VERS UNE MODÉLISATION TRANSVERSE ET MODULAIRE DES COLLABORATIONS PAR COUPLAGE DES TRAITS ET DES CLASSBOXES

Crosscutting and scoped collaborations

Florian MINJAT

encadré par Pierre Cointe et Stéphane Ducasse

Laboratoire d'Informatique de Nantes Atlantique
2, rue de la Houssinière
B.P. 92208
F-44322 NANTES

Projet EMN/INRIA OBASCO
Ecole des Mines de Nantes
4, rue Alfred Kastler
La Chantrerie - B.P. 20722
F-44307 NANTES

Software Composition Group
University of Bern
Neubrückstrasse 10
CH-3012 Bern

RAPPORT DE STAGE DE DEA
Septembre 2004

Florian MINJAT (encadré par Pierre Cointe et Stéphane Ducasse)

VERS UNE MODÉLISATION TRANSVERSE ET MODULAIRE DES COLLABORATIONS PAR COUPLAGE DES TRAITS ET DES CLASSBOXES

Crosscutting and scoped collaborations

VERS UNE MODÉLISATION TRANSVERSE ET MODULAIRE DES COLLABORATIONS PAR COUPLAGE DES TRAITS ET DES CLASSBOXES

Crosscutting and scoped collaborations

Florian MINJAT (encadré par Pierre Cointe et Stéphane Ducasse)

Remerciements

Je tiens tout d'abord à remercier Pierre Cointe et Stéphane Ducasse pour m'avoir encadré et guidé durant ce stage.

Je tiens à remercier Pierre Cointe pour m'avoir accueilli au sein de l'équipe EMN/INRIA OBASCO et Oscar Nierstrasz pour m'avoir accueilli au sein de son équipe du Software Composition Group de Bern.

Je souhaite également remercier Alexandre Bergel, pour son soutien et son implication dans la réalisation de ce projet.

Enfin, je remercie toutes les personnes des départements informatiques de l'École des Mines de Nantes et du Software Composition Group de Bern, pour leur accueil et pour l'ambiance conviviale qui y règne.

Chapitre 1

Introduction

Contexte de l'étude : réutilisation et factorisation de code.

Dans le domaine de la réutilisation et factorisation de code, la granularité a beaucoup évoluée aux cours des années dans le monde de la programmation orientée objet. Initialement au niveau des fonctions, puis des classes, et même aujourd'hui jusqu'aux composants et packages. Mais si l'on considère comme unité de réutilisation une fonctionnalité d'un programme, on se rend compte que ces granularités ne sont pas optimales : dans un programme orienté objet, une classe ou une méthode à elle seule ne permet pas d'accomplir une tâche, la granularité est trop petite. Et au niveau d'un module, composant ou package, on obtient souvent une entité destinée à remplir plusieurs fonctions. Une fonctionnalité, dés qu'elle est un minimum complexe, est issue de la collaboration de plusieurs entités logicielles : "*no object is an island*". Cette étude s'intéresse donc aux fonctionnalités issues de la collaboration de plusieurs classes. Dans ce cadre, la granularité qui semble la plus valable pour identifier des éléments de code réutilisable se trouve donc au niveau de fragments de plusieurs classes, voir de fragments de plusieurs fonctions [53].

Ce niveau de granularité a déjà été caractérisé par plusieurs études, et nommé collaboration. Le but de cette étude est donc de réifier cette notion de collaboration, de pouvoir composer plusieurs collaborations entre elles et de pouvoir en vérifier la cohérence. Dis de façon plus large, nous allons définir un langage de description de composition de composants basé sur la notion de collaboration.

Organisation du document.

La première partie de ce rapport aura pour but d'introduire et d'expliciter les notions clefs de cette étude. Elle consistera en un état de l'art de la programmation par aspects ou AOP (Aspect Oriented Development), puis d'une étude des architectures et frameworks de composants, suivi des notions de modules pour enfin finir sur les mixins, qui sont les plus entités les plus utilisées jusqu'à présent pour implémenter la notion de collaboration.

Suite à cette introduction des notions de base de mon étude, la deuxième partie du rapport détaillera les concepts du design par collaboration avant d'énumérer les solutions existantes d'implémentation de ces concepts, leurs avantages et leurs inconvénients.

Et enfin la troisième partie servira à présenter une solution au problème abordé : elle com-

mencera par une présentation des paradigmes de Trait et Classbox développés au sein du Software Composition Group de l'université de Bern, puis par la description d'un design par collaboration utilisant ces deux concepts. Cette section se finira par la comparaison des intérêts et inconvénients de cette solution par rapport aux solutions existantes avant de conclure ce rapport.

Chapitre 2

État de l'Art sélectif sur la réutilisation

La problématique de réutilisation du code n'est pas des plus jeunes et de nombreux concepts existent aujourd'hui qui permettent la réutilisation. On peut ainsi faire de la réutilisation en utilisant les aspects, les composants, les modules ou encore les mixins. Nous allons présenter rapidement ces notions nécessaires à la présentation de notre modèle.

2.1 Aspects

2.1.1 Introduction à la programmation par aspects

La séparation des fonctionnalités d'une application [1] est un concept clef de l'informatique d'aujourd'hui. Il spécifie qu'un problème donné implique l'utilisation d'un certain nombre de fonctionnalités, qui doivent être identifiées et séparées pour réduire la complexité, et augmenter la robustesse, la maintenabilité, la réutilisabilité et la vérifiabilité de l'application.

Dans un monde idéal, on pourrait par ce découpage arriver à décomposer et isoler toutes les fonctionnalités d'une application. Mais comme nous l'avons dit plus haut, une fonctionnalité est souvent étalée sur plusieurs classes à travers l'application, on dit alors que c'est une fonctionnalité transverse [2]. Des exemples célèbres de fonctionnalités transverses sont la synchronisation, la gestion de la persistance, la gestion de journals d'événements dans l'application...

Ces concepts transverses posent de gros problèmes. En effet, ils sont dur à identifier, comprendre, réutiliser, étendre et maintenir du fait de leur explosion en de nombreux endroits de l'application.

La programmation par Aspects [3] fournit des abstractions pour représenter ces fonctionnalités transverses, aussi appelées *Aspects*. Il existe deux types d'interaction avec l'application pour un langage d'aspects : la gestion au niveau du comportement et au niveau structurel.

[1] Separation of concern
[2] Crosscuting concern
[3] Aspect-Oriented Software Development

Gestion du comportement. Il est possible de définir et de gérer des aspects qui vont affecter le comportement de l'application (*behavioral aspect language*). Pour définir ces aspects, il est nécessaire d'avoir plusieurs notions :

- **Les points de jonction** : un point de jonction [4] est l'ensemble des endroits où l'aspects traverse l'application de base. La granularité du point de jonction est très variable, au niveau de la classe, de la méthode, du package...
- **Les points de coupe** : un point de coupe [5] correspond à la spécification de l'ensemble des points de jonction d'une fonctionnalité. Un point de coupe peut énumérer les points de jonction ou utiliser une définition plus abstraite.
- **Les actions** : si le point de coupe spécifie à quel endroit dans le code l'aspect doit se trouver, il faut encore définir cet aspect. C'est le rôle des actions [6], qui définit le comportement qui doit être attaché à un point de coupe ou un ensemble de points de jonction.

Cette terminologie a été introduite en 1997 par AspectJ [28], qui est une implémentation spécifique des Aspects pour Java. Mais malgré son caractère spécifique, ces termes ont rapidement été globalement acceptés par la communauté concernée par l'AOSD. De nombreux autre langages orientés aspects existent qui bien évidement ont des méthodes différentes pour spécifier les aspects, points de coupe et advises, mais les concepts restent les mêmes, même si leurs noms changent parfois.

Gestion de la structure. Un langage d'aspect peut inclure des moyens d'altérer la structure du programme de base et non pas seulement interagir avec la structure existante. Cette 'altération' consiste à modifier la structure des classes, par exemple en ajoutant de nouvelles variables ou méthodes.

2.1.2 Principaux langages d'aspect existants

Il existe actuellement deux approches différentes pour expliciter, définir et gérer les fonctionnalités transverses d'un programme : l'approche par filtres de composition [7] et par attachement d'aspect [8]. Nous allons voir rapidement les caractéristiques de ces deux approches et les langages qui ont étés implémentés à partir d'elles.

Filtres de composition. L'approche par filtres de composition [4] date de 1991. Elle vise à utiliser des filtres pour abstraire la communication entre les objets. Les techniques de filtres de composition sont une extension du modèle de la programmation orienté objet, où les points de jonctions considérés sont les envois et receptions de message entre les objets. Le modèle basique est de mettre des filtres d'entrée et de sortie qui affectent les envois et réceptions de

[4] Joinpoints
[5] Pointcut
[6] Advices
[7] Composition Filters
[8] Aspect Attachement

messages. Le but de cette approche est de construire sur l'existant et d'étendre le concept de programmation orientée objet plutôt que créer de nouvelles entités logicielles complexes.

Chacun des messages qui arrivent sur un objet est soumis à une évaluation et manipulation par les filtres associés à cet objets. Ces filtres peuvent permettre la délégation, la vérification de pré et post conditions, la synchronisation ou encore la détection d'erreurs. Ils ont une sémantique claire et n'ont pas une structure figée, ce qui permet de vérifier et de faire évoluer les filtres facilement. Les filtres peuvent être ajoutés directement sur des langages à Objets comme Java ou Smalltalk sans modifier le langage. Un langage spécifique est utilisé pour définir le comportement des filtres et pour les appliquer sur les objets du langage.

Il existe deux langages principaux basés sur cette approche :

- **ComposeJ** [4] qui est le résultat de la combinaison du modèle des filtres de composition et du langage Java. Ce langage peut être utilisé très facilement comme add-on du compilateur Java standard.
- **ConcernJ** [45] permet de surimposer les filtres de composition sur un système Java et étend le concept initial des filtres de compositions

Attachement d'aspects. L'approche par attachement d'aspect a été présentée pour la première fois en 1997. Le but des aspects comme présenté dans [27] est de permettre au développeur un moyen de séparer clairement les composants du langage et les aspects transverses en lui fournissant des outils pour les créer, gérer et enfin composer pour constituer l'application finale utilisant utilisant du code normal et des aspects.

Pour réaliser celà, il faut ajouter un language d'aspect et un tisseur [9] d'aspects aux outils standards du développeur. Le langage d'aspect est différent du langage objet (Java, C++, etc.) et sert à programmer les aspects. Et le tisseur sert à utiliser le code des aspects pour les intégrer dans le code du programme.

Le langage le plus avancé et le plus médiatisé à ce jour est AspectJ [28], et l'amalgame entre AOP et AspectJ est souvent fait par les néophytes. Mais il existe de nombreuses solutions pour répondre au problème du tissage d'aspects :

- **Des aspects pour des problèmes spécifiques** (*Specific problem domains*) : Ces langages datent de 1997 et des premiers systèmes basés sur les aspects. Ils consistent en des implémentations spécifiques à des problèmes, où un nombre réduit d'aspects génériques ont été identifiés et sont composables avec le code de l'application. Ainsi *D* a été proposé par Lopes en 1997 [30] et permet de composer des aspects de concurrence au sein d'un programme Java. Ou encore *AML* [40], un système fait pour le calcul de matrice [10] a été implémenté par Kiczales en séparant les fonctionnalités transverses du langage fonctionnel de base pour optimiser les calculs.

[9] Weaver

[10] Sparce matrix computation

– **AspectJ** : AspectJ [27] est une extension orientée aspect [11] de Java. Les aspects en AspectJ sont des entités différentes des classes écrites dans un langage similaire à Java mais avec de nouveau mots clefs. Ces aspects sont composés d'un ensemble de points de coupe, d'advices liés à ces points de coupe et de variables utilisables par les advices. AspectJ commence à être utilisé aujourd'hui dans l'industrie. Son principal défaut réside dans la difficulté à composer des aspects multiples conflictuels (qui utilisent un même point de jonction par exemple), il est très difficile actuellement avec AspectJ de définir des règles de précédence évoluées pour résoudre des conflits.

– **EAOP** [43] est une approche par aspects basée sur les événements [12], proposée en 2001 par Douence et Südholt. Ce modèle est basé sur une exécution par événements, les points de jonctions sont aussi des événements et le système est une machine à état, avec ses états, ses branchements... Cette utilisation de machine à état permet de faire de la résolution explicite de conflits entre les aspects, d'utiliser des opérateurs pour composer les aspects ou encore de pouvoir composer incrémentalement des couches d'aspects [13].

