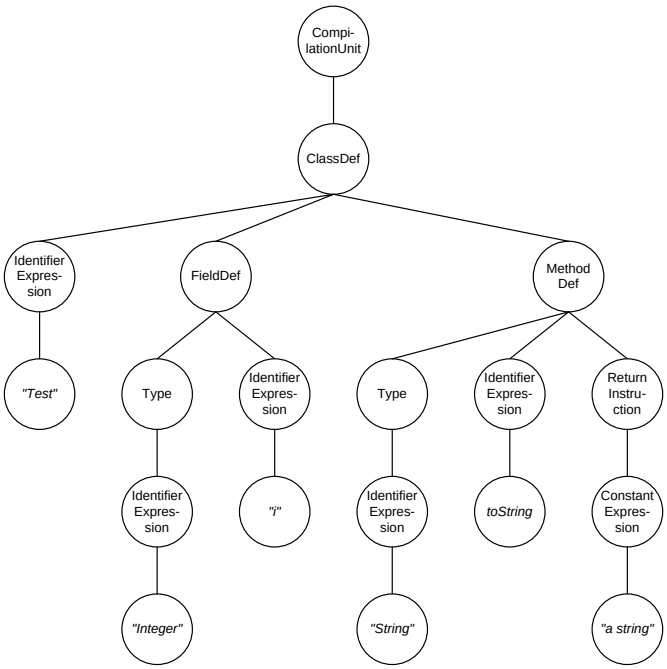

FIG. 4.3 – Un exemple d'*arbre syntaxique abstrait*

tation sous forme d'arbre. Le compilateur originel serait modifié afin qu'il construise un arbre syntaxique abstrait incluant de nouveaux nœuds pour représenter les traits. Dans une seconde phase, cet arbre serait « aplati » en un arbre de sémantique équivalente n'incluant plus la notion de trait. Par exemple, le recopiage d'une méthode d'un trait dans une classe serait symbolisé par la copie du sous-arbre représentant la méthode dans l'arbre symbolisant la classe.

Lors de cet aplatissement, les nombreux algorithmes d'analyse et de résolution du compilateur pourraient alors être utilisés par le translateur sans qu'il y ait besoin de les réécrire. L'arbre syntaxique abstrait aplati étant de même forme que les arbres générés originellement, l'étape de synthèse du compilateur resterait alors inchangée. La figure 4.4 illustre cette intégration du translateur dans le compilateur originel.

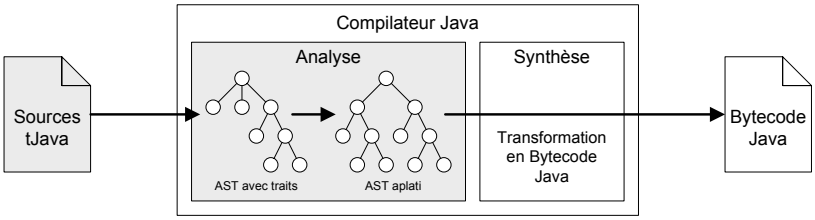

FIG. 4.4 – Intégration du translateur dans le compilateur originel

Implémenter la réécriture d'un arbre syntaxique abstrait n'étant pas une chose aisée, de nouvelles grammaires descriptives ont été imaginées afin de simplifier la définition des réécritures à opérer. C'est le cas des Rewritable Reference Attributed Grammars (ReRAGs) [EH04] introduites par Torbjörn Ekman et Görel Hedin en 2004 dont nous allons parler au chapitre suivant et qui offrent également d'autres outils pour simplifier la manipulation des arbres syntaxiques abstraits.

Chapitre 5

Les Rewritable Reference Attributed Grammars

Dans ce chapitre, on va s'intéresser aux *Rewritable Reference Attributed Grammars* (ReRAGs) introduites par Torbjörn Ekman et Görel Hedin en 2004 [EH04] et qui pourraient constituer une solution de choix pour implémenter tJava, soit la version modifiée de Java dont la grammaire comprend les traits à états.

Les Rewritable Reference Attributed Grammars sont une extension des traditionnelles canonical attributed grammars. Les ReRAGs permettent notamment aux attributs d'être des références vers d'autres noeuds de l'arbre syntaxique abstrait et de définir des règles de réécriture automatique de l'AST. Pour présenter les Rewritable Reference Attributed Grammars, nous allons procéder par étapes en rappelant d'abord ce qu'est une canonical attributed grammar (AGs) [Knu68], ensuite en introduisant les Reference Attributed Grammars (RAGs) [Hed99] et leur extension orienté objet pour finalement terminer par les Rewritable Reference Attributed Grammars (ReRAGs).

Enfin, nous terminerons ce chapitre en évaluant dans quelle mesure les Rewritable Reference Attributed Grammars peuvent aider à l'implémentation d'un compilateur Java intégrant l'aplatissement de traits.

5.1 Les canonical attributed grammars

5.1.1 Présentation

Les canonical attributed grammars (AGs), introduites par Knuth [Knu68] en 1968, permettent *d'associer un sens à un string d'un langage context-free*. Cette association de sens est réalisée en ajoutant des *attributs* aux non-terminaux (en d'autres termes aux noeuds de l'AST). Afin de définir comment sont obtenues les valeurs des attributs, on associe des *règles sémantiques* aux règles de production.

Les canonical attributed grammars distinguent deux types d'attributs associés aux noeuds de l'AST :

- Les attributs *hérités* dont la valeur ne peut dépendre que d'attributs des noeuds parents ou frères ;
- Les attributs *synthétisés* dont la valeur ne peut dépendre que d'attributs des noeuds fils, ou, si le noeud est une feuille, est donnée par l'analyseur lexical (ou syntaxique).

Imaginons que l'on veuille réaliser un additionneur, en d'autres mots un petit programme capable de réaliser une somme de nombres entiers. Utilisant les grammaires génératives traditionnelles, introduites au chapitre 4.3.2, cet additionneur pourra être décrit par les règles de production suivantes :

somme → *expression* `+` *expression*
expression → *somme*
→ `nombre`

Une somme est définie comme étant deux expressions séparées par le symbole terminal `+`. Une expression, quant à elle, est définie comment étant soit une somme, soit un symbole terminal représentant un nombre entier.

Ces règles de production permettent de définir l'ensemble des strings qui peuvent être considérés comme une somme d'entiers, mais elles ne permettent pas de décrire *comment* une somme d'entiers est calculée.

Les grammaires attribuées canoniques vont nous permettre d'exprimer *comment* calculer une somme d'entiers. D'une part, un attribut sera associé à chaque symbole non-terminal de la grammaire, représentant sa valeur, d'autre part des règles sémantiques vont définir comment sont calculées les valeurs associées aux symboles de la grammaire.

Le code du listing 5.1, dont la syntaxe est issue de *GNU Bison*[1], illustre l'usage d'attributs synthétisés et de règles sémantiques. L'exemple choisi simplifie le concept d'attributs puisque chaque noeud de l'arbre syntaxique abstrait ne peut posséder qu'un seul attribut, anonyme et de type entier. En d'autres mots, seule une valeur entière peut être associée à chaque noeud de l'AST. Cet unique attribut servira dans cet exemple à représenter la valeur sémantique des noeuds.

Listing 5.1 – Exemple de grammaire attribuée canonique

```
sum  : expr '+' expr     { $$ = $1 + $3; }
expr : sum               { $$ = $1; }
     | NUMBER            { $$ = /* Valeur numérique de NUMBER */; };
```

Dans ce code, sont exprimées les trois mêmes règles de production qui permettent de définir une somme. À ces règles de production sont associées des règles (ou actions) sémantiques définissant comment est calculée la valeur sémantique associée aux noeuds de l'AST représentant une somme ou une expression. Pour effectuer ce calcul, la première règle sémantique fait usage

[1] *GNU Bison* [4] permet de générer des parseurs pour certaines grammaires attribuées canoniques.

de la valeur sémantique des noeuds enfants représentant chacun une opérande de la somme. La deuxième règle sémantique définit simplement qu'une expression qui se dérive en somme reprend sa valeur sémantique. La troisième règle sémantique, quant à elle, a pour tâche de convertir le string du symbole non-terminal représentant le nombre en une valeur numérique. Cette conversion n'est pas explicitée dans le code d'exemple et est remplacée par un commentaire.

Une application de cette mini-grammaire est illustrée à la figure 5.1 qui représente l'arbre résultant de l'analyse de la somme $3+2$. Cet exemple illustre le mécanisme de synthétisation des attributs. On constate que les attributs synthétisés servent à faire « remonter » de l'information. On peut, de la même manière, montrer que les attributs hérités servent à faire « descendre » de l'information.

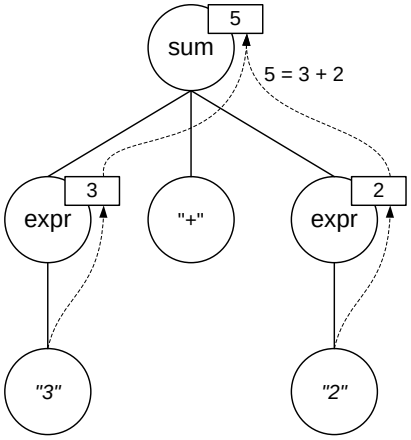

FIG. 5.1 – Arbre syntaxique abstrait de la somme $3 + 2$

5.1.2 Problèmes et limitations des canonical attributed grammars

Les canonical attributed grammars conviennent parfaitement pour résoudre les problèmes de *dépendances locales* qui suivent la structure de l'AST. Par exemple, lorsque la valeur d'un noeud dépend de la valeur des noeuds fils. Par contre, les canonical attributed grammars sont moins adaptées pour traiter les dépendances non-locales.

