# Mise en symbiose des traits et des classboxes : Application à l'expression des collaborations

**Florian Minjat**[1,2] — **Alexandre Bergel**[2]
— **Pierre Cointe**[1] — **Stéphane Ducasse**[2]

[1]*Projet Obasco (EMN - INRIA)*
*Ecole des Mines de Nantes/LINA*
*{florian.minjat, pierre.cointe}@emn.fr*

[2]*Software Composition Group*
*University of Berne, Switzerland*
*{bergel, ducasse}@iam.unibe.ch*

ABSTRACT. *The trait model is complementary to class inheritance and allows collections of methods to be reused by several classes. The classbox model allows a collection of classes to be locally extended with variables and/or methods addition. This paper describes a symbiosis of these two models: classes can be locally extended by using a trait. It is illustrated by an efficient implementation of the collaboration model where a collaboration is represented by a a classbox and a role by a trait.*

RÉSUMÉ. *Le modèle des traits propose un complément à l'héritage des classes permettant la réutilisation d'une collection de méthodes par différentes classes. Le modèle des classboxes permet l'extension locale d'une collection de classes par l'ajout de variables et/ou de méthodes d'instance. Cet article présente une symbiose de ces deux modèles : permettre l'extension locale d'une classe par l'utilisation d'un trait et au delà, proposer une réalisation du modèle des collaborations pour lequel une collaboration est assimilée à un classbox et un rôle à un trait.*

KEYWORDS: *open class, trait, classbox, module, mixin, reuse, extensibility, composition, introduction, collaboration, role*

MOTS-CLÉS : *classe ouverte, trait, classbox, module, mixin, réutilisation, extensibilité, composition, introduction, transversalité, collaboration, rôle*

## 1. Introduction

Dans l'esprit des mixins, le modèle des traits [DUC 05] permet de définir, sous forme de collection de méthodes, des fragments de comportement pouvant être réutilisés par plusieurs classes indépendamment de leur relation d'héritage. En plus de sa définition habituelle, une classe peut se composer de plusieurs traits, charge lui étant laissée de résoudre explicitement les conflits grâce à un ensemble d'opérateurs prédéfinis. Néanmoins ce modèle ne se prête pas à l'évolution non anticipée d'une application.

Dans l'esprit des modules, le modèle des classboxes [BER 05] permet d'étendre une collection de classes en introduisant un espace de nommage et un mécanisme de contrôle de visibilité. Ces extensions ne sont visibles que dans un espace bien délimité : le classbox qui les définit. De telles extensions de classe permettent de spécifier des modifications non anticipées d'une application [BER 05]. Une limitation du modèle est de ne pas permettre la réutilisation d'extension de classes.

Très clairement ces deux améliorations du modèle traditionnel des classes vont séparément dans le sens d'une meilleure réutilisabilité et adaptabilité des applications. L'idée de cet article est de combiner les deux approches. Concrètement nous montrons d'abord comment étendre le modèle des classboxes pour permettre l'introduction de traits. Nous présentons ensuite une mise en œuvre des collaborations basée sur ce couplage trait/classbox, celle-ci est l'occasion de revenir sur les architectures en couche [SMA 98] et la modularisation de propriétés transverses.

Cet article est ainsi construit : la section 2 présente le modèle des traits et l'illustre par un exemple de représentation d'objets géométriques. La section 3 introduit le modèle des classboxes et montre sa mise en œuvre dans la vérification des liens dans des pages html. La section 4 développe la symbiose trait/classbox. La section 5 rappelle les principaux travaux sur les collaborations et revisite une application de parcours de graphes. La section 6 évalue rapidement notre solution avant de conclure.

## 2. Traits

Les *traits* sont une extension de l'héritage simple dans la lignée des *mixins* mais qui résolvent les problèmes de leur mise en œuvre [BRA 90, DUC 05]. Un trait est un groupe de méthodes dont certaines sont *fournies* par le trait et d'autres *requises*. Un trait ne peut contenir d'état et les méthodes fournies par les traits ne peuvent pas accéder directement à une variable d'instance. Une classe est alors composée de traits, variables d'instances et de *méthodes de colle* dont le but est de résoudre les conflits ou de satisfaire les méthodes requises par les traits qui la composent. La sémantique d'une telle classe est définie par les règles suivantes : (1) les méthodes d'instance sont prioritaires sur les méthodes définies dans un trait; (2) une méthode non masquée dans un trait a la même sémantique que si elle était définie dans la classe qui utilise le trait; (3) tous les traits qui composent une classe ont la même priorité. Un conflit apparaît dès que deux traits ou plus fournissent des méthodes de même nom. Un conflit doit