– **AO Logic Meta Programming (AOLMP** [13] a été introduit en 1999. Ses auteurs proposent d'unifier les langages d'aspects et weavers en un seul langage utilisé par tous.

– **JBoss AOP** [25] est un framework open-source pour programmer des aspects en Java. Les aspects sont décrits en XML, mais même si la terminologie et la syntaxe diffèrent, les concepts sont globalement les mêmes qu'avec AspectJ. Une différence importante est que le déploiement des aspects peut se faire dynamiquement pendant l'exécution de l'application. JBoss propose également un ensemble d'aspects génériques sur la sécurité, les transactions, etc..

– **AspectS** [21] est une implémentation des aspects pour Squeak, l'environnement Open-Source de programmation en SmallTalk.

– **Autres...** il existe encore de nombreuses différentes implémentation des aspects, dans quasiment tous les langages, et pas seulement les langages à Objets (en C par exemple)

[11] Aspect Oriented
[12] event-based
[13] aspect layers

2.2 Composants et Frameworks

2.2.1 Introduction à la notion de composant

La notion de composants est relativement récente, elle est apparue autour du milieu des années 90. Elle promeut la construction d'applications à partir de l'assemblage de composants [10]. De façon simpliste, un composant peut être décrit comme étant une brique logicielle préfabriquée qui est conçue pour être composée, c'est à dire assemblée, avec d'autres composants [32]. Un composant est réutilisable, c'est à dire qu'un même composant peut être employé dans la construction de différentes applications, et sa réutilisation ne nécessite, à priori, pas de connaissances sur son implémentation. L'approche à composants est fondée sur l'idée que le développement et l'assemblage de composants peuvent être réalisés de façon totalement séparés par des acteurs différents, à des moments différents.

Il n'existe pas de définition formelle de la notions de composants. Mais la littérature nous offre quelques définitions informelles qui vont nous permettre de clarifier cette notion :

Une abstraction statique avec des prises. Oscar Nierstrasz dans [39] dit : "Un component logiciel est une abstraction static avec des prises"[14]. Cette définition révèle plusieurs choses : tout d'abord un composant une abstraction, c'est à dire qu'il peut être n'importe quelle entité (interface, object, classe, template, type, fonction...) dont les détails d'implémentation sont cachés. Ensuite, un composant est statique, *i.e.*,c'est une entité qui peut être stockée indépendamment des applications dans lesquelles elle joue un rôle et être réutilisée plusieurs fois. Enfin, elle possède des prises : l'interaction entre le composant et les autres entités de l'application sont bien définies, la 'prise' pouvant être des paramètres, des ports, des références à d'autres objets...

Réutilisation et flexibilité des composants. Tichelaar a écrit dans sa thèse de master [56] : "Un composant est une abstraction générique boite noire qui est (re)configurable et composable par des prises"[15]. Le terme générique nous montre que les composants sont utilisables dans une large fourchette de problèmes connus. Ils sont de plus configurables : les composants peuvent être adaptés pour les besoins spécifiques de l'application au moment de la composition. Et enfin qu'un composant soit une abstraction sous la forme d'une boite noire nous montre que l'implémentation interne des composants est cachée. L'encapsulation des abstractions permet de rendre explicite l'architecture. Concrètement, un composant est donc composé d'un noyau boite-noire, dont l'implémentation est cachée, et d'une interface composée des 'prises', qui elle est visible, boite blanche. Cette interface permet de fournir une spécification du composant à l'utilisateur.

On voit ici la principale différence avec l'approche par objet, qui base la composition et l'extension de l'application sur l'héritage et une visibilité totale : c'est une framework à boite blanche. A l'opposé, la composition de composants ne demande pas de compréhension de leur rouages internes. Il suffit de comprendre les interfaces des composants à utiliser. Les frame-

[14] "A software component is a static abstraction with plugs."

[15] "A component is a generic black-box abstraction which is (re)configurable and composable by plugs"

works en boite-blanche sont plus adaptables et flexibles, mais ceux en boite noire ou grise sont plus faciles d'utilisation et supportent mieux le passage à l'échelle et la création de grosse applications.

2.2.2 Présentation de quelques modèles de composants existants

Il existe un très grand nombre de modèle de composants, mais nous allons voir les plus reconnus.

COM. Le modèle COM (*Component Object Model*) de Microsoft [7] a été conçu pour résoudre le problème de l'interopérabilité au niveau binaire entre des composants appartenant à des applications écrites par différents vendeurs. Une interface dans COM contient un ensemble de méthodes, est décrite dans un langage de description spécifique (IDL) et est identifiée de façon unique. COM est un modèle à composant qui souffre de problèmes de complexité au niveau de l'implémentation mais qui cependant a été novateur à son époque et reste probablement aujourd'hui le modèle à composants le plus répandu. COM a inspiré d'autres modèles qui ont repris ses idées fondatrices (par exemple l'Object Modeler de Dassault Systèmes [46] ou Bonobo de Gnome). Aujourd'hui, la nouvelle plateforme .Net de Microsoft vise à remplacer COM mais les idées de ce dernier y sont incorporées.

JavaBeans. En 1997, Sun crée la spécification du modèle à composants appelé JavaBeans [24]. L'objectif principal de ce modèle est de simplifier la construction d'applications par composition de composants. Le modèle à composants JavaBeans introduit un aspect original permettant de faciliter l'assemblage visuel : un composant peut être livré avec un ensemble de classes destinées à être employées par l'environnement d'assemblage pour faciliter la configuration des instances du composant. Ces classes peuvent cependant être retirées du paquetage lors de la livraison définitive des composants. Dans le cas où un composant n'est pas accompagné de classes de configuration, l'environnement d'assemblage doit être capable de générer de façon autonome des dialogues permettant de configurer les instances à travers l'introspection.

EJB. Le modèle à composants EJB (*Enterprise JavaBeans*)de Sun [15] se place dans un contexte d'applications construites selon une architecture répartie en trois tiers [44] : un tiers de présentation, qui réside principalement dans une machine du côté de l'utilisateur, un tiers applicatif, localisé dans un serveur, et un tiers de données, correspondant par exemple à une base de données. L'idée de cette architecture est de permettre l'interaction des utilisateurs avec les données en passant par le tiers applicatif, interaction souvent réalisée à partir d'un navigateur Web. Ces applications trois tiers nécessitent plusieurs fonctionnalités telles que la transaction, la sécurité, la distribution et la persistance. Le modèle EJB est spécifiquement orienté à la construction du tiers applicatif et à la mise en place de ces fonctionnalités.

CCM. Le modèle à composants de CORBA (CCM) [19] ajoute une couche au dessus de l'intergiciel CORBA [58]. Il permet de définir l'architecture d'une application distribuée sous forme de composition de composants. Un autre intérêt de ce modèle est qu'il supporte l'implémentation de composants dans différents langages.

Fractal. Fractal est un framework de composition [14] qui définit un modèle à composants générique car non orienté vers un domaine d'application particulier. Le framework contient un ensemble d'interfaces de programmation (API) orientées à la réalisation d'aspects tels que la définition de classes de composants, la reconfiguration dynamique des instances, la composition hiérarchique et l'introspection. L'implémentation d'un composant est divisée en deux parties : le contrôleur et le contenu. Le contrôleur peut implémenter un nombre variable d'interfaces de contrôle dont un certain nombre sont définies dans la spécification du modèle à composants. Le contrôleur agit sur le contenu qui peut contenir du code fonctionnel ou bien être un ensemble d'autres contrôleurs ; de cette manière, le modèle supporte la composition hiérarchique. Lorsque le contrôleur agit sur le contenu, il joue un rôle similaire à celui du conteneur. Une implémentation du framework nommée Julia [26] permet de décrire le contrôleur à partir d'un langage spécialisé et réalise ensuite un mélange du code de contrôle et du code fonctionnel à travers une approche à mixins. Un des aspects intéressants du modèle Fractal est qu'il n'impose pas un nombre fixe d'interfaces de contrôle, celles ci sont découvertes en temps d'exécution.

2.3 Modules

2.3.1 Définition du concept de module

Le terme de module a eu de nombreux sens. Nous utiliserons ici la définition donnée par Modular Smalltalk [60] et Clemens Szyperski [55] : "Un module est une capsule contenant des définitions d'éléments. Le module place une frontière claire entre les éléments qui sont défini en son sein et les éléments extérieurs définis dans d'autres modules. Un élément d'un module A ne sera visible depuis un autre module B uniquement si B importe A."

De plus un module à besoin d'avoir au minimum une interface extérieure.

Et enfin, un module seul doit pouvoir fonctionner indépendamment, sans connaissance des détails d'autres modules ([42]).

Il y a cinq critères pour considérer un concept comme modulaire [34] :

- Décomposition modulaire : un problème doit pouvoir être décomposé en plusieurs sous problèmes ;
- Composition modulaire : des modules doivent pouvoir être composés librement entre eux ;
- Compréhension modulaire : les modules doivent pouvoir fonctionner séparément ;

- Continuité modulaire : de petits changements dans les spécifications du problème doivent correspondre à des petits changements dans peu de modules ;
- Protection modulaire : une condition d'exécution anormale doit pouvoir être confinée au sein d'un module.

2.3.2 Modèles de modules existants

Nous allons maintenant faire un tour rapide des implémentations de module les plus intéressantes.

Selector Namespaces. Des langages comme ModularSmalltalk [60], Subsystems [59] and Smallscript [50] fournissent un mécanisme de gestion du contexte appelé *Selector Namespace*. Les méthodes qui sont insérées dans un tel namespace sont locales à ce namespace. Ainsi les conflits d'extension de classes sont évités, et plusieurs applications peuvent introduire une extension pour les mêmes classes et méthodes sans rentrer en conflit. Les extensions de classes ne sont alors plus visibles globalement mais uniquement dans le contexte du namespace. Mais la nouvelle définition locale de ces extensions de classe ne prend pas le pas sur le code original lorsque celui-ci est appelé. En effet, le système des Selector Namespace ne supporte pas le mécanisme de boucle locale [16] qui lui permettrait de rediriger localement les appels au code originel vers le code modifié.

Multijava. Multijava [9] est une extension de Java qui fournit les mécanismes de classes ouvertes [17] et de déploiement multiple de méthode [18]. Une classe ouverte est une classe dont les méthodes sont extensibles, de nouvelles méthodes peuvent lui être ajoutée. Ces nouvelles méthodes sont alors visibles dans le *package* fournissant ou important l'extension. Par contre les redéfinitions de méthodes ne sont pas possibles dans une classe ouverte. D'un autre côté, deux extentions de classes peuvent définir deux méthodes différents sur la même classe ouverte avec la même signature. Mais dans ce cas les extensions ne peuvent pas être utilisées dans le même contexte.

Unit. MZScheme [18] offre un module système évolué dans lequel un *Unit* est le bloc de construction de base. Un Unit est une entité logicielle composée de définitions, de requis et d'exports. Pour former une applications fonctionnelle, il faut instancier et composer les Units entre eux, car un Unit seul n'est pas fonctionnel. Le grand intérêt de ce modèle est que les connections entre les modules ou classes sont spécifiées séparément de leur définitions. Ce principe permet à un module d'être instancié au moment de sa liaison avec un autre. La réutilisabilité et l'extensivité sont possible en recombinant les Units. Une application, faite de Units peut être recomposée différemment et de nouveaux Units peuvent y être insérés grâce à un pro-

[16] Local rebinding
[17] Open Classes
[18] Multiple Method Dispatch

cédé de renommage [19]. Une Unit agit comme une boite noire : une classe interne à un Unit ne peut pas être étendue, à la place, un nouvel Unit à besoin d'être fourni et recomposé avec la classe.

Hyper/J. Hyper/J [41] est basé sur la notion d'*hyperspaces* et met en avant les compositions de fonctionnalités indépendantes. Les *hyperslices* sont des blocs élémentaires de construction contenant des fragments de classe. Ils sont destinés à être composés pour former des blocs de construction plus gros (ou une application complète) appelés *hypermodules*. Un hyperslice peut définir des méthodes sur des classes qui n'y sont pas forcement définies. De telles méthodes définissent une extension de classe, et les classes destinées à être étendues peuvent n'être connues qu'au moment de l'intégration. Mais les hyperslices ne permettent pas la redéfinition de méthodes et n'aident donc pas à supporter les évolutions non anticipées d'un système.