Imaginons par exemple un noeud d'un AST représentant l'*utilisation d'une variable*. Il est fort probable que les règles sémantiques associées à ce noeud, pour effectuer leurs calculs et traitements, aient besoin d'informations provenant du noeud représentant la *déclaration de la variable*. Afin de satisfaire ce besoin d'informations, la seule solution est de propager l'information requise du noeud de déclaration jusqu'au noeud d'utilisation sur tous les noeuds intermédiaires, ce qui n'est pas des plus efficaces. Cette situation est illustrée à la figure 5.2.

Alors que cette propagation d'informations est encore envisageable avec des langages de programmation suivant une structure simple en blocs, elle l'est beaucoup moins avec les langages modulaires ou orientés objet.

En effet, idéalement, chaque noeud de l'AST représentant l'utilisation potentielle d'une variable devrait connaître l'ensemble des déclarations de variables auxquelles il peut avoir accès,

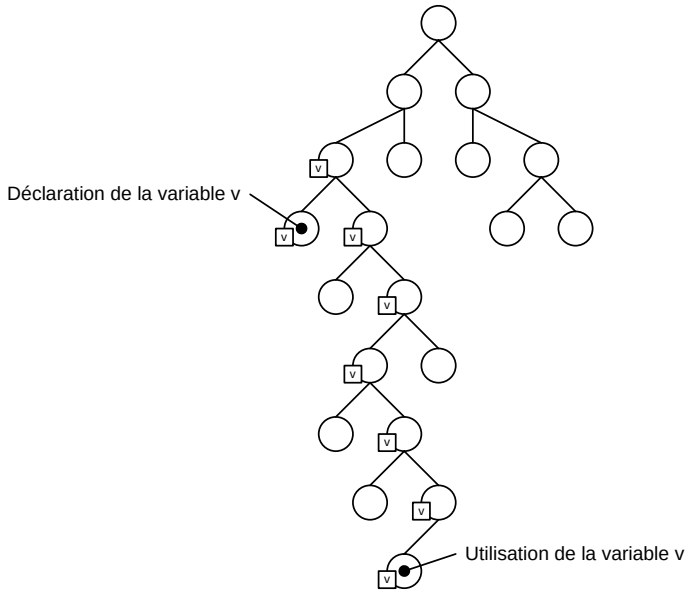

FIG. 5.2 – Propagation d'informations dans un AST

en d'autres mots, l'ensemble des variables qui lui sont « visibles ».

Un ensemble de déclarations de même portée est généralement agrégé dans un seul attribut que l'on dénomme « environnement ». La propagation de ces environnements est assez simple pour les langages ayant des règles de portées de variables triviales où les portées sont imbriquées : il suffit de propager l'information aux noeuds fils du bloc contenant les déclarations.

Par contre, pour les langages modulaires ou orientés objet, les règles de portées étant bien plus complexes, l'étendue de la propagation n'est plus aussi évidente et n'est d'ailleurs plus bornée. Avec comme fâcheuse conséquence que, bien souvent, les environnements se voient propagés dans tout l'arbre.

De plus, un noeud représentant l'utilisation potentielle d'une variable peut être soumis à plusieurs environnements conflictuels ; les méthodes de résolution de noms sont donc également bien plus complexes, complexité qui n'est généralement pas exprimable par le formalisme des canonical attributed grammars.

5.2 Les Reference Attributed Grammars

5.2.1 Présentation

Les Reference Attributed Grammars (RAGs) [Hed99, Mag03], *a contrario* des Attributed Grammars (AGs), permettent l'usage d'attributs qui sont des références vers d'autres noeuds de l'AST. De même, les attributs structurés (dictionnaires, listes, etc.) peuvent également faire usage d'attributs références. Il est également possible, avec ces grammaires, de déréférencer un attribut référence afin d'accéder aux attributs du noeud pointé. Cette approche permet donc de

propager de l'information d'un noeud référencé vers un noeud référenceur sans avoir à impliquer d'autres noeuds dans l'AST. La figure 5.3 illustre l'usage de Reference Attributed Grammars en revisitant l'exemple de déclaration et d'utilisation d'une variable présenté à la figure 5.2.

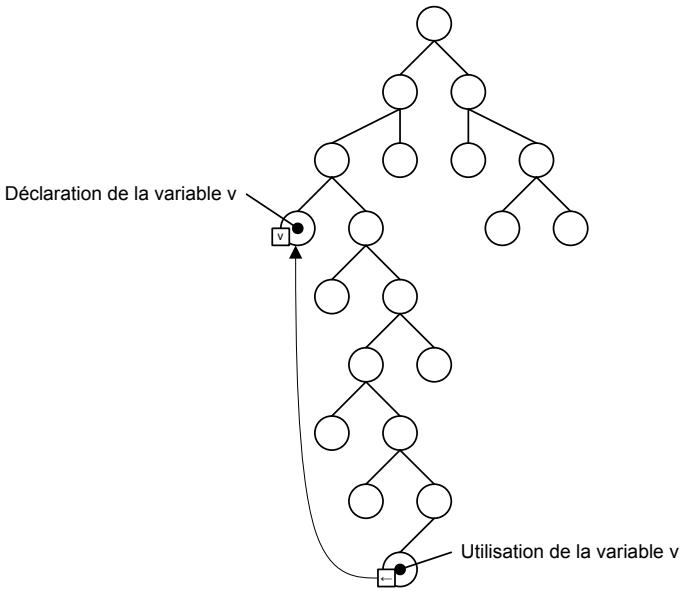

FIG. 5.3 – Référencement d'un noeud dans un AST

Les arbres syntaxiques produits par ces grammaires peuvent donc être vus comme des graphes syntaxiques, en considérant les attributs références en plus des arêtes de l'AST. Ce graphe syntaxique peut contenir des cycles. Un exemple de cycle serait un noeud A possédant un attribut référençant un noeud B, lui-même possédant un attribut référençant le noeud A. Ces cycles sont à distinguer d'une dépendance cyclique entre attributs dont la résolution peut être problématique.

5.2.2 Object-Oriented RAGs

Afin de simplifier le formalisme utilisé pour décrire les Reference Attributed Grammars et d'apporter plus de flexibilité, une vue orientée objet des RAGs a été imaginée par Görel Hedin [Hed99] où les symboles non-terminaux sont vus comme des classes, et leurs règles de production comme des sous-classes. De même, les attributs seront vus comme des fonctions virtuelles.

Hiérarchisation en classes

Dans cette optique orientée objet, les symboles non-terminaux sont vus comme des classes dont les sous-classes correspondent aux règles de productions associées ; une règle de production étant considérée à juste titre comme une « spécialisation » d'un symbole non-terminal. Nous avons donc deux niveaux dans la hiérarchie de classes.

Une nouvelle sorte de symboles non-terminaux est également introduite : les symboles non-terminaux *abstraits*. À ces symboles ne correspondent aucune règle de production. Afin de les distinguer des symboles possédant au moins une règle de production, ces derniers seront désor-

mais dénommés symboles non-terminaux *concrets*. Le rôle des symboles non-terminaux abstraits est de permettre l'enrichissement de la hiérarchie de classes. En effet, les classes associées aux symboles non-terminaux abstraits serviront de classes parentes aux classes représentant les symboles non-terminaux concrets. L'introduction de nouveaux niveaux supérieurs dans la hiérarchie de classes permet d'une part, de factoriser des comportements communs à plusieurs symboles non-terminaux ou à plusieures règles de production, d'autre part de servir comme type générique pour un attribut référence.

La hiérarchie de classes est basée sur le principe d'héritage simple : à un symbole non-terminal ou à une production ne correspond qu'un et un seul symbole non-terminal parent. À la racine de la hiérarchie se trouve un symbole non-terminal abstrait, usuellement dénommé *Any* dont tous les autres noeuds de l'AST seront une instance. Ce symbole permet de regrouper le comportement commun à tous les noeuds de l'AST.

Afin que chaque classe dans la hiérarchie puisse être identifiée par un nom, les règles de production sont nommées. Si une règle de production est la seule à dériver un symbole non-terminal, alors les classes représentant le symbole et son unique production seront fusionnées en une seule classe qui portera le nom du symbole. Dans le cas contraire, un nom explicite devra être donné à la règle de production.

Un exemple de grammaire présentée sous une forme orientée objet est illustré à la table 5.1. Les deux premières colonnes de cette table reprennent les symboles non-terminaux abstraits. La colonne suivante reprend les symboles non-terminaux concrets, soit ceux auxquels sont associées une ou plusieurs règles de production, ces dernières étant reprises dans l'avant-dernière colonne. Précisons qu'un astérisque accolé à un symbole signifie « zéro ou plusieurs occurrences du symbole ».

Enfin, la dernière colonne reprend le nom donné aux règles de production. Comme convenu ci-dessus, lorsqu'un symbole non-terminal est associé à une et une seule règle de production, une même classe est utilisée pour représenter à la fois le symbole non-terminal et la règle de production. Le nom de cette classe (en grisé dans la table) est alors implicite et correspond au nom du symbole non-terminal.

Présentée sous cette forme, la hiérarchie de classes associée à la grammaire est composée de gauche à droite. Par exemple, *Expression* est super-classe de *BoolExpression* et *ArithmExpression*.

La figure 5.4 illustre un exemple d'arbre syntaxique abstrait généré sur base de la grammaire de la table 5.1. À gauche de la figure est représentée la hiérarchie de classes. À droite de la figure est représenté l'arbre syntaxique abstrait correspondant au programme du listing 5.2. Chaque noeud de l'arbre syntaxique abstrait est une instance d'une des classes de la hiérarchie dont le type dépend de la règle de production appliquée.