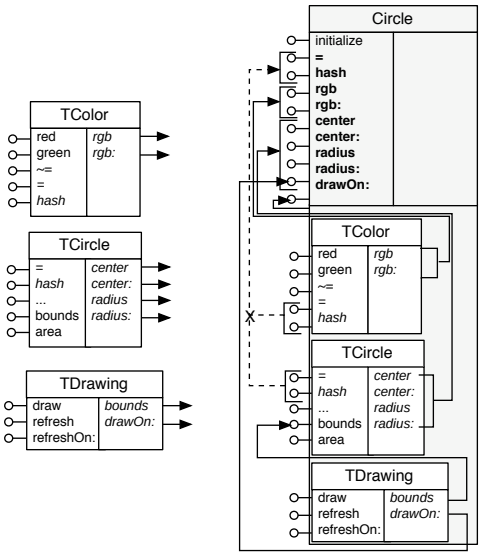


**Figure 1.** *La classe Circle est composée des trois traits TColor,TCircle, et TDrawing. Un trait est défini par son nom, une liste de méthodes définies (située à gauche sur la figure) et une liste de méthodes requises (situées à droite). Un conflit est résolu en redéfinissant les méthodes hash et =.*

être résolu *explicitement* en définissant une méthode de même nom dans la classe ou en excluant une des méthodes utilisées. De plus un mécanisme d'alias permet d'accéder si nécessaire aux méthodes en conflit. Les règles de composition des traits ont été traitées dans d'autres travaux [DUC 05].

**Exemple : les objets géométriques**. Un objet graphique (cercle, rectangle ...) peut être décomposé en trois fragments de comportement réutilisables : sa couleur, sa géométrie et son rendu par l'application. La figure 1 illustre[1] ce principe dans le cas d'un cercle. La géométrie est exprimée par le trait TCircle, la gestion de la couleur est fournie par le trait TColor et les méthodes d'affichage par le trait TDrawing. Le trait TCircle a pour prérequis les méthodes center, center:, radius, et radius:. Il fournit des méthodes telles que bounds, area ou =. Le trait TDrawing a pour pré requis les méthodes drawOn: et bounds et fournit les méthodes draw, refresh, et refreshOn:. Et enfin, TColor nécessite les méthodes rgb, rgb: et offre les méthodes liées aux traitements des couleurs.

1. Le modèle étant implanté en Squeak, le code présenté dans cet article utilise une syntaxe Smalltalk.

La classe Circle est une sous-classe de la classe Object. Elle définit trois variables d'instance center, radius et rgb et de leurs accesseurs respectifs. Elle est composée de trois traits TDrawing, TCircle et TColor. Le conflit entre les méthodes hash et = des traits TCircle et TColor est résolu par la suppression des entrées hash et = dans les deux traits et la création de nouvelles entrées correspondant aux anciennes (colorHash, circleHash, . . . ).

Syntaxiquement cette composition est explicitée au moment de la définition de la classe Circle. Le nouveau mot clef uses: est introduit pour définir la composition des trois traits vus précédemment.

```
Object subclass: #Circle
 instanceVariableNames: 'center radius rgb'
 uses: {    TDrawing +
            TCircle @ {#circleHash → #hash . #circleEqual: → #=} +
            TColor @ {#colorHash → #hash . #colorEqual: → #=} }
```

L'opérateur + compose deux traits pour en réaliser l'*union*, l'opérateur - permet de retirer une entrée dans un trait (non illustré ici) et enfin l'opérateur → copie une entrée sous un nouveau nom (par exemple : m1 → m2 définit m1 comme nouveau nom pour la méthode m2).

La résolution des conflits se fait explicitement par la (re)définition au niveau de la classe des méthodes hash et = en utilisant les alias des méthodes hash et = des traits.

```
Circle≫hash
 ↑self circleHash
   bitXor: self colorHash
```

```
Circle≫= anObject
 ↑(self circleEqual: anObject)
   and: [self colorEqual: anObject]
```

## 3. Classboxes

Le modèle des *classboxes* est un système de modules permettant, grâce à un mécanisme de contrôle de visibilité, de limiter l'impact d'une extension apportée à une application. Plusieurs versions d'une classe peuvent coexister dans un même système, chaque version correspondant à une vue de cette classe [BER 05, BER 04]. Un classbox se compose d'un ensemble de définitions de classes, de méthodes, de variables et d'imports de classes. Un classbox peut définir des méthodes ou des variables d'instance sur n'importe quelle classe visible (*i.e.*, définie ou importée) dans ce classbox .