Classes Virtuelles. La spécification d'une classe virtuelle [17, 31] est similaire à celle d'une méthode virtuelle. En introduisant une adaptation dynamique [20] du nom d'une classe dans une hiérarchie d'entités encapsulées (des modules pour Keris [61], des interfaces de collaboration pour Caesar [35, 36], des classes pour gbeta [16], ou encore des *teams* pour Objectteams [20]) il est possible de raffiner une classe en l'étendant par une sous-entité (par exemple en utilisant l'héritage pour des classes). Une limitation des classes virtuelles est que la virtualité de celles-ci est limitée au contexte d'une hiérarchie : en dehors de cette hiérarchie une classe n'est plus virtuelle et ne peux être redéfinie.

Object-based Inheritance. En proposant le mécanisme de vraie délégation [21], Lava [29] permet des changements dynamiques non anticipés en utilisant des enrobeurs de classes (*class wrappers*) : par le biais d'un nouveau élément de langage, un objet a (instance de A) peut déléguer tous les messages qu'il reçoit et ne comprend pas à un autre objet b (instance de B). Lava permet alors une vraie délégation : la référence propre (constructeur *self* en SmallTalk, *this* en Java) de la classe b réfère à l'objet délégué a. Et les méthodes définies dans b qui ne sont pas connues de a sont les extensions apportées à a. Dans le modèle de Lava, les méthodes nouvelles ou redéfinies sont attachées à un objet précis plutôt qu'à une classe.

2.4 Mixins

Le double rôle d'une classe. Les classes jouent deux rôles qui entrent souvent en conflit. Le principal rôle d'une classe est d'être instanciable, et pour cela une classe se doit d'être complète. Mais elle joue aussi le rôle d'unité de réutilisation, et se doit donc de rester petite. De plus du fait de son rôle de génératrice d'instance, une classe à une rôle unique dans la hiérarchie de

[19] Aliasing
[20] Dynamic lookup
[21] True delegation

classe, alors que du fait de son rôle d'unité de réutilisation, elle se devrait d'y être en plusieurs endroits...

Solutions actuelles. Le système *Flavors* de Moon [37] a été une des premières tentatives pour résoudre ce problème : Les Flavours sont petits, pas forcèment complètes, et elle peuvent être mélangées (*mixed-in*, qui donnera le nom *mixin* au concept) au code en plusieurs endroits de la hiérarchie de classe. Des notions plus sophistiquées de mixins ont été ensuite développées par Bracha et Cook [8], Mens et Van Limberghen [33], puis par Ancona, Lagorio et Zucca [1]. Ces approches reprennent toutes le même concept qui permet au programmeur de créer des composants destinés à la réutilisation au lieu de l'instanciation. Mais toutes ces tentatives ont de gros impacts sur la facilité de compréhension du code comme nous allons le voir.

Utilisation de l'héritage simple comme mécanisme de composition. Les mixins utilisent la sémantique de l'héritage simple pour étendre une classe de base avec un ensemble de fonctionnalités. Mais bien que ce mécanisme soit bien étudié pour dériver depuis des classes existantes de nouvelles classes, il n'est pas très pratique pour en composer de nouvelles à partir d'éléments de base. En effet, l'usage de l'héritage comme mécanisme de composition oblige les mixins à être composés de façon linéaire, ce qui restreint grandement les possibilités de définition de la colle qui permet de lier les mixins ensembles.

L'héritage par Mixin. Un mixin est une spécification de sous-classage pouvant être appliqué à plusieurs classes parentes pour leur ajouter le même ensemble de fonctionnalités. Mais bien que le mécanisme d'héritage fonctionne très bien pour étendre une classe avec un seul mixin présentant une fonctionnalité orthogonale, il ne fonctionne pas de manière satisfaisante lorsqu'on essaye de lier une classe avec plusieurs mixins. Le problème est au niveau des mixins eux-même, qui souvent rentrent en conflit au moment de la composition sur une même classe. Et le mécanisme d'héritage n'a pas une syntaxe suffisamment expressive pour résoudre ces conflits. On peut voir le problème sous plusieurs formes :

Ordre total. La composition par mixin se doit d'être linéaire pour respecter le mécanisme d'héritage simple. Du coup, tous les mixins utilisés par une classe doivent être hérités par celle-ci un à la fois. Les fonctionnalités implémentées dans les mixins composés les plus tard remplacent [22] toutes les fonctionnalités de même nom ajoutées par les précédents mixins. Et quand on veut sélectionner les fonctionnalités pour résoudre les conflits entre plusieurs mixins qui entrent en conflit, un bon ordre total de composition n'existe pas toujours.

Composition complexe. Avec le mécanisme des mixins, l'entité composite finale ne peut pas contrôler la façon dont les mixins l'ont composée. Pour effectuer ce contrôle, le code permettant de résoudre les conflits doit donc être dispersé sur toutes les classes intermédiaires servant à la composition linéaire des mixins. Pour obtenir la bonne combinaison de fonctionnalités, il peut être nécessaire de modifier les mixins, d'introduire de nouveaux

[22] overrides

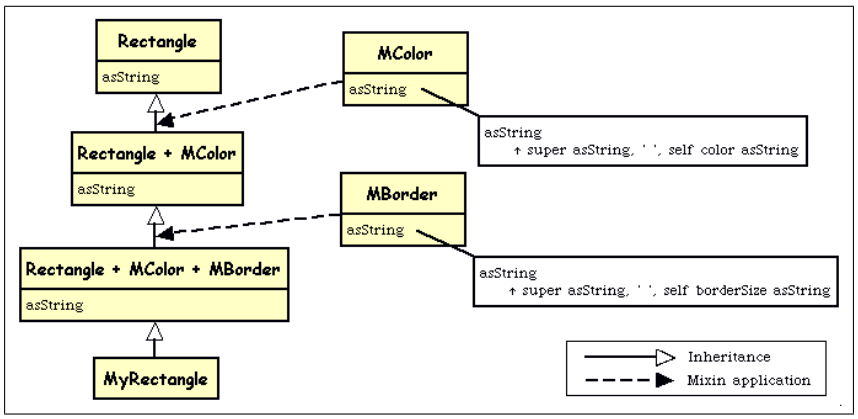

FIG. 2.1 – Le code qui interconnecte les mixins est spécifié dans le mixin MBorder. L'entité MyRectangle ne peut pas accéder à l'implémentation de asString défini dans le mixin MColor et dans la classe Rectange. (Les classe avec un + dans leur nom sont des intermédiaires générés pour appliquer les mixins.

mixins, ou même d'utiliser le même mixin plusieurs fois... Le code permettant de composer les mixins sur une même classe se retrouve donc dispersé entre plusieurs entités, et peut être intrusif au sein des fonctionnalités elles-mêmes.

Cette dispersion du code de composition est illustrè par la figure **??**, où une classe MyRectangle utilises deux mixinsMColor et MBorder qui fournissent deux méthodes différentes nommés toutes deux asString. Lors de la composition, il faut choisir quel mixin est composé en dernier et donc quel implémentation de asString masquera l'autre. Mais il n'est pas possible de définir comment composer les deux méthodes asString ensembles.

Hiérarchies fragiles. Du fait de la linéarité de la composition et des moyens limités de résolution de conflits, l'utilisation de plusieurs mixins pour composer des classes produit des chaînes d'héritage fragiles, très sensibles aux changements. Ainsi ajouter une méthode à un mixin pourrait avoir pour conséquence de masquer une méthode de même nom implémentées pour une autre fonctionnalité dans un mixin composé plus tôt dans la chaîne d'héritage. Et si l'on se rend compte du problème et que l'on veut rétablir le précédent comportement tout en conservant les dernières modifications, il peut être nécessaire d'ajouter ou de modifier plusieurs mixins dans la chaîne d'héritage. De plus comme les mixins sont des unités de réutilisations, ce problème devient particulièrement grave si quelqu'un modifie un mixin utilisé de nombreuses fois dans la hiérarchie des classes.

Pour illustrer ce problème, on peut imaginer que dans l'exemple précédent (figure 2.1) le mixin MBorder n'a pas initialement de méthode asString et que c'est l'implémentation de asString introduite par MColor qui est présente dans MyRectangle. Et maintenant on ajoute une méthode asString dans le mixin MBorder. Celle-ci va masquer la précédente implémentation dans MyRectangle du fait de l'ordre total. Et pour rétablir le comportement original de MyRectangle, il va falloir modifier plusieurs mixins...

Chapitre 3

Le principe de la construction par collaboration

En résumé après cet état de l'Art des moyens permettant la réutilisation, il apparaît clair que dans le cadre des collaborations, ces moyens ne sont actuellement pas optimaux. En effet, les Aspects souffrent de leur définition et impact global, qui ne permet pas de faire des changements sur un ensemble de classes sans poser problème à toutes les applications les utilisant. Les composants quand à eux sont des entités devant être indépendantes, elles encapsulent donc bien souvent plusieurs fonctionnalités et ne peuvent servir comme unité de réutilisation de la fonctionnalité. Les modules ont une vocation de conteneurs, ils n'ont des fonctionnalités que grâce aux classes qu'il contient, et ne peux donc bien évidement pas jouer ce rôle de réutilisisation. Et enfin les mixins ayant à une granularité au niveau de la classe, il n'est pas aisé de s'en servir pour encapsuler une collaboration entre plusieurs classes, même si comme nous allons le voir, ce sont eux qui sont les plus proches de l'entité recherchée. Aucune solution présentée n'est donc en mesure d'assurer ce rôle, il va falloir chercher d'un autre côté.

Dans ce chapitre, nous allons présenter plus en détail le design par collaboration et son concept, que nous illustrerons par un exemple, avant de regarder rapidement les solutions existantes et leur problèmes.

3.1 Présentation de la conception par Collaboration

Collaborations in a Nutshell

Une fonctionnalité, définie par une Collaboration, correspond à un ensemble de comportements destiné à être appliqués sur une ou plusieurs classes [22, 53]. L'idée est qu'une fonctionnalité est toujours issue de la collaboration de plusieurs classes, et qu'il faut donc utiliser un nouveau niveau de granularité pour la réutilisation de fonctionnalité : un ensemble de méthodes provenant de plusieurs classes, autrement appelé une Collaboration.

Dans ce concept, on désigne par rôle l'ensemble des méthodes d'une classe qui jouent un rôle dans la fonctionnalité. Une Collaboration est donc composée d'un ensemble de rôles, chaque rôle devant s'appliquer sur une classe. Elle sert à étendre un ensemble de classe pour

lui ajouter une fonctionnalité. Il est également possible de composer les Collaborations les unes après les autres sur un ensemble de classes pour ainsi construire une application possédant plusieurs fonctionnalités.

Problématiques du design par Collaboration

Réutilisabilité d'une Collaboration. La principale problématique du design par Collaboration est bien sûr celle de la réutilisabilité. En effet tout le but de ce design est de pouvoir réutiliser facilement les entités fonctionnelles créées. Il faut donc pouvoir aisément définir une collaboration, la stocker et pouvoir l'appliquer sur plusieurs applications sans modification.

Composition de collaboration. Une des hypothèses de cette étude est que la granularité de la Collaboration est la plus indiquée pour isoler une fonctionnalité d'une application. Mais une application est forcément composée de plusieurs fonctionnalités. Il convient donc de pouvoir composer facilement les collaborations, en les appliquant de façon itérative sur un ensemble de classes de base. On obtient alors une application modulable, où il devient possible de facilement y ajouter, supprimer ou modifier des fonctionnalités (voir l'application de graphe dans la partie suivante de ce rapport pour un exemple de modularité).

Cohérence entre les Collaboration. La composition des Collaborations n'est possible qu'avec le respect de certaines règles de cohérence. En effet, il est très rare que les fonctionnalités d'une application soit complètement indépendantes les unes des autres. Le bon fonctionnement d'une fonctionnalité repose quasiment toujours sur l'existence et le bon fonctionnement des fonctionnalités de plus bas niveau. Si l'on considère les fonctionnalités d'un programme comme applicables indépendamment et modulables, un certain ordre d'application de ces fonctionnalités doit être respecté. Il faut donc pouvoir exprimer la notion de dépendance entre les collaborations, pouvoir définir des règles permettant de vérifier au moment de la composition d'un nouvelle collaboration sur un programme que celui-ci possède déjà tous les éléments nécessaires à la composition. Il peut être également intéressant en cas de conflicts de facilement en discerner la cause et de pouvoir les résoudre simplement.

