N° d'ordre : 3998

Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Unification des approches par aspects et à composants

THÈSE

présentée et soutenue publiquement le Juin 2007

pour l'obtention du

Doctorat de l'université des Sciences et Technologies de Lille
(spécialité informatique)

par

Nicolas Pessemier

Composition du jury

Président :	M. Pierre Boulet, *Université de Lille I*
Rapporteurs :	M. Pierre Cointe, *École des mines de Nantes* M. Jean-Bernard Stefani, *INRIA Rhône-Alpes*
Examinatrice :	Mme Françoise André, *IRISA Rennes*
Encadrants :	M. Lionel Seinturier, *Université de Lille I* M. Thierry Coupaye, *France Telecom R&D*
Directeur de thèse :	Mme Laurence Duchien, *Université de Lille I*

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022 — INRIA Futurs

Mis en page avec la classe thloria.

Remerciements

Je tiens en tout premier lieu à remercier les membres du jury : les deux rapporteurs de cette thèse Jean-Bernard Stefani et Pierre Cointe, l'examinatrice Françoise André, et Pierre Boulet d'être le président de ce jury.

Je remercie chaleureusement Lionel Seinturier pour son encadrement durant cette thèse. J'ai énormément apprécié ton encadrement pendant ces trois années. Ton grand recul et ton cadrage m'ont permis de canaliser mes travaux de recherche, tout en me laissant une grande marge d'autonomie. Tu as su à chaque me donner les bons conseils et orientations lorsque j'étais en proie au doute ou à l'hésitation.

Je remercie également Laurence Duchien directrice de cette thèse et Thierry Coupaye encadrant, pour leurs conseils avisés me donnant à chaque fois un éclairage tout particulier sur mes travaux de recherche.

Je souhaite remercier Johan Fabry et Denis Conan pour leur collaboration sur les deux études de cas présentées dans ce document. Je garde un très bon souvenir de cette collaboration qui a été fructueuse en débats et idées neuves, J'ai beaucoup appris notamment de votre rigueur et organisation.

Je tiens à remercier les thésards de l'équipe avec qui j'ai passé de bons moments lors de pérégrinations en France et à l'étranger pour des conférences et ateliers : Emmanuel Renaux, Romain Rouvoy, Dolores Diaz, Carlos Noguera, Frédéric Loiret et Areski Flissi.

Je remercie tout particulièrement Olivier Caron pour les agréables moments au stade du LOSC, et les longs débats autour de la vie footballistique lilloise.

Je remercie tous les anciens de l'équipe qui nous ont manqué fortement après leur départ : Olivier Barais, Alexis Muller, Jérome Moroy, Nicolas Petitprez, Christophe Demarey et Christophe Contreras et Renaud Pawlak.

Je remercie les ingénieurs et thésards de l'équipe animant les pauses cafés : Nicolas Dolet, Ales Plsek, Jeremy Dubus, Guillaume Dufresne et Guillaume Waignier et tous ceux que j'aurais oublié. Je souhaite bonne chance à ceux qui doivent soutenir leur thèse dans les années à venir.

Enfin, je remercie pour leur soutiens mes parents, ma petite soeur et mes amis proches, en particulier Julien et Sonia.

Résumé

Cette thèse adresse le rapprochement de deux paradigmes mettant en oeuvre le principe de séparation des préoccupations en ingénierie logicielle : les approches par aspects et les approches à composants. Ces différents types d'approche constituent un enjeu majeur pour le développement d'intergiciels adaptatifs et à large échelle et se positionnent aux limites de l'approche par objets. Les aspects peuvent apporter aux composants logiciels un support pour les propriétés transverses d'un système à base de composants. Réciproquement, les composants apportent aux aspects des propriétés structurantes, ce qui permet de gagner en modularité ainsi qu'en abstraction à l'aide de la vue architecturale proposée par les langages d'architecture.

Notre proposition, FAC pour Fractal Aspect Component, est construite comme une extension du modèle à composants Fractal qui est un modèle général, hiérarchique, réflexif et extensible. Notre extension offre à Fractal un support pour la gestion des préoccupations transverses. Un travail d'unification est réalisé entre les principes fondamentaux de l'approche par aspects et les propriétés structurantes de l'approche à composants. Nous validons notre étude par deux études de cas sur les transactions étendues et sur la communication de groupe. Les spécifications de FAC sont implantées en Java par Julius, une extension de Julia, l'implantation de référence de Fractal en Java.

Abstract

This thesis focuses on the unification of two software engineering paradigms applying separation of concerns : aspect-based approach and component-based approach. These two kinds of approaches constitute a major point of interest for the development of large-scale adaptive middleware and go beyond the limits of object-based approach. Aspects can bring to components a support for the modularization of crosscutting properties of a component-based system. Conversely, components give to aspects more structural properties, which improve modularity and provide more abstraction using architectural views using architecture description languages.

Our proposal, Fractal Aspect Component (FAC) is built as an extension of the Fractal component model, which is a general, hierarchical, reflexive and extensible model. Our extension provides to Fractal a support for the management of crosscutting concerns. Our work unifies the fundamental principles of aspect-based approach and structural properties of the component-based approach. We validate our proposal with two case studies on advanced transaction management and group communication systems. Julius, an extension of Julia, the reference implementation of Fractal in the Java language, implements FAC specifications.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

1.1 Motivation et objectifs

Depuis les débuts de l'ingénierie logicielle, la complexité des systèmes à prendre en compte n'a cessé de croître. De nombreux outils, méthodes, canevas, et architectures ont été proposés pour maîtriser la conception, l'implémentation, le déploiement et la maintenance tout en respectant les spécifications et les échéances établies en amont. Dans cette évolution au fil des ans, on a vu apparaître le principe d'intergiciel, qui fournit une couche intermédiaire permettant d'abstraire un certain nombre de services pour simplifier la mise en œuvre du logiciel dans tous ses cycles de développement. Couche après couche, l'ingénierie logicielle n'a fait que grandir en pouvoir d'abstraction. Cependant, quelque soit l'intergiciel considéré, le principe de séparation des préoccupations a toujours tenu une place primordiale. Il permet de diviser un logiciel en parties plus petites pour en maîtriser la complexité. Lorsque ces parties sont considérées indépendamment leur conception, développement, adaptation, évolution , ou maintenance en est fortement facilité. Sur ce principe de base, de nombreux courants ont vu le jour comme la programmation par objets, et plus récemment le développement par composants et le développement par aspects.

Le développement par objets Se basant sur les concepts d'encapsulation, d'héritage et de polymorphisme, le développement par objets a su apporter une avancée certaine en termes de réutilisation et de séparation des préoccupations [Meyer, 1997]. Un objet dont la représentation statique est appelée une classe, permet de modéliser les éléments, idées ou entités du monde physique facilement en reposant sur le principe simple d'attribut et de méthode. Les attributs de classe caractérisent les données, là où les méthodes en sont les fonctions, les actions. La réutilisation est renforcée par le mécanisme d'héritage qui introduit une hiérarchie de type parent/enfant entre les classes. Les classes filles héritent donc des classes parentes les méthodes et attributs définis, ce qui permet de spécialiser facilement le rôle joué par une classe. Cependant, ce mécanisme d'héri-

tage a été rapidement décrié à travers le problème de *fragile base class* qui identifie des problèmes importants d'incohérences au niveau des classes filles, lorsqu'une classe parente est modifiée. L'encapsulation est un mécanisme qui permet de protéger les données ou les accès aux méthodes des objets et qui en renforce donc la modularité. Enfin, par le polymorphisme, une classe enfant peut redéfinir certains comportements des classes dont elle hérite.

Néanmoins, les objets n'ont pas complètement tenus leur promesses en termes de réutilisation et de séparation des préoccupations. Les approches à composants [Szyperski, 2002] et les approches par aspects [Kiczales et al., 1997] se sont attaquées aux points faibles des objets.

Le développement par composants Le développement par composants améliore l'approche par objets en termes de spécification des dépendances entre objets ainsi, et de gestion du déploiement. De ce fait, on voit apparaître la notion de service délimités par le biais d'interfaces contractualisées, et une composition plus forte des composants, là où les relations entre objets étaient limitées à la structuration en paquetages qui correspond à une distribution physique des fichiers des classes dans une hiérarchie de répertoires. Dans le développement par composants, les préoccupations métiers sont incarnées dans des composants offrant chacun un ensemble de services qui sont liés fonctionnellement. Szyperski définit un composant logiciel comme une entité dotée d'interfaces contractuelles, conditionné de telle sorte qu'il soit déployé indépendamment, et sujet à composition par un tiers [Szyperski, 2002]. Cette définition est de loin la plus connue concernant les composants. Elle met l'accent, en particulier, sur le déploiement et la réutilisation des composants. Les interactions entre composants sont mis de côté.

Parfois, les liens fonctionnels entre composants donnent lieu au concept de liaison ou de connecteur, permettant ainsi de renforcer l'indépendance des composants et leur modularité. C'est ce que défendent les langages d'architecture (*ADL – Architecture Description Language*) [Medvidovic and Taylor, 2000]. Ces langages permettent de décrire structurellement une architecture à composants ainsi que leurs interactions. Les langages d'architecture définissent en général les quatre notions de composant, liaison, interface et composite. Les services sont spécifiés contractuellement par les interfaces qui peuvent être requises ou fournies par les composants. Ensuite des liaisons sont établies entre les interfaces. Un composant contenant d'autre composant est appelé un composite. Ainsi, par composition on obtient une architecture complète décrite par le langage, indépendamment de l'implantation des composants. Ces langages permettent de gagner en abstraction.

Une autre caractéristique importante du développement par composants est la séparation stricte observée entre services fonctionnels (encapsulés par les composants) et services techniques. Ces derniers sont généralement pris en charge par des conteneurs à composants. Ces conteneurs facilitent l'instanciation et le déploiement des composants. Ils fournissent un certain nombre de services techniques comme la gestion des transactions ou de la persistance des données. L'intégration des services techniques n'est pas une tâche aisée. Bien souvent elle impose un éparpillement des appels au code technique depuis le code fonctionnel des composants. En effet, les composants possédant un service transactionnel doivent définir du code dit de démarcation qui délimite les différentes phases d'une transaction : le commencement, la validation, l'abandon. Ces différentes phases sont localisées de manière éparse dans le code de ce service, et de plus, peuvent impliquer de nombreux composants. Il en résulte un éparpillement de code d'ordre technique dans le code fonctionnel. Ce phénomène a été identifié par l'approche par aspects que nous détaillons maintenant.

Le développement par aspects Dans le développement par aspects, la modularité s'exprime à travers la séparation des préoccupations transverses [Kiczales et al., 1997]. Le code transverse peut être modularisé par de nouvelles entités appelées des aspects et chaque partie (code fonctionnel et aspects) peut être développée, maintenue et améliorée en parallèle. Ensuite, les aspects

sont tissés au code fonctionnel pour former l'application complète. A la base du développement par aspects repose donc l'identification du phénomène d'entrelacement de code, qui a été largement identifié au niveau des services techniques. Ce code éparpillé limite fortement la modularité des services techniques ainsi que celle des composants fonctionnels impliqués. L'approche par aspect ne se limite cependant pas aux seuls services techniques. Toute fonctionnalité transverse peut être modularisée par un aspect. Par définition, l'approche considère que toute décomposition conduit à un moment donné à ces phénomènes d'entrelacement de code (phénomène dit de la décomposition dominante) [Tarr et al., 1999]. L'approche par aspect fournit donc des méthodes pour une correcte modularisation de ce code.

Les approches par aspects n'ont pas les propriétés structurantes fortes des composants. Elle n'imposent pas un système de décomposition donné. Au contraire, l'approche par aspects se définit plutôt comme une approche complémentaire. Elle offre la capacité d'adapter le comportement d'éléments dits de base (par opposition aux aspects). Ces éléments s'appliquent donc potentiellement à de nombreux autres paradigmes, les éléments peuvent être des modules, des composants, ou des objets. Malgré le fait que l'approche par aspects soit apparue depuis maintenant plus de dix ans, l'immense majorité des recherches et études de cas réalisées jusqu'ici se sont focalisées sur la partie liaison des aspects : les langages de coupe. Ces langages sont utilisés pour sélectionner les éléments de base qui seront sous le contrôle des aspects. Si cette partie joue un rôle primordial, car caractérisant le pouvoir de composition des aspects, il n'en demeure pas moins que la structure du comportement des aspects eux-même a souvent été mis de côté. Ce comportement est même souvent caractérisé par des extensions de langage de programmation, rendant les aspects et les éléments de base asymétriques. L'asymétrie limite, entre autres, la capacité des aspects à être réutilisés ou à évoluer. Il semble donc naturel de regarder en quoi les propriétés structurantes des composants peuvent venir en aide aux aspects pour gagner en modularité et en réutilisation.

Vers un rapprochement des deux styles de développement L'approche à composants, malgré ses propriétés structurantes fortes, subit donc la loi de la décomposition dominante pointée dans la communauté aspect. Il semble donc naturel de vouloir appliquer les bonnes pratiques amenées par l'approche par aspects aux composants logiciels. Parallèlement, les aspects auraient beaucoup à gagner à être structuré comme des composants et à contractualiser et encapsuler correctement les aspects, pour gagner en modularité.

Notre objectif est de rapprocher le développement par composants et par aspects. Cependant nous considérons que les composants logiciels et les objets ne sont pas dotés des même propriétés et principes, et ne doivent en conséquence pas être traité de manière identique. Le paradigme servant comme base aux aspects doit être fortement pris en compte dans la proposition. Par conséquent, nous cherchons donc à garder les bonnes propriétés des deux styles de développement, à savoir :

+ les propriétés structurantes des composants (interface offertes et requises, entité modulaire et réutilisable),
+ la faculté des aspects à réduire l'entrelacement de préoccupations dans le code.

Le rapprochement doit également réduire les points faibles de chacun des deux styles de développement, à savoir :

+ le manque de support pour la modularisation des préoccupations transverses des composants,
+ le manque de propriétés structurantes, les faiblesses en visualisation et en composition des aspects.

Enfin, rapprocher ces deux styles de développement peut introduire de nouvelles problématiques, et notre approche devra également répondre à la question suivante :

+ Comment concilier d'une part l'encapsulation forte des composants logiciels, qui définissent des services contractualisés, et d'autre part le côté envahissant des aspects, qui modifient le comportement des éléments de la base, c'est-à-dire potentiellement les contrats des composants.

L'objectif principal de notre travail reste l'obtention d'une séparation des préoccupations optimale en combinant deux approches ciblant cette propriété mais n'y arrivant pas complètement individuellement.

1.2 Contexte de travail

Ce travail de thèse a été financé par un contrat de recherche externe France Télécom R&D -INRIA sous le numéro 46131097, et réalisé au sein de l'équipe ADAM, précédemment l'équipe JACQUARD. Cette équipe est sous la tutelle de l'INRIA, du CNRS et du laboratoire d'informatique fondamentale de Lille (LIFL) et de l'université des sciences et technologies de Lille (USTL). L'objectif du projet cible deux défis majeurs : les systèmes adaptatifs et les environnements multi-échelle. Le premier défis consiste à maîtriser d'adaptation d'intergiciels distribués. Le tissage d'aspects ou la re-configuration d'assemblages de composants font partie intégrante de ce premier défis. Le second défis cible des environnements multi-échelle allant des réseaux de capteurs à l'internet en passant par les réseaux pour grilles de calcul, etc.

L'équipe est membre de l'organisme de standardisation OMG (*Object Management Group*) et du consortium open source Objectweb[1], et du réseau d'excellence européen sur le développement par aspects (*AOSD NoE – Network of Excellence on Aspect-Oriented Software Development*).

Consortium ObjectWeb Nous avons participé au consortium ObjectWeb qui vise le développement d'intergiciels notamment à base de composants et en *open source*. Créé en 2002, notamment par les deux partenaires de cette thèse, France Télécom R&D et l'INRIA, nous avons fréquemment contribué lors de Workshops autour du modèle à composants Fractal ou lors de réunions plénières.

AOSD NoE Nous avons participé au réseau d'excellence européen sur le développement par aspects créé en 2004, qui réunit neuf partenaires dont l'INRIA. En particulier, nous avons collaboré au laboratoire travaillant sur l'élaboration d'une architecture de référence servant de preuve de concept et d'expérience majeure pour la validation du développement par aspects comme une valeur technologique sûre pour l'avenir.

1.3 Démarche de travail

Le problème qui nous intéresse particulièrement est la définition d'un modèle général rapprochant le développement par composants et le développement par aspects. Les deux approches s'unissent sur un certain nombre de points comme la recherche de la modularité, et la séparation des préoccupations. Cependant, elles diffèrent dans leur philosophie : les modèles à composants imposent une base forte pour la modularisation, l'encapsulation, la définition de contrats autour des propriétés fonctionnelles d'un système ; le développement par aspects cherche à se greffer sur un paradigme existant pour permettre la modularisation des préoccupations transverses aux systèmes, qui sont souvent les services techniques de ces systèmes. Les deux approches apparaissent

[1] http ://objectweb.org

donc comme à la fois proche sur la volonté de séparation des préoccupations et complémentaire (séparation des préoccupations fonctionnelles, séparation des préoccupations transverses qui sont souvent non-fonctionnelles).

Nous nous sommes donc tourné, dans un premier temps, vers une étude de l'existant. Nous avons identifié plusieurs manières possibles d'aborder le rapprochement des deux styles de développement. Dans certaines approches, un module aspect est ajouté à un canevas à composants. Les aspects sont alors utilisés au niveau de l'implantation des composants, donc en arrière plan et n'apparaissent pas comme des entités de premier ordre. Dans d'autres approches, les aspects sont ramenés au statut d'entité de premier ordre au même titre que les composants, parfois même aucune distinction n'est réalisée entre un composant et un aspect et la composition est enrichie des techniques de l'approche par aspects.

Pour pouvoir comparer toutes ces approches, nous nous basons sur un certain nombre de propriétés d'une part de l'approche à composants et d'autre part de l'approche par aspects qui nous semble essentielles. Nous considérons que le rapprochement des deux styles de développement doit se faire en respect de ces propriétés, c'est-à-dire en gardant toutes ces bonnes propriétés mais tout en comblant les faiblesses de chacune. Autrement dit, les composants doivent être dotés d'un support pour la modularisation des préoccupations transverses fourni par les techniques par aspects. Cependant, ceci doit être réalisé au niveau structurel des composants et non de leur implantation. D'autre part, les aspects doivent bénéficier de plus de modularité et d'encapsulation, clairement spécifier ses interactions avec le système, tel un composant avec des interfaces contractualisées et dans l'idée des architectures à base de composants.

Nous choisissons donc de proposer un modèle unifié pour composants et aspects. Comme base à cette unification nous choisissons le modèle de composant FRACTAL, qui est un modèle général, extensible et fortement réflexif, offrant ainsi toutes les bonnes propriétés que nous avons identifiés dans notre étude de l'existant. Dans notre extension de FRACTAL appelée FAC pour *Fractal Aspect Component*, nous introduisons un certain nombre de concepts à FRACTAL pour le support des préoccupations transverses. Du point de vue aspect, nous obtenons un certain nombre de propriétés qui enrichissent l'approche aspect. Parmi ces contributions notons :

- la structuration des aspects comme des composants,
- la réification du tissage sous forme d'éléments architecturaux, permettant de rendre explicite les interactions aspects dans un système,
- une gestion à granularité fine de la composition d'aspects, permettant de sortir de nombreux problèmes de composition comme le phénomène d'entrelacement dans le code des aspects eux-même,
- un support sûr des aspects par le modèle de composant, par support sûr nous entendons une résolution de la contradiction entre l'encapsulation forte des composants et la nature envahissante des aspects,
- une forte indépendance entre la fonction d'un aspect et son implantation,

La contribution majeure est la possibilité d'obtenir une séparation des préoccupations complète aussi bien dans les composants que dans les aspects. Nous démontrons cette propriété à travers plusieurs études de cas : la gestion des transactions étendues, la gestion de la communication de groupe. Ces deux domaines sont reconnus comme exemples caractéristiques de préoccupations transverses à un système et suffisamment complexes pour être sujet à des problèmes de composition lorsqu'ils sont développés par des aspects.

1.4 Organisation du document

Ce document se décompose en dix chapitres. Ces chapitres suivent la démarche détaillée à la section précédante.

Le Chapitre 2 Ce chapitre introduit toutes les bases nécessaires à la compréhension du paradigme de séparation des préoccupations et de ses deux mises en œuvre par d'une part le développement par aspects, et d'autre part le développement avec des modèles à composants. Nous faisons également le point sur la famille des langages d'architecture et la réflexivité, qui tiennent une place importante dans nos travaux.

Le Chapitre 3 Ce chapitre explore les approches qui tentent de rapprocher les composants et les aspects. Nous montrons des exemples reposant sur des serveurs d'applications, sur des conteneurs légers et enfin sur des approches plus académiques. A chaque fois nous utilisons des critères que nous avons identifié dans le chapitre précédant pour évaluer la qualité de ces approches.

Le Chapitre 4 Ce chapitre introduit le modèle à composants FRACTAL et ses outils. Il s'agit du modèle à composants que nous choisissons comme base pour notre unification des composants et des aspects. Nous détaillons en conséquence tous les concepts et les outils que nous réutilisons dans notre contribution.

Le Chapitre 5 Ce chapitre constitue une introduction à notre approche et à notre démarche. Il motive tout d'abord le choix de FRACTAL comme base pour notre contribution. Il en montre également les limites pour justifier la plus-value de nos travaux. Il justifie notre choix d'unifier composants et aspects.

Le Chapitre 6 Ce chapitre détaille notre contribution FAC, une extension de FRACTAL pour le support d'aspects. Tous les concepts sont exposés ainsi que la méthodologie à appliquer pour l'utilisation du modèle. FAC est également évalué au regard des critères identifiés dans le Chapitre 2.

Le Chapitre 7 Ce chapitre propose un certain nombre de concepts avancés de FAC. Il détaille notamment la méthodologie associée à l'utilisation de FAC, et montre comment les aspects peuvent être supportés de manière sûre par les composants FRACTAL.

Le Chapitre 8 Ce chapitre présente notre première étude de cas. Nous choisissons de regarder comment FAC permet de concevoir et implanter la gestion de transactions étendues dans un système. Nous montrons comment les approches par aspects classiques échouent dans la conception d'un tel aspect, et comment FAC permet d'obtenir une séparation des préoccupations complète.

Le Chapitre 9 Ce chapitre est une seconde étude de cas. Nous étudions l'ingénierie de la gestion de la communication de groupe en utilisant le développement par aspects. En premier lieu, nous montrons pourquoi ce domaine de la communication de groupe peut gagner à utiliser le développement par aspects. Ensuite nous mettons en évidence un problème de composition qui peut être adresser par FAC grâce aux propriétés structurantes de ses composants. Par cette seconde

étude nous montrons comment les composants peuvent aider les aspects, là où la première étude montre comment les aspects permet de venir à bout de l'entrelacement dans les composants.

Enfin le Chapitre 10 conclut ce document. Il reprend notre problématique et la solution proposée pour donner des perspectives à ces travaux.

Première partie

Etat de l'art

Chapitre 2

Principes fondamentaux des approches à composants et par aspects

Sommaire

Ce chapitre présente les grands principes des approches à composants et des approches par aspects. La Section 2.1 introduit le principe de séparation des préoccupations sur lequel reposent les deux types d'approche. Puis, la Section 2.2 trace les grandes lignes de l'approche par aspects. Enfin, la Section 2.3 présente les concepts fondateurs de l'approche à composants et les langages d'architecture qui leurs sont parfois associés.

2.1 Séparation des préoccupations

Le principe de la séparation des préoccupations est à la base de l'ingénierie logicielle et permet de diviser le logiciel en propriétés à chaque fois plus petites pour en maîtriser la complexité, de la conception jusqu'à la réalisation. Ces préoccupations doivent par nature être orthogonales pour renforcer la modularité, la réutilisation et la compréhensibilité du programme. La plupart des paradigmes de programmation repose sur ce principe de séparation des préoccupations, depuis

la programmation modulaire, jusqu'à la programmation par aspects, à composants, ou encore la définition de patron de conception et bien entendu la programmation par objets.

Parnas, avec la programmation modulaire, fût un des premiers à mettre en avant ce principe et, en particulier, à identifier deux manières distinctes de décomposer un système [Parnas, 1972]. Une première possibilité consiste à décomposer le système en phases de traitement, qui sont les étapes d'exécution du système. La seconde possibilité repose sur des choix de conception, et s'avère être plus propice à la maintenance, à l'appréhension des préoccupations et au développement séparé de chaque module.

Succédant à la programmation modulaire, la programmation par objets a proposé l'utilisation d'objets pour la représentation d'un système, en se reposant fortement sur les principes de polymorphisme, d'encapsulation (reposant sur les informations masquées) et d'héritage [Meyer, 1997]. Dans ce paradigme, les objets sont des instances particulières des classes qui sont des représentations abstraites définissant des attributs (données) et des méthodes (fonctions). L'héritage propose une forme avancée de modularisation et de réutilisation en introduisant la notion de sous classe ou classe héritée comme une version spécialisée de sa classe dite parente. Le polymorphisme s'exprime de trois manières. La première consiste à surcharger les méthodes d'une classe parente. La seconde, appelée aussi généricité, représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents. Enfin, la troisième et dernière est la possibilité de redéfinir les méthodes d'une classe parente (appelée la spécialisation). Les paradigmes de programmation à composants et par aspects qui ont succédé aux objets ont insisté sur un certain nombre de limites de la programmation par objets.

L'approche par aspects, que nous présentons dans la Section 2.2, a mis en lumière une autre problématique reliée à la séparation des préoccupations dites transverses, c'est-à-dire des préoccupations qui ne sont pas idéalement incarnées dans un module, objet ou composant défini selon l'axe d'une décomposition fonctionnelle, mais éparpillées dans plusieurs [Kiczales et al., 1997]. Ainsi, il est courant de voir apparaître dans le code différentes préoccupations, indépendantes du code métier de l'application, et se faisant concurrence. Les aspects permettent de modulariser ces préoccupations en les extrayant. Par la suite, un tisseur d'aspects permettra de faire le lien entre les aspects et ce qu'on appelle la base, c'est-à-dire le code où s'appliquent les aspects.

L'approche à composants, que nous présentons dans la Section 2.3, a mis l'accent un peu plus sur l'encapsulation et la contractualisation des modules ou objets, qui sont alors appelés composants logiciels [Szyperski, 2002]. Malgré de nombreux apports de la programmation par objets, certaines limitations ont vu le jour comme la lisibilité des applications construites et la clarté de la spécification des relations entre les objets. De plus, aucune représentation globale de l'architecture du programme n'est proposée. La famille des approches à composants à base de conteneurs comme EJB [Bodoff et al., 2004], ont pointé sur un certain nombre de services techniques redondants, comme la gestion des transactions de la persistance et ont proposé de fournir ces services dans des conteneurs de composants. D'autres approches, plus académiques, ont élaboré des modèles permettant une véritable composition d'entités indépendantes et une vision claire des dépendances entre ces unités : la programmation par assemblage de composants, et leur description par des langages d'architecture.

2.2 L'approche par aspects

En 1997, Gregor Kiczales et le Xerox Parc introduisent une nouvelle technique de programmation, la programmation par aspects (*AOP – Aspect-Oriented Programming*), permettant une isolation, une composition et une réutilisation efficaces du code dit transverse dans une application [Kiczales et al., 1997]. AspectJ [Team, 2006] en fût la première réalisation concrète appliquée au langage Java. De ce fait, AspectJ reste aujourd'hui largement cité dans tous les travaux autour

de l'approche par aspects et reste considéré comme la référence, notamment pour son langage de coupe.

La programmation par aspects est née, un peu comme l'approche à composants, des limitations de l'approche objet. En effet, dans la pratique de la programmation par objets, on constate que chaque fonctionnalité n'est pas idéalement incarnée par une classe ou un groupe restreint de classes. Ainsi, il est courant de voir apparaître dans le code différentes préoccupations, indépendantes du code métier de l'application, et se faisant concurrence. On parle alors d'entrelacement de code.

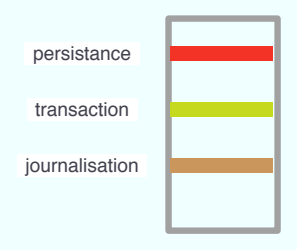

FIG. 2.1 – L'entrelacement par mélange.

Cet entrelacement prend deux formes distinctes :

– le **mélange** : au sein du code d'une classe ou d'un composant on retrouve plusieurs préoccupations non directement reliées, comme des aspects de journalisation, de persistance ou encore de sécurité. La Figure 2.1 propose une illustration où la boîte représente un élément de base (classe ou composant) et les lignes colorées, les lignes de code concernant une propriété transverse.

– la **dispersion** : certaines préoccupations ne sont pas modularisées au sein d'une même entité et sont présentes dans la majeure partie des classes ou composants. La Figure 2.3 propose une illustration de ce phénomène avec trois éléments de base et une propriété transverse à ces trois éléments, qui est éparpillée dans les trois éléments.

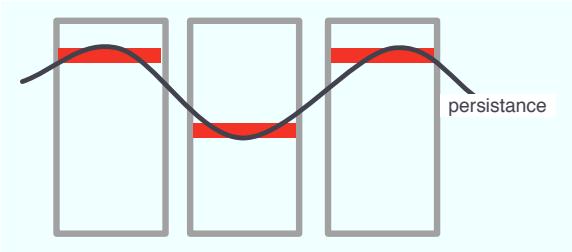

FIG. 2.2 – L'entrelacement par dispersion.

Ainsi, à travers ces différentes formes d'entrelacement, on constate que des portions identiques de code sont réécrites dans de nombreux composants ou classes que nous appelons éléments de base. Evidement, cette mauvaise séparation des préoccupations met en exergue une réutilisabilité faible des éléments de base. Pour remédier à ce problème d'entrelacement, l'approche par aspects propose de regrouper tout code concernant la même préoccupation applicative au même endroit, dans ce que l'on appelle un aspect, ce qui permet d'améliorer la modularité. L'approche par aspects s'articule alors autour d'un certain nombre de notions essentielles que nous détaillons ci-dessous.

Base La base correspond au programme sans aspects, qui supportera donc le tissage de préoccupations transverses par le biais d'aspects.

Aspect Un aspect est la modularisation d'une préoccupation transverse. Un aspect prend plus ou moins la forme d'une classe ou d'un composant, selon l'approche, et contient donc des champs et des méthodes. Dans certaines approches également, la coupe peut être définie dans l'aspect lui même qui contient des codes advice. Comme nous le verrons par la suite, dans la Section 2.2.2, il est possible d'avoir une vision symétrique ou asymétrique d'un aspect. Dans les approches symétriques, aucune distinction n'est faite entre un aspect et un élément de base (un objet ou un composant par exemple).

Tissage Le principe de tissage d'un aspect à un ensemble d'éléments de la base consiste à assembler ces entités de telle sorte que le produit final soit l'application de base étendue par le comportement de l'aspect. Les tisseurs peuvent être statiques ou dynamiques. Dans le premier cas l'opération est réalisée pendant la phase de compilation, dans le second cas pendant l'exécution. AspectJ [Team, 2006] permet également de tisser des aspects au chargement des classes par la machine virtuelle Java.

Code advice Un code advice implante le comportement d'un aspect. De manière similaire aux classes et méthodes, le comportement d'un aspect peut être décomposé en plusieurs codes advice. Plusieurs types de code advice existent : avant, après, autour d'un élément de base.

Point de jonction Un point de jonction est un point dans le flot d'exécution d'un programme. En général, les points de jonction sont des invocations de méthodes, des blocs d'exceptions, ou des accès aux attributs.

Coupe Une coupe permet de désigner un ensemble de points de jonction. Elle est utilisée pour spécifier où un code advice s'applique. Comme nous le verrons par la suite avec la symétrie de relation (voir Section 2.2.2) et en particulier le placement, il est possible ou non de séparer la définition d'une coupe du code advice.

2.2.1 L'approche par aspects : bonne pratique ou paradigme ?

On notera que l'approche par aspects repose sur une utilisation combinée de patrons de conception (*design paterns*) définis quelques années auparavant [Gamma et al., 1995] et plus particulièrement sur le patron adaptateur (*design pattern adaptor*). L'idée de base est de fournir des moyens pour ajouter à une application un aspect (un morceau de code) qui décrit un traitement particulier. De ce fait, la programmation par aspects est, par moment, plus qualifiée de combinaisons de bonnes pratiques, que de nouveau paradigme de programmation. Ceci vient du fait que l'approche par aspects nécessite une base sur laquelle s'appliquer : la base peut être un ensemble de classes ou de composants, mais reste nécessaire pour donner du sens à l'approche. L'approche par aspects est une application du principe d'inversion des dépendances et utilise fortement la reflexivité, principes que nous présentons dans les paragraphes suivants.

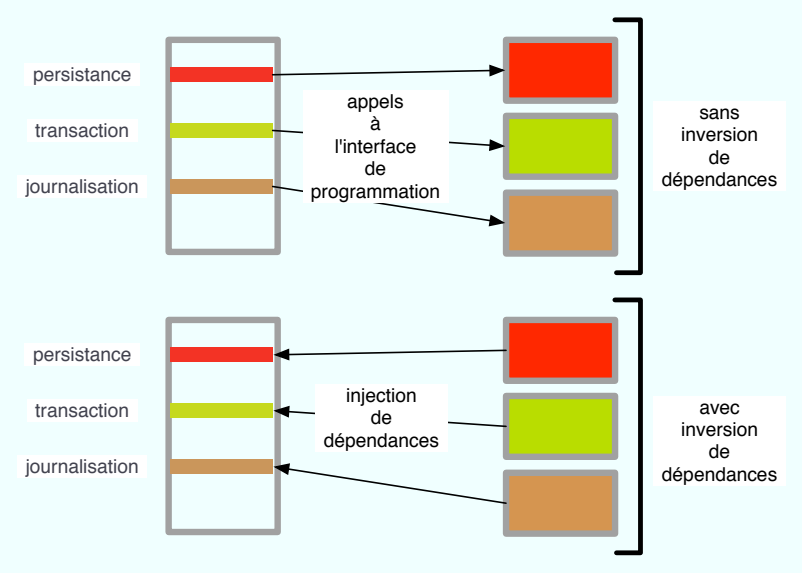

FIG. 2.3 – L'inversion de dépendances.

Inversion des dépendances

L'inversion de dépendances permet de changer la dépendance entre deux entités où la moins abstraite (ou de plus bas niveau) des deux dépend de la plus abstraite (ou de plus haut niveau). Prenons, par exemple, la gestion de la préoccupation de la journalisation. Sans l'inversion de dépendances, le programme doit systématiquement récupérer la référence vers l'interface de programmation et appeler cette interface. Ce qui revient à dire que la partie implantation d'un programme dépend d'une interface de plus haut niveau, la gestion de la journalisation. Considérons le second exemple suivant : une classe A dépend directement d'une classe B. Si l'on souhaite inverse cette dépendance, on peut alors faire intervenir une interface I, implantée par la classe B. La classe B dépend à présent de A et de I. On a gagné en abstraction et réduit le couplage. En réduisant ainsi le couplage entre les entités, on augmente leur réutilisation.

D'autres techniques d'inversion de dépendances proposent au second module d'injecter cette dépendance dans le code du premier. Le problème du sens de la dépendance est fortement lié à la notion de transversalité de l'approche par aspects. Ainsi, l'approche par aspect est souvent considérée comme une des mises en œuvre de ce principe d'inversion. Notons qu'il est également possible de réaliser de l'injection de référence, par constructeur, par méthodes *setters*, par transformateur de code. En bref l'inversion de dépendances, par son mécanisme de découplage, permet d'améliorer la modularité grâce à une meilleure séparation des préoccupations.

Méta-programmation et réflexivité

La réflexion est la capacité pour un programme à découvrir son comportement, sa structure, et à le modifier, l'adapter [Smith, 1982]. La méta-programmation caractérise cette capacité. De nombreux langages objet supportent la réflexion, comme SmallTalk, Ruby, ou à un moindre degré Java. Les connexions entre la méta-programmation et l'approche par aspects sont nombreuses. Par essence, l'approche par aspects cherche à séparer les préoccupations linéaires des préoccupations qui leur sont transverses. Par essence, la méta-programmation cherche à séparer les objets applicatifs (niveau base), des objets qui réifient l'application (niveau meta). Or, on remarque que

	Relation Symétrique	Relation Asymétrique
Symétrie d'élement	Placement : Tout, Portée : Totale e.g . : JAC	
Asymétrie d'élément	Placement : Tout, Portée : Totale e.g . : Filtres de composition	Placement : Aspect, Portée : Binaire e.g . : AspectJ

TAB. 2.1 – La correspondance entre symétrie de relation et d'élément

dans l'approche par aspects, les préoccupations transverses raisonnent fréquemment sur les préoccupations de base sur lesquelles elles viennent se tisser. Il en est de même pour le niveau méta en méta-programmation. La méta-programmation est donc un moyen pour la réalisation d'une approche par aspects.

2.2.2 Le rôle de la symétrie

Nous entrons à présent un peu plus en détail dans l'approche par aspects en introduisant la notion de symétrie pour caractériser les principaux canevas par aspects. Pour ceci, nous nous appuyons sur une étude de [Harrison et al., 2003] qui définit différentes formes de symétrie. Cette étude spécifie trois types d'entités pouvant être organisées symétriquement ou asymétriquement : les éléments sujets à composition (*composeable elements*), les points de jonctions et les relations de composition.

Les éléments sujets à composition sont symétriques lorsqu'ils sont identiques au niveau de la base et de l'aspect. Implicitement, l'asymétrie d'éléments exclut la composition aspect-sur-aspect, car il serait alors possible, partant d'un aspect nul, de construire un système, n'utilisant ainsi qu'un type d'élément. Un asymétrie d'éléments autorise donc les aspects à ne s'appliquer que sur la base, et la base n'est jamais consciente des aspects.

Les points de jonction symétriques correspondent aux traditionnels appels de méthodes et accès aux champs. Un point de jonction asymétrique se différencie en traitant plusieurs préoccupations comme les appels à *proceed* qui traitent à la fois de l'appel local provenant de la position de la méthode, et du module sur lequel l'aspect se compose. Un autre exemple d'asymétrie concerne les blocs de traitement d'exception en Java. Le sujet de l'exception ne peut être extrait de la préoccupation du module et, de ce fait, le point de jonction correspondant au *try* traite plusieurs préoccupations.

La symétrie de relation est subdivisée en symétrie de portée et de placement. Le placement est relié à la symétrie d'éléments. Le placement indique si la composition est dans l'aspect (`Placement : Aspect`) ou peut être n'importe où (`Placement : Tout`). Le second cas dénote de l'indépendance de la coupe (composition) du comportement (code advice). Dans AspectJ, par exemple, le code advice et les déclarations d'advice (les coupes) figurent tous les deux dans l'aspect. A l'opposé dans l'approche *Composition Filters* [Aksit et al., 1992], des séquences de filtres sont associées à des objets. Bien que deux types d'entités existent, dénotant une asymétrie d'élément, la composition des filtres est adressée symétriquement. La composition des filtres avec les objets est indépendante de la déclaration des filtres. La table 2.1 résume les combinaisons possibles.

Concernant la portée, lorsqu'elle est définie comme binaire (`Portée : Binaire`), la relation connecte simplement le code advice au code de base sans être capable d'être consciente de

tout autre aspect. Alternativement, une portée totale (`Portée : Totale`) permet à la relation de raisonner sur tous les aspects s'appliquant sur le même point de jonction. A titre d'exemple dans AspectJ la portée est binaire alors que dans les filtres de composition elle est totale. Ceci s'explique parce que AspectJ ne propose comme seul mécanisme de composition que le séquencement d'advice, c'est-à-dire la possibilité de déclarer un ordre global de précédence sur les aspects définis dans un système. AspectJ ne permet donc pas de contrôler l'ensemble des aspects s'appliquant sur un même point de jonction. Il est même impossible de détecter ce genre de situation, sinon par le biais d'outils spécifiques de visualisation. A contrario, les filtres de composition sont entièrement contrôlables dynamiquement comme des objets.

Une asymétrie de relations combinée avec une symétrie d'éléments est considérée comme une combinaison impossible. Ceci signifierait que des éléments de la base contiendraient une composition asymétrique vis-à-vis de la base elle-même, impliquant ainsi une rupture de la symétrie d'élément.

Pour compléter les exemples donnés dans le Tableau 2.1, nous détaillons les approches suivantes : AspectJ, Les filtres de composition (CFs), JAC et leur position par rapport à la symétrie d'élément et de relation. De nombreuses autres langages par aspects peuvent venir compléter ce tableau, nous avons choisi dans cet état de l'art de n'en présenter qu'un par catégorie possible.

AspectJ

AspectJ, comme nous le disions en introduction de cette section, est la première implantation des principes de l'approche par aspects en Java et, est aujourd'hui l'implantation la plus connue et utilisée [Team, 2006]. AspectJ se caractérise par son asymétrie d'éléments et de relations. Les aspects sont des entités différentes des objets de base, AspectJ est avant tout un langage permettant de caractériser des coupes et des codes advice. Du fait du mélange de la définition de la coupe et des codes advice au sein d'un aspect, l'approche est asymétrique de relation. Il n'est pas non plus possible de raisonner sur l'ensemble des aspects s'appliquant sur le même point de jonction (portée binaire).

Les filtres de compositions (*Composition Filters*)

Les filtres de compositions de l'Université de Twente ne sont pas une approche par aspects en tant que telle mais sont considérés comme source d'inspiration du paradigme [Aksit et al., 1992]. Notamment, la similitude entre les filtres placés autour des objets, et les code advice de type autour qui viennent se tisser autour des méthodes des objets. Par conséquent nous incluons les filtres de composition dans cette comparaison. Leur particularité est de définir à la fois une asymétrie d'éléments et une symétrie de relations. Il existe en effet deux types d'éléments : les objets et les filtres (asymétrie d'élément) ; mais la composition peut être positionnée indifféremment dans tout le programme (placement symétrique) et la portée est également totale du fait qu'il est possible de manipuler tous les filtres s'appliquant sur le même objet (portée totale).

JAC

JAC est l'approche symétrique que nous retenons dans notre comparaison [Pawlak et al., 2004]. JAC est un canevas pour aspects en pur Java, où aspects et objets de base sont des objets (symétrie d'élément). La composition des aspects autour d'un point de jonction (portée totale) est configurable, ainsi que la définition des coupes qui est extérieure aux aspects (placement total).

HyperJ

HyperJ, initié par IBM, [Tarr et al., 1999] est l'approche symétrique par excellence. L'approche met en avant la séparation des préoccupations multi-dimensionnelle en Java. Le principe est de décomposer un programme en modules et de pouvoir les composer librement pour former le programme final. Contrairement à la majorité des approches en ingénierie logicielle, un module n'est pas nécessairement encapsulé, indépendant et fonctionnel. Il représente simplement une fonctionnalité et souvent doit être composé avec d'autres modules pour former un ensemble cohérent.

Bilan

Nous avons étudié dans cette section les principes fondamentaux de l'approche par aspects. Nous avons introduit les concepts de base, les paradigmes et mécanismes proches comme la reflexivité et l'inversion de dépendances. Nous avons ensuite suivi une étude sur la notion de symétrie pour classifier les principales approches par aspects. Nous retenons ces critères de symétrie comme critère d'analyse du chapitre suivant, qui pousse l'étude des approches par aspects un peu plus loin, en regardant les approches par aspects appliquées aux approches à composants. Nous résumons ici ces critères et en définissons les avantages.

La symétrie d'élément permet de ne faire aucune distinction entre la base et les aspects. Ceci a deux conséquences majeures en termes de séparation des préoccupations. Aucune séparation artificielle n'est réalisée entre une préoccupation de nature transverse ou simplement indépendante. Ceci facilite l'évolution du système sur une seule dimension (la base), et non sur deux dimensions comme avec AspectJ (la base et l'aspect). Seconde conséquence, la symétrie d'éléments permet à des aspects de s'appliquer à nouveau sur des aspects. Ceci est très important lorsque l'on cherche à obtenir une séparation des préoccupations complète. En effet, un aspect peut lui même mélanger plusieurs sous-préoccupations qu'il est alors impossible d'extraire dans le cas d'une asymétrie d'éléments.

La symétrie de relation est décomposée en deux sous catégories : le placement et la portée. Le placement symétrique (placement total) permet de séparer strictement la partie comportementale d'un aspect et la coupe, la partie liaison à la base. La portée autorise la manipulation complète des aspects s'appliquant sur un même point de jonction et augmente donc le pouvoir de composition et la maîtrise des aspects, ce qui est important à la vue du côté envahissant des aspects. La symétrie de portée est donc utile pour composer plusieurs aspects de nature différente en collision sur le même point de jonction. Globalement, la symétrie de relations permet donc une meilleure évolution et une meilleure visibilité de l'action des aspects.

	AspectJ	**Filtres de Composition**	**JAC / HyperJ**
Symétrie d'élément	non	non	oui
Symétrie de relation Placement	non Aspect	oui Tout	oui Tout
Symétrie de relation Portée	non Binaire	oui Totale	oui Totale
Séparation des préoccupations transverses	oui	oui	oui

TAB. 2.2 – Comparaison des approches par aspects

Le Tableau 2.2 résume nos différents critères et les approches que nous avons considérées dans cette section. A noter que nous n'avons pas parlé d'HyperJ [Tarr et al., 1999] qui pour nous

figure dans la même case que JAC sur ces critères.

2.3 L'approche à composants

Les composants logiciels sont nés d'un besoin de réutilisation et doivent être conçus comme un investissement pour de nombreux systèmes ou applications. L'expression de «composant pris sur l'étagère» renforce cette volonté de réutilisation. Bien que les composants ne soient pas les seules entités ciblant la réutilisation, leur particularité réside en trois points : ils sont dotés d'interfaces contractuelles, ils sont conditionnés de telle sorte qu'ils puissent être déployés indépendamment, enfin ils sont sujet à composition par un tiers [Szyperski, 2002].

«A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.» [Szyperski, 2002]

D'une manière générale, on peut dire qu'un composant est un élément ou une entité qui est vu comme une boîte, blanche ou noire (selon que le contenu est rendu visible ou non depuis l'extérieur). Il est doté de points d'entrée et de sortie (des interfaces) qui lui permettent d'échanger des messages ou des références vers d'autres composants. Ces points d'accès représentent les services offerts et requis par le composant.

Un composant doit être réutilisable et peut être déployé de manière indépendante, dans différentes applications ou contextes d'application. Il est à noter qu'il existe de nombreux modèles à composants industriels et académiques et que chacun introduit ses propres définitions, mais que pratiquement tous s'accordent sur les quelques principes que nous venons d'énoncer. Nous regardons à présent les principales approches à composants que nous avons sélectionnées, à savoir la famille des langages d'architecture (Section 2.3.1), des approches à conteneurs (Section 2.3.2), avec en particulier JBoss (Section 2.3.3) et Spring (Section 2.3.4), et les modèles OpenCOM (Section 2.3.5) et FRACTAL (Section 2.3.6).

2.3.1 Les langages d'architecture (*Architecture Description Language – ADL*)

Les langages d'architecture [Medvidovic and Taylor, 2000] permettent de spécifier les entités d'un système (ses composants) ainsi que leurs relations ou interactions, et ainsi d'avoir une gestion fine de l'évolution des entités définies, de contrôler leur déploiement, de définir des contraintes sur le système. Cela permet de gérer et définir, à un haut niveau et de manière hiérarchique, les propriétés d'un système.

Dans sa classification des langages d'architecture, Medvidovic [Medvidovic and Taylor, 2000] définit un ADL comme un langage permettant d'avoir une vision abstraite des composants. La description met en œuvre :

- des connecteurs qui décrivent les connexions entre composants,
- des ports qui sont des points d'entrée et de sortie des composants,
- la possibilité d'obtenir un modèle hiérarchique de composition et de restructurer cette composition de manière dynamique.

Composant Un composant est l'élément de base. Il offre généralement des ports requis ou fournis qui sont les points d'entrée et de sortie de l'entité, lui permettant de communiquer avec d'autres composants.

Composite Un composant qui rend visible sa structure interne et contenant d'autres composants est appelé composite. A ce niveau, les composites sont rarement dotés de comportement propre et se retrouvent donc fréquement à réaliser ce que l'on appelle de la délégation de ports. La délégation consiste à rediriger les interfaces ou ports des composants contenus dans le composite vers l'extérieur. On parle également dans ce cas d'importation ou d'exportation de port.

Connecteur Un connecteur symbolise les interactions entre composants et définit des règles qui permettent la gestion du comportement de ces interactions. Les connecteurs peuvent prendre toutes sortes de formes selon les langages d'architecture, comme par exemple la forme d'un composant «composant-connecteur» offrant une vision plus claire d'une architecture dans certaines situations.

Les points d'accès : port, interface, opération Les concepts de port, interface et opération permettent de désigner les points d'accès d'un composant, les points ouvert à la composition par des connecteurs ou des liaisons. La notion de port n'est pas toujours présente dans les langages d'architecture. Il s'agit souvent d'interfaces fournies ou requises. Lorsque à la fois les notions de port et d'interface sont présents dans un langage d'architecture, le port apporte un niveau d'abstraction supplémentaire, en précisant les opérations requises et fournies, là où, par exemple, une interface fournie ne propose que des opérations fournies. Les opérations sont souvent des méthodes ou des fonctions que le composant rend accessible et regroupe par ses interfaces et/ou ports.

Liaisons entre composants La plupart des ADL offrent des mécanismes de liaison entre les composants qui exposent leurs services à travers leurs ports ou interfaces.

De nombreux ADL existent dont les plus connus sont : Wright [Allen, 1997], Darwin [Magee et al., 1995], Rapide [Rapide, 1997], et ArchJava [Aldrich et al., 2002]. Nous nous intéressons à présents aux modèles à conteneurs.