Définir une variable ou une méthode sur une classe importée constitue une *extension de classe*. Une extension de classe est soit (i) l'ajout ou la redéfinition de méthode, soit (ii) l'ajout de variable sur une classe définie dans un autre classbox . L'ensemble des modifications apportées par un classbox sur un système a une visibilité limitée au classbox qui les définit. En dehors d'un classbox, une extension ou

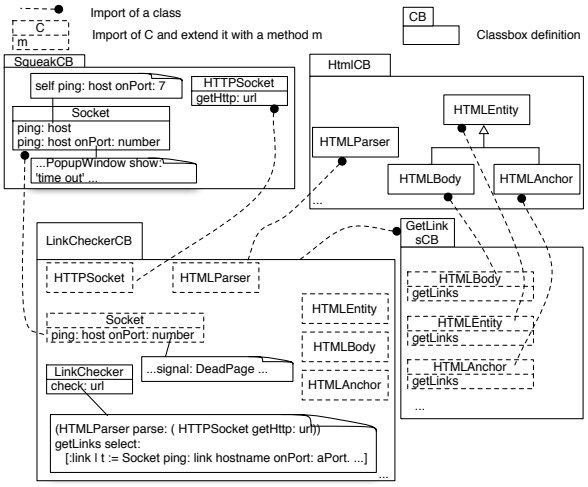


**Figure 2.** *Un détecteur de liens HTML morts comme exemple d'utilisation des classboxes.*

une définition n'est pas visible : aucune invocation de méthode ou référence n'est possible. Une définition n'est visible que dans le classbox qui la définit et dans les classboxes important les classes étendues ou définies.

Une propriété importante de ce modèle est que *les définitions locales prennent précédence sur les définitions importées*. En cas de redéfinition d'une méthode, tous les appels à partir du classbox définissant cette méthode utilisent la nouvelle implémentation [BER 05].

**Exemple : détection de liens morts d'une page web**. Cet exemple d'application réalisé avec les classboxes illustre l'utilisation des extensions de classes ainsi que la visibilité des changements. Un vérificateur de liens est une application acceptant une URL et retournant la liste des liens morts contenus dans la page désignée par l'url.

L'idée est de télécharger une page, puis de générer un arbre de syntaxe abstraite (AST) à partir de celle-ci et d'appliquer un algorithme récursif sur cet arbre pour récupérer l'ensemble des liens (figure 2). Pour chaque lien, l'existence du serveur correspondant est vérifiée en effectuant un *ping* sur celui-ci.

Le classbox HtmlCB définit l'architecture HTML contenant un parseur (HTMLParser) et une hiérarchie de nœuds (HTMLAnchor, HTMLEntity, ...). Le classbox GetLinksCB contient l'algorithme récursif utilisé pour récupérer l'ensemble des liens contenus dans une page : chaque nœud de la hiérarchie est importé depuis HtmlCB et étendu avec une méthode getLinks.

Le classbox LinkCheckerCB contient le cœur de l'application en définissant la classe LinkChecker. Cette classe possède la méthode check: url représentant le point

d'entrée de l'application. Tout d'abord l'application télécharge le contenu de l'url avec HTTPSocket getHttp: url. Un AST est ensuite construit en utilisant la méthode parse: de la classe HTMLParser. L'ensemble des liens contenus dans cette page est obtenu en envoyant le message getLinks à la racine de l'arbre. Pour chacun des liens obtenus, l'existence du site est vérifié en faisant un *ping* sur le serveur correspondant.

L'implémentation de la méthode Socket»ping: host onPort: number définie dans le classbox SqueakCB affiche le résultat du *ping* dans une fenêtre (PopupWindow). Ceci ne convient pas à notre application, car en cas de *timeout*, une exception doit être levée (signal: DeadPage) : LinkCheckerCB redéfinit donc la méthode ping: host onPort: number. C'est cette nouvelle implémentation qui est utilisée par notre application et ce même lorsque la méthode ping: host onPort: number est invoquée indirectement par la méthode ping: host.

```
SqueakCB:
    Socket≫ping: host onPort: number
      ...PopupWindow show: 'timeout'...
    Socket≫ping: host
      self ping: host onPort: 7

    "Si erreur, affiche un message"
    Socket new ping: 'www.zork.com'
```

```
LinkCheckerCB:
    import: Socket from: SqueakCB
    Socket≫ping: host onPort: number
      ...PopupWindow show: 'timeout'...

    "Si erreur, leve une exception"
    Socket new ping: 'www.zork.com'
```

C'est un point important du modèle : les définitions locales ont précédence sur les définitions importées. Appeler la méthode ping: host depuis LinkCheckerCB provoque l'exécution de la méthode ping: host onPort: number redéfinie dans ce classbox. Alors que depuis SqueakCB la définition originale est utilisée.