Cohérence dans une Collaboration. Enfin, l'application une Collaboration sur un ensemble de classes doit être fait de façon complète et cohérente. Il est par exemple évident que pour le bon fonctionnement d'une Collaboration tous les rôles de celle-ci doivent être présents et effectivement appliqués. Il faut donc également pouvoir vérifier la validité de l'application d'une Collaboration non seulement vis-à-vis des fonctionnalités déjà présentes dans le programme mais en interne au niveau des éléments qui la composent.

3.2 Les collaborations par l'exemple

Pour illustrer le design par Collaboration, considérons une application de parcours de graphe qui a été présentée initialement par Holland [22], puis dans de nombreux autres travaux des recherches (VanHilst et Notkin [57], Smaragdakis [51, 52] et enfin Nierstrasz et Achermann [38]). Tous ces travaux utilisent cette application comme un exemple typique de design basé sur les rôles, et il est donc intéressant de l'étudier et le réutiliser avec notre approche pour pouvoir en comparer les bénéfices et inconvénients avec les précédentes.

Application. L'exemple de Holland définit trois opérations sur un graphe non-orienté générique, basées sur un parcours en profondeur du graphe ('Depth First Traversal') : l'opération '*Vertex Numbering*' permet de numéroter et dénombrer tous les noeuds du graphe en donnant la priorité sur la profondeur, '*Cycle Checking*' permet de vérifier si le graphe est cyclique, et enfin '*Connected Regions*' classe les noeuds du graphe en différentes regions liées entre elles. Un utilisateur doit pouvoir créer un graph, et utiliser indépendamment ces trois opérations sur celui-ci.

Structure en collaborations. Considérons cette application du point de vue d'un design par Collaboration. Il faut identifier d'une part les classes participantes et d'autre part les Collaborations.

L'implémentation du graphe générique nécessite trois classes : une classe *Graph* qui définie le graphe comme conteneur de noeuds (ou vertices), la classe *Vertex* qui définit les propriétés de chaque noeud et enfin la classe *Workspace* qui permet d'avoir des variables globales à l'application.

Cette implémentation peut aussi être vue comme une combinaison de trois Collaboration différentes : *Basic Graph Undirected Graph* et *Depth-First Traversal*.

- La Collaboration *Basic Graph* définie les propriétés minimale d'un graphe, composé d'un ensemble de noeuds.
- La Collaboration *Undirected Graph* encapsule les propriétés d'un graphe non-orienté, est responsable du respect de ces propriétés et de la représentation du graphe en noeuds et arêtes.
- Et la Collaboration *Depth-First Traversal* est responsable du parcours de la structure, et permet de visiter les noeuds en profondeur.

De même, les trois opérations définies au dessus de ce graphe générique utilisent ces trois classes et se définissent chacune par une nouvelle collaboration. Elle permet aussi de définir une interface propre de ce parcours pour une possible extension. C'est à dire que les étapes clef de l'algorithme utilisé pour réaliser le parcours doivent être explicites pour pouvoir être potentiellement spécialisées plus tard. L'opération de *Vertex Numbering* peut être considérée comme une extension du parcours en profondeur, dans lequel la numérotation d'un noeud est effectué la première fois qu'il est parcouru.

De cette façon, les trois opérations définies par Holland sur le graphe générique sont elles aussi des collaborations, qui spécialisent l'algorithme de parcours de graphe.

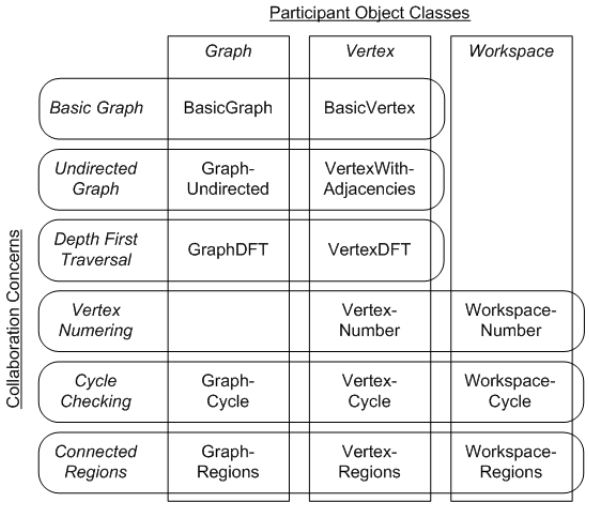

FIG. 3.1 – Double représentation de l'exemple de parcours de graph en classes (rectangles) et collaborations (ovales). Les rôles joués sont définis aux intersections

La figure Figure 3.1 empruntée à VanHilst et Notkin [57] montre cette double représentation de l'application en classes et collaborations. On peut ainsi voir les Collaborations horizontalement sous forme d'ovales et les classes verticalement sous forme de rectangles. Chaque intersection entre classes et collaborations représente le rôle joué par la classe dans la collaboration, c'est l'unité de design commune aux deux visions. Ce rôle correspond à la part de la classe qui est spécifique à la collaboration. Par exemple, le rôle de la classe *Vertex* dans la collaboration *Vertex Numbering* est d'avoir une variable d'instance contenant le numéro du noeud. Ou encore le rôle de *Graph* dans la Collaboration *Undirected Graph* est de pouvoir ajouter, supprimer et lier des noeuds entre eux.

Dépendance entre collaborations. Dans cette application, les collaborations ne sont pas complètement indépendantes les unes de autres. En effet, la collaboration *Depth First Traversal* nécessite que les fonctionnalités amenées par la collaboration *Undirected Graph* soient déjà présentes dans l'application.

Les Collaborations correspondant aux opérations sur le graphe sont indépendantes entre elles : il n'est pas nécessaire qu'elles soient toutes là, et leur ordre d'application n'est pas important. Mais elles ont besoin qu'une structure de graphe existe dans l'application pour fonctionner. Elles sont donc dépendantes des Collaborations *Depth First Traversal* et *Undirected Graph*.

Mais il est important de noter que chacune de ces Collaborations peut être remplacée par une Collaboration fournissant une fonctionnalité similaire. On pourrait ainsi très bien changer uniquement la Collaboration *Undirected Graph* pour changer la structure en un graphe orienté ou un arbre. De même on pourrait décider de faire un parcours de la structure non pas en profondeur mais largeur, en remplaçant la Collaboration *Depth-First Traversal*.

La figure Figure 3.2 illustre très bien ces dépendances entre les différentes collaborations.

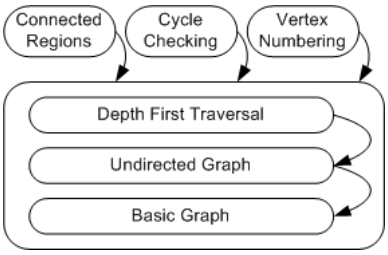

FIG. 3.2 – Représentation des dépendances entre les collaborations de l'application de parcours graphe. Les dépendances sont représentées par les flèches.

3.3 Problème avec les implémentations proposées

Tour rapide des implémentations. VanHilst et Notkin [57] ont les premiers utilisé des rôles et des mixins en C++ pour implémenter cet exemple. Ils utilisèrent les templates du C++ pour implémenter les mixins et ainsi chaque rôles. Mais dans leur modèle il n'y a quasiment pas de vérification de cohérence au sein des collaborations, pour diverse raison techniques.

Smaragdakis [51, 52] a ensuite proposé d'utiliser les mixins-layers, qui sont un moyen d'appliquer un ensemble de mixins sur un ensemble de classes. Cette méthode est très efficace pour rassembler les rôles au sein d'une même entité et pour les appliquer sur un ensemble de classes à étendre, ou pour composer des collaborations entres elles. Mais la méthode de Smaragdakis pour vérifier la cohérence des composition est compliquée et surtout ne donne pas de messages d'erreur explicites, ce qui rend impossible la réutilisation par un développeur qui ne comprend pas exactement comment marche la vérification de cohérence.

Enfin Nierstrasz et Achermann [38] ont repris cet exemple pour illustrer les mixins-layers aisément en Piccola. Mais leur modèle ne permettait pas de vérifier facilement la cohérence des mixin-layers et compositions de ceux-ci.

Cohérence. Un des problème majeur des implémentations proposées est l'absence ou la lourdeur du dispositif de résolution de conflit et de validation de la cohérence des compositions des collaborations.

Utilisation des mixin-layers. La méthode la plus utilisée pour implémenter les collaborations est donc d'utiliser des mixins via les mixin-layers. Actuellement, une technique pour implémenter les mixins est d'utiliser l'héritage pour appliquer le raffinage. Mais l'utilisation de l'héritage pose ici un problème. En effet, lorsque une application externe utilise déjà la classe que l'on veut raffiner, une fois le raffinement par héritage fait, l'application ne verra jamais les raffinements, puisque elle utilise toujours la classe non raffinée. Il faut donc modifier le code de l'application pour qu'elle voit les changements.

Chapitre 4

Contribution : un nouveau modèle de collaboration produit du couplage des Traits et des Classboxes

Cette partie va exposer le modèle de collaboration qui a pu être développé en utilisant deux paradigmes imaginés au sein du Software Composition Group de l'université de Bern, les Traits et les Classboxes. Alors que les Traits permettent une réutilisation fine de fonctionalités au niveau des classes, ils ont besoin des Classboxes pour structurer les Collaborations entre elles par leur visibilité et leur notion de couche.

Cette partie sera composée de la présentation de ces deux concepts, suivi de la présentation du modèle à proprement parler et enfin d'une évaluation de ce modèle.

4.1 Présentation des Traits

Le modèle des *Traits* [49, 47, 6, 48, 5, 12, 11] est une extension de l'héritage simple dans un but similaire au mixins, mais en évitant les problèmes qu'ils posent. Les Traits sont essentiellement des groupes de méthodes pouvant servir comme blocs de construction de classes réutilisables. De cette façon, ils permettent de factoriser les comportements similaires entre les classes et ajoutent un nouveau niveau d'abstraction entre les méthodes et une classe complète.

Un Trait consiste en un ensemble de méthodes fournies [1] qui implémentent les comportements du Trait, et de méthodes requises [2] qui paramétrisent ces comportements. De plus, les Traits ne peuvent pas définir de variables d'instance, et de même les méthodes fournies par les Traits ne peuvent pas accéder directement à une variable d'instance. Mais les Traits peuvent requérir la présence d'accesseurs par le biais des méthodes requises et être ainsi relié à l'étant de la classe qu'il compose.

Avec les Traits, le comportement d'une classe est spécifié comme une composition de Traits et quelques méthodes servant à lier les états de la classe aux Traits et les Traits entre eux,

[1] provided methods
[2] required methods

appelée méthodes de colle (*glue method*). La sémantique d'une telle classe est définie par les régles suivantes :

- *Priorité des méthodes de classe sur les méthodes définies dans un Trait de la classe.*
- *Propriété de mise à plat des classes.* Une méthode non masquée (*non-overridden*) dans un Trait a la même sémantique que la même méthode implémentée dans la classe qui contient le Trait.
- *Pas d'ordre de composition des Traits.* Tous les Traits qui composent une classe ont la même priorité, ce qui impose que tous les conflits de méthode entre les Traits doivent être résolus explicitement.

Puisque l'ordre de composition est inutile, un conflit apparaît dés que deux Traits ou plus fournissent des méthodes de même signature. Lorsque il reste un conflit dans la composition des Traits d'une classe, une interruption est déclenchée lors d'une tentative d'instantiation de la classe. Ainsi les Traits forcent la résolution explicite des conflits. Cette résolution peut-être faite en implémentant des méthodes de colle au niveau de la classe pour masquer les méthodes en conflit, ou par le *mécanisme d'exclusion* de méthode qui permet de cacher une méthode d'un Trait localement à la composition.

En plus du mécanisme d'exclusion, les Traits fournissent un *mécanisme d'alias*. Ce mécanisme permet au développeur de donner une deuxième signature pour une méthode fournie par un Trait. Ce nouveau nom peut permettre d'accéder à une méthode du Trait qui autrement aurait été inaccessible, par exemple suite à une occlusion.

Les Traits peuvent de plus être composés de sous-traits ; un Trait contenant un sous-trait est appelé un Trait composite (*composite Trait*). La sémantique de la composition est la même que celle expliquée ci-dessus, à la différence prés qu'ici c'est le Trait qui joue le rôle de la classe. Les mêmes propriétés s'appliquent donc : pas ordre de composition des sous-traits, priorité des méthodes définies dans le Trait composite sur les méthodes définies dans les soustraits, et identité des sémantiques d'une méthode dans le Trait composite ou dans un sous-trait.