Listing 5.2 – Programme dont l'AST est représenté à la figure 5.4

```
int i
i = 10
```

Non-terminaux			**Productions**	
Abstraits		**Concrets**	**Dérivation**	**Nom**
Any		*Program*	→ *Declarations Statements*	Program
		Declarations	→ *Declaration**	Declarations
		Statements	→ *Statement**	Statements
		Type	→ `int`	IntType
			→ `double`	DoubleType
		Declaration	→ *Type* `Identifier`	Declaration
		Statement	→ `Identifier =` *ArithmExpression*	AssignStatement
			→ `while` *BoolExpression* `do` *Statement*	WhileStatement
	Expression	*BoolExpression*	→ `true`	FalseExpression
			→ `false`	TrueExpression
			→ *BoolExpression* `and` *BoolExpression*	AndExpression
			→ *BoolExpression* `or` *BoolExpression*	OrExpression
			→ `not` *BoolExpression*	NotExpression
		ArithmExpression	→ `Identifier`	UseExpression
			→ `Number`	ConstExpression
			→ *ArithmExpression* `+` *ArithmExpression*	SumExpression

TAB. 5.1 – Exemple de grammaire présentée sous une forme orientée objet

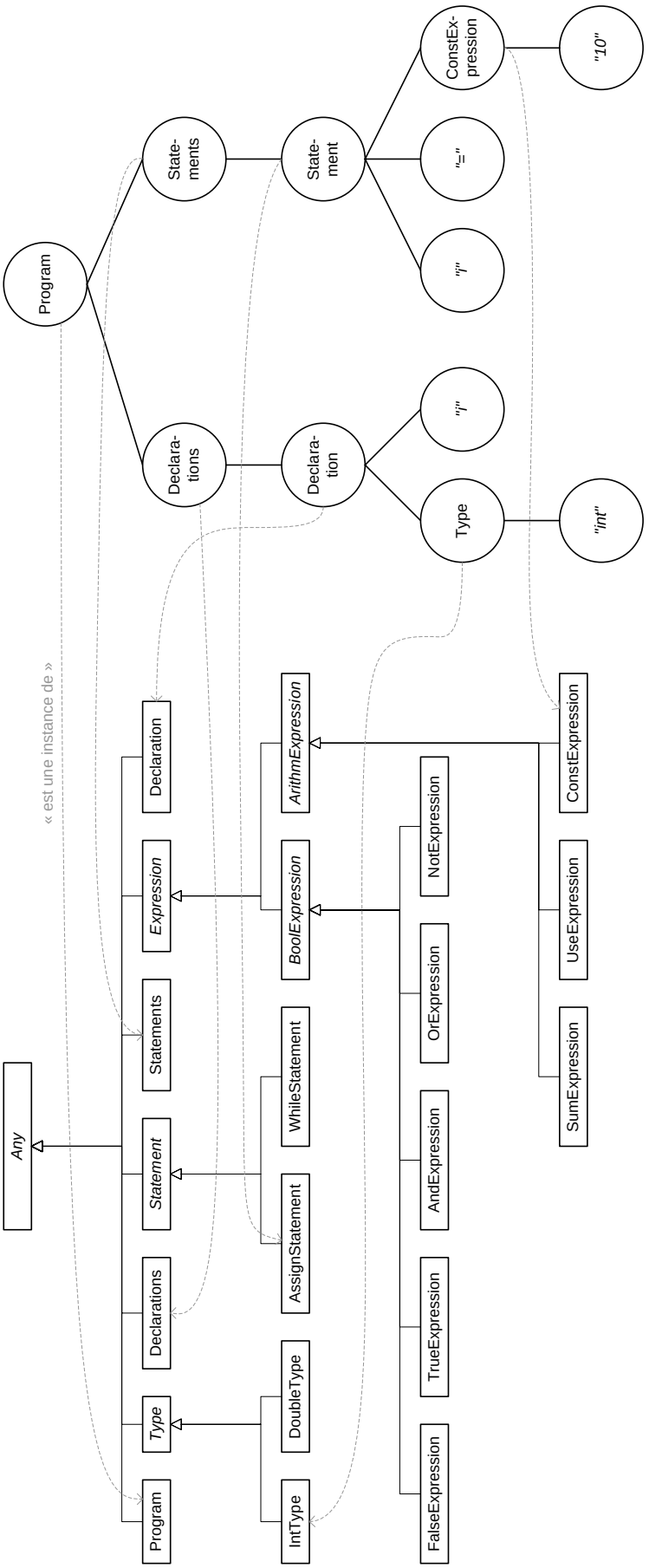

FIG. 5.4 – Exemple d'arbre syntaxique abstrait d'une grammaire orientée objet

Des fonctions virtuelles comme attributs

Avec les RAGs orientées objet, les attributs sont modélisés par des fonctions virtuelles. Par exemple, si un symbole non-terminal X possède un attribut a, ce dernier sera modélisé par une fonction virtuelle $a()$ dans une classe X. De la même manière, une règle sémantique définissant la valeur de a dans une production p sera modélisée par l'implémentation d'une fonction virtuelle dans une classe p, sous-classe de X.

Cette approche par fonctions virtuelles permet la définition de « valeurs par défaut » pour les attributs. En effet, une implémentation de la fonction $a()$ pourrait être faite dans la classe X, cette définition par défaut pouvant être *overridée* par sa sous-classe p. Il est à noter que cette démarche revient donc à permettre d'associer des règles sémantiques à des non-terminaux (concrets ou abstraits) alors que les canonical attributed grammars ne permettaient l'association de règles sémantiques qu'à des règles de production.

Les RAGs orientées objet permettent également d'associer des attributs à des règles de production, alors que les canonical attributed grammars n'associent des attributs qu'aux non-terminaux. On parlera dans ce cas *d'attributs locaux*, car propres à chaque règle de production.

Les RAGs orientées objet étendent l'approche par fonctions virtuelles en leur octroyant la possibilité d'avoir des paramètres. Cette généralisation n'était pas nécessaire avec les canonical attributed grammars. En effet, avec les AGs, le nombre d'accès à un attribut est toujours borné, donc, si des paramètres sont nécessaires, leur nombre sera également toujours borné. Ceux-ci pourront dès lors être modélisés par des attributs hérités. Malheureusement, ce n'est plus le cas avec les RAGs. L'introduction d'attributs références ne permet en effet plus de borner le nombre d'accès à un attribut. Par conséquent, le nombre de paramètres d'un attribut n'est également plus borné, ils ne peuvent donc plus être modélisés par des attributs hérités. C'est pourquoi les RAGs orientées objet introduisent la possibilité de modéliser les attributs par des fonctions virtuelles à paramètres.

5.3 Les Rewritable Reference Attributed Grammars

Les Rewritable Reference Attributed Grammars (ReRAGs) [EH04] étendent les Reference Attributed Grammars en offrant la possibilité de pouvoir réécrire dynamiquement l'AST durant l'évaluation des attributs. Alors qu'avec les Reference Attributed Grammars l'AST définitif est construit avant que les attributs ne soient évalués, il est parfois intéressant de pouvoir modifier l'AST en fonction de la valeur de certains attributs, en d'autres mots, en fonction d'un certain *contexte*.

5.3.1 Un exemple d'application

Un exemple souvent repris où l'AST gagnerait à être réécrit en fonction de la valeur de certains attributs est l'*élimination des raccourcis* d'un langage. En effet, de nombreux langages permettent l'utilisation de raccourcis afin de simplifier et de diminuer la longueur du code à écrire

par le programmeur. Dans ce cas, il est intéressant que l'AST produit ne reflète pas l'usage de ces raccourcis, mais, au contraire, qu'il reflète le code complet réellement attendu, ce qui permet de simplifier les traitements ultérieurs (comme la génération de code). Or il se peut que le passage d'un code raccourci au code complet équivalent soit dépendant d'un certain contexte.

Un exemple intéressant est la concaténation de string en Java (illustré à la figure 5.5). Celle-ci peut s'écrire à l'aide de l'opérateur binaire d'addition +. Mais en réalité, le code qui sera réellement généré pour réaliser la concaténation de deux strings est un appel à la fonction *concat* de la classe *String*. Or cette réécriture de l'AST ne peut s'opérer que si les deux opérandes de l'opérateur + sont de type *String* et donc dépend de la valeur des attributs représentant le type des opérandes.

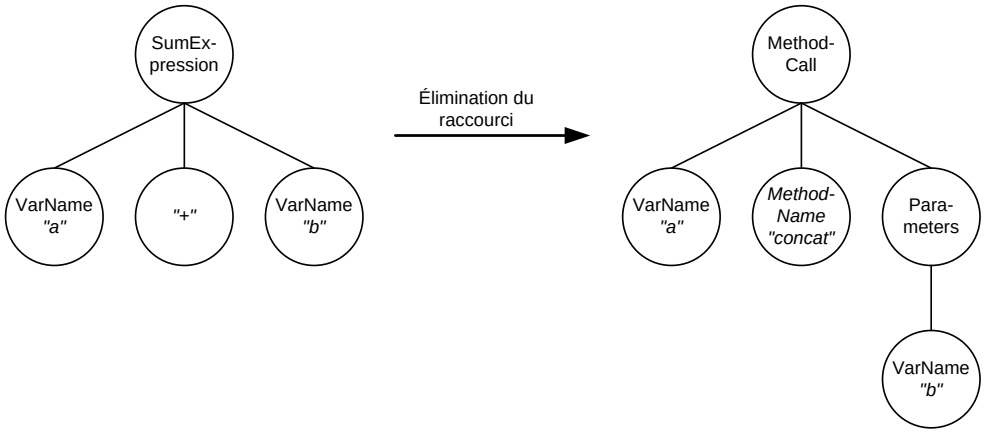

FIG. 5.5 – Exemple de réécriture d'un AST

5.4 Les ReRAGs pour implémenter les traits

Les Rewritable Reference Attributed Grammars offrent une aide substantielle pour l'écriture de compilateurs, en particulier les compilateurs de langages orienté objets, dont Java. En effet, l'introduction d'attributs *références* permet de simplifier la propagation d'informations au sein de l'arbre, facilitant ainsi le travail des concepteurs de compilateurs.