2.3.2 Les approches à conteneurs

Dans les approches à conteneurs, un conteneur est une entité spécialisée dans le suivi des composants, qui gère à la fois leur instanciation, leurs interactions, ainsi que l'environnement de l'application, en particulier les services techniques requis par les composants, comme la gestion des transactions ou de la persistance. Deux dimensions sont considérées : la dimension dite fonctionnelle ou métier implantée par les composants, et la dimension technique implantée par les conteneurs.

Les approches à conteneurs reposent sur une architecture dite trois-tiers ou à trois niveaux, qui divise une application en couches : la présentation, le métier, les données. Les serveurs d'application reposant sur cette architecture les plus connus du moment sont les serveurs d'EJB (*Entreprise Java Beans*) [Bodoff et al., 2004].

2.3.3 Le conteneur d'EJB JBoss

Les EJB sont la solution proposée par Sun de la version entreprise des spécifications de Java (*J2EE – Java 2 Platform, Enterprise Edition*) [Bodoff et al., 2004]. J2EE est un ensemble d'extensions de Java pour la conception d'applications réparties (voir Figure 2.4). Les spécifications EJB sont

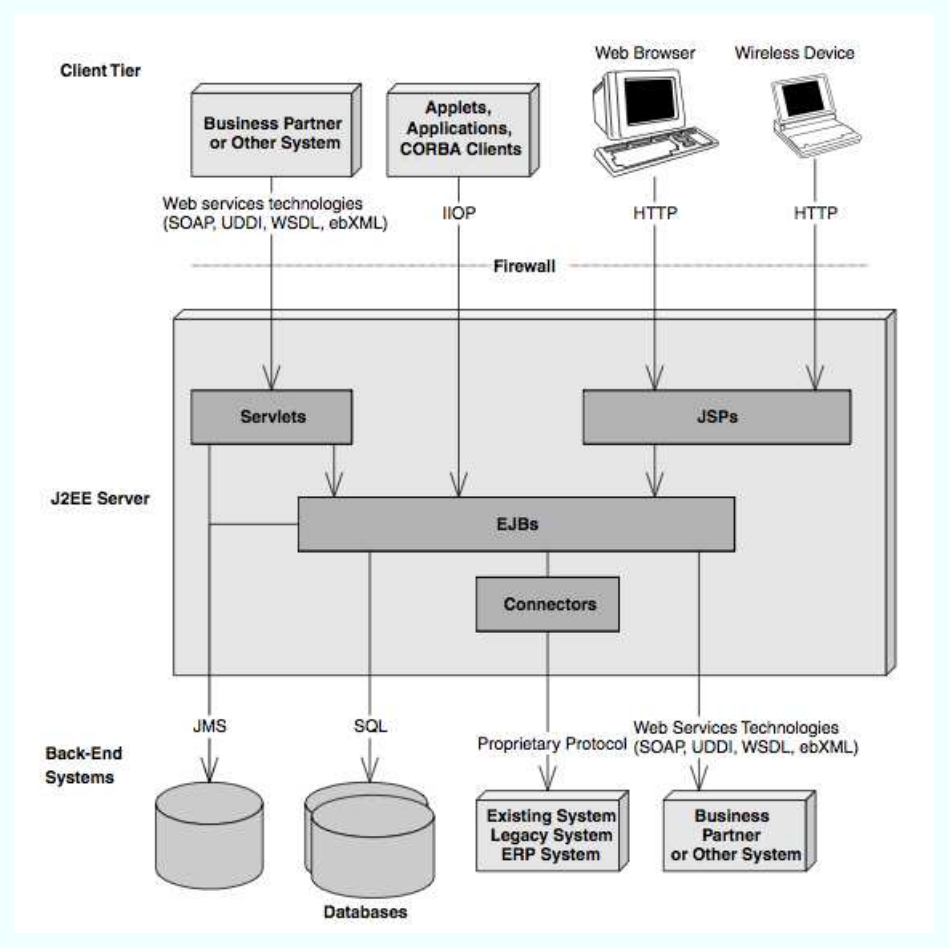

FIG. 2.4 – La plate-forme J2EE.

une sous partie des spécifications J2EE pour la définition d'une architecture à composants logiciels côté serveur. Elles reposent sur un serveur d'application qui héberge les différents types de composants EJB. Malgré le fait que EJB soit une spécification, il existe de nombreuses différences d'un serveur d'application à l'autre qui empêche fréquemment leur interopérabilité. Ceci constitue un défaut majeur de l'approche, car, nous l'avons vu, en définissant la notion de composant logiciel, la propriété première d'un composant est sa capacité à être réutilisé, ce qui reste très difficile à faire d'un serveur d'application à l'autre.

Les composants EJB se distinguent en trois types : entité, session et orienté messages. Les EJB entités représentent les données, les EJB sessions représentent les traitements, les EJB orientés messages sont des EJB sessions fonctionnant par envois de messages. Les conteneurs EJB hébergent les composants EJB. Ils ont à charge la gestion des composants, depuis leur création et leur destruction (cycle de vie), jusqu'à la gestion des ressources transactionnelles et de la persistance de données (par les EJB entités), ou encore la gestion de la concurrence ou de la répartition de charge.

Les composants EJB restent largement utilisés en industrie, en particulier par le support technique fourni autour des différents serveurs d'applications. Il n'en reste pas moins que de nombreuses limitations viennent entraver les fonctions premières des composants. Nous avons déjà évoqué le problème de la non-interopérabilité des serveurs d'applications entre eux. S'ajoute à

cette limitation une intrinsèque mauvaise séparation des préoccupations techniques. Cette affirmation peut paraître surprenante car nous avons vu que les approches à conteneurs, par définition, font une séparation claire du fonctionnel et technique. Cependant, cette séparation théorique n'existe pas dans la réalité de l'implantation. Pour pouvoir connecter des services techniques à du code fonctionnel, le code fonctionnel doit appeler l'interface de programmation du service technique requis. Ceci constitue une mauvaise séparation des préoccupations car l'appel à l'interface de programmation technique se retrouve éparpillée dans tout le code fonctionnel. Ce phénomène d'éparpillement est à la base de l'émergence de l'approche par aspects.

2.3.4 Les conteneurs «légers» de Spring

Constatant de nombreux défauts de lourdeur d'utilisation et de mise en place — en particulier la disproportion du temps passé à mettre en œuvre les services techniques à appliquer au fonctionnel —dans les approches à conteneurs «classiques» comme les conteneurs EJB, des approches dites à conteneurs légers sont apparues. Un des plus en vogue actuellement est le canevas Spring [R. Johnson, 2006].

Le canevas Spring (*Spring Framework*) est un canevas open source pour la plate-forme Java. Il s'impose comme une alternative aux serveurs d'application EJB en implantant entre autre un système pour l'inversion de dépendances (*IoC – Inversion of Control* en anglais) qui permet de faire de l'injection de code. L'inversion de dépendances qui est notamment un des principes de base de l'approche par aspects (voir Section 2.2.1 p.15), permet d'atténuer l'éparpillement des appels aux services techniques dans le code fonctionnel que nous avions identifié dans les composants EJB. Pour réaliser son inversion de contrôle, Spring est doté d'un noyau aspect qui montre qu'aspect et composant peuvent être associés. Ici les aspects sont utilisés pour faire la glu entre les services techniques et le code fonctionnel.

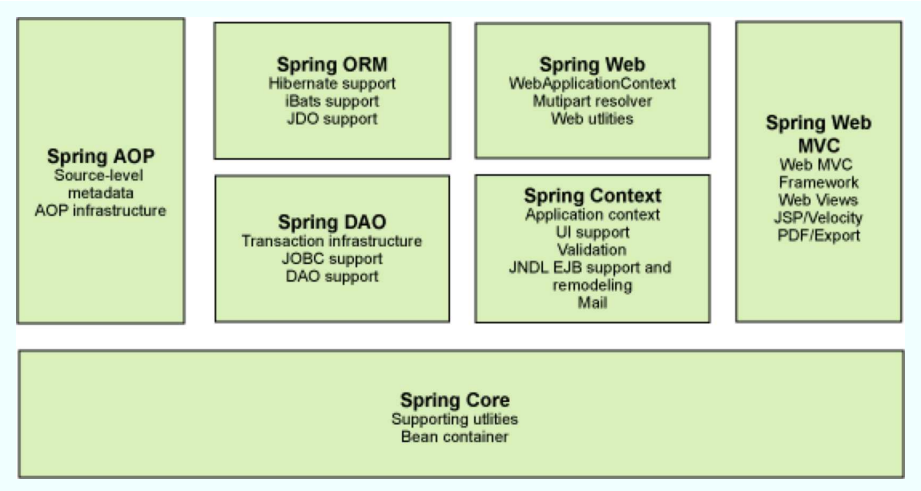

FIG. 2.5 – Architecture du canevas Spring Framework.

Cependant, malgré ces bonnes propriétés, Spring reste une approche à conteneur, et comme nous allons le voir à travers des modèles plus académiques, des manques au niveau de la représentation du code métier se font sentir. Entre autres, il nous semble primordial de rendre explicite les déclarations de dépendances entre composants, en mettant en avant la notion d'architecture logicielle et, en particulier, de donner la possibilité de définir des hiérarchies de composants.

Nous introduisons à présent les modèles OpenCOM et FRACTAL.

2.3.5 OpenCOM

OpenCOM est un modèle à composants de l'université de Lancaster [Coulson et al., 2004a]. Il s'agit d'un modèle réflexif et général pour le développement d'intergiciels. Le modèle est dit général car indépendant de tout langage de programmation. A ce jour, il existe une implantation en C++ et en Java. Le modèle est réflexif entre autres grâce à une interface spécifique `IUnknown` qui permet d'introspecter un composant.

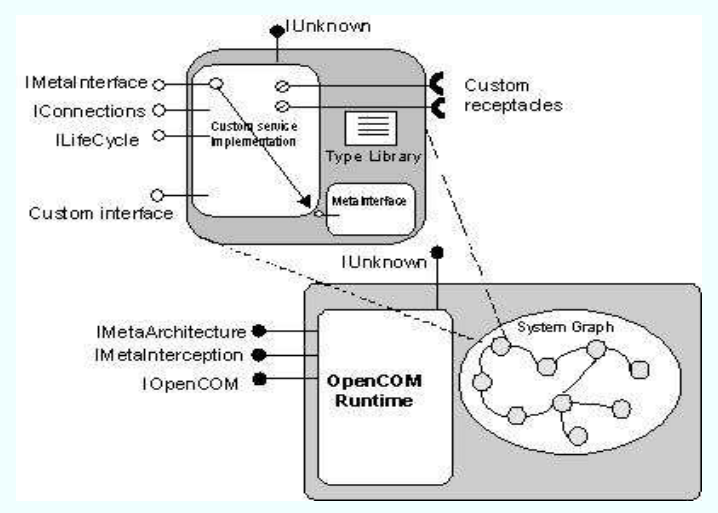

FIG. 2.6 – Le modèle OpenCOM.

OpenCOM dans sa première version reposait sur la technologie COM (*Component Object Model*) de Microsoft[2]. Par conséquent, le langage de description d'interfaces de Microsoft est utilisé pour spécifier les services d'un composant. La Figure 2.6 présente les concepts fondamentaux d'OpenCOM dans la première version, expliqués ci-dessous.

Tout comme de nombreux langages d'architecture, OpenCOM repose sur les concepts d'interfaces, de réceptacles et des connexions. Les interfaces correspondent aux interfaces fournies dans le vocabulaire habituel des langages d'architecture. Les réceptacles correspondent aux interfaces requises. Enfin, les connexions sont les liaisons entre les interfaces et les réceptacles de même type.

OpenCOM repose sur la notion d'espace d'adressage qui définit des espaces de nom dans lequel les composants peuvent se référencer. Ensuite, la gestion de la création et de la suppression des composants, ou l'établissement de liaisons, se fera à l'exécution pour chaque espace d'adressage. OpenCOM est un modèle réflexif, le système maintient un graphe de dépendance représentant les dépendances entre les composants, que ces derniers peuvent introspecter. Les interfaces pour la réflexion suivent le méta-modèle proposé par OpenORB, à savoir une interface `IMetaInterface` pour les interfaces, une interface `IMetaArchitecture` pour l'architecture, et une interface `IMetaInterception` pour le comportement.

Un composant OpenCOM doit implanter les interfaces `ILifeCycle` qui fournit les opérations pour le démarrage et l'arrêt du composant, `IConnections` pour la gestion des réceptacles, et `IMetaInterface` pour pouvoir inspecter le type des interfaces. En plus de ces trois interfaces obligatoires, l'interface `IMetaInterception` et `IMetaArchitecture` peuvent être implantée pour introspecter le comportement des opérations fournies et l'architecture environnante.

[2] http://www.microsoft.com/com/

Au niveau des limitations, le modèle n'est pas extensible, aucun mécanisme n'est offert pour facilement rajouter de nouveaux concepts ou interfaces pour le contrôle des composants. De plus, l'architecture n'est pas purement hiérarchique, la notion de composite est absente. Les composants cohabitent tous au même niveau dans un espace d'adressage donné. OpenCOM reste principalement un prototype utilisé dans des projets de recherche à l'université de Lancaster.

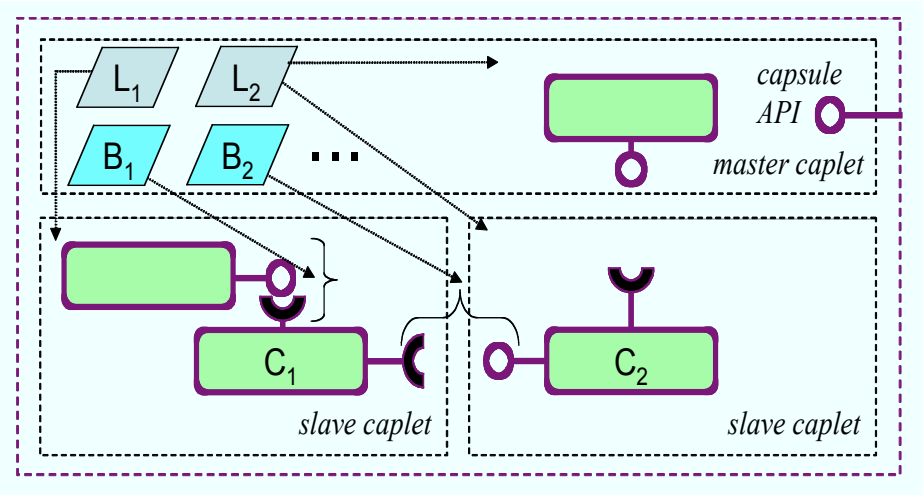

FIG. 2.7 – Le modèle OpenCOM version 2.

Dans la seconde version du modèle [Coulson et al., 2004b] (voir Figure 2.7), la dépendance avec COM a disparu, et le modèle est hiérarchique grâce à de nouvelles notions notamment de *capsule* et de *caplet*. Les *capsules* se situent à mi-chemin entre la notion de conteneur et de composite : elles définissent une portée et sont des unités de gestion des *caplets*. Les *caplets* sont des sous-unités des *capsules* que l'on peut rapprocher de la notion de composite. Les *caplets* peuvent être de type *root* ou *slave*. Il existe un seul *caplet* de type *root* par *capsule* pour le support dynamique du noyau de l'interface de programmation d'OpenCOM, tous les autres *caplets* sont de type *slave* et doivent être connecté aux noyau pour pouvoir créer de nouveau *caplet* ou gérer des connexions. Les *caplets* sont implantés par des composants tout comme les *loaders* et les *binders*. Les *loaders* permettent de charger des composants dans les *caplets*. Les *binders* offrent différent moyens de connecter des composants dans des *caplets*.

En résumé, si la seconde version du modèle a su améliorer la dynamicité et les capacités de réflexion de la première version, l'extensibilité des mécanismes de contrôle du modèle restent figés et constituent la limite principale du modèle.

2.3.6 FRACTAL

Le modèle de composant FRACTAL du consortium Objectweb [3] [Bruneton et al., 2006] est un modèle qui met en avant les concepts énoncés autour des langages d'architecture, à savoir la notion de composant, composite, interface et liaison. FRACTAL est d'ailleurs doté d'un langage d'architecture FRACTAL-ADL en XML.

La particularité de FRACTAL est de proposer des interfaces dites de contrôle reposant fortement sur le principe de réflexion (voir Section 2.2.1). Les interfaces de contrôle ont à charge de définir les fonctions de contrôle des composants, c'est-à-dire de gérer les propriétés des composants : contrôle du cycle de vie, des liaisons, du nom, etc.

[3] http://objectweb.org

Un point caractéristique du modèle est son indépendance de tout langage de programmation. De ce fait, les spécifications FRACTAL ont été implantées dans de nombreux langages comme Java, C, C++, SmallTalk, ou encore porté sur la plate-forme .NET. Nous caractérisons donc FRACTAL comme un modèle général.

FRACTAL est un modèle hiérarchique contrairement à OpenCOM qui ne fournit qu'un niveau de composition. Un composant FRACTAL peut même être partagé par plusieurs composites, pour par exemple mettre en œuvre du partage de service ou de ressources.

Enfin, FRACTAL est un modèle ouvert et extensible qui permet de facilement étendre l'ensemble des interfaces de contrôle définies. En offrant ainsi un ensemble ouvert d'interfaces de contrôle, il est possible de définir la personnalité du modèle qui convient le mieux aux besoins. Le modèle FRACTAL est complètement détaillé, ainsi que ses outils majeurs dans le Chapitre 4.

2.3.7 Bilan

Nous venons d'étudier les principes fondamentaux de l'approche à composants. Nous avons passé en revue quelques modèles dits à conteneurs et des modèles plus académiques. A travers ces descriptions, des critères sont apparus naturellement. Nous nous proposons de les définir explicitement dans cette section à titre de bilan de l'approche et avant de procéder au même exercice avec l'approche par aspects.

Voici les critères qui retiennent notre attention :

	EJB	Spring	OpenCOM	Fractal
Séparation des préoccupations fonctionnelles	oui	oui	oui	oui
Séparation des préoccupations techniques	non	oui	oui	oui
Général	non	non	oui	oui
Réflexif	non	non	oui	oui
Hiérarchique	non	non	oui	oui
Extensible	non	non	non	oui
Séparation des préoccupations transverses	non	non	non	non

TAB. 2.3 – Comparaison des approches à composants

Modèle général

Par modèle général nous entendons un modèle indépendant de tout langage de programmation, donc indépendant de l'implémentation et du serveur d'application utilisés. Nous avons vu que ceci constitue une forte limite dans le cas des composants EJB, dont les implantations réalisées d'un serveur d'application à l'autre sont incapables d'être réutilisées. Par contre les modèles FRACTAL et OpenCOM gardent l'indépendance vis-à-vis du langage d'implantation.

Modèle réflexif

Nous avons vu que les modèles FRACTAL et OpenCOM faisaient un fort usage de la réflexion pour pouvoir introspecter une architecture de composants, connaître les types des interfaces d'un

composant, etc. La réflexivité permet donc à un système d'adapter sa structure pendant sa propre exécution.

Modèle hiérarchique

Un modèle hiérarchique utilise la notion de composite, c'est-à-dire la possibilité de composer sur plusieurs niveaux des composants. L'avantage majeur est de pouvoir raisonner sur une architecture à plusieurs niveaux d'abstraction.

Modèle extensible

Par modèle extensible nous nous attendons à ce que le modèle offre un certain nombre de mécanismes pour faciliter l'extension des notions du modèle ou l'ajout de nouveaux mécanismes. Ce critère est essentiel pour nous car nous souhaitons mettre en relation les approches par composants et par aspects. En conséquence le modèle de composant doit être extensible pour y ajouter des mécanismes aspects.

2.4 Synthèse

Nous avons étudié dans ce chapitre les caractéristiques principales des approches à composants et des approches par aspects. Dans cette étude nous avons mis en évidence un certain nombre de critères qui nous permettent de classifier les modèles dominants dans chaque catégorie. Ces critères nous semblent être essentiels pour chacune des familles d'approche. Le chapitre suivant étudie les approches combinant composants et aspects. Nous évaluons ces modèles en suivant les critères extraits dans ce chapitre.

Chapitre 3

Les approches à composants et par aspects

Sommaire

Ce chapitre compare les principales approches combinant aspects et composants. Après un bref rappel des critères qui ont attiré notre attention dans le chapitre précédent (Section 3.1), nous abordons les approches suivantes.

- JBoss AOP (Section 3.2), un module du serveur d'application JBoss qui permet de tisser des aspects sur des objets quelconques ou avec les EJB.
- Spring AOP (Section 3.3) est un module aspect en pur Java du canevas Spring. Spring AOP propose également un support du langage AspectJ.
- CAM/DAOP (Section 3.4) est un modèle pour composants et aspects qui repose sur la plate-forme dynamique DAOP qui fonctionne par envois de messages.
- FuseJ (Section 3.5) est un modèle pour composants et aspects qui propose une unification des composants et des aspects.

3.1 Rappel des critères

Dans le chapitre précédent, nous avons introduit un certain nombre de notions et de propriétés des approches à composants et par aspects. Nous les regroupons ici en expliquant pourquoi elles sont importantes pour notre problématique de rapprochement de ces deux types d'approche sur le thème de la séparation des préoccupations. Nous utilisons ensuite l'ensemble de ces critères pour évaluer les différentes propositions de ce chapitre.

Langage d'architecture La présence d'un langage d'architecture nous semble être un paramètre très important. Il permet la description structurelle d'un système : ses blocs élémentaires et les relations entre ces blocs. Cette description se fait dans un langage indépendant de ceux employés pour la mise en œuvre du système. Le comportement du système peut ainsi être vu comme un tout et toutes les propriétés fonctionnelles et non-fonctionnelles peuvent être décrites à haut niveau.

Modèle général Par modèle général nous entendons un modèle qui reste indépendant de tout langage de programmation, dont les concepts élémentaires comme la notion de composant, de liaison, d'interface, sont définis indépendamment d'un langage de programmation. Cette propriété est proche de l'idée portée par les langages d'architecture, même si un modèle peut être général sans forcément être doté d'un langage pour la description d'architectures de composants.

Modèle réflexif Par modèle à composants réflexif, nous entendons un modèle qui permet d'introspecter et intercesser une archicture, donc de naviguer dans cette architecture. Autrement dit, cela donne accès à chaque composant à un moyen de connaître son contenu, ses interfaces, ses opérations (introspection), ou à connaître son environnement, comme par exemple le composite qui le contient dans le cas d'un modèle hiérarchique (intercession).

Modèle extensible Un modèle extensible est un modèle fournissant naturellement un support pour l'ajout de nouveaux concepts de nouvelles capacités non-fonctionnelles. Si l'on considère les approches à conteneurs, par exemple, l'ensemble des services techniques fournis est souvent fermé et difficilement extensible.

Modèle hiérarchique Le modèle est dit hiérarchique lorsqu'il est possible d'ajouter un composant dans un composant qui est alors un composite et ainsi de suite pour construire une hiérarchie de composants. Les modèles hiérarchiques permettent de facilement gérer différents niveaux de granularité et un meilleur passage à l'échelle.

Symétrie d'éléments La symétrie d'éléments consiste à ne pas faire de distinction entre un composant et un aspect (aussi appelée unification). La symétrie d'éléments permet entre autres de pouvoir réappliquer des aspects sur des aspects, d'améliorer la modularité d'un aspect alors géré comme un composant, ou encore de ne considérer qu'une seule dimension pour l'évolution du système, et par conséquent, supprime la séparation base/aspect.

Symétrie de placement Une symétrie de placement permet de séparer clairement la définition d'un aspect de sa coupe. Cette symétrie permet d'améliorer la modularité des aspects car la coupe définit où l'aspect doit être tissé, là où sa définition spécifie son comportement, qui peut alors être réutilisé lorsque ces deux parties sont séparées.

Symétrie de portée La symétrie de portée permet d'extérioriser la gestion de la composition des aspects sur un même point de jonction. Tous les aspects sont complètement visibles et manipulables sur chaque point de jonction. Une telle symétrie autorise un meilleur contrôle des aspects agissant dans un système en augmentant le pouvoir de composition de ces aspects.

Nous regardons à présent différentes approches qui combinent l'utilisation de composants et d'aspects.

3.2 JBoss AOP

JBoss AOP est un canevas pour aspects qui peut aussi bien s'utiliser individuellement ou conjointement avec le serveur d'application JBoss [Burke, 2003]. Dans cette comparaison nous nous intéresserons à JBoss AOP intégrée à JBoss car nous nous intéressons aux approches mêlant composants et aspects.

3.2.1 Présentation

Les aspects dans JBoss AOP sont appliqués aux objets d'implantation de l'application. De ce fait, le paradigme aspect est ici utilisé en totale ignorance du modèle à composants sous-jacent. Ces aspects sont cependant écrits en pur Java, ce qui rend l'approche symétrique au niveau des éléments, d'un point de vue objet, mais asymétrique d'un point de vue composant (un aspect ne peut être un composant EJB). Ensuite les tissages sont réalisés grâce à des descripteurs XML ou par le biais d'annotations dans le code. Ainsi, la définition du comportement d'un aspect et sa liaison à la base sont détachées, caractéristique d'une symétrie de placement.

JBoss AOP, en plus d'offrir un mécanisme d'interception classique des approches par aspects, offre également un mécanisme d'introduction, ainsi que quelques mécanismes de mixins que nous ne détaillons pas plus car non considérés dans notre étude. Les points de jonctions pris en compte par JBoss AOP sont les accès aux attributs, les invocations de méthodes, ou encore un constructeur d'objet. Lors d'une interception de méthode, il est possible d'intervenir du côté de l'appelant ou de l'appelé. L'interception ne s'arrête pas aux frontières d'une classe, il est possible d'atteindre les éléments publics tout comme privés ou protégés.

Les coupes sont décrites indépendamment d'un aspect sous forme d'expressions régulières portées au niveau des descripteurs de l'application ou sous forme d'annotations. En ce sens, JBoss AOP suit la philosophie employée avec les EJB, qui consiste à configurer les services techniques dans des descripteurs XML ou à annoter directement le code des composants. JBoss AOP vise principalement l'intégration des services techniques à l'aide d'aspects, comme l'exemple classique de la journalisation, ou encore les services transactionnels. Pour relier une coupe indépendante d'un code advice, JBoss AOP utilise un mécanisme de liaison qui encore une fois fonctionne soit par annotations soit dans les descripteurs XML. Les définitions de coupes et les liaisons sont ensuite résolues à l'exécution.

Les codes advice sont écrits en pur Java. Une interface de programmation permet d'accéder au contexte d'interception par réflexion. Il est donc possible de connaître l'appelant, l'appelé, la méthode interceptée, etc. Le Listing 3.1 propose un exemple de définition d'aspect en utilisant les annotations. Comme on peut l'observer, il s'agit d'une classe normale qui n'a aucune interface spécifique à implanter et qui est annotée `@Aspect`. Ensuite, une annotation pour la liaison `@Bind` permet de définir une coupe. Dans cet exemple on voit que la coupe est liée au code advice (ici la méthode `pojoAdvice`). L'autre possibilité de liaison d'un aspect est d'utiliser les descripteurs XML comme illustré dans le Listing 3.2.

```
1 @Aspect
2 public class MyAspect {
3    @PointcutDef("execution(* POJO->*(..))")
4    public static Pointcut pojoMethods;
5    @PointcutDef("execution(POJO->new(..))")
6    public static Pointcut pojoConstructors;
7      @Bind(pointcut = "MyAspect.pojoMethods OR MyAspect.pojoConstructors")
8         public Object pojoAdvice(Invocation invocation) throws Throwable {
9         ...
10     }
11}
```

Listing 3.1 – JBoss AOP : exemple d'aspect utilisant les annotations.

```
1 <!-- This expression matches any public constructor of the POJO class. -->
2 <pointcut name="allPublicConstructors" expr="execution(public POJO->new(..))"/>
3 <!-- This expression matches any public constructor of the POJO class. -->
4 <bind pointcut="MyAspect.pojoMethods OR MyAspect.pojoConstructors">
5    <interceptor class="SimpleInterceptor"/>
6 </bind>
```

Listing 3.2 – JBoss AOP : exemple d'aspect utilisant les descripteurs XML.

L'instanciation des aspects peut se réaliser par machine virtuelle, par instance, par classe, ou enfin par point de jonction. JBoss AOP lorsqu'il est employé conjointement avec JBoss repose sur une architecture en quatre couches :

- le micro-noyau : un modèle à composants légers offrant le minimum en termes de déploiement à chaud et de chargement de classes utilisant JMX (Java Management Extensions),
- la couche des services comme les transactions, la gestion des messages, la sécurité,
- la couche aspect du modèle aspect de JBoss AOP,
- la couche applicative qui correspond à l'application finale.

Sur cette architecture, l'application a le choix d'utiliser les services du conteneur ou la couche aspect. Une série d'aspects pré-empaquetés est fournie avec JBoss et peut être utilisée directement via les annotations ou les descripteurs XML, parmi lesquels figurent certains célèbres patrons de conception du GoF (*Gang of Four*) [Gamma et al., 1995]. De plus, des outils existent pour le support de la couche aspect de JBoss, comme notamment des outils de visualisation qui permettent de suivre les classes qui seront instrumentées par les aspects.

3.2.2 Evaluation

En résumé, JBoss AOP est avant tout une approche par aspects qui s'applique au niveau objet, et donc au niveau de l'implantation des composants EJB. L'approche reste donc éloignée de ce que nous attendons ici en termes de symétrie, de modèle extensible, hiérarchique et réflexif.

Les éléments intéressants de l'approche sont la séparation claire entre la définition d'un aspect et sa coupe. Ceci renforce la réutilisation des aspects. Cette propriété est mise en avant dans l'approche qui fournit un ensemble d'aspects pré-définis.

ADL	Général	Réflexif	Extensible	Hiérarchique	Symétrie Elément	Symétrie Placement	Symétrie Portée
non	non	non	non	non	non	oui	non

TAB. 3.1 – Bilan de JBoss AOP

Langage d'architecture Pas de langage d'architecture.

Modèle général Le modèle est dédié au langage Java, et JBoss est un serveur d'application des spécifications EJB pour Java.

Modèle réflexif Le modèle ne propose pas d'architecture logicielle et donc pas de mécanisme de navigation dans cette architecture par réflexion.

Modèle extensible Le modèle n'est pas extensible car il suit les spécifications EJB.

Modèle hiérarchique Le modèle n'est par hiérarchique.

Symétrie d'élément Asymétrie d'élément, les aspects sont en pur Java mais ne sont pas des composants. Les aspects sont donc appliqués au niveau de l'implantation des composants et non comme concept de premier ordre.

Symétrie de placement Le modèle sépare clairement la définition d'un aspect de sa liaison aux composants (placement symétrique) qui peut être définie par annotations dans le code ou à l'aide des descripteurs XML de l'application.

Symétrie de portée Il n'est pas possible de voir l'ensemble des aspects s'appliquant sur un même point de jonction comme un tout et de les manipuler. Par contre, des outils de visualisation sont fournis pour pouvoir détecter éventuellement des conflits d'aspects.

Nous poursuivons notre étude avec le canevas Spring et son module aspect, ce qui nous permet de voir ce qui est produit au niveau des conteneurs légers et des aspects.

3.3 Spring AOP

Dans le chapitre précédent, nous avons déjà présenté le canevas open source Spring pour applications Java/J2EE. Dans cette section, nous discutons du module Spring AOP [R. Johnson, 2006]. Pour rappel, Spring est un modèle à conteneur léger qui permet une intégration des services J2EE. Les modèles à conteneurs légers marquent un retour vers le paradigme objet, là où les serveurs d'applications ont prôné l'utilisation de composants EJB à gros grain, difficiles à configurer et à maintenir, et parfois inadaptés aux besoins.

Nous présentons les concepts fondamentaux de Spring AOP avant d'en faire l'évaluation. A noter que Spring dans sa seconde version offre un support pour le langage AspectJ [Team, 2006].

3.3.1 Présentation

La partie aspect de Spring est en pur Java et implante donc des interfaces du canevas Spring. Ces interfaces existent pour les coupes et les codes advice, ou encore les *advisor* qui encapsulent les deux à la fois.

De manière similiaire à JBoss AOP, les aspects Spring peuvent s'appliquer sur les composants *beans* qui sont de toute manière des objets. Les aspects peuvent à la fois atteindre les objets hébergés par un conteneur à inversion de contrôle ou des objets extérieurs.

Contrairement à JBoss AOP, Spring AOP supporte uniquement l'interception et non l'introduction. De plus, l'interception se limite aux méthodes, car l'accès aux attributs de classe est considéré comme une violation de l'encapsulation par les auteurs de Spring.

```
1@Aspect
2public class AnAspect {
3  @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
4  public void businessService() {}
```

```
5}
```

Listing 3.3 – Spring AOP : exemple d'aspect utilisant les annotations et AspectJ.

Les coupes sont définies en implantant des interfaces fournies par le canevas, par annotations (voir Listing 3.3), ou alors par le biais de descripteurs XML à l'aide d'expressions régulières (voir illustration du Listing 3.4).

```
1<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
2<aop:config>
3  <aop:advisor
4      pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
5      advice-ref="tx-advice"/>
6</aop:config>
7</bean>
```

Listing 3.4 – Spring AOP : exemple d'aspect utilisant les descripteurs XML.

Le Listing 3.5 propose un exemple de code advice de type «autour» utilisant les annotations et AspectJ. Sur le Listing 3.6 nous voyons le même aspect écrit en Spring AOP sans AspectJ. On voit que l'approche est similaire à JBossAOP ou JAC, avec en paramètre de la méthode le contexte d'interception qui permet de réifier l'appel à la méthode de base par un appel à *proceed()*.

```
1 @Aspect
2 public class AroundExample {
3    @Around("com.xyz.myapp.SystemArchitecture.businessService()")
4    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
5       // partie avant
6       Object retVal = pjp.proceed(); // réification de l'appel de méthode
7       // partie après
8       return retVal;
9  }
10}
```

Listing 3.5 – Spring AOP : exemple d'aspect utilisant les annotations et AspectJ.

```
1public class AroundExample implements MethodInterceptor {
2    public Object invoke(MethodInvocation invocation) throws Throwable {
3        // partie avant
4        Object rval = invocation.proceed();
5        // partie après
6        return rval;
7    }
8}
```

Listing 3.6 – Spring AOP : exemple d'aspect sans les annotations.

La composition d'aspect se fait par le classique mécanisme de précédence d'advice d'AspectJ, c'est-à-dire qu'il est possible de spécifier globalement sur l'application que tel ou tel aspect sera appliqué avant tel autre. Nous considérons que les approches proposant uniquement ce type de composition, lorsque plusieurs aspects s'appliquent sur le même point de jonction, sont asymétriques de portée. Une approche symétrique à ce niveau propose une manipulation totale de ces éléments et donc bien entendu un moyen direct de connaître les collisions possibles. Le Listing 3.7 propose un exemple de séquencement qui suit tout simplement l'ordre de déclaration des codes advice.

```
1 @Aspect
2 public class AspectWithMultipleAdviceDeclarations {
3
4  @Pointcut("execution(* foo(..))")
5  public void fooExecution() {}
6
7  @Before("fooExecution()")
8  public void doBeforeOne() { ... }
9
10  @Before("fooExecution()")
```

```
11   public void doBeforeTwo() { ... }
12
13   @AfterReturning("fooExecution()")
14   public void doAfterOne() { ... }
15 }
```

Listing 3.7 – Spring AOP : mécanisme de séquencement d'advice.

3.3.2 Evaluation

Spring propose un canevas léger pour le support des services J2EE, en ajoutant des propriétés intéressantes, comme l'abstraction réalisée au niveau des fabriques. Il reste indépendant des spécifications et laisse le choix à l'utilisateur de l'implantation ou de l'outil qu'il désire. En ce sens nous considérons Spring comme un canevas extensible. Cependant cela reste un canevas J2EE donc avec les défauts que nous avions soulignés pour l'approche JBoss AOP : pas de langage d'architecture, pas de notion de hiérarchie de composants et pas de réflexion permettant d'introspecter une hiérarchie de composants.

ADL	Général	Réflexif	Extensible	Hiérarchique	Symétrie Elément	Symétrie Placement	Symétrie Portée
non	non	non	oui	non	oui	oui	non

TAB. 3.2 – Bilan de Spring AOP

Du point de vue aspect nous considérons que l'approche est symétrique simplement par le fait que Spring marque un retour à l'utilisation des POJOs, même pour les composants beans. Les aspects étant eux mêmes écrits en pur Java, l'approche est dite symétrique. La symétrie de placement est également respectée, car les descriptions de coupes sont séparées des codes advice. Par contre, la symétrie de portée n'existe pas, la composition d'aspects sur un même point de jonction reste du séquencement simple d'aspects, sans possibilité de manipuler ou «voir» les autres aspects en collision sur ce même point. Le Tableau 3.2 résume les caractéristiques de Spring AOP vis-à-vis de ces critères.

3.4 CAM/DAOP

CAM/DAOP est un modèle à composants et par aspects qui combine les bénéfices des deux approches [M. Pinto, 2005, M. Pinto, 2003]. CAM (Component Aspect Model) est le modèle et DAOP (Dynamic Aspect Oriented Platform) en est la plate-forme d'exécution.

3.4.1 Présentation

CAM est un nouveau modèle à composants et par aspects et s'inspire des standards EJB et CCM. Deux entités de premier ordre existent dans CAM : les composants et les aspects. Cela la place dans la catégorie des approches asymétriques (asymétrie d'éléments).

Les composants et les aspects sont des unités de composition au sens de la définition de Szyperski [Szyperski, 2002], c'est-à-dire des entités auto-contenues à gros grain, qui peuvent être déployées indépendamment. Tout comme CCM [OMG, 2002], CAM utilise un langage de description d'interface qui permet de décrire les services fournis et requis d'une interface. Ces descriptions d'interfaces font partie du langage d'architecture DAOP-ADL utilisé pour décrire les

composants et les aspects. À noter que les aspects sont définis dans ce langage avec les points de jonctions qu'ils peuvent intercepter et évaluer. Par la suite, des règles de composition spécifiques permettent de tisser ces aspects. CAM n'est pas un modèle hiérarchique et la notion de composite n'existe pas, les composants ne peuvent contenir d'autres composants.

Les communications dans CAM se font uniquement par message, DAOP étant une plate-forme par envoi de messages. La gestion de ces messages est résolue à l'exécution et coordonnée par un aspect qui encapsule les interactions d'un composant pour retrouver la cible d'un message émis. Un système de nom de rôle est également utilisé pour les composants et les aspects pour pouvoir se référencer plus facilement en fonction de la propriété fournie. De cette manière, un nom de rôle donné peut être implanté par divers composants. Il est alors possible d'avoir différentes stratégie pour trouver un composant implantant un rôle.

Les points de jonction supportés par CAM sont les interfaces publiques d'un composant. Le contenu d'un composant ne peut être affecté par des aspects pour préserver la propriété d'encapsulation des composants. Les points de jonction considérés sont donc les messages émis et reçus entre les composants ainsi que la création ou destruction d'un composant. Les coupes sont définies au niveau du langage d'architecture DAOP-ADL, donc extérieurement aux aspects respectant ainsi une symétrie de placement (symétrie de relation). Les aspects ont accès à un contexte d'interception permettant de connaître le composant émetteur ou récepteur, ainsi que le message.

CAM/DAOP est un modèle général au sens où il a été conçu pour être indépendant de tout langage de programmation. Pour le moment, CAM/DAOP est implanté en Java et Java RMI est utilisé pour la communication entre composants.

```
1<component role="chat">
2  <providedInterface>ChatProv.xml</providedInterface>
3  <requiredInterface>
4    <fromTargetComponent role=chat/>
5    <requiresMessage>ChatReq.xml</requiresMessages>
6  </requiredInterface>
7  <requiredInterface>
8    <fromTargetComponent role=awarenessList/>
9    <requiresMessage>AwarReqlnt.xml</requiresMessages>
10 <requiredInterface>
11 <implementations>
12   <implementation>
13     <name>chat1</name>
14     <language>java</language>
15     <class>Chat.class</class>
16   </implementation>
17 </implementations>
18</component>
```

Listing 3.8 – CAM/DAOP : exemple d'un composant en DAOP-ADL.

```
1<interface name="chatProvInt">
2  <message ID="1" name="sendText">
3    <argument type="String"/>
4  </message>
5  ...
6</interface>
```

Listing 3.9 – CAM/DAOP : exemple de définition d'une interface en DAOP-ADL.

Le Listing 3.8 illustre la définition d'un composant avec DAOP-ADL. Les interfaces fournies et requises sont décrites et font à chaque fois référence à des messages. Par exemple, l'interface fournie de la Ligne 2 fait référence à un autre fichier XML donné dans le Listing 3.9. Enfin les informations sur l'implantation du composant sont données aux Lignes 11–17.

```
1<aspect role="persistence">
2  <evaluatedInterface>
3    <joinpoint>BEFORE_SEND</joinpoint>
4    <capturedMessages>PersistenceEval.xml</capturedMessages >
5  </evaluatedInterface >
```

```
 6  <implementations>
 7    <implementation>
 8      <name>persistence1</name>
 9      <language>java</language>
10      <class>LDAPPersistence.class</class>
11    </implementation>
12     <implementation>
13      <name>persistence2</name>
14      <language>java</language>
15      <class>OraclePersistence.class</class>
16    </implementation>
17  </implementations>
18</aspect>
```

Listing 3.10 – CAM/DAOP : exemple de définition d'un aspect en DAOP-ADL.

Le Listing 3.10 montre comment un aspect est défini dans le langage d'architecture DAOP-ADL. Tout comme un composant, un rôle est attribué ainsi qu'une interface dite évaluée qui définit le type de point de jonction à la Ligne 3.

```
1<compositionRules>
2  <componentCompositionRules>
3    <compositionRuleFor role="chat">
4       <compositionRule>
5         <formatRole>awarenessList</formatRole>
6         <realRole>userList</realRole>
7       <compositionRule>
8     </compositionRuleFor>
9   <componentCompositionRules>
10
11  <aspectEvaluationRules>
12    <createComponent role="chat">
13      <BEFORE_NEW>
14        <concurrent>
15          <aspect role="authentification"/>
16        </concurrent>
17      </BEFORE_NEW>
18    </createComponent>
19    ...
20   <aspectEvaluationRules>
21 <compositionRules>
```

Listing 3.11 – CAM/DAOP : exemple de définition d'une composition en DAOP-ADL.

Enfin le Listing 3.11 propose une déclaration de composition entre le composant et l'aspect. Il s'agit en réalité de la déclaration de coupe qui est séparée de la déclaration des codes advice. Les Lignes 2–9 définissent une mise en relation des noms de rôle avec ceux définis «en dur» dans le code et ceux utilisés au niveau du langage d'architecture. Ensuite les Lignes 11–20 définissent la coupe en tant que telle (appelé règle d'évaluation dans CAM/DAOP). Le terme évaluation est employé car ces évaluations ont lieu pendant l'exécution du programme, la plate-forme DAOP étant complètement dynamique.

3.4.2 Evaluation

CAM/DAOP se distingue par son modèle général et indépendant de tout langage de programmation et de son langage d'architecture. CAM sépare avantageusement la définition des coupes des aspects, ce qui permet d'accroître la réutilisation de ces derniers. Cependant, l'approche propose une asymétrie d'élément et le modèle n'est pas hiérarchique ni facilement extensible. Nous passons en revue nos critères d'évaluation sur le modèle. Le Tableau 3.3 résume les caractéristiques de CAM/DAOP vis-à-vis de ces critères.

Langage d'architecture DAOP-ADL est le langage d'architecture de CAM/DAOP. Il permet de définir les interfaces fournies et requises ainsi que les composants et les aspects et leur composition.

ADL	Général	Réflexif	Extensible	Hiérarchique	Symétrie Elément	Symétrie Placement	Symétrie Portée
oui	oui	non	non	non	non	oui	non

TAB. 3.3 – Bilan de CAM/DAOP

Modèle général Le modèle est bien général car indépendant de tout langage de programmation. Pour le moment CAM/DAOP a été implanté dans le langage Java.

Modèle réflexif Le modèle n'est pas réflexif dans le sens où seuls les aspects ont accès dans leur contexte d'interception aux informations du modèle à composants lui-même. Les composants lors de l'envoi de message peuvent seulement connaître l'émetteur et le récepteur mais ne peuvent naviguer dans l'architecture.

Modèle extensible Aucun mécanisme particulier n'est mis en œuvre pour permettre l'ajout de nouveaux concepts.

Modèle hiérarchique Le modèle n'est pas hiérarchique, les aspects et composants cohabitent au même niveau.

Symétrie d'élément Deux entités de premier ordre existent : les composants et les aspects. Le modèle est asymétrique.

Symétrie de placement Le modèle sépare clairement la définition d'un aspect de sa liaison aux composants (placement symétrique).

Symétrie de portée Aucun mécanisme ne permet une manipulation totale des aspects s'appliquant sur le même point de jonction (asymétrie de portée).

3.5 FuseJ

FuseJ [Suvée et al., 2006] est un modèle pour composants et aspects qui propose une unification des deux paradigmes. L'unification signifie qu'aucune distinction n'est faite entre la partie comportementale d'un composant et d'un aspect. De ce fait l'approche est dite symétrique (du point de vue des éléments) selon les critères que nous avons définis.

FuseJ introduit un nouveau modèle à composants pour ses besoins ainsi qu'une terminologie associée pour la définition des interfaces fournies et requises qui sont des portes (*gates*). Nous regardons à présent en détail le modèle de FuseJ et la composition offerte par la partie aspect avant d'évaluer l'approche selon les critères de comparaison que nous avons définis.

3.5.1 Présentation

L'objectif du projet de recherche FuseJ est donc d'étudier les caractéristiques de la symétrie d'éléments (appelé aussi unification). L'approche fait le choix de ne faire aucune distinction entre un aspect et un composant. Seule l'interaction entre deux modules dont l'un joue le rôle d'aspect requiert un nouveau mécanisme : une composition orientée aspect.

Les composants et les aspects sont des beans Java dans FuseJ. La composition par aspects de FuseJ permet de les tisser sur d'autres beans.

Le modèle à composant est très simple et reste très proche du langage Java. Ainsi, un compo-

sant est une classe Java et le principe d'interfaces fournies et requises est mis en œuvre par des classes Java. Les services d'un composant sont ensuite spécifiés dans un nouveau langage, que l'on peut rapprocher des langages d'interfaces (*IDL – Interface Description Language*). Une spécification de services fournit donc une liste des interfaces offertes et implantées par le composant et des interfaces requises ou attendues par ce même composant.

```
interface TransferI {
byte[] getFileFragment(String aFileName)
  FileFragementInfo findFileFragment(String aFileName);
}

interface NetworkI {
   void send(String host, String info);
   byte[] get();
}

service TransferS {
   provides TransferI;
   expects NetworkI;
}
```

FIG. 3.1 – FuseJ : Spécification du service *TransferS*

La Figure 3.1 fournit un exemple de spécification de service. Deux interfaces sont spécifiées, qui sont des interfaces Java classiques. Ensuite un service est vu comme un ensemble d'interfaces fournies ou requises. Le service du Listing 3.1 déclare l'interface `TransferI` comme une interface fournie et l'interface `NetworkI` comme une interface requise. Le Listing 3.12 montre comment le service *TransferS* est représenté par l'interface Java `TransferS` implantée par la classe `TransferC` représentant le composant *TransferC*. Dans ce même listing, les appels aux méthodes `send` et `get` sont des appels aux opérations requises de l'interface `NetworkI` de la spécification de la Figure 3.1. Cela signifie que le code d'un composant utilisant des services requis ne peut compiler sans être complété a posteriori par FuseJ qui injectera les bonnes dépendances. Ceci constitue une première embûche à l'utilisation de FuseJ. Comme nous le voyons par la suite en étudiant la composition dans le modèle, ceci n'est pas le seul défaut du modèle à composants.

```
1 public class TransferC implements TransferS {
2    public byte[] getFileFragment(String aFileName) {
3       FileFragementInfo info = findFileFragment(aFileName);
4       send(info.host(), "get|" + aFileName + "|" + info.filefragement());
5       return get();
6    }
7    public FileFragementInfo findFileFragment(String aFileName) {
8       /* Code for sequential retrieval of file fragments */
9    }
10 }
```

Listing 3.12 – FuseJ

Un des avantages néammoins d'un tel système est le faible couplage qu'il existe entre les composants. Chaque service possède une spécification comme celle de la Figure 3.3 et correspond à un composant FuseJ. Ces services peuvent ensuite être utilisés comme simple service de composant ou comme code advice d'un aspect selon les besoins. Ceci est la conséquence directe du choix d'unification.

FuseJ ne supporte pas comme JBoss ou Spring des services techniques pré-empaquetés et réutilisables. Ces services qui sont de nature transverses et reconnus comme tels, doivent être implantés comme des composants FuseJ pour être réutilisés.