Pour résumer, la composition des Traits est automatique avec une résolution explicite des conflits. Il est important de comprendre que la composition des Traits n'a pas pour but de remplacer l'héritage simple. Ces deux notions sont complémentaires. Alors que l'héritage est utilisé pour dériver une classe depuis une autre, les Traits sont utilisés pour élaborer la structure et permettre la réutilisabilité au sein de la classe. On peut résumer ces relations par cette équation :

$$Class = Superclass + State + Traits + Gluemethods$$

Exemple d'utilisation des Trais : les objets Géométriques.. Imaginons que l'on veuille représenter un objet graphique comme un cercle ou un carré qui sera dessiné sur un canevas. Un tel objet peut être décomposé en trois aspects réutilisables : sa couleur, sa géométrie et la façon dont il sera dessiné.

La figure 4.1 nous le montre dans le cas d'un cercle. On peut voir que la géométrie du cercle est exprimée dans le Trait TCircle. De même la gestion de la couleur est fournie par le Trait TColor et celle de la méthode de dessin sur le canevas par le Trait TDrawing :

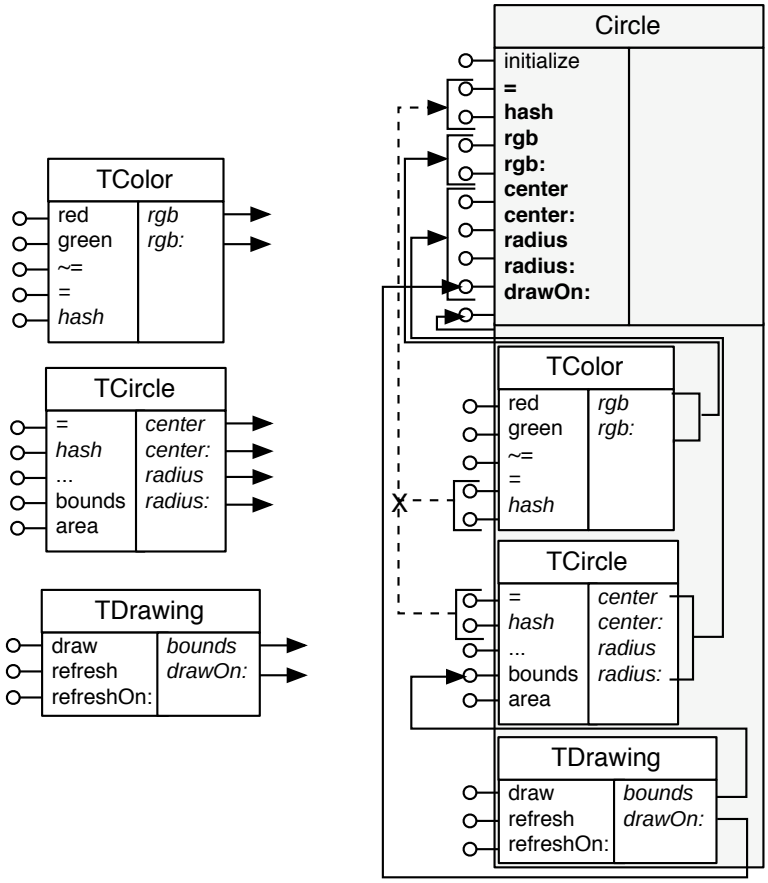

FIG. 4.1 – A gauche : trois Traits TColor,TCircle, et TDrawing. A droite : la classe Circle est composée à partir de ces trois Traits. La classe Circle résous les conflits en redéfinissant les méthodes hash et =. On peut remarquer que les figures les Traits ont leur méthodes fournies sur la gauche et les méthodes fournies sur la droite.

- TCircle définit la géométrie d'un cercle : il a besoin des méthodes center, center :, radius, et radius : et peut alors fournir des méthodes comme bounds, hash, ou encore =.
- TDrawing a pour requis les méthodes drawOn : et bounds et fournit les méthodes draw, refresh, et refreshOn :.
- TColor demande les méthodes rgb, rgb : et fournit des méthodes diverses sur la gestion des couleurs. On ne voit sur la figure que les méthodes hash and = car elle vont créer un conflit au moment de la composition des Traits.

La classe Circle est ensuite définie comme ceci : elle spécifie trois variables d'instance center, radius et rgb et leurs accesseurs respectifs. Elle est composée avec les trois Traits TDrawing, TCircle, et TColor. Comme il y a un conflit pour les méthodes hash et = entre les Traits TCircle et TColor, on effectue un alias de ces méthodes dans les deux Traits et on les masque pour résoudre le conflit.

Cette composition est exprimée dans le code SmallTalk de la classe circle qui suit. Les méthodes qui permettent de résoudre les conflits sont en gras. On peut remarquer que la syntaxe de définition d'une classe est resté quasiment le même. La seule différence est l'ajout du mot clef uses : pour introduire le paramètre contenant la composition des Traits : l'opérateur + est utilisé pour exprimer la composition entre deux Traits, l'opérateur - est utilisé pour exprimer le masquage d'une méthode d'un Trait, et enfin l'opérateur →est utilisé pour exprimer la définition d'un alias (par exemple : m1 →m2 montre que m1 est un deuxième nom pour la méthode m2).

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

Circle≫rbg                      Circle≫rgb : aNumber
 ↑rgb                            rgb := aNumber

Circle≫center                   Circle≫center : aNumber
 ↑center                         center := aNumber

Circle≫radius                   Circle≫radius : anInteger
 ↑radius                         radius := anInteger

Circle≫hash
 ↑self circleHash
   bitXor : self colorHash

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

4.2 Présentation des Classboxes

Introduction des Classboxes

Le modèle de *Classbox* [3] [2] est un système de module pour les langages orientés objets dynamiquement typé. Le modèle est actuellement implémenté en Squeak [54, 23], qui est un environnement open-source de développement en Smalltalk, mais il pourrait être porté dans des langages comme Ruby ou Python. Le but des Classboxes est de restreindre la visibilité des extensions de classe et d'agir comme un contexte [3] (un Classbox agit comme un espace de nommage [4]).

Un Classbox est un module dans l'on peut (i) définir des classes, (ii) importer des classes depuis d'autres Classboxes, (iii) étendre ces classes en leur ajoutant des variables d'instance, et enfin (iv) ajouter ou redéfinir des méthodes de ces classes.

La figure 4.2 montre deux Classboxes GraphicalElementsCB et ColoredGraphicalElementsCB. Les classes importées sont représentée par des boites en pointillés et sont reliées à leur définition originelle par une courbe en pointillés. Ainsi on peut voir par exemple que le Classbox ColoredGraphicalElementsCB importe les classes Rectangle, Line et Text depuis le Classbox GraphicalElements en redéfinissant leurs méthodes draw.

Un système d'extension de classe

Dans le contexte des Classbox, une extension de classe importée est une modification du comportement (par ajout ou redéfinition d'une méthode) ou pas extension de l'état de la classe (par ajout de nouvelles variables d'instance).

Le mécanisme d'importation des classes est relativement simple : une classe importée devient visible dans le contexte du Classbox. Un Classbox peut importer des classes depuis n'importe quel autre Classbox (que celui-ci les étende déjà où non) et les étendre.

Une extension de classe dans le contexte des Classboxes a cinq propriétés spécifiques :

Ajout ou redéfinition de méthodes. Le comportement d'une classe importée d'une autre Classbox peut être modifié par la définition de nouvelles méthodes ou la redéfinition d'une méthode existante. Dans la figure 4.2 ont peut ainsi voir que le Classbox ColoredGraphicalElementsCB définit de nouvelles méthodes et en redéfinit d'autres existantes. Chacune des classes concrètes gagne une nouvelle méthode aliasing et voit sa méthode draw redéfinie. On peut noter que l'ajout d'une méthode dans une classe la rend bien sûr accessible par ses sous-classes. On peut voir ce cas dans l'exemple, où la méthode setColor : color est ajoutée à la classe Graphical, ce qui permet par exemple aux instances de Rectangle de comprendre cette méthode (dans le contexte de la Classbox ColoredGraphicalElementsCB).

[3] Unit of scoping
[4] Namespace

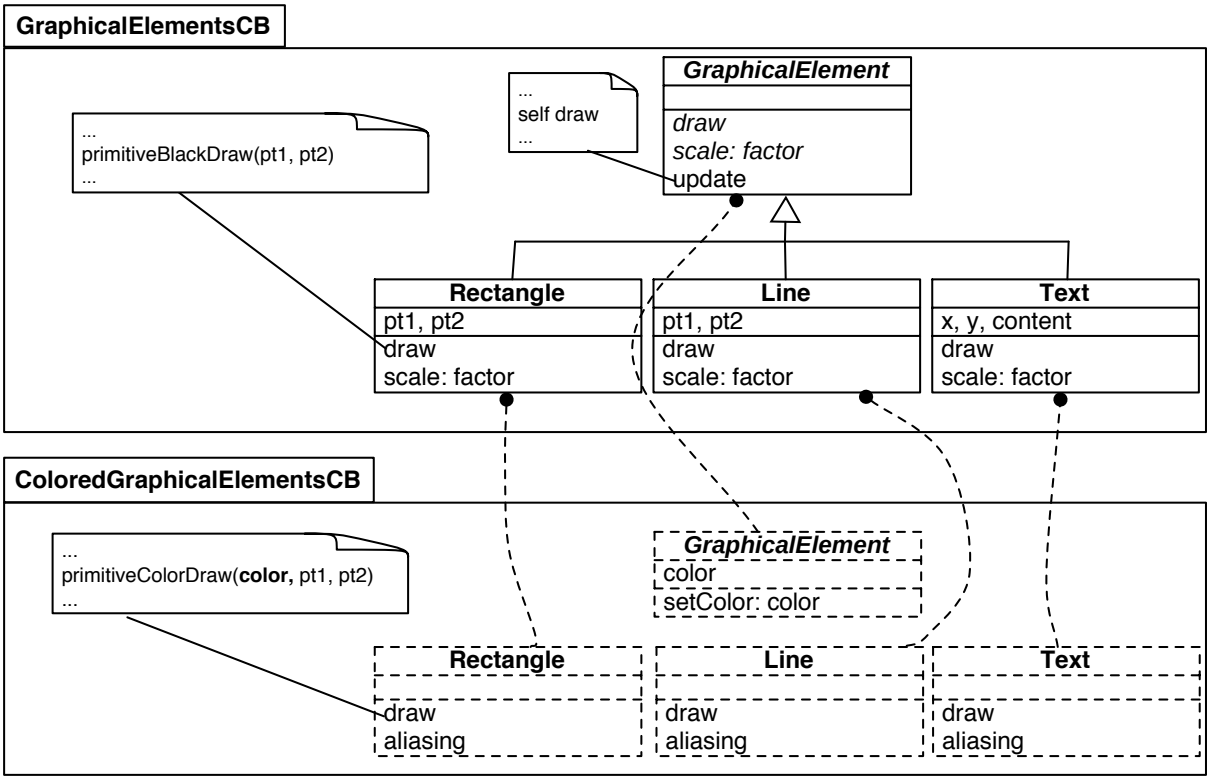

FIG. 4.2 – Exemple d'utilisation des Classboxes : Ajout d'une fonctionnalité de coloration sur une hiérarchie d'éléments graphiques.

Ajout de variables d'instance. Au sein d'un Classbox, de nouvelles variables d'instances peuvent être ajoutée dans les classes importées. On peut noter que ces variables seront bien sûr visible par les sous-classes. On peut le voir dans la figure 4.2 où le Classbox ColoredGraphicalElementsCB ajoute la variable color à la classe GraphicalElement. Cette variable est alors accessible par la nouvelle méthode setColor : color ou par les méthodes redéfinies draw des classes Rectangle, Line et Text. Lorsque un ajout de variable déjà existante dans la classe, la précédente déclaration de la variable est masquée et son ancienne valeur inaccessible. Bien sûr tous ces changements de visibilité sont valable dans le contexte de la Classbox qui définie ces modification où qui importe la classe modifiée.

Localité des extensions. Un Classbox étendant une classe restreint la visibilité des changements qu'il lui applique à lui-même et aux Classboxes qui importent cette classe. Cela signifie qu'une extension qui modifie en profondeur le comportement d'une classe n'affecte absolument pas une application qui reposerait sur la classe originelle, puisque dans le contexte de l'application, aucun changement n'a été appliqué à la classe. De la même façon, plusieurs Classbox modifiant une classe ne créent pas de conflits entre eux.