De plus, un compilateur écrit à l'aide de ReRAGs peut être facilement étendu afin d'intégrer la notion de traits. D'une part la grammaire peut être aisément complétée afin d'intégrer la syntaxe propre aux traits et générer ainsi des arbres syntaxiques abstraits dans lesquels se retrouvent les traits. D'autre part, l'aplatissement de ces AST afin d'obtenir des AST de sémantiques équivalentes dans lesquels on ne retrouve plus de traits peut également s'exprimer facilement grâce aux ReRAGs. Il suffit en effet d'ajouter une série de règles de réécriture afin que les noeuds de l'AST représentant l'usage d'un trait soient remplacés par des nœuds représentant les méthodes et attributs issus de ce trait. Ainsi la deuxième phase de compilation, à savoir la génération de Bytecode final, reste inchangée.

En conclusion, l'implémentation de tJava, notre version modifiée de Java incluant les traits à états, peut être réalisée facilement grâce aux Rewritable Reference Attributed Grammars, pour

autant que l'on dispose des sources d'un compilateur Java déjà écrit à l'aide de ReRAGs.

Chapitre 6

Choix des outils

Pour implémenter tJava, soit la version modifiée de Java dont la grammaire comprend les traits à états, plusieurs outils peuvent être utilisés, chacun offrant des possibilités différentes avec leurs avantages et inconvénients. Afin de choisir au mieux les outils et méthodes qui serviront à l'implémentation de tJava, une méthodologie de choix doit être développée. Nous établirons dans un premier temps une liste de critères que devront respecter les outils candidats. Deux types de critères sont à distinguer : les critères indispensables et les critères souhaitables. Les critères indispensables étant les critères qui, s'ils ne sont pas respectés, ne permettront pas d'implémenter efficacement tJava, les critères souhaitables étant quant à eux des critères plus de « confort ». Ensuite, pour chaque outil, nous analyserons leur respect des critères afin de finalement retenir l'outil convenant le mieux.

6.1 Critères de choix

6.1.1 Capacité de production de l'AST Java

Les outils éligibles pour implémenter tJava devront être capables de générer un arbre syntaxique abstrait complet représentant un programme Java donné. Cet AST devra comprendre une série d'informations résultant d'une première phase d'analyse du programme. En particulier, doivent se trouver dans l'arbre, les résultats des résolutions de noms et de types. En effet, ces informations seront cruciales afin de pouvoir par exemple :

- Lister les *signatures* des méthodes définies dans une classe ;
- Détecter au sein des méthodes les appels non-résolus vers des méthodes appartenant à soi (méthode appelée sur *this*) ou appartenant à la classe parente (méthode appelée sur *super*) ;
- Connaître la hiérarchie de classes ;
- Lister les champs définis dans une classe.
- Détecter les accès aux champs définis dans une classe.

6.1.2 Capacité de modification de l'AST Java

Les outils devront être également capables de modifier l'arbre syntaxique abstrait Java produit et de générer le code Java correspondant à l'arbre modifié. On devra par exemple pouvoir ajouter des méthodes à une classe, renommer ou ajouter un attribut à une classe, etc.

6.1.3 Capacité de dériver l'outil

L'outil devra être facilement extensible afin qu'il puisse reconnaître les nouveaux éléments de syntaxe introduits par tJava. Il devra par conséquent être capable de produire des AST légèrement différents dans lesquels on retrouvera la notion de traits.

Néanmoins, cette fonctionnalité n'est pas indispensable. En effet, les nouveaux mots-clés introduits peuvent éventuellement être traités séparément par un *préprocesseur* dans une phase précédente à la création de l'AST Java. Un code entièrement conforme à la grammaire de base de Java duquel aura été retiré le code « parasite » propre aux traits pourrait alors ensuite être présenté au générateur de l'AST. Cependant, si cette approche est retenue, il est alors indispensable que l'outil expose ses fonctionnalités d'analyse afin que le préprocesseur puisse, par exemple, résoudre noms et types.

6.2 Outils analysés

6.2.1 Eclipse Java Development Tools (JDT)

Les *outils de développement Java* (JDT) [10] d'Eclipse sont un ensemble de plugins Eclipse[1] [6] destinés à enrichir la plateforme Eclipse avec un environnement de développement intégré. Parmi ces plugins, il en est un à retenir car il pourrait convenir pour l'implémentation des traits : *JDT Core*. Ce plugin définit en effet un modèle complet pour représenter un programme Java ainsi qu'un ensemble d'APIs permettant de manipuler ce modèle. Ce modèle répond en tout point aux deux premiers critères de sélection énoncés ci-dessus : il permet à la fois de générer l'AST d'un code Java, de modifier cet AST et ensuite de produire le code correspondant au nouvel AST.

Malheureusement, la dérivation du modèle présenté par JDT afin d'ajouter la reconnaissance de nouveaux éléments du langage ne peut se faire qu'au prix de lourdes modifications et d'une réécriture complète de certains composants. En atteste la complexité de l'implémentation d'AJDT [1], un modèle dérivé de JDT ajoutant le support d'AspectJ.

On retiendra également la maturité de cet outil. JDT bénéficie en effet déjà de plusieurs années de développement et est largement utilisé par la communauté Eclipse, ce qui est un avantage certain par rapport à d'autres outils beaucoup moins éprouvés.

[1] Eclipse est un environnement de développement intégré destiné originellement au langage Java.

6.2.2 OpenJava

OpenJava [13, TCKI00, Tat99] est un outil créé par Michiaki Tatsubori permettant, d'une part, d'analyser du code Java et d'en construire l'AST correspondant, d'autre part d'étendre le langage Java en lui ajoutant de nouvelles fonctionnalités par le biais d'*annotations*. OpenJava permet également de modifier très facilement l'analyseur syntaxique sous-jacent en lui ajoutant de nouvelles règles de syntaxe, règles qui peuvent être la combinaison de règles existantes.

Cet outil pourrait donc *a priori* permettre de développer une version de Java modifiée pour prendre en compte la nouvelle syntaxe apportée par les traits. Malheureusement, OpenJava offre un degré de liberté fortement limité ; les annotations ne peuvent être placées qu'à certains endroits bien précis et les règles d'analyse syntaxique personnalisées ne peuvent être appliquées qu'après une annotation. En l'occurrence, la syntaxe introduite par tJava pour ajouter les traits à états à Java n'est pas réalisable telle quelle à l'aide d'OpenJava. Il n'est par exemple pas envisageable de définir un nouveau mot-clé « trait » destiné à englober les déclarations d'attributs et de méthodes.

Il est également à regretter le manque de vitalité de ce projet pourtant prometteur. Le développement et la maintenance de cet outil semblent s'être arrêtés depuis quelques années déjà, les deux dernières mises à jour datant de 2005 et 2003. OpenJava ne supporte d'ailleurs que le code Java dans sa version 1.1. De plus, l'aide fournie avec l'outil est assez succincte et la communauté gravitant autour de ce projet – dont l'aide aurait pu être appréciable – est quasi inexistante.

6.2.3 JavaCC et JJTree

Java Compiler Compiler (JavaCC) [9] est un générateur d'analyseurs syntaxiques écrit en Java. En d'autres mots, il permet de générer, à partir d'une spécification de grammaire context-free, un programme capable de reconnaître si une chaîne fournie en entrée appartient au langage défini par la grammaire. Des actions sémantiques peuvent également être associées aux règles de production. Elles pourront être par exemple utilisées afin de décrire la génération de l'arbre syntaxique abstrait.

JavaCC est généralement utilisé de concert avec JJTree, un préprocesseur permettant d'automatiser l'écriture d'actions sémantiques utilisées à la création d'arbres syntaxiques abstraits.

Malheureusement, JavaCC n'inclut aucune fonctionnalité d'analyse ou de résolution de types ou de noms. Ces analyses sont en effet propres à chaque langage de programmation et JavaCC se veut le plus générique possible puisqu'il permet la création de compilateurs pour de nombreuses grammaires.

6.2.4 Le Java programming language compiler (javac)

Fin de l'année 2006, Sun Microsystems, l'entreprise à l'origine du langage Java, a publié les sources de son compilateur *javac* [8] qui permet de compiler un programme écrit en Java et d'en

produire le Bytecode correspondant. Le compilateur javac a été publié sous licence GPL[2], une licence qui autorise la réutilisation et la modification du code source.

Il est évident que ce compilateur respecte l'ensemble des critères de choix imposés. L'accès aux sources étant offert, d'une part, celui-ci peut être modifié afin de reconnaître les nouveaux éléments de syntaxe, d'autre part, les fonctions d'analyse et de résolution, incluses dans le compilateur, peuvent être réutilisées.

Néanmoins, le compilateur n'ayant été mis à disposition de tous que très récemment, la communauté l'entourant n'est encore que très restreinte et balbutiante. Peu de documentation et d'aide sont dès lors disponibles, et il peut être extrêmement ardu de se plonger dans un code assez conséquent dont on ne connaît rien de l'architecture et de la structure interne.