## 4. De la symbiose entre traits et classboxes

Alors que les traits structurent les classes, les classboxes contrôlent les modifications d'un système par un mécanisme de visibilité. Nous présentons maintenant un modèle alliant les traits et les classboxes. Un classbox peut maintenant étendre une classe avec des traits tout en contrôlant la visibilité de l'extension. Ainsi un ensemble de traits peut être appliqué sur un ensemble de classes pour mettre en place des collaborations en forme de couches [SMA 98] comme présenté en section 5.

### 4.1. *Import de trait*

D'une façon analogue aux classes, un trait est défini dans un et un seul classbox. Un classbox définit un espace de nommage permettant d'éviter des conflits liés aux noms : plusieurs traits dotés d'un même nom peuvent être simultanément présents dans un système.

L'unique relation liant les classboxes est l'*import*. Syntaxiquement l'opération d'import se décrit comme CB2 import: #T from: CB1. Importer dans un classbox CB2

un trait T défini dans un classbox CB1 rend T visible dans CB2. Un trait peut être importé par un autre classbox dans le but d'être utilisé par une classe.

### 4.2. *Extension de classe*

Une classe C importée par un classbox CB2 depuis un classbox CB1 est visible dans CB2. C peut donc être étendue par de nouvelles méthodes [BER 05]. Etendre une classe en la faisant utiliser un trait nécessite de séparer la relation classe-trait de la définition de classe. Nous proposons donc d'enrichir la définition d'*extension de classe* donnée initialement par Bergel avec la notion d'*utilisation de traits* et celle d'*ajout de variables d'instances* à une classe [BER 04, BER 05]. Une extension de classe consiste maintenant à étendre une classe avec (i) de nouvelles méthodes, et/ou (ii) des redéfinitions de méthodes, et/ou (iii) de nouvelles variables d'instances, et/ou (iv) l'utilisation d'un ou plusieurs traits.

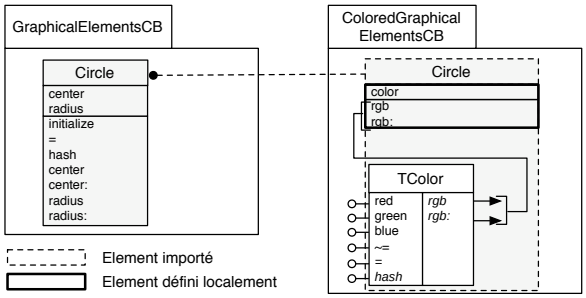


**Figure 3.** *La classe Circle du classbox GraphicalElementsCB est importée par le classbox ColoredGraphicalElementsCB. Elle est étendue par ce dernier avec la variable color, les deux méthodes de colle rgb et rgb: et le trait TColor.*

La classe Circle est définie dans le classbox GraphicalElementsCB. Circle est étendue par le classbox ColoredGraphicalElementsCB avec une nouvelle variable d'instance color, deux accesseurs (rgb, rgb:) et une utilisation du trait TColor défini par ColoredGraphicalElementsCB mais qui pourrait être importé d'un autre classbox.

```
ColoredGraphicalElementsCB import: #Circle from: GraphicalElementsCB;
 addVariableNamed: #color to: #Circle; "Ajout de la variable color a la classe Circle"
 addMethod: 'rgb ^color' to: #Circle;
 addMethod: 'rgb: aColor color := aColor' to: #Circle;
 createNewTrait: #TColor; "Creation du trait TColor"
 addMethod: 'red ...' to: #TColor; ... "Definition des methodes"
 trait: #TColor require: {#rgb . #rgb:}; "Specifications des methodes requises"
 make: #Circle useComposition: {TColor} "Compose Circle avec TColor"
```

Dans le classbox ColoredGraphicalElementsCB, l'application du trait TColor est complète car toutes les méthodes requises sont satisfaites et l'état nécessaire est ajouté à la classe Circle.

### 4.3. *Visibilité d'une extension*

Le modèle des classboxes permet de contrôler la visibilité des changements apportés à une classe. Contrairement aux mécanismes d'introduction proposés par AspectJ [KIC 01] et MultiJava [CLI 00, MIL 03] pour lesquels la visibilité des méthodes ajoutées est globale, les méthodes ainsi définies dans un classbox ont une visibilité limitée au classbox qui les définit. Combiner ce mécanisme avec l'utilisation d'un trait revient à limiter la visibilité de la relation entre une classe et un trait.