Importation des classes étendues. Toute classe visible, étendue ou non, d'un Classbox peut être importée par d'autres Classbox. Il est donc possible de faire des chaînes de Classboxes étendant incrémentalement une classe. De plus, dans le contexte de chaque Classbox il n'y a pas de différence entre une classe importée sans extension ou une classe importée et étendue. La seule restriction à cette importation est qu'il n'est pas possible de faire cohabiter deux classes ayant le même nom dans une Classbox. Les extensions faites par deux Classboxes sur une classe ne peuvent pas toutes être visibles au même moment dans un même contexte.

Liaison locale. La propriété de liaison locale [5] est la plus intéressante des Classboxes : lorsque une méthode est redéfinie par un Classbox, elle devient prioritaire sur la méthode originelle de la classe et c'est elle qui est exécutée lorsqu'elle est invoquée. Du point de vue d'une application dans le contexte du Classbox, c'est comme si la redéfinition était globale. Cette propriété n'est évidement valable que dans le cas d'une redéfinition. On peut voir un exemple de liaison locale dans la figure **??**, où la méthode draw est redéfinie dans le Classbox ColoredGraphicalElementsCB pour les classes Rectangle, Line et Text. Cette nouvelle version de la méthode est alors utilisée par les classes pour se dessiner.

[5] Local Rebinding

4.3 Notre modèle : le couplage des Traits et des Classboxes

4.3.1 Hypothèses

Rappelons tout d'abord les hypothèses de base du design par collaboration pour notre modèle :

Une collaboration a donc été définie précédemment comme un ensemble de rôles à appliquer à un ensemble d'entités pour y ajouter une fonctionnalité. Nous allons ici considérer les collaborations au niveau classe. Un rôle est donc défini au niveau d'une classe, et correspond à la partie de cette classe nécessaire à la fonctionnalité ajoutée par la collaboration. Cela peut donc être un ensemble de méthodes, mais aussi des variables et les accesseurs qui vont avec.

D'après les problématiques on doit également pouvoir inclure dans une collaboration un ensemble de règles ou contraintes définissables au cas par cas et dont la validation est nécessaire pour que la fonctionnalité ajoutée au programme par la collaboration fonctionne correctement.

4.3.2 Utilisation des Traits et des Classboxes dans l'optique des collaborations

Nous venons de voir les principes et intérêts des Classboxes et des Traits. Nous allons maintenant voir où ceux-ci sont intéressants dans le concept des collaborations.

Des Traits pour modéliser les rôles. Nous avons défini les rôles comme des 'ensemble de méthodes', ce qui correspond parfaitement à la définition d'un Trait. De plus les Traits permettent de regrouper les méthodes d'une classe par fonctionnalités, ce qui améliore la visibilité du code. De cette façon grâce aux Traits les différents rôles joués par chaque classe peuvent être bien différenciés dans une application construite avec plusieurs couches de collaborations.

Enfin, un des problèmes récurrent du concept de collaboration est que souvent une même méthode peut jouer plusieurs rôles au sein d'une même classe (par exemple 'removeVertex : aVertex' dans la classe Graph de l'application générique de graphe). Mais les mécanismes d'aliasing et de suppression de méthodes utilisables lors d'une composition de Traits permettent facilement de contourner ce problème. Le seul avantage non utilisé des Traits est la réutilisabilité, mais celle-ci est retrouvée au niveau des collaborations.

Un Classbox pour l'ajout des variables et accesseurs. Nous pouvons donc utiliser un Trait au niveau implémentation comme l'équivalent d'un rôle au niveau concept. Sauf qu'il manque l'ajout de variables et d'accesseurs, qui ne sont pas possible dans la version actuelle des Traits. Mais les Classboxes permettent de répondre à ce problème. En effet il est possible d'étendre une classe au sein d'un Classbox par des méthodes (nouvelles ou redéfinies), mais aussi par des variables. Cet ajout peut-être considéré comme une colle entre les Traits et les objets : les Traits, par leur nature non instanciable n'ont pas besoin de variables et d'accesseurs, et peuvent exister indépendamment des objets et être réutilisables. Mais pour fonctionner et ajouter des fonctionnalités au niveau des objets, ils ont besoin d'avoir un accés à l'état de l'objet, via ces

variables et accesseurs, fournis par les classboxes au moment de la liaison entre la classe et le Trait.

Un Classbox pour contenir les Traits. Les Classboxes présentent donc un grand intérêt au niveau des rôles pour l'ajout des variables et accesseurs. Mais ils sont aussi tout indiqués pour être utilisés au niveau des collaborations et réifier la notion de conteneur de rôle : en effet il est possible dans un Classbox d'ajouter ou d'importer des Traits, et d'ensuite composer ces Traits sur des classes, voire même de changer cette composition dynamiquement.

De plus les classboxes permettent de faire cohabiter plusieurs version de la même classe dans le même environnement de développement. On peut ainsi faire évoluer une classe en lui appliquant plusieurs collaborations sans avoir à faire de sous-classage, ce qui peut permettre de faire évoluer une application sans changer le code des objets qui utilise cette classe.

Un système de contrainte configurable. Le système de contrainte apporté par notre modèle sera détaillé dans la section suivante

En résumé :

1. Un rôle : un ensemble de méthodes -> un Trait
2. Une Collaboration : un conteneur de rôle + des variables et accesseurs -> un Classbox
3. Des contraintes

4.3.3 Conception de la solution

La solution se présente sous la forme d'une classe Role contenant toute les éléments caractérisant un rôle et d'une classe Collaboration contenant tous les éléments et méthodes pour créer, instancier et composer une collaboration.

La classe Role. La classe Role contient tout ce qui compose un rôle, c'est à dire des méthodes, des accesseurs et des variables :

1. une variable methods contenant un dictionnaire de méthodes (clefs = nom de la méthode) ;
2. une variable accessors contenant un dictionnaire de méthodes (clefs = nom de l'accesseur) ;
3. une variable variables contenant un dictionnaire une liste de nom de variables ;
4. les accesseurs de variables et tests.

Au niveau de la classe Collaboration. Au niveau classe, Collaboration contient tout ce qui sert à décrire une collaboration, mais sans encore l'instancier :

1. une variable roles contenant un dictionnaire de Role (clefs = nom du rôle) ;
2. une variable constraints contenant une liste de tests ;
3. les accesseurs de variables et tests.

Au niveau d'une instance de Collaboration. Au niveau instance, Collaboration contient tout ce qui sert à instancier et utiliser une collaboration :

1. un Classbox
2. une méthode initialize pour crée la Classbox, y créer les Traits correspondant aux rôles, les accesseurs et les variables ;
3. une méthode useClass :asRole : pour importer puis utiliser une classe pour tel rôle de la collaboration ;
4. une méthode useClass :asRole :fromCollaboration pour utiliser dans un rôle une classe provenant d'une autre collaboration (pour composer les collaborations) ;
5. une méthode evaluate servant à évaluer du code dans le contexte de la collaboration, avec tous les tests de cohérence nécessaire ;
6. diverses méthodes, accesseurs de variables et tests.

4.3.4 Détails sur les contraintes

Pour que la notion de collaboration soit intéressante, il est nécessaire de vérifier la cohérence de la composition comme nous l'avons vu précédemment.

Et un des gros problème dans les implémentations actuelles de la notion de collaboration est l'expression et l'application des contraintes nécessaires pour vérifier cette cohérence. Nous allons voir que notre modèle répond en partie à ce problème.

Rappelons qu'il y a deux niveaux de cohérence, la cohérence **dans** une collaboration, et la cohérence **entre** les collaborations.

Cohérence dans une collaboration. Pour qu'une collaboration soit complète, elle doit avoir tous ses rôles composés. Il faut donc d'une part que toutes les compositions sont valides, c'est à dire vérifier que les appels de méthodes entre le rôle et la classe avec laquelle il est composé sont compris, mais également que tous les rôles sont composés, pour vérifier la validité des appels entre les classes.

Par abus de langage, on peut considérer que les méthodes publiques d'une classes sont son interface externe, et que ses méthodes privées sont son interface interne. On peut alors dire que l'on doit vérifier la cohérence entre les interfaces interne des classes et des rôles qui les étendent, mais aussi la cohérence entre les interfaces externe rajoutées par les rôles aux classes.

La cohérence interne de l'extension des classes par les rôles est validée par l'utilisation des Traits. Pour deux raisons : premièrement, l'usage des Traits oblige à la résolution explicite

des conflits, donc si la composition entre le rôle et le Trait du rôle produit un conflit, il sera détecté et une erreur sera déclenchée. Deuxièmement, les Traits ont des méthodes requises pour fonctionner et renvoient une erreur lors de la composition si elles ne sont pas toutes présentes.

La cohérence entre classe est très simple à faire avec notre modèle, puisqu'il suffit de valider chaque rôle au moment où il étend une classe, et tester que tous les rôles sont validés.

Donc en partant de l'hypothèse qu'une collaboration est cohérente (ce qui est le problème du developpeur), il est possible de vérifier la cohérence de sa composition avec un ensemble de classes.

Cohérence entre les collaborations. Il faut également vérifier la cohérence de la composition entre les collaborations. En effet, l'utilisation d'une collaboration sur un ensemble de classes peut requérir la présence d'une autre collaboration composée de façon valide et cohérente sur ce même ensemble de classes. Par exemple, dans l'application de graphe, pour utiliser la fonctionnalité fournie par la collaboration Undirected Graph, il est nécessaire que la collaboration Basic Graph soit déjà composée.

Cette collaboration dépend beaucoup de la fonctionnalité fournie. Il faut donc que le développeur puisse spécifier des contraintes de cohérence au moment de la définition de la collaboration pour que celle-ci puissent être vérifier.

Dans notre modèle, il est possible de fournir des contraintes de cohérence à valider sous la forme de tests. Plus spécifiquement à l'implémentation en Smalltalk, on peut fournir des blocs dont la valeur de retour doit être true pour que la contraintes soit validée. L'implémentation des collaborations fournit également de nombreux tests sur sa structure, comme la présence des rôles, de leurs variables, accesseurs et méthodes, mais aussi la référence de la précédente collaboration composées sur le groupe de classe, ce qui permet de remonter la chaîne de composition facilement. Ainsi le développeur peut requérir la présence de telle collaboration dans la chaîne, ou de tel rôle...

4.3.5 Exemple d'utilisation du modèle : collaboration Observer/Observee

Pour utiliser la solution de collaboration, il faut procéder en trois étapes qui seront illustrées par l'utilisation du design pattern Observer/Observee comme collaboration.

Création d'une nouvelle collaboration. La création d'une nouvelle collaboration se fait en créant une nouvelle classe héritant de Collaboration :

```
Collaboration subclass : #ObserverObserveeCollaboration
    uses :
    instanceVariableNames : ''
    classVariableNames : ''
    poolDictionaries : ''
    category : 'myCategory'.
```

Puis en définissant les rôles de cette collaboration et en ajoutant à chaque rôle ses méthodes, variables et accesseurs :

```
ObserverObserveeCollaboration createNewRole : #Observee.
ObserverObserveeCollaboration addMethod : 'notifyAll self observers do :
    [ :each | each update : self]' toRole : #Observee.
ObserverObserveeCollaboration addMethod : 'addObserver : anObject
    self observers ifNil : [self observers : Set new].
    self observers add : anObject' toRole : #Observee.
ObserverObserveeCollaboration addMethod : 'removeObserver : anObject
    self observers ifNotNil : [self observers remove : anObject]' toRole : #Observee.
ObserverObserveeCollaboration addVariable : #observers toRole : #Observee.
ObserverObserveeCollaboration addAccessor : 'observers : aSet observers := aSet' toRole : #Observee.
ObserverObserveeCollaboration addAccessor : 'observers ôbservers' toRole : #Observee.
ObserverObserveeCollaboration createNewRole : #Observer.
ObserverObserveeCollaboration addMethod : 'update : anObject' toRole : #Observer.
```

Puis en définissant les contraintes de cohérence de la collaboration :

```
ObserverObserveeCollaboration addConstraint : '((self classUsingRole : #Observer) ==
    (self classUsingRole : #Observee)) not'.
```

Instanciation de la collaboration. L'instanciation de la collaboration se fait alors normalement autant de fois que nécessaire :

```
aOOCollab := ObserverObserveeCollaboration new.
```

Composition dans la collaboration. Pour définir les classes à utiliser dans la collaboration, on utilise la méthode 'useClass :asRole :'

```
aOOCollab useClass : #Obj1 asRole : #Observer.
aOOCollab useClass : #Obj2 asRole : #Observee.
```

Execution de code dans la collaboration. Et enfin, pour exécuter du code dans le contexte de la collaboration, on utilise la méthode 'evaluate' :

```
aOOCollab evaluate : '
    | o1 o2 |
    o1 := Obj1 new.
    o2 := Obj2 new.
    o2 addObserver : o1.
    o2 notifyAll
'
```

4.3.6 Exemple de composition : l'application de graphe

Nous allons voir maintenant l'application de graph présentée dans la partie *Exemple de motivation* réalisée avec notre solution.