6.2.5 JastAdd II et le JastAdd Extensible Java Compiler

JastAdd, tout comme JavaCC, est un générateur d'analyseurs syntaxiques développé par Torbjörn Ekman et Görel Hedin [7, HM03]. Cependant, alors que JavaCC ne reconnaît que les canonical attributed grammars (cf. section 5.1), JastAdd (dans sa deuxième version) implémente les concepts introduits par les *Rewritable Reference Attributed Grammars* (cf. section 5.3). En particulier, JastAdd II reconnaît la définition :

- d'attributs synthétisés ou hérités (cf. section 5.1) ;
- d'attributs pouvant être des références vers des nœuds de l'AST (cf. section 5.2) ;
- d'attributs circulaires (cf. section 5.2) ;
- d'une vue orientée objet de la grammaire (cf. section 5.2.2) ;
- de règles de réécriture des nœuds de l'AST (cf. section 5.3).

En plus de toutes ces fonctionnalités, JastAdd II permet également une découpe en modules d'*aspects* du compilateur. En d'autres mots, les différentes classes composant la hiérarchie de classes de la grammaire peuvent voir certaines de leurs fonctionnalités regroupées dans des aspects, en reprenant les concepts de la programmation orientée aspect (cf. section 4.2).

Enfin, notons que JastAdd II base son fonctionnement sur d'autres outils. Il impose en effet que lui soit adjoint un analyseur syntaxique afin de lui déléguer certains tâches. Fort avantageusement, JastAdd II est compatible avec la plupart des analyseurs syntaxiques existant (dont JavaCC [9] et Beaver [3]).

Seul, JastAdd II ne répond pas aux critères imposés au début de ce chapitre. Tout comme JavaCC, cet outil est indépendant du langage cible du compilateur généré et ne reconnaît aucune des caractéristiques de Java. Cependant, les auteurs de JastAdd II ont également réalisé avec succès un compilateur Java (le JastAdd Extensible Java Compiler) en utilisant leur outil. Ce compilateur est libre de droit[3] et ses sources peuvent être librement consultées ou modifiées.

Par conséquent, JastAdd II et le JastAdd Extensible Java Compiler constituent, ensemble, une solution de choix pour intégrer les traits au langage Java. D'une part nous avons vu que

[2] `http://www.gnu.org/licenses/gpl.html`

[3] Publié sous la licence BSD (`http://www.opensource.org/licenses/bsd-license.php`)

les Rewritable Reference Attributed Grammars peuvent être avantageusement utilisées pour implémenter les traits, d'autre part le JastAdd Extensible Java Compiler permet de baser notre implémentation sur un compilateur existant et d'en reprendre les fonctionnalités d'analyse.

Enfin, notons que le JastAdd Extensible Java Compiler est un compilateur à destination de la version 1.4 du langage Java. Or la version actuelle du langage est la 6[4]. Heureusement, des mises à jour du JastAdd Extensible Java Compiler sont en cours d'implémentation afin de prendre en charge les dernières versions du langage Java (le compilateur pour la version 5 du langage est actuellement en version de test).

6.3 Récapitulatif

La table 6.1 récapitule les résultats de l'analyse des outils sélectionnés. Différents critères sont repris auxquels sont associés des scores pour chaque outil. Un coche signifie que le critère est peu respecté par l'outil, deux coches signifient une concordance totale avec l'exigence requise. Un tiret signifie, quant à lui, que le critère n'est pas rempli par l'outil (ce qui peut résulter en son exclusion). La dernière colonne reprend le total de coches obtenus par chacun des outils.

JDT possède de nombreuses qualités, malheureusement aucune fonction d'analyse n'a été trouvée. De plus, JDT reste difficilement dérivable du fait d'un code source très confus et peu documenté.

OpenJava possède également de nombreuses qualités, malheureusement cet outil ne permet pas d'être étendu comme on le souhaite. En particulier, la syntaxe de tJava n'est pas réalisable. Cet outil est donc exclu.

JavaCC est un outil de trop bas niveau : il ne possède aucune fonction d'analyse, indispensable à l'implémentation de tJava. Cet outil n'est donc également pas retenu.

javac et JastAdd II (ce dernier utilisé de concert avec le JastAdd Extensible Java Compiler) partagent de nombreuses similitudes, et donc de nombreux avantages. Étant déjà des compilateurs, ce sont les seuls outils qui respectent l'entièreté des critères définis. Néanmoins, JastAdd II semble plus facilement dérivable grâce à son utilisation des Rewritable Reference Attributed Grammars et à sa découpe en aspects. Les deux outils souffrent cependant tout de même d'un manque de documentation.

En conclusion, c'est l'outil JastAdd II qui récolte le plus de coches et que nous allons retenir pour implémenter les traits à états en Java.

[4]La différence de versions peut paraître énorme, mais en fait le système de numérotation a changé. La version succédant à la version 1.4 est la version 5, bien qu'elle aurait dû être numérotée 1.5.

	Production de l'AST	*Modification de l'AST*	*Dérivation de l'outil*	*Fonctions d'analyse*	*Vivacité du projet*	*Documentation*	Total
JDT	✓ ✓	✓ ✓	✓	-	✓ ✓	✓	8
OpenJava	✓ ✓	✓ ✓	-	✓ ✓	✓	✓ ✓	9
JavaCC	✓ ✓	✓	✓ ✓	-	✓ ✓	✓	8
javac	✓ ✓	✓ ✓	✓	✓ ✓	✓	✓	9
JastAdd II	✓ ✓	✓ ✓	✓ ✓	✓ ✓	✓	✓	10

TAB. 6.1 – Récapitulatif de l'analyse des outils

Chapitre 7

Validation

Dans ce chapitre nous allons expliquer notre tentative d'implémentation de tJava sur base du JastAdd Extensible Java Compiler. Dans un premier temps, nous allons décrire les principaux concepts de l'outil JastAdd II (qui a servi à l'implémentation du compilateur Java précité) ainsi que son utilisation. Nous décrirons ensuite l'architecture du JastAdd Extensible Java Compiler et analyserons comment le modifier afin d'y inclure la syntaxe et la sémantique de tJava.

7.1 JastAdd II

Dans cette section, nous allons brièvement introduire l'outil JastAdd II qui permet de réaliser des compilateurs sur base de Rewritable Reference Attributed Grammars. Nous passerons en revue ses différents concepts fondateurs et son utilisation par le biais de quelques exemples.

7.1.1 Une hiérarchie de classes

JastAdd II permet de définir une vue orientée objet de la grammaire (cf. section 5.2.2) appelée *grammaire abstraite* ou *syntaxe abstraite*. En d'autres mots, les symboles non-terminaux sont vus comme des classes abstraites, et leurs règles de production comme des sous-classes concrètes. Une nouvelle sorte de symboles non-terminaux est également introduite, les symboles non-terminaux abstraits qui n'interviennent dans aucune règle de production. Ceux-ci sont utilisés comme classes parentes des classes représentant les symboles non-terminaux concrets et les règles de production. Leur rôle est d'enrichir la hiérarchie de classes et de permettre la factorisation de comportements communs à plusieurs types de nœuds de l'AST.

Un exemple de grammaire abstraite est illustré au listing 7.1. La hiérarchie définie est une partie de la hiérarchie associée à l'exemple de grammaire proposé au chapitre sur les Rewritable Reference Attributed Grammars (cf. table 5.1). Le mot-clé « abstract » permet de déclarer une classe abstraite. En l'absence de ce mot-clé, la classe est définie comme concrète. Les deux-points (**:**) sont utilisés pour définir les relations d'héritage (i.e. le parent d'une classe). Par exemple, « TrueExpression » est une classe enfant de « BoolExpression ». L'ensemble des définitions de

la grammaire doit être placé dans un fichier d'extension `.ast`.

Listing 7.1 – Exemple de définition d'une hiérarchie de classes

```
abstract statement;
WhileStatement : statement;

abstract Expression;
abstract BoolExpression : Expression;
FalseExpression : BoolExpression;
TrueExpression : BoolExpression;
```

Au sommet de la hiérarchie de classes se trouve la classe « ASTNode », prédéfinie par l'outil JastAdd II. Toute classe de la hiérarchie possède cette classe comme ancêtre. Cette classe possède diverses méthodes permettant de parcourir l'AST. En particulier, on y retrouve des méthodes pour énumérer les nœuds enfants et récupérer le nœud parent.

JastAdd II permet une définition plus fine de la grammaire abstraite. Il est en effet possible de préciser les types et la cardinalité des nœuds enfants. Par exemple, si l'on reprend la grammaire de la table 5.1, on constate que les nœuds de type « WhileStatement » posséderont exactement deux nœuds (non-terminaux) de type respectivement « BoolExpression » et « Statement ». Ces informations sont définies à l'aide de l'opérateur `::=`. Une illustration est donnée au listing 7.2.

Listing 7.2 – Exemple de définition du type des nœuds enfants

```
abstract Statement;
WhileStatement : Statement ::= Statement BoolExpression;
```

Les cardinalités des nœuds enfants peuvent s'exprimer de diverses manières. Des exemples sont donnés au listing 7.3.

Listing 7.3 – Exemple de définition de la cardinalité des nœuds enfants

```
A := B*;    // A possède un nombre quelconque de fils de type B
C := [D];   // C possède un ou zéro fils de type D
E;          // E ne possède pas de fils
```

Les nœuds enfants peuvent également être nommés. Un nœud père *doit* d'ailleurs nommer ses enfants lorsque plusieurs d'entre eux sont de même type. Un exemple est donné au listing 7.4.

D'autres constructions, moins triviales, sont encore proposées par JastAdd. Pour ces constructions, nous renvoyons le lecteur au manuel de JastAdd consultable sur le site web de l'outil [7].