Une extension de classe n'est visible que dans le classbox qui la définit et dans les classboxes important les classes étendues. Cette règle permet d'apporter des changements non anticipés à une application sans risquer de perturber les clients reposant sur la version originale de l'application. Etendre une classe revient à créer une vue différente de celle-ci pourvue d'une définition enrichie par des extensions. Les classes en relation avec la classe étendue ne sont pas affectées puisqu'elles ont une vue différente de celle-ci ne montrant pas ces extensions.

## 5. Collaborations transverses et modifications non anticipées

Le concept de collaboration [HOL 92, SMA 98] permet de réifier et gèrer des fonctionnalités transverses à plusieurs classes. Un *rôle* est un ensemble de fonctionnalités s'appliquant à une classe et une *collaboration* un ensemble de rôles. On assimile un rôle à un trait et une collaboration à un classbox.

### 5.1. *Exemple de conception par collaboration*

Pour mettre à l'épreuve l'expressivité de la symbiose ainsi réalisée entre le mécanisme de visibilité offert par les classboxes et les facilités de structuration introduites par les traits, nous considérons une application de parcours de graphes qui a été initialement présentée par Holland [HOL 92], puis dans de nombreux autres travaux (VanHilst et Notkin [VAN 96], Smaragdakis [SMA 98]). Ces différents travaux utilisent cette application comme l'exemple canonique de conception basée sur les rôles.

L'exemple de Holland définit trois opérations (algorithmes) sur un graphe non orienté, basées sur un parcours en profondeur du graphe : (i) l'opération *Vertex Numbering* numérote et dénombre tous les nœuds du graphe, (ii) *Cycle Checking* vérifie si le graphe est cyclique et enfin (iii) *Connected Regions* classe les nœuds du graphe en différentes regions liées entre elles. L'application est composée de trois classes : Graph définit le graphe comme conteneur de nœuds , Vertex définit les pro-

priétés de chaque nœud et enfin la classe Workspace contient les variables globales de l'application.

Cette application se décompose suivant cinq collaborations différentes (figure 4) : (i) *Undirected Graph* définit les propriétés d'un graphe non orienté (représentation du graphe sous forme de nœuds se référençant mutuellement), (ii) *Depth-First Traversal* est responsable du parcours en profondeur de la structure définie par les nœuds, (iii) *Vertex Numbering* permet de numéroter l'ensemble des nœuds , (iv) *Cycle Checking* détermine la présence de cycles et finalement (v) *Connected Region* classifie les nœuds d'un graphe en régions de nœuds connectés.

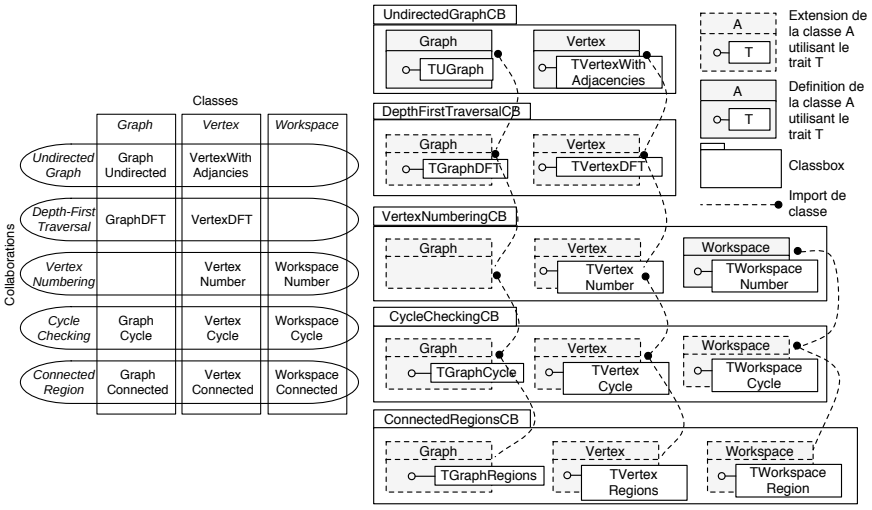


**Figure 4.** *Décomposition en collaborations. Partie gauche : les ovales représentent les collaborations, les rectangles les classes et leurs intersections les rôles. Partie droite : Les collaborations sont représentées par des classboxes et les rôles par des traits.*

### 5.2. *Collaboration avec les traits et les classboxes*

Une collaboration est représentée par un classbox, et un rôle par un trait. La figure 4 illustre l'application du parcours de graphe exprimé à l'aide de notre modèle. Cette modélisation est composée de cinq classboxes représentant les collaborations, chacun définissant un ensemble de traits représentant les rôles de la collaboration. Ces traits sont utilisés par les classes importées ou définies. Par souci de clarté la figure 4 ne montre que les définitions de classes et de traits et l'application de ces derniers sur les classes importées.