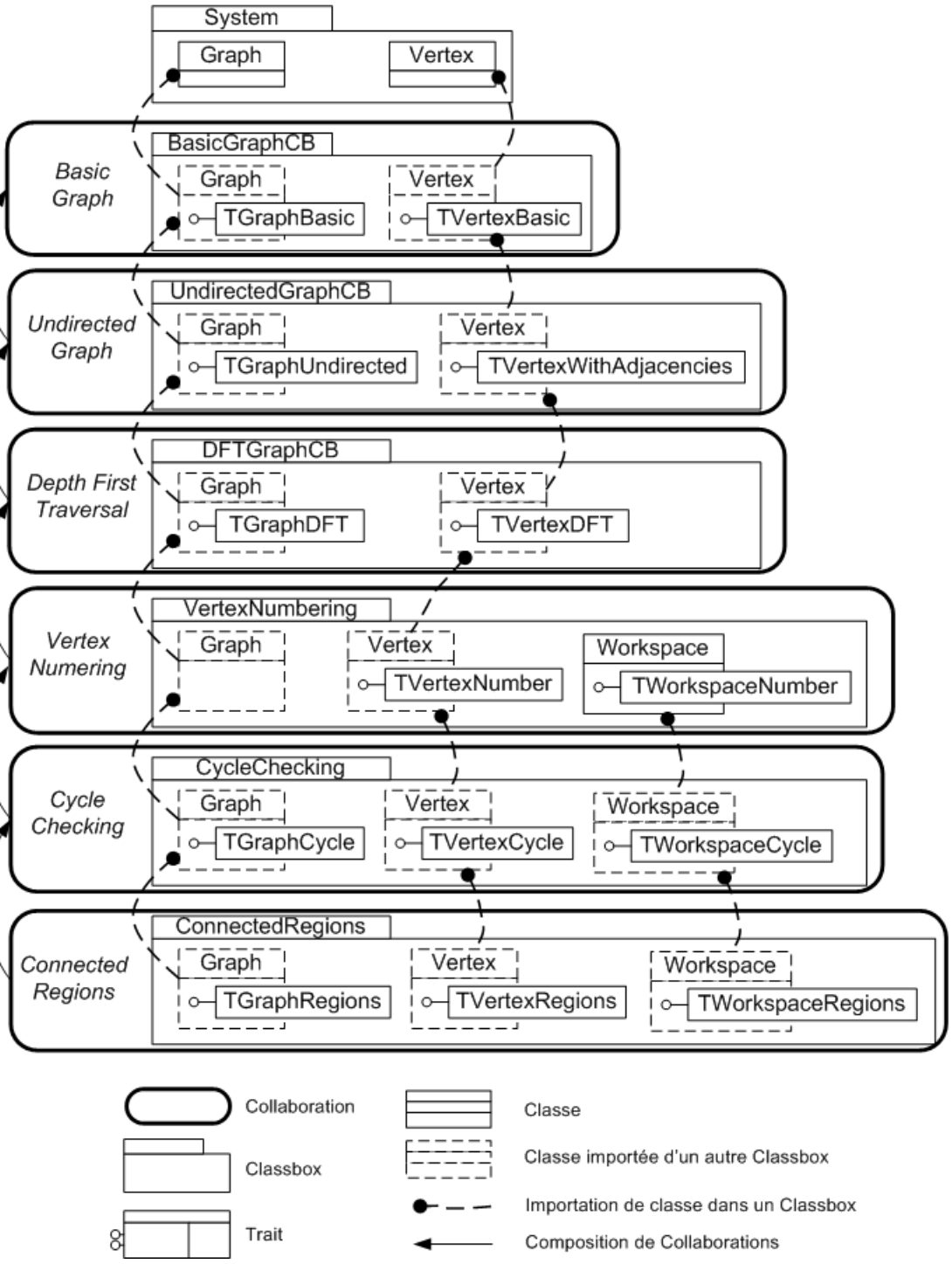

FIG. 4.3 – L'application de parcours de graphes décomposée en collaboration selon notre modèle. Les collaborations sont représentées par des ovales en gras, leurs composition par des flèches, et les autres conventions de présentation des Traits et Classboxes sont respectées.

La figure 4.3 montre un exemple de composition de l'application avec ses différentes collaborations qui sont représentées par un oval à bord épais. Chacune est composée de son Classbox, et de ses Traits. On voit également sur cette figure les compositions incrémentales des collaborations (compositions représentées par des flèches) et les importations des classes dans les Classboxes. A noter que le Classbox System correspond à toutes les classes existant avant la définition des Classboxes.

Les variables et accesseurs sont volontairement masqués sur la figure 4.3 pour ne pas la surcharger. Il existe d'autres compositions possibles pour cette application, car comme nous l'avons vu les trois dernières collaborations n'ont pas de dépendances entres elles et il est donc possible de changer leur ordre de composition.

La figure 4.4 montre les détails de la composition des deux premières collaborations de l'applications, BasicGraph et UndirectedGraph pour le cas de la classe Graph. Elle ne détaille que la classe Graph dans un souci de clarté. De même, dans mon explication, je ne parlerais que de la classe Graph.

On peut voir sur cette figure l'application de la première collaboration *Basic Graph* sur les classes Graph et Vertex. Ces deux classes sont importées dans le Classbox de la collaboration, BasicGraphCB, et leurs rôles leurs sont ajoutés : ainsi le Trait TGraphBasic est ajouté à la classe Graph au moment de la composition, ainsi qu'une nouvelle variable d'instance, vertices, et deux accesseurs vertices et vertices :.

Lors de l'application de la deuxième collaboration, les classes sont importées de la collaboration précédente, *i.e.*,de la Classbox de la collaboration précédente, et leurs rôles leurs sont ajoutés : on ajoute le Trait TGraphUndirected à la classe Graph (dans cette collaboration, le rôle joué par la classe Graph n'a pas de variable ni d'accesseur).

Cette composition est intéressante pour le cas de la méthode removeVertex : du rôle GraphUndirected de la collaboration Undirected Graph. En effet, dans l'application de graphe, cette méthode est déjà définie par la collaboration Basic Graph par le rôle GraphBasic : un graphe défini comme un ensemble de vertices doit pouvoir supprimer un de ces vertex. Mais avec l'ajout de l'indirection par la deuxième collaboration, cette méthode doit être redéfinie pour prendre en compte le voisinage des vertices (cf.le code de l'application en annexe pour plus de détails). Cette redéfinition est faite grâce aux propriétés d'aliasing et d'exclusion des Traits : en combinant les deux propriétés, la méthode removeVertex : originelle est renommée en oldRemoveVertex :. Et ainsi le rôle GraphUndirected de la collaboration Undirected Graph peut définir une méthode removeVertex : utilisant la méthode renommée oldRemoveVertex :.

La composition de Trait nécessaire à cette opération est la suivante :

```
Graph uses : {
        TBasicGraph
        @ { #oldRemoveVertex : -> #removeVertex : }
        - { #removeVertex : }
        + TUndirectedGraph
}
```

Voyons maintenant rapidement comment cette composition s'exprime du point de vue du code :

Instanciation de la collaboration.

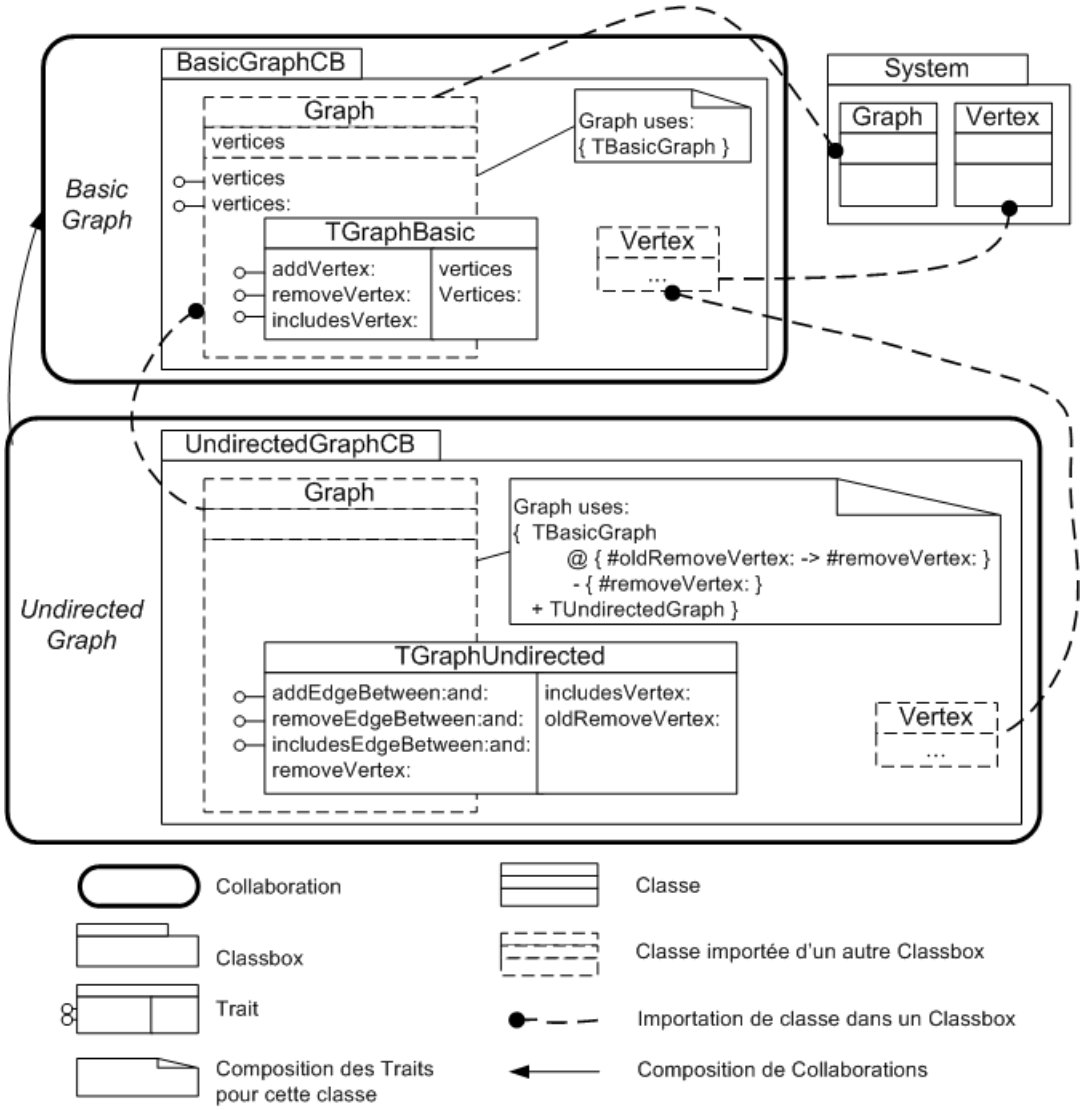

FIG. 4.4 – Détail de la composition des collaborations *Basic Graph* et *Undirected Graph* sur la classe Graph. Les collaborations sont représentées par des ovales en gras, leurs composition par des flèches, et les autres conventions de présentation des Traits et Classboxes sont respectées

```
aBGCollab := BasicGraphCollaboration new.
aUGCollab := UndirectedGraphCollaboration new.
```

Composition dans la collaboration Basic Graph.

```
aBGCollab useClass : #Graph asRole : #GraphBasic.
aBGCollab useClass : #Vertex asRole : #VertexBasic.
```

Composition dans la collaboration Undirected Graph.

```
aUGCollab useClass : #FMCTGraph
    asRole : #GraphUndirected
    fromCollaboration : aBGCollab
    withChanges : [ :previousComposition | previousComposition
        @{#oldRemoveVertex :->#removeVertex :}
        - {#removeVertex :} ].
aUGCollab useClass : #FMCTVertex asRole : #VertexWithAdjacencies fromCollaboration : aBGCollab.
```

Execution de code dans la collaboration.

```
assertresult := aBGCollab evaluate : '
    | aGraph v1 v2 v3 |

        aGraph := FMCTGraph new.
        v1 := FMCTVertex new value : 1.
        v2 := FMCTVertex new value : 2.
        v3 := FMCTVertex new value : 3.

        aGraph addVertex : v1.
        aGraph addVertex : v2.
        aGraph addVertex : v3.

        aGraph addEdgeBetween : v1 and : v2.
        aGraph addEdgeBetween : v1 and : v3.
        aGraph removeEdgeBetween : v1 and : v2.
        aGraph addEdgeBetween : v2 and : v3.

        aGraph removeVertex : v1.
'.
```

4.4 Evaluation de notre solution

Dans cette section, nous allons rapidement évaluer notre solution de collaboration selon les problèmatique précedemment identifiées.