Une fois la grammaire abstraite définie, JastAdd va générer automatiquement le code des classes correspondantes. Ce code sera fonction des directives données. Par exemple, si un nœud de type A déclare posséder un enfant de type B nommé C, alors une classe nommée A sera générée dans laquelle sera implémentée une méthode *getC* retournant un objet de type B. Des

Listing 7.4 – Exemple de nommage des nœuds enfants

```
A := First:B Second:B;    // A possède deux noeuds enfants de type B, l'un
                          // nommé First, l'autre nommé Second
```

constructeurs *ad hoc* seront également générés afin de pouvoir associer aux nœuds leurs enfants. Un exemple plus complet est donné aux listings 7.5 et 7.6. Les types `Opt` et `List` du constructeur sont des types un peu particuliers, prédéfinis par JastAdd. Ils permettent de représenter respectivement un nœud optionnel et une liste de nœuds de même type. Nous renvoyons le lecteur à la documentation de JastAdd [7] pour plus de détails.

Listing 7.5 – Grammaire abstraite correspondant aux classes du listing 7.6

```
abstract A;
E: A ::= A [B] C* First:D Second:D;
```

Listing 7.6 – Classes générées à partir de la grammaire abstraite du listing 7.6

```
class ASTNode extends Object {
   int getNumChild();
   ASTNode getChild(int);
   ASTNode getParent();
}

abstract class A extends ASTNode { }

class E extends A {
   A getA();
   boolean hasB();
   B getB();
   int getNumC();
   C getC(int);
   D getFirst();
   D getSecond();
   // Constructeur
   E(A, Opt, List, D, D);
}
```

7.1.2 Construction de l'AST

Bien que JastAdd permette de décrire et de générer les classes qui serviront à instancier les nœuds de l'AST, il ne se charge pas de construire l'AST en lui-même. Un analyseur syntaxique tiers doit donc être utilisé. Celui-ci est laissé au choix du programmeur, pour autant que cet outil puisse faire usage des classes Java générées par JastAdd.

L'analyseur syntaxique tiers aura donc deux tâches : d'une part, vérifier qu'un programme

donné appartient bien au langage (sur base de sa grammaire), d'autre part, construire l'AST correspondant à l'aide d'actions sémantiques associées aux règles de production. Un exemple est donné à la table 7.1. Une action sémantique est associée à la règle de production définissant un « WhileStmt ». Cette action sémantique a pour tâche de construire le nœud représentant une instruction « While » en générant un nouvel objet de type « WhileStmt » (dont la classe a été générée par JastAdd). Deux paramètres sont passés à la construction de cet objet, représentant les deux nœuds fils.

Production	**Action sémantique**
WhileStmt → `while` *BoolExpr* `do` *Stmt*	**return new** WhileStmt(BoolExpr, Stmt);

TAB. 7.1 – Exemple d'action sémantique participant à la construction de l'AST

7.1.3 Aspects

Seules, les classes générées par JastAdd ne permettent pas grand-chose, si ce n'est définir un type pour les nœuds de l'AST et à caractériser leurs descendants directs. Afin de pouvoir ajouter du comportement aux classes générées (par exemple des fonctions d'analyse ou de résolution), JastAdd introduit une notion d'*aspects*, similaire à celle introduite par la programmation orientée aspect (cf. section 4.2). Les aspects servent à regrouper du code qui sera automatiquement ajouté aux classes correspondantes par JastAdd.

Deux types d'aspects sont à distinguer :

- Les aspects *déclaratifs* (placés dans un fichier d'extension `.jrag`) dans lesquels peuvent être définis des éléments propres aux Rewritable Reference Attributed Grammars, à savoir des attributs (hérités ou synthétisés), des équations et des règles de réécriture. Ces éléments seront étudiés aux sections suivantes.
- Les aspects *impératifs* (placés dans un fichier d'extension `.jadd`) dans lesquels peuvent être définis des champs et des méthodes ordinaires écrites en Java.

Le rôle de JastAdd est double : il convertit dans un premier temps les aspects déclaratifs en aspects impératifs (en transformant les attributs, équations et règles de réécriture en code Java ordinaire), ensuite il recopie le code des différents aspects dans les classes correspondantes.

Un exemple d'aspect impératif est donné au listing 7.7. Une méthode et un champ sont associés à la classe « WhileStmt ».

Listing 7.7 – Exemple d'aspect impératif

```
aspect A {
  public void WhileStmt.aMethod() { /* Code de la méthode */ }
  private Integer WhileStmt.aField = 0;
}
```

7.1.4 Attributs et équations

Pour rappel, les Rewritable Reference Attributed Grammars étendent indirectement les canonical attributed grammars. Elles en reprennent donc les concepts introduits, notamment les notions d'attributs hérités et synthétisés (cf. section 5.1).

La définition d'attributs se fait au sein d'aspects déclaratifs, lesquels peuvent également contenir des définitions d'équations. Les équations permettent de définir la valeur des attributs. Les attributs peuvent également être paramétrisés. Un exemple d'aspect déclaratif est donné au listing 7.8.

Listing 7.8 – Exemple d'aspect déclaratif

```
aspect A {
  syn int WhileStmt.aSynthesizedAttribute();
  inh boolean WhileStmt.aInheritedAttribute();
  syn int WhileStmt.aParameterizedAttribute(int i);

  eq WhileStmt.aSynthesizedAttribute() = /* Expression Java */;
  eq WhileStmt.aParameterizedAttribute(int i) = /* Expression Java */;
}
```

Les attributs hérités et synthétisés sont sensiblement différents :

- Lorsqu'une classe C définit un attribut *synthétisé*, si C est concrète, elle doit également définir une équation correspondante pour cet attribut. Par contre, si elle est abstraite, toutes les sous-classes concrètes devront définir l'équation correspondante à l'attribut.
- Lorsqu'une classe C définit un attribut *hérité*, toute classe représentant un nœud de l'AST dont un des fils est de type C doit définir une équation correspondant à cet attribut.

Afin de pouvoir accéder aux valeurs des différents attributs, des méthodes correspondantes (portant les mêmes noms que les attributs) sont automatiquement générées par JastAdd.

7.1.5 Règles de réécriture

Des règles de réécriture peuvent également être définies dans les aspects déclaratifs. Elles permettent le remplacement d'un nœud par un autre sous certaines conditions. Les règles de réécriture sont totalement transparentes et exécutées dès que l'AST est traversé. En d'autres mots, un code parcourant l'arbre syntaxique abstrait ne verra que la version finale de l'arbre. Un exemple est donné au listing 7.9.

7.1.6 Récapitulatif de l'architecture

La figure 7.1 récapitule le processus de conception d'un compilateur avec JastAdd II. JastAdd est utilisé afin de générer les classes de l'AST. JavaCC est utilisé pour la construction de l'AST,

Listing 7.9 – Exemple de règle de réécriture

```
rewrite WhileStmt {
   when ( /* Conditon */ )
   to ForStmt {
      /* Du code */
      return aInstanceOfForStmt;
   }
}
```

à partir des classes générées par JastAdd. Enfin, les classes auxiliaires sont utilisées pour implémenter les points d'entrée du compilateur. Leur rôle sera essentiellement d'activer les processus de construction et d'analyse de l'AST ainsi que la génération de code.

7.2 Modification du JastAdd Extensible Java Compiler

Le JastAdd Extensible Java Compiler est un compilateur Java dont l'implémentation a été réalisée à l'aide de JastAdd II.

7.2.1 Front-end et Back-end

L'architecture du compilateur a été scindée en deux grandes composantes. D'une part le *front-end*, chargé de construire l'AST et d'y ajouter de la sémantique, d'autre part le *back-end* chargé d'optimiser l'AST et de générer le Bytecode. Les différentes étapes de compilation sont représentées à la figure 7.2.

Pour la réalisation de notre compilateur tJava, seul le Front-end sera modifié. En effet, c'est durant cette phase que l'AST sera « aplati » à l'aide de règles de réécriture.

Il est à noter que le Front-end peut fonctionner indépendamment du Back-end ; on peut en effet restreindre la compilation à une simple création de l'AST sémantique. Un module greffé au Front-end permet alors d'imprimer le code Java correspondant à cet AST sémantique.

7.2.2 Analyse syntaxique

Bien que la documentation de JastAdd II cite souvent JavaCC [9] comme exemple d'analyseur syntaxique pour la construction de l'AST, ce n'est pas cet outil qui a été retenu par les concepteurs du compilateur Java qui lui ont préféré Beaver [3].