Le premier classbox UndirectedGraphCB définit les classes Graph et Vertex dotées des propriétés d'un graphe non orienté. Il définit deux traits TUGraph et TVertexWith-

Adjacencies respectivement utilisés par Graph et Vertex. Le deuxième classbox DepthFirstTraversalCB offre un algorithme de parcours en profondeur à travers deux traits TGraphDFT et TVertexDFT. Les deux classes Graph et Vertex sont importées du premier classbox et sont étendues en utilisant les deux nouveaux traits. Le classbox VertexNumberingCB importe les deux classes étendues par les classboxes précédents et définit une classe Workspace. La collaboration *VertexNumbering* ne définit pas de rôle pour la classe Graph, cette classe n'est donc pas étendue. Deux traits représentant les rôles *VertexNumber* et *WorkspaceNumber* sont utilisés par Vertex et Workspace. Les deux autres classboxes CycleCheckingCB et ConnectedRegionsCB sont construits d'une façon similaire.

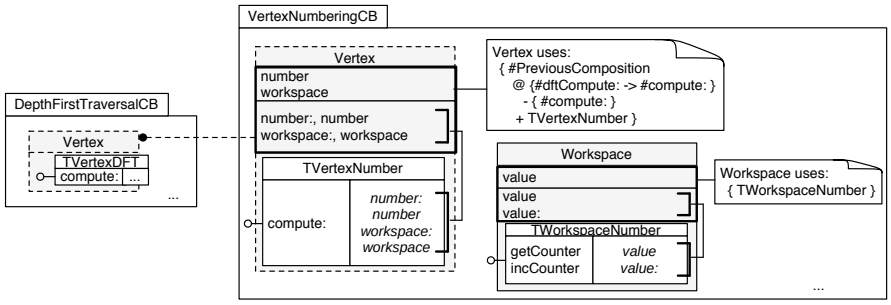


**Figure 5.** *Composition du rôle VertexNumber avec la classe Vertex.*

La figure 5 détaille le raffinement apporté à la collaboration *DepthFirstTraversal* par la collaboration *VertexNumbering*. Le rôle *VertexNumber* (modélisé par le trait TVertexNumber) est composé avec la classe Vertex importée. La classe Vertex est ainsi étendue avec deux variables (number et workspace) et quatre méthodes représentant les accesseurs de ces variables. Le trait TVertexNumber définit une méthode compute: et nécessite l'accès aux variables en requiérant leur accesseurs.

Cette composition est réalisée en utilisant une référence à la précédente composition de trait (définie dans le classbox d'où provient Vertex) à travers l'identifiant #PreviousComposition. Le mécanisme d'alias (@ { #dftCompute: -> #compute:}) permet d'accéder à la méthode compute: définie par TVertexDFT sous le nom dftCompute: dans le classbox DepthFirstTraversalCB (non représentée sur la figure). Cette méthode est ensuite masquée (- { #compute: }) pour éviter un conflit lors de l'ajout du trait TVertexNumber.

Le code de la méthode compute: définie par le trait *TVertexNumber* fait référence à la méthode du même nom fournie par la précédente collaboration :

```
TVertexNumber≫compute: aBloc
    number := workspace value.          "Affectation d'un numero au vertex"
    workspace incCounter.               "Incrementation du compteur"
    self dftCompute: aBloc              "Appel de la precedente version de compute:"
```

Le classbox VertexNumberingCB définit la classe Workspace contenant une variable d'instance value et deux accesseurs value et value:. Cette classe utilise le trait TWorkspaceNumber défini dans ce classbox, en spécifiant une composition triviale. TWorkspaceNumber a pour prérequis les deux accesseurs de value.

```
TWorkspaceNumber≫getCounter
    self value ifNil: [ self value: 0 ].
    ↑self value
```

```
TWorkspaceNumber≫incCounter
    self value: (self value + 1)
```

### 5.3. *Apports des classboxes et des traits*

**Identité des participants préservée**. Utiliser une *mixin* pour modéliser un rôle implique un sous classage [SMA 98]. Apporter une extension non anticipée en utilisant le sous classage nécessite une adaptation des clients [FIN 98]. Ces derniers doivent référencer les sous-classes pour bénéficier des extensions. L'utilisation des traits et des classboxes permet d'apporter une extension tout en conservant l'identité des classes étendues. Appliquer un rôle à un participant se fait sans sous-classage, l'identité de ces derniers est donc préservée.