Composition de collaborations. Dans le cadre de la composition de collaboration, notre modèle répond parfaitement à la problèmatique. En effet, il permet de définir des collaborations,

puis de les instancier, et ensuite de les composer de façon simple. La composition d'une collaboration sur un groupe de classes fonctionne bien, et il est possible de faire des compositions de collaborations de façon itératives facilement.

Cohérence dans une collaboration. La cohérence au sein d'une collaboration est une problèmatique bien résolue par notre modèle. En effet, avec l'utilisation des Traits, les problèmes de conflits internes aux classes une fois composées doivent être explicitement résolus. Et les erreurs renvoyées par les Traits sont explicites et dirigent rapidement le développeur vers la source du problème. Et de plus, la vérification de cohérence entre les rôles, *i.e.*,la vérification que tous les rôles sont appliqués au sein de la collaboration, est fait aisement par notre modéle.

Cohérence entre les collaborations. La cohérence entre les collaborations est spécifique à l'application, et nécessite donc une définition par le développeur. Dans notre modèle, il est possible d'ajouter autant de contraintes que l'on désire, qui seront toutes vérifiées pour valider la cohérence. Ces contraintes peuvent utiliser une *API* simple mais compléte de tests divers (présence d'un rôle, méthode, variable, accesseur dans la collaboration...) au sein d'une collaboration et de navigation le long de la chaîne de composition des collaborations. Il est donc possible de spécifier des contraintes globales relativement compléxes et efficaces. Malheureusement je n'ai pas eu le temps de définir et de tester un grand nombre de ces contraintes...

Comparaison avec les implémentations existantes. Nous avons vu que les implémentations existantes de design par collaborations ne répondent pas à toutes les problèmatiques. Notre modèle semble donc plus robuste et complet. Néanmoins, il n'a pas été possible de comparer plus en détails les implémentations existantes et notre modèle pour des problèmes logisitique (pas de temps, pas le code des autres implémentations...). Il sera intéressant de le faire dans un travail futur.

Chapitre 5

Conclusion

Lors de la réalisation de cette étude, nous avons pu montrer que la réutilisation est un enjeu important dans le génie logiciel. Or les solutions existantes que nous avons étudié ne réponde pas complétement à ce besoin. Nous avons vu que la programmation par objet répond en partie à cette problèmatique mais ne supporte pas le passage à l'échelle pour une utilisation de la réutilisabilité entre plusieurs applications. Et elle ne peux pas répondre au besoin de définition et de gestion des fonctionnalités transverses à l'application. Nous avons également vu au travers de l'étude des Aspects, Composants, Modules et Mixins que ceux-ci ont leur avantages et inconvénients dans le cadre de la réutilisation de fonctionnalités, mais qu'il n'y a pas actuellement de modèle parfait répondant à ce problème de façon compléte.

Le problème est donc toujours ouvert et c'est en essayant d'y répondre que nous avons décidé d'étudier la notion de collaboration. Cette notion a pour but de réifier des fonctionnalités transverses aux classes d'une application. Une fonctionnalité transverse au sein d'un programme étant composée de comportements issus de plusieurs classes dans l'application, les collaborations se placent à une granularité particulière : une collaboration est composée d'un ensemble de méthodes appartenant à plusieurs classes. Il devient donc intéressant de pouvoir décomposer les fonctionnalités d'une application en différents rôles et d'ensuite composer ces collaborations pour créer l'application compléte.

Dans ce cadre, l'utilisation des collaborations requiert un certain nombre de vérifications de cohérence pour être validée. Il faut en effet vérifier les cohérences internes aux collaboration, et vérifier que celles-ci sont bien composées sur leurs ensemble de classe. Mais il faut aussi vérifier la cohérence et validité des compositions externes, c'est à dire vérifier que les compositions entre les collaborations sont valides.

Nous avons étudié les implémentations actuelles du modèle des collaborations et remarqué que si elles arrivent sans problème à exprimer la notion de collaboration et à permettre les compositions entre les collaborations, leurs résultats ne sont pas très intéressant du point de vue de l'expressivité des contraintes de cohérence et de leur validation.

Nous avons donc évalué la possibilité d'une combinaison des deux concepts de Traits et de Classboxes pour créer un nouveau modèle de collaboration. Après avoir explicité les deux concepts, nous avons présenté notre solution puis nous l'avons illustré par quelques exemples. Nous avons ainsi pu montrer que la gestion des conflits des Traits ajouté au pouvoir de modu-

larisation des Classboxes permet d'avoir un modèle de collaboration valide.

Perspectives. Dans le cadre de la validation de notre solution, nous n'avons pas eu le temps ni les moyens de vraiment faire des tester les implémentations existantes et de les comparer quantitativement à notre modéle. Il aurait été intéressant de faire des comparaisons de performance entre les différents modèles. Cela fera partie d'un travail futur.

Bibliographie

[1] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam — a smooth extension of Java with mixins. In *ECOOP 2000*, number 1850 in Lecture Notes in Computer Science, pages 145–178, 2000.

[2] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. The classbox module system. In *Proceedings of the ECOOP '03 Workshop on Object-oriented Language Engineering for the Post-Java Era*, July 2003.

[3] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes : A minimal module model supporting local rebinding. In *Proceedings of JMLC 2003 (Joint Modular Languages Conference)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003. Best award paper.

[4] Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10) :51–57, 2001.

[5] Andrew P. Black and Nathanael Schärli. Traits : Tools and methodology. In *Proceedings ICSE 2004*, May 2004. To appear.

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

[7] D. Box. *Essential COM*. Addison-Wesley, 1998.

[8] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/E-COOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[9] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava : Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

[10] Componentsource. the evolution of components.

[11] Gael Curry. An approach to type safety in a traits system. In *Proceedings of 1986 joint computer conference*, pages 25–30, 1986.

[12] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. TRAITS : an approach to multiple inheritance subclassing. In *Proceedings ACM SIGOA, Newsletter*, volume 3, Philadelphia, June 1982.

[13] Kris de Volder. Aspect-oriented logic meta programming. *Meta-Level Architectures and Reflection, 2nd International Conference on Reflection, LNCS*, 1616, 1999.

[14] Bruneton E, Coupaye T, and Stefani J. B. The fractal composition framework. version 1.0., June 2002.

[15] Sun microsystems. enterprise javabeans specification version 2.0.

[16] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[17] Erik Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP 2001*, LNCS, pages 303–326. Springer Verlag, 2001.

[18] Matthew Flatt and Matthias Felleisen. Units : Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[19] Object Management Group. Corba components, version 3.0. OMG Specification formal/02-06-65, Object Management Group, 2002.

[20] Stephan Herrmann. Object confinement in Object Teams – reconciling encapsulation and flexible integration. In *3rd German Workshop on Aspect-Oriented Software Development*. SIG Object-Oriented Software Development, German Informatics Society, 2003.

[21] Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays 2002*, pages 216–232, Erfurt, 2003. Springer.

[22] Ian M. Holland. Specifying reusable components using contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 287–308, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[23] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future : The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[24] Sun microsystems. java beans specification.

[25] Jboss aspect oriented programming.

[26] Julia, une implémentation du framework fractal.

[27] Gregor Kiczales. Aspect-oriented programming : A position paper from the xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. ?, 1996.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, 2001.

[29] Günter Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.

[30] Cristina Videira Lopes. D : A language framework for distributed programming. Technical Report SPL-97-010, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston MA 02115, 1997.

[31] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual classes : A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 397–406, October 1989.

[32] Theo Dirk Meijler and Oscar Nierstrasz. Beyond objects : Components. In M.P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems : Current Trends and Directions*, pages 49–78. Academic Press, November 1997.

[33] Tom Mens and Marc van Limberghen. Encapsulation and composition as orthogonal operators on mixins : A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1) :1–30, 1996.

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

[35] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[36] Mira Mezini and Klaus Ostermann. Modules for crosscutting models. In *8th International Conference on Reliable Software Technologies (Ada-Europe '03)*. svlncs, June 2003.

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

[38] Oscar Nierstrasz and Franz Achermann. Supporting Compositional Styles for Software Evolution. In *Proceedings International Symposium on Principles of Software Evolution (ISPSE 2000)*, pages 11–19, Kanazawa, Japan, November 2000. IEEE.

[39] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[40] ohn Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming os sparse matrix code. *International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), LNCS*, 1343, 1997.

[41] Harold Ossher and Peri Tarr. Hyper/J : multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[42] David L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12) :1053–1058, December 1972.

[43] M. Südholt R. Douence. A model and a tool for event -based aspect-oriented programming (eaop). *LMO*, 1343, February 2003.

[44] E. Roman, S. Ambler, and T. Jewell. *Mastering Enterprise JavaBeans, second edition*. Wiley Computer Publishing, 2002.

[45] Patricio Salinas. Adding systemic crosscutting and super-imposition to composition filters. Emoose msc. thesis, Vrije Universiteit Brussel, 2001.

[46] R. Sanlaville. *An Architectural Environment for Dassault Systemes*. Ph.D. thesis, University of Grenoble, 2001.

[47] Nathanael Schaerli, Stéphane Ducasse, and Oscar Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, 2002.

[48] Nathanael Schärli. Traits—composable units of behavior, September 2003. http ://www.iam.unibe.ch/∼scg/Research/Traits.

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

[50] Dave Simmons. Smallscript, 2002. http ://www.smallscript.com.

[51] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 550–570, Brussels, Belgium, July 1998.

[52] Yannis Smaragdakis and Don Batory. Implementing reusable object-oriented components. In *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[53] Yannis Smaragdakis and Don Batory. Mixin layers : an object-oriented implementation technique for refinements and collaboration-based designs. *ACM TOSEM*, 11(2) :215–255, April 2002.

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

[55] Clemens A. Szyperski. Import is not inheritance — why we need both : Modules and classes. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, volume 615 of *LNCS*, pages 19–32, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[56] Sander Tichelaar. A coordination component framework for open distributed systems. Master's thesis — software composition group, University of Groningen, NL — University of Bern, CH, May 1997.

[57] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer Verlag, 1996.

[58] S. Vinoski. Corba : Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, February 1997.

[59] Allen Wirfs-Brock. Subsystems — proposal. OOPSLA 1996 Extending Smalltalk Workshop, October 1996.

[60] Allen Wirfs-Brock and Brian Wilkerson. An overview of modular Smalltalk. In *Proceedings OOPSLA '88*, pages 123–134, November 1988.

[61] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

Table des figures

Table des matières

LINA - EMN/INRIA OBASCO - SCG

VERS UNE MODÉLISATION TRANSVERSE ET MODULAIRE DES COLLABORATIONS PAR COUPLAGE DES TRAITS ET DES CLASSBOXES

Crosscutting and scoped collaborations

Florian MINJAT
(encadré par Pierre Cointe et Stéphane Ducasse)

Résumé

Dans le domaine de la conception de langages, la réutilisation et la factorisation du code sont deux enjeux majeurs. Dans le but de répondre à ces questions, de très nombreuses solutions ont été proposées, avec plus ou moins de succès. Mais aucune n'a répondu de manière parfaite au problème qui reste ouvert. Le concept de collaboration est ainsi très intéressant pour la réutilisation de fonctionnalités transverses, mais aucun modèle de ce concept ne permet de rendre compte des problèmes induits. Dans ce rapport nous introduisons un modèle original de collaboration basé sur les concepts de Traits et de Classboxes, développés par le Software Composition Group de l'université de Bern, que nous illustrons au travers de deux exemples. Après évaluation, il s'avère que ce modèle répond de manière simple et explicite aux problématiques des collaborations.

Termes généraux : design par collaboration, rôles, réutilisation, fonctionnalité, aspects, composants, mixins, modules, Traits, Classboxes