Beaver, comme de nombreux autres analyseurs syntaxiques, délègue les tâches d'analyses lexicales à un autre outil, laissé au choix du programmeur. L'analyse lexicale a pour objectif de décomposer le programme source en une suite de *lexèmes* (nom de variables, mots-clés, parenthèses, opérateurs, etc.). L'analyseur lexical (scanner) retenu par les concepteurs du compilateur

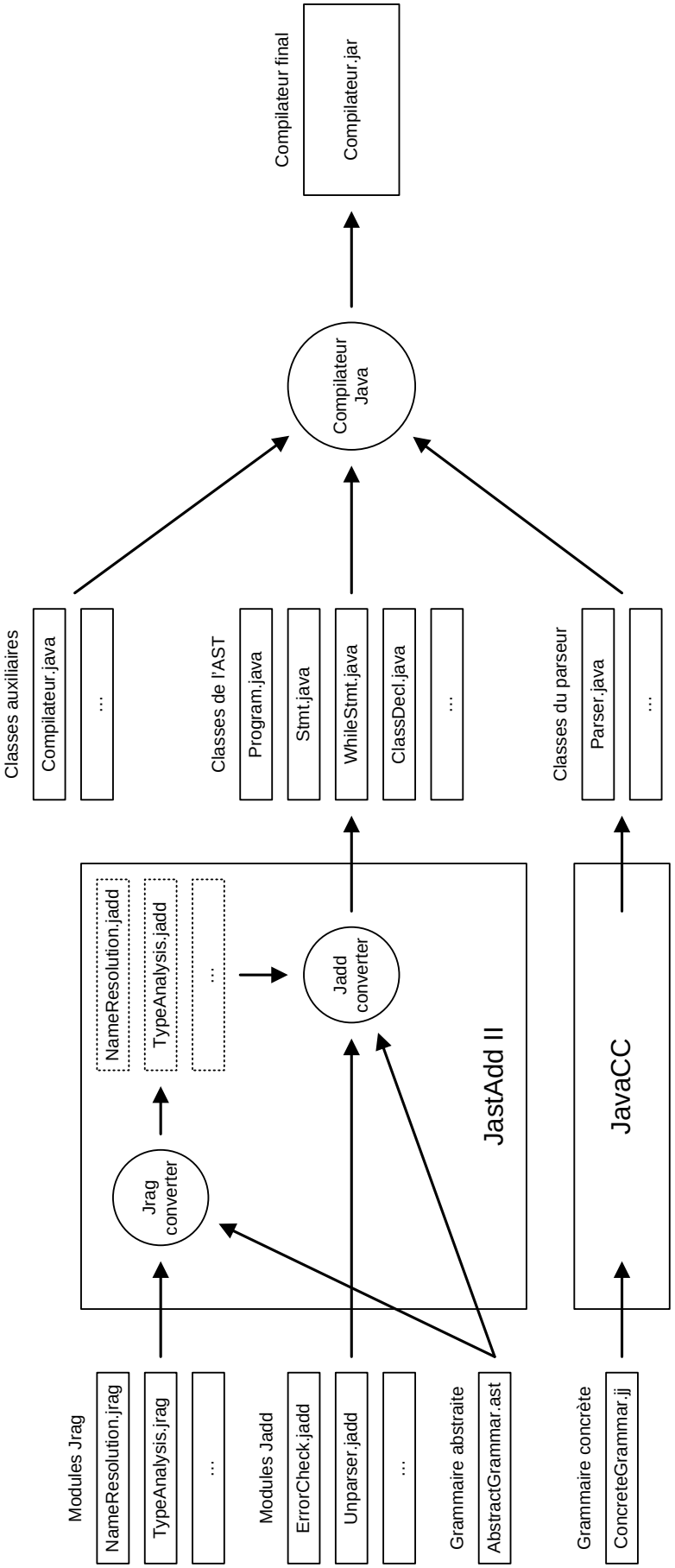

FIG. 7.1 – Génération d'un compilateur avec JastAdd II

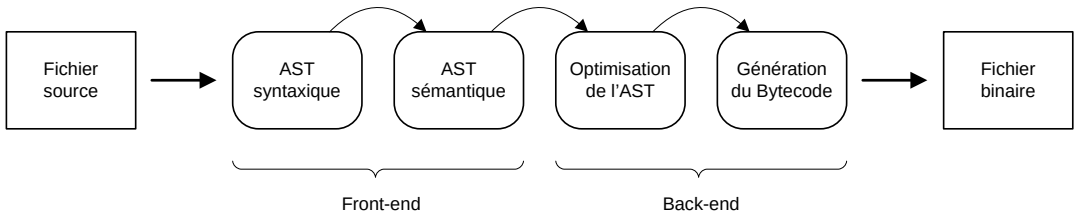

FIG. 7.2 – Étapes de compilation du JastAdd Extensible Java Compiler

est JFlex [11].

Un premier fichier doit donc être modifié, `java14.flex`, afin de rajouter les nouveaux lexèmes introduits par la syntaxe de tJava (par exemple `trait`, `uses`, `->`, etc.).

La modification suivante à opérer concerne la grammaire concrète, i.e. la grammaire utilisée par Beaver pour réaliser son analyse syntaxique. Les nouvelles règles de production propres à tJava doivent y être ajoutées. Les spécifications de la grammaire concrète se trouvent dans le fichier `java14.parser`. Après examen de ce fichier, il s'avère que sa syntaxe diffère légèrement de la syntaxe de Beaver. Les concepteurs du compilateur ont, semble-t-il, élaboré une syntaxe simplifiée qu'un outil *ad hoc* est chargé de transformer en syntaxe reconnue par Beaver. Si les simplifications opérées ne sont pas trop difficiles à comprendre, un minimum de documentation aurait tout de même été appréciable.

Lors de l'analyse syntaxique opérée par Beaver, doit être construit l'arbre syntaxique abstrait correspondant. Des actions sémantiques doivent donc être ajoutées afin de créer les nœuds de cet AST. Les classes génératrices de ces nœuds sont définies dans le fichier `java.ast` qui contient la définition de la grammaire abstraite. Il n'est malheureusement pas évident de savoir si on peut réutiliser les classes déjà définies pour composer nos propres classes. La sémantique et le comportement associé aux classes déjà définies sont-ils adaptés à nos besoins ? Nos ajouts ne risquent-ils pas de générer des effets de bord ? Il est donc important d'analyser auparavant les différents modules déclaratifs et impératifs associés aux classes prédéfinies.

7.2.3 Analyse sémantique

La réutilisation et la compréhension des modules de l'analyse sémantique sont nettement plus problématiques du fait de la complexité et de la longueur du code des différent modules déclaratifs et impératifs. Par exemple, les résolutions de noms et de types font intervenir de nombreux modules de plusieurs centaines de lignes de code chacun.

Malheureusement, le manque de documentation, déjà pointé lors du choix de l'outil (cf. section 6.3), est bien plus problématique que prévu. Seuls quelques aspects bien précis du compilateur sont décrits dans de trop rares articles [NIEH04, Ekm06]. De plus, le code source manque de commentaires et de documentation. Sans de plus amples informations, il est extrêmement difficile de dégager les composantes importantes de chaque module et de comprendre pleinement l'architecture de l'analyse sémantique.

Par exemple, quels sont les rôles des modules « LookupMethod », « NameCheck », « QualifiedNames » ou « ResolveAmbiguousNames » ? Comment, sans comprendre les fonctions remplies par les centaines d'attributs de ces modules, réaliser un lookup à partir d'une signature de méthode (composée d'un nom et d'un ensemble de types) ? Il aurait été opportun qu'une documentation plus fournie explique les concepts généraux de chacun des modules et leurs liens avec les autres modules.

7.2.4 Conclusion

Ce manque de documentation ne nous aura dès lors pas permis de finaliser l'implémentation de notre compilateur tJava. Néanmoins, de nombreuses bases ont été posées que de travaux futurs pourront reprendre. Nous espérons que cela donnera naissance à une implémentation prochaine.

Malgré cette insuffisance de documentation, JastAdd II n'en reste pas moins un outil de choix pour implémenter les traits. Fondé sur les Rewritable Reference Attributed Grammars, dont il reprend les concepts novateurs, JastAdd amène sans nul doute une meilleure découpe de l'architecture des compilateurs qu'il permet de construire. Ce n'est certainement pas l'outil en lui-même qui est à remettre en question, mais bien le manque de documentation de l'implémentation réalisée avec cet outil. Carence, qui nous l'espérons, sera comblée dans le futur par les auteurs à l'origine de l'implémentation de ce compilateur Java.

Chapitre 8

Conclusion

La première partie de ce travail a été consacrée à une présentation des mécanismes de réutilisation de code de l'orienté objet, que nous avons voulu la plus pédagogique et la plus accessible possible. Il nous est dès lors apparu essentiel de commencer par introduire les concepts de la programmation orientée objet, paradigme de programmation que les traits se proposent par ailleurs d'enrichir. Pour ce faire, nous avons procédé par petits incréments, afin d'éviter de noyer le lecteur novice dans une pléthore de concepts. Nous avons également enrichi notre propos à l'aide de nombreux exemples (illustrés de schémas) afin de faciliter la compréhension de ces concepts.

Nous nous sommes ensuite attelés à décrire deux mécanismes de réutilisation de code : l'*héritage* et les *mixins*, lesquels ont aussi bénéficié d'un large travail pédagogique. Ces deux mécanismes ont ensuite été soumis à évaluation. De nombreux problèmes et limitations ont été dégagés, dont le programmeur n'a pas toujours spécialement conscience car parfois peu évidents à exprimer. Divers exemples ont dès lors également été imaginés afin d'accompagner notre critique.

Nous avons ensuite présenté la notion de traits, selon leurs deux variantes, *sans état* et *à états*. Une attention toute particulière a été accordée à la clarté de nos explications, ponctuées elles aussi d'exemples. Pour chacune des deux variantes des traits, nous avons procédé à leur évaluation afin de les confronter aux limitations et problèmes relevés lors de notre étude des méthodes existantes. Si, dans l'ensemble, les traits apportent une solution à ces limitations, nous avons néanmoins mis en évidence le fait qu'ils ne permettent pas de résoudre les conflits sémantiques (au contraire de certaines méthodes de réutilisation de code étudiées au premier chapitre). Mis à part ce petit bémol, nous avons pu montré que les traits constituent une réelle avancée dans le domaine de la réutilisation de code. Simples, intuitifs et efficaces, les traits sont certainement destinés à un avenir prometteur.

Notre présentation des traits nous aura permis d'évoquer leur intégration dans un système de typage existant. Plusieurs solutions ont été proposées, approfondissant celles trouvées dans la littérature scientifique. Nous avons enfin conclu notre exposé sur les traits par un tour d'horizon des travaux actuels et futurs sur ce sujet, en présentant brièvement les différentes intégrations des traits à des langages existants réalisées à ce jour, ainsi que diverses pistes pour des recherches futures.