En ce qui concerne la composition, FuseJ propose un langage de configuration qui utilise des constructions explicites proche des langages d'architecture. Les Figures 3.2 et 3.3 montrent comment les connexions entre les services des composants sont établies. La configuration de la Figure 3.3 s'assure que chaque exécution d'une opération du composant *TransferNetC* est enregistrée. Pour cela, les bonnes méthodes sont sélectionnées grâce à la clause *execute* qui choisit

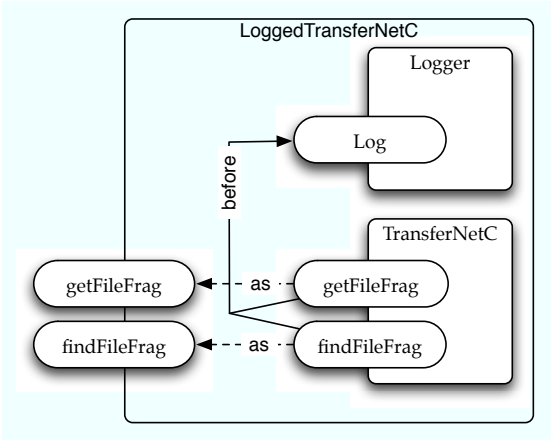

FIG. 3.2 – Une interaction orientée aspect entre les composants `TransferNetC` et `LoggerC`.

```
1 configuration LoggedTransferNetC configures
2 TransferNetC, LoggerC as TransferNetS {
3
4    linklet log {
5       execute:
6          Logger.log(String st);
7       before:
8          TransferNetC.*(..);
9       where:
10         st = Source.getMethodSignature();
11    }
12
13 }
```

FIG. 3.3 – Une interaction orientée aspect entre les composants `TransferNetC` et `LoggerC`.

l'opération *Log* du composant *Logger*. La partie *execute* correspond bien à l'aspect. Le mot-clé *before* permet de définir le type de jonction ainsi que la cible de l'aspect, c'est-à-dire le composant *TransferNetC*. Cette partie constitue donc la coupe. Enfin, La clause *where* initialise le paramètre *st* avec la signature de méthode qui sera interceptée par l'aspect. Ceci permet de faire une connexion d'information de contexte entre le composant et l'aspect, qui a besoin parfois de raisonner sur un contexte donné.

A travers l'exemple de la Figure 3.2 nous observons également comment la hiérarchie de composants est organisée en FuseJ. A chaque interaction entre deux composants un composite résultant est créé. Ce composite peut exporter certaines opérations des deux composants qu'il contient pour une nouvelle interaction. En bref, on a ici une hiérarchie qui est une fausse hiérarchie, puisque chaque composite contient à chaque fois strictement deux composants. Lors d'un passage à l'échelle, on se retrouvera dans une situation où l'architecture de l'application sera noyée par toutes ces interactions et ces délégations d'interfaces. Ceci est une des lacunes du modèle à composant de FuseJ.

3.5.2 Evaluation

Le principal atout du projet de recherche FuseJ est son choix marqué d'unifier composants et aspects. De ce fait l'approche est symétrique du point de vue des éléments et des relations, aussi bien la portée que le placement. Aucune distinction n'est observée entre un composant et un aspect. Tout service d'un composant peut être potentiellement employé comme un code advice.

Toute la composition dans FuseJ intervient au niveau de son langage de configuration et des *linklet*. Ces *linklet* peuvent à la fois spécifier des compositions classiques de type client/serveur entre composants ou alors des compositions par aspects.

ADL	Général	Réflexif	Extensible	Hiérarchique	Symétrie Elément	Symétrie Placement	Symétrie Portée
oui[a]	non	non	non	oui[b]	oui	oui	non

[a] Le langage de composition et de spécification de FuseJ.
[b] La hiérarchie est possible, mais chaque composite ne compose que deux composants à la fois.

TAB. 3.4 – Bilan de FuseJ

Ainsi, du point de vue aspect, l'évaluation de FuseJ est plutôt satisfaisante (voir Tableau 3.4). Cependant, en ce qui concerne le modèle à composants, nous relevons plusieurs manques importants. Par exemple, le modèle FuseJ reste limité au langage Java (modèle non général), il n'est pas extensible, et pas vraiment hiérarchique du fait de la complexité de la composition de composants entre eux. En effet, chaque composition dans FuseJ produit un composite, qui à nouveau, peut être composé avec un autre composant en déléguant toutes ses opérations. Nous sommes loin, dans ce cas de figure, des modèles hiérarchiques classiques. Enfin, notons que FuseJ est encore à l'état de prototype et que l'instrumentation du code est encore réalisée manuellement.

Langage d'architecture Pas de langage d'architecture, mais un langage pour la description de la composition propre à FuseJ.

Modèle général Le modèle est lié à Java.

Modèle réflexif Aucun mécanisme mis en œuvre pour l'introspection.

Modèle extensible Aucun mécanisme d'extension.

Modèle hiérarchique Comme nous l'avons expliqué précédemment la hiérarchie est présente mais très limitée.

Symétrie d'élément Du fait de l'unification réalisée par FuseJ, l'approche est symétrique du point de vue des éléments.

Symétrie de placement La définition de la composition, la partie coupe est clairement extériorisée de la définition d'un aspect.

Symétrie de portée Aucun mécanisme ne permet de connaître ou manipuler l'ensemble des aspects s'appliquant sur un point de jonction donné.

3.6 Synthèse

Nous venons d'étudier un certain nombre d'approches mêlant simultanément composants et aspects. Le Tableau 3.5 propose une synthèse des approches abordées dans ce chapitre. Nous tirons six conclusions de cette étude

Langage d'architecture et modèle général Parmi les quatre approches étudiées, seul CAM/-DAOP offre un langage d'architecture et un modèle général indépendant de tout langage de programmation. Dans les autres approches, le modèle est fortement lié à Java. Avec FuseJ, le modèle n'est pas général car lié à Java, mais il existe un langage de composition proche, dans l'idée, des langages d'architecture.

	JBoss AOP	Spring AOP	CAM/DAOP	FuseJ
Critères du point de vue composant et architecture				
ADL	non	non	oui	oui*
Général	non	non	oui	non
Réflexif	non	non	non	non
Extensible	non	oui	non	non
Hiérarchique	non	non	non	oui**
Critères du point de vue aspect				
Symétrie d'élément	non	oui	non	oui
Symétrie de placement	oui	oui	oui	oui
Symétrie de portée	non	non	non	non
** Le langage de composition et de spécification de FuseJ*				
*** La hiérarchie est possible, mais chaque composite ne compose que deux composants à la fois.*				

TAB. 3.5 – Bilan des approches à composants et aspects

Aucun modèle extensible ou réflexif Aucun des des modèles étudiés n'est réflexif ou extensible. Autrement dit, aucun modèle n'offre un mécanisme permettant de facilement enrichir le modèle de nouveaux concepts ou de nouvelles capacités de contrôle des composants. Aucun des modèles présentés n'est réflexif, donnant la possibilité aux composants de naviguer dans l'architecture auxquels ils participent (intercession) ou de découvrir ses interfaces (introspection).

Aucun modèle réellement hiérarchique Aucun modèle de notre étude n'offre une vraie hiérarchie. Au mieux CAM/DAP permet de définir un seul niveau de hiérarchie où tous les composants figurent à la manière d'OpenCOM.

Seul FuseJ propose une symétrie des éléments La seule approche à être symétrique d'éléments est FuseJ. C'est son atout majeur, composants et aspects ne sont pas différenciés, seule leur interaction donne naissance à un nouveau type de composition par aspects.

La symétrie de placement se vérifie pour chaque approche L'asymétrie de placement était un défaut majeur d'AspectJ. Les approches de notre étude offrent la possibilité de séparer la définition de la coupe des codes advice.

La symétrie de portée n'apparaît dans aucune approche Aucune approche ne permet de manipuler clairement l'ensemble des aspects s'appliquant sur un point de jonction donné. Avec FuseJ et CAM/DAOP, par exemple, la connexion d'un composant avec un aspect se fait en les connectant explicitement, mais ceci reste insuffisant. Ce que nous recherchons ici est le mécanisme inverse, c'est-à-dire, pouvoir, depuis un composant donné, lister ses points de jonction et pouvoir composer comme on le souhaite les aspects sur chacun d'entre eux, en changer l'ordre, en limiter l'accès, etc.

Pour conclure sur ce chapitre, nous observons que d'une manière générale aucun modèle à composant ne satisfait nos conditions. Or, nous l'avons vu dans notre présentation de l'approche par aspects, les aspects doivent être greffés sur un paradigme existant. Le chapitre suivant détaille

le modèle à composants FRACTAL que nous avions succinctement abordé au Chapitre 2. FRACTAL répond à tous les critères que nous nous sommes fixé pour la partie composant. Ensuite la Partie II détaillera notre contribution au modèle FRACTAL en termes d'ajout des caractéristiques aspect.

Chapitre 4

Le modèle FRACTAL et ses outils

Sommaire

Ce chapitre présente en détail le modèle à composants FRACTAL et ses outils majeurs. Au chapitre précédent, nous avons décrit et analysé un ensemble d'approches utilisant conjointement composants et aspects. Nous en avons conclu qu'aucune de ces approches ne répond aux critères que nous avions sélectionnés au Chapitre 2. En particulier, aucun des modèles à composants ne répond à nos critères concernant la partie composant. Dans notre travail de rapprochement des composants et aspects, nous souhaitons nous appuyer sur un modèle répondant à tous ces critères : modèle général, hiérarchique, réflexif, extensible et doté d'un langage d'architecture. Le Chapitre 2 avait montré que FRACTAL y répondait positivement. Nous nous servons donc de FRACTAL comme base, et, en conséquence, présentons le modèle en détail dans ce chapitre. Le chapitre suivant justifiera les limites de FRACTAL en termes de séparation des préoccupations et positionnera notre contribution. Nous présentons donc dans ce chapitre FRACTAL et les outils de FRACTAL dont nous faisons usage par la suite.

FRACTAL est un projet du consortium ObjectWeb[4] initié par France Telecom R&D et l'INRIA [Bruneton et al., 2002, Bruneton et al., 2006]. Il se découpe en de nombreux sous projets dont les principaux sont un modèle à composants extensible et réflexif, un langage d'architecture extensible, un ensemble d'outils et des implantations dans différents langages de programmation. La Section 4.1 introduit les concepts et la notion graphique de FRACTAL. La Section 4.2 présente les outils que nous utilisons dans notre contribution. Enfin, la Section 4.3 conclut.

[4] http://www.objectweb.org/

4.1 Concepts et notation graphique

Un composant est traditionnellement introduit comme une unité de composition et de déploiement hautement réutilisable. En ce sens FRACTAL est doté des concepts classiques de composant, interface, composite, et liaison. Cependant FRACTAL se démarque des autres modèles en mettant l'accent sur la **variabilité**.

Définition 1. *Un composant* FRACTAL *est une entité dotée d'un ensemble* ***ouvert*** *de* ***capacités de contrôle****.*

Un composant FRACTAL a des capacités réflexives variables, c'est à dire des capacités à raisonner sur sa propre structuration et sur son propre comportement, grâce à sa membrane. Nous commençons par présenter les notions classiques d'interface et de liaison, avant d'étudier les spécificités de FRACTAL avec les notions de membrane et de composant partagé. Enfin, nous discutons de déploiement et de fabrique de composant.

La Figure 4.1 présente un méta modèle possible des concepts FRACTAL. La Figure 4.2 la notation graphique habituellement employée pour les composants FRACTAL, les notions décrites ci-après y sont représentées. Le méta-modèle met en évidence les notions classiques de composant, liaison , interface et composite. La particularité de FRACTAL, comme nous le voyons par la suite, est d'introduire la notion d'interface de contrôle et de composant partagé (un composant peut appartenir à plusieurs composites).

Interface et liaison

Un composant FRACTAL est doté d'interfaces fournies et requises qui sont appelées respectivement interfaces serveurs et clientes. Ces interfaces sont des points d'accès au composant. De plus, ces interfaces sont dites externes ou internes en fonction de la manière dont elles sont accédées : les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes ne sont visibles que par les sous-composants du composant (le composant est alors un composite).

Une liaison est un chemin de communication entre une interface cliente et une interface serveur. Une liaison se distingue par sa contingence et sa cardinalité. La contingence peut être optionnelle ou obligatoire : une liaison de type optionnelle autorise une interface à ne pas être liée lors du démarrage d'un composant, alors que dans le cas d'une liaison obligatoire le démarrage ne peut avoir lieu. La cardinalité d'une liaison est collection ou singleton : une liaison de type collection signifie que l'interface cliente ou serveur supporte plusieurs liaisons vers d'autres composants là où l'interface singleton n'en autorise qu'une seule.

La membrane ou le contrôle du composant

Un composant FRACTAL distingue deux parties : son *contenu* et sa *membrane*. Le contenu d'un composite est un ensemble de sous composants, et celui d'un composant dit primitif est l'implémentation de ses services fournis. La membrane d'un composant peut offrir un niveau d'interception et un niveau de contrôle (les capacités de contrôle du composant).

Le niveau d'interception est une capacité donnée à la membrane d'intercepter les communications entrantes et sortantes des composants. En d'autres termes, il s'agit d'intercepter les exécutions des opérations de ses interfaces serveurs et les appels vers des composants extérieurs grâce aux interfaces clientes.

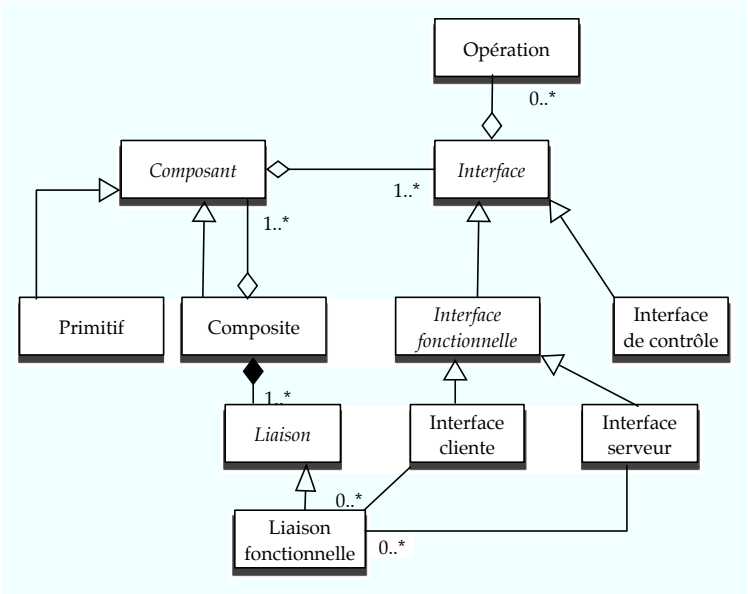

FIG. 4.1 – Le méta modèle FRACTAL.

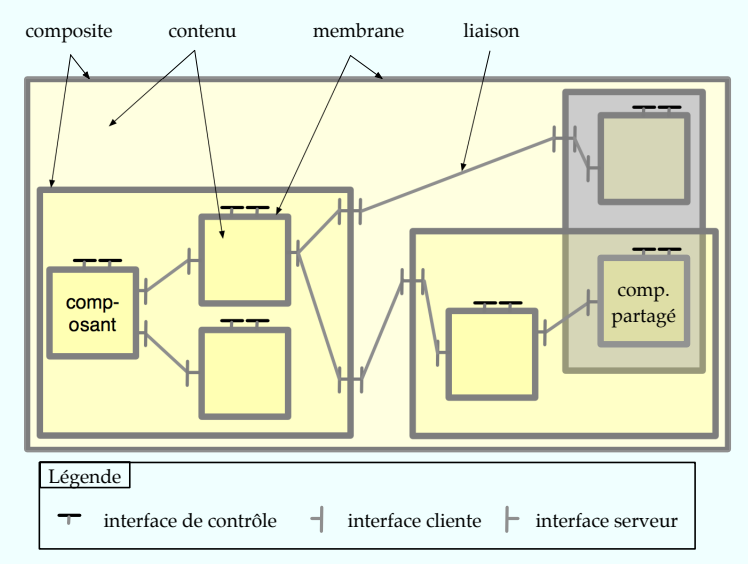

FIG. 4.2 – La notation FRACTAL.

Le niveau de contrôle est un ensemble d'interfaces qui gèrent les propriétés dites de contrôle du composant comme son cycle de vie, ses liaisons, son contenu, ou encore ses attributs. L'ensemble des interfaces de contrôle d'un composant est un ensemble ouvert. De nouvelles interfaces peuvent y être ajoutées. Les interfaces de contrôle prédéfinies du modèle ont à charge :

- le contrôle du contenu d'un composant permettant d'ajouter ou de retirer un sous composant,
- le contrôle des super composants permettant au composant de connaître ses composants parents,
- le contrôle du cycle de vie permettant d'arrêter ou démarrer le composant,

– le contrôle des liaisons permettant d'établir ou rompre un chemin de communication vers une interface fournie d'un autre composant ou encore de lister ces liaisons,

– le contrôle des attributs qui sont des propriétés définies par le programmeur du composant.

Le modèle s'affiche comme un modèle complètement décentralisé, chaque composant étant responsable, à travers sa membrane de la sémantique de son cycle de vie, de sa communication avec l'extérieur, ou de tout autre aspect extra-fonctionnel.

Le composant partagé

Le concept de *composant partagé* est induit par le contrôle des supers composants. Lorsqu'un composant appartient à plus d'un composant, cela signifie qu'il est partagé par ces composants. Une illustration est fournie sur la Figure 4.2. Notons, pour lever toute ambiguïté sur la notion de partage de composant, qu'un composant partagé par deux composants A et B, est différent d'un composant contenu dans le composant A, lui-même contenu dans le composant B. Ce concept est particulièrement utile pour le partage de ressources ou de services communs et évite de multiplier les interfaces clientes et la délégation vers ces ressources ou services, qui peuvent alors simplement être partagés par les composites qui les utilisent.

Par ce concept plutôt nouveau dans le monde des approches à composants, FRACTAL se présente comme un modèle récursif car le modèle d'une architecture n'est plus un arbre mais un graphe acyclique. Par contre, si l'on restreint le nombre possible de supers composants à un seul — choix réalisable dans l'implantation du contrôle des supers composants —-, le modèle est alors strictement hiérarchique.

Instanciation et déploiement

En ce qui concerne le déploiement et l'instanciation des composants, FRACTAL met en œuvre deux mécanismes : les fabriques et les *templates*. Tout comme les interfaces de contrôle, les spécifications FRACTAL déterminent une interface pour les fabriques. Cette interface s'articule autour de plusieurs opérations permettant d'instancier un composant à partir de la description de son contenu et de sa membrane. Le système de template permet quant à lui d'introduire la notion de composant-template. Ces composants sont des composants FRACTAL avec une interface de contrôle supplémentaire en charge de la fabrique du composant. Les templates sont particulièrement intéressants lorsqu'il est nécessaire d'instancier des assemblages de manière semblables.

4.2 Les outils FRACTAL existants

Nous nous attardons à présent sur un certain nombre d'outils autour de FRACTAL. Nous les réutiliserons à plusieurs reprises dans ce mémoire de thèse, notamment au chapitre qui introduira FAC, notre extension de FRACTAL pour le support d'aspects. Il nous semble donc important d'en donner les grandes lignes ici.

– AOKELL est une implantation des spécifications FRACTAL utilisant des aspects et des composants pour l'ingénierie du niveau de contrôle des composants.

– FRACTAL-ADL est un langage de description d'architecture et de déploiement.

– FRACTAL EXPLORER est une console graphique d'administration d'applications FRACTAL à l'exécution.

– FSCRIPT est un langage de script pour la reconfiguration dynamique d'applications FRACTAL.
– FRACLET est un modèle de programmation à base d'annotations pour l'implémentation des composants FRACTAL en Java.

A noter qu'il existe de nombreux outils autour de FRACTAL en plus de ceux présentés dans cette Section, ainsi que différentes implémentations dans des langages comme C, C++, SmallTalk (voir `http://fractal.objectweb.org` pour plus d'informations à ce sujet).

4.2.1 AOKELL

AOKELL [Seinturier et al., 2006] est une implantation en Java des spécifications du modèle FRACTAL. La particularité de cette implantation est d'utiliser des aspects et des composants pour l'ingénierie de la membrane d'un composant, l'objectif étant de fournir des mécanismes de haut niveau pour l'implantation de la sémantique de contrôle des composants. Les aspects sont utilisés pour fournir la glu entre la membrane et le contenu d'un composant. La membrane d'un composant est elle-même écrite à l'aide de composants, offrant ainsi le même pouvoir de composition qu'au niveau des composants de base.

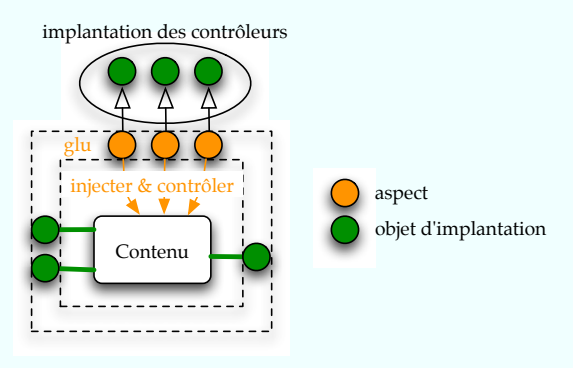

FIG. 4.3 – AOKELL : des aspects pour lier les interfaces de contrôle au contenu.

Des aspects pour lier le contrôle à la base fonctionnelle.

AOKELL est une illustration de l'utilisation des aspects au niveau de l'intégration des services de contrôle (voir Figure 4.3). L'approche permet de faciliter l'écriture de tels services et de les positionner dans la membrane des composants. Des modèles à base de conteneurs, tels que EJB ou CCM, définissent des architectures où les composants sont hébergés par des conteneurs qui fournissent les services techniques. Ces services vont de la gestion de la sécurité à la persistance, en passant par la gestion du cycle de vie. Généralement, ces services font partie du serveur d'applications et ne sont pas programmés avec les mêmes principes qui dirigent les composants : par exemple implantation en objet pur, alors qu'on est dans le contexte d'un modèle à composants. Ces services sont également difficilement modifiables ou extensibles, d'où le courant des approches à conteneurs ouverts [Vadet, 2004]. Le serveur d'application JBoss [Fleury and Reverbel, 2003] fait cependant exception en fournissant des services accessibles à des aspects du canevas JBoss AOP [Burke, 2003].

AOKELL suit cette même idée d'utilisation d'aspects pour lier le code des composants aux services techniques. La différence vient du fait que l'approche repose sur FRACTAL qui n'est pas

un serveur d'application, et que la notion de service technique n'existe pas en FRACTAL. On parle plutôt de dimension fonctionnelle et de dimension de contrôle. Ainsi, dans ce cas de figure, on voit que l'on peut utiliser des aspects conjointement avec des composants pour y intégrer des services techniques d'une manière plus souple.

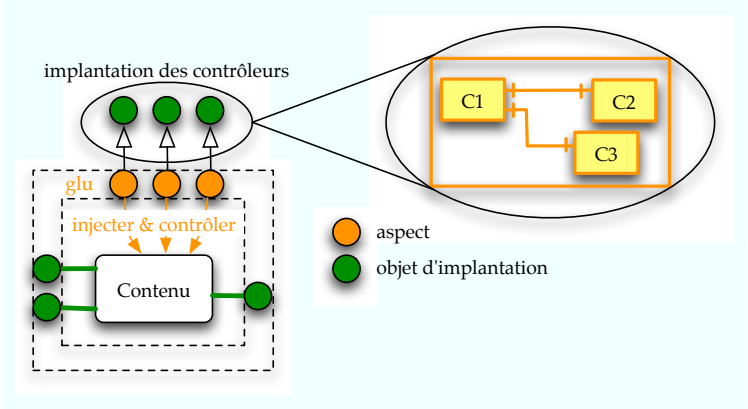

FIG. 4.4 – AOKELL : des composants pour l'ingénierie de la membrane.

Des composants pour une ingénierie plus flexible du contrôle

La seconde contribution d'AOKELL est de fournir une ingénierie de la membrane à l'aide de composants ce qui constitue une approche novatrice. Ainsi AOKELL définit un niveau méta et méta-méta (voir Figure 4.5). Le niveau de base correspond aux composants fonctionnels, le niveau méta aux composants de contrôle, le niveau méta-méta au contrôle des composants de contrôle, c'est-à-dire au contrôle du cycle de vie et des liaisons des composants méta. Ce dernier niveau n'est lui-même pas représenté à l'aide de composants, pour éviter une récursion infinie. Il semble en effet inutile de faire évoluer le contrôle du contrôle. Cependant rien n'empêche de faire remonter le pouvoir de composition des composants au niveau méta-méta, il s'agit ici simplement d'un choix somme toute raisonnable.

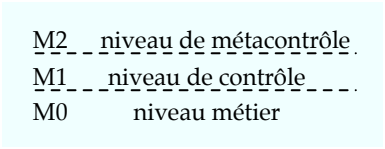

FIG. 4.5 – AOKELL : les couches de contrôle.

L'utilisation de composants pour le niveau de contrôle des composants FRACTAL est justifié par l'existence de dépendances entre ces interfaces. En effet, ces interfaces ne sont pas autonomes, mais collaborent pour réaliser les fonctions requises par la membrane. Par exemple, lorsqu'un composite est démarré, son contenu est traversé pour également démarrer ses sous composants. Le contrôle du cycle de vie utilise donc le contrôle du contenu (sous composants). D'autres dépendances similaires existent entre les contrôleurs. Nous ne les détaillons pas toutes ici. La Figure 4.6 propose la membrane d'un composant primitif (non composite) avec les dépendances entre contrôleurs. L'idée générale est d'appliquer à ces contrôleurs les mêmes principes qu'aux composants du niveau de base. Les dépendances vont se matérialiser en interfaces et liaisons entre ces interfaces. Ainsi l'ingénierie du contrôle peut bénéficier des mêmes propriétés que le niveau de base. Adapter une membrane devient une tâche plus aisée dès lors que les dépen-

dances sont explicites.

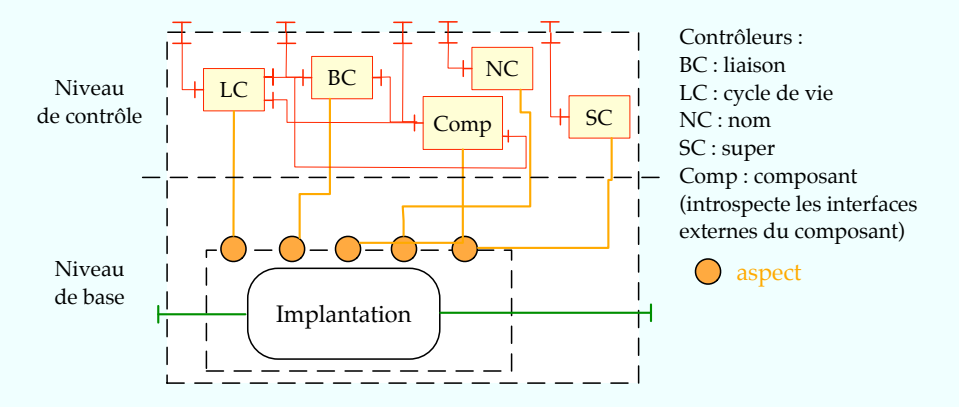

FIG. 4.6 – AOKELL : la membrane d'un composant primitif.

4.2.2 FRACTAL-ADL

FRACTAL-ADL est un Langage de Description d'Architecture (ADL) pour le modèle à composants FRACTAL. Il s'agit d'un langage en XML, extensible, dont la chaîne de compilation est elle-même implantée avec FRACTAL. Grâce à ce langage, les types de composant, leur implantation, les composites et les liaisons peuvent être décrits, indépendamment de l'implantation des composants.

FRACTAL-ADL est un canevas modulaire, qui peut être facilement étendu pour supporter des extensions de FRACTAL. Ainsi de nouvelles balises XML peuvent être ajoutées à la DTD, et la chaîne de compilation de l'ADL peut être étendue pour supporter cette balise. Cette chaîne de compilation est conçue comme un assemblage de composants FRACTAL. Un composant frontal (*frontend*) en amont de la chaîne analyse les fichiers XML. Un composant constructeur (*builder*) compile ces fichiers. Des composants dorsaux (*backend*) peuvent dynamiquement instancier l'architecture décrite par les fichiers XML. Un composant dorsal statique est également disponible pour la génération de code source Java, qui, lorsqu' il est exécuté, déploie l'architecture.

```
<definition name="comanche.Handler" extends="comanche.HandlerType">
  <component name="rd" definition="comanche.Dispatcher"/>
  <component name="frh" definition="comanche.FileHandler"/>
  <component name="erh" definition="comanche.ErrorHandler"/>
  <binding client="this.rh" server="rd.rh"/>
  <binding client="rd.h0" server="frh.rh"/>
  <binding client="rd.h1" server="erh.rh"/>
</definition>

<definition name="comanche.Backend" extends="comanche.HandlerType">
  <component name="ra" definition="comanche.Analyzer"/>
  <component name="rh" definition="comanche.Handler"/>
  <component name="l" definition="comanche.Logger"/>
  <binding client="this.rh" server="ra.a"/>
  <binding client="ra.rh" server="rh.rh"/>
  <binding client="ra.l" server="l.l"/>
</definition>

<definition name="comanche.Frontend" extends="comanche.FrontendType">
  <component name="rr" definition="comanche.Receiver"/>
  <component name="s" definition="comanche.MultiThreadScheduler"/>
  <binding client="this.r" server="rr.r"/>
  <binding client="rr.s" server="s.s"/>
  <binding client="rr.rh" server="this.rh"/>
</definition>

<definition name="comanche.Comanche" extends="comanche.RunnableType">
```

```
28   <component name="fe" definition="comanche.Frontend"/>
29   <component name="be" definition="comanche.Backend"/>
30   <binding client="this.r" server="fe.r"/>
31   <binding client="fe.rh" server="be.rh"/>
32 </definition>
```

Listing 4.1 – Description de l'assemblage de Comanche en FRACTAL-ADL

Le Listing 4.1 illustre la description de l'architecture du serveur web Comanche, preuve de concept pour une architecture reconfigurable en FRACTAL représentant un serveur web minimal. Nous ne présentons ici que la description des composites. Les primitifs seront présentés en Section 4.2.5 avec l'outil FRACLET. Une balise `definition` délimite un composant ou un composite, voir une définition partielle de composant ou composite, par exemple le type d'un composant. Une définition peut être étendue (option `extends`), au sens de l'inclusion des éléments de la définition étendue. Les déclarations du Listing 4.1 sont des descriptions de composites, dans lesquels nous trouvons donc naturellement des définitions de composants (par exemple Lignes 2–4) et de liaisons imbriquées (par exemple Lignes 5–7).

FRACTAL-ADL est un module essentiel à FRACTAL. Il permet de manipuler à plus haut niveau les assemblages de composants, indépendamment de leur langage d'implantation. Ainsi une architecture FRACTAL-ADL donnée peut être implantée dans différents langages de programmation. Dans le Chapitre 2, nous avons insisté sur l'importances des langages de description d'architecture (ADL) et dans notre travail de rapprochement des approches à composants et par aspects. Par conséquent, il nous semble important de pouvoir bénéficier d'un langage d'architecture.

4.2.3 FRACTAL EXPLORER

FRACTAL EXPLORER est une console graphique générique pour le management d'applications FRACTAL. FRACTAL EXPLORER est une personnalité du canevas Browser [Merle and Moroy, 2006] qui permet le développement de consoles graphiques. La console graphique d'OpenCCM [Merle, 2006] est une autre personnalité de ce canevas. L'outil permet de découvrir, introspecter, gérer et re-configurer des applications FRACTAL pendant l'exécution. Une application FRACTAL est représentée graphiquement comme un arbre matérialisant la hiérarchie entre les composants.

Comme beaucoup d'outils autour de FRACTAL, FRACTAL EXPLORER peut être étendu avec de nouvelles opérations. Le canevas est lui-même construit comme une application FRACTAL d'une soixantaine de composants. Nous faisons usage de cet outil dans notre contribution pour mettre en évidence différentes vues sur une architecture FRACTAL. Nous avons étendu l'outil pour supporter la représentation graphique de préoccupations transverses. La Figure 4.7 présente une capture d'écran de la console. On observe sous forme d'une représentation en arbre la liste des composants et des sous-composants. Il est possible d'agir sur certains contrôleurs de composant pendant l'exécution, comme dans l'exemple de la capture avec le contrôleur de cycle de vie pour démarrer ou arrêter un composant.

FRACTAL EXPLORER constitue un bon outil de suivi et de gestion d'une architecture FRACTAL en cours d'exécution. Le tissage d'aspects pouvant être réalisé dynamiquement, on peut alors piloter ce tissage d'aspects pour adapter une application pendant son exécution.

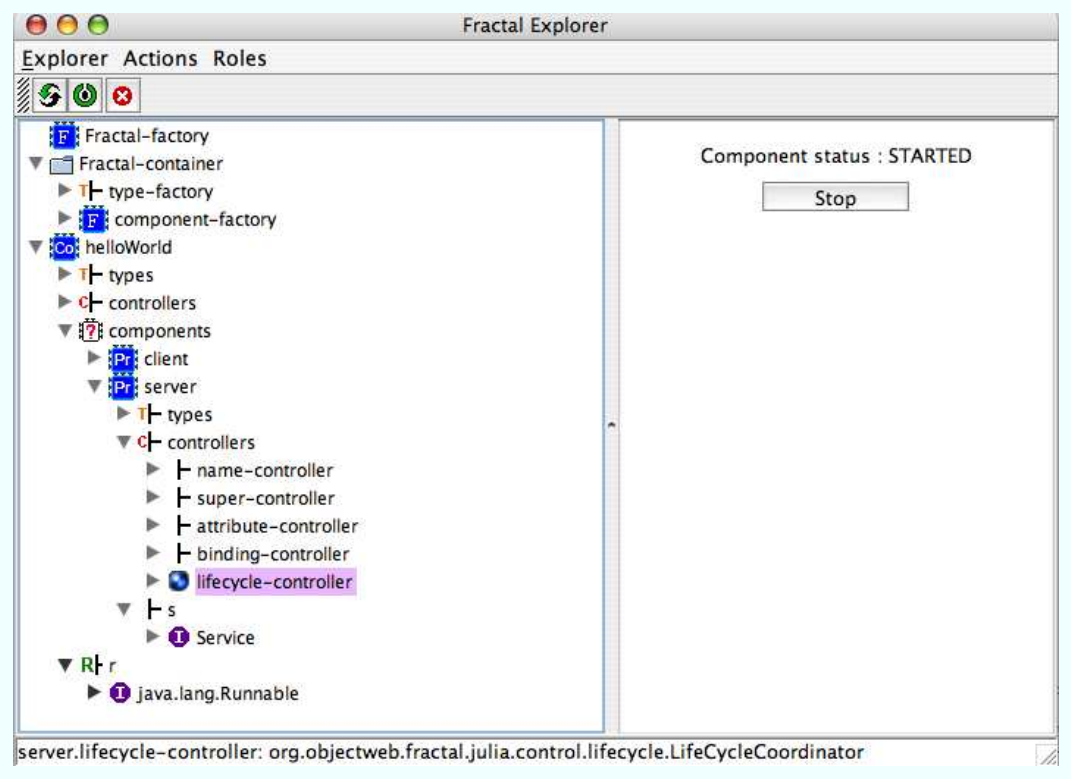

FIG. 4.7 – FRACTAL EXPLORER : capture d'écran d'une application simple client/serveur.

4.2.4 FSCRIPT

FSCRIPT est «*un langage dédié pour la reconfiguration dynamique consistante de composants* FRACTAL » [David, 2005]. A la manière du langage XPath [W3C, 1999] pour les documents XML, une architecture FRACTAL est représentée sous la forme d'un graphe qui peut être alors soumis à des requêtes (navigation dans une architecture) ou des reconfigurations (adaptation dynamique d'une architecture). La partie ayant en charge la gestion des requêtes sur une architecture est appelée FPATH. FSCRIPT englobe FPATH et permet de faire des reconfigurations dynamiques. Les similitudes entre la navigation dans une architecture à composants et la recherche de points de jonction dans l'approche par aspects, nous ont conduit à utiliser FPATH dans notre approche. Pour faciliter la discussion du chapitre suivant sur l'utilisation de FPATH et FSCRIPT, nous faisons ici une présentation rapide des principes de l'outil.

Fonctionnement des requêtes FPATH. FPATH peut être utilisé comme un module indépendant de FSCRIPT. La notation est inspirée d'XPath [W3C, 1999] — un langage de navigation dans des documents XML —, et permet de naviguer dans une architecture de composants FRACTAL représentée sous forme d'un graphe orienté. Les nœuds de ce graphe correspondent aux composants, à leurs interfaces et opérations, ou attributs. Les arcs connectant ces nœuds sont étiquetés du type de la relation les reliant. La Figure 4.8 présente une application FRACTAL client/serveur comportant un composite `root` et deux sous-composants `client` et `server`. La Figure 4.9 présente le graphe associé à cette application. Des arcs orientés étiquetés parent/enfant relient le composant `root` à ses deux sous-composants. Les arcs orientés étiquetés `binding` associent les interfaces clientes aux interfaces serveurs.

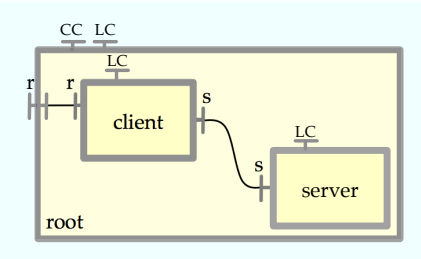

FIG. 4.8 – FPATH : une application type client/serveur.

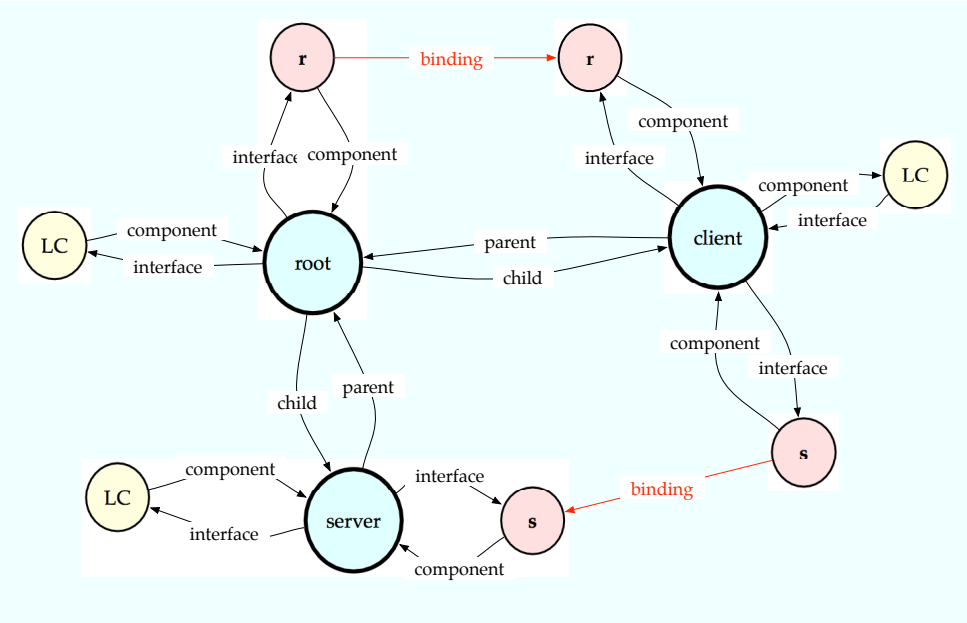

FIG. 4.9 – FPATH : le graphe de l'application client/serveur.

Une requête FPATH est donc une expression définissant un chemin dans le graphe représentant une architecture donnée. La syntaxe d'un chemin est de la forme suivante :

```
axe1 : :test1[pred1]/axe2 : :test2[pred2]/...
```

L'axe (`axe1`, `axe2`) représente une étiquette sur un arc. Un test (`test1`, `test2`) peut être soit un nom, soit une étoile (n'importe quel nom). Un prédicat (`pred1`, `pred2`) est une expression FPATH à valeur booléenne. L'expression suivante, par exemple, permet de trouver les composants partagés d'une architecture.

```
descendant-or-self : :*[count(parent : :*) > 1]
```

Dans l'exemple ci-dessus, l'axe `descendant-or-self` concerne le composant racine lui-même et tous ses sous composants. Il s'agit donc d'une requête qui parcourt toute la hiérarchie de l'architecture en partant du composant courant. Ensuite, le prédicat utilise la fonction `count` qui ici évalue le nombre de réponses à la requête `parent : :*` qui trouve tous les parents d'un nœud donné. Un composant partagé est bien un composant possédant plus d'un père. Pour plus de détails sur le langage FPATH, se référer à [David, 2005]. L'outil FPATH est particulièrement intéressant pour l'implantation d'un langage de coupe. En effet, les similitudes sont importantes,

et l'outil permet de facilement naviguer dans tous les éléments d'une architecture. Traditionnellement, les langages de coupe sélectionnent des points de jonction dans les éléments de base. Dans un scénario d'unification des composants et aspects, le langage de coupe doit pouvoir facilement identifier différents points dans l'architecture. FPATH semble donc très proche de ce qui est nécessaire de réaliser pour le tissage d'aspects dans FRACTAL.

Reconfigurations d'architecture avec FSCRIPT.

FSCRIPT est un langage pour la reconfiguration d'applications FRACTAL. Si FPATH permet simplement de parcourir le graphe représentant l'application, FSCRIPT définit un ensemble de fonctions de configuration et reconfiguration. Entre autres, FSCRIPT définit des structures de contrôle permettant la définition de scripts de reconfiguration avancés. FSCRIPT permet de mettre en oeuvre des mécanismes d'adaptation dynamique.

Ainsi FSCRIPT permet de naviguer et d'introspecter la structure, l'état, la configuration des composants. Cette structure peut être modifiée, les connexions peuvent être changées, et des composants peuvent être créés. Les états et la configuration sont au même titre modifiables.

FSCRIPT est un langage de script extensible, visant l'adaptation d'applications par reconfiguration. Si l'on considère le tissage d'aspects comme une reconfiguration, il semble évident que FSCRIPT peut jouer un rôle important pour l'intégration des mécanismes de l'approche par aspects à l'approche à composants.

4.2.5 FRACLET

FRACLET est une approche dirigée par les annotations pour le développement d'applications FRACTAL [Rouvoy, 2006, Rouvoy et al., 2006b, Rouvoy et al., 2006a]. Un ensemble d'annotations sont définies pour enrichir le code des composants et la sémantique de certains éléments du code (classe, champs) reliés au modèle de programmation FRACTAL. Par exemple, les champs d'une classe utilisés par FRACTAL comme interfaces requises sont annotés pour spécifier cette dépendance. Toutes ces annotations permettent de simplifier l'écriture du code des composants et de retirer une certaine redondance d'information. Les parties de ce code propres à l'implantation du modèle lui-même, c'est-à-dire reliées aux interfaces de contrôle, sont substituées par ces annotations. Ainsi le programmeur de composant peut focaliser son attention sur le code fonctionnel du composant, et non sur les artefacts techniques ou de contrôle du modèle à composants lui-même.

Dans notre contribution FAC, nous faisons usage de FRACLET auquel nous rajoutons de nouvelles annotations pour capturer nos besoins propres au développement par aspects. De plus, les extraits de code FRACTAL que nous montrerons dans ce mémoire seront annotés avec des annotations FRACLET qui rendent le code plus concis et plus lisible. Pour toutes ces raisons, nous présentons l'architecture du canevas FRACLET, ainsi que les annotations que nous utilisons dans ce mémoire.

Architecture du canevas.

FRACLET est une approche générative qui permet de soulager la tâche du programmeur de composant qui peut se concentrer uniquement sur l'écriture du comportement des composants. La Figure 4.10 présente l'architecture du processus de génération de FRACLET. Le code métier annoté des composants et l'ensemble des définitions des annotations et les générateurs sont passés en entrée du moteur FRACLET (voir étape 1 sur la figure). Des fichiers de description d'architecture sont générés ainsi que des classes Java qui sont les classes annotées métiers, passées en

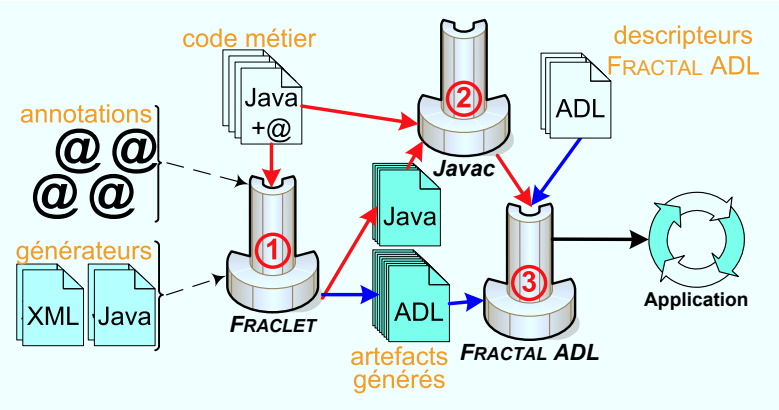

FIG. 4.10 – FRACLET : architecture du processus de génération.

entrée, enrichies de la sémantique de FRACTAL. Dans l'étape 2, les fichiers Java générés lors de la première étape sont compilés. Enfin lors de l'étape 3, les fichiers d'architectures générés sont compilés par FRACTAL-ADL. Sur la Figure 4.10, d'autres fichiers d'architecture sont fournis à FRACTAL-ADL, il s'agit des fichiers décrivant l'assemblage des composants. En effet, FRACLET permet de générer les descripteurs d'architecture des composants seuls, les composites décrivant un assemblage doivent être toujours écrits avec FRACTAL-ADL.

Annotation	Element	Description
`@Interface`	Interface	Définit une interface de composant
`@Component`	Classe	Définit un composant
`@Attribute`	Champ	Définit un attribut de composant
`@Provides`	Classe	Définit une interface fournie par un composant
`@Requires`	Champ	Définit une interface requise par un composant

TAB. 4.1 – FRACLET : les annotations principales.

En résumé, FRACLET permet de répondre aux problèmes du mélange de code métier et technique et de redondance des méta-informations lors du développement de composants FRACTAL. FRACLET se propose de réduire la taille du code écrit par le développeur en remplaçant le code technique des composants FRACTAL par des annotations. Ces annotations expriment la sémantique des concepts FRACTAL : interface, liaison, attribut, cycle de vie, sans imposer un modèle de programmation particulier. Le code annoté est ensuite analysé pour générer non seulement le code technique des composants FRACTAL mais aussi les fichiers FRACTAL-ADL qui leur sont associés.

4.3 Synthèse : vers une unification

Ce chapitre nous a permis d'introduire en détail FRACTAL, son modèle, ses concepts majeurs, et ses outils. Pour rappel nous redonnons le tableau faisant le bilan des approches à composants de notre étude du Chapitre 2 dans le Tableau 4.2. FRACTAL répond positivement à tous nos critères pour la partie «modèle de composant», à l'exception de la séparation des préoccupations dites transverses. Nous avons pu voir également, à travers plusieurs des outils autour de FRACTAL, qu'il existe une prédisposition particulière à l'ajout d'un mécanisme pour le tissage d'aspects. Nous avons établi des similitudes entre le langage FPATH et les langages de coupe dans les approches par aspects. Les facilités offertes par l'injection de dépendances et les annotations

de FRACLET, la navigation pendant l'exécution d'une architecture à composants avec FRACTAL EXPLORER qui utilise fortement le pouvoir réflexif de FRACTAL.

	EJB	Spring	OpenCOM	Fractal
Séparation des préoccupations fonctionnelles	oui	oui	oui	oui
Séparation des préoccupations techniques	non	oui	oui	oui
Général	non	non	oui	oui
Réflexion	non	non	oui	oui
Hiérarchique	non	non	oui	oui
Extensibilité	non	non	non	oui
Séparation des préoccupations transverses	non	non	non	non

TAB. 4.2 – Comparaison des approches à composants

Cependant, malgré toutes ces bonnes prédispositions au niveau des outils et des concepts, FRACTAL échoue dans le support des préoccupations transverses. De plus, nous l'avons vu au chapitre précédent, nous avons besoin de construire une approche fortement symétrique : symétrie des éléments, de la portée et du placement. Aucun de ces mécanismes n'est naturellement pris en charge par FRACTAL. Le chapitre suivant présente nos objectifs sur l'extension de FRACTAL pour la construction d'une approche unifiée à composants et par aspects. Le chapitre commence par montrer les limites de FRACTAL pour le support de préoccupations transverses, puis justifie le choix d'un modèle homogène et unifié.

Deuxième partie

Un modèle unifié pour composants et aspects

Chapitre 5

Objectifs : vers un modèle homogène et unifié

Sommaire

Ce chapitre introduit Fractal Aspect Component (FAC), un modèle homogène et unifié qui étend FRACTAL pour le support du tissage des préoccupations transverses. Notre démarche cherche à rapprocher les modèles à composants et par aspects. Les modèles à composants, grâce à leur nature structurante, peuvent naturellement servir de support pour l'intégration des techniques aspects qui sont souvent greffées sur des paradigmes existants, comme l'approche objet.

Notre choix s'est ainsi porté sur le modèle FRACTAL. Le Chapitre 4 a mis en évidence les propriétés de modèle général, extensible, réflexif, et hiérarchique de FRACTAL. Nous avons également pu étudier quelques outils autour du modèle qui mettent FRACTAL dans de bonnes prédispositions pour une intégration des principes de l'approche par aspects. Cependant, malgré toute la richesse du modèle, et sa séparation originale du fonctionnel et du contrôle, lorsque des préoccupations d'autres transverses sont à intégrer dans une application FRACTAL, du mélange de code intervient. La Section 5.1 démontre ces limites de FRACTAL pour le support des préoccupations transverses. La Section 5.2 justifie l'emploi d'un modèle homogène où services techniques et fonctionnels sont considérés au même plan. Enfin, la Section 5.3 explique pourquoi nous choisissons une unification, c'est-à-dire de ne faire aucune différence entre un composant et la partie comportementale d'un aspect.

Le chapitre suivant présente FAC, notre proposition de modèle homogène et unifié pour composants et aspects. FAC étend FRACTAL pour le support du tissage des préoccupations transverses. Dans ce chapitre, nous justifions l'emploi d'un modèle homogène et unifié et nous présentons les éléments consituant l'approche FAC.