**Une collaboration comme un changement non planifé**. L'utilisation de *mixin* convient à une architecture planifiée avant l'élaboration d'une application. Elle ne permet donc pas de représenter une collaboration comme un changement *non planifié* devant être appliqué sur une application existante. L'utilisation des traits se fait donc sans sous-classage, ce qui permet à des clients d'utiliser des classes ainsi raffinées. Une application peut donc être raffinée avec de nouvelles collaborations sans affecter ses clients.

**Visibilité des extensions contrôlée**. La propagation des extensions doit nécessairement être contrôlée pour préserver le comportement de certains clients. Apporter une extension à une classe sans affecter tous les utilisateurs de cette classe nécessite un contrôle de la visibilité de cette extension. Les classboxes permettent de localiser une extension et de la rendre disponible aux classboxes clients.

**Les méthodes requises définissent la cohérence**. Assurer un ordonnancement cohérent entre les collaborations constitue un problème crucial déjà abordé par l'utilisation de propriétés propagées le long de la hiérarchie [BAT 97] ou encore par l'utilisation d'un système de contraintes [STE 93]. Avec notre solution, une collaboration ne peut être appliquée que si les méthodes requises par les traits et donc par la collaboration sont fournies.

## 6. Discussion et évaluation de notre modèle

D'après le modèle original des traits, l'utilisation d'un trait est spécifiée dans la définition de la classe l'utilisant [DUC 05]. Etendre une classe en la faisant utiliser un trait nécessite de séparer la déclaration entre un trait et la classe de la définition de celle-ci. Dans notre proposition, la responsabilité d'utiliser un trait relève du classbox apportant l'extension et non plus de la classe.

Une classe peut être incrémentalement étendue par une chaîne de classboxes l'important chacun à leur tour. Dans chaque classbox, une extension par une utilisation de trait doit être le résultat d'une composition entre un trait et la composition précédente (visible dans le classbox d'où provient la classe). Notre modèle inclut donc une variable implicitement déclarée #PreviousComposition référençant la composition de trait visible dans le classbox d'où est importée la classe. L'utilisation de cette variable est illustrée par la figure 5. L'introduction d'une telle variable permet de cacher la composition des traits précédemment utilisés par la classe et donc de voir une classe comme une composition opaque de traits.

Assimiler l'utilisation d'un trait à une extension de classe permet de restreindre la portée de la relation classe-trait. Cette relation n'existe que dans un espace bien déterminé, défini par le classbox où cette relation est déclarée et par les classboxes important la classe ainsi étendue.

Un trait est originalement conçu pour factoriser la définition de méthodes pouvant être utilisées dans plusieurs classes n'appartenant pas à une même hiérarchie. Notre approche propose d'enrichir la notion d'extension de classe en limitant la visibilité de la relation classe-trait. Elle permet d'utiliser un trait pour apporter un changement non anticipé à une application.

Un classbox permet de limiter à un espace bien défini la visibilité d'un fragment de comportement ou d'état défini comme une extension de classe. Cela offre une solution à la notion de subjectivité qui est à la fois (i) générique car particularisable pour les sous-classes et les simples références et (ii) légère car le contexte lors d'un envoi de message est implicite. Cette approche est à comparer avec les travaux liés à la notion de point de vue [CAR 90], qui permettent d'utiliser des interfaces particulières lors de l'héritage, et à la programmation orientée sujet [HAR 93], qui particularise l'interface d'une classe en fonction des objets référençants. Allier les traits avec les classboxes définit une forme de programmation subjective pour laquelle la définition d'une classe peut être différente pour l'héritage et pour les simples références.

Un raffinement pouvant toucher un certain nombre de classes est défini par un classbox étendant chacune de ces classes en la faisant utiliser un trait. Un certain nombre de méthodes stipulées comme requises dans les traits doivent être présentes pour pouvoir utiliser un trait, et donc appliquer une collaboration. Par exemple, dans la figure 4, la collaboration VertexNumberingCB ne peut pas être placée entre UndirectedGraphCB et DepthFirstTraversalCB puisque cette dernière definie la methode compute: (montrée dans la figure 5). Cette methode doit être présente avant d'appliquer

VertexNumberingCB. Déclarer des méthodes requises garantit un ordonnancement cohérent des collaborations. Cependant, le modèle ne fournit aucune garantie que des traits, destinés à interagir entre eux, interagiront une fois le classbox appliqué. Par exemple, supposons un classbox ObserverPattern définit deux traits TObserver et TObservable. Au moment d'appliquer ce classbox, le modèle ne garantit pas qu'un objet "observer" observe que des objets "observable".