Une fois les traits introduits, nous avons étudié leur intégration au langage de programmation Java. Après une brève introduction de ce langage, une première approche a été présentée, soit une émulation des traits basée sur la programmation orientée aspect et imaginée par Simon Denier de l'École des Mines de Nantes. Nous avons ensuite évalué cette approche pour en soulever un certain nombre de limitations. En particulier, la résolution de conflits, qui, lorsqu'elle n'est pas impossible, s'avère assez complexe.

Face à ce constat, nous avons privilégié une intégration native des traits à états dans le langage Java. Nous avons dès lors proposé diverses extensions à la syntaxe originelle de Java, que nous avons formalisées en modifiant la grammaire officielle de Java. Un nouveau langage était né : tJava qui étend le langage Java en lui incluant la notion de traits.

L'étape suivante de notre démarche a été d'étudier les différentes méthodes pour implémenter tJava. Deux méthodes ont été avancées : une modification des mécanismes internes du langage Java et de sa machine virtuelle et l'usage de la propriété d'aplatissement des traits afin de convertir un code écrit en tJava en un code Java de sémantique équivalente. Universelle et plus accessible, c'est la deuxième méthode que nous avons retenue.

Le chapitre suivant a été consacré à l'étude des Rewritable Reference Attributed Grammars. Ce nouveau type de grammaires propose en effet divers mécanismes simplifiant l'implémentation de compilateurs. Toujours dans le respect de notre démarche pédagogique, une introduction progressive a été retenue, faisant usage d'exemples choisis. Le concept précurseur de canonical attributed grammars a dès lors été rappelé avant d'introduire les Reference Attributed Grammars pour enfin présenter les Rewritable Reference Attributed Grammars. Nous avons terminé cette étude en analysant dans quelle mesure ces grammaires pourraient aider à l'implémentation d'un compilateur pour tJava basé sur la propriété d'aplatissement. Nous avons en particulier dégagé l'intérêt que présentent les règles de réécriture pour réaliser l'aplatissement de traits.

Avant de se lancer dans une quelconque implémentation, il convient de choisir les bons outils. C'est pourquoi le chapitre suivant a été consacré à l'évaluation, sur base de critères bien définis, d'une série d'outils permettant l'implémentation d'un compilateur pour tJava. L'outil JastAdd II, couplé au JastAdd Extensible Java Compiler, a retenu notre attention, notamment en raison de son implémentation des concepts introduits par les Rewritable Reference Attributed Grammars.

Le dernier chapitre a été consacré à la description de notre tentative d'implémentation d'un compilateur tJava. Nous avons, dans un premier temps, décrit les possibilités et présenté l'utilisation de l'outil JastAdd. Nous avons également présenté le JastAdd Extensible Java Compiler, compilateur Java écrit par les auteurs de JastAdd que nous nous proposions de modifier afin d'y intégrer les traits. Nos changements ont tout d'abord porté sur l'analyse lexicale pour ensuite se concentrer sur l'analyse syntaxique afin d'y intégrer la nouvelle syntaxe de tJava. Nous avons ensuite tenté de modifier la grammaire abstraite et les modules d'analyse du compilateur. Cette tâche s'est révélée autrement plus complexe, en cause un manque crucial de documentation rendant les méandres de cette implémentation d'un compilateur Java difficilement accessibles. Par conséquent, l'implémentation d'un compilateur tJava basée sur JastAdd Extensible Java Compiler ne pourra être finalisée que lorsqu'une documentation plus fournie sera publiée. Laquelle, aux dires des auteurs de JastAdd, est prévue pour un avenir proche.

Enfin, en guise de conclusion, nous espérons sincèrement que les traits sont destinés à un avenir prometteur et, pourquoi pas, qu'ils soient un jour reconnus comme mécanisme standard de l'orienté objet, tel le principe d'héritage. Car nous pensons que les traits offrent de réels bénéfices pour la conception de logiciels et qu'ils contribuent pleinement à répondre à la question initialement posée « Comment concevoir rapidement des logiciels de qualité ? ».

Bibliographie

[Arm06] Deborah J. Armstrong : The quarks of object-oriented development. *Commun. ACM*, 49(2):123–128, 2006.

[BC90] Gilad Bracha et William Cook : Mixin-based inheritance. *In Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, octobre 1990.

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford et P. Tucker Withington : A monotonic superclass linearization for dylan. *In Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 69–82, octobre 1996.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz et Roel Wuyts : Stateful traits. *In Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 de *LNCS*, pages 66–90. Springer, 2007.

[BMN03] Robert Biddle, Angela Martin et James Noble : No name: just notes on software reuse. *SIGPLAN Not.*, 38(12):76–96, 2003.

[Can82] H. I. Cannon : Flavors: A non-hierarchical approach to object-oriented programming. Rapport technique, Symbolics Inc., 1982.

[Car97] Luca Cardelli : Type systems. *In The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[CDM+05] Yaofei Chen, Rose Dios, Ali Mili, Lan Wu et Kefei Wang : An empirical study of programming language trends. *IEEE Software*, 22(3):72–78, 2005.

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

[CW85] Luca Cardelli et Peter Wegner : On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, décembre 1985.

[Del04] Claude Delannoy : *Programmer en Java.* Eyrolles, troisième édition, 2004.

[Den04] Simon Denier : Traits programming with AspectJ. *In* Pierre Cointe, éditeur : *Actes de la Première Journée Francophone sur le Développement du Logiciel par Aspects (JFDLPA'04)*, pages 62–78, Paris, France, septembre 2004. Available at http://www.emn.fr/x-info/obasco/events/jfdlpa04/.

[DT88] S. Danforth et Chris Tomlinson : Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, mars 1988.

[EFB01] Tzilla Elrad, Robert E. Filman et Atef Bader : Aspect-oriented programming. *cacm*, 44(10), octobre 2001.

[EH04] Torbjörn Ekman et Görel Hedin : Rewritable reference attributed grammars. *In* Martin Odersky, éditeur : *ECOOP*, volume 3086 de *Lecture Notes in Computer Science*, pages 144–169. Springer, 2004.

[Ekm06] Torbjörn Ekman : *Extensible Compiler Construction*. Thèse de doctorat, Lund University, Dept. of Computer Science, Lund, Sweden, 2006. Dissertation 25.

[GJSB05] James Gosling, Bill Joy, Guy Steele et Gilad Bracha : *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[Gro05] Object Management Group : *Unified Modeling Language: Superstructure Specification, version 2.0*. Object Management Group, août 2005.

[Hed99] Gorel Hedin : Reference Attributed Grammars. *In* D. Parigot et M. Mernik, éditeurs : *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153–172, Amsterdam, The Netherlands, 1999. INRIA rocquencourt.

[HM03] Görel Hedin et Eva Magnusson : Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm et William G. Griswold : An overview of AspectJ. *In Proceeding ECOOP 2001*, numéro 2072 de LNCS, pages 327–353. Springer Verlag, 2001.

[Knu68] Donald E. Knuth : Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Mag03] Eva Magnusson : Applications and extensions of reference attributed grammars, juin 2003.

[Mas05] Thierry Massart : *Théorie des langages et de la compilation (volumes 1 et 2)*. Presses Universitaires de Bruxelles, a.s.b.l, première édition, 2005.

[Moo86] David A. Moon : Object-oriented programming with Flavors. *In Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, novembre 1986.

[NDRS05] Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart et Nathanael Schärli : Adding Traits to (statically typed) languages. Technical Report IAM-05-006, Institut für Informatik, Universität Bern, Switzerland, décembre 2005.

[NIEH04] Anders Nilsson, Anders Ive, Torbjörn Ekman et Görel Hedin : Implementing java compilers using rerags. *Nordic J. of Computing*, 11(3):213–234, 2004.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman et Matthias Zenger : An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon et Matthias Zenger : An overview of the Scala programming language. Technical Report 001, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2006.

[Rei05] Stefan Reichhart : A prototype of Traits for C#. Informatikprojekt, University of Bern, 2005.

[SD05] Charles Smith et Sophia Drossopoulou : Chai: Typed traits in java. *In 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, pages 543–576, Glasgow, Scotland, July 2005.

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

[Smi04] Charles Smith : Typed traits in java. Mémoire de D.E.A., Imperial College of Science Technology and Medicine (University of London), 2004.

[Tat99] M. Tatsubori : An extension mechanism for the java language, 1999.

[TCKI00] Michiaki Tatsubori, Shigeru Chiba, Marc-Oliver Killijian et Kozo Itano : OpenJava: A class-based macro system for Java. *In* Walter Cazzola, Robert J. Stroud et Francesco Tisato, éditeurs : *Reflection and Software Engineering*, LNCS 1826, pages 119–135. Springer-Verlag, juillet 2000.

[Weg90] Peter Wegner : Concepts and paradigms of object-oriented programming. *ACM OOPS Messenger*, 1(1):7–87, août 1990.

Webographie

[1] AspectJ Development Tools home page. `http://www.eclipse.org/ajdt/`.

[2] AspectJ home page. `http://eclipse.org/aspectj/`.

[3] Beaver home page. `http://beaver.sourceforge.net/`.

[4] GNU Bison home page. `http://www.gnu.org/software/bison/`.

[5] Chai Prototype home page. `http://chai-t.sourceforge/`.

[6] Eclipse home page. `http://www.eclipse.org/`.

[7] JastAdd home page. `http://jastadd.cs.lth.se/web/`.

[8] The Java compiler (javac) group. `http://openjdk.java.net/groups/compiler/`.

[9] Java Compiler Compiler home page. `https://javacc.dev.java.net/`.

[10] Eclipse Java Development Tools home page. `http://www.eclipse.org/jdt/`.

[11] JFlex home page. `http://www.jflex.de/`.

[12] The Java Language Specification. `http://java.sun.com/docs/books/jls/`.

[13] OpenJava home page. `http://www.csg.is.titech.ac.jp/openjava/`.