5.1 Limitations pour le support des préoccupations transverses

En dépit de ses bonnes propriétés en termes de séparation des préoccupations sous forme de composants, FRACTAL n'échappe pas au problème de l'entrelacement de code. C'est ce que nous montrons ici à travers une étude de cas sur un exemple d'application à base de composants fourni avec la distribution FRACTAL : Comanche.

Comanche est un exemple d'assemblage de composants représentant un serveur HTTP minimal. Son rôle consiste à accepter des connexions TCP et à traiter les requêtes en lançant un fil d'exécution (*thread*). Pour réaliser ces tâches, le serveur doit : analyser la requête, la journaliser, et renvoyer une réponse sous forme d'un fichier HTML ou une erreur si le fichier n'existe pas. A partir de cette description, nous pouvons extraire plusieurs services : la réception de requêtes, l'analyse de requêtes, la gestion de requêtes et la journalisation de requêtes. Ces services vont être fournis par un certain nombre d'interfaces de composants qui, une fois assemblés, conduisent à l'architecture présentée sur la figure 5.1. L'architecture comporte un composant pour la journalisation (*logger*), la planification (*scheduler*), la gestion de l'expédition des requêtes (*dispatcher*) pour séparer la gestion des erreurs (*error manager*) de l'envoi des réponses aux requêtes par le serveur de fichiers (*file server*).

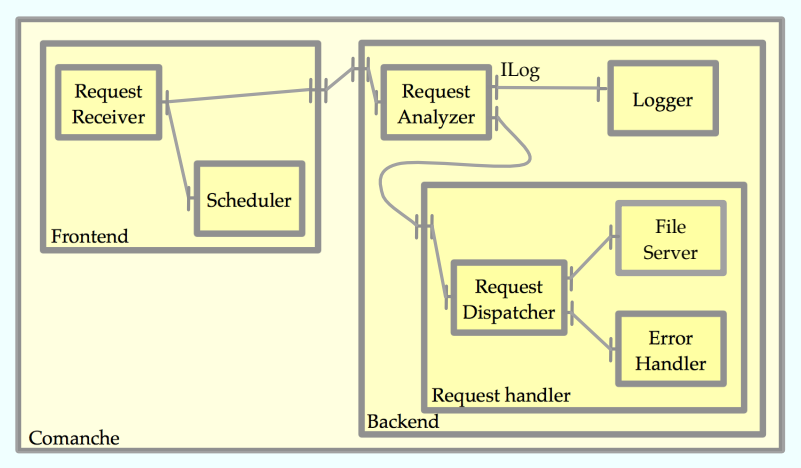

FIG. 5.1 – L'architecture de Comanche.

L'architecture obtenue fournit une séparation des préoccupations complète, dans le sens où chaque préoccupation est correctement incarnée dans un composant. Cependant, supposons que la fonctionnalité de journalisation concerne à présent plusieurs composants, en s'étendant par exemple au serveur de fichiers ou au gestionnaire d'erreurs. Ceci implique une restructuration par un ajout d'interfaces clientes pour ces nouveaux composants et une modification de leur code pour implanter cette nouvelle dépendance. Les composants *file server* et *error manager* doivent donc supporter la même interface cliente que le composant d'analyse de requêtes (*request analyzer*) : l'interface *ILog* notée sur la Figure 5.1.

La solution apportée par l'approche à composants seule, impose une restructuration forte de l'architecture. Le principal inconvénient vient de la nécessité d'intervenir au niveau du code d'un composant. Ainsi, le problème d'entrelacement de code des approches à objets persiste ici avec les composants. Les conséquences sont ennuyeuses car la séparation des préoccupations n'est plus vérifiée. Les dépendances ne sont pas uniquement limitées à un ajout d'interface mais se retrouvent dans le code concernant l'implantation d'autres interfaces. De plus, ces dépendances ne sont pas naturelles dans le sens où elles sortent du cadre de la fonctionnalité primaire de ces composants. Dans notre exemple, la préoccupation de journalisation n'est au fond pas requise,

de manière inhérente, par la préoccupation d'analyse de requêtes, ou celles de gestion de fichier ou d'erreur. En résumé, nous observons que l'approche à composants tout comme l'approche à objets n'est pas à l'abri des fâcheuses conséquences du mélange et de l'entrelacement de préoccupations pointées par l'approche par aspects.

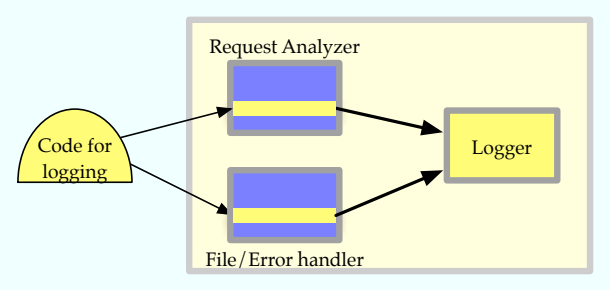

FIG. 5.2 – Mélange des préoccupations dans un composant.

Les causes de l'apparition du mélange de préoccupations dans un composant partent tout d'abord du constat suivant : la plupart des approches à composants font usage d'objets pour implanter les composants et les programmeurs continuent à suivre les concepts de l'objet pour développer les applications à base de composants. Les objets étant sensibles au phénomène de mélange de code, on retrouve ce mélange dans l'implantation des composants. Mais la cause principale semble plus profonde et plus en amont de l'étape d'implantation des composants, et semble venir de la conception. Comprenons bien ici que cette transversalité va au delà de l'introduction d'une nouvelle dépendance entre deux composants. Il ne s'agit pas ici d'un simple ajout d'interfaces requises. Fondamentalement, ajouter ces interfaces requises aux composants cités introduit du mélange dans ces composants, comme le montre la Figure 5.2. On observe à chaque fois que le code implantant ces composants va devoir appeler ce service de journalisation requis, introduisant du mélange dans le code implantant les interfaces fournies du composant.

Il semble donc manquer à FRACTAL, un pouvoir de composition plus fort permettant d'inverser les dépendances entre un composant encapsulant une préoccupation transverse et les composants sur lesquels va s'appliquer cette préoccupation. L'utilisation d'interfaces requises et de liaisons entraîne l'apparition de mélange dans le code implantant les interfaces fournies de ces composants. En ce qui concerne l'exemple Comanche, nous considérons donc que le composant de journalisation ne peut être correctement connecté à l'architecture avec les moyens mis à notre disposition dans le modèle sans briser la séparation des préoccupations.

5.2 Un modèle homogène

Dans la plupart des modèles à composants, une séparation stricte du métier et du technique est observée, comme dans les approches à base de conteneurs tels EJB [Bodoff et al., 2004] ou CCM [OMG, 2002]. Les services techniques dans les approches à conteneurs sont fournis par la plate-forme à composants. Parfois ces services techniques, de par leur nature transverse vis-à-vis des propriétés fonctionnelles, sont traités comme des aspects. Le serveur d'application JBoss [Fleury and Reverbel, 2003] met en oeuvre une utilisation d'aspects pour l'intégration des services techniques avec JBoss AOP [Burke, 2003]. Les aspects peuvent ainsi intégrer ses services techniques au code fonctionnel écrit par le programmeur de composants. Le tableau 5.1 caractérise cette séparation stricte. Deux niveaux existent : celui des composants et celui des conteneurs de composants. Dans le premier niveau, un paradigme composant est utilisé pour implanter les propriétés métiers et, dans le second, un paradigme différent (généralement objet) est utilisé pour

implanter les propriétés techniques (support transactionnel, persistance, etc.).

niveau	domaine	paradigme
conteneur base	technique métier	objet ou/et aspect composant

TAB. 5.1 – Modèle hétérogène : approches à base de conteneurs.

L'inconvénient majeur de cette famille d'approches vient de la réduction du pouvoir de composition due au fait que les services techniques et les propriétés fonctionnelles ne sont pas développés ni dans le même paradigme ni au même niveau. On a une séparation entre les conteneurs, qui peuvent utiliser des aspects ou des objets pour leur implantation, et les composants fonctionnels reposant sur un paradigme composant. Du fait de cette séparation stricte, les services techniques sont souvent figés, et leur liaison avec la base et les services techniques requiert de nouveaux mécanismes, comme par exemple de l'injection de dépendances.

Nous souhaitons construire un modèle pour composants et aspects *homogène*, c'est-à-dire ne faisant aucune distinction entre les fonctionnalités dites techniques ou non-fonctionnelles et les fonctionnalités métiers ou fonctionnelles. De plus nous voulons que tous ces services bénéficient du même paradigme. A noter que si toute préoccupation technique peut être qualifiée de non-fonctionnelle l'inverse n'est pas forcément toujours vrai. Par exemple, la gestion du cycle de vie d'un composant n'est peut être pas tout à fait une préoccupation technique au sens propre, puisque reliée au modèle à composants, mais certainement non-fonctionnelle car sortant du métier d'une application à composants. Cette distinction est importante car, dans notre définition de modèle homogène, nous souhaitons positionner les préoccupations techniques comme la persistance, la gestion de transactions, ou encore la journalisation au même niveau que les composants applicatifs, ce que nous appelons la *base*. Par contre, les propriétés non-fonctionnelles minimales et nécessaires au fonctionnement du modèle à composants lui-même, comme la gestion du cycle de vie et de la liaison, doivent rester selon nous en dehors de la base.

niveau	domaine	paradigme
Hétérogène		
contrôle base	contrôle & technique métier	objet composant
Homogène		
contrôle base	contrôle métier & technique	objet composant

TAB. 5.2 – Modèle homogène et hétérogène avec FRACTAL.

Si l'on revient au modèle FRACTAL, qui n'est pas une approche à base de conteneurs, mais qui, cependant, maintient une séparation entre fonctionnel et contrôle au niveau d'un composant, nous obtenons la situation du tableau 5.2. Appliquer un modèle hétérogène à FRACTAL revient à positionner les services techniques au niveau du contrôle des composants. Par exemple, le support transactionnel pour les composants ferait l'objet d'une interface de contrôle positionnée dans la membrane des composants. Cette étude a été menée par [Prochazka et al., 2003] où un contrôleur de transaction a été ajouté aux contrôleurs FRACTAL. Malheureusement, dans ce travail, le service transactionnel n'est pas construit à l'aide de composants, limitant ainsi le pouvoir de composition. Le service transactionnel ne bénéficie pas du paradigme composant.

Une manière de contourner le problème serait de faire appel aux membranes componentisés d'AOKELL (voir Section 4.2.1). AOKELL permet de construire la membrane d'un composant à l'aide de composants. Si le pouvoir de composition est plus grand que dans une approche comme JBoss AOP. La situation est alors celle présentée dans le tableau 5.3. On voit dans la partie haute

niveau	domaine	paradigme
Hétérogène		
méta contrôle	contrôle	objet
contrôle	contrôle & technique	composant
base	métier	composant
Homogène		
méta contrôle	contrôle	objet
contrôle	contrôle	composant
base	métier & technique	composant

TAB. 5.3 – Modèle homogène et hétérogène avec AOKELL COMP.

du tableau (*hétérogène*) qu'il est possible de faire bénéficier du paradigme composant pour l'implantation des services techniques. Cependant, il n'en reste pas moins le problème du lien entre la base et le contrôle, difficile à maintenir notamment lorsqu'il existe de nombreux liens entre les deux. Toute la clarté offerte par un assemblage de composants est ainsi perdue, du fait que l'on travaille avec deux dimensions différentes, leur interaction est invisible au niveau des architectures. Le choix de mettre toute préoccupation transverse dans la membrane d'un composant semble donc trop limitatif. Si la préoccupation transverse s'avère ne pas être technique et qu'elle utilise d'autres composants du niveau de base, la cohérence de l'ensemble du système sera difficile à maintenir et à établir.

En bref, au delà du débat sur la localisation des services techniques dedans/en dehors de la membrane, nous soulevons ici la relativité du fonctionnel et du non fonctionnel. Si la gestion de la transaction n'en demeure pas moins un service non-fonctionnel vis-à-vis des composants de base d'une application, rien n'empêche l'implantation de services transactionnels d'utiliser d'autres services, alors présents comme services fonctionnels. Le choix de reporter la gestion de la transaction dans la membrane oblige donc la création de liaisons entre des composants de niveau méta avec des composants du niveau de base. Dans un tel cas de figure, pourquoi ne pas tout positionner au niveau de base, laissant à la membrane la gestion minimale nécessaire au modèle à composants : gestion de la liaison, cycle de vie, interfaces, attributs de composant (voir les modèles homogènes des tableaux 5.2 et 5.3).

En résumé nous définissons un modèle homogène, comme un modèle positionnant les services techniques et fonctionnels au même plan et sous l'égide du même paradigme, dans notre situation le paradigme composant. Nous venons de positionner le rôle des composants dans FAC avec le modèle homogène, nous dévoilons à présent celui des aspects.

5.3 Un modèle unifié

Nous choisissons de construire un modèle qui *unifie* l'approche à composants et l'approche par aspects. Le terme est employé pour marquer une non distinction entre un composant et la partie comportementale d'un aspect. Ainsi, toute préoccupation, transverse ou non, peut être implantée par un composant, profitant du même pouvoir de composition. Le choix d'utiliser une préoccupation, de manière transverse ou non, doit être indépendant de l'écriture de la préoccupation même. Ce choix doit pouvoir être retardé au maximum. Dans l'esprit des composants pris sur l'étagère, un composant doit pouvoir être conçu indépendamment du fait d'être utilisé ensuite de manière transverse ou non.

A ses débuts, l'approche par aspects comblait un manque dans les approches à objets en termes d'extraction et de modularisation des préoccupations transverses au code. Les recherches

se sont ensuite tournées vers les modèles à composants. On constate que, à chaque fois, une nouvelle entité est introduite, l'aspect, nécessitant un nouveau langage. Ce langage permet d'exprimer ce nouveau pouvoir de composition. L'aspect est donc souvent représenté comme une entité à part, capable de raisonner sur la base (objets ou composants). La relation entre aspect et base est donc unidirectionnelle respectant ainsi la célèbre propriété de non conscience (*obliviousness*). Il peut alors être intéressant de se demander si cette distinction est réellement justifiée. En y regardant de plus prêt, on remarque que l'approche par aspects introduit deux nouveaux mécanismes : le comportement de l'aspect et sa connexion (tissage) à la base. On constate que le comportement d'un composant et d'un aspect ne sont pas si différents. Est-il réellement nécessaire de les distinguer ? L'écriture du comportement d'un aspect est en lui-même très similaire à celui d'un composant implantant des services décrits par des interfaces.

Si l'on accepte que seule l'interaction vis-à-vis de la base est nouvelle, alors, un nouveau type d'interaction est nécessaire, mais certainement pas une nouvelle entité indépendante. Ainsi, un aspect peut tout à fait être implanté par un composant. Seule la façon dont cet aspect va s'appliquer à la base sera différente, ce qui semble donc relever de l'ordre de la composition. Une approche radicalement asymétrique, *i.e.*, un aspect est une nouvelle entité, autorise uniquement un aspect à raisonner sur la base et non l'inverse. Ceci est limitatif, et engendre un certain nombre de problèmes lorsqu'un aspect devient complexe et requiert un pouvoir de composition plus fort que celui offert au niveau de l'aspect. Nous avons déjà évoqué ce manque de pouvoir de composition en discutant de la position des services techniques dans ou hors de la membrane dans la section précédente. Il nous semble donc important de ne pas faire de distinction entre un composant et un aspect pour la partie comportementale.

5.4 Conclusion

En résumé, nous optons pour un modèle homogène et unifié à base de composants et aspects. Les propriétés techniques ou métiers sont traitées au niveau de base, les propriétés transverses ou indépendantes sont implantées par des composants. Lorsqu'un composant intéragit de manière transverse avec d'autres composants, il est connecté à ces composants par un nouveau type de liaison que nous appelons la liaison d'aspect. La section suivante présente cette notion, ainsi que les autres concepts ajoutés à FRACTAL par FAC pour le support de l'approche par aspects. Les trois objectifs majeurs sont donc :

- Un support pour les préoccupations transverses,
- Représenter les aspects comme des composants (unification),
- Services fonctionnels et techniques sont tous représentés au même niveau (modèle homogène).

Chapitre 6

FAC : une extension de FRACTAL pour le support d'aspects

Sommaire

Ce chapitre présente les bases de Fractal Aspect Component (FAC), une extension du modèle FRACTAL qui unifie les principes des approches à composants et par aspects. Dans le chapitre précédant nous avons motivé la construction d'un modèle homogène et unifié et montré les limites de FRACTAL pour la séparations des préoccupations. Par modèle homogène nous entendons positionner les services techniques et fonctionnels au même niveau ; par modèle unifié nous entendons effacer les différences entre la partie comportementale d'un aspect et d'un composant. Par conséquent, notre modèle nous conduit vers l'ajout d'un nouveau mécanisme de composition à FRACTAL. Ce nouveau mécanisme a pour but d'augmenter la composition entre des composants pour l'expression de la transversalité.

Nous ajoutons un certain nombre de concepts au modèle FRACTAL que nous détaillons dans cette section. Ces concepts sont tantôt inspirés de l'approche par aspects, tantôt de l'approche

à composants. La Figure 6.1 propose un méta-modèle possible de FAC qui est basé sur celui que nous avions proposé pour FRACTAL (boite claire) auquel de nouveaux concepts sont ajoutés (boîtes grisées).

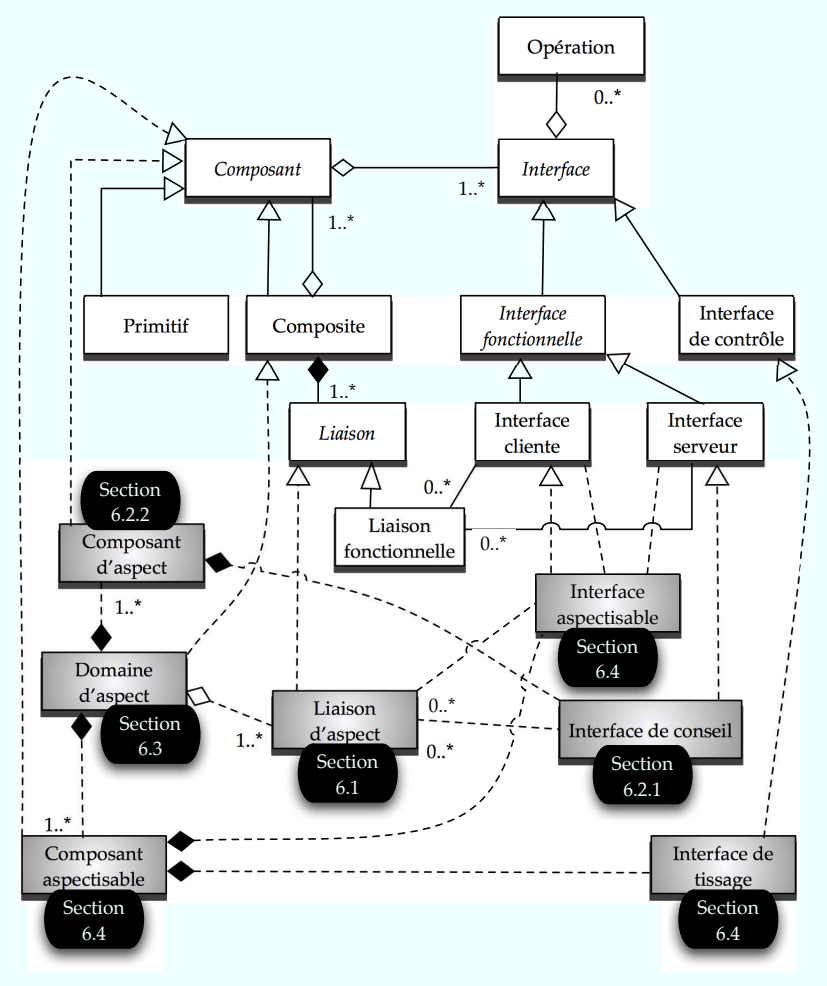

FIG. 6.1 – Le méta modèle FAC.

Ce chapitre va principalement se concentrer sur l'objectif d'unification définit au chapitre précédent. Les objectifs de modèle homogène et de support des préoccupations transverses sera plus détaillé dans le chapitre suivant. Comme nous introduisons dans ce chapitre les bases de FAC, nous définissons un certain nombre de sous-objectifs liés au travail de correspondance que nous réalisons entre l'approche par aspect et le modèle à composants FRACTAL. Nous cherchons à rapprocher les grands concepts de l'approche par aspects et à les intégrer à FRACTAL, de la manière la plus intuitive possible, en tenant compte des concepts composants existants. Nous allons donc montrer progressivement comment les concepts d'aspect, de code advice, de coupe, de point de jonction, et enfin de tissage sont vus avec FAC. Nous cherchons également, lorsque cela est possible, à améliorer les insuffisances de l'approche par aspects. Nous nous fixons, en conséquence, les trois objectifs suivants :

- renforcer l'encapsulation des aspects en leur donnant plus de propriétés structurantes grâce aux composants,
- rendre plus visible et plus explicite l'action des aspects dans un système,
- rendre plus sûre l'utilisation d'aspects dans un système, en contrôlant plus les actions de

tissage, en vérifiant, par exemple, avant ou après tissage la pertinence d'une coupe.

Ce chapitre est décomposé de la manière suivante. La Section 6.1 introduit le concept de *liaison aspect* qui permet de connecter deux composants ensemble, l'un des deux constituant la partie comportementale d'un aspect, le *composant d'aspect* présenté en Section 6.2 ainsi que son *interface de conseil* qui est l'interface serveur acceptant les liaisons d'aspect. Nous détaillons ensuite le concept de *domaine d'aspect*, dans la Section 6.3 qui permet de réifier l'impact d'un composant d'aspect.

Pour la mise en place de la liaison d'aspect, et par analogie à la gestion de la liaison en FRACTAL, une nouvelle interface de contrôle est nécessaire, l'*interface de tissage*, présentée dans la Section 6.4. La Section 6.5 présente le modèle de points de jonction de FAC, ce qui nous permet d'expliquer comment nous interceptons le comportement des composant, ainsi que le langage de coupe utilisé par FAC. Enfin la Section 6.6 explique comment le tissage est réalisé avec FAC.

6.1 Liaison d'aspect

La liaison d'aspect introduit une sémantique différente de la liaison FRACTAL habituelle. Cette nouvelle sémantique traduit un besoin qui s'articule autour de deux points fondamentaux. Premièrement, un composant requiert une fonctionnalité de nature transverse. Deuxièmement, cette dépendance sort du cadre de la fonctionnalité première du composant, donc ne peut être incarnée par une simple interface requise.

Définition 2. *Une **liaison d'aspect**, ou liaison transverse, caractérise un lien à connotation aspect ou transverse entre deux composants. Le client doit être un composant aspectisable, donc être doté de l'interface de tissage, le composant serveur doit fournir l'interface de conseil.*

Le premier point seul ne suffit pas à justifier le besoin d'une nouvelle sémantique de composition. En effet, si un composant a une dépendance vis-à-vis d'une propriété qui s'avère être transverse à plusieurs composants, ces composants peuvent simplement être liés au composant incarnant cette propriété transverse. Cependant, si cette propriété sort totalement du cadre des fonctionnalités fondamentales de ces composants, alors l'introduction d'une interface requise introduira un mélange comme nous l'avons montré dans le chapitre précédent avec les limites de FRACTAL pour le support de préoccupations transverses (voir Section 5.1). Ce second point est essentiel, car il est la traduction de la propriété fondamentale d'*obliviousness* en aspect [Filman and Friedman, 2000], c'est-à-dire la *non conscience* d'un composant d'être sous le contrôle d'un aspect.

6.2 L'interface de conseil et le composant d'aspect

En liant deux composants FRACTAL par une liaison d'aspect, nous donnons la possibilité au *composant d'aspect* de définir le comportement transverse à tisser. Ce comportement, nous le verrons dans notre modèle de points de jonction, s'applique sur les opérations interceptées par l'interface de tissage des composants aspectisables. Dans l'idée de donner un maximum d'informations liées au contexte d'interception, une interface spécifique est définie pour le composant d'aspect : l'**interface de conseil**. Comme nous pouvons le voir sur le méta-modèle de la Figure 6.2, le composant d'aspect est simplement un composant FRACTAL fournissant l'interface de conseil, une interface serveur fonctionnelle.

Définition 3. *Un **composant d'aspect** est un composant* FRACTAL *fournissant au moins une interface de conseil. Il définit le comportement d'un aspect (voir métamodèle de la Figure 6.2).*

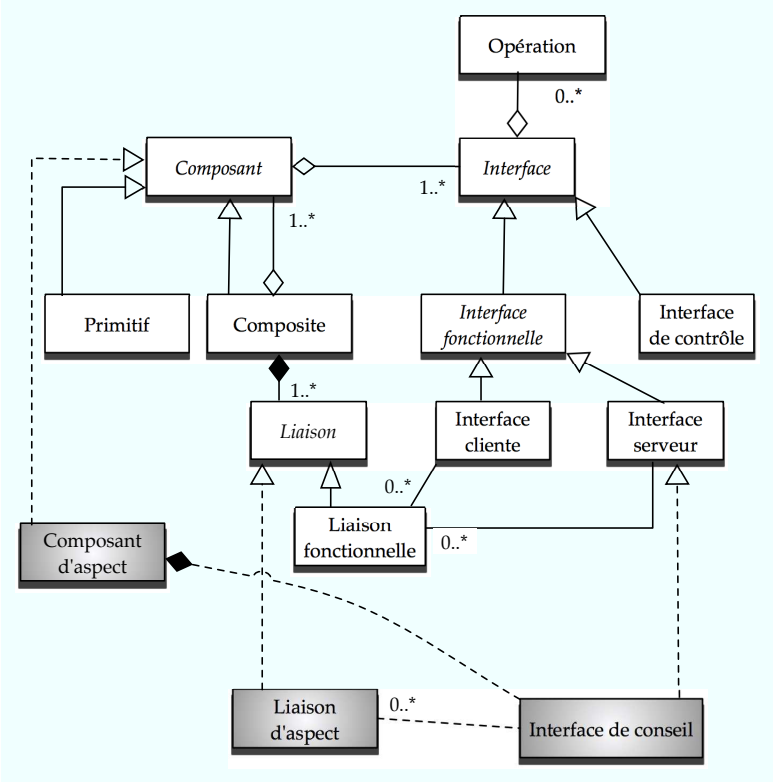

FIG. 6.2 – Le méta modèle FAC : interface de conseil et composant d'aspect.

Définition 4. *Une **interface de conseil** – ou interface de code advice – est une interface serveur définissant un code advice de type before, after, ou around, c'est-à-dire un code qui doit être tissé avant, après ou autour une opération de composant interceptée par l'interface de tissage. Elle fournit une réification d'une invocation d'opération de composant en délivrant un certain nombre d'informations liées au contexte d'interception, tels le nom de l'opération interceptée, son composant, ses paramètres et leurs types.*

6.2.1 Interface de conseil

Le Listing 6.1 fournit un exemple d'implantation de l'interface de conseil. L'interface de programmation définit un code advice de type autour ainsi que son implantation, et utilise l'interface de programmation définie par AOP Alliance[5], un projet open-source pour la définition d'une interface de programmation commune pour un certain nombre d'approches par aspects dynamiques.

```
1/**
2 * Spécifie une interface de conseil de type around
3 *
4*/
5interface AroundAdviceInterface extends org.aopalliance.intercept.Interceptor {
6  /**
7   * Réification d'une invocation sur une opération d'interface de composant.
8   * @param m l'objet réifiant l'invocation d'opération
9  */
10   Object invoke(FcMethodInvocation m) throws Throwable;
11}
```

[5] http://aopalliance.sourceforge.net/

Listing 6.1 – Interface de conseil : exemple d'interface de programmation sur le type autour

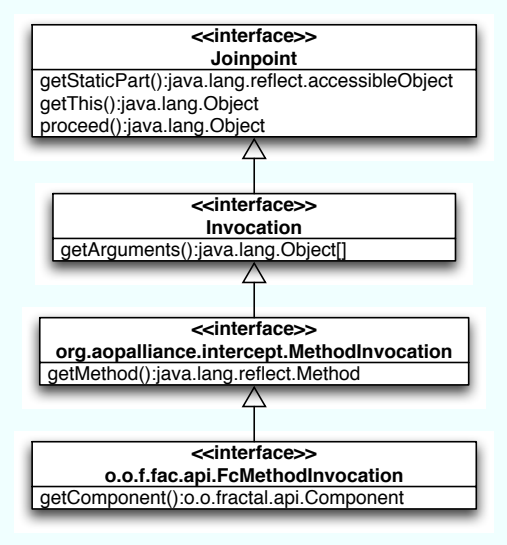

FIG. 6.3 – Interface de conseil : informations de contexte.

Implanter l'interface de conseil revient à implanter l'interface `AdviceInterface` et sa méthode `invoke`. Une interface de conseil est donc un *Interceptor* au sens de l'interface de programmation AOP Alliance. Le paramètre `m` de la méthode `invoke` réifie une invocation sur une interface FRACTAL. Ce paramètre donne donc accès à un certain nombre d'informations liées à ce contexte : le nom du composant, le nom de la méthode, de l'interface, les arguments passés à cette méthode, leur type, etc. Ces informations de contexte sont résumées sur la Figure 6.3. Il est par exemple possible de connaître le composant concerné, l'opération interceptée, ou encore ses paramètres.

```
1 /** Implantation d'un composant d'aspect de journalisation */
2 public class Trace implements AdviceInterface{
3   /** Implantation de l'interface AdviceInterface et de sa méthode invoke */
4   public Object invoke(FcMethodInvocation m) throws Throwable {
5     // traitement avant l'invocation de l'opération
6     System.out.println(m.getComponent() + " before method " + m.getMethod().getName());
7     // permet de contrôler l'exécution du code intercepté déclanché par l'appel à proceed()
8     Object ret = m.proceed();
9     // traitement après l'invocation de la méthode
10    System.out.println("/// Trace AC: after method "+ m.getMethod().getName());
11    return ret;
12  }
13}
```

Listing 6.2 – Exemple d'une implantation de l'interface de conseil pour la journalisation sur la sortie standard

Le code du Listing 6.2 est un exemple d'implantation d'un service d'impression sur la sortie standard. Cet exemple fournit une illustration de l'utilisation du contexte d'information. Le service de journalisation affiche sur la sortie standard (ligne 6) : le composant et le nom de la méthode interceptée. Le code reste très similaire à l'écriture d'un code *advice* de type *autour* dans les approches aspects appliquées aux objets. Nous présentons à présent la notion de composant d'aspect qui est un composant équipé de l'interface de conseil.

6.2.2 Le composant d'aspect

Au delà de la définition d'un comportement transverse, le rôle principal du composant d'aspect est de matérialiser une bonne pratique dans les approches par aspects, à savoir l'utilisation de la délégation pour l'implantation d'un aspect. Le principe est le suivant.

Dans les approches dites asymétriques — un aspect et un élément de la base (objet, composant) sont deux entités différentes —, il est alors souvent conseillé de déléguer l'implantation des codes advice à des éléments de base, ainsi la propriété devient réutilisable quelle que soit sa nature. Le code advice se charge simplement d'appeler ce comportement. Dans le contexte composant, le composant d'aspect ne joue alors qu'un rôle de délégation vers le composant implantant la propriété à tisser. Cette propriété est donc plus réutilisable, et le composant d'aspect a simplement un rôle de communication, ou de choix de bonne stratégie, car il peut choisir au cours de son exécution d'utiliser différents services de différents composants en fonction du contexte.

6.2.3 Exemple sur la journalisation.

Reprenons notre exemple de journalisation du Listing 6.2, et supposons que la politique du composant de journalisation change. Au lieu d'afficher sur la sortie standard, il serait souhaitable d'enregistrer ces informations dans un fichier de journalisation. L'implantation du Listing 6.2 n'est pas propre dans le sens, où notre modification requiert le changement du code du composant. En ce sens, il serait plus clair de séparer les deux rôles actuellement mélangés dans ce code : celui du choix de la stratégie de journalisation et celui de la journalisation elle-même. Le Listing 6.3 fournit une implantation différente et recommandée, qui sépare correctement ces deux rôles. Nous avons à présent un composant d'aspect et un composant tous les deux reliés par une liaison standard, chacun implante leur rôle premier. Ainsi le passage à notre politique de journalisation, dans des fichiers, plutôt que sur la sortie standard, peut être réalisé en implantant un nouveau composant et en changeant la connexion entre le composant d'aspect et le composant comme illustrée sur la Figure 6.4.

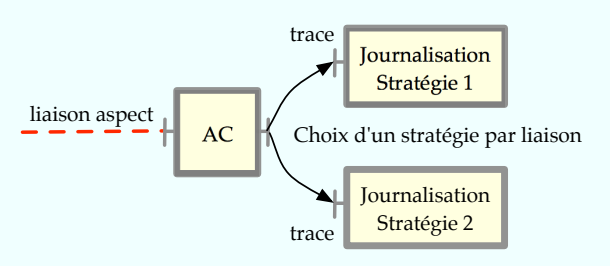

FIG. 6.4 – Composant d'aspect : exemple de la journalisation.

```
1  /**Implantation d'un composant d'aspect déléguant à un composant de journalisation */
2  @AspectComponent
3  public class TraceAC implements AdviceInterface{
4    @Requires(name="traceinterface")
5    TraceInterface trace;
6     /** Implantation de l'interface AdviceInterface et de sa méthode invoke */
7    public Object invoke(FcMethodInvocation m) throws Throwable {
8      // Appelle le composant de trace
9      trace.trace(m.getComponentName() + " before method " + m.getMethod().getName());
10     // permet de contrôler l'exécution du code de base (intercepté)
11     Object ret = m.proceed();
12     // Appelle le composant de trace
13     trace.trace("/// Trace AC: after method "+ m.getMethod().getName());
14     return ret;
15   }
16 }
```

```
17
18 /** Interface de journalisation */
19 @Interface(name="traceinterface")
20 public interface TraceInterface { void trace(String msg); }
21
22 /** Composant de journalisation sur la sortie standard */
23 @FractalComponent
24 @Provides(name="traceInterface")
25 public class SimpleTrace implements TraceInterface {
26   /** implantation de la journalisation pour écrire sur la sortie standard */
27   public void trace(String msg) { System.out.println(msg); }
28}
29
30 /** Composant de journalisation par écriture dans des fichiers */
31 @FractalComponent
32 @Provides(name="traceInterface")
33 public class SimpleTrace implements TraceInterface {
34   /** implantation de la journalisation dans des fichiers */
35   public void trace(String msg) { (...) }
36 }
```

Listing 6.3 – Journalisation avancée : séparation du choix de la stratégie, de son implantation

Le code du Listing 6.3 est découpé en trois parties : les lignes 1–16 correspondent au composant d'aspect, les lignes 18–20 à l'interface de journalisation, enfin les lignes 22–36 aux deux composants de journalisation (politique sur la sortie standard et par fichiers de journalisation). Concernant le composant d'aspect, remarquons tout d'abord la présence de l'annotation `@AspectComponent` qui est un raccourci et une combinaison des annotations `@FractalComponent` et de `@Provides(interfaces=@Interface(name="ac", signature=AdviceInterface.class))`. Un composant d'aspect est donc bien un composant FRACTAL doté de l'interface de conseil. Dans l'implantation de cette interface nous pouvons remarquer, Ligne 9, que l'opération `trace()` de l'interface requise *trace* (déclarée Lignes 4 et 5) est appelée. Ainsi le choix d'utiliser telle ou telle stratégie est reporté au moment de la liaison du composant d'aspect au composant de journalisation. L'interface du service est déclarée Lignes 18–20 et le reste du listing fournit les deux implantations correspondant aux deux stratégies.

En résumé, les notions d'interface de conseil et de composant d'aspect, permettent de valider la projection des notions de l'approche par aspects sur le modèle FRACTAL. En particulier, les notions d'aspect et de code advice. Nous rapprochons l'interface de conseil de la définition d'un code advice, ainsi qu'un composant d'aspect d'un aspect. Nous réalisons bien une unification car le composant d'aspect est un composant FRACTAL. Nous étudions à présent une notion originale vis-à-vis de l'approche par aspect : le domaine d'aspect.

6.3 Le domaine d'aspect

Le domaine d'aspect est la matérialisation de l'impact d'un composant d'aspect connecté à de nombreux composants par une liaison d'aspect par un composite FRACTAL. Si la liaison d'aspect permet de connaître du point de vue du composant aspectisable, les aspects actifs sur ses opérations, le domaine d'aspect est en quelque sorte une vue sur l'influence d'un composant d'aspect. Chaque tissage de composant d'aspect va donner lieu à la création d'un domaine d'aspect, et tous les composants liés à ce composant d'aspect seront ajoutés au composite ainsi que le composant d'aspect lui même. Nous revenons plus tard sur le tissage d'aspects dans la Section 6.6 p.78.

Définition 5. *Un **domaine d'aspect** est un composant-composite* FRACTAL *contenant un ou plusieurs composants d'aspect et l'ensemble des composants qui y sont connectés par une liaison d'aspect. Il matérialise l'influence de composant(s) d'aspect et délimite le domaine du connecteur aspect. Etant donnée la nature transverse des liaisons d'aspect, le domaine d'aspect permet également l'établissement de ces liaisons de manière valide car établies sous l'égide d'un composite. En effet, en* FRACTAL*, toute liaison doit être établie dans un composite contenant les deux composants à lier.*

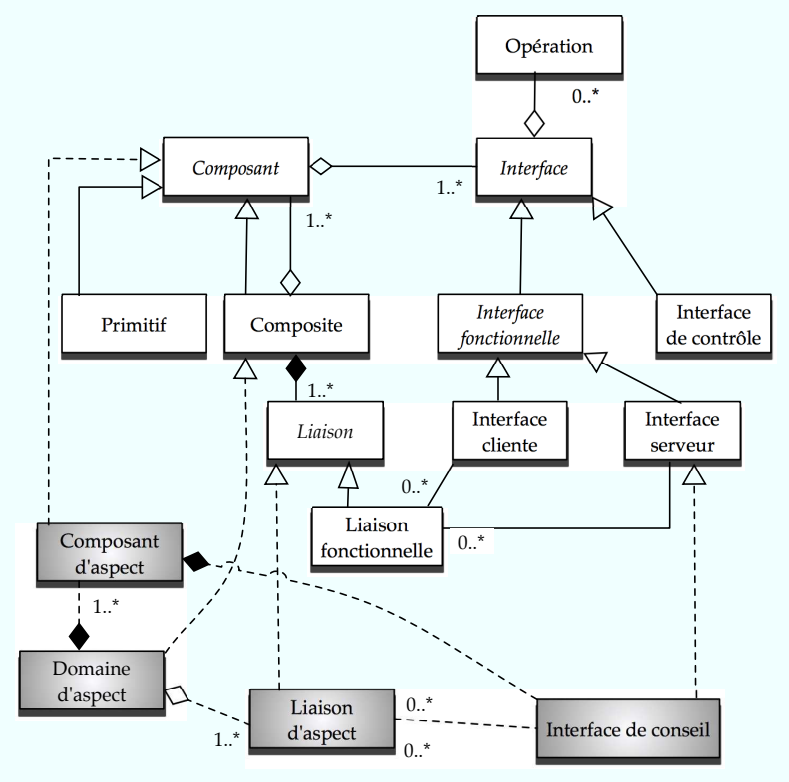

FIG. 6.5 – Le méta modèle FAC : le domaine d'aspect.

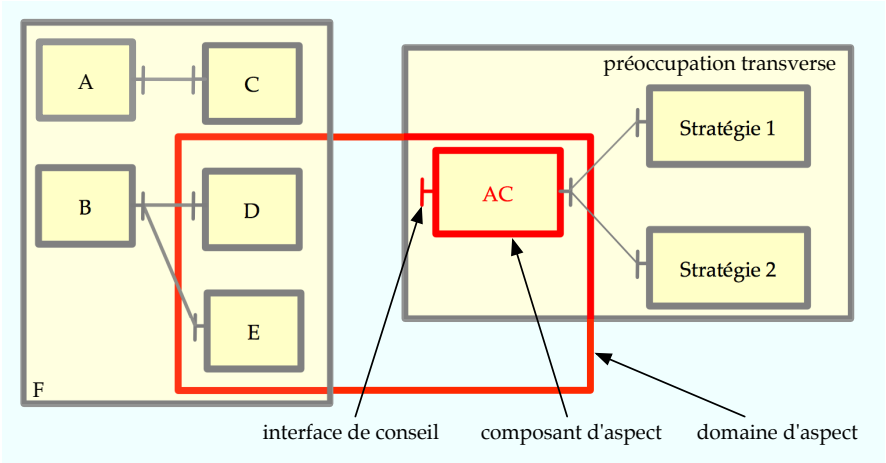

FIG. 6.6 – Le domaine d'aspect : exemple d'un composant d'aspect

Le domaine d'aspect utilise la notion de composant partagé de FRACTAL. Sur la Figure 6.6, par exemple, le domaine d'aspect du composant d'aspect concerne les composants D et E. Ces deux composants se retrouvent alors partagés par le domaine d'aspect et le composite F. Nous expliquons à présent comment le tissage de composants d'aspect est mis en œuvre grâce à l'interface de tissage.

6.4 Interface de tissage

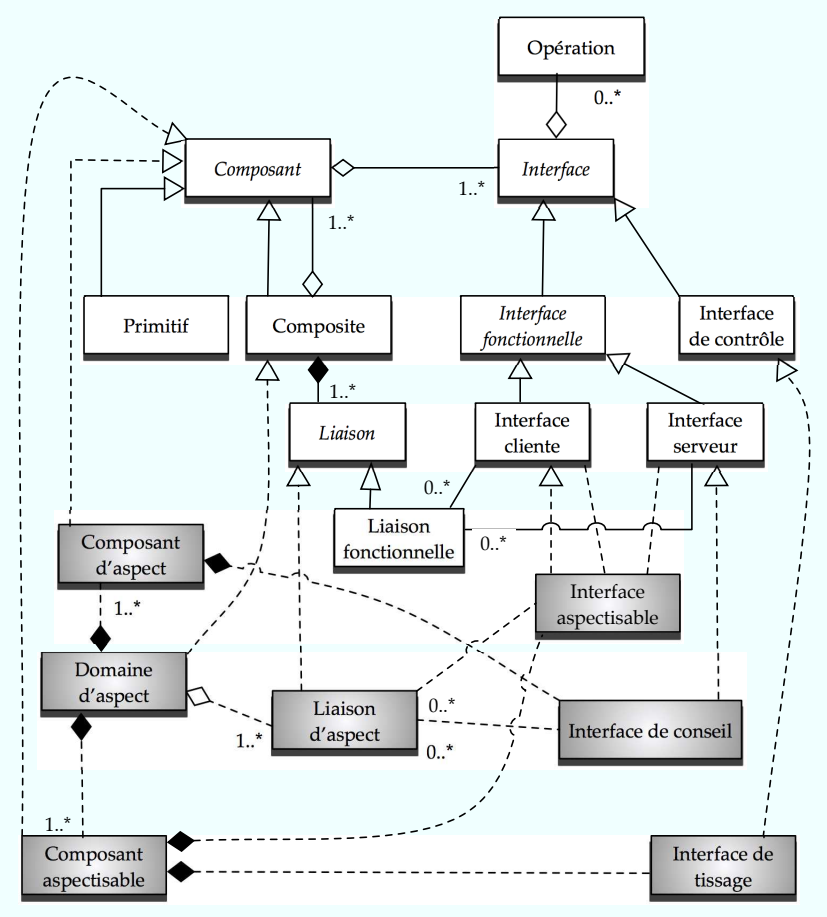

FIG. 6.7 – Le méta modèle FAC : l'interface de tissage, le composant d'aspect, l'interface aspectisable et le composant aspectisable.

L'interface de contrôle du tissage ou **interface de tissage** est à la liaison aspect ce qu'est le contrôleur de liaison est à la liaison FRACTAL «classique». Elle a deux rôles majeurs. Premièrement, elle a à charge la bonne gestion de cette liaison : établissement, annulation et introspection. Deuxièmement, de part son existence, elle introduit la notion de *composant aspectisable*, un composant qui s'ouvre au tissage d'aspects sur le comportement de ses interfaces. Les éléments dont le comportement peut être enrichi par des aspects — encore appelés points de jonction en aspect —, sont les opérations des interfaces clientes et serveurs en FAC.

Définition 6. *Une **interface de tissage** est une interface de contrôle de gestion des liaisons d'aspect d'un composant aspectisable. Elle utilise les capacités d'interception de la membrane du composant, pour intercepter les appels entrants et sortants du composant (appel sur ses interfaces), pour les re-diriger vers le composant cible de la liaison d'aspect, la partie comportementale de l'aspect. En plus de son rôle de gestion des liaisons d'aspect, l'interface doit être capable d'ordonnancer ces liaisons lorsque plusieurs aspects s'appliquent sur la même opération. Enfin elle doit donner la possibilité de verrouiller l'accès à une opération.*

Définition 7. *Un composant est dit **aspectisable**, s'il est doté d'une interface de contrôle du tissage, ou interface de tissage. Il profite ainsi des capacités d'interception de sa membrane, pour le tissage d'aspects*

sur les opérations de ses interfaces. Un composant aspectisable peut être primitif ou composite (voir le méta modèle de FAC *de la Figure 6.1). Les interfaces d'un composant aspectisable sont des* ***interfaces aspectisables****. Seul un composant aspectisable peut supporter le tissage de préoccupations transverses à son comportement.*

L'interface de tissage permet donc de mettre en œuvre les liaisons d'aspect. Cependant, il peut être parfois intéressant d'utiliser un langage de coupe pour pouvoir sélectionner un ensemble de points de jonction sur lesquels tisser un composant d'aspect. L'interface de tissage offre pour cela une seconde opération dite de tissage. Avant d'en donner la sémantique, nous introduisons le modèle de points de jonction de FAC et son langage de coupe.

6.5 Modèle de points de jonction et langage de coupe

6.5.1 Modèle de points de jonction

Le modèle de points de jonction de FAC repose sur les opérations des interfaces clientes et serveurs d'un composant aspectisable. Ainsi les appels entrants ou sortants d'un composant peuvent être sous le contrôle d'un aspect. L'interception de ces appels est réalisée grâce à l'interface de tissage.

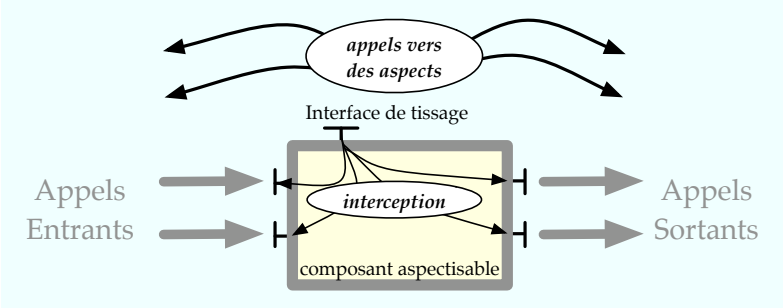

FIG. 6.8 – FAC : interception des appels entrants et sortants.

Un point de jonction est un élément du flot d'exécution d'un programme. Lorsque l'approche par aspects s'applique sur des objets, ces points de jonction sont souvent des méthodes, attributs, exceptions, etc. Dans le cadre d'une approche à composants, les points de jonction considérés sont les appels entrants et sortants d'un composant comme illustrés sur la Figure 6.8, c'est-à-dire, les opérations des interfaces clientes et serveurs d'un composant aspectisable.

Généralement, les approches par aspects reposent sur deux mécanismes importants : l'introduction et l'*advising*. FAC reposant sur un modèle à composants, le mécanisme d'introduction est déjà supporté par le modèle lui-même. Il est par exemple possible d'ajouter un composant dans un composite. L'ajout d'interfaces à un composant pourrait suivre l'idée d'introduction de champs ou méthodes à une classe dans les approches par aspects traditionnelles. Cependant, une telle action va à l'encontre de la définition d'un composant et en particulier de la définition d'un type de composant. Si de nouvelles interfaces sont ajoutées à ce composant, son type est corrompu. L'ajout d'opérations à une interface corrompt de la même façon la définition du type d'une interface. Pour toutes ces raisons, FAC se concentre uniquement sur l'*advising* qui consiste à intercepter l'exécution d'une méthode, par exemple, et à en modifier et/ou en enrichir le comportement.

De plus, nous n'interceptons que les interfaces externes d'un composant, car nous voulons préserver l'encapsulation des composants. Des approches comme JBoss AOP ou Spring AOP permettent de faire de l'*advising* sur les composants en ignorant cette propriété. Nous considérons donc ces deux approches comme contre-productives vis-à-vis de l'approche à composants. Pour rappel nous cherchons à tirer le meilleur des composants et des aspects, sans en retirer les composantes de base. Nous détaillons à présent le mécanisme d'interception requis pour la mise en place de l'*advising*.

6.5.2 Mécanisme d'interception

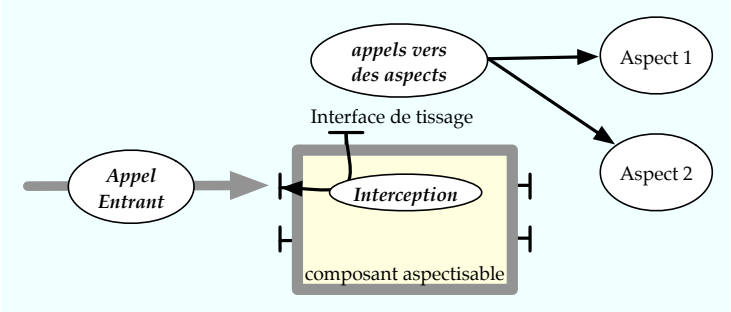

FIG. 6.9 – Mécanisme d'interception, schéma du scenario

FRACTAL fournit naturellement ce support au niveau de la membrane des composants. Ainsi, l'interface de tissage tire partie de cette propriété du modèle, pour accéder à chaque opération des interfaces fournies et requises d'un composant aspectisable. Lorsqu'une de ces opérations est appelée, l'interface de tissage en prend le contrôle, vérifie la présence d'aspects tissés sur cette opération. En général l'*advising* définit trois types de code advice, selon que ce code doit s'exécuter *avant*, *après*, ou *autour* de la méthode interceptée. Les Figures 6.9 et 6.10 fournissent une illustration du processus d'interception avec FAC qui utilise de l'*advising* de type *autour*.