## 7. Conclusion

Cet article a d'abord été l'occasion d'enrichir le modèle des classboxes en permettant l'introduction de variables d'instance dans une classe importée. Etendre une classe avec l'état facilite l'utilisation des traits qui peuvent accéder à celui-ci par l'intermédiaire des méthodes dites de colle. Cette amélioration du modèle des classboxes constitue une solution au problème de l'encapsulation de l'état des classes. Cependant cette solution ne permet pas la réutilisation d'une introduction de variables par différentes classes.

Cet article a ensuite été l'occasion de montrer comment le couplage du modèle des traits et de celui des classboxes permet une mise en œuvre particulièrement expressive de la programmation dite subjective. A titre d'exemple, nous avons présenté une modélisation des collaborations. L'utilisation de traits (à la place de *mixins*) préserve l'identité des participants et celle des classboxes limite la propagation de ces traits à un espace bien délimité.

En conclusion, les traits et les classboxes sont deux extensions du modèle des classes *simples* et *expressives*. Leur couplage permet de définir des modifications de comportement (à base d'extensions de classe) pouvant être appliquées sur un grand nombre de classes tout en limitant la visibilité de ces modifications à des points bien précis d'un système.

**Remerciements**. Nous remercions chaleureusement pour son support financier la Swiss National Science Foundation pour le projet "Tools and Techniques for Decomposing and composing Software" (SNF Project No. 2000-067855.02) et "Recast: Evolution of ObjectOriented Applications" (SNF 2000-061655.00/1). Nous remiercions également Jérôme Bergel pour ses précieuses relectures.

## 8. References

[BAT 97] BATORY D., GERACI B. J., "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, 1997.

[BER 04] BERGEL A., DUCASSE S., "Dynamically Scoped Aspects with Classboxes", *Proceedings of the First JFDPA (Journée française de la programmation par aspects)*, 2004.

[BER 05] BERGEL A., DUCASSE S., NIERSTRASZ O., WUYTS R., "Classboxes: Controlling Visibility of Class Extensions", *Computer Languages, Systems and Structures*, 2005, Elsevier, To appear.

[BRA 90] BRACHA G., COOK W., "Mixin-based Inheritance", *Proceedings OOPSLA/ECOOP '90*.

[CAR 90] CARRÉ B., GEIB J.-M., "The Point of View Notion for Multiple Inheritance", *Proceedings OOPSLA/ECOOP '90*.

[CLI 00] CLIFTON C., LEAVENS G. T., CHAMBERS C., MILLSTEIN T., "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java", *Proceedings OOPSLA 2000*.

[DUC 05] DUCASSE S., SCHAERLI N., NIERSTRASZ O., WUYTS R., BLACK A., "Traits: A Mechanism for fine-grained Reuse", *Transactions on Programming Languages and Systems* , 2005, under revision.

[FIN 98] FINDLER R. B., FLATT M., "Modular object-oriented programming with units and mixins", *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ACM Press, 1998.

[HAR 93] HARRISON W., OSSHER H., "Subject-Oriented Programming (A Critique of Pure Objects)", *Proceedings OOPSLA '93*.

[HOL 92] HOLLAND I. M., "Specifying Reusable Components Using Contracts", MADSEN O. L., Ed., *Proceedings ECOOP '92*, 1992.

[KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. G., "An overview of AspectJ", *Proceeding ECOOP 2001*.

[MIL 03] MILLSTEIN T., REAY M., CHAMBERS C., "Relaxed MultiJava: balancing extensibility and modular typechecking", *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, ACM Press, 2003.

[SMA 98] SMARAGDAKIS Y., BATORY D., "Implementing Layered Design with Mixin Layers", JUL E., Ed., *Proceedings ECOOP '98*.

[STE 93] STEYAERT P., CODENIE W., D'HONDT T., HONDT K. D., LUCAS C., LIMBERGHEN M. V., "Nested Mixin-Methods in Agora", NIERSTRASZ O., Ed., *Proceedings ECOOP '93*.

[VAN 96] VANHILST M., NOTKIN D., "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer Verlag, 1996, p. 22–37.