1. Une opération d'une interface fournie est appelée sur le composant,
2. L'appel est intercepté par son interface de tissage,
3. L'interface de contrôle vérifie si des aspects sont tissés sur cette opération,
4. Dans notre scenario, deux aspects sont tissés sur l'opération, l'interface de tissage crée donc un objet `m` de type `MethodInvocation` qui réifie le contexte d'interception (La notion de contexte d'interception est expliquée avec la notion d'interface de conseil dans la Section 6.2),
5. L'interface de tissage transmet la liste des aspects tissés sur l'opération interceptée à l'objet `m`,
6. Un appel récursif est initié par l'appel de la méthode `proceed()` de l'objet *m*,
7. L'objet *m* appelle la méthode `invoke` du premier aspect de la liste (*aspect1*), l'objet *m* transmet sa propre référence à l'aspect *aspect1* comme objet d'appel de retour ou *callback*,
8. Le premier aspect applique son code défini comme *avant*, puis appelle l'objet de *callback m* en rappelant `proceed()` — d'où la récursion —,
9. L'objet *m* appelle le second aspect de sa liste,
10. l'aspect *aspect2* exécute son code de type *avant* puis à l'instar du premier aspect la méthode `proceed` sur l'objet *m*,

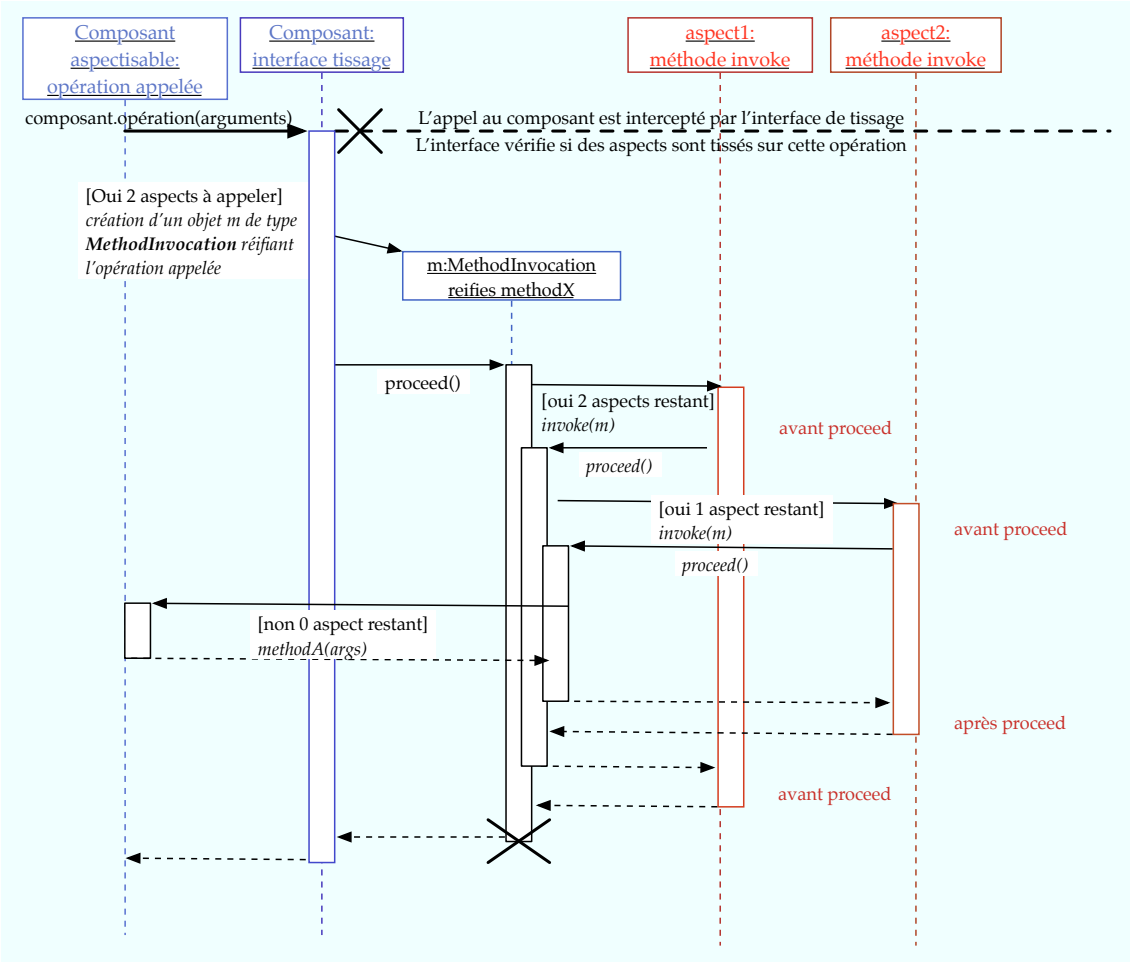

FIG. 6.10 – Mécanisme d'interception, diagramme de flot d'exécution du scenario

11. Le dernier aspect a été atteint dans la récursion, l'opération interceptée peut donc être appelée normalement `composant.opération()`, le résultat de cette exécution est capturé et transmis au second aspect,
12. L'aspect *aspect2* exécute son code de type *après*,
13. L'objet *m* transmet le résultat du code de type *après* du second aspect au premier aspect,
14. L'aspect *aspect1* exécute son code de type *après*,
15. L'objet *m* et sa méthode `proceed` se terminent et le résultat final est renvoyé à l'interface de tissage du composant, puis le résultat est transmis au composant en ayant fait la requête.

6.5.3 Langage de coupe

Avec FAC il existe deux façons de connecter un aspect dans une architecture à composants : établir des liaisons d'aspect manuellement, en appelant l'interface de tissage de chaque composant aspectisable, ou faire un tissage global, à l'aide d'une expression de coupe du langage de coupe de FAC. Un tissage global peut être réalisé en appelant directement l'interface de tissage, ou grâce à notre extension de FRACTAL-ADL, donc au niveau de la description de l'architecture. Nous détaillons l'utilisation de FRACTAL-ADL et de FAC dans la Section 6.6.1.

L'interface de tissage permet d'établir des liaisons d'aspect vers des composants incarnant des fonctionnalités transverses. Cependant, dans la littérature aspect, ce concept de liaison n'existe pas, il prend plutôt la forme d'un langage, dit langage de coupe. Un langage de coupe a pour but de fournir une manière efficace de sélectionner un ensemble de points de jonction dans un programme. Sur ces points de jonction viennent s'appliquer les comportements des aspects. Nous avons vu que dans notre modèle les points de jonction sont les opérations des interfaces clientes et serveurs d'un composant. Il est donc intéressant de pouvoir utiliser un langage de pattern, capable de les retrouver dans une architecture. FRACTAL étant un modèle hierarchique, nous savons qu'une opération appartient toujours à une interface, qui est toujours rattachée à un composant. Nous commençons par définir le langage de coupe de FAC, puis nous explicitons le fonctionnement d'un tissage en revenant sur les rôles de l'interface de tissage.

```
1  pcd::= <jp_type> <component>;<interface>;<method>
2  jp_type::= CLIENT | SERVER | BOTH
3  component::= une expression régulière sur le nom des composants
4  interface::= une expression régulière sur le nom des interfaces
5  method::= une expression régulière sur le nom des opérations
```

Listing 6.4 – Grammaire du langage de coupe de FAC

Le langage de coupe de FAC repose sur la grammaire du Listing 6.4. La grammaire présente une coupe comme une expression (*PcD : pointcut déclaration*) qui spécifie un type d'interception : `CLIENT`, `SERVER` ou `BOTH` et trois expressions régulières, séparées par des point-virgules. Ces expressions régulières définissent les noms des composants, interfaces et opérations à intercepter. Une expression de coupe prend ainsi la forme d'une requête sur une architecture de composants qui sélectionne un ensemble d'opérations de ces composants. Les expressions régulières et les types d'interception agissent comme des filtres sur cette requête.

Partant de cette similitude d'une coupe avec une requête sur une architecture, nous utilisons FPATH [David, 2005] dans l'implantation de notre mécanisme de tissage. Ainsi, lors du tissage d'un composant d'aspect, la coupe associée est transformée en requête FPATH sur l'architecture. Une expression de coupe doit être associée à un composite et peut être définie dans n'importe quel composant aspectisable (doté de l'interface de tissage). Ainsi, partant de ce composite, le contenu est inspecté par FPATH pour identifier les opérations correspondant à la coupe. Les capacités d'introspection et d'intercession permettent de retrouver ces opérations. Ensuite, pour chacun des composants contenant une ou plusieurs de ces opérations, les liaisons d'aspects sont établies en appelant à chaque fois l'interface de tissage du composant aspectisable. En quelque sorte, cette opération de tissage, utilisant une expression de coupe, est une factorisation des opérations individuelles d'établissement de liaisons d'aspect.

6.5.4 Introspection de coupe

L'introspection de coupe est la capacité d'une architecture de composants à rendre compte des interactions d'aspects qui s'y manifestent, c'est-à-dire donner des informations sur les aspects tissés sur ses composants, leur ordre de composition, sur quelles opérations ils s'appliquent, etc. Le rôle de l'interface de tissage a été étendu, lui conférant la possibilité de rendre compte à un instant donné des aspects opérant éventuellement sur chacune des opérations du composant contrôlé par l'interface.

```
1 Component[] listAC();
2 Component[] listCrosscutComps(Component root, Component ac);
3 PointcutRepresentation aspectizableComps(Component rComp, ItfPointcutExp pcut);
```

Listing 6.5 – Interface de tissage : introspection de coupe

Le Listing 6.5 (voir Annexe A pour le code complet de l'interface de tissage) présente la partie dite d'*introspection de coupe* du contrôleur de tissage. Trois opérations sont spécifiées pour obtenir la liste des aspects tissés sur un composant donné (Ligne 1), la liste des composants liés à un aspect donné dans une architecture donnée (Ligne 2) – un composant racine est donné, et ses sous-composants sont récursivement analysés –, la liste des opérations, interfaces et composants potentiellement capturés par une coupe donné (Ligne 3). Cette dernière opération peut être particulièrement utile pour vérifier avant tissage les éléments effectivement concernés.

Nous avons présenté le modèle de points de jonction de FAC ainsi que son langage de coupe. Nous nous limitons volontairement à l'interception des appels entrants et sortants d'un composant, ceci pour respecter la propriété d'encapsulation des composants. Ce choix valide notre objectif d'unification des composants et aspects car, un modèle de points de jonction qui atteindrait des points internes d'un composant, mettrait en péril les fondements de l'approche à composants et briserait le principe d'unification. Dans un second, temps nous avons présenté notre langage de coupe qui suit directement les types de points de jonction. Il permet donc de sélectionner des opérations d'interfaces des composants, grâce à un système classique d'expressions régulières. Enfin, nous avons vu comment nous tirons profit de la grande réflexivité de FRACTAL, en introduisant un mécanisme original, vis-à-vis de l'approche par aspects, d'introspection de coupe. Le principe est d'offrir plus de visibilité sur les composants d'aspects agissant dans une architecture à composants, ainsi que de pouvoir identifier à l'avance les éléments d'un composants correspondant à une expression de coupe. Cette dernière propriété constitue, en quelque sorte, un mécanisme de pré-condition ou de pré-vérification de coupe qui renforce notre objectif de support sécurisé des aspects au sein d'un système à base de composants. Nous rentrons à présent plus en détail dans le mécanisme de tissage d'aspects de FAC.

6.6 Le tissage d'aspects

Dans le Chapitre 4 nous avons vu qu'une architecture FRACTAL, du fait de la notion de composant partagé, ne correspond pas au parcours d'un arbre (modèle hiérarchique), mais plutôt de celui d'un graphe direct acyclique. FPATH permet la navigation dans une application FRACTAL représentée comme un graphe. Les composants, interfaces et opérations sont modélisés sous forme de nœuds reliés par des axes. Dans l'implantation de l'interface de tissage nous utilisons directement des requêtes FPATH pour obtenir les opérations correspondant à une expression de coupe. Nous avons étendu FPATH pour le support d'expressions régulières. Les nœuds parcourus sont ainsi comparés avec les éléments de l'expression de coupe. Par introspection, les interfaces de tissage contrôlant ces opérations sont récupérées et invoquées pour établir les liaisons d'aspects.

Il existe ainsi, deux façons de tisser un aspect avec FAC : établir les liaisons d'aspect manuellement, en appelant l'interface de tissage de chaque composant aspectisable, ou faire un tissage global, à l'aide d'une expression de coupe, transformée en requête FPATH. Dans ce dernier cas, l'opération s'effectue également depuis une interface de tissage. Cependant un paramètre supplémentaire doit être donné : le nom d'un composant dit *racine* qui est le point de départ de la requête FPATH.

Le Listing 6.6 (voir Annexe A pour le code complet de l'interface de tissage) présente les deux opérations liées à ce processus. Elles permettent de tisser ou d'annuler le tissage d'un aspect sur un ensemble de composants. Une coupe est donnée en paramètre — l'expression qui sera traduite en requête FPATH—, ainsi qu'un composant dit *racine*, le point de départ de la recherche. Tisser un aspect revient alors à établir un ensemble de liaisons d'aspect. Un dernier paramètre permet de spécifier un nom pour le domaine d'aspect associé au tissage. Ce domaine d'aspect sera créé automatiquement. L'opération de tissage devient donc une opération de recon-

figuration architecturale à part entière : parcours de l'architecture, création de liaisons d'aspect, création d'un composite pour le domaine d'aspect, et ajout des composants aspectisables dans ce composite. Pour réaliser toutes ces tâches nous avons défini l'opération de tissage d'aspect de l'interface de tissage comme un script FSCRIPT. Nous avons déjà vu précédemment que nous utilisons FPATH pour trouver tous les points de jonction qui correspondent à la coupe, nous utilisons en fait complètement FSCRIPT pour à la fois repérer les points de jonction et réaliser les tâches de reconfiguration architecturale.

```
void weave(Component root, Pointcut pcut, Component aspect, String aDomain);
void unweave(Component aspect);
```

Listing 6.6 – Interface de tissage : tissage d'aspect avec domaine d'aspect

En bref le tissage d'un aspect peut être résumé par les étapes suivantes :

- récupérer la référence du composant racine,
- créer le domaine d'aspect (composite FRACTAL) si il n'existe pas, sinon récupérer sa référence (requête FPATH),
- ajouter le composant d'aspect au domaine d'aspect,
- lancer une requête FPATH sur le composant racine donnant la liste des composants aspectisables concernés par la coupe,
- sur chaque élément de cette liste :
  - vérifier qu'il s'agit bien d'un composant aspectisable (doté de l'interface de tissage),
  - appeler sur l'interface de tissage le tissage des opérations concernés par la coupe (nouvelle requête FPATH),
  - positionner chacune de ces opérations comme aspectisée par le composant d'aspect,
  - ajouter le composant aspectisable au domaine d'aspect,
  - établir la liaison d'aspect vers le composant d'aspect (liaisons alors possible car les composants font tous partie du domaine d'aspect),
- notifier de la réussite ou de l'échec du tissage (par exemple aucune opération ne correspond à la coupe).

6.6.1 Le tissage avec FRACTAL-ADL

FRACTAL-ADL est le langage d'architecture de FRACTAL (voir Section 4.2.2 p. 49). Le langage étant aspectisable et extensible, nous avons ajouté les concepts de FAC à l'architecture de l'outil, en particulier la notion de liaison d'aspect et de tissage. La définition d'un composant d'aspect ne demande rien de particulier, car un composant d'aspect est un composant FRACTAL simplement doté d'une interface serveur : l'interface de conseil. FAC introduit donc deux modules dans FRACTAL-ADL pour la liaison aspect et le tissage d'aspect. Ces deux modules sont associés à deux nouvelles balises XML dans le langage (voir DTD Figure 6.11).

6.6.2 Le tissage avec FRACTAL EXPLORER

FRACTAL EXPLORER permet également de définir une opération de tissage et a été étendu pour un tel support. En particulier le contrôle de tissage est visible dans la console. Sur la Figure 6.12, on peut voir l'interface de tissage apparaître au milieu des autres interfaces de contrôle. Dans la partie droite de la figure, des informations sont données sur les tissages réalisés sur le composant courant. Ainsi, la partie supérieure liste les composants d'aspect tissés sur ce composant. Dans la partie basse sont représentées toutes les opérations du composant, et pour chaque opération, les composants d'aspect tissés ainsi que leur ordre.

```
<!ELEMENT aspectBinding (comment*) >
<!ATTLIST aspectBinding
  pointcutExp CDATA #REQUIRED
  ac CDATA #REQUIRED
>

<!ELEMENT weave (comment*) >
<!ATTLIST weave
  root CDATA #REQUIRED
  ac CDATA #REQUIRED
  pointcutExp CDATA #REQUIRED
  aDomain CDATA #REQUIRED
>
```

FIG. 6.11 – FRACTAL-ADL : DTD des balises de liaison d'aspect et de tissage

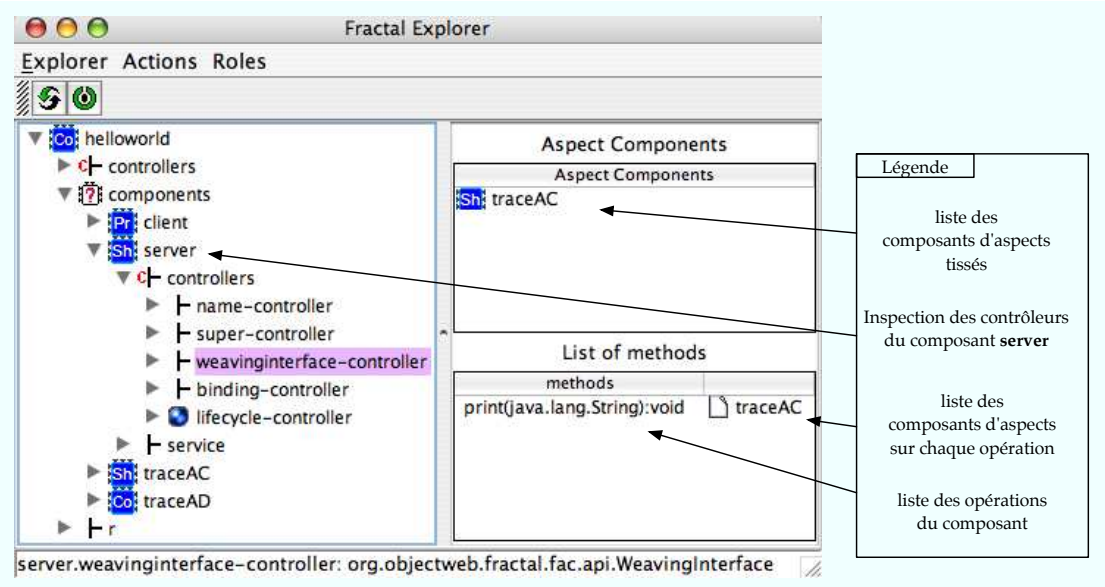

FIG. 6.12 – Support du tissage avec FRACTAL EXPLORER.

6.7 Bilan des concepts aspects introduits par FAC

Le tableau 6.1 résume les concepts ajoutés à FRACTAL par FAC. Une première observation importante est que tous les concepts ajoutés à FRACTAL sont des extensions du modèle comme celui-ci l'autorise : un nouveau type de liaison et une nouvelle interface de contrôle. L'ensemble des interfaces de contrôle en FRACTAL est aspectisable. La liaison d'aspect est supportée par l'interface de tissage, tout comme la liaison standard est supportée par le contrôle de liaison. En ce qui concerne l'interface de conseil, le composant d'aspect et le domaine d'aspect, nous voyons que les concepts d'interface serveur, composant et composite sont utilisés tels quels.

Concept FAC	**Utilise le concept FRACTAL de**
Liaison d'aspect	Liaison
Interface tissage	Interface de contrôle
Interface de conseil	Interface serveur
Composant d'aspect	Composant
Domaine d'aspect	Composite

TAB. 6.1 – Bilan des concepts de FAC.

Concept FAC	**Concept aspect correspondant**
Liaison d'aspect	Pas d'équivalent
Interface tissage	Tisseur
Interface de conseil	Advice
Composant d'aspect	Aspect
Domaine d'aspect	Pas d'équivalent

TAB. 6.2 – Comparaison des concepts de FAC aux concepts aspects.

Le tableau 6.2 propose un bilan sur les concepts de FAC vis-à-vis des concepts de l'approche par aspects. Nous pouvons voir que certains de ces concepts, n'ont pas d'équivalent, tel que la liaison d'aspect et le domaine d'aspect. Ces deux concepts sont, cependant, de notre point de vue très important pour donner une visibilité accrue des effets d'aspects dans un système. De plus cela permet d'unifier composants et aspects, de les doter des même pouvoir de composition et de représentation architecturale.

A travers ces deux tableaux, nous voyons que tout le travail d'intégration de préoccupations transverses se réalise en choisissant parfois des concepts issues de l'approche à composants, parfois de l'approche par aspects, enrichissant ainsi les deux approches, tout en réduisant leurs différences.

6.8 Synthèse

Nous avons introduit dans ce chapitre les concepts principaux de FAC. Nous avons validé notre objectif d'unifier l'approche par aspects et l'approche à composants, e proposant de ne faire aucune distinction entre la partie comportementale d'un aspect et un composant. Seule la composition d'un aspect avec des composants requiert un nouveau mécanisme pour le support de tissage d'aspects : la liaison d'aspect. Nous avons vu que les mécanismes autour de la liaison d'aspect sont plus complexes qu'une simple liaison FRACTAL. Ceci n'est pas surprenant, l'approche par aspects introduit un certain nombre de concepts pour l'intégration de préoccupations transverses. Les mécanismes utilisés sont à l'intersection de nombreuses disciplines comme la méta-programmation, la théorie des graphes ou de transformation de programme. Du fait de

l'unification réalisée par FAC nous utilisons les concepts et outils sous-jacents de l'approche à composants. Ainsi par moment les concepts constituant FAC sont inspiré de l'approche à composants (notion de liaison d'aspect, inédit par rapport à l'approche par aspects) et par moment de l'approche par aspects (introduction d'interfaces orientées aspect, comme l'interface de tissage ou de conseil).

Ce chapitre a également permis de dresser une comparaison poussée entre les concepts habituellement rencontré dans l'approche par aspects et la façon dont nous les intégrons dans l'approche à composants. Ainsi FAC contribue sur un certain nombre de faiblesses reconnues à l'approche par aspects. En introduction de ce chapitre nous nous fixions de renforcer la structure des aspects grâce à l'emploi de composants, d'améliorer la visibilité de l'action des aspects dans un système à composants, de rendre plus sûre l'utilisation des aspects. Nous avons mis en place ces trois objectifs de la manière suivante :

- les aspects sont des composants FRACTAL, bénéficiant ainsi de toutes les propriétés structurantes de ces derniers,
- la notion de liaison d'aspect, de domaine d'aspect, et les capacité d'introspection de coupe améliore grandement la visibilité sur les aspects agissant dans un système,
- la sûreté de support peut être reliée à la visibilité dans le sens où, par exemple, les capacité d'introspection permettent de tester une coupe avant de la rendre active, ou encore de vérifier l'ensemble des composants impactés par une coupe donnée. Les notions de liaison d'aspect et de domaine d'aspect concourent également à ce support sûr.

Le chapitre suivant s'étend sur les concepts avancés de FAC. En particulier le chapitre insiste sur la validation des objectifs de modèle homogène et support des propriétés transverses en donnant la méthodologie de FAC. Nous revenons également plus en détail sur cette objectif de support sûr des aspects par les composants.

Chapitre 7

Concepts avancés de FAC

Sommaire

Ce chapitre présente les concepts avancés de Fractal Aspect Component (FAC). Dans le chapitre précédent, nous avons présenté les notions de base qui ont été ajoutées au modèle FRACTAL pour le support d'aspects. Nous avons vu que FAC respecte l'objectif d'unification et que certaines notions améliorent même l'approche par aspects en termes de visibilité des coupes (introspection de coupe et liaison d'aspect). Nous nous intéressons à présent aux deux autres objectifs établis dans le Chapitre 5, à savoir :

- La définition d'un modèle homogène où aucune distinction n'est faite entre propriétés fonctionnelles ou techniques, qui sont toutes représentées au même niveau à l'aide de composants,
- Le support pour la séparation des préoccupations non couvert par le modèle FRACTAL.

Pour répondre à ces deux objectifs, nous présentons un certain nombre de concepts avancés de FAC. La Section 7.1 montre comment des coupes comportementales peuvent être définies grâce aux coupes sur les traces d'exécution. Ces coupes permettent entre autres de capturer des comportements résultant d'une interaction entre plusieurs composants. La Section 7.2 développe notre objectif de support sûr des aspects avec FAC. Par support sûr nous entendons préserver l'encapsulation des composants malgré le côté envahissant des aspects. Nous regardons comment FAC permet un contrôle fin de l'accès des aspects aux composants aspectisables.

Enfin, la Section 7.3 spécifie la méthodologie de l'utilisation de FAC, et permet de mettre au point nos objectifs de modèle homogène et de support des préoccupations transverse. Nous regardons comment les propriétés transverses peuvent être modularisées et isolées une à une, qu'elle soit d'ordre technique ou fonctionnelle, et quelle que soit sa granularité.

7.1 Coupes sur les traces d'exécution

Les **coupes sur les traces d'exécution** permettent de capturer des comportements plus complexe dans un système. Elles définissent une **séquence de points de jonction passés**, c'est-à-dire sur un historique d'événements, d'enchaînement d'occurrences dans un système. Le langage de coupe que nous avons introduit précédemment permet de décrire, structurellement, les points de jonction à capturer et à relier à un aspect (opération de tissage de cet aspect). Cependant, il peut parfois être intéressant de définir une coupe plus comportementale, décrivant le comportement d'une architecture, plutôt que sa simple structure. Nous définissons dans FAC des coupes sur les traces d'exécution — l'historique — des composants. Le principe est le suivant. Plutôt que d'identifier et de regrouper des points de jonction éparses et correspondant à un événement unique, isolé — dans FAC, il s'agit une invocation d'opération de composant —, une coupe spécifie une **séquence de points de jonction**. Autrement dit, la coupe ne signifie plus : exécuter le code aspect lorsque tel point **ou** tel autre point de jonction est exécuté ; mais déclencher le code aspect lorsque tel point **puis** tel autre point de jonction est exécuté. La séquence étant vérifiée à l'exécution, l'historique ne porte donc que sur **les événements passés.**

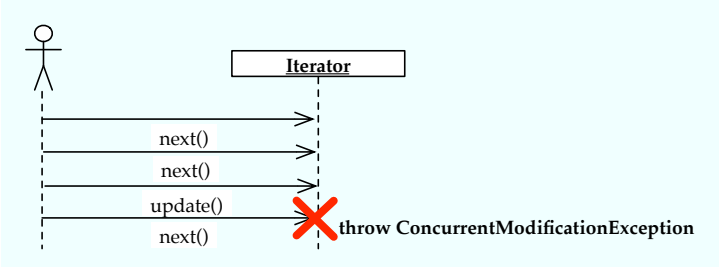

FIG. 7.1 – Exemple du patron d'itération.

Le principe des coupes sur les traces d'exécution n'est pas nouveau. Il porte le nom de *tracematches pointcut* dans la communauté aspect. La première approche à l'avoir intégré est EAOP (*Event AOP*) [Douence et al., 2003], qui repose entièrement sur le principe d'événements. Ensuite le principe a été reproduit et intégré à JAsCo (*stateful aspects*) [Vanderperren et al., 2005], puis à AspectJ (équipe ABC) [Allan et al., 2005]. Un certain nombre d'exemples concrets d'utilisation de telles coupes est donné dans les travaux de [Allan et al., 2005], comme l'implantation d'une méthode de sauvegarde automatique qui cherche à enregistrer des données toutes les cinq opérations exécutées (voir Figure 7.1), ou encore la possibilité de réaliser une implantation plus sûre du patron d'itération. Dans ce dernier exemple, l'ordre d'exécution des opérations peut être contrôlé prévenant ainsi la modification de données pendant l'exécution de l'itération.

FIG. 7.2 – Coupes sur les traces d'exécution : un premier exemple.

```
1 tracematchespcd::=tracematchespcd .tracematchespcd
2                  | (tracematchespcd)
3                  | tracematchespcd+tracematchespcd
4                  | tracematchespcd
5                  | #
6  #::= désigne une étoile : 0 à n transitions quelconques
7  pcd::= jp_type  component;interface;method
8  jp_type::= CLIENT | SERVER | BOTH
9  component::= une expression régulière sur le nom des composants
10 interface::= une expression régulière sur le nom des interfaces
11 method::= une expression régulière sur le nom des opérations
```

Listing 7.1 – Grammaire du langage de coupe *tracematches* de FAC

Avec FAC, les coupes sur les traces d'exécution délimitent un chemin d'exécution dans une architecture et se déclarent par une liste de coupes structurelles, séparées par des points virgules (voir la grammaire du Listing 7.1). Par exemple, la coupe `A.m1*;B.m2*` signifie que le code d'un aspect est exécuté lorsque une opération `m1` est appelée sur le composant `A` suivie par une opération `m2` du composant `B` (voir illustration de la Figure 7.2).

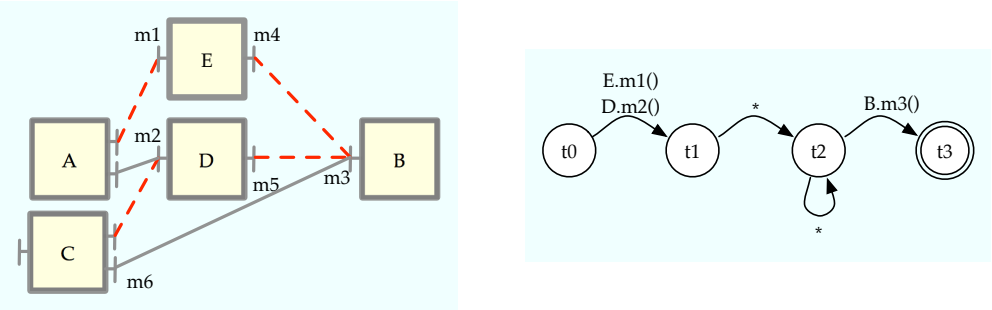

FIG. 7.3 – Coupes sur les traces d'exécution : un second exemple.

Pour la formalisation de notre langage de coupe, nous nous reposons sur la famille des langages d'automates [Lynch and Tuttle, 1989]. Ainsi, une coupe sur les traces d'exécution produit un automate à états finis où les transitions correspondent aux occurrences d'événements dans l'architecture. Les événements pris en compte sont les appels entrants ou sortants d'un composant. Par exemple, la coupe définie précédemment produit l'automate représenté sur la Figure 7.2. En utilisant le langage d'automate, il est possible de définir des coupes plus élaborées comme la coupe `A.m1()+A.m2();*;B.m3()`, qui produit l'automate présenté sur la Figure 7.3. La figure fournit également un exemple d'architecture avec les chemins qui sont capturés par la coupe (représentés en pointillés).

7.1.1 Implantation du mécanisme des coupes sur les traces d'exécution à l'aide de composants d'aspect

Un autre exemple d'utilisation des composants d'aspect est une implantation possible du mécanisme des coupes sur les traces d'exécution. Nous avons vu, dans la Section 7.1, que les coupes sur les traces d'exécution reposent sur le principe d'historique de points de jonction, et que cet historique peut être ramené à un automate. Dans notre exemple, le composant d'aspect est tissé sur l'ensemble des points de jonction d'une architecture donnée, et implante l'automate de la coupe de type trace d'exécution.

Comme le composant d'aspect est tissé sur tous les points de jonction, il est possible d'analyser l'évolution du système et de vérifier son historique. Par conséquent, le composant implante l'automate de la coupe sur les traces d'exécution et navigue dans l'automate pour savoir à quel

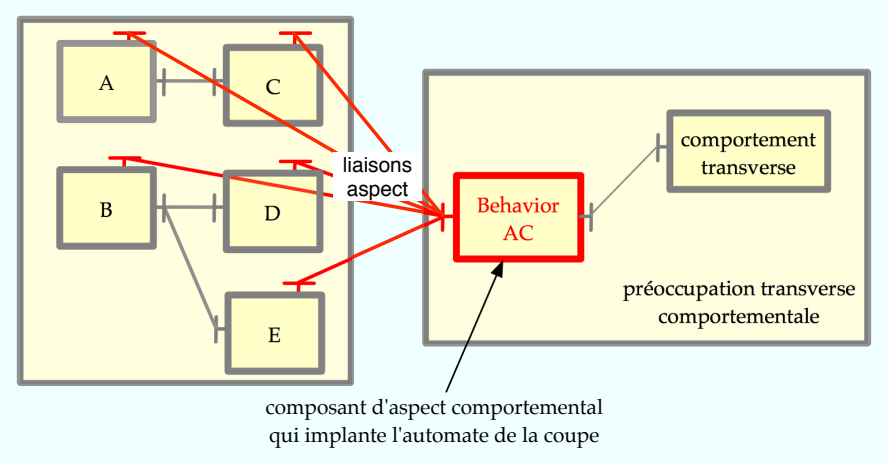

FIG. 7.4 – Composant d'aspect : exemple d'implantation des coupes *tracematches*.

moment exécuter le comportement transverse. La Figure 7.4 fournit une illustration de ce mécanisme. On observe que le composant d'aspect implantant l'automate est tissé sur tous les composants de l'architecture. Lorsque l'état final de l'automate est atteint, le composant est appelé.

En résumé, les coupes sur les traces d'exécution permettent de définir des coupes plus comportementales, c'est-à-dire capable de capturer un ensemble d'interaction entre composants. Elles vont plus loin que les coupes strictement structurelles et caractérisant un seul événement. Grâce à ce genre de mécanisme, FAC permet d'élaborer des liaisons d'aspect capturant des séquences d'événements, qui auraient requis le tissage de nombreux aspects pour arriver au même résultat. Les coupes sur les traces d'exécution permettent donc d'automatiser ce mécanisme. Elles augmentent le pouvoir d'expression des coupes FAC.

7.2 Sûreté du support des aspects : les modules ouverts et FAC

Nous avons vu dans les chapitres précédents que l'approche à composants souffre d'un manque pour le support des propriétés transverses. Notre proposition adresse cette limitation en unifiant les approches à composants et par aspects. Cependant, nous éclaircissons ici un paradoxe émergeant d'une contradiction entre l'encapsulation forte de l'approche à composants, d'un côté, et le côté envahissant de l'approche par aspects, de l'autre. En effet, l'approche par aspects fournit des mécanismes puissants pour la séparation des préoccupations, mais plusieurs de ces constructions violent l'encapsulation en créant des dépendances entre les préoccupations, rendant l'évolution parfois difficile et sujette à erreurs. Si cela est déjà problématique lorsque l'approche par aspects s'applique sur des objets, le pouvoir d'encapsulation dans l'approche à composants est encore plus fort.

Pour palier à ce phénomène, un système ouvert de modules (*Open Modules*) a été proposé par [Aldrich, 2005] s'appliquant sur l'approche à objets. L'idée des modules ouverts est d'ouvrir un programme à l'approche par aspects tout en gardant la modularité en cachant les détails d'implantation d'un module. Un module est défini comme un ensemble d'entités qui partagent des points d'accès qui sont des points de jonction exportés par le module. L'utilisation de ce système de modules permet de préserver le contenu d'un module en déclarant explicitement les points de variations sur lesquels les aspects peuvent agir.

Les similitudes entre ce système de modules et l'approche à composants sont nombreuses et nous proposons dans cette section une comparaison. En particulier, nous élevons les principes des modules au niveau composant. A travers cette étude, nous montrons comment le tissage d'aspects peut être supporté de manière sûre par les composants. Nous commençons par présenter les propriétés des modules ouverts, puis nous comparons ces propriétés à celles de FAC.

7.2.1 Principes des modules ouverts

L'objectif des modules ouverts est de limiter l'accès aux points de jonction d'un système, qui sont atteint par les aspects rendant ces derniers envahissants, c'est-à-dire capable d'accéder à des attributs privés et de briser l'encapsulation des objets. Un module est un regroupement de classes associé à une définition de points d'accès. Cette définition est l'interface du module. Un point de jonction devient accessible lorsqu'il est déclaré comme point d'accès d'un module. L'interface d'un module peut donc déclarer : des points de jonction, des fonctions, ou des coupes pré-définies sur un ensemble d'éléments internes. Un module vérifie les propriétés définies ci-dessous, que nous établissons comme règles pour pouvoir s'y référer plus facilement dans la suite de notre discussion.

- ***Règle 1*** *Des aspects extérieurs à un module peuvent interagir entre ce module et le monde extérieur — incluant des appels extérieurs à des fonctions de l'interface d'un module.*
- ***Règle 2*** *Des aspects extérieurs peuvent aussi être tissés sur l'interface d'un module.*
- ***Règle 3*** *Les modules externes ne peuvent être tissés directement sur les événements internes d'un module, comme par exemple, les appels depuis un module à d'autres fonctions du module (d'une autre classe), même si ces fonctions sont exportées.*

Dans des travaux plus récents, les principes gouvernant les modules ouverts ont été appliqués à AspectJ [Ongkingco et al., 2006]. Dans leur étude, les auteurs ont ajoutés de nouveaux concepts, qui sont souvent assez spécifique à AspectJ. Cependant, nous en avons relevé deux qui nous semblent généralisables à notre application des modules ouverts aux composants. Le premier concept intéressant est la possibilité de ne pas seulement réduire et contrainte l'accès à un module, mais de pouvoir moduler cet accès : pouvoir ouvrir et fermer l'accès de certains points. Ceci peut être particulièrement utile dans un scenario d'aspect de déverminage. Il peut ainsi être intéressant de fermer certains accès après cette période de déverminage. Un second concept intéressant de cette étude est la possibilité de désigner les aspects pouvant accéder l'interface du module. Avec AspectJ, cette opération consiste à donner une expression régulière sur les noms de paquetage des aspects autorisés. Encore une fois dans notre exemple de déverminage nous pouvons imaginer spécifier tous les aspects du paquetage `debug.*`.

- ***Règle 4*** *Un module peut désigner le ou les aspect(s) qui peuvent accéder son interface.*
- ***Règle 5*** *Un module peut ouvrir ou réduire l'accès à ses points de jonction.*

7.2.2 Supports des règles dans FAC

La première similitude frappante réside dans la notion de module et celle de composant. Un module est une collection de classes qui partagent des points d'accès aux aspects par une interface. Un composant est une entité contractuelle qui fournit et requiert des services par le biais d'interfaces. Un composant est une «boîte noire» qui naturellement cache ses détails d'implantation comme le définit la *Règle 3* des modules ouverts. Dans FAC les points de jonction sont les opérations des interfaces externes (fournies et requises) des composants ouverts. Cette définition

s'applique parfaitement aux définitions des *Règles 1, 2* et *3*. La *Règle 2* correspond à la définition même de notre modèle de point de jonction, les points d'accès sont accessible par les aspects, comme les interfaces clientes ou serveurs des composants aspectisables FAC. La *Règle 1* précise que les aspects peuvent intercepter les appels sortants d'un module, comme c'est le cas en FAC sur les interfaces clientes. La *Règle 3* interdit l'interception des appels internes, ce qui est le cas également avec FAC.

Malgré cette préservation dans FAC du comportement interne des composants, il est important de remarquer que le fait qu'un aspect intercepte les appels entrants ou sortants d'un composant aspectisable peut modifier le comportement attendu du composant. En ce sens, il parait normal de pouvoir contrôler l'accès à ces points de jonction. C'est ce que FAC fournit avec son interface de tissage et le verrouillage de l'accès aux opérations.

La seconde similitude importante vient de la *Règle 4* définie pour les besoins d'AspectJ mais qui peut aussi s'appliquer à notre modèle. Cette règle précise qu'il est possible de désigner les aspects ayant accès à l'interface du module. Dans FAC, chaque composant aspectisable par l'intermédiaire de son interface de tissage gère les composants d'aspects s'appliquant sur ses interfaces. En ce sens, un composant aspectisable choisit ses préoccupations transverses, en établissant ses liaisons d'aspects. Nous pouvons donc considérer que FAC et les modules ouverts se rapprochent sur cette quatrième règle.

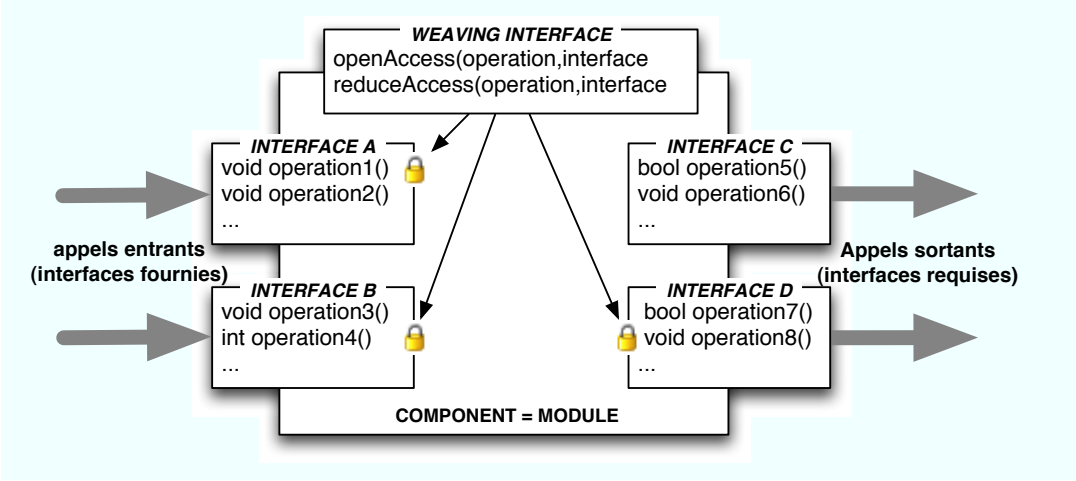

FIG. 7.5 – Contrôle de l'accès aux points de jonction dans FAC.

La troisième similitude concerne la *Règle 5*. Celle-ci établit que les modules ouverts ne doivent pas être limités à la seule réduction d'accès aux points de jonction mais aussi à l'ouverture. Autrement dit, cette règle impose un système plus souple où l'ouverture fait preuve de variabilité, se configure. Cette idée va dans le sens de FRACTAL et de la définition d'un composant comme une entité avec des capacités ouvertes de contrôle, donc variable. Ainsi, comme nous l'avons vu dans notre définition de l'interface de tissage dans la Section 6.4 p. 73, l'interface de tissage permet d'ouvrir ou fermer l'accès des aspects aux points de jonction d'un composant, — c'est-à-dire l'accès aux opérations de ses interfaces externes —. La Figure 7.5 fournit une vue conceptuelle du rôle de l'interface de tissage dans le verrouillage de l'accès aux opérations par les aspects. L'interface peut ainsi empêcher le tissage d'aspects lorsque ces derniers ne sont pas autorisés. Le Listing 7.2 présente les opérations de verrouillage des opérations d'un composant aspectisable en FAC, qui montre qu'il est possible de définir des niveaux d'ouverture `OpennessLevel` sur les opérations.

```
1 void openAccessTo(Operation op, Interface itf, OpennessLevel lvl);
2 void reduceAccess(Operation op, Interface itf);
```

Listing 7.2 – Interface de tissage : contrôle d'accès aux opérations

Cette étude comparative avec les modules ouverts a montré que le support d'aspect peut être maîtrisé dans le cadre de l'intégration dans un modèle de composant. Avec FAC, nous fournissons un cadre sûr de supports du tissage de préoccupations transverses. Cet étude sur les modules ouverts conclut également notre section sur les concepts avancés de FAC. Nous étudions à présent la méthodologie de FAC pour l'identification et le tissage d'aspects.

7.3 Méthodologie de FAC

Nous définissons à présent la méthodologie de FAC. Dans cette méthodologie, nous montrons comment construire une application utilisant harmonieusement des composants et des aspects de manière unifiée. Nous expliquons comment une préoccupation conçue avec des composants est intégrée de manière classique avec FRACTAL par des liaisons ou alors grâce aux mécanismes aspects de FAC. Nous répondons donc à l'objectif de support des préoccupations transverses fixé au Chapitre 5, et nous justifions, à travers un exemple de service technique, l'objectif de modèle homogène.

Pour se faire, nous nous appuyons sur un exemple que nous avons déjà abordé dans le chapitre précédent et ce chapitre : Comanche et la journalisation. Pour rappel, Comanche est un serveur HTTP minimal conçu à l'aide de composants FRACTAL, qui consiste à accepter des connexions TCP et à traiter ces requêtes par l'envoi du fichier demandé ou d'une erreur.

Dans la Section 5.1 p. 60, nous mettions en exergue les limitations de FRACTAL pour la modularisation des préoccupations transverses comme la journalisation. Nous concluions qu'il manquait un mécanisme plus riche de composition pour pouvoir connecter le composant de journalisation avec le reste de l'architecture. Nous allons à présent étudier l'application des principes de FAC sur cet exemple de Comanche.

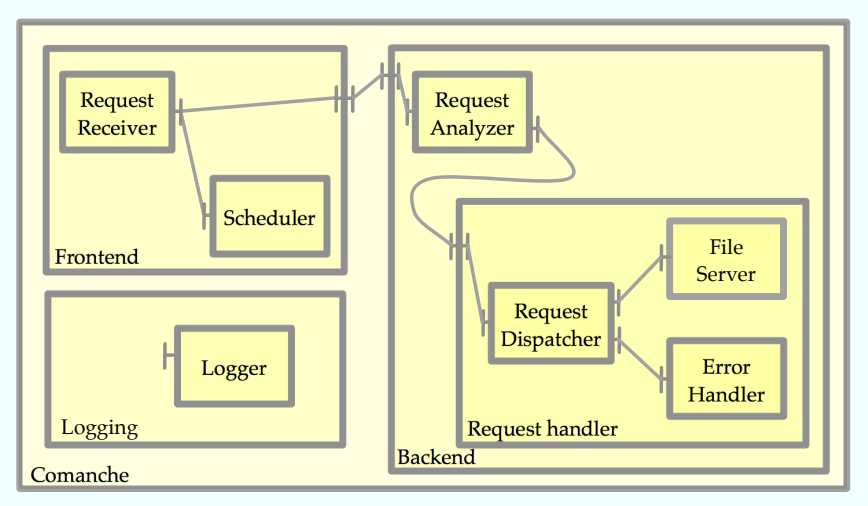

FIG. 7.6 – Comanche : la journalisation

La **méthodologie** que nous mettons en place repose sur le fait que les concepts de FAC doivent être utilisés dans un certain ordre et d'une certaine manière. La procédure est la suivante,

pour toute propriété transverse :

- **Etape 1** : Définition de la propriété transverse : il s'agit simplement de définir un assemblage de composants caractérisant la propriété à tisser (partie comportementale de l'aspect),
- **Etape 2** : Définition du ou des composants d'aspects qui vont permettre d'intégrer la propriété transversalement au reste du système,
- **Etape 3** : Tissage des composants d'aspects à l'aide de FRACTAL-ADL, FRACTAL EXPLORER, ou encore par appel de l'interface de programmation de l'interface de tissage.

La Figure 7.6 fournit une illustration conceptuelle de ce qui est attendu pour cet exemple. On observe que la préoccupation de journalisation est à présent isolée dans un composite *Logging* de l'analyseur de requête et de la gestion de l'expédition des requêtes (*dispatcher*). Ces deux composants ne peuvent être dotés d'une interface requise vers le composant de journalisation, car cette propriété sort de leur fonctionnalité première.

Avec FAC nous allons pouvoir connecter le composant de journalisation à ces deux composants tout en respectant la propriété de non-conscience et sans introduire de mélange. Regardons comment utiliser les concepts FAC et dans quel ordre les opérations sont effectuées.

7.3.1 Etape 1 : définition de la propriété transverse

La première étape de notre méthodologie consiste à définir la préoccupation transverse. Comme nous l'avons expliqué au chapitre précédent, FAC est une approche unifiée, ce qui signifie qu'une propriété est conçue de la même manière, qu'elle soit transverse ou non. Dans notre exemple de la journalisation, la fonctionnalité reste donc identique. Le composant *Logger* de Comanche est pris tel quel. Le Listing 7.3 présente le code du composant *BasicLogger* et de son interface *Logger*.

```
1 /** Interface de journalisation */
2@Interface(name="logger")
3public interface ILogger {  void log (String msg); }
4
5 /** Composant de journalisation sur la sortie standard */
6 @FractalComponent
7public class Logger implements Logger {
8  public void log (String msg) { System.out.println(msg); }
9}
```

Listing 7.3 – Composant *Logger*

7.3.2 Etape 2 : définition du/des composant(s) d'aspect

La seconde étape consiste à définir un connecteur pour adapter le composant *Logger* à une utilisation transverse. Autrement dit, un composant d'aspect doit être défini pour capturer le contexte d'interception des composants sur lequel la journalisation va s'appliquer. Dans la Section 6.2.2 p. 70, le concept de composant d'aspect a été décrit comme une matérialisation d'une bonne pratique dans les approches par aspects, qui consiste à déléguer l'implantation de la préoccupation transverse. Ainsi, notre composant d'aspect de journalisation ici se contente d'être connecté, par une liaison standard, au composant *Logger*.

7.3.3 Etape 3 : tissage du/des composant(s) d'aspect

Dernière étape de notre méthodologie : l'intégration de la préoccupation. Pour ce faire, nous tissons le composant d'aspect sur les composants *Request Analyzer* et *Request Dispatcher*. Dans la

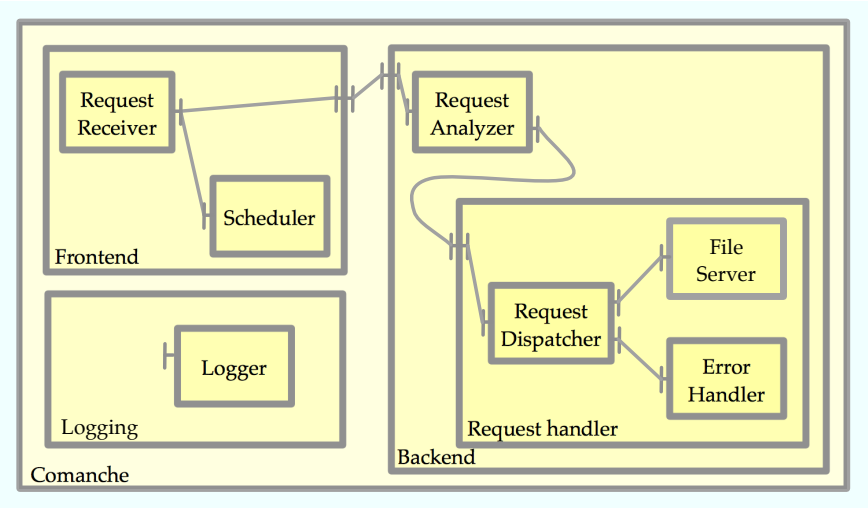

FIG. 7.7 – Comanche : la journalisation et le composite *Logging*

Section **??** p. **??**, nous avons défini les différentes étapes du tissage et vu qu'il était nécessaire de préciser une coupe, un nom pour le domaine d'aspect, ainsi qu'un composant «racine» qui sera parcouru pour retrouver les points de jonction spécifiés dans la coupe. Pour notre exemple la coupe `request*;rh;*`, qui capture les composants dont le nom commence par *request* et possédant une interface *rh*, permet de bien capturer nos deux composants.

Trois solutions s'offrent à nous pour pouvoir tisser le composant d'aspect *LoggerAC*. La première solution consiste à appeler directement l'interface de tissage. La seconde solution consiste à déclarer le tissage au niveau de l'architecture de l'application grâce à FRACTAL-ADL. La troisième et dernière solution consiste à utiliser la console d'administration FRACTAL EXPLORER. L'opération de tissage est alors vue comme une opération d'administration. Nous détaillons à présent le tissage grâce à FRACTAL-ADL, puis grâce à FRACTAL EXPLORER

Tissage avec FRACTAL-ADL A titre d'exemple, le Listing 7.4 est la description de l'assemblage de l'exemple Comanche de la Figure 7.6. L'assemblage du composite *Logging* de la Figure 7.7 est présenté sur le Listing 7.5. Si nous ajoutons un composant d'aspect pour y connecter un composant de journalisation, l'architecture devient celle de la Figure 7.8. La description de ce nouvel assemblage requiert la déclaration d'une opération de tissage Ligne 9 du Listing 7.6.

```
1<definition name="comanche.Comanche" >
2  <interface name="r" signature="java.lang.Runnable" role="server"
3              cardinality="singleton" contingency="mandatory"/>
4  <component name="frontend" definition="comanche.FrontendComposite" />
5  <component name="backend" definition="comanche.BackendComposite" />
6  <component name="logging" definition="comanche.LoggingComposite" />
7  <binding client="this.r" server="frontend.r" />
8  <binding client="frontend.rh" server="backend.rh" />
9</definition>
```

Listing 7.4 – Comanche : l'assemblage avec FRACTAL-ADL

```
1<definition name="comanche.LoggingComposite" >
2  <component name="logger">
3      <interface name="logger" signature="comanche.ILogger" role="server"/>
4  </component>
5  <component name="loggingAC">
6      <interface name="aspectComponent" signature="fac.AdviceInterface" role="server" />
7      <interface name="logger" signature="comanche.ILogger" role="client"/>
8  </component>
```

```
 9  <binding client="loggingAC.logger" server="logger.logger" />
10</definition>
```

Listing 7.5 – Assemblage du composite *Logging* avec FRACTAL-ADL

```
 1<definition name="comanche.Comanche" >
 2  <interface name="r" signature="java.lang.Runnable" role="server"
 3               cardinality="singleton" contingency="mandatory"/>
 4  <component name="frontend" definition="comanche.FrontendComposite" />
 5  <component name="backend" definition="comanche.BackendComposite" />
 6  <component name="logging" definition="comanche.LoggingComposite" />
 7  <binding client="this.r" server="frontend.r" />
 8  <binding client="frontend.rh" server="backend.rh" />
 9  <weave root="this" ac="loggerAC" pointcutExp="request*;rh;*"/>
10</definition>
```

Listing 7.6 – Comanche : l'assemblage avec FRACTAL-ADL et tissage de *LoggingAC*

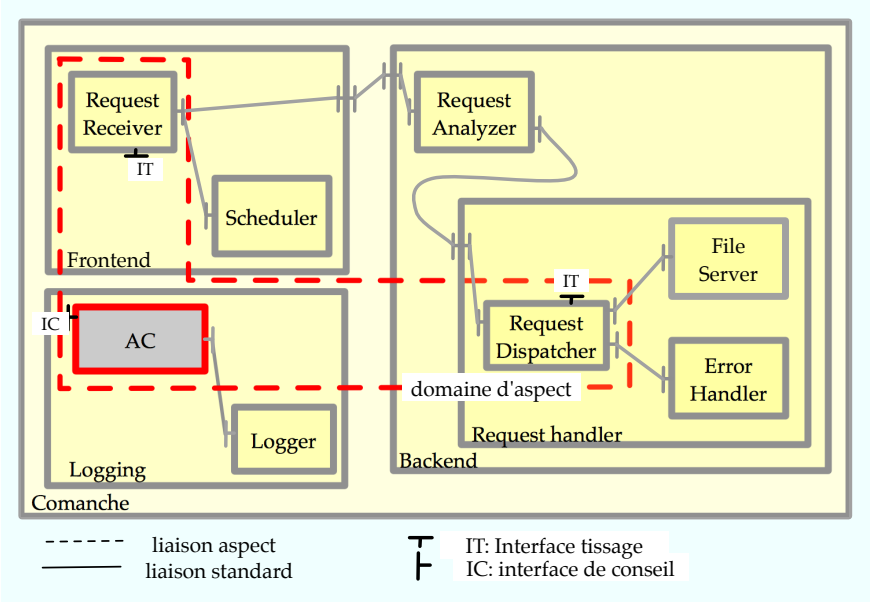

FIG. 7.8 – Comanche : la journalisation après tissage

Tissage avec FRACTAL EXPLORER Sur la Figure 7.9 on voit qu'il est possible de lancer une action de tissage depuis FRACTAL EXPLORER. La Figure 7.10 montre la fenêtre qui apparaît alors. Un composite «racine» doit être choisi, le composant d'aspect, et enfin une coupe. Le résultat final est l'architecture de la Figure 7.8.

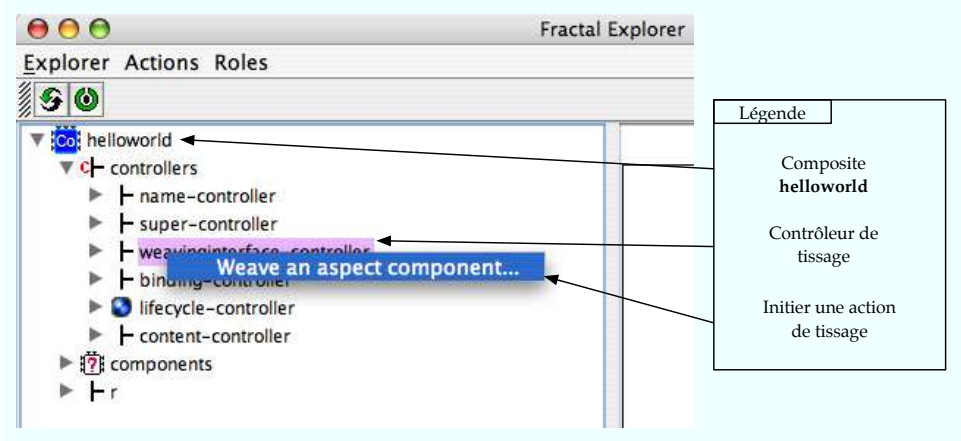

FIG. 7.9 – Support du tissage avec FRACTAL EXPLORER.

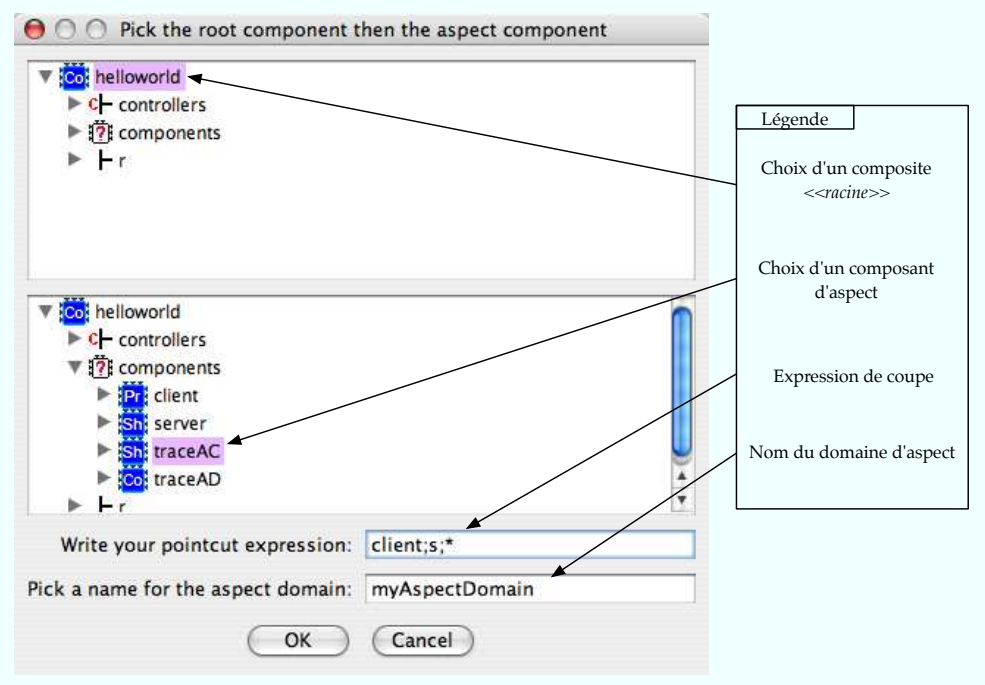

FIG. 7.10 – Support du tissage avec FRACTAL EXPLORER.

7.3.4 Bilan

En résumé, la méthodologie de FAC reste intuitive vis-à-vis de l'approche par aspect. Une fois les propriétés transverses identifiées, le comportement de la propriété est mis en œuvre avant sa liaison au reste de l'application. Du fait de la symétrie le comportement de toute propriété, transverse ou non, est implanté par des composants FRACTAL. Ensuite, le tissage permet d'établir les liaisons d'aspects vers les composants impliqué par la propriété transverse. Cette étape de tissage peut être réalisée de diverse manière et est une opération dynamique (à l'exécution).

Nous avons pu voir dans cette méthodologie que FAC est bien une approche unifiée, que le modèle est homogène (tout propriété est implantée par des composants). Nous remplissons donc bien tous les objectifs fixé dans le Chapitre 5.

7.4 Evaluation

	JBoss AOP	Spring AOP	CAM/DAOP	FuseJ	FAC
Critères du point de vue composant et architecture					
ADL	non	non	oui	oui*	oui
Général	non	non	oui	non	oui
Réflexif	non	non	non	non	oui
Extensible	non	oui	non	non	oui
Hiérarchique	non	non	non	oui**	oui
Critères du point de vue aspect					
Symétrie d'élément	non	oui	non	oui	oui
Symétrie de placement	oui	oui	oui	oui	oui
Symétrie de portée	non	non	non	non	oui
** Le langage de composition et de spécification de FuseJ*					
*** La hiérarchie est possible, mais chaque composite ne compose que deux composants à la fois.*					

TAB. 7.1 – Bilan des approches à composants et aspects avec FAC

Nous reprenons à présent les critères définis dans notre état de l'art et regardons comment FAC y répond. Le Tableau 7.1 résume ces critères et si les approches de l'état de l'art et FAC les supportent. Concernant les critères pour la partie composant, nous ne revenons pas dessus, nous avons vu que FRACTAL est un modèle général, réflexif, extensible et hiérarchique. Avec FRACTAL-ADL, FRACTAL est doté d'un langage d'architecture. Nous détaillons à présent les critères concernant la partie aspect : les critères de symétrie.

Symétrie des éléments Au niveau de la symétrie des éléments, FAC réalise comme FuseJ une unification. Ce qui veut dire qu'aucune distinction n'est effectuée entre la partie comportementale d'un aspect et un composant. FAC est donc bien une approche symétrique du point de vue des éléments.

Symétrie de placement Une approche est symétrique de placement lorsque la partie comportementale d'un aspect est détachée de la définition de la coupe. Avec FAC la partie comportementale des aspects sont des composants, et la partie coupe est définie soit dans la description

de l'architecture soit en manipulant l'interface de tissage d'un composant sur lequel l'aspect se tisse. Nous pouvons donc dire que ces deux parties sont clairement détachées dans FAC. FAC est bien symétrique du point de vue du placement.

Symétrie de portée Par symétrie de portée nous entendons que la composition de plusieurs aspects sur le même point de jonction est entièrement manipulable. Avec FAC la composition des aspects est gérée au niveau de chaque composant aspectisable avec l'interface de tissage. Cette interface de tissage est capable sur chaque point de jonction d'ordonnancer les aspects s'y appliquant de bloquer éventuellement l'aspect dans une perspective de sécurité. Ainsi les aspects sur un point de jonction particulier sont entièrement manipulables. FAC est donc bien symétrique du point de vue de la portée.

7.5 Synthèse

Nous avons présenté dans ce chapitre les concepts avancés de FAC. Dans un premier temps, nous nous sommes intéressé à la gestion des coupes sur les traces d'exécution. Ces coupes augmentent le pouvoir d'expression du langage de coupe de FAC en permettant de définir des coupes, qui capturent une interaction entre plusieurs composants, une séquence d'événements. Grâce à ce mécanisme, il est possible de définir des aspects agissant sur un comportement qu'il est possible d'identifier au sein d'une architecture.

Dans un second temps, nous avons validé notre objectif de gestion sûre du support du tissage des aspects. Nous avons pour cela réalisé une étude avancée du principe des modules ouverts. Nous avons montré comment FAC supporte les règles définis dans ce système, qui offre la possibilité de concilier encapsulation des modules d'une part et le côté envahissant des aspects d'autre part. Avec FAC, et en particulier, avec l'interface de tissage, il est possible d'avoir un contrôle total des points de jonction des composants, et ce contrôle est local à chaque composant.

Enfin, nous avons présenté la méthodologie de FAC qui, une fois la propriété transverse identifiée, repose sur trois étapes : la définition de la propriété transverse par des composants réutilisables, la définition du ou des composants d'aspects, et enfin le tissage de ces composants au reste de l'application. Nous avons illustré la méthodologie avec un exemple sur la journalisation. Cette étude valide l'objectif de support des préoccupations transverses, ainsi que notre modèle homogène car il s'agit d'une préoccupation technique. Pour rappel, dans un modèle homogène, seuls les services propres au fonctionnement du modèle à composants lui-même font partie de la membrane ; ceci pour pouvoir profiter du pouvoir de composition offert au niveau de base, celui du contenu des composants (par opposition à la membrane).

Les deux chapitres suivants proposent deux études de cas sur les transactions avancées et sur la communication de groupe. Dans ces exemples, nous montrons comment des aspects complexes peuvent être supporté par FAC, et comment la méthodologie de FAC peut être appliquée sur plusieurs niveaux pour décomposer, une à une, toutes les propriétés transverses.

Troisième partie

Etudes de cas

Chapitre 8

FAC et les transactions étendues

Sommaire

Ce chapitre présente une étude de cas autour des transactions étendues. Après une présentation du domaine, nous justifions l'utilisation d'un aspect pour modulariser un service implantant les transactions étendues. Nous montrons, dans un second temps, comment la conception de cet aspect conduit à un code entrelacé. Nous étudions ensuite la décomposition de cet aspect complexe avec FAC qui nous permet d'obtenir une séparation de toutes les préoccupations.

La Section 8.1 introduit le domaine des transactions étendues, en particulier les transactions imbriquées et les Sagas. Après une courte justification du ressort de l'approche par aspects pour la gestion des transactions étendues, la Section 8.2 précise la problématique abordée dans ce chapitre : l'entrelacement dans le code de l'aspect implantant les transactions étendues. La Section 8.3 montre comment nous résolvons cette problématique avec FAC. Enfin, la Section 8.4 évalue notre solution et la Section 8.5 conclut ce chapitre.

8.1 Introduction aux transactions étendues

La gestion des transactions est depuis longtemps la pierre angulaire de la gestion de la concurrence dans les systèmes distribués multi-tiers. A l'origine, les transactions étaient conçues pour des accès brefs et non structurées à des bases de données. De ce fait, la demande de certaines applications d'accès aux données structurés sur de longues durées n'étaient pas pourvues. Par conséquent, le temps nécessaire aux transactions n'est optimal que lorsque chaque transaction a une durée de vie courte. Dès lors que ces transactions ont une durée de vie plus longue, le débit chute, et le système n'est plus en mesure de prendre en charge correctement cette problématique.

Des modèles pour la gestion des transactions étendues ont ainsi été proposés pour capturer ces besoins, appelés modèles étendus de transactions (ATMS – *advanced-transaction models*). Les modèles les plus connus sont celui des transactions imbriquées (Nested Transactions) [Moss, 1981] et les Sagas [Garcia-Molina and Salem, 1987] que nous détaillons.

8.1.1 Les transactions imbriquées

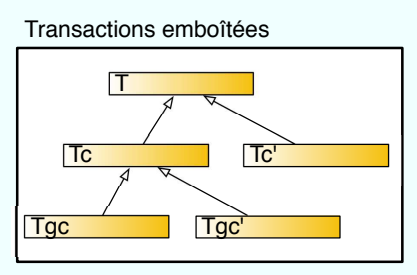

FIG. 8.1 – Les transactions imbriquées

Les transactions imbriquées constituent les plus anciennes et les plus connues des transactions étendues [Moss, 1981]. A la base, une transaction imbriquée T est composée d'un certains nombre de transactions filles Tf. Chacune de ces transactions filles peut voir les données utilisées par la transaction parente et peut elle-même posséder à nouveau des transactions filles. L'ensemble forme ainsi un arbre de transactions. Quand une transaction fille «valide» (*commit*) des données, ces dernières ne sont pas inscrites dans la base de données, mais déléguées à sa transaction parente. Lorsqu'une transaction racine de l'arbre est validée, toutes les transactions filles le sont aussi. Enfin, si une transaction fille est abandonnée (*abort*), la transaction parente n'est pas affectée. Ce qui veut dire que la transaction parente choisit elle-même de valider ou d'abandonner au vu des actions de ses transactions filles.

8.1.2 Les Sagas

Les Sagas font part des modèles de transactions étendues les plus citées [Garcia-Molina and Salem, 1987]. Les sagas sont conçues tout particulièrement pour les transactions longues. Au lieu de traiter une longue transaction, une saga va être découpée en une séquence de sous-transactions. Chacune de ces sous-transactions est une transaction classique et cette séquence est exécutée entièrement avant que la saga soit validée. Pour abandonner ou revenir en arrière sur une saga, la sous-transaction courante est abandonnée, et le travail effectué par toutes les sous-transactions précédentes est défait. Pour permettre cela, le programmeur de l'application doit définir pour chaque sous-transaction une transaction dite

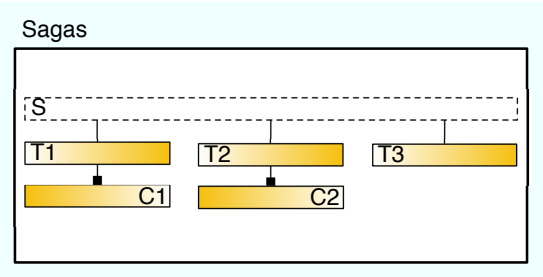

FIG. 8.2 – Les Sagas

de compensation qui réalise une compensation sémantique de l'action effectuée par la sous-transaction. Pour défaire le travail de toutes les sous-transactions précédentes, les transactions de compensation sont exécutées dans l'ordre inverse de l'exécution des transactions initiales.

Comme l'illustre ces deux exemples d'ATMS (les Sagas et les transactions imbriquées), chaque modèle met en avant un système de gestion de la concurrence différent, avec des propriétés différentes. Ceci permet à une application de choisir le modèle étendu qui répond le mieux à ses besoins de concurrence. Si aucun modèle ne convient, il est possible en théorie de créer un nouveau modèle qui fournit les propriétés requises. Au final, quel que soit le modèle employé, il existe un fort couplage entre le code transactionnel et l'application qui est valable autant pour les transactions classiques que pour les transactions étendues comme nous le montrons maintenant.

8.1.3 Transactions étendues et aspects

Les transactions ont souvent servi d'exemple ou de contre-exemple pour la mise en œuvre de développement par aspects [Kienzle and Guerraoui, 2002, Rashid and Chitchyan, 2003, Soares et al., 2002]. Pour utiliser les transactions dans une application, le programmeur doit ajouter du code dit de démarcation pour indiquer les données incluses dans telle ou telle transaction. A l'exécution, une entité séparée du système, le moniteur transactionnel, gère la concurrence en limitant l'accès aux données partagées. La première fonction du code de démarcation est d'informer le moniteur transactionnel du début de la transaction et si elle se termine sur un succès ou un abandon. La seconde fonction du code de démarcation est d'informer le moniteur de tout accès aux données. Le moniteur gère alors les conditions de concurrence. Toutes les données concernées par la concurrence doivent être encadrées par du code de démarcation. Par conséquent, ce code se retrouve éparpillé dans toute l'application. L'approche par aspects peut donc naturellement venir en aide pour la modularisation de ce code.

Dans le domaine des transactions étendues, le code de démarcation s'avère être encore plus important [Fabry, 2005], principalement parce que les transactions étendues sont des combinaisons de transactions classiques auxquelles un certain nombre de propriétés supplémentaires sont ajoutées. Par exemple, pour les transactions imbriquées, un arbre de transactions peut être vu comme de multiples transactions étendues. Avec les Sagas, il s'agit d'une séquence de transactions. Autrement dit, dans les deux cas, le code de démarcation est plus volumineux, et le moniteur transactionnel requiert plus de propriétés pour la gestion de la concurrence.

Le langage KALA L'approche KALA est un langage par aspects dédié au domaine des transactions étendues [Fabry and D'Hondt, 2006]. KALA permet de modulariser ces transactions étendues comme un aspect. KALA repose sur le formalisme d'ACTA qui est un modèle formel pour les transactions étendues largement accepté dans la communauté et qui couvre le champs des

transactions étendues [Chrysanthis and Ramamritham, 1991]. ACTA définit des axiomes qui sont des propriétés supplémentaires ajoutées aux transactions classiques pour composer des transactions étendues. Trois types de propriétés sont définies : des propriétés de *dépendance*, des propriétés de *vue* et des propriétés de *délégation*. Les propriétés de vue et de délégation correspondent simplement à la visibilité d'une transaction par rapport à une autre et à la délégation ascendante ou descendante comme nous l'avons vu précédemment. Les propriétés de dépendance définissent la nature d'une relation entre deux transactions.

Utiliser KALA revient à spécifier certaines méthodes transactionnelles dans l'application. Ainsi, le contrôle de l'accès aux données transactionnelles démarre lorsque la méthode est appelée et se termine lorsque la méthode se termine. De plus, KALA réifie le formalisme ACTA, ce qui permet de définir les propriétés de vue, de dépendance et de délégation, comme des constructions du langage. Pour chaque méthode transactionnelle, le code KALA déclare quelle propriété s'applique à quelle étape du cycle de vie de la transaction : après que la transaction démarre, après qu'elle se termine par validation ou abandon. KALA offre aussi un service de nom pour les transactions, ce qui permet aux transactions de se référencer facilement, notamment dans les déclarations de dépendances. Pour plus de détails sur KALA se référer à [Fabry and D'Hondt, 2006].

Pour illustrer l'utilisation du langage KALA, nous montrons un exemple de code pour une transaction fille dans un arbre de transactions imbriquées (voir Figure 8.3). Nous considérons qu'une méthode fictive `childMethod` dans la classe `util.strategy.Hierarchical` est appelée depuis une autre méthode transactionnelle, qui s'est enregistrée elle-même sous le nom de `parent`. La méthode `childMethod` doit être déclarée comme une transaction imbriquée de cette méthode. Le code KALA de la Figure 8.3 spécifie ces propriétés.

```
util.strategy.Hierarchical.childMethod() {
  alias(root <"parent"> );
  begin {
     dep(self wd root, root cd self);
     view(self, parent) }
     commit { del(self, parent) }
}
```

FIG. 8.3 – Exemple de code KALA

Dans ce code, la seconde ligne permet de récupérer une référence vers la transaction parente grâce au mot-clé `alias`. La spécification des dépendances, qui vient après le début de la transaction de la troisième ligne, s'effectue grâce à la commande `dep` à la quatrième ligne, incluse dans un bloc `begin`. Le mot réservé `self` fait référence à la transaction courante, `wd` et `cd` sont respectivement les *Weak abort Dependency* et *Commit Dependency* d'ACTA [Chrysanthis and Ramamritham, 1991]. Dans le même bloc, à la ligne 5, une relation de vue est déclarée entre une transaction enfant et sa transaction parente. Enfin, la ligne 6 fournit une illustration de la spécification de la délégation dans la phase de validation, par la commande `del`.

Nous avons présenté le langage par aspects KALA, car l'étude réalisée dans ce domaine a mis en évidence un problème important dans la conception et la modularisation de la préoccupation de gestion de transactions étendues. En effet, le code aspect pour les transactions étendues, s'avère être de taille conséquente et complexe, si bien que de nombreuses sous-préoccupations apparaissent que nous détaillons à présent.

8.1.4 Les sous-préoccupations de l'aspect ATMS

Les transactions étendues ne peuvent être considérées comme un seul bloc rigide mais plutôt comme une composition de sous-préoccupations. Si l'on regarde la structure des Sagas, par exemple, on peut identifier deux sous-préoccupations comme la gestion des échecs (*rollback*) et la gestion de la structure des Sagas.

Dans les Sagas, des actions de compensation sont associées aux échecs. Ces actions sont exécutées dans l'ordre inverse d'exécution des sous-transactions, c'est-à-dire des étapes de la Saga. Au niveau du code de l'application, cela revient à définir pour chaque méthode, l'étape inverse. Le code de démarcation correspondant à ces étapes doit également traiter la gestion des échecs. Ce code réalise la tâche de lancer les actions de compensation seulement lorsque la transaction est abandonnée, et qu'elle s'exécutent dans le bon ordre. Ces tâches de bas niveau correspondent donc à la gestion des échecs. De ce fait, cette gestion peut être considérée comme une préoccupation à part entière dans le code de démarcation, car il s'agit d'un choix de conception d'un haut niveau d'abstraction par rapport aux détails d'implantation du code. Ceci suit bien la définition de Parnas [Parnas, 1972] qui établit que tout module doit correspondre à un choix de conception et non une étape de cycle de vie. Il s'agit bien d'un choix de conception car différentes stratégies pourraient très bien être implantées, comme une stratégie consistant à lancer les compensations en parallèle. La sous-préoccupation de gestion des échecs est donc bien légitime.

Le code de démarcation pour les Sagas n'est pas limité au code qui traite les échecs, mais aussi à la gestion de la structure. Chaque méthode correspondant à une étape de la Saga doit être aussi rendue transactionnelle. Cette tâche n'est pas directement reliée à la tâche de définition des actions de compensation. Elle consiste plutôt à construire la structure dans laquelle les sous-préoccupations d'action de compensation doivent agir.

En résumé, on peut voir que les tâches réalisées par le code de démarcation des Sagas concernent deux sous-préoccupations différentes : gestion des échecs et gestion de la structure de la transaction complète. Ces deux sous-préoccupations sont des préoccupations à part entière, au sens de choix de conception. Les lignes de code KALA manipulent donc des concepts de haut niveau et les choix effectués au niveau de ce code impliquent des changements qui modifient le sens de la transaction. Par exemple, la préoccupation de gestion des échecs est implantée à travers une série de définitions de noms et de définitions de dépendances. Le fait de placer ces dépendances différemment en utilisant potentiellement la délégation et les vues formerait une implantation différente de la sous-préoccupation de gestion des échecs.

La séparation du code Sagas pour les deux sous-préoccupations a un impact important sur la façon de voir l'aspect d'ATMS. Par conséquent, l'aspect dans le cas des Sagas traite ici non pas d'une seule préoccupation transactionnelle mais de plusieurs sous-préoccupations. On pourrait proposer d'avoir deux aspects, un pour chaque sous-préoccupation, la structure et les échecs. Mais ce choix serait incohérent dès lors que ces sous-préoccupations n'ont pas de sens, isolément. Elles ne sont pas des préoccupations mais des sous-préoccupations. En effet, si les échecs étaient implantés dans un aspect séparé, tisser cet aspect n'aurait aucun sens sans l'aspect de gestion de la structure des Sagas. En conclusion, nous établissons que ces deux aspects doivent former un tout pour garder du sens.

La séparation des transactions étendues en plusieurs sous-préoccupations n'est pas limitée aux Sagas : des analyses similaires montrent que les différents modèles sont composés de sous préoccupations [Fabry, 2005]. En plus des deux sous-préoccupations identifiées jusqu'ici, s'en ajoutent deux supplémentaires : la gestion des vues et la gestion de la délégation. A noter que cette liste de quatre sous-préoccupations est ouverte. Bien que qu'elles aient été identifiées dans de nombreux modèles, il est toujours possible d'en voir apparaître de nouvelles dans d'autres modèles.

Le langage KALA ainsi que les langages généralistes par aspects pêchent au niveau de leur pouvoir de composition si bien qu'il est impossible d'obtenir une séparation des préoccupations claire. Nous étudions cette problématique dans la section suivante, avant d'étudier la solution dédiée avec le tisseur KALA et la solution générale fournie par FAC.

8.2 Problématique

L'ingénierie de l'aspect de gestion des transactions étendues, que nous appellerons *aspect ATMS* par la suite, est très complexe comme nous le démontrons dans cette section. Nous n'avons pas ici à justifier la pertinence du choix de l'approche par aspects pour la modularisation du code de transaction, du fait des nombreuses études sur le domaine [Kienzle and Guerraoui, 2002, Rashid and Chitchyan, 2003, Soares et al., 2002]. Nous présentons notre problématique à travers le langage dédié KALA. Cependant, le problème d'entrelacement dans le code de l'aspect que nous identifions n'est pas limité à KALA, il s'agit d'un problème général de pouvoir d'expression de la composition. Nous étudions donc ensuite les mécanismes offerts dans les langages par aspects généralistes pour la composition d'aspects.

Lorsque l'on regarde de plus près, l'aspect ATMS lui-même, nous trouvons que, de par sa complexité et sa taille, nous rencontrons des problèmes de modularité. La Section 8.1.4 nous a permis d'en identifier les sous-préoccupations. La Section 8.2.1 étudie l'entrelacement qui apparaît dans la composition de ses sous-préoccupations. La Section 8.2.2 regarde ensuite les mécanismes de composition fournis par les langages par aspects comme AspectJ [Team, 2006]. Enfin, la Section 8.2.2 présente la solution offerte par le tisseur dédié de KALA.

8.2.1 La tyrannie de la décomposition dominante

Nous prenons comme point de départ l'extrait de code donné précédemment dans la Figure 8.3. Nous y identifions trois sous-préoccupations, sans pour autant montrer une séparation claire. La structure de la transaction étendue est maintenue par les déclarations de dépendance situées dans le bloc *begin*. La gestion de vues est réalisée par les déclarations de vues dans le bloc *commit*. La gestion de la délégation, quant-à-elle ; figure dans le bloc *commit*. La déclaration d'alias de la première ligne est, elle, utilisée par les trois déclarations précédentes, entrelaçant ainsi les trois sous-préoccupations.

Cet entrelacement observé dans ces quelques lignes de code KALA se densifie à mesure que la taille du code augmente [Fabry, 2005]. Ceci est dû à la structure en phase du code KALA, qui n'est autre que sa décomposition dite dominante. Le code KALA suit le cycle de vie d'une transaction : phase de démarrage (*begin*), de validation (*commit*), et d'abandon (*abort*). L'implantation d'une sous-préoccupation peut affecter plusieurs de ces phases ou étapes de cycle de vie. Nous prenons l'exemple du scenario suivant de définition d'une propriété de délégation : lorsque la transaction commence, les données d'une autre transaction sont récupérées ; lorsque la transaction est validée, elle est déléguée à cette autre transaction ; lorsque la transaction est abandonnée, elle est déléguée à une troisième transaction. Le code pour cette sous-préoccupation de délégation est éparpillée dans les trois phases comme l'illustre le code KALA de la Figure 8.4.

Lorsque le code de la Figure 8.4 est combiné avec du code d'une autre sous-préoccupation, par exemple la gestion de vues, pour former un ATMS, il est alors inévitable que le code des deux sous-préoccupations soit entrelacé. Ceci parce que la gestion des vues est aussi implantée avec du code qui doit être inséré dans les même blocs *begin*, *commit* et *abort*.

La Figure 8.5 propose un diagramme de flot d'exécution correspondant aux phases d'une

```
[...]
begin { [...]
   del(donor, self)}
commit { [...]
   del(self, donor)}
abort { [...]
   del(self, compensator)}
```

FIG. 8.4 – Exemple de code KALA éparpillant du code de délégation. On observe que la délégation, par exemple, figure dans les trois phases.

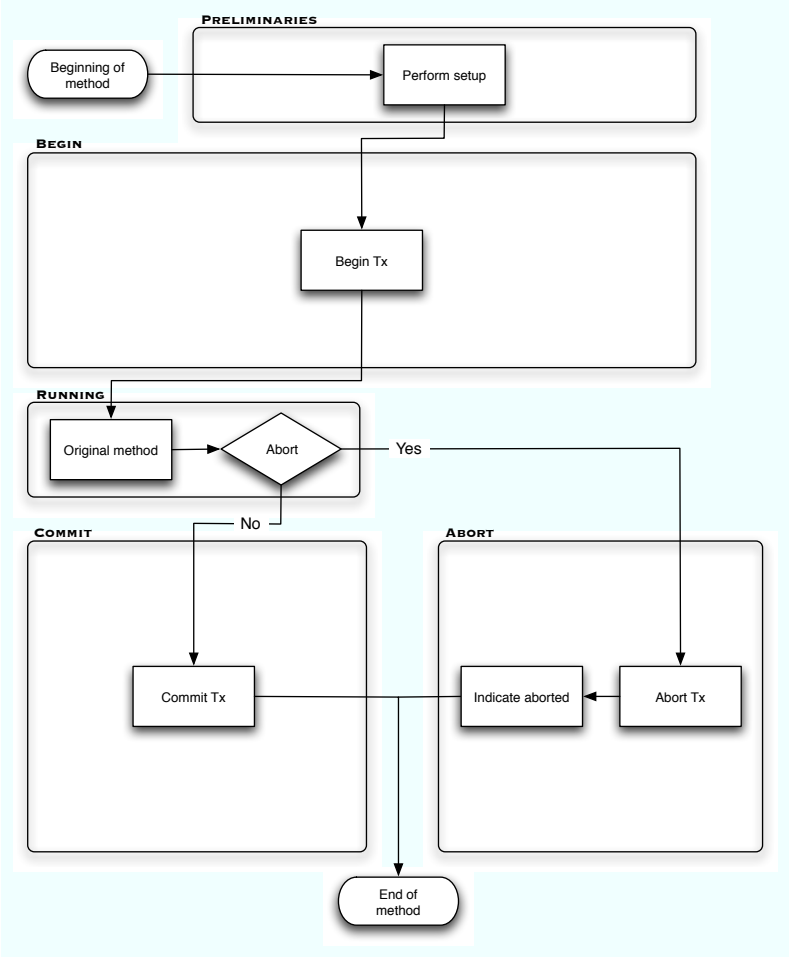

FIG. 8.5 – Le diagramme de flot d'exécution correspondant aux phases de KALA. Ce diagramme sera ensuite complété par les sous-préoccupations.

transaction. Chaque bloc, par exemple les blocs *begin* ou *commit* correspond à une phase du cycle de vie d'une transaction. Une décomposition en phase permet de choisir de représenter chacune de ces phases par un aspect. Le problème, cependant, apparaît dans l'illustration de la Figure 8.6 qui présente le diagramme de flot d'exécution complété avec toutes les sous-préoccupations à insérer dans chacune des phases[6]. On remarque bien l'entrelacement de code qui résulte alors

[6]Ce diagramme est celui fourni par le travail réalisé autour du langage KALA

dans chacun des aspects représentant les phases.

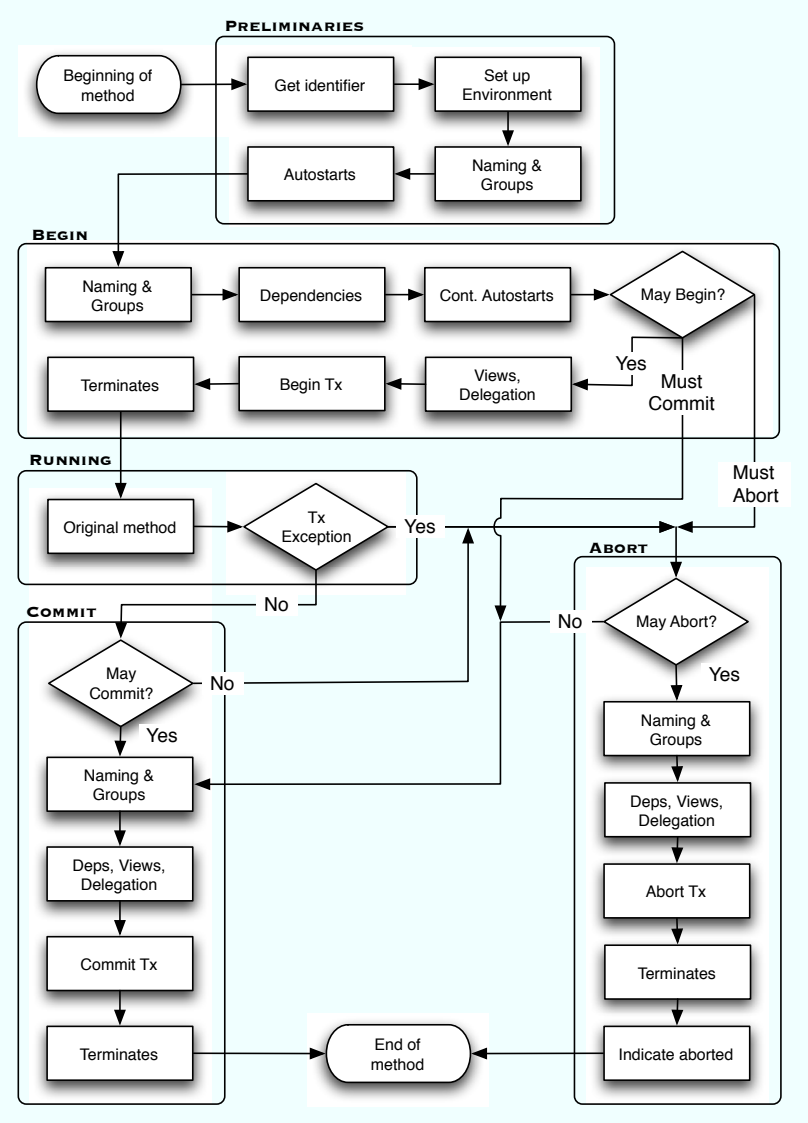

FIG. 8.6 – Diagramme de flot d'exécution de KALA complet.

En bref, nous voyons cet entrelacement comme un cas de la célèbre tyrannie de la décomposition dominante [Tarr et al., 1999]. La décomposition dominante dans le cas de KALA correspond au cycle de vie d'une transaction, c'est-à-dire à ses phases. La modularisation des sous-préoccupations d'un ATMS est cependant orthogonal à son cycle de vie. Une sous-préoccupation peut agir à différents points du cycle de vie, conduisant à de l'entrelacement et de l'éparpillement. Nous regardons à présent le support offert actuellement par les langages par aspects ainsi que la solution apportée par la solution dédiée KALA

8.2.2 Support de composition des approches par aspects

Notre problématique d'entrelacement dans le code de l'aspect ATMS étant établie, nous regardons les mécanismes de composition offerts par les langages par aspects généralistes dans la

Section 8.2.2. Puis, la Section 8.2.2 étudie la solution offerte par le tisseur dédié de KALA.

La précédence d'advice

Le principal outil de composition fourni par l'approche par aspects est le séquencement d'advice, comme on peut la trouver dans AspectJ [Team, 2006]. D'autres langages par aspects comme JAsCo [Suvée et al., 2003] ou Reflex [Tanter and Noyé, 2005] offrent des mécanismes de composition plus sophistiqués. Le cœur de la composition que fournissent ces langages reste néanmoins similaire au séquencement d'advice, c'est-à-dire à un séquencement des aspects par déclaration globale de leurs relations d'ordre. Un des principaux problèmes du séquencement est que sa définition s'effectue de manière globale à une application. Pour rappel, avec FAC, le séquencement d'advice est local à chaque composant (par l'intermédiaire de son interface de tissage). Une granularité fine de gestion d'ordre comme celle de FAC, autorise au moins d'avoir des stratégies différentes sur différentes parties d'un programme.

Il semble dommage de limiter le pouvoir de composition à ce simple mécanisme qui, d'une part, ne permet pas de fournir tous les ordres souhaités entre aspects et, de plus, force à décomposer en aspects en phases qui sont les phases du cycle de vie de l'aspect et non des choix de conception. Chaque sous-préoccupation est incarnée par un aspect qui consiste à définir un comportement avant, après ou autour d'une méthode. Ainsi, ce mécanisme traduit une décomposition suivant les étapes d'exécution des aspects et non des choix de conception. Par conséquent, lorsque des sous-préoccupations doivent agir sur différentes étapes du cycle de vie d'une transaction, la modularisation n'est pas possible. La Figure 8.7 montre la composition requise (bas de la figure) pour l'aspect ATMS face à la composition fournie (haut de la figure) par le séquencement d'advice. Nous avons limité volontairement cet exemple aux blocs *begin* et *commit* qui doivent respectivement être placés avant et après la méthode transactionnelle, qui est la méthode de base sur laquelle sont tissés les deux aspects.

Avec pour seul mécanisme de composition le séquencement d'advice, chaque sous-préoccupation doit être implantée par un aspect. La définition de chaque sous-préoccupation va correspondre donc à une partie «avant la méthode» et «après la méthode». La partie avant va contenir toutes les déclarations de propriétés qui s'appliquent dans les phases situées avant la méthode comme *begin* et la partie après va contenir toutes les propriétés s'appliquant après comme dans les phases de *commit* ou d'*abort*. Pour composer deux sous-préoccupations différentes nous avons besoin de déclarer leur séquencement pour à la fois leur partie avant et après. Ce qui nous conduit comme seul ordre possible le haut de la Figure 8.7. Il est impossible d'obtenir l'ordre requis par le bas de la figure.

Nous observons sur le bas de la figure que toutes les propriétés de dépendance (`DEP`), délégation (`DEL`) et de vue (`VIEW`) doivent être regroupées et ceci dans chaque phase. Cet ordre est très important car si l'on considère par exemple les propriétés de dépendance dans la phase *begin*, il est clair que ces dépendances impactent l'amorçage de la transaction elle-même et doivent être fixés en priorité. L'ordre proposé dans le haut de la figure, éparpillant les parties `DEP` conduit à des erreurs.

En bref, le séquencement d'aspects ne répond pas à nos besoins car il permet à des propriétés de dépendance, vue et délégation d'être définis plusieurs fois dans une phase. Il est donc impossible de fusionner, de la manière voulue, deux sous-préoccupations en un aspect complet définissant un ATMS. Nous regardons à présent comment KALA vient à bout de ce problème de composition grâce à son tisseur dédié.

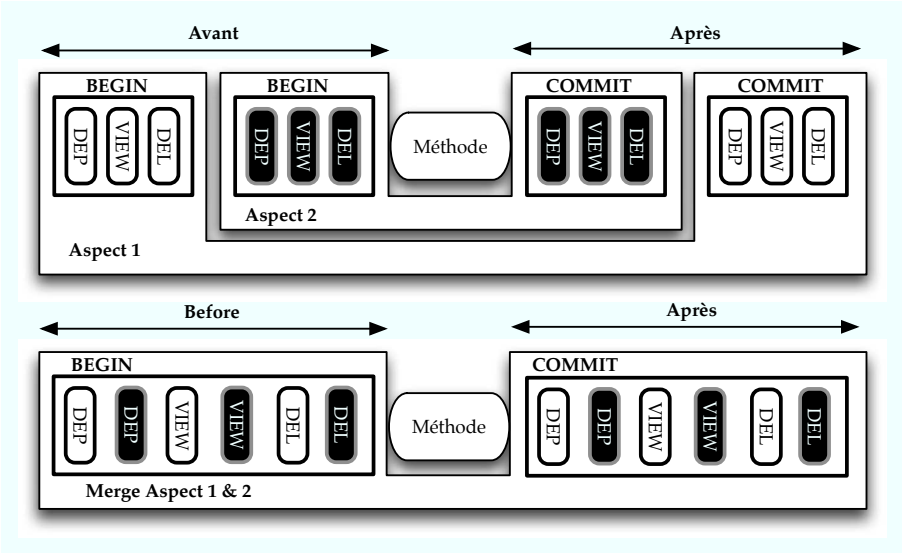

FIG. 8.7 – Composition fournie par séquencement d'advice versus composition souhaitée

KALA : Une solution dédiée

Par contraste avec le mécanisme de séquencement d'advice, KALA fournit un support pour la modularisation de sous-préoccupations d'un ATMS. KALA étant un langage dédié, il a été spécialement conçu pour venir à bout de ce problème d'entrelacement de code dans l'aspect ATMS. Ainsi, le langage peut prendre en compte les mécanismes de composition requis pour modulariser les propriétés du domaine.

Dans KALA, l'aspect ATMS peut être défini à travers l'utilisation de multiples fichiers KALA pour une seule préoccupation. Par exemple, les transactions imbriquées peuvent être définies à travers trois spécifications, chacune dans un fichier à part.

```
util.strategy.Hierarchical.childMethod() {
   alias(root <"parent"> );
   begin { dep(self wd root, root cd self) }
}
```

FIG. 8.8 – Exemple de fichier KALA pour la gestion de la structure

```
util.strategy.Hierarchical.childMethod() {
   alias(root <"parent"> );
   begin { view(self, parent) }
}
```

FIG. 8.9 – Exemple de fichier KALA pour la gestion des vues

Les spécifications des Figures 8.8, 8.9 et 8.10 correspondent respectivement à la gestion de la structure, des vues et de la délégation. Ces trois sous-préoccupations forment un module auto-contenu dans lequel aucun code d'autres préoccupations n'est présent. Par conséquent il n'existe pas d'entrelacement de code.

A l'étape du tissage, le tisseur de KALA combine ces trois spécifications qui s'appliquent sur

```
util.strategy.Hierarchical.childMethod() {
   alias(root <"parent"> );
   commit { del(self, parent) }
}
```

FIG. 8.10 – Exemple de fichier KALA pour la gestion de la délégation

la même méthode, en un seul aspect. En résumé, le tisseur est capable de restituer l'entrelacement pour obtenir le bon séquencement de chaque morceau de sous-préoccupation. Les potentiels conflits de nommage sont également pris en charge.

En bref, la composition de code KALA est réalisée au niveau des éléments du domaine en utilisant les concepts du domaine et ses propriétés de composition. Cependant une telle étude du domaine et la création d'un langage dédié, ainsi que d'un tisseur dédié pour chaque aspect complexe n'est pas toujours possible. Il semble plus intéressant de se tourner vers une solution plus générale comme nous allons le voir à présent avec FAC.

8.3 La solution apportée par FAC

La solution que nous avons construite avec FAC s'inspire fortement du travail réalisé autour du langage KALA. Nous reprenons le travail préliminaire réalisé autour de l'approche en termes d'identification des sous-préoccupations. En suivant la décomposition réalisée par KALA, nous avons pu construire un assemblage de composants FRACTAL et de composants d'aspects FAC modularisant la préoccupation complexe de gestion de transactions étendues (ATMS).

Nous détaillons donc l'architecture obtenue en présentant tout d'abord un scénario d'intégration (Section 8.3.1) et nous faisons une correspondance entre le vocabulaire de FAC et celui employé pour le domaine des transactions avancées. Puis, nous détaillons l'architecture de l'aspect ATMS : les préoccupations de base (Section 8.3.2), les préoccupations transverses (Section 8.3.3). Enfin, la Section 8.3.4 montre comment l'aspect ATMS est architecturé pour obtenir la composition requise.

8.3.1 Scénario d'intégration

Avant de se plonger dans l'architecture de l'aspect ATMS, nous introduisons un scénario pour l'intégration de la préoccupation de gestion de transactions étendues. Nous reprenons pour cela l'exemple du serveur HTTP minimal Comanche. Pour rappel, le rôle de Comanche consiste à accepter des requêtes TCP et de transmettre le fichier demandé lorsqu'il existe, ou de renvoyer une erreur si le fichier n'est pas supporté (voir Figure 8.11).

Nous souhaitons étendre cette architecture avec un support transactionnel. Supposons que nous voulions rendre le traitement d'une connexion transactionnelle. Cette opération concerne plusieurs composants de l'architecture car il requiert l'identification unique d'une connexion et le stockage de tout changement d'état pouvant intervenir pendant cette opération. Nous considérons donc que le composant *Request receiver* et le composant *Request handler* doivent être rendus transactionnels. Nous ne re-démontrons pas l'apparition d'entrelacement de code qu'engendrerait l'ajout d'interfaces clientes sur ces deux composants pour l'appel du moniteur transactionnel, et nous reportons le lecteur au Chapitre 4 p.60. Nous n'argumentons pas non plus sur la possibilité de placer la gestion du service transactionnel dans la membrane, puisque FAC repose sur un modèle homogène (voir discussion du Chapitre 6 p.61).

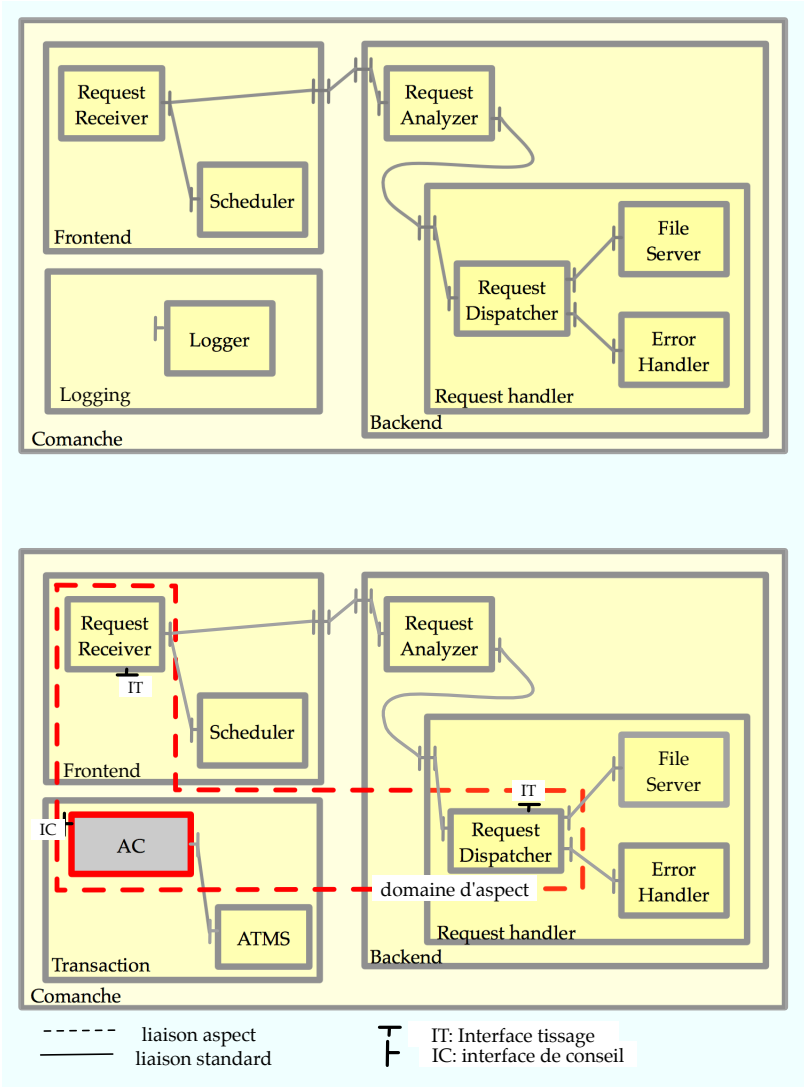

FIG. 8.11 – L'architecture de Comanche étendue par un service de transaction. Liaisons d'aspect omises par souci de clarté.

Nous proposons donc d'extraire la préoccupation ATMS sous forme d'un aspect. Si l'on suit la méthodologie FAC, nous avons à décrire un composant d'aspect qui va déléguer son implantation, dans notre cas, à un moniteur transactionnel (`ATMS Monitor` sur la Figure 8.11). Ensuite, le composant d'aspect est tissé sur les méthodes qui doivent être rendues transactionnelles. L'intégration est donc ici strictement la même que l'exemple de la journalisation du Chapitre 6 p. 91.

Cependant, comme nous avons pu le voir dans notre problématique de la Section **??**, les transactions avancées constituent un aspect complexe, c'est-à-dire composé de nombreuses sous-préoccupations. Nous utilisons le terme sous-préoccupation pour signifier qu'il s'agit de préoccupations mais liées fortement à l'aspect ATMS et qui doivent donc être composées d'une certaine manière pour former un ATMS. Du fait de cette configuration très particulière nous employons le terme *sous-préoccupation*. Cependant, il n'en demeure pas moins que, du point de vue de FAC, ces sous-préoccupations seront traitées comme toutes autre préoccupations. Si certaines de ces sous-préoccupations s'avèrent être de nature transverse, nous aurons à ré-appliquer la méthodologie

FAC au sein même de l'aspect.

Avant de rentrer dans le détail de la solution proposée par FAC, nous clarifions le vocabulaire employé pour le domaine des transactions avancées avec le vocabulaire de FAC. Lorsque, par la suite nous évoquons les cycles d'une transactions (les phases *begin*, *commit*, *abort*), il s'agira de préoccupations avec FAC, donc de composants. Ces composants peuvent être ou non de nature transverse, et donc être intégré par des liaisons classiques FRACTAL ou des liaisons d'aspects FAC. Lorsque nous évoquons les propriétés d'un ATMS (délégation, vue, dépendance), il s'agit aussi de préoccupations, qui peuvent aussi être transverse ou non. Enfin, ce que nous appelons sous-préoccupation pour le domaine des transactions avancées, et qui implique une composition de propriétés intervenant dans certains cycles, il s'agit également d'une préoccupation avec FAC, qui correspond aussi à un composant ou à un assemble de composants. Dans ce dernier cas, les composants feront partie d'un composite qui caractérisera cette préoccupation. Et encore une fois, la nature transverse ou non de la préoccupation fera ou non intervenir les mécanismes aspects de FAC.

Cette section nous a permis de faire un éclaircissement du vocabulaire propre au domaine des transactions avancées et d'observer que tout est vu comme préoccupation avec FAC. Nous nous intéressons à présent à la description de l'architecture de l'ATMS en lui même, pour apprécier comment FAC permet l'obtention de la séparation de toutes les préoccupations.

8.3.2 Les sous-préoccupations de base du cycle de vie d'une transaction

Comme nous l'avons vu précédemment, la décomposition dominante de l'aspect ATMS correspond à la décomposition suivant le cycle de vie d'une transaction. Ces phases viennent se tisser autour de la méthode rendue transactionnelle. La Figure 8.12 fournit une représentation de l'architecture construite à partir de cette décomposition dominante, que nous appellerons le cœur de l'aspect ATMS. Le cœur correspond à la gestion standard d'une transaction. Il peut donc fonctionner seul et représente une transaction classique avec ses phases. Nous décrivons les composants de cette architecture présentée sur la Figure 8.12. Cette architecture suit le diagramme de flot d'exécution utilisé dans KALA de la Figure 8.5 p. 105. Ce diagramme caractérise le rôle des composants du cœur de l'aspect ATMS et sera par la suite complété avec les sous-préoccupations de l'aspect ATMS.

Le composant `Preliminaries` Ce composant permet de définir une configuration générale du système. Ceci inclut également l'obtention d'un identifiant unique pour la transaction qui est utilisé après dans tous les cycles.

Le moniteur transactionnel, le composant `TP Monitor` Ce composant est le classique moniteur transactionnel qui permet de gérer la concurrence. Ce composant est central et requis par les composants `Begin`, `Commit` et `Abort`.

Le composant d'aspect et la phase de *Running* Dans cette architecture, le composant d'aspect *ATMS AC* joue un rôle important. Il permet de réifier la méthode interceptée et d'appeler les composants correspondant aux parties «avant» et «après» de l'exécution de la méthode transactionnelle. En bref, il pilote la progression des étapes du cycle de vie de la transaction, d'où les liaisons entre le composant d'aspect et les composants *begin*, *commit*, et *abort*. La phase de *Running* est représentée par le composant d'aspect qui enveloppe la méthode interceptée et permet de capturer une éventuelle exception pour procéder à un abandon.

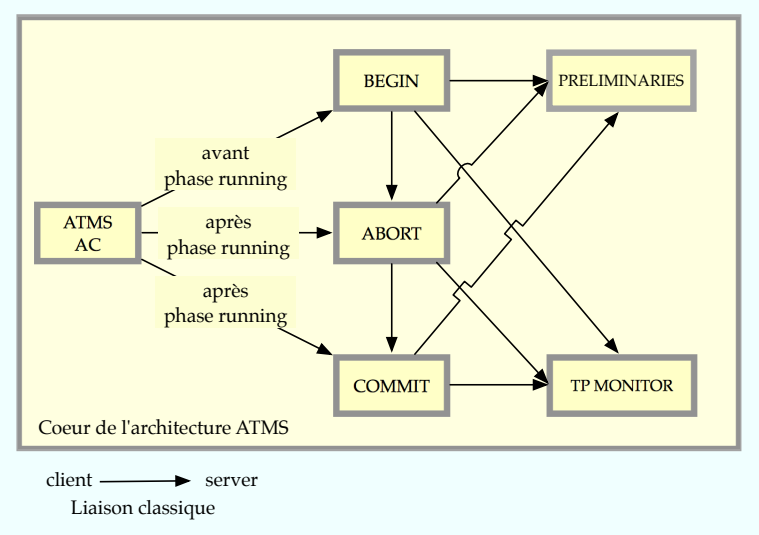

FIG. 8.12 – Cœur de l'architecture de l'aspect ATMS. Par souci de clarté, les liaisons entre composants sont représentées par des flèches.

Le Listing 8.1 correspond au code du composant d'aspect de la Figure 8.12. Les dépendances vers les composants utilisés par le composant d'aspect sont déclarés aux Lignes 3–6. Le composant d'aspect commence par fixer les conditions d'environnement (Ligne 11). Ensuite, il appelle la phase de *begin* (Ligne 13), puis selon que la transaction est validée (Ligne 18) ou abandonnée (Ligne 17) les composants concernés sont appelés, respectivement le composant `Commit` ou `Abort`.

```
1@AspectComponent
2public class AtmsAC implements AroundAspectComponent {
3    @Requires(name = "abort") private IAbort abort;
4    @Requires(name = "commit") private ICommit commit;
5    @Requires(name = "begin") private IBegin begin;
6    @Requires(name ="preliminaries") private IPreliminaries preliminaries;
7    // code de l'advice de type autour
8    public Object invoke(FcMethodInvocation m) throws Throwable {
9      Object retval = null;
10     // fixe l'environnement de la transaction
11     preliminaries.setEnvironnement();
12     // phase de begin
13     if (begin.begin()) // may begin and begin
14       try {
15          // phase de running
16          retval = m.proceed();
17       } catch (TxException ex) {  abort.abort(tx_id); // phase d'abort }
18       commit.commit(tx_id); // phase de commit
19       return retval;
20   }
21}
```

Listing 8.1 – Le code du composant d'aspect ATMS

Le composant `Begin` Le composant correspondant à la phase de *begin* dépend du moniteur transactionnel (voir Lignes 5-6 du Listing 8.2) et du composant *abort* (Lignes 5-6). Le moniteur transactionnel est requis pour savoir si l'amorçage de la transaction est autorisé (Ligne 9). Le moniteur transactionnel peut alors fournir deux types de réponses : la transaction peut commencer (Ligne 21), ou doit être immédiatement abandonnée (Ligne 12).

```
1@FractalComponent(controllerDesc = "aspectizableComponent")
```

```
2public class Begin implements IBegin {
3   @Requires(name = "tpmonitor") private static ITPMonitor txmgr;
4   @Requires(name = "abort") private IAbort abort;
5   public Boolean begin(Integer tx_id) throws TxException {
6      // may begin?
7      Forcing beginForcing = txmgr.mayBegin(tx_id);
8      Boolean runTx = true;
9      if (beginForcing != null) {
10        if (beginForcing.direction.equals("Abort"))
11           try {
12              abort.abort(tx_id);
13           } catch (TxException e) {
14              throw e;
15           }
16           else
17              runTx = false;
18     }
19     // begin
20     if (runTx) {
21        txmgr.begin(tx_id);
22     }
23     return runTx;
24  }
25}
```

Listing 8.2 – Code du composant *Begin*

Le composant `Commit` Le composant correspondant à la phase de `commit` fonctionne de manière similaire à la phase de *begin* dans le sens où le composant vérifie auprès du moniteur transactionnel si il peut valider la transaction (Listing 8.3 Ligne 8), avant de lancer cette validation en tant que telle (Ligne 12). Si la validation n'est pas possible, la phase d'abandon est lancée (Ligne 9), d'où la dépendance vers le composant `Abort` Ligne 4.

```
1@FractalComponent(controllerDesc = "aspectizableComponent")
2public class Commit implements ICommit {
3   @Requires(name = "tpmonitor") private static ITPMonitor txmgr;
4   @Requires(name = "abort") private IAbort abort;
5   public void commit(Integer tx_id) throws TxException {
6      //  May Commit
7      Forcing commitForcing = txmgr.mayCommit(tx_id);
8      if (commitForcing != null) {
9         abort.abort(tx_id);
10     }
11     try {
12        txmgr.commit(tx_id);
13     } catch (TxException e) {
14        abort.abort(tx_id);
15     }
16  }
17}
```

Listing 8.3 – Code du composant *Commit*

Le composant `Abort` La phase d'*abort* est identique à la phase de *commit* à deux différences près. Tout d'abord si la transaction doit être validée de manière forcée, le flot de contrôle évite le choix effectué (*may commit*) dans la phase de *commit* pour prévenir des éventuelles boucles infinies dans le cas d'une transaction qui est à la fois forcée à la validation et à l'abandon à cause de ses dépendances. Seconde différence, une exception est levée et récupérée par le composant d'aspect pour signifier à la méthode transactionnelle que la transaction est abordée. Ceci est important car le composant du niveau de base appelant cette méthode transactionnelle peut elle-même être une méthode transactionnelle qui doit être informée de l'abandon de la transaction.

Le Listing 8.4 présente le code FRACTAL-ADL décrivant l'architecture de la Figure 8.12. Nous observons deux parties distinctes : Les Lignes 2–8 les déclarations des composants, et les lignes 9–11 les liaisons entre ces composants.

```
1<definition name="core.ATMSComposite" >
2  <!-- composants de l architecture coeur -->
3  <component name="atmsaspect" definition="core.AtmsAC" />
4  <component name="environment" definition="core.Preliminaries" />
5  <component name="begin" definition="core.Begin" />
6  <component name="commit" definition="core.Commit" />
7  <component name="abort" definition="core.Abort" />
8  <component name="tpmonitor" definition="core.TPMonitor" />
9  <!-- déclarations de liaisons -->
10 <binding client="atmsaspect.begin" server="begin.begin" />
11 (...)
12</definition>
```

Listing 8.4 – Description de l'assemblage du cœur de l'architecture de l'aspect ATMS avec FRACTAL-ADL

Nous avons vu dans cette première étape comment nous construisons l'architecture correspondant à la décomposition dominante en phase. À présent, il nous reste à venir greffer sur cette première architecture les propriétés des sous-préoccupations qui lui sont transverses, comme les propriétés de dépendance, de délégation, de vue, etc. Ces sous-préoccupations permettent de former un ATMS complet.

8.3.3 Les sous-préoccupations ou propriétés des ATMS

Nous détaillons à présent les architectures correspondant aux sous-préoccupations de l'aspect ATMS. Ces sous-préoccupations doivent être tissées sur les composants des sous-préoccupations de base que nous avons présentés dans la section précédente et qui correspondent aux cycles de vie d'une transaction. L'intégration d'une sous-préoccupation se fait en deux phases : définition de ses propriétés qui sont des composants d'aspects configurables, composition des ces propriétés pour former une sous-préoccupation, comme, par exemple, la gestion de la structure d'un ATMS.

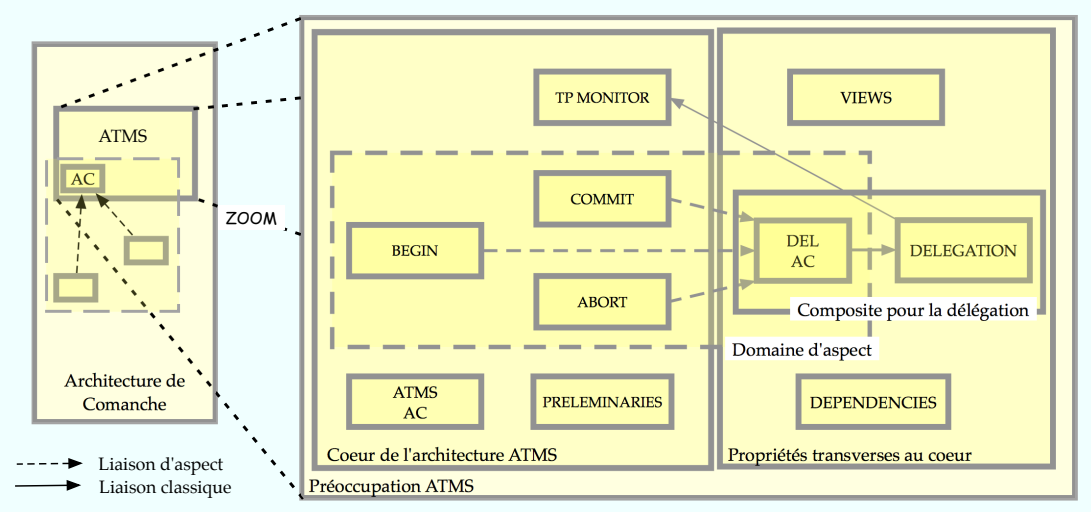

FIG. 8.13 – Schéma conceptuel de l'architecture étendue pour l'aspect ATMS

Les propriétés La Figure 8.13 présente comment les propriétés de délégation sont intégrées dans l'architecture de base (ce que nous avons appelé l'architecture cœur précédemment). Par souci de clarté, nous avons omis toutes les liaisons qui ne concernent pas cette intégration. Les

autres propriétés identifiées par KALA sont traitées et intégrées de manière identique à ce qui a été présenté sur la Figure 8.13. Ainsi on remarque que du fait de l'unification de FAC il est possible d'appliquer des aspects sur des composants qui font déjà partie d'un aspect : dans notre cas, des aspects pour les propriétés qui s'appliquent sur les composants représentant des cycles de l'aspect ATMS. Par conséquent, étape après étape, il est possible d'extraire tout l'entrelacement dans les composants y compris un aspect, conséquence directe de l'unification réalisée par FAC.

```
1@FractalComponent
2public class Delegation implements IDelegation {
3   // Required interfaces
4   @Requires(name = "tpmonitor") private static ITPMonitor txmgr;
5   @Requires(name = "preleminaries") private IPreleminaries preleminaries;
6   // 2 attributs de composant
7   @Attribute private String delsource;
8   @Attribute private String deltarget;
9   // Implementation of the IDelegation interface
10    public void performDelegation() throws TxException {
11          ...
12    }
13}
```

Listing 8.5 – Code du composant Delegation

Le Listing 8.5 présente le code du composant traitant de la propriété de délégation. Le code se distingue en trois parties : la déclaration d'interfaces requises Lignes 4–5, la déclaration d'attributs de composant Lignes 7–8, l'implantation de la méthode de l'interface *IDelegation*. Le point important dans cette implantation, est la présence d'attributs de composant. Ils permettent de rendre les instances de ce composant paramétrables. Autrement dit, ces attributs sont le point de variation d'une propriété de délégation à une autre, ce qui fait que ces composants peuvent être directement réutilisés et configurés au niveau même de l'architecture. En particulier, FRACTAL-ADL offre la possibilité de fixer les valeurs d'attributs de composants au niveau de la déclaration des composants. Ainsi avec FAC, nous rendons les composants utilisés pour les propriétés de délégation, vue et dépendance complètement paramétrables depuis la description d'architecture. Ce détail n'est pas des moindre car ces déclarations se rapprochent alors de lignes de code KALA. Ce qui signifie que les déclarations KALA se retrouvent directement dans FRACTAL-ADL. On voit donc que le langage généraliste d'architecture FRACTAL-ADL supporte les déclarations spécifiques d'un langage dédié KALA par le biais des attributs des composants.

Définition d'une sous-préoccupation Pour rappel, avec le langage dédié KALA, une sous-préoccupation, comme la gestion de la structure de l'ATMS peut être définie dans un fichier indépendant fixant des propriétés de vue, délégation, dépendance dans les cycles d'une transaction. Avec FAC le processus est identique à l'exception que les cycles sont des composants, les propriétés des aspects tissés sur ces cycles (action de mettre une propriété dans un bloc correspondant à un cycle en KALA), les paramètres des propriétés sont également fixés au niveau de l'architecture comme l'illustre le code du Listing 8.6. Le code XML présenté dans ce listing comporte plusieurs parties. Les Lignes 2–8 concernent la déclaration des composants cœurs, c'est-à-dire les composants que nous avons étudiés dans la section précédente. Ensuite, nous voyons comment l'architecture de base peut être enrichie de propriétés spécifiques qui sont les propriétés de délégation, dépendance et vue. La Ligne 3 montre comment une dépendance est fixée. La partie passée en paramètre du composant (entre parenthèse dans le code XML) est tout simplement une ligne de configuration en KALA qui fixe une dépendance : `self;wd;parent,parent;cd;self`. Le tissage du composant de la propriété de dépendance est ensuite tissé dans la phase *begin* à la Ligne 5. Ce fichier d'architecture FRACTAL-ADL est l'équivalent de celui fourni par KALA. Le service de nom de KALA est avec FAC simplement le service de nom des composants FRACTAL, donc naturellement supporté.

Pour généraliser notre discours nous observons sur la Figure 8.6 p.106 présente le diagramme de flot d'exécution de KALA complet avec tous les blocs constitutifs. FAC reconstitue ce flot

en venant tisser les différentes propriétés autour des éléments de base du cycle de vie d'une transaction.

```
1<definition name="subconcern.StructureManagement" >
2  <!-- composants spécifiques définissant des propriétés -->
3  <component name="dep" definition="additional.Dep(self;wd;parent,parent;cd;self)" />
4  <!-- weaving declarations -->
5  <weave root="this" ac="dep" pointcutExp="begin;begin;begin*" aDomain="depAD" />
6</definition>
```

Listing 8.6 – Description de l'assemblage d'une sous-préoccupation FRACTAL-ADL

8.3.4 Analyse de la séparation des préoccupations fournie par FAC

Dans la Section 8.2.2 nous avons pointé un manque de pouvoir de composition dans les langages par aspects, et nous avions regardé en détail la solution proposée par le tisseur dédié KALA. Nous revenons ici sur cette discussion et notamment sur les limitations du séquencement d'advice. La Figure 8.14 compare la composition fournie par le séquencement d'advice en haut, et la composition fournie par FAC en bas.

Nous observons qu'avec FAC, les deux aspects correspondant à deux sous-préoccupations différentes sont bien séparés dans deux composites différents. Les propriétés de dépendance, vue et délégation sont également identifiés dans des composites. Les composants correspondant aux phases *begin* et *commit* sont indépendants et les propriétés de dépendance, vue et délégation sont tissés sur leur comportement de base. La glu entre toutes ces préoccupations est réalisée tantôt par des liaisons FRACTAL tantôt par des liaisons d'aspect, selon que l'application d'une préoccupation est transverse ou non vis-à-vis des autres composants. L'ordre requis sur les définitions des propriétés de dépendance, vue et délégation est respecté, ceci grâce à la granularité d'ordonnancement fine de FAC qui permet de définir un ordre pour chaque opération aspectisable d'un composant. Sur les composants *Commit* et *Begin*, on peut donc constater que ces propriétés sont bien tissées dans le bon ordre tout en étant bien identifiées dans des composants différents.

Enfin, notons que cette séparation des préoccupations complète obtenue dans notre architecture est possible grâce à la notion de composant partagé. A titre d'exemple, le composant *DEP* en noir à gauche de la Figure 8.14 est partagé à la fois par le composite concernant la sous-préoccupation de dépendance et par le second aspect (*aspect 2* sur la figure). Nous procédons à présent à une évaluation de notre solution, notamment en comparaison avec la solution dédiée KALA.

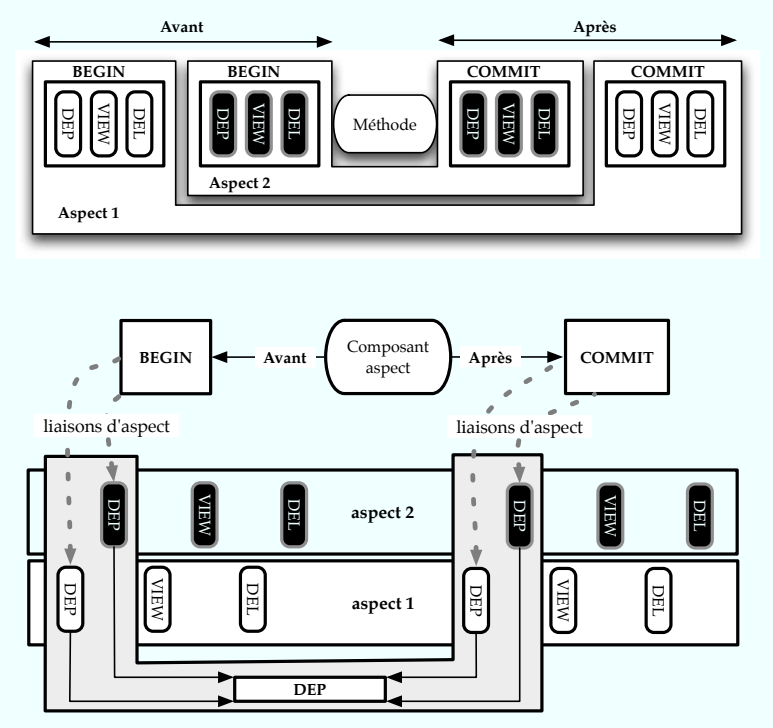

FIG. 8.14 – Composition fournie par le séquencement d'advice contre composition fournie par FAC.

8.4 Evaluation

La solution proposée avec FAC reste très proche des résultats obtenus avec KALA, car nous nous sommes servi des mêmes bases. La différence majeure réside dans le fait que FAC est une approche générale, qui peut s'appliquer à d'autres domaines : le chapitre suivant étudie le domaine de la communication de groupe. FAC sert de support au langage KALA, car, nous l'avons vu les paramètres des composants correspondent à du code KALA. Ainsi FAC peut fournir un support général au langage KALA sans nécessiter un tisseur dédié pour pouvoir constituer la bonne composition des blocs de l'aspect ATMS. En résumé, FAC permet de porter KALA à un niveau composant sans nécessiter le développement d'un tisseur dédié.

Le Tableau 8.1 compare FAC et KALA sur plusieurs critères que nous analysons dans les paragraphes suivants.

Critère	KALA	FAC
Cycle d'une transaction		
Représentation	Bloc de code KALA	Composant
Ordonnancement	Tisseur KALA	Composant d'aspect `ATMS AC`
Propriété		
Représentation	Ligne de code KALA	Composant d'aspect déléguant éventuellement à un composant
Configuration	paramètre d'une fonction del, dep, ou view	Attribut de composant configurable au niveau du fichier ADL
Sous-préoccupations)		
Représentation	Fichier KALA	Fichier FRACTAL-ADL avec les composants et leur tissage
Séparation des préoccupations		
Propriétés	+	+
Cycle d'une transaction	+	+
Base/aspect GCS	+	+

TAB. 8.1 – Comparaison de KALA et FAC

Cycle d'une transation Un cycle d'une transaction de type *begin*, *commit* ou *abort* est représenté par un bloc du même nom dans du code KALA et par un composant en FAC. Leur ordonnancement est géré automatiquement par le tisseur dédié de KALA et par le composant d'aspect *ATMS AC* avec FAC qui appelle dans l'ordre requis les composants *begin*, *commit* et *abort* ainsi que le moniteur transactionnel.

Propriété Une propriété comme celle de délégation, vue ou dépendance correspond à une ligne de code du langage KALA, alors qu'elle correspond à un composant d'aspect avec FAC. Ce composant d'aspect est ensuite tissé sur les composants représentant les cycles d'une transaction. De plus de part leurs attributs ces composants sont configurables au niveau de leur description architecturale avec FRACTAL-ADL, ce qui correspond au paramètre passé à une fonction de type *dep*, *del* ou *view* dans le langage KALA faisant références au propriétés du même nom.

Sous-préoccupation Une sous-préoccupation comme la gestion de la structure est représentée en KALA par un fichier indépendant fixant une combinaison de propriétés de dépendance, délégation ou vue associé à des phases de type *begin*, *commit* ou *abort*. De manière similaire FAC permet de définir le même type de configuration dans un fichier FRACTAL-ADL qui constitue un composite réifiant la sous-préoccupation et incluant les composants correspondant à ces propriétés de dépendance, vue, délégation ainsi que leur tissage aux composants représentant les cycles de type *begin*, *commit* ou *abort*.

Séparation des préoccupations Aussi bien KALA que FAC permettent d'obtenir une séparation de toutes les préoccupations. Cependant la différence de FAC est de porter les principes de KALA au niveau composant en ne nécessitant aucun tisseur dédié.

8.5 Bilan

Nous avons présenté dans ce chapitre une étude de cas sur les transactions étendues de type Saga. En particulier, nous nous somme intéressé à l'ingénierie d'un aspect dit complexe. En effet, en choisissant de représenter la gestion des transactions étendues sous forme d'un aspect des problèmes de composition apparaissent.

Nous avons montré que lorsqu'on choisit de représenter les transactions étendues par un aspect qui ne peut être considéré comme un seul bloc, mais que des sous préoccupations apparaissent. Ces sous-préoccupations qui correspondent à des choix de conception viennent s'entrelacer entre autres avec les cycles d'une transaction : l'amorçage, la validation et l'abandon. Les langages par aspects généralistes ne fournissent pas le pouvoir de composition nécessaire pour venir à bout de ce problème.

Nous avons donc étudié la solution proposée par le langage dédié KALA ainsi que son tisseur dédié. Puis, nous avons montré comment FAC pouvait fournir un support de composition suffisant pour supporter les principes du langage KALA sans nécessiter le développement d'un tisseur dédié. FAC permet de fournir une séparation des préoccupations complète au sein de cet aspect complexe qu'est la gestion des transactions étendues.

Chapitre 9

La gestion de la communication de groupe

Sommaire

Ce chapitre expose le travail que nous avons réalisé dans le cadre de la conception d'un service de communication de groupe (*GCS Group Communication Systems*) selon les principes du développement par aspects. Nous justifions tout d'abord le besoin de modulariser la communication de groupe comme un aspect. Puis, nous mettons en évidence la complexité de l'aspect et sa décomposition en sous-préoccupations. Nous montrons que nous nous heurtons à des problèmes de séparation des préoccupations au sein du code de l'aspect lui-même, dû à un entrelacement de ses sous-préoccupations. Nous montrons comment la solution générale proposée par FAC réduit cet entrelacement.

9.1 La communication de groupe

Les systèmes à communication de groupe que nous étudions dans ce chapitre reposent sur le paradigme utilisé dans le contexte des données et de la réplication de services [Chockler et al., 2001]. À la base, nous trouvons la notion de processus qui peut représenter une machine, un processeur d'une machine, ou encore un fil d'exécution d'un processeur. Les processus sont situés sur différents nœuds d'un réseau distribué et opèrent comme un groupe.

Ils sont alors appelés *membres* et sont gérés par le service de gestion de groupes (*group membership service*). La coopération au sein d'un groupe requiert un échange de messages en multipoint. Cette communication est prise en charge par le *service de groupe multicast*. Ce service garantit certaines propriétés des messages comme la fiabilité ou un ordre logique (FIFO, causal ou total).

9.1.1 Cas de la réplication active

Pour illustrer les mécanismes des systèmes de gestion de groupe, nous prenons le cas classique de la réplication active [Schneider, 1990]. Comme le montre la Figure 9.1, le principe est de répliquer les serveurs et de coordonner les interactions clientes avec ces répliques. Le scénario que nous présentons a été mis en place au sein d'applications à grande échelle dans l'étude [Rhee et al., 1997]. La réplication active permet de faire de la tolérance aux fautes de manière transparente, de telle sorte que les processus client, comme serveur, n'ont pas conscience du fait que les serveurs sont répliqués. Cependant, dans la pratique, cette transparence est difficilement atteinte. En effet, une des propriétés que doit respecter la réplication active est l'*atomicité* de l'exécution de chaque commande. Ceci est difficilement réalisable sans avoir conscience de faire partie de cette réplication active, car le client doit être isolé des réponses des serveurs ; les appels et traitements interrompus au niveau des serveurs doivent être pris en compte par le client ; etc. Il existe un entrelacement entre le code de base des clients et serveurs et celui requis par les mécanismes de la réplication active. Il peut donc être intéressant d'envisager un développement par aspects pour ce service de gestion de groupe.

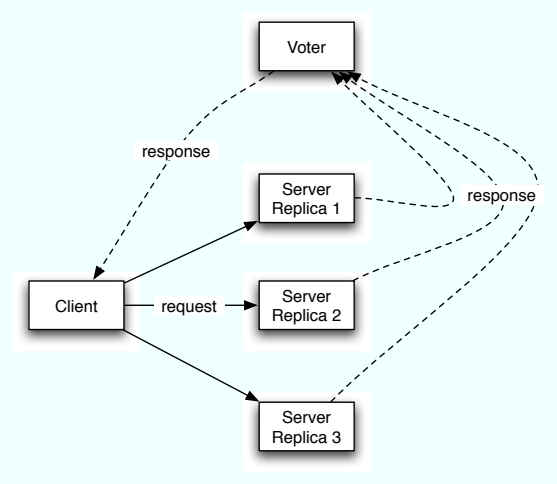

FIG. 9.1 – Communication de groupe : exemple de la réplication active

9.1.2 Notion de pile de protocole

La réplication active impose le respect de l'atomicité des commandes exécutées. Cette propriété prend place parmi de nombreuses autres propriétés, comme le fait de joindre un groupe, envoyer ou recevoir un message, etc. Ces propriétés, en communication de groupe, sont généralement représentées comme une pile de protocoles. Chaque propriété correspond à un microprotocole. Les microprotocoles sont ensuite ordonnancés et liés pour former ce qu'on appelle une *pile*. Le modèle de programmation des piles de protocoles repose sur la réaction à des événements de nature asynchrone (voir Figure 9.2) transitant entre les microprotocoles (appelés composants sur la Figure 9.2). Chaque composant (microprotocole) est construit comme une machine à état dont

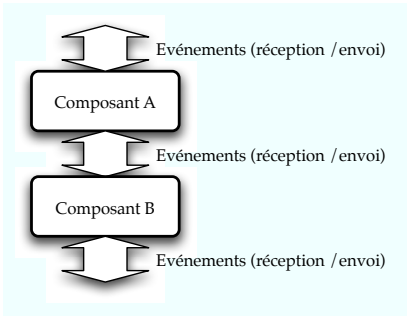

FIG. 9.2 – Communication de groupe : modèle de composition d'une pile de microprotocoles

les transitions sont déclenchées par les événements transitant. A titre d'exemple la liste suivante correspond à la pile de Ensemble/JGroups [Hayden, 2005, JGroups, 2006] que nous présentons du bas vers le haut.

- le multicast UDP IP (`UDP`),
- l'ordre FIFO fiable (`FIFO`),
- l'ordre total dans l'absence de fautes (`TOTAL`),
- la couche de représentation de l'application, accédée par l'application pour envoyer et recevoir les messages de l'application (`APPL`),
- le ramasse-miette des messages délivrés par les membres du groupe (`STABLE`),
- la détection de fautes (`FD`),
- le changement de la synchronisation des messages de l'application qui sont bloqués lors des changements dans la structure d'un groupe (`SYNC`),
- la gestion de la composition des groupes (`GMS`),
- le traitement de tous les messages en attente avant de modifier la composition du groupe (`FLUSH`),
- le transfert d'état (`STATE_TRANSFER`).

A noter que la couche `APPL` ne se situe pas en haut de la pile mais au milieu. Pour des raisons de performance lorsqu'un message applicatif est envoyé, le message traverse les trois microprotocoles du dessus de la pile (`TOTAL`, `FIFO` et `UDP`). A l'opposé, des messages moins courants, comme un message dit `SUSPECT` pour la détection de fautes, traversent la pile en entier. Dans le cas de la réplication active, un microprotocole est ajouté pour réaliser un vote sur les résultats des serveurs à un message. Ce microprotocole est appelé *coordination*. Nous obtenons donc la pile de la Figure 9.3

Dans notre exemple, pour alléger notre explication nous utilisons une pile simplifiée qui exclut les microprotocoles suivants : `STABLE`, `FD`, `SYNC`, `FLUSH` et `STATE_TRANSFER`. Notre scénario d'étude est donc le suivant :

1. Le microprotocole *coordination* intercepte l'appel,
2. Il récupère la liste des répliques serveurs depuis le microprotocole `GMS` via le microprotocole `APPL` et prépare le vote (ces étapes ne sont pas énumérées ici),
3. Le message entre dans la pile grâce au microprotocole `APPL`,
4. Le microprotocole `TOTAL` ajoute un numéro unique, partagé par toutes les répliques pour la requête et pour assurer l'ordonnancement,
5. Le microprotocole `FIFO` ajoute un nombre localement à la requête du client,
6. Le microprotocole `UDP` diffuse par *broadcast* la requête à toutes les répliques,

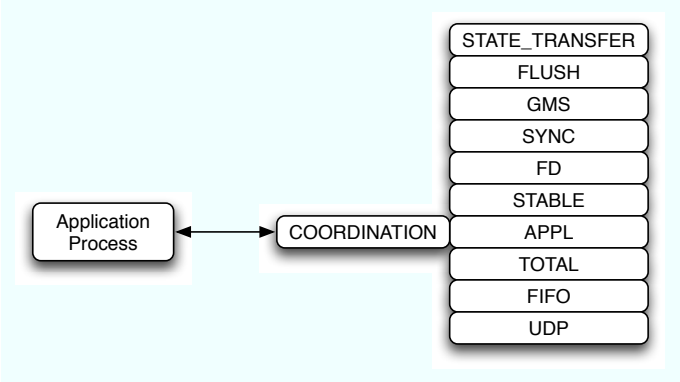

FIG. 9.3 – Cas de la réplication active : la pile de microprotocoles côté client

7. Les microprotocoles `UDP` côté serveur récupèrent les requêtes et renvoient un accusé de réception,

8–9. Les microprotocoles `FIFO` et `TOTAL` vérifient la séquence de la requête (en comparant les nombres) en ordre FIFO et s'assurent qu'un ordre total est maintenu,

10. La requête est exécutée sur les répliques,

11–19. Les réponses sont renvoyées à la pile applicative du client,

20. Le microprotocole *coordination* réalise un vote,

21. Pour finir, le client reçoit la réponse.

9.2 Problématiques

La mise en place de la gestion de la communication de groupe comme un aspect induit plusieurs problématiques que nous développons dans cette section. Tout d'abord nous justifions l'utilisation d'un aspect pour extraire et modulariser cette préoccupation, puis nous montrons comment la complexité de l'aspect engendre de l'entrelacement dans le code de l'aspect lui-même, dû à une mauvaise séparation des préoccupations.

9.2.1 Entrelacement de la communication de groupe avec le reste de l'application

Comme nous l'établissions dans la section précédente, il est difficile en communication de groupe d'obtenir une transparence complète vis-à-vis du client et des serveurs, qui ne doivent pas avoir conscience de prendre part à la réplication active. En particulier, la propriété d'atomicité de l'exécution impose de nombreux ancrages dans le code du client et des serveurs pour la gestion d'une annulation, ou d'erreurs partielles.

Pour rentrer un peu plus dans les détails, nous donnons ici un exemple tiré de la documentation de JGroups [JGroups, 2006] qui fournit une boîte à outils pour la communication de groupe en IP multicast en assurant à la fois la fiabilité et la gestion de groupe en Java. JGroups fonctionne sur le modèle de composition à base de piles que nous avons présenté à la section précédente. Ainsi, l'utilisation de JGroups impose l'instanciation et la déclaration de la pile au sein du code de base de l'application, la connexion à un canal, l'envoi et la réception de messages et la gestion

de la déconnexion, comme illustré sur le Listing 9.1. On observe que tous ces appels à l'interface de programmation de JGroups doivent être manuellement insérés à divers endroits du code de base. Clairement, l'emploi de l'approche par aspects permettrait d'extraire et de modulariser cette préoccupation de communication de groupe.

```
1  String props="UDP:PING:FD:STABLE:NAKACK:UNICAST:" +
2             "FRAG:FLUSH:GMS:VIEW_ENFORCER:STATE_TRANSFER:QUEUE";
3  Message send_msg;
4  Object  recv_msg;
5  Channel channel=new JChannel(props);
6  channel.connect("MyGroup");
7  send_msg=new Message(null, null, "Hello world");
8  channel.send(send_msg);
9  recv_msg=channel.receive(0);
10 System.out.println("Received " + recv_msg);
11 channel.disconnect();
12 channel.close();
```

Listing 9.1 – Exemple d'utilisation de JGroups

À notre connaissance, aucune étude existante n'a cherché à utiliser l'approche par aspects pour éviter cet entrelacement de code. Cependant des travaux de recherche ont fait le rapprochement entre gestion des transactions et gestion de groupes :

- [Shrivastava et al., 1987] démontre par exemple la dualité entre objets et actions (transactions) et processus et conversations (groupes),
- [Guerraoui and Schiper, 1995] élaborate un modèle générique très proche des modèles de transaction et du modèle de synchronisation virtuelle,
- [Schiper and Raynal, 1996] montre que les systèmes orientés transaction et ceux orientés groupes sont loin d'être éloignés par le fait qu'il est possible d'utiliser la communication de groupe pour supporter une catégorie de transaction.

En bref, même si le domaine de la communication de groupe n'est pas doté d'un formalisme tel que ACTA [Chrysanthis and Ramamritham, 1991] pour les transactions étendues, il est possible de rapprocher les événements en GCS avec le code de démarcation des transactions. Par exemple :

- Trouver le nom du groupe est similaire à un amorçage de transaction,
- Démarrer un fil d'exécution pour surveiller le statut d'une requête et pour compenser éventuellement son action est semblable aux procédures de retours arrière (*rollback*) des transactions.

Tous les événements sur lesquels les microprotocoles d'une pile pour GCS réagissent sont, comme le code de démarcation des transactions, éparpillés dans le code applicatif. Les quelques méthodes présentées dans le Listing 9.1 ne sont qu'une partie des méthodes utilisées pour la configuration, liaison, capture d'événements, gestion de connexion, création/destruction de sockets, création destruction de pile, etc. L'extraction de ce code sous forme d'un aspect peut alors faire gagner en clarté et en modularité. Partant donc du postulat que la gestion de la communication de groupe a à gagner à être modularisée par des aspects, nous nous intéressons à présent à la problématique de la conception de cet aspect, qui s'avère être un aspect particulièrement complexe.

9.2.2 Entrelacement dans l'aspect lui même

La conception de la gestion de la communication de groupe comme un aspect est un problème de conception complexe. En effet, comme nous l'avons vu en introduction de ce chapitre, la communication de groupe fait appel à de nombreux microprotocoles, organisés en couches et formant une pile. Les piles reposent sur un modèle de composition basé sur des événements transmis entre les microprotocoles. Cette propriété de pile et la notion d'événement à gérer par

chaque couche de la pile nous place devant deux décompositions différentes possibles de l'aspect. Chaque décomposition propose un ensemble de ce que nous appelons des sous-préoccupations. Les sous-préoccupations ont la propriété particulière de devoir être assemblées d'une certaine manière pour pouvoir fonctionner. Chaque sous-préoccupation n'a pas de sens seule, et doit faire partie du tout que constitue l'aspect complexe : la préoccupation de communication de groupe. Nous avions rencontré le même souci de décomposition en sous-préoccupation dans l'étude des transactions étendues au Chapitre 8.

Nous regardons à présent les décompositions possibles de la communication de groupe en sous-préoccupations. La première décomposition que nous étudions suit la séparation en micro-protocoles.

Les microprotocoles comme sous-préoccupations de l'aspect de communication de groupe

Une première approche de l'aspectisation de la communication de groupe est d'extraire la pile de protocole dans un aspect. D'un point de vue de la séparation des préoccupations, il est ainsi possible d'extraire proprement la fonctionnalité de communication de groupe de la logique métier de l'application.

Dans le cas de la communication de groupe, nous pouvons dire que par essence la pile de microprotocoles constitue bien une composition de préoccupations bien modularisées. Avec l'émergence de systèmes comme Horus [van Renesse et al., 1996], Ensemble [Hayden, 2005], ou encore Coyote [Bhatti et al., 1998], la décomposition de l'aspect de communication de groupe fournit une bonne séparation des préoccupations. Chaque microprotocole encapsule correctement une décision de conception. Cependant, si cela est vrai en théorie, en pratique, chacun de ces microprotocoles n'est pas complètement indépendant dans son implantation et doit faire un certain nombre de choix en fonction des protocoles prenant part à la pile. Comme semble le préciser la structure de la pile, chaque couche devrait reposer sur les couches inférieures. Par exemple, `FIFO` délivre des messages *multicast* via `UDP` ; `TOTAL` assume la sûreté d'une diffusion *broadcast* renforcée par `FIFO`. Cependant de manière moins évidente on voit apparaître des dépendances vers des couches supérieures. Par exemple `TOTAL` garantit un ordonnancement total seulement en l'absence de fautes, nécessitant de la part de `FD` une détermination du domaine de validité, par exemple quand `APPL` peut se reposer uniquement sur `TOTAL`. Par conséquent, le microprotocole `TOTAL` seul sans `FD` n'a pas de sens. Tous ces exemples nous montrent que par cette décomposition en sous-préoccupations, chacune de ces sous-préoccupations n'a de sens que combinée avec les autres pour former le tout de l'aspect.

Avant de montrer comment cette première décomposition conduit à de l'entrelacement, nous étudions une seconde décomposition possible de la préoccupation de communication de groupe.

Les messages comme sous-préoccupations de l'aspect de communication de groupe

Une autre décomposition possible de l'aspect de communication de groupe est de se baser sur les types des messages de l'application. Ces messages correspondent à la structure de données transitant sur le réseau. Un message est décomposé en deux parties distinctes : la partie utile ou de donnée (*payload*) et l'*en-tête*. La partie utile correspond aux données applicatives. Les microprotocoles y ajoutent des en-têtes au moment de la transmission du message, et en retire certains à la réception. Les en-têtes sont par exemple utilisées pour adjoindre un nombre aux parties utiles pour implanter une logique d'ordre, comme FIFO ou TOTAL. Ainsi, lorsqu'un message traverse la pile de l'émetteur, chaque microprotocole peut ajouter un en-tête au message. Généralement, un microprotocole ne peut avoir accès aux en-têtes d'autres microprotocoles. Les événements sont les structures de données utilisées pour la communication à l'intérieur de la pile. Ils ne

sont jamais transmis sur le réseau. Des exemples typiques d'événements sont : «un processus demande à quitter le groupe», « un processus est suspecté d'être défaillant», ou encore «délai expiré». Pour simplifier notre discours, nous parlons désormais indifféremment de messages pour les événements et les en-têtes.

Les messages peuvent être utilisés comme décomposition de l'aspect de communication de groupe. Si l'on suit cette idée, tout le code concernant un type de message donné, serait regroupé dans une même entité, disons un bloc. Dans chacun de ces blocs, on retrouverait les différents comportements des différents microprotocoles. Ces microprotocoles nous ont servi à la première décomposition. En pratique, comme les messages de l'application contiennent plusieurs en-têtes, ils sont consumés plusieurs fois, une fois par en-tête. Leur ordre détermine leur priorité de traitement.

Par comparaison des deux décompositions, nous observons que, à chaque fois, les sous-préoccupations d'une décomposition se retrouvent entrelacées dans la seconde, et vice-versa. Quel que soit la décomposition choisie, on retrouve de l'entrelacement. Les deux catégories de sous-préoccupations sont entremêlées. Nous généralisons à présent cette remarque.

Vers de l'entrelacement des sous-préoccupations

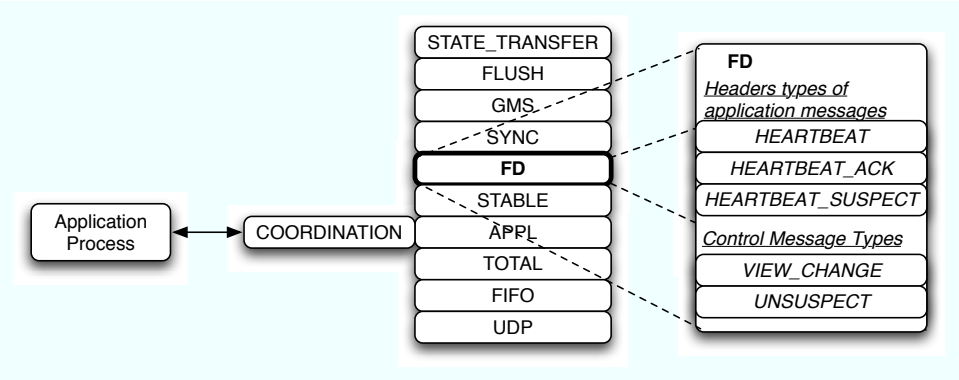

FIG. 9.4 – Décomposition de la préoccupation GCS en suivant les micro-protocoles de la pile.

Dans la première décomposition, chaque sous-préoccupation (microprotocole) encapsule de nombreux blocs de nature différente qui correspondent au traitement des différents types de message. Par exemple, on voit sur la Figure 9.4 que le microprotocole `FD` détecte les fautes grâce à un algorithme classique de *hearbeat*. Lorsque ce microprotocole reçoit un message de l'application, trois cas sont possibles, correspondant aux messages : `HEARTBEAT` pour émettre un *ping*, `HEARTBEAT_ACK` qui correspond à la réponse d'un *ping*, et `HEARTBEAT_SUSPECT` pour informer les autres microprotocoles de la suspicion d'une faute. `FD` doit aussi répondre aux messages de contrôle qui remontent la pile lors de changements concernant les membres (message du type `VIEW_CHANGE`) ou pour corriger les fausses suspicions (messages du type `UNSUSPECT`). A travers cet exemple, nous voyons comment l'entrelacement ce manifeste au sein d'un microprotocole et comment cet entrelacement correspond à l'entrelacement des sous-préoccupations de l'autre décomposition.

De manière similaire, la seconde décomposition entrelace les sous-préoccupations de la première. La Figure 9.5 en propose une vue conceptuelle. L'exemple du message *UNSUSPECT* est intéressant car il agit sur la détection de fautes (retirer les hôtes concernés de la liste), et sur la gestion de groupe (décider d'un nouveau changement de vue). Le code correspondant donc à *UNSUSPECT* entrelace donc au moins ces deux sous-préoccupations. On peut donc dire que la

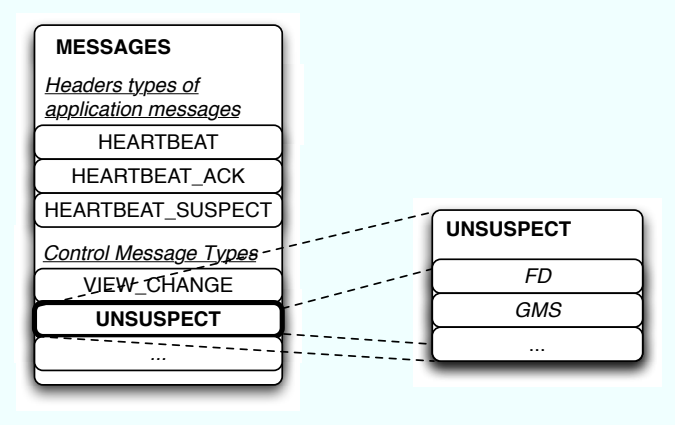

FIG. 9.5 – Décomposition de la préoccupation GCS en suivant les types de message.

seconde décomposition entrelace les sous-préoccupations de la première.

En résumé, nous avons obtenu deux décompositions orthogonales. Ce qui signifie que, quel que soit le choix effectué, les blocs résultants souffrent d'entrelacement de préoccupation.

Nous venons de voir à travers l'exemple de communication de groupe que l'entrelacement de code dans les aspects intervient lorsque les sous-préoccupations sont nombreuses au vue de la complexité de l'aspect et sont entrelacées. Nous regardons à présent les mécanismes offerts pour la composition d'aspects dans la littérature.

L'approche par aspects seule ne suffit pas

Nous ne re-développons pas ici l'incapacité de l'approche à composants seule à venir à bout de l'entrelacement de préoccupation (se référer à la Section 5.1 p.60). Nous justifions à présent l'incapacité de l'approche par aspects seule à traiter de la séparation des sous-préoccupations de l'aspect complexe qu'est la communication de groupe, que nous désignerons par GCS (*Group Communication System*) à partir de maintenant.

Nous nous sommes déjà heurté à ce problème d'entrelacement dans un aspect complexe au Chapitre 8 où nous discutions de la modularisation de la gestion des transactions étendues comme un aspect. Pour l'aspect GCS, nous sommes dans un cas très semblable, où deux décompositions sont possibles, toutes deux conduisant à de l'entrelacement. Nous l'avions vu également avec les transactions étendues, le mécanisme de composition présenté par l'approche par aspects est le séquencement d'advice. Pour rappel, le séquencement d'advice est un mécanisme d'ordonnancement des aspects lorsque ces derniers interviennent sur le même point de jonction.

Appliquée à la décomposition en pile de microprotocoles, le séquencement d'advice peut fournir un ordre qui correspond à ce qui est attendu au niveau de la pile (voir la Figure 9.6). Chaque microprotocole est encapsulé dans un aspect, et tous ces aspects sont composés dans l'ordre de la pile autour de la méthode de base — qui nécessite une communication de groupe —. Cette idée est suggérée dans l'étude [Hiltunen et al., 2006] qui fait un rapprochement entre le modèle de programmation Cactus [Bhatti et al., 1998, Hiltunen et al., 1999] et les langages par aspects. Il existe en effet une similitude importante entre une pile de microprotocoles et un ensemble d'aspects tissés autour d'un même point de jonction. Il est donc possible d'émuler le comportement d'une pile avec des aspects.

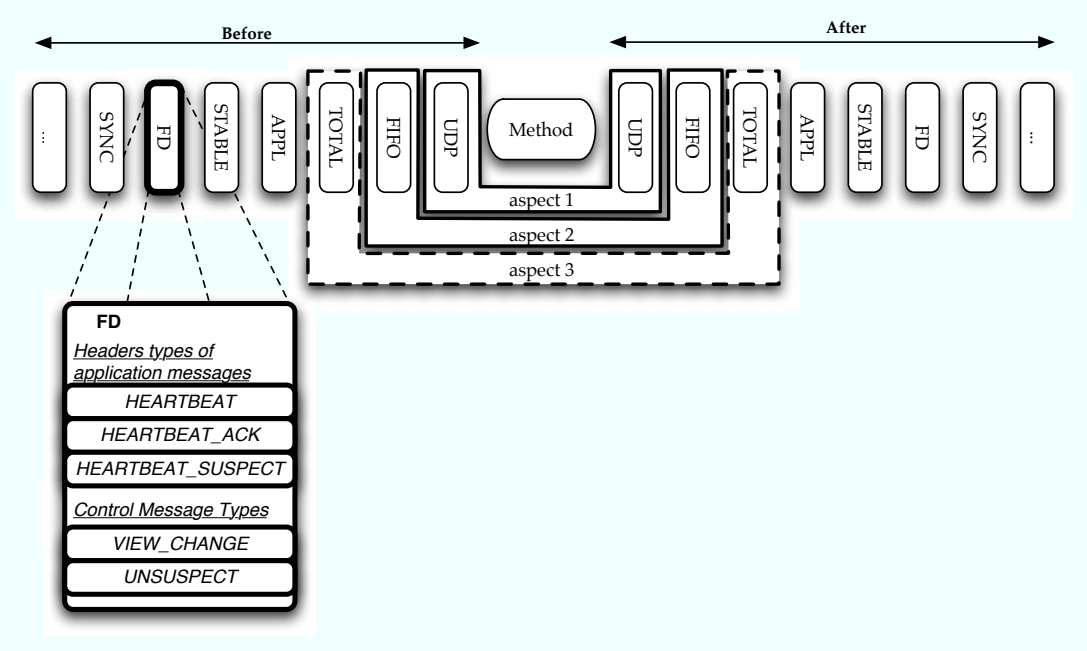

FIG. 9.6 – Composition sur l'exemple de la communication de groupe

Un premier problème important qui se pose dans cette approche est qu'une pile, comme nous l'avons vu précédemment, ne se parcourt pas toujours dans un seul sens de la même manière. Des messages peuvent indifféremment monter ou redescendre. Cette possibilité est d'ailleurs largement utilisée dans les optimisations de piles qui consistent à court-circuiter certains protocoles. Ce mécanisme ne peut être reproduit par séquencement. Au moins un mécanisme de communication inter-aspects est nécessaire. Le second problème majeur est que même avec un mécanisme de communication inter-aspects le problème d'entrelacement resterait présent comme illustré sur la Figure 9.6, le microprotocole *FD* entrelace différentes sous-préoccupations.

En conclusion, nous avons étudié la modularisation de la préoccupation GCS sous forme d'un aspect. Du fait de sa complexité cet aspect est décomposé en sous-préoccupations. Nous avons découvert deux décompositions possibles en deux familles de sous-préoccupations. Nous avons montré que, dans les deux cas, l'aspect entrelace au niveau de son code les sous-préoccupations officiant dans l'autre décomposition. Par conséquent, l'aspect souffre d'entrelacement et la composition fournie par l'approche par aspects seule ne semble pas suffire selon notre analyse. Nous regardons à présent comment FAC utilisant conjointement composants et aspects peut venir à bout de cet entrelacement pour offrir une séparation des préoccupations ainsi qu'une extraction de la préoccupation de gestion de groupe du code de l'application de base.

9.3 Support avec FAC

Nous montrons à présent que les problèmes de l'entrelacement dans l'aspect et d'ordonnancement complexe ne sont pas différents de l'entrelacement dans la base en appliquant la méthodologie FAC. Entre autres, nous voyons en quoi notre choix de modèle homogène et unifié se justifie pleinement lorsque l'on a à traiter d'aspects complexes.

Le domaine de la communication de groupe Nous nous situons ici au niveau de la conception de l'architecture de l'aspect lui-même, c'est-à-dire son ingénierie. Nous sommes donc dans la même situation que pour l'aspect ATMS du Chapitre 8. Nous rappelons que dans FAC tout est préoccupation. La séparation des préoccupations d'un aspect complexe comme celui de la communication de groupe ou celui des préoccupations transverses au sein d'une application comme Comanche, par exemple, suit le même processus. La première étape délimite donc les sous-préoccupations et leurs dépendances. Si des dépendances de nature transverse existent entre ces sous-préoccupations, il faut étudier si ces dépendances transverses sortent du cadre de la fonction première des composants concernés, auquel cas elles doivent être extraites à l'aide des mécanismes de FAC.

Comme nous l'avons fait pour les transactions avancées nous allons préciser le vocabulaire du domaine en rapport avec celui employé avec FAC. Nous avons vu qu'il existait des sous-préoccupations qui sont les microprotocoles des piles, et qu'il existait également des sous-préoccupations qui sont les messages échangé par ces microprotocoles. Avec FAC, encore une fois, toutes ces sous-préoccupations sont des préoccupations, donc représenté par des composants. Le fait que ces préoccupations soient transverse ou non, engendrera l'utilisation de liaisons classiques ou de liaisons d'aspects et des composants d'aspect.

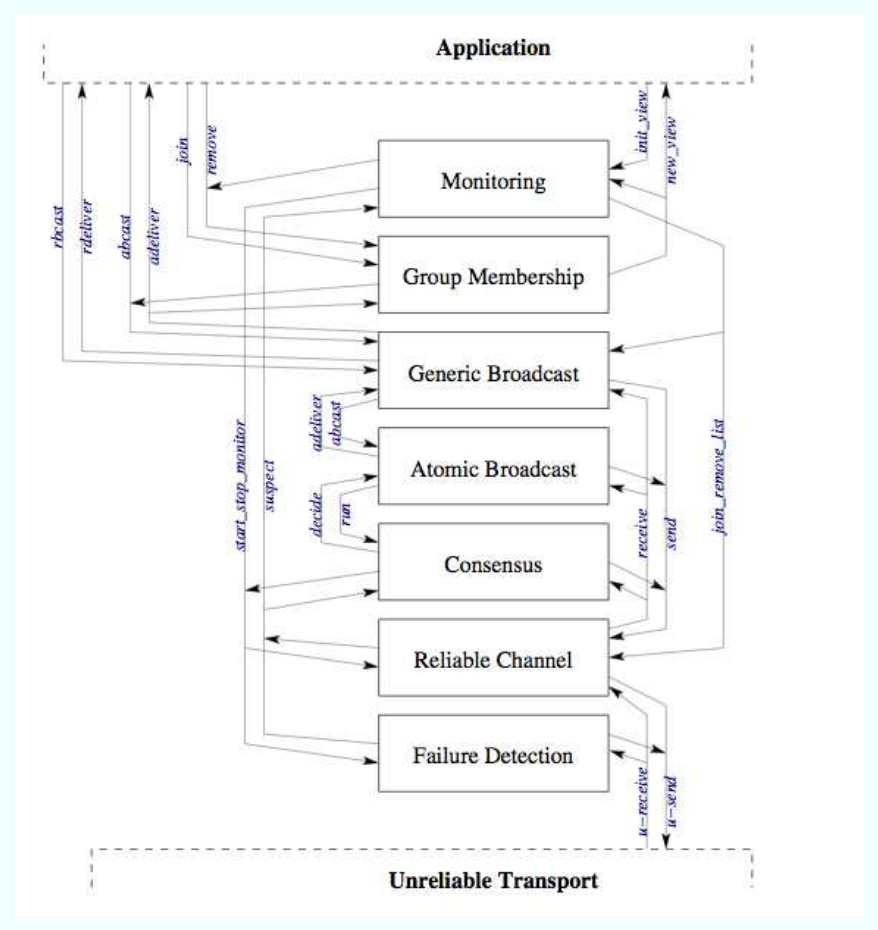

FIG. 9.7 – La pile Cactus et les communications entre microprotocoles

La décomposition en préoccupations Il est important de noter que le choix des sous-préoccupations doit correspondre à des choix de conception et non d'étapes d'exécution de l'aspect. Dans l'exemple de la communication de groupe, nous avons vu que les messages de l'application correspondaient plus à un choix d'étapes d'exécution et les microprotocoles de la pile à des choix de conception. Nous prenons donc la pile de microprotocoles comme décomposition. Chaque microprotocole peut être représenté comme un composant et toute dépendance avec d'autres microprotocoles est exposée par des interfaces fournies et requises puis des liaisons entre ces interfaces.

Nous obtenons ainsi une architecture de composants pour la représentation de la pile. Si l'on se réfère à la pile proposée par le modèle de composition Cactus [Hiltunen et al., 1999], de nombreuses communications existent entre les microprotocoles d'une pile, et par moment certains microprotocoles doivent directement communiquer ensemble, même s'ils ne sont pas directement voisins dans la pile. La Figure 9.7 est la pile proposée par Cactus avec la représentation de certaines relations entre les microprotocoles. Ce que nous proposons avec FAC est de matérialiser ces dépendances sous forme d'interfaces et de liaisons rendant l'architecture de la pile visible dans les fichiers d'architecture FRACTAL-ADL.

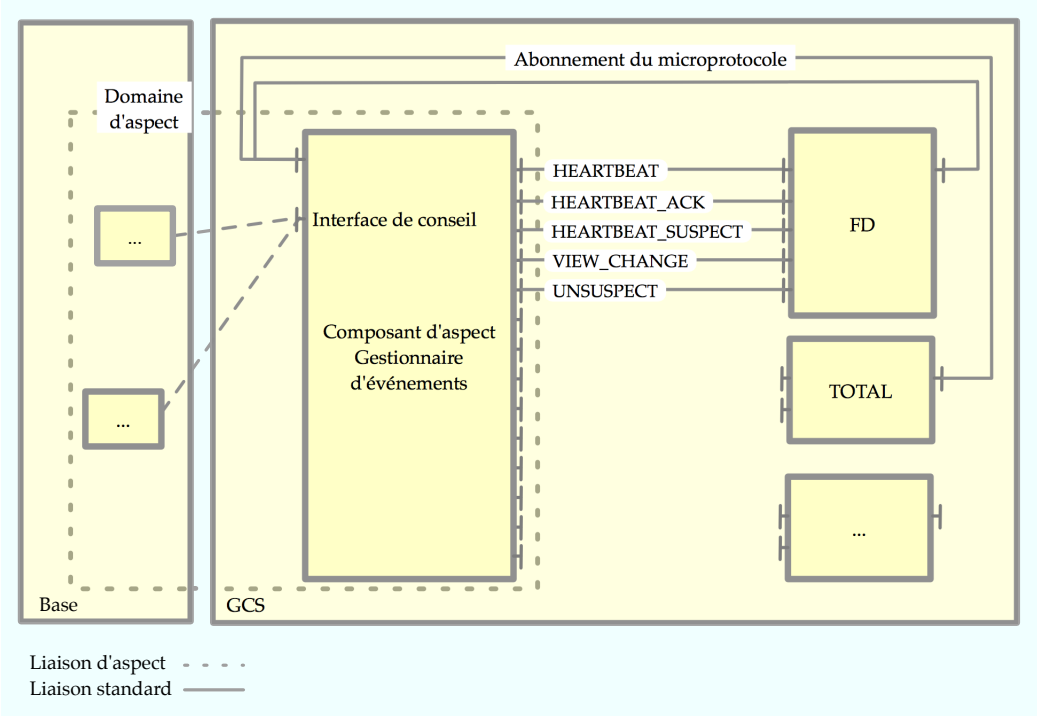

FIG. 9.8 – GCS : solution FAC

Identification des préoccupations transverses Cependant, comme nous l'avons vu précédemment, les communications entre microprotocoles ne sont pas les seules dépendances observées. Nous avons mis également en évidence une autre série de sous-préoccupations, qui sont les types des messages échangés entre les microprotocoles. Un même type de message peut donc apparaître dans de nombreux microprotocoles. Il semble donc nécessaire d'extraire tous les traitements de ces messages. D'autant plus qu'il existe deux types de messages : ceux venant de l'application et ceux uniquement utilisés entre les microprotocoles. Ces messages sont transverses aux microprotocoles de la pile, certes, mais font partie de la logique première de chaque microprotocole. Ils ne nécessitent pas l'emploi des mécanismes de FAC. Il est possible de regrouper

leur gestion dans un ou plusieurs composants qui auront un rôle de gestionnaire d'événements de l'application. Les événements servant à communiquer entre deux protocoles restent des liaisons entre ces composants.

Nous obtenons ainsi une architecture plane, faite de composants représentant les microprotocoles et les événements au même plan. La Figure 9.8 propose une schéma conceptuel de cette solution. Nous montrons comment un ou plusieurs composant(s) d'aspect peuvent être tissés à l'application qui doit supporter l'aspect GCS, et comment les événements perçus depuis l'application sont transmis aux microprotocoles, qui au préalable, se sont enregistrés auprès du composant d'aspect gestionnaire d'événements. Ensuite, les dépendances entre microprotocoles sont traitées directement entre les microprotocoles concernés.

Notons que dans cette solution les microprotocoles sont des composants (première décomposition) et les événements sont des interfaces (seconde décomposition). Tous les messages de l'application sont bien modularisés par le gestionnaire d'événements qui est le composant d'aspect. Tous les messages correspondant à des communications entre microprotocoles font également l'objet d'une interface distincte par type de message. Ainsi tous les messages sont clairement identifiables au niveau de l'architecture au même titre que les protocoles. Lors de la description de l'architecture du composite GCS, toutes les préoccupations apparaissent donc comme entités de premier ordre. On voit dans une telle étude de cas tout l'apport de la solution composant qui est une des deux facettes de l'unification fournie par FAC. La composition des composants vient, dans cette étude de cas, en aide au manque de pouvoir de composition des aspects. Les aspects viennent en aide aux composants lorsque ceux-ci ne peuvent extraire clairement une préoccupation transverse qui sort de la préoccupation première d'un composant.

Nous voyons donc dans ce cas de figure que FAC a servi à extraire l'aspect GCS de l'application d'où la présence de composants d'aspect gestionnaires d'événements, mais que de nouveaux aspects ne sont pas nécessaires pour l'ingénierie de l'aspect GCS lui-même. En effet, a contrario de la modularisation de l'aspect de gestion de transactions étendues du Chapitre 8, une séparation des préoccupations peut être obtenue grâce aux composants seuls.

9.4 Evaluation

Pour cette étude de cas, nous ne sommes pas allés aussi loin que lors de l'étude de cas sur les transactions étendues (Chapitre 8), c'est-à-dire jusqu'à l'implantation. Cependant nous pouvons évaluer dans cette section l'apport de notre solution abstraite en termes de séparation des préoccupations. Pour cela nous comparons la solution obtenue à la modularisation fournie par JGroups [JGroups, 2006] une implantation de référence en Java pour un support de communication multicast sûre. Pour réaliser cette comparaison nous regardons un certain nombre de critères sur les microprotocoles, les événements ou messages échangés et, enfin, nous évaluons la séparation des préoccupations obtenue.

Microprotocole Au niveau de la représentation d'un microprotocole, nous avons vu que FAC les représente comme des composants alors que JGroups utilise des classes. La gestion des microprotocoles se fait sous forme d'une pile avec JGroups et constitue un assemblage de composants avec FAC. Par conséquent, les optimisations réalisées traditionnellement sur les piles qui court-circuitent le fonctionnement normal de la pile sont simplement des appels entre composants avec FAC.

Message Un message correspond à un élément d'un bloc *switch* dans le code d'une classe représentant un microprotocole avec JGroups, alors qu'un message est caractérisé par une interface

Critère	JGroups	FAC
Microprotocole		
Représentation	Classe	Composant
Gestion	Pile	Assemblage de composants
Communication entre microprotocoles	Pull/push sur les messages passant par la pile	Lien entre deux interfaces
Message		
Représentation	Bloc *switch* dans le code des microprotocoles	Interface de composant
Gestion	Eparpillée dans chaque microprotocole	centralisée dans le composant d'aspect
Séparation des préoccupations		
Microprotocoles	+	+
Gestion messages	-	+
Base/aspect GCS	-	+

TAB. 9.1 – Comparaison de JGroups et FAC

de composant avec FAC. De ce fait, la gestion des messages de l'application est éparpillée dans les classes des microprotocoles avec JGroups, alors que cette gestion est centralisée dans un composant d'aspect avec FAC.

Séparation des préoccupations En conséquence des deux catégories précédentes, la séparation des préoccupations nous semble bonne au niveau des microprotocoles dans les deux approches. En revanche si l'on considère la gestion des messages de l'application FAC permet de les centraliser. Enfin la séparation entre la base (l'application nécessitant une réplication active, par exemple) et la préoccupation de GCS, FAC de par sa partie aspect, permet de clairement modulariser la préoccupation alors que du code JGroups doit être inséré à plusieurs endroits de la base.

9.5 Bilan

Nous avons présenté une étude de cas sur la gestion de la communication de groupe. Nous nous sommes intéressés tout comme pour l'étude sur les transactions étendues, à l'ingénierie de l'aspect lui même. Nous avons donc tout d'abord justifié l'intérêt de l'emploi de l'approche par aspects pour la modularisation de cette préoccupation. Nous nous sommes ensuite tourné vers l'ingénierie de cet aspect complexe.

De manière similaire à l'étude de cas sur les transactions étendues, nous avons observé que la gestion de la communication de groupe était en fait constituée de sous-préoccupations qui conduisent à de l'entrelacement dans le code de l'aspect du fait d'une mauvaise séparation des préoccupations. Nous avons à nouveau pointé le manque de composition fourni par les langages par aspects généralistes.

Nous avons enfin montré qu'avec FAC ce problème de composition était adressé sans même l'utilisation d'aspects à nouveau dans l'aspect complexe. Nous avons identifié des préoccupations certes de nature transverse, mais ne remplissant pas la condition de sortir du rôle premier des composants que nous avons identifié.

Dans cette étude de cas, nous avons mis en évidence l'intérêt de FAC à extraire la préoccupation de gestion de communication de groupe par un aspect FAC. L'entrelacement émergeant

dans l'ingénierie de cet aspect complexe peut être éliminé par le paradigme composant seul, tout en identifiant clairement toutes les sous-préoccupations de l'aspect, là où l'approche par aspects seule ne suffisait pas.

Chapitre 10

Conclusion

Ce chapitre dresse un bilan des contributions de nos travaux et propose les perspectives qui en découlent.

Résumé des contributions

L'objectif de cette thèse est d'étudier les rapprochements possibles entre le développement par composants et le développement par aspects. Dans la première partie de ce document, nous avons discuté des critères prépondérants de chacun des deux styles de développement. Puis, nous avons évalué, au regard de ces critères, les approches mêlant le concept de composant et d'aspect. Partant du constat qu'aucun modèle ne satisfaisait nos critères, du point de vue de l'approche à composants et de celui l'approche par aspects, nous nous sommes tourné vers le modèle à composants FRACTAL, qui répond à toutes nos attentes du point de vue du développement par composants.

La seconde partie de ce document développe notre proposition, une extension pour le modèle à composants FRACTAL pour le support des préoccupations transverses. Ce support constitue notre premier objectif majeur. Notre second objectif est de proposer un modèle unifié, c'est-à-dire ne distinguant pas la partie comportementale d'un aspect et d'un composant FRACTAL. Enfin, notre dernier objectif majeur est de proposer un modèle homogène, où toute propriété d'origine fonctionnelle ou technique est traitée à l'identique. Notre proposition appelée FAC pour Fractal Aspect Component étend donc FRACTAL avec des concepts issus de l'approche par aspects. Cependant, les concepts sont, autant que possible, unifiés avec le modèle FRACTAL, pour induire le minimum de changements ou d'extensions du modèle. Autrement dit, lorsque cela est possible les concepts sont représentés par des concepts issus de FRACTAL : un aspect est donc un composant, un code advice une interface, un tisseur d'aspects et une interface de contrôle d'un composant. De plus, nous avons ajouté de nouveaux concepts vis-à-vis de l'approche par aspects qui en améliore la visibilité : le domaine d'aspect qui permet de caractériser l'impact d'un aspect tissé sur un ensemble de composants, la liaison d'aspect qui permet également de visualiser les aspects agissant sur un composant.

Dans cette seconde partie, nous avons aussi mis en évidence la possible conciliation entre l'encapsulation forte des composants logiciels et de leurs contrats, et le côté envahissant des aspects qui modifient le comportement d'un système. Nous avons montré que les aspects pouvaient être

supportés de manière sûre et très contrôlée. L'interface de tissage, qui équipe chaque composant FAC pour le support d'aspects permet de contrôler finement (sur chaque point de jonction) les droits de tissage ou encore le séquencement d'aspects. Ces propriétés sont des contributions importantes pour l'approche par aspects.

Dans la troisième et dernière partie de ce document, nous avons proposé deux études de cas mettant en jeu la conception et l'implantation d'un service de transaction avancée d'une part, et d'un service de gestion de communication de groupe d'autre part, selon les principes du développement par aspects avec FAC. Dans ces deux études nous nous trouvons face à des aspects dits complexes, c'est-à-dire que chaque étude constitue une propriété transverse vis-à-vis d'un système car il s'agit de propriété techniques. Par ailleurs chacun de ces deux aspects est composé de nombreuses préoccupations qui sont parfois transverses les unes par rapport aux autres. Nous avons montré dans les deux études qu'il est possible d'obtenir une séparation de toutes les préoccupations.

En résumé, avec FAC, nous proposons un modèle :

- **général**, **hiérarchique**, **réflexif**, **extensible** et **doté par un langage d'architecture** : ces propriétés étant celles du modèle FRACTAL, étendu pour le support de l'approche par aspects,
- **supportant les préoccupations transverses** : FAC ajoute des concepts à FRACTAL pour ce support,
- **unifié** : un aspect est un composant FRACTAL,
- **homogène** : tout propriété technique ou fonctionnelle est conçue au même niveau, par des composants,
- **symétrique d'élément** : la symétrie d'élément rejoint le terme d'unification, la non distinction de la partie comportementale d'un aspect et d'un composant,
- **symétrique de placement** : la partie comportementale d'un aspect et sa liaison au reste d'un système sont strictement séparées,
- **symétrique de portée** : grâce à l'interface de tissage il est possible de contrôler entièrement l'ensemble des aspects agissant sur le même point de jonction.

Si la contribution du point de vue de l'approche à composants est le support des préoccupations transverses, FAC fournit à l'approche par aspects :

- une structuration plus forte et une meilleure encapsulation des aspects qui sont représentés par des composants,
- une visibilité accrue des aspects agissant dans un système grâce au concept de domaine d'aspect, de liaison d'aspect et d'introspection de coupe,
- une sûreté du support des aspects grâce à un contrôle total des aspects agissant sur chaque point de jonction d'un composant par son interface de tissage,
- une granularité de séquencement d'aspect très fine : un ordre différent est possible sur chaque point de jonction.
- une gestion avancée des aspects dits complexes, en donnant la possibilité d'extraire toutes les préoccupations transverses, y compris au sein d'un aspect, avec comme validation deux études de cas d'aspects complexes.

Publications

Revues internationales

[1] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A component-based and aspect-oriented model for software evolution. *IJCAT journal special issue on : Concern-Oriented Software Evolution*, 2007. To appear.

Conférences internationales

[2] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*, volume 4089 of *Lecture Notes in Computer Science*, pages 259—273, Vienna, Austria, mar 2006. Springer-Verlag.

[3] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien, and Thierry Coupaye. A component model engineered with components and aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'06)*, volume 4063 of *Lecture Notes in Computer Science*, Västeras, Sweden, jun 2006. Springer.

Ateliers internationaux

[4] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'aop. In *Première journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2004)*, Paris, France, sep 2004.

[5] Nicolas Pessemier, Lionel Seinturier, and Laurence Duchien. Components adl and aop : Towards a common approach. In *Workshop ECOOP Reflection ; AOP and Meta-Data for Software Evolution (RAM-SE 2004)*, Oslo, Norway, jun 2004.

[6] Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A safe aspect-oriented programming support for component-oriented programming. In *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming (WCOP'06)*, volume 2006–11 of *Technical Report*, Nantes, France, jul 2006. Karlsruhe University.

[7] Nicolas Pessemier, Olivier Barais, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A three level framework for adapting component based architectures. In *2nd Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT) at ECOOP'05*, jul 2005. 139

[8] Olivier Barais, Eric Cariou, Laurence Duchien, Nicolas Pessemier, and Lionel Seinturier. Transat : A framework for the specification of software architecture evolution. In *ECOOP First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04)*, Oslo, Norway, jun 2004.

[9] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe Merle. Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06)*, Nantes, France, jul 2006. 53

[10] Lionel Seinturier, Nicolas Pessemier, Clément Escoffier, and Didier Donsez. Towards a reference model for implementing the fractal specifications for java and the .net platform. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal'06)*, Nantes, France, jul 2006. 47

Revues nationales

[11] Olivier Barais, Alexis Muller, and Nicolas Pessemier. Extension de fractal pour le support des vues au sein d'une architecture logicielle. In *Numéro spécial de la revue L'OBJET-RSTI*, volume 11. Hermès Sciences, 2005. ISBN : 2–7462–1321–4.

Ateliers nationaux

[12] Olivier Barais, Alexis Muller, and Nicolas Pessemier. Extension de fractal pour le support des vues au sein d'une architecture logicielle. In *Objets Composants et Modèles dans l'ingénierie des SI (OCM-SI 2004)*, Biarritz, France, jun 2004.

[13] Romain Rouvoy, Nicolas Pessemier, Renaud Pawlak, and Philippe Merle. Apports de la programmation par attributs au modèle de composants fractal. In *Actes des 5èmes Journées Composants (JC'06)*, Perpignan, France, October 2006. 53

Perspectives

Les travaux réalisés ouvrent de nombreuses pistes de recherche. Nous en explorons quelques unes dans cette section.

Vers un catalogue de services intergiciels transverses

A travers les deux études de cas de nos travaux, nous avons pu voir comment FAC permet de modulariser des aspects complexes, qui sont des services intergiciels complexes comme la gestion des transactions avancées ou la gestion de la communication de groupe. Ces services doivent interagir fortement avec un système qui les utilise et demandent de nombreux efforts de configuration. Dans un tel contexte, la souplesse des composants logiciels, nous l'avons vu, apporte beaucoup à chacun de ces domaines d'application. Concernant leur connexion au reste du système, les aspects FAC permettent de facilement les faire interagir, tout en gardant une grande lisibilité. Nous pouvons donc nous projeter un peu plus loin et songer à une bibliothèque plus complète de services intergiciels pour des systèmes à base de composants. De plus, certains services techniques, déjà conçus par assemblage de composants sont directement utilisables avec FAC, qui peut alors contribuer sur les deux points suivants :

- amélioration de la séparation des préoccupations au sein même du service technique, chose que nous avons réalisé dans nos deux études de cas,
- suppression de la dépendance entre le reste de l'application et ses services, grâce à des liaisons d'aspect FAC.

A terme, une bibliothèque de services technique serait une contribution très intéressante qui validerait pleinement le passage à l'échelle de l'approche FAC.

Comment faire coexister de nombreux aspects avec FAC

Si l'on poursuit l'idée discutée dans la perspective précédente, c'est-à-dire se diriger vers une bibliothèque de services techniques conçus comme des aspects complexes avec FAC, un certain nombre de problèmes devront être pris en compte. En particulier, comment résoudre les conflits qui pourraient apparaître entre deux services techniques de nature assez similaire. Un des exemples les plus célèbre est incarné par les services de persistance et de transaction. Comment FAC permettra de composer ces services ensemble. Ces services du fait de leur proximité peuvent partager des *sous-préoccupations* ce qui reviendra avec FRACTAL à utiliser le partage de composant. La question se pose alors de savoir comment la configuration des deux services se ferra si chacun requiert une configuration différente. Le problème de composition d'aspect est aujourd'hui encore ouvert dans la communauté de l'approche par aspects. Il serait certainement intéressant d'aborder cette problématique à travers FAC, pour étudier l'apport notamment en termes de visibilité des coupes par le domaine d'aspect et les liaisons d'aspect. Est-ce que cette amélioration de visibilité peut permettre une collaboration plus aisée ?

Vers un modèle à trois niveaux d'aspect

Toute notre contribution autour de FAC propose d'unifier composants et aspects de telle sorte que les aspects sont traités à un niveau architectural. Il s'agit d'entité de premier plan, des composants, qui sont tissés par le biais d'un nouveau type de liaison : la liaison d'aspect. On peut donc qualifier l'approche d'architecturale. Dans notre état de l'art, nous avions étudié des approches utilisant composant et aspect, mais qui utilisaient les aspects au niveau des objets, c'est-à-dire appliqués au niveau de l'implantation des composants. Nous avions exclu cette approche car, d'une part, elle ignore le modèle sous-jacent (les composants) et, d'autre part, elle prend le risque de briser les contrats des composants en venant modifier le comportement de leur implantation.

Cependant, si l'on se replace dans un contexte large échelle, où les aspects peuvent être utilisés massivement et à plusieurs niveaux, il peut être intéressant d'avoir aussi la possibilité d'intervenir au niveau de l'implantation des composants. De la même façon, le niveau de contrôle des composants dans FRACTAL peut aussi gagner à utiliser les techniques aspects. Par conséquent, nous avons commencé à dessiner un modèle général capturant ces trois niveaux pour pouvoir bénéficier d'aspects sur plusieurs niveaux [Pessemier et al., 2005] :

- niveau architectural actuellement adressé par FAC,
- niveau technique/contrôle qui consiste à utiliser des aspects dans la membrane des composants dans le cas de FRACTAL ou dans les conteneurs pour une approche comme EJB,
- niveau implantation (la quasi totalité des approches par aspects aujourd'hui se positionnent à ce niveau).

Au delà de l'identification de ces trois niveaux, il semble encore plus profitable de pouvoir avoir un modèle les englobant toute trois et permettant même de les faire communiquer, d'obtenir une cohérence forte, voir même d'avoir une répercussion des tissages effectués à un des trois niveaux, aux deux autres.

Bibliographie

[Aksit et al., 1992] Aksit, M., Bergmans, L., and Vural, S. (1992). An object-oriented language-database integration model : The composition-filters approach. In Lehrmann Madsen, O., editor, *ECOOP'92 : Proc. of the European Conference on Object-Oriented Programming*, pages 372–395. Springer, Berlin, Heidelberg. 16, 17

[Aldrich, 2005] Aldrich, J. (2005). Open modules : Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586, pages 144–168. Springer. 86

[Aldrich et al., 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava : connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197, New York. ACM Press. 20

[Allan et al., 2005] Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2005). Adding trace matching with free variables to AspectJ. In *OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 345–364, New York, NY, USA. ACM Press. 84

[Allen, 1997] Allen, R. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144. 20

[Bhatti et al., 1998] Bhatti, N., Hiltunen, M., Schlichting, R., and Chiu, W. (1998). Coyote : A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4) :321–366. 126, 128

[Bodoff et al., 2004] Bodoff, S., Armstrong, E., Ball, J., and Carson, D. (2004). *The J2EE Tutorial*. AW. 2nd edition.
`java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html`. 12, 20, 61

[Bruneton et al., 2006] Bruneton, É., Coupaye, T., Leclercq, M., Quéma, V., and Stéfani, J.-B. (2006). The fractal component model and its support in java. In *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*. 24, 43

[Bruneton et al., 2002] Bruneton, E., Coupaye, T., and Stefani, J. (2002). Recursive and dynamic software composition with sharing. In *Workshop on Component-Oriented Programming (WCOP) at ECOOP'02*. `http ://fractal.objectweb.org/current/fractalWCOP02.pdf`. 43

[Burke, 2003] Burke, B. (2003). It's the aspects. Java's Developer's Journal.
`www.sys-con.com/story/?storyid=38104&DE=1`. 29, 47, 61

[Chockler et al., 2001] Chockler, G., Keidar, I., and Vitenberg, R. (2001). Group Communication Specifications : A Comprehensive Study. *ACM Computing Surveys*, 33(4) :427–469. 121

[Chrysanthis and Ramamritham, 1991] Chrysanthis, P. K. and Ramamritham, K. (1991). A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112. 102, 125

[Coulson et al., 2004a] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004a). A component model for building systems software. 23

[Coulson et al., 2004b] Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004b). Opencom v2 : A component model for building systems software. In *IASTED Software Engineering and Applications (SEA'04)*, Cambridge, MA, ESA. 24

[David, 2005] David, P.-C. (2005). *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation.* PhD thesis, École des Mines de Nantes and Université de Nantes, Nantes, France. 51, 52, 77

[Douence et al., 2003] Douence, R., Fradet, P., and Südholt, M. (2003). Trace-based aspects. In et al., M. A., editor, *Aspect-Oriented Software Development*. Addison-Wesley. 84

[Fabry, 2005] Fabry, J. (2005). *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde (PROG). 101, 103, 104

[Fabry and D'Hondt, 2006] Fabry, J. and D'Hondt, T. (2006). KALA : Kernel aspect language for advanced transactions. In *Proceedings of the 2006 ACM Symposium on Applied Computing Conference*, pages 1615–1620. ACM Press. 101, 102

[Filman and Friedman, 2000] Filman, R. and Friedman, D. (2000). Aspect-oriented programming is quantification and obliviousness. 67

[Fleury and Reverbel, 2003] Fleury, M. and Reverbel, F. (2003). The JBoss extensible server. In *Proceedings of the 4th ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'03)*, volume 2672 of *Lecture Notes in Computer Science*, pages 344–373. Springer-Verlag. 47, 61

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J., and Booch, G. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Westley Professional Computing, USA. 14, 30

[Garcia-Molina and Salem, 1987] Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259. 100

[Guerraoui and Schiper, 1995] Guerraoui, R. and Schiper, A. (1995). Transaction model vs Virtual Synchrony model : bridging the gap. In *Proc. International Workshop on Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science*, pages 121–132. Springer. 125

[Harrison et al., 2003] Harrison, W. H., Osscher, H. L., and Tarr, P. L. (2003). Asymmertically vs. symmetrically organised paradigms for software composition. Technical report, IBM Research Division, Thomas J. Watson Research Center. 16

[Hayden, 2005] Hayden, M. (2005). *The Ensemble System*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New-York (USA). 123, 126

[Hiltunen et al., 1999] Hiltunen, M., Schlichting, R., Han, X., Cardozo, M., and Das, R. (1999). Real-Time Dependable Channels : Customizing QoS Attributes for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :600–612. 128, 131

[Hiltunen et al., 2006] Hiltunen, M., Taïani, F., and Schlichting, R. (2006). Reflections on Aspects and Configurable Protocols. In *Proc. 4th of International Conference on Aspect-Oriented Software Development*, pages 87–98, Bonn (Germany). 128

[JGroups, 2006] JGroups (2006). JGroups Home Page. `http://www.jgroups.org`. 123, 124, 132

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York. 2, 12

[Kienzle and Guerraoui, 2002] Kienzle, J. and Guerraoui, R. (2002). Aop : Does it make sense ? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag. 101, 104

[Lynch and Tuttle, 1989] Lynch, N. A. and Tuttle, M. R. (1989). An introduction to input/output automata. CWI Quarterly, 2(3) :219–246. 85

[M. Pinto, 2003] M. Pinto, L. Fuentes, J. T. (2003). Daop-adl : An architecture description language for dynamic component and aspect-based development. *Generative Programming and Component Engineering (GPCE)*. 33

[M. Pinto, 2005] M. Pinto, L. Fuentes, J. T. (2005). A component and aspect dynamic platform. *The Computer Journal*. 33

[Magee et al., 1995] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989. 20

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1) :70–93. 2, 19

[Merle, 2006] Merle, P. (2003-2006). OpenCCM website. 50

[Merle and Moroy, 2006] Merle, P. and Moroy, J. (2006). The explorer framework. 50

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, New Jersey, 2 edition. 1, 12

[Moss, 1981] Moss, E. B. (1981). *Nested transactions : an approach to reliable distributed computing*. PhD thesis, Massachusetts institute of Technology. 100

[OMG, 2002] OMG (june 2002). CORBA Components, v3.0 (full specification), Document formal/02-06-65. 33, 61

[Ongkingco et al., 2006] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., and Sittampalam, G. (2006). Adding Open Modules to Aspectj. In *Proceedings of the 5nd International Conference on Aspect-Oriented Software Development (AOSD'03)*. ACM Press. 87

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058. 12, 103

[Pawlak et al., 2004] Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., and Martelli, L. (2004). JAC : An aspect-based distributed dynamic framework. *Software Practise and Experience (SPE)*, 34(12) :1119–1148. 17

[Pessemier et al., 2005] Pessemier, N., Barais, O., Seinturier, L., Coupaye, T., and Duchien, L. (2005). A three level framework for adapting component based architectures. In *2nd Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT) at ECOOP ?05*. 139

[Prochazka et al., 2003] Prochazka, M., Rouvoy, R., and Coupaye, T. (2003). On enhancing component-based middleware with transactions. In Springer-Verlag, editor, *Proceedings of On The Move to Meaningful Internet Systems 2003 : OTM 2003 Workshops*, pages 1–2, Catania, Sicile. Lecture Notes in Computer Sciences. ISBN : 3–540–20494–6. 62

[R. Johnson, 2006] R. Johnson, J. Hoeller, e. a. (2006). Spring - java/j2ee application framework. `http://static.springframework.org/spring/docs/2.0.x/spring-reference.pdf`. 22, 31

[Rapide, 1997] Rapide (1997). *Guide to the Rapide 1.0 Language Reference Manuals*. Rapide Design Team Program Analysis and Verification Group Computer Systems Lab Stanford University. 20

[Rashid and Chitchyan, 2003] Rashid, A. and Chitchyan, R. (2003). Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM. 101, 104

[Rhee et al., 1997] Rhee, I., Cheung, S., Hutto, P., Krantz, A., and Sunderam, V. (1997). Group Communication Support for Distributed Collaboration Systems. In *Proc. 17th of IEEE International Conference on Distributed Computing Systems*, Baltimore, USA. 122

[Rouvoy, 2006] Rouvoy, R. (2006). *Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France. 53

[Rouvoy et al., 2006a] Rouvoy, R., Pessemier, N., Pawlak, R., and Merle, P. (2006a). Apports de la programmation par attributs au modèle de composants fractal. In *Actes des 5èmes Journées Composants (JC ?06)*, Perpignan, France. To appear. 53

[Rouvoy et al., 2006b] Rouvoy, R., Pessemier, N., Pawlak, R., and Merle, P. (2006b). Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal ?06)*, Nantes, France. 53

[Schiper and Raynal, 1996] Schiper, A. and Raynal, M. (1996). From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4) :84–87. 125

[Schneider, 1990] Schneider, F. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach : A Tutorial. *ACM Computing Surveys*, 22(4) :299–319. 122

[Seinturier et al., 2006] Seinturier, L., Duchien, N. P. L., and Coupaye, T. (2006). A component model engineered with components and aspects. In *In Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, Stockholm, Sweden. 47

[Shrivastava et al., 1987] Shrivastava, S., Mancini, L., and Randell, B. (1987). On the Duality of Fault Tolerant System Structures. In Nehmer, J., editor, *Experiences with Distributed Systems*, volume 309, pages 19–37. Springer-Verlag, Kaiserslautern, Germany. 125

[Smith, 1982] Smith, B. C. (1982). *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology. 15

[Soares et al., 2002] Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM. 101, 104

[Suvée et al., 2006] Suvée, D., Fraine, B. D., and Vanderperren, W. (2006). A symmetric and unified approach towards combining aspect-oriented and component-based software development. In *Component-Based Software Engineering, 9th International Symposium, CBSE 2006, Västerås, Sweden, June 29 - July 1, 2006, Proceedings*, pages 114–122. 36

[Suvée et al., 2003] Suvée, D., Vanderperren, W., and Jonckers, V. (2003). JAsCo : an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press. 107

[Szyperski, 2002] Szyperski, C. (2002). *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc. 2, 12, 19, 33

[Tanter and Noyé, 2005] Tanter, É. and Noyé, J. (2005). A versatile kernel for multi-language AOP. In Gluck, R. and Lowry, M., editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia. Springer-Verlag. 107

[Tarr et al., 1999] Tarr, P. L., Ossher, H., Harrison, W. H., and Jr., S. S. (1999). N degrees of separation : Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119. 3, 18, 106

[Team, 2006] Team, A. (2006). The AspectJ project. http ://eclipse.org/aspectj/. 12, 14, 17, 31, 104, 107

[Vadet, 2004] Vadet, M. (2004). *Un Modèle de Services Logiciels pour la Spécialisation des Intergiciels à Composants*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Thalès, Lille, France. 47

[van Renesse et al., 1996] van Renesse, R., Birman, K., and Maffeis, S. (1996). Horus : A Flexible Group Coomunication System. *Communications of the ACM*, 39(4) :76–83. 126

[Vanderperren et al., 2005] Vanderperren, W., Suvee, D., Cibran, M., and Fraine, B. D. (2005). Stateful aspects in jasco. In *SC 05 : Software Composition*, Edinburgh, Scotland. LNCS. 84

[W3C, 1999] W3C (1999). World Wide Web Consortium, XPath project. 51

Annexe A

Interface de programmation de l'interface de tissage

```
1 interface WeavingInterface {
2  // Partie gestion des liaisons d'aspect
3  /** Etablit une liaison d'aspect
4   * @param regExp une expression régulière sélectionnant les opérations à intercepter
5   * @param aspect l'aspect auquel le composant se lie
6  */
7  void setAspectbinding(ItfPointcutExp regExp, Component aspect);
8  /** Rompt une liaison d'aspect vers un aspect
9   * @param aspect l'aspect concerné
10 */
11 void unsetAspectbinding(Component aspect);
12
13 // Partie ordonnancement
14 /** Change l'ordre d'un aspect.
15  * @param acName le nom de l'aspect
16  * @param oldPosition l'ancienne position
17  * @param newPosition la nouvelle position
18 */
19 void changeACorder(String methodName, int oldPosition, int newPosition);
20 /** Donne la position d'un aspect sur une opération
21  * @param ac le nom de l'aspect
22  * @param methodName le nom de l'opération concernée
23 */
24 int[] getACPosition(AspectComponent ac, String methodName);
25
26 // Partie verrouillage
27 /** Ouvre l'accès à une opération
28  * @param op l'opération concernée
29  * @param itf l'interface concernée
30 */
31 void openAccess(Operation op, Interface itf);
32 /** Verrouille l'accès à une opération
33  * @param op l'opération concernée
34  * @param itf l'interface concernée
35 */
36 void reduceAccess(Operation op, Interface itf);
37 /** Ouvre l'accès à une opération avec une politique particulière
38  * @param op l'opération concernée
39  * @param itf l'interface concernée
40  * @param lvl le niveau d'ouverture de l'opération
41 */
42 void openAccessTo(Operation op, Interface itf, OpennessLevel lvl);
43
44 // Partie gestion tissage
45 /** Tisse un aspect sur un ensemble de composants
46  * @param root le composant racine de la navigation
47  * @param regExp la coupe sélectionnant les opérations à intercepter
48  * @param aspect l'aspect à tisser
49  * @param aDomain le nom du domaine d'aspect à créer
```

```
50  */
51  void weave(Component root, Pointcut pcut, Component aspect, String aDomain);
52  /** Opération inverse du tissage
53   * @param aspect l'aspect à dé-tisser
54   * @param
55   */
56  void unweave(Component aspect);
57
58  // Partie introspection de coupe
59  /** Donne la liste des aspects tissés sur le composant
60   * @return un tableau contenant les références vers ces aspects (composants)
61   */
62  Component[] listAC();
63  /** Donne la liste des composants possédant des opérations liées à l'aspect passé en paramètre.
64   * L'architecture est introspectée à partir du composant racine (root).
65   * @param root le composant racine à partir duquel initier la recherche
66   * @param ac l'aspect recherché
67   * @return un tableau de références vers les composants concernés
68   */
69  Component[] listCrosscutComps(Component root, Component ac);
70  /** Donne une représentation sous forme d'arbre des opérations, interfaces et composant
71  * potentiellement concernés par la coupe passée en paramètre. Comme l'opération précédente,
72  * la recherche récursive démarre d'un composant racine.
73  * @param root le composant racine
74  * @param pcut la coupe
75  * @return une représentation objet des opérations, interfaces et composants concernés
76  */
77  PointcutRepresentation aspectizableComps(Component rComp, ItfPointcutExp pcut);
78 }
```